**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.

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 прирщение 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 $\geq 0$).

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 предоставлены схема алгоритма Копперсмита-Винограда и его схемы распараллеливания.

## 2.1 Схема алгоритма Копперсмита-Винограда

Схема алгоритма Копперсмита-Винограда предоставлена на рисунках 2.1-2.3.

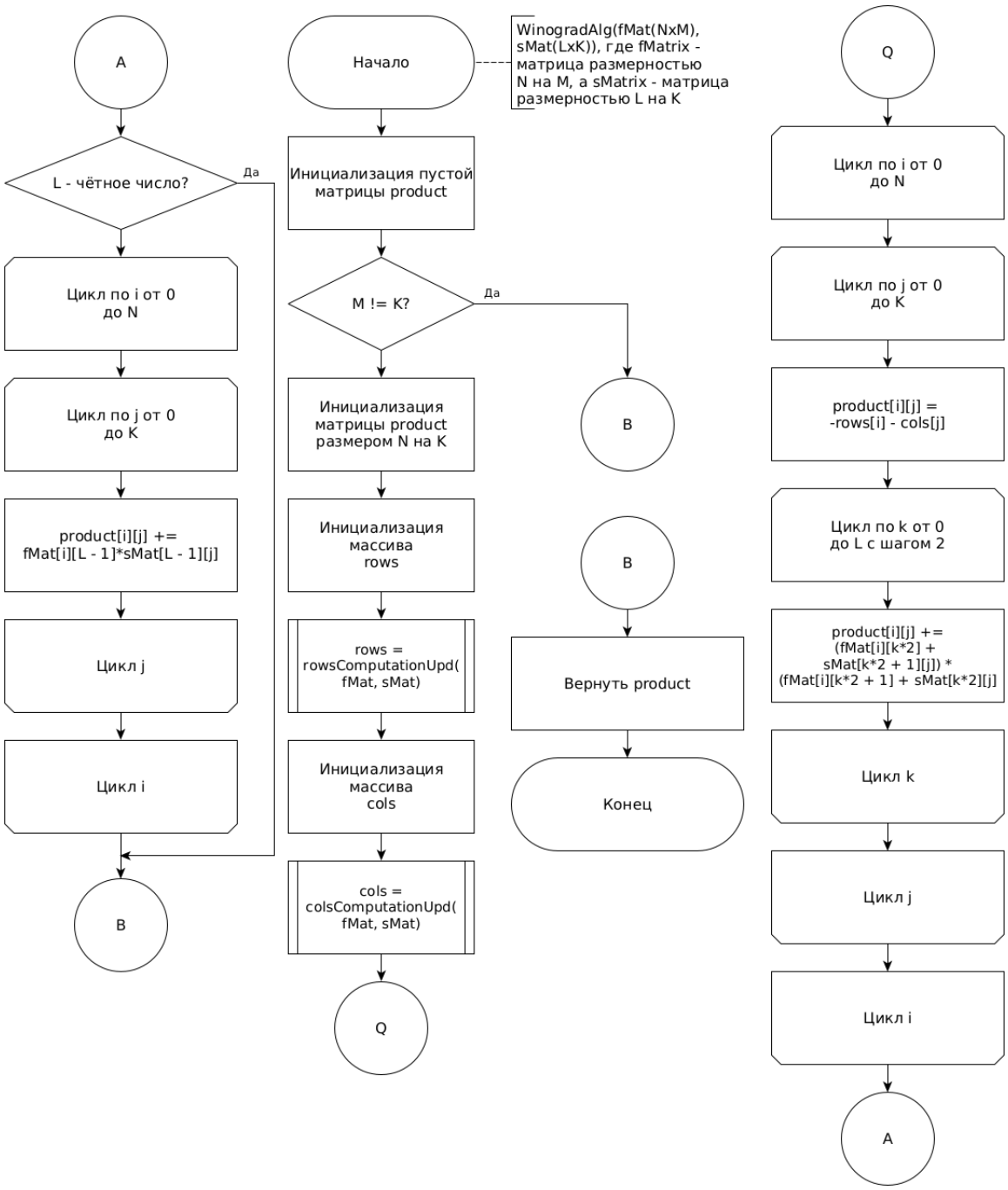


Рис. 2.1: Схема алгоритма Копперсмита-Винограда.

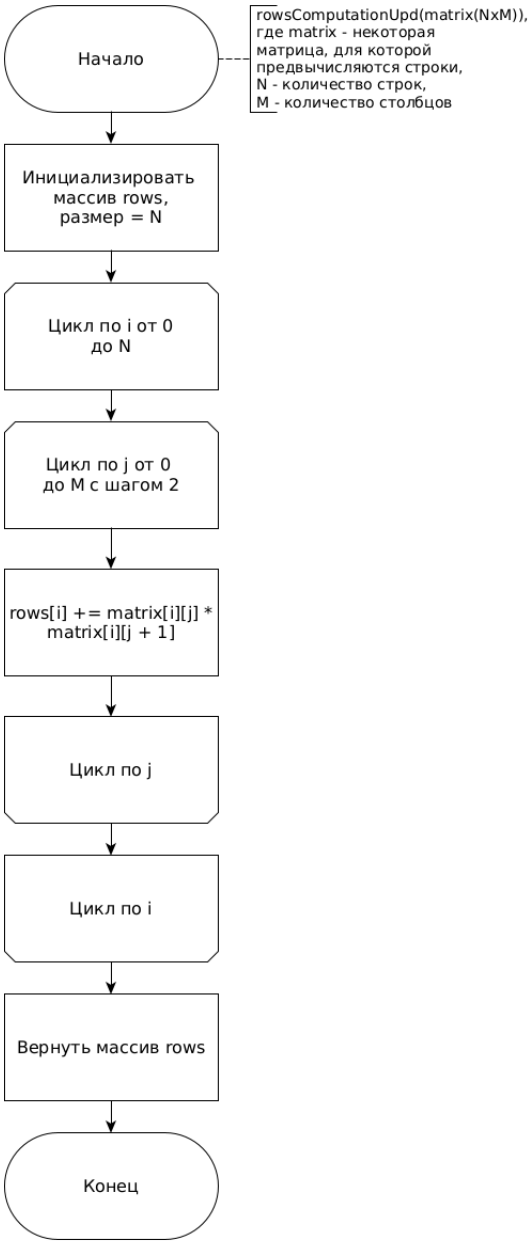


Рис. 2.2: Схема предрасчёта строк для алгоритма Копперсмита-Винограда.

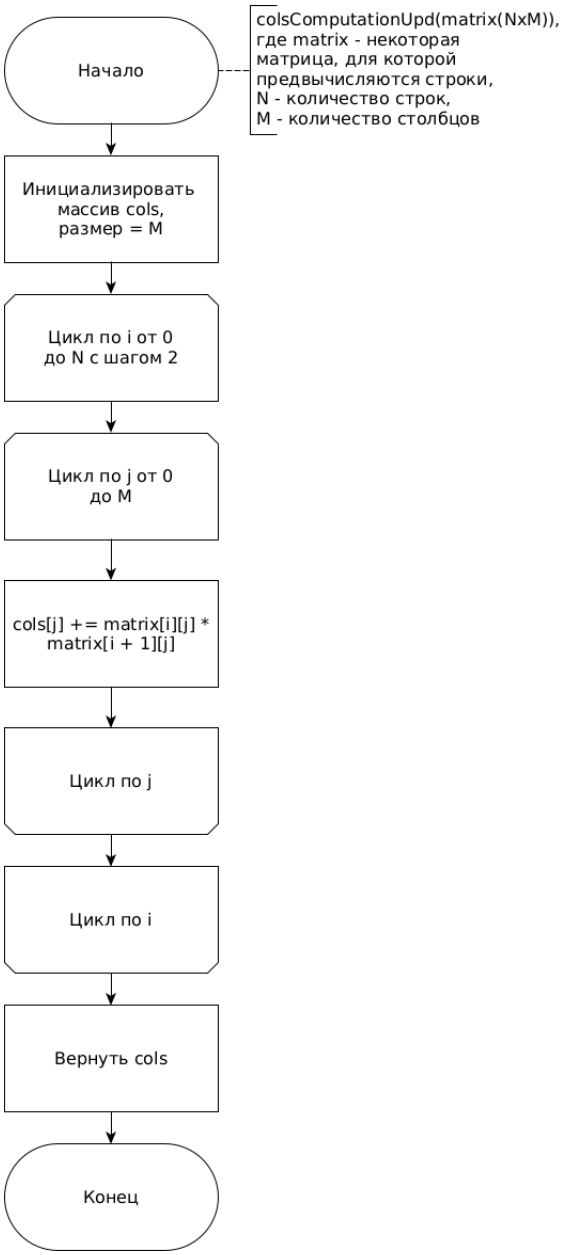


Рис. 2.3: Схема предрасчёта столбцов для алгоритма Копперсмита-Винограда.

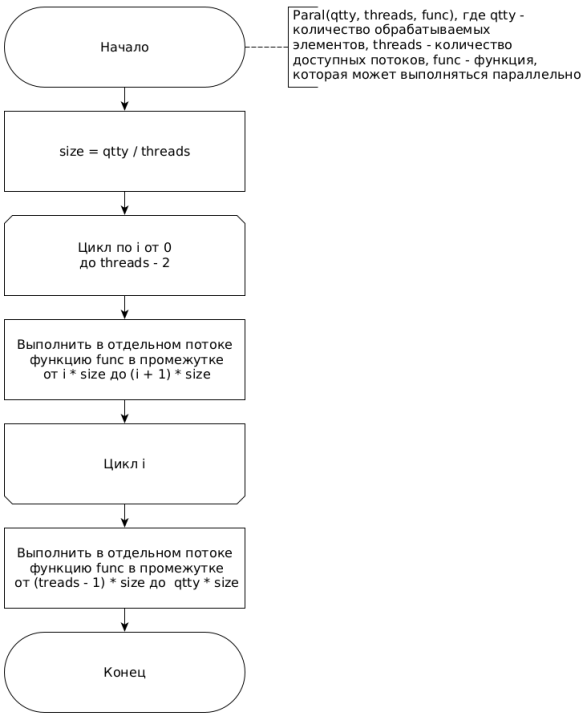


Рис. 2.4: Схема алгоритма работы функции, запускающей в требуемом количестве потоков функцию-аргумент.

## 2.2 Схема алгоритма работы функции, запускающая в требуемом количестве потоков функцию-аргумент

На рисунке 2.4 предоставлена схема алгоритма работы функции, запускающей в требуемом количестве потоков функцию-аргумент.

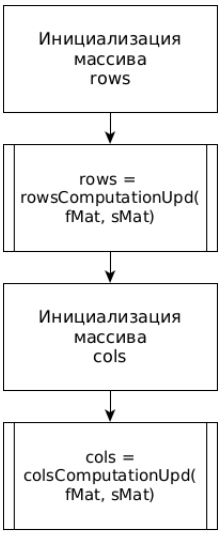


Рис. 2.5: Часть блок-схемы, в которой происходит вызов подпрограмм предрасчёта строк и столбцов.

## 2.3 Параллельная реализация алгоритма Копперсмита-Винограда

Для распараллеливания алгоритма требуется определить, какие из частей алгоритма могут быть выполнены вне зависимости друг от друга.

Часть блок-схемы, отвечающая за предрасчёт строк и столбцов (рисунок 2.5), однозначно может быть распараллелен. Но до завершения двух этих подпрограмм к выполнению следующих этапов приступать нельзя, так как вычисленные массивы будут использоваться далее. В данной части будет использоваться полное выделенное количество потоков последовательно - первоначально для вычисления строк, а далее - для вычисления столбцов.

Непосредственное вычисление значения каждого элемента матрицы, показанное на рисунке 2.6, может быть также поэлементно распараллелено. Однако тут важно заметить, что часть данного этапа, показанная на рисунке 2.7, и часть, показанная на рисунке 2.8, могут быть выполнены как раздельно, так и совместно при обработке каждого из элементов. Важно заметить, что, при раздельном выполнении, потребуется создавать новые потоки, что является достаточно затратным по времени действием.

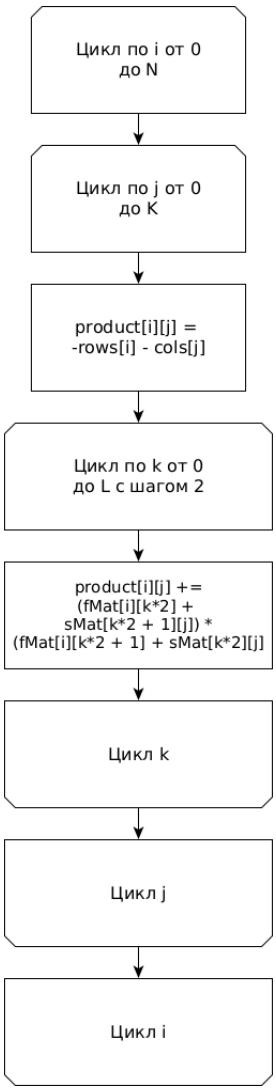


Рис. 2.6: Часть блок-схемы, в которой происходит расчёт значения каждого элемента матрицы.

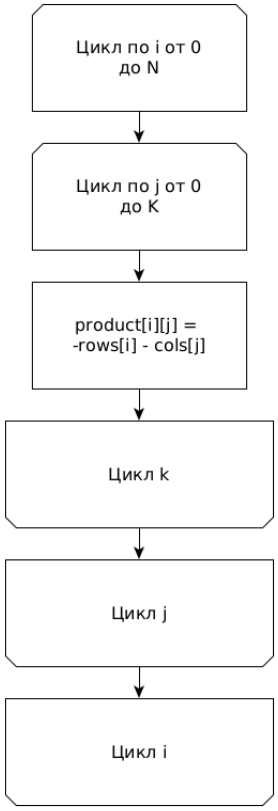


Рис. 2.7: Первая часть рассматриваемого этапа.

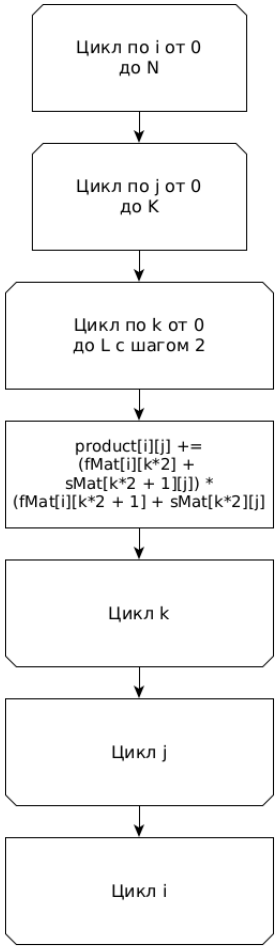


Рис. 2.8: Вторая часть рассматриваемого этапа.

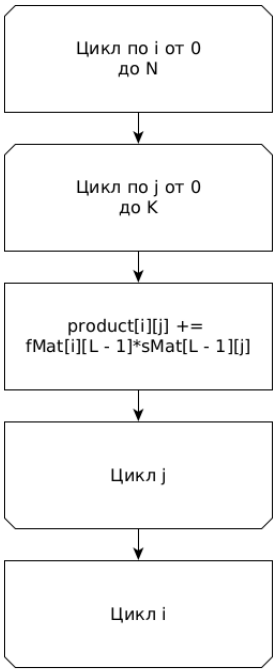


Рис. 2.9: Довычисление значений итоговой матрицы при нечётной размерности оной.

Довычисление значений в случае нечётной размерности итоговой матрицы, показанное на рисунке 2.9 в каждой итерации цикла требует обращения к матрице, что при распараллеливании приведёт к большому числу блокирований разделяемой памяти и будет неэффективно. Поэтому данный этап останется без изменений.

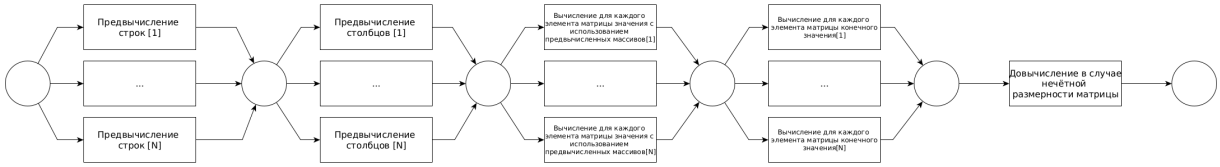


Рис. 2.10: Схема с одновременным выполнением двух частей этапа вычисления элементов матрицы.

## 2.4 Схема параллельной реализации алгоритма Копперсмита-Винограда с разделением этапа вычисления элементов матрицы

На рисунке 2.10 предоставлена схема с одновременным выполнением двух частей этапа вычисления элементов матрицы.

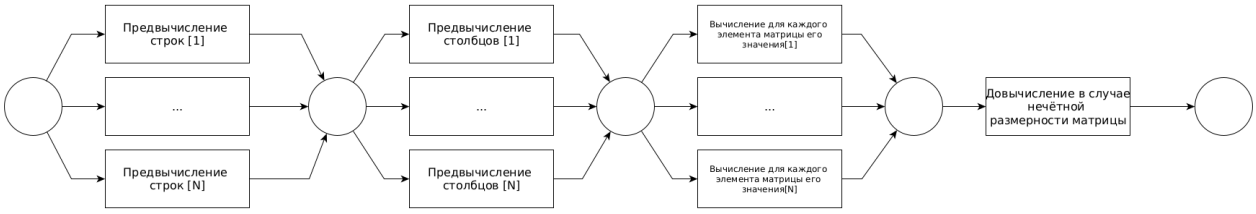


Рис. 2.11: Схема с последовательным выполнением двух частей этапа вычисления элементов матрицы.

## 2.5 Схема параллельной реализации алгоритма Копперсмита-Винограда без разделения этапа вычисления элементов матрицы

На рисунке 2.11 предоставлена схема с последовательным выполнением двух частей этапа вычисления элементов матрицы.

# Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема алгоритма Копперсмита-Винограда. Были предложены 2 схемы параллельной реализации рассматриваемого алгоритма.

# 3 | Технологическая часть

## 3.1 Требования к программному обеспечению

- входные данные - две матрицы размерностью $MxN$ и $KxL$;
- выходные данные - результат умножения двух переданных матриц.

## 3.2 Средства реализации программного обеспечения

При написании программного продукта был использован язык программирования Nim [3].

Данный выбор обусловлен следующими факторами:

- Компилируемость языка в C, C++, Objective C и JavaScript;
- Синтаксис языка близок к синтаксису ЯП Python.

Для тестирования производительности реализаций алгоритмов использовалась библиотека times.

При написании программного продукта использовалась среда разработки IntelliJ IDEA.

Данный выбор обусловлен тем, что данная среда разработки имеет плагин поддержки языка программирования Nim.

## 3.3 Листинг кода

В листингах 3.1 - 3.3 предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Алгоритм Копперсмита-Винограда в последовательной реализации

```
proc rowsComp(matrix : seq[seq[int]]) : seq[int]=
    var computedRows = newSeq[int](matrix.len)

    for i in 0..matrix.len - 1:
        var j = 0
        while j < matrix[0].len - 1:
            computedRows[i] += matrix[i][j] * matrix[i][j + 1]
            j += 2

    return computedRows

proc colsComp(matrix : seq[seq[int]]) : seq[int]=
    var computedCols = newSeq[int](matrix[0].len)

    var i = 0
    while i < matrix.len - 1:
        for j in 0..matrix[0].len - 1:
            computedCols[j] += matrix[i][j] * matrix[i + 1][j]
        i += 2

    return computedCols

proc winogradMult(fMat : seq[seq[int]], sMat : seq[seq[int]]) :
    seq[seq[int]]=
    if (fMat[0].len != sMat.len):
        return

    var computedRows = rowsComp(fMat)
    var computedCols = colsComp(sMat)

    var product = newSeqWith(fMat.len, newSeq[int](sMat[0].len))
    for i in 0..product.len - 1:
        for j in 0..product[0].len - 1:
            product[i][j] += -computedRows[i] - computedCols[j]

            var k = 0
            while k < sMat.len - 1:
                product[i][j] += (fMat[i][k] + sMat[k + 1][j]) *
                    (fMat[i][k + 1] + sMat[k][j])
                k += 2

    if sMat.len %% 2 != 0:
        var curK = sMat.len - 1
```

```
42          for i in 0..product.len - 1:
43              for j in 0..product[0].len - 1:
44                  product[i][j] += fMat[i][curK] * sMat[curK][j]
45
46      return product
```

Листинг 3.2: Реализация первой схемы параллельного алгоритма Копперсмита-Винограда

```
1  proc rowsCompThreadFunc(info: tuple[startOfInterval,
       endOfInterval : int, computedRows : ptr seq[int], matrix :
       seq[seq[int]]])=
2      for i in info.startOfInterval..info.endOfInterval - 1:
3          var j = 0
4          while j < info.matrix[0].len - 1:
5              info.computedRows[i] += info.matrix[i][j] * info.
                   matrix[i][j + 1]
6              j += 2
7
8  proc rowsCompParallel(matrix : seq[seq[int]]) : seq[int]=
9      var compRows = newSeq[int](matrix.len)
10     var compRowsPtr = addr compRows
11     var thr : array[0..THREADS, Thread[tuple[startOfInterval,
           endOfInterval: int, computedRows : ptr seq[int], matrix :
           seq[seq[int]]]]]
12     var size = (matrix.len / THREADS).int
13     for i in 0..thr.len - 2:
14         createThread(thr[i], rowsCompThreadFunc, (i * size, (i +
               1) * size, compRowsPtr, matrix))
15
16     createThread(thr[thr.len - 1], rowsCompThreadFunc, ((thr.len
           - 1) * size, matrix.len, compRowsPtr, matrix))
17
18     joinThreads(thr)
19     return compRows
20
21 proc colsCompThreadFunc(info : tuple[startOfInterval,
       endOfInterval : int, computedCols : ptr seq[int], matrix :
       seq[seq[int]]])=
22     var i = 0
23     while i < info.matrix.len - 1:
24         for j in info.startOfInterval..info.endOfInterval - 1:
25             info.computedCols[j] += info.matrix[i][j] * info.
                   matrix[i + 1][j]
26         i += 2
```

```
27
28
29 proc colsCompParallel(matrix : seq[seq[int]]) : seq[int]=
30     var compCols = newSeq[int](matrix[0].len)
31     var compColsPtr = addr compCols
32     var thr : array[0..THREADS, Thread[tuple[startOfInterval,
       endOfInterval: int, computedCols : ptr seq[int], matrix :
       seq[seq[int]]]]]
33     var size = (matrix[0].len / THREADS).int
34     for i in 0..thr.len - 2:
35         createThread(thr[i], colsCompThreadFunc, (i * size, (i +
           1) * size, compColsPtr, matrix))
36
37     createThread(thr[thr.len - 1], colsCompThreadFunc, ((thr.len
       - 1) * size, matrix[0].len, compColsPtr, matrix))
38
39     joinThreads(thr)
40     return compCols
41
42
43 proc prepareThreadProd(info : tuple[startOfInterval,
    endOfInterval: int, product : ptr seq[seq[int]], computedRows
    , computedCols : seq[int]]) {.thread.}=
44     for i in info.startOfInterval..info.endOfInterval - 1:
45         for j in 0..info.product[0].len - 1:
46                 info.product[i][j] += -info.computedRows[i] -
                    info.computedCols[j]
47
48 proc finThreadProd(info : tuple[startOfInterval, endOfInterval:
    int, product : ptr seq[seq[int]], fMat, sMat : seq[seq[int
    ]]]) {.thread.}=
49     for i in info.startOfInterval..info.endOfInterval - 1:
50         var k = 0
51         for j in 0..info.product[0].len - 1:
52             k = 0
53             while k < info.sMat.len - 1:
54                 info.product[i][j] += (info.fMat[i][k] + info.
                    sMat[k + 1][j]) * (info.fMat[i][k + 1] + info
                    .sMat[k][j])
55                 k += 2
56
57
58 proc multParallelFirst(fMat : seq[seq[int]], sMat : seq[seq[int
    ]], computedRows, computedCols : seq[int]) : seq[seq[int]]=
59     var product = newSeqWith(fMat.len, newSeq[int](sMat[0].len))
```

```
60
61      var prodPtr = addr product
62      var fThr : array[0..THREADS, Thread[tuple[startOfInterval,
            endOfInterval : int, product : ptr seq[seq[int]],
            computedRows, computedCols : seq[int]]]]
63      var size = (product.len / THREADS).int
64      for i in 0..fThr.len - 2:
65          createThread(fThr[i], prepareThreadProd, (i * size, (i +
                1) * size, prodPtr, computedRows, computedCols))
66      createThread(fThr[fThr.len - 1], prepareThreadProd, ((fThr.
            len - 1) * size, product.len, prodPtr, computedRows,
            computedCols))
67      joinThreads(fThr)
68
69      var sThr : array[0..THREADS, Thread[tuple[startOfInterval,
            endOfInterval : int, product : ptr seq[seq[int]], fMat,
            sMat : seq[seq[int]]]]]
70      for i in 0..sThr.len - 2:
71          createThread(sThr[i], finThreadProd, (i * size, (i + 1) *
                size, prodPtr, fMat, sMat))
72      createThread(sThr[sThr.len - 1], finThreadProd, ((sThr.len -
            1) * size, product.len, prodPtr, fMat, sMat))
73      joinThreads(sThr)
74
75      if sMat.len %% 2 != 0:
76          var curK = sMat.len - 1
77          for i in 0..product.len - 1:
78              for j in 0..product[0].len - 1:
79                  product[i][j] += fMat[i][curK] * sMat[curK][j]
80
81      return product
82
83
84  proc winogradMultParallelFirst(fMat : seq[seq[int]], sMat : seq[
        seq[int]]) : seq[seq[int]]=
85      if (fMat[0].len != sMat.len):
86          return
87
88      var computedRows = rowsCompParallel(fMat)
89      var computedCols = colsCompParallel(sMat)
90
91      var product = multParallelFirst(fMat, sMat, computedRows,
            computedCols)
92
93      return product
```

Листинг 3.3: Реализация второй схемы параллельного алгоритма Копперсмита-Винограда

```
proc twinThreadProd(info : tuple[startOfInterval, endOfInterval :
    int, product : ptr seq[seq[int]], computedRows, computedCols
    : seq[int], fMat, sMat : seq[seq[int]]]) {.thread.}=
    for i in info.startOfInterval..info.endOfInterval - 1:
        for j in info.startOfInterval..info.endOfInterval - 1:
            info.product[i][j] += -info.computedRows[i] - info.
                computedCols[j]
            var k = 0
            while k < info.sMat.len - 1:
                info.product[i][j] += (info.fMat[i][k] + info.
                    sMat[k + 1][j]) * (info.fMat[i][k + 1] + info
                    .sMat[k][j])
                k += 2

proc multParallelSecond(fMat : seq[seq[int]], sMat : seq[seq[int
    ]], computedRows, computedCols : seq[int]) : seq[seq[int]]=
    var product = newSeqWith(fMat.len, newSeq[int](sMat[0].len))

    var prodPtr = addr product
    var thr : array[0..THREADS, Thread[tuple[startOfInterval,
        endOfInterval : int, product : ptr seq[seq[int]],
        computedRows, computedCols : seq[int], fMat, sMat : seq[
        seq[int]]]]]
    var size = (product.len / THREADS).int
    for i in 0..thr.len - 2:
        createThread(thr[i], twinThreadProd, (i * size, (i + 1) *
            size, prodPtr, computedRows, computedCols, fMat,
            sMat))
    createThread(thr[thr.len - 1], twinThreadProd, ((thr.len - 1)
        * size, product.len, prodPtr, computedRows, computedCols
        , fMat, sMat))
    joinThreads(thr)

    if sMat.len %% 2 != 0:
        var curK = sMat.len - 1
        for i in 0..product.len - 1:
            for j in 0..product[0].len - 1:
                product[i][j] += fMat[i][curK] * sMat[curK][j]

    return product

proc winogradMultParallelSecond(fMat : seq[seq[int]], sMat : seq[
    seq[int]]) : seq[seq[int]]=
```

```
30      if (fMat[0].len != sMat.len):
31          return
32
33      var computedRows = rowsCompParallel(fMat)
34      var computedCols = colsCompParallel(sMat)
35
36      var product = multParallelSecond(fMat, sMat, computedRows,
            computedCols)
37
38      return product
```

## 3.4 Тестирование программного продукта

В таблице 3.1 приведены тесты для функций, реализующих алгоритм Копперсмита-Винограда. Тесты пройдены успешно.

Таблица 3.1: Тестирование функций

| Матрица 1 | Матрица 2 | Ожидаемый результат |
|---|---|---|
| $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 3 & 3 \\ 1 & 3 & 3 \\ 1 & 2 & 2 \end{pmatrix}$ | $\begin{pmatrix} 6 & 15 & 15 \\ 6 & 15 & 15 \\ 3 & 8 & 8 \end{pmatrix}$ |
| $\begin{pmatrix} 1 & 2 & 4 \\ 1 & 2 & 4 \end{pmatrix}$ | $\begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}$ | $\begin{pmatrix} 32 \\ 32 \end{pmatrix}$ |
| $\begin{pmatrix} 5 \end{pmatrix}$ | $\begin{pmatrix} 666 \end{pmatrix}$ | $\begin{pmatrix} 3330 \end{pmatrix}$ |
| $\begin{pmatrix} -1 & -2 & 3 \\ 1 & 2 & 3 \\ -1 & -2 & 3 \end{pmatrix}$ | $\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$ | $\begin{pmatrix} 4 & 4 & 4 \\ 14 & 14 & 14 \\ 4 & 4 & 4 \end{pmatrix}$ |
| $\begin{pmatrix} 666 & 666 \end{pmatrix}$ | $\begin{pmatrix} 777 & 777 \end{pmatrix}$ | Ошибка |

# 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.

```
Hint: /home/trvehazzk3r/Desktop/AlgorithmAnalysis/AA_lab_04/src/main  [Exec]
Rows: 3
Cols: 3
1 2 3
1 2 3
1 2 3

Got matrix:
    1     2     3
    1     2     3
    1     2     3

Rows: 3
Cols: 3
1 2 3
4 3 2
1 2 3

Got matrix:
    1     2     3
    4     3     2
    1     2     3

First parallel result:
   12    14    16
   12    14    16
   12    14    16

Second parallel result:
   12    14    16
   12    14    16
   12    14    16

Classic Winograd result:
   12    14    16
   12    14    16
   12    14    16
```

Рис. 4.1: Пример работы ПО.

```
Hint: /home/trvehazzk3r/Desktop/AlgorithmAnalysis/AA_lab_04/src/main  [Exec]
Rows: 3
Cols: 3
1 2 3
2 2 2
3 3 3

Got matrix:
    1     2     3
    2     2     2
    3     3     3

Rows: 3
Cols: 3
1 2 3
3 2 1
9 6 6

Got matrix:
    1     2     3
    3     2     1
    9     6     6

First parallel result:
   34    24    23
   26    20    20
   39    30    30

Second parallel result:
   34    24    23
   26    20    20
   39    30    30

Classic Winograd result:
   34    24    23
   26    20    20
   39    30    30
```

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 2.10 параллельного алгоритма Копперсмита-Винограда, Парал2КВ - реализация схемы 2.11 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.

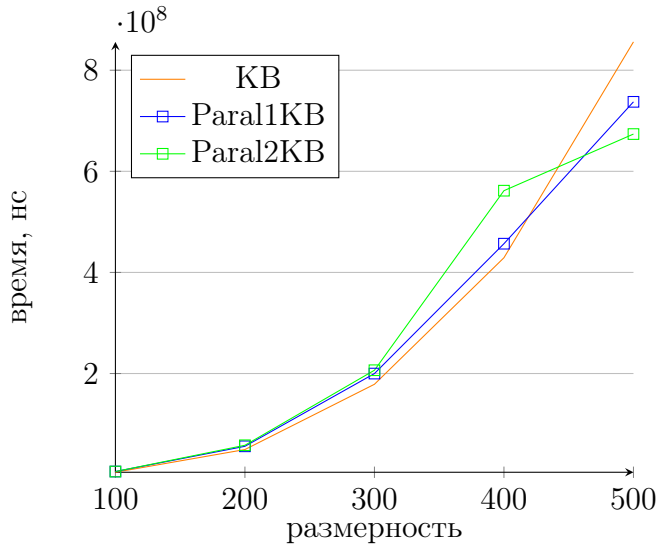


Рис. 4.3: Зависимость времени работы от размерности матриц (при 1 потоке).

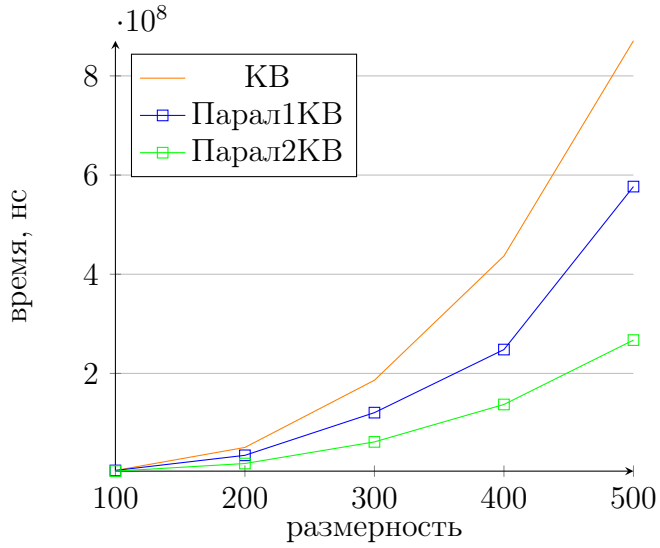


Рис. 4.4: Зависимость времени работы от размерности матриц (при 2 потоках)

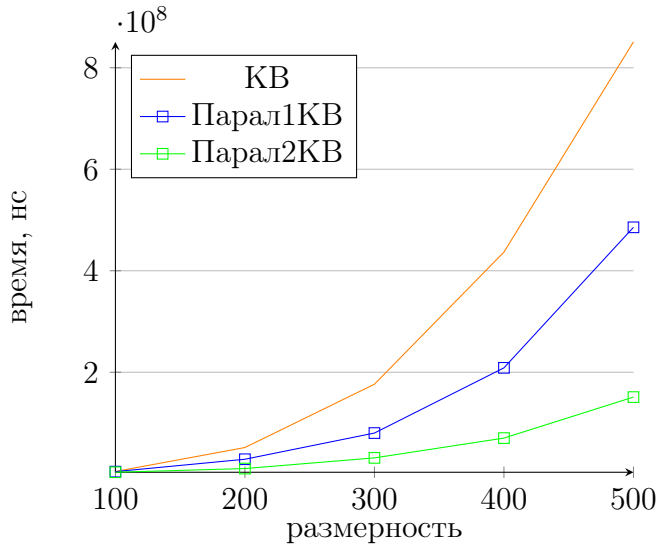


Рис. 4.5: Зависимость времени работы от размерности матриц (при 4 потоках)

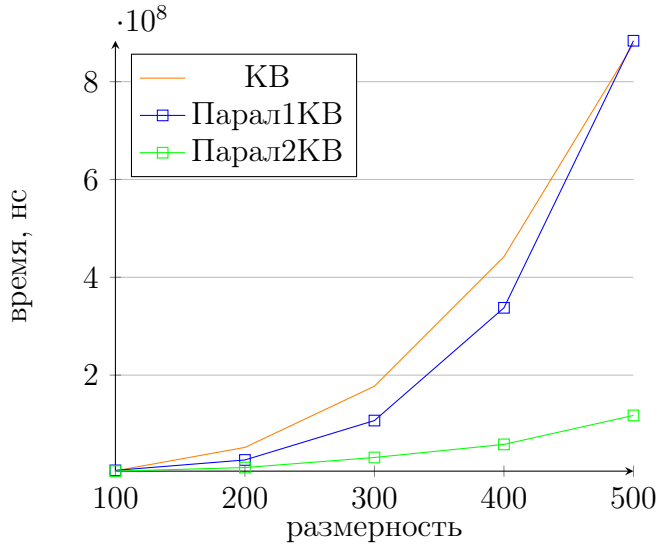


Рис. 4.6: Зависимость времени работы от размерности матриц (при 8 потоках)

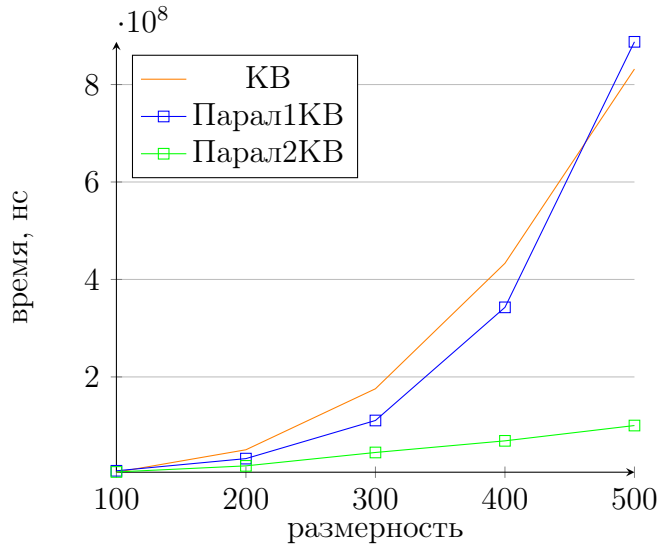


Рис. 4.7: Зависимость времени работы от размерности матриц (при 16 потоках)

# 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).