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 №4 по курсу "Анализ алгоритмов"

Тема Реализация параллельного алгоритма Копперсмита-Винограда

Студент Якуба Д. В.

Группа ИУ7-53Б

Оценка (баллы) _____

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

Москва — 2020 г.

Оглавление

Введение

Цели лабораторной работы

1. изучение параллельных вычислений;
2. изучение последовательного и двух параллельных реализаций алгоритмов Винограда;
3. реализация последовательного и двух параллельных алгоритмов Винограда;
4. проведение сравнительного анализа временных характеристик реализованных алгоритмов на основе экспериментальных данных;
5. подготовка отчёта по лабораторной работе;
6. получение практических навыков реализации алгоритмов на ЯП Nim.

Определение

Многопоточность — это способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB).

В тех случаях, когда многопроцессорные системы включают в себя несколько полных блоков обработки, многопоточность направлена на

максимизацию использования ресурсов одного ядра, используя параллелизм на уровне потоков, а также на уровне инструкций. Поскольку эти два метода являются взаимодополняющими, их иногда объединяют в системах с несколькими многопоточными ЦП и в ЦП с несколькими многопоточными ядрами.

Многопоточная парадигма стала более популярной с конца 1990-х годов, поскольку усилия по дальнейшему использованию параллелизма на уровне инструкций застопорились. Смысл многопоточности — квазимногозадачность на уровне одного исполняемого процесса. Значит, все потоки процесса помимо общего адресного пространства имеют и общие дескрипторы файлов. Выполняющийся процесс имеет как минимум один (главный) поток.

1 Аналитическая часть

1.1 Алгоритм Копперсмита-Винограда умножения матриц

Пусть даны две прямоугольные матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.3}$$

будет называться произведением матриц A и B.

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4$, что эквивалентно (1.4):

$$V \cdot W = (v_1+w_2)(v_2+w_1)+(v_3+w_4)(v_4+w_3)-v_1v_2-v_3v_4-w_1w_2-w_3w_4. \quad (1.4)$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволит для каждого элемента выполнять лишь два умножения и пять сложений, складывая затем только лишь с 2 предварительно посчитанными суммами соседних элементов текущих строк и столбцов [1]. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного [2].

Важно заметить, что каждый элемент матрицы C вычисляется независимо от других, а матрицы A и B не изменяются, поэтому для распараллеливания алгоритма будет достаточно распределить элементы каждой строки матрицы C между потоками.

Вывод

Был рассмотрен алгоритм умножения матриц Копперсмита-Винограда. В данной работе стоит задача реализации данного алгоритма в последовательном и параллельных видах. Независимость вычислений элементов результата даёт возможность реализовать несколько параллельных вариантов исполнения данного алгоритма.

2 | Конструкторская часть

2.1 Блок-схема алгоритма Копперсмита-Винограда

Блок-схема алгоритма Копперсмита-Винограда предоставлена на рисунках 2.1-2.3.

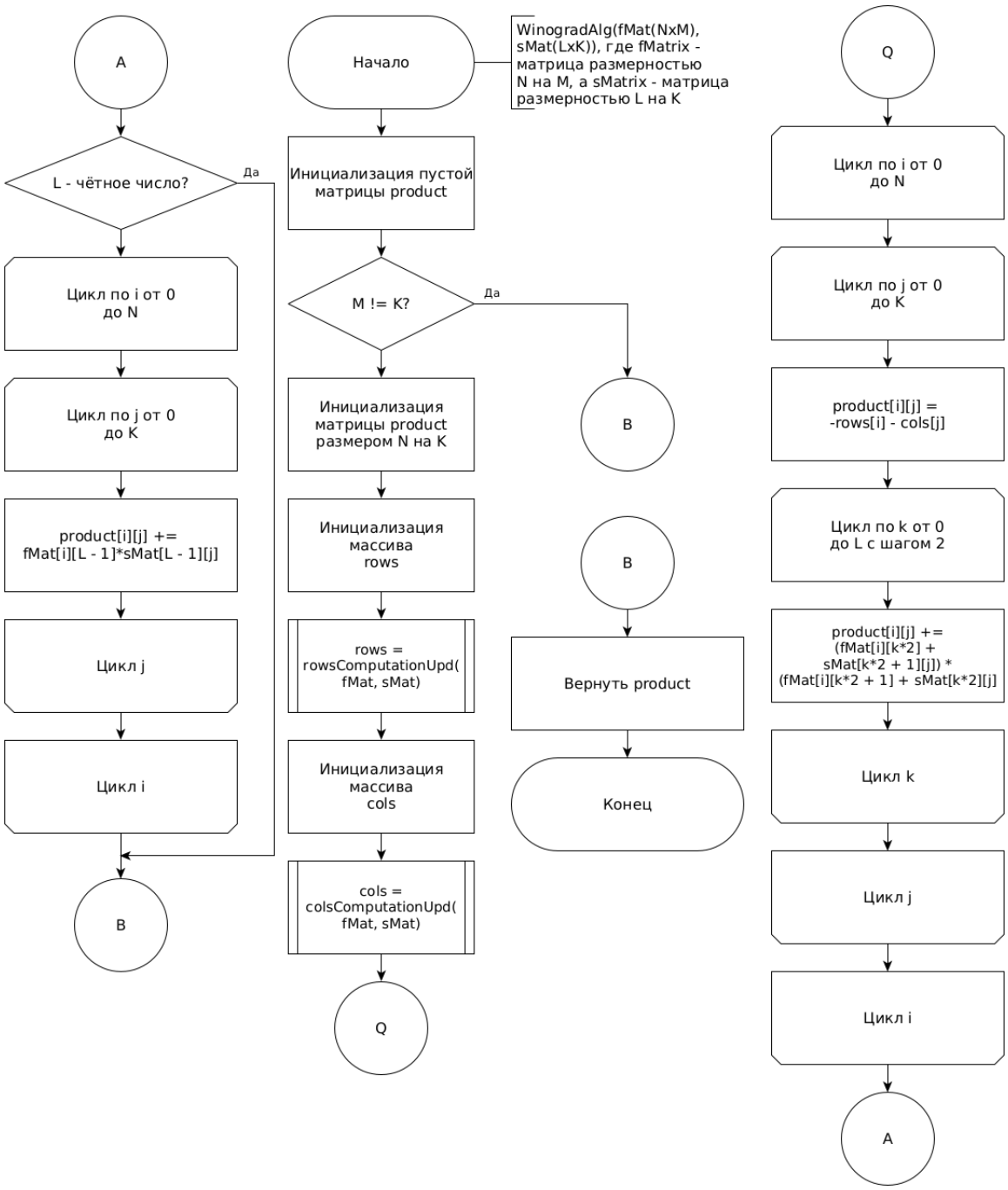

Рис. 2.1: Блок-схема алгоритма Копперсмита-Винограда.

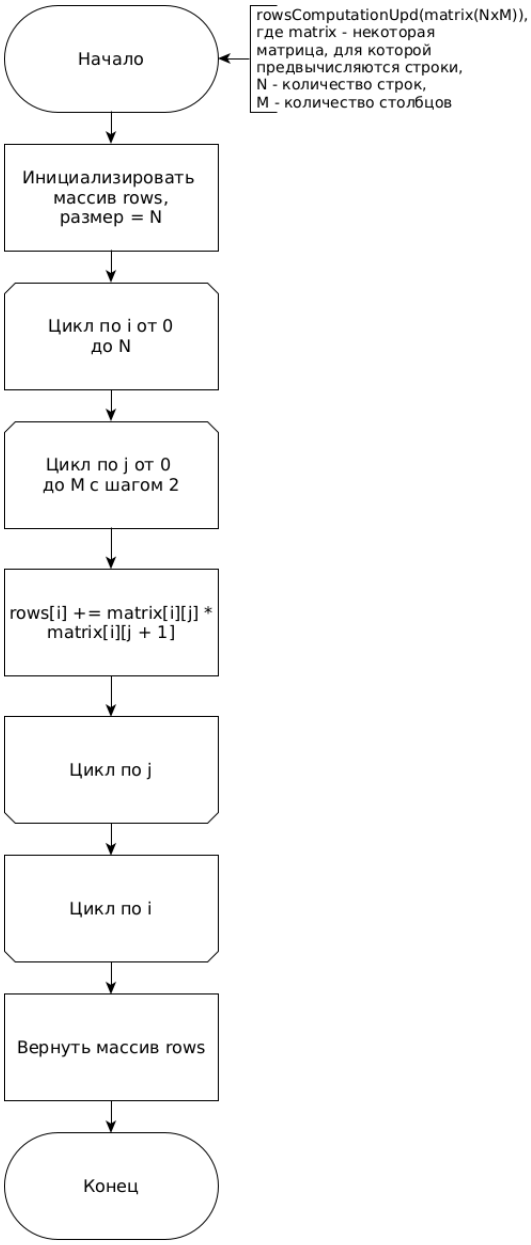

Рис. 2.2: Блок-схема предрасчёта строк для алгоритма Копперсмита-Винограда.

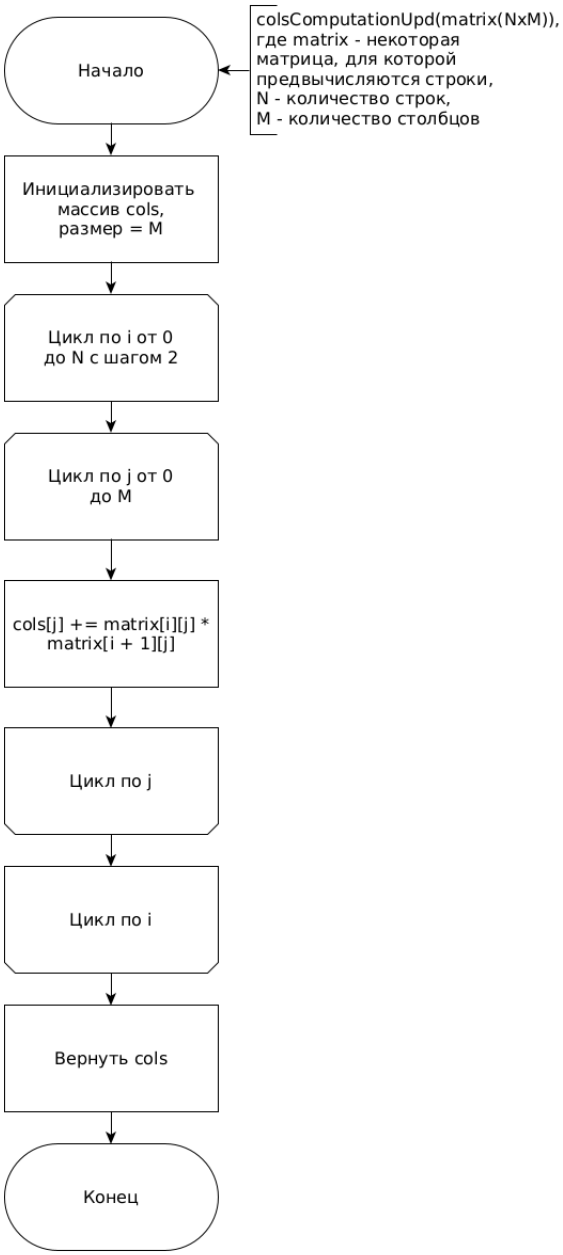

Рис. 2.3: Блок-схема предрасчёта столбцов для алгоритма Копперсмита-Винограда.

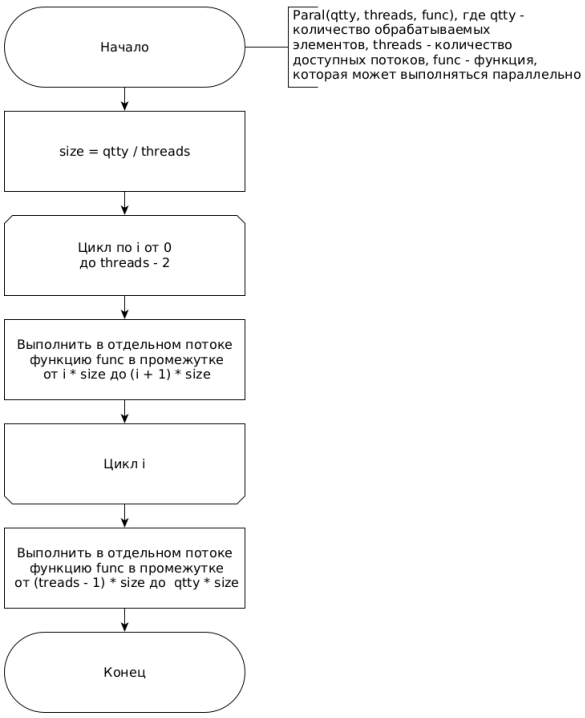

Рис. 2.4: Блок-схема алгоритма работы функции, запускающей в требуемом количестве потоков функцию-аргумент.

2.2 Блок-схема алгоритма работы функции, запускающая в требуемом количестве потоков функцию-аргумент

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

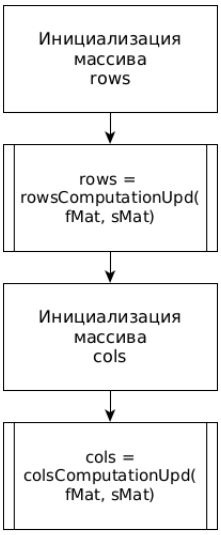

Рис. 2.5: Часть блок-схемы, в которой происходит вызов подпрограмм предрасчёта строк и столбцов.

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.

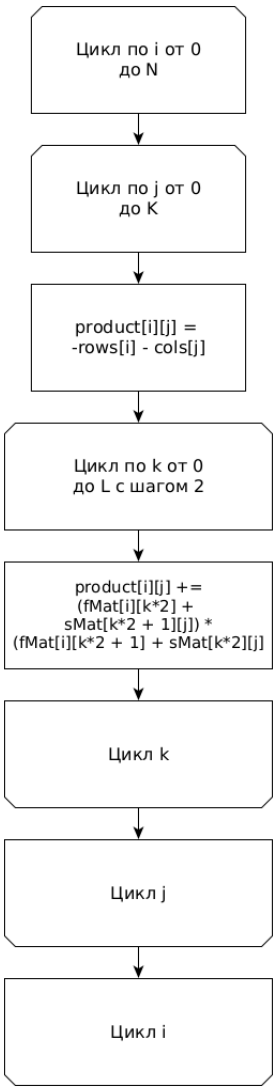

Рис. 2.6: Часть блок-схемы, в которой происходит расчёт значения каждого элемента матрицы.

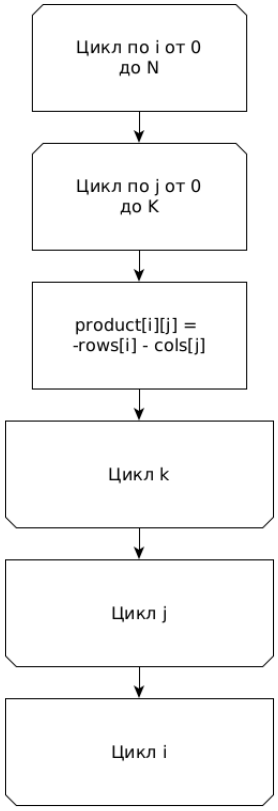

Рис. 2.7: Первая часть рассматриваемого этапа.

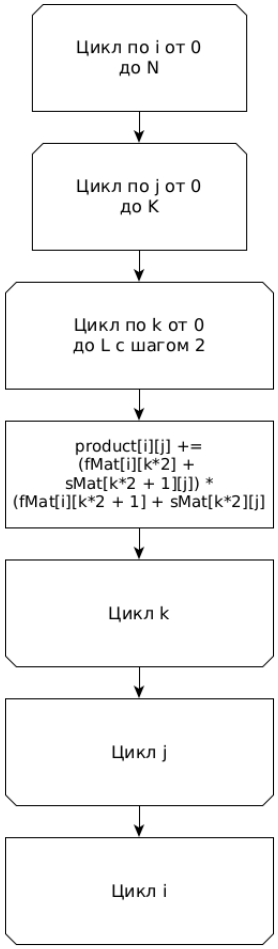

Рис. 2.8: Вторая часть рассматриваемого этапа.

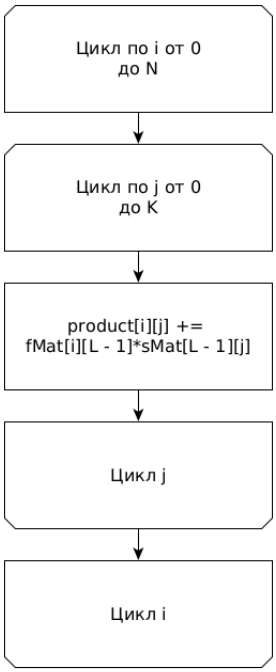

Рис. 2.9: Довычисление значений итоговой матрицы при нечётной размерности оной.

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.

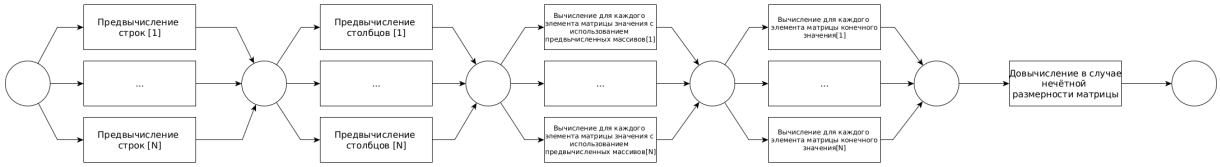

Рис. 2.10: Схема с одновременным выполнением двух частей этапа вычисления элементов матрицы.

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.

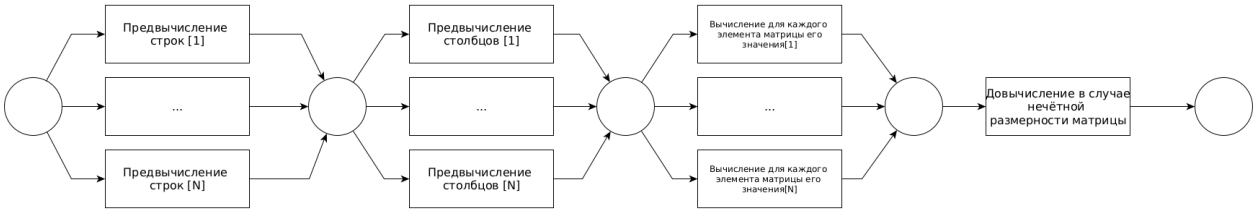

Рис. 2.11: Схема с последовательным выполнением двух частей этапа вычисления элементов матрицы.

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 ?? - ?? предоставлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Алгоритм Капперсмита-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
```

```
        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
```

Листинг 3.2: Реализация первой схемы параллельного алгоритма Копперсмита-Виноградa

```
proc rowsCompThreadFunc(info: tuple[startOfInterval,
    endOfInterval : int,
                                    computedRows : ptr seq[int],
                                    matrix : seq[seq[int]]])=
    for i in info.startOfInterval..info.endOfInterval - 1:
        var j = 0
        while j < info.matrix[0].len - 1:
            info.computedRows[i] += info.matrix[i][j] * info.
                matrix[i][j + 1]
            j += 2

proc rowsCompParallel(matrix : seq[seq[int]]) : seq[int]=
    var compRows = newSeq[int](matrix.len)
    var compRowsPtr = addr compRows
    var thr : array[0..THREADS, Thread[tuple[startOfInterval,
        endOfInterval: int,
                                            computedRows : ptr
                                                seq[int],
                                            matrix : seq[seq[int
                                                ]]]]]
    var size = (matrix.len / THREADS).int
    for i in 0..thr.len - 2:
        createThread(thr[i], rowsCompThreadFunc, (i * size, (i +
            1) * size, compRowsPtr, matrix))

    createThread(thr[thr.len - 1], rowsCompThreadFunc, ((thr.len
        - 1) * size, matrix.len, compRowsPtr, matrix))

    joinThreads(thr)
    return compRows

proc colsCompThreadFunc(info : tuple[startOfInterval,
    endOfInterval : int,
                        computedCols : ptr seq[int],
                        matrix : seq[seq[int]]])=
    var i = 0
```

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

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

```
93          for i in 0..product.len - 1:
94              for j in 0..product[0].len - 1:
95                  product[i][j] += fMat[i][curK] * sMat[curK][j]
96
97      return product
98
99
100 proc winogradMultParallelFirst(fMat : seq[seq[int]], sMat : seq[
        seq[int]]) : seq[seq[int]]=
101     if (fMat[0].len != sMat.len):
102         return
103
104     var computedRows = rowsCompParallel(fMat)
105     var computedCols = colsCompParallel(sMat)
106
107     var product = multParallelFirst(fMat, sMat, computedRows,
            computedCols)
108
109     return product
```

Листинг 3.3: Реализация второй схемы параллельного алгоритма Копперсмита-Винограда

```
1  proc twinThreadProd(info : tuple[startOfInterval, endOfInterval :
       int,
2                                   product : ptr seq[seq[int]],
3                                   computedRows, computedCols :
                                        seq[int],
4                                   fMat, sMat : seq[seq[int]]])
                                        {.thread.}=
5      for i in info.startOfInterval..info.endOfInterval - 1:
6          for j in info.startOfInterval..info.endOfInterval - 1:
7              info.product[i][j] += -info.computedRows[i] - info.
                   computedCols[j]
8              var k = 0
9              while k < info.sMat.len - 1:
10                 info.product[i][j] += (info.fMat[i][k] + info.
                       sMat[k + 1][j]) * (info.fMat[i][k + 1] + info
                       .sMat[k][j])
11                 k += 2
12
13 proc multParallelSecond(fMat : seq[seq[int]], sMat : seq[seq[int
       ]], computedRows, computedCols : seq[int]) : seq[seq[int]]=
14     var product = newSeqWith(fMat.len, newSeq[int](sMat[0].len))
15
```

```
16      var prodPtr = addr product
17      var thr : array[0..THREADS, Thread[tuple[startOfInterval,
            endOfInterval : int,
18                                              product : ptr seq[
                                                    seq[int]],
19                                              computedRows,
                                                    computedCols :
                                                    seq[int],
20                                              fMat, sMat : seq[seq
                                                    [int]]]]]
21      var size = (product.len / THREADS).int
22      for i in 0..thr.len - 2:
23          createThread(thr[i], twinThreadProd, (i * size, (i + 1) *
                size, prodPtr, computedRows, computedCols, fMat,
               sMat))
24      createThread(thr[thr.len - 1], twinThreadProd, ((thr.len - 1)
            * size, product.len, prodPtr, computedRows, computedCols
           , fMat, sMat))
25      joinThreads(thr)
26
27      if sMat.len %% 2 != 0:
28          var curK = sMat.len - 1
29          for i in 0..product.len - 1:
30              for j in 0..product[0].len - 1:
31                  product[i][j] += fMat[i][curK] * sMat[curK][j]
32
33      return product
34
35  proc winogradMultParallelSecond(fMat : seq[seq[int]], sMat : seq[
        seq[int]]) : seq[seq[int]]=
36      if (fMat[0].len != sMat.len):
37          return
38
39      var computedRows = rowsCompParallel(fMat)
40      var computedCols = colsCompParallel(sMat)
41
42      var product = multParallelSecond(fMat, sMat, computedRows,
            computedCols)
43
44      return product
```

3.4 Тестирование программного продукта

В таблице 3.1 приведены тесты для функций, реализующих алгоритм сортировки пузырьком, вставками и выбором. Тесты пройдены успешно.

Начальный массив	Ожидаемый результат	Полученный результат
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[0, 0, 0]$	$[0, 0, 0]$	$[0, 0, 0]$
$[1, 2, 3]$	$[1, 2, 3]$	$[1, 2, 3]$
$[-8, 2, 3, -2]$	$[-8, -2, 2, 3]$	$[-8, -2, 2, 3]$
[]	[]	[]

Таблица 3.1: Тестирование функций

Вывод

Спроектированные алгоритмы сортировок 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.

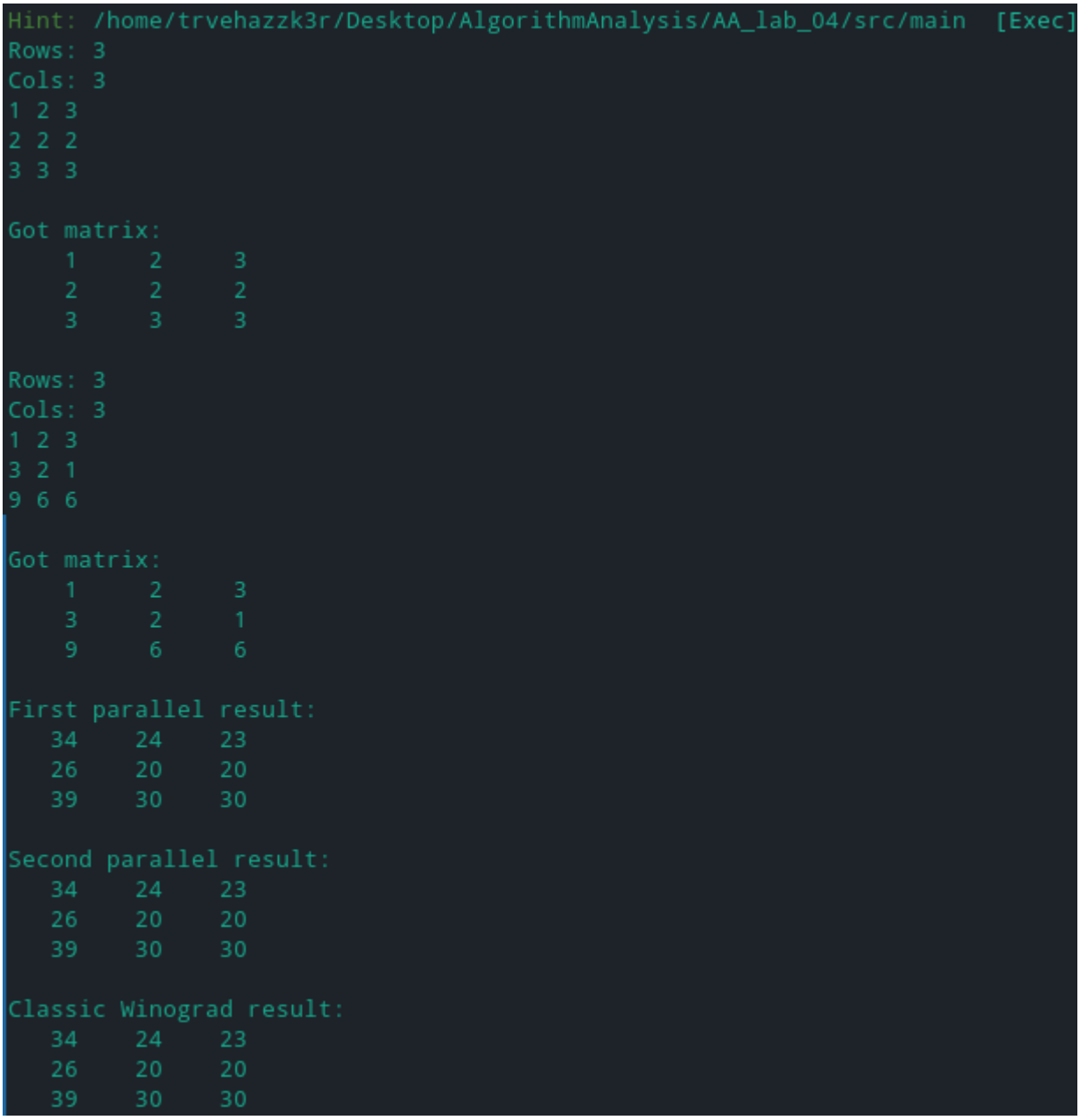

Рис. 4.2: Пример работы ПО.

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 выводы в каждом из случаев...

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

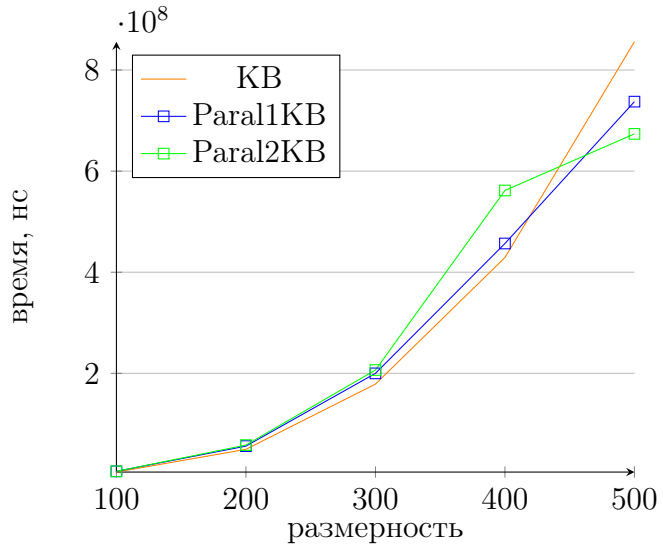

Рис. 4.3: Зависимость времени работы от размерности матриц (при 1 потоке).

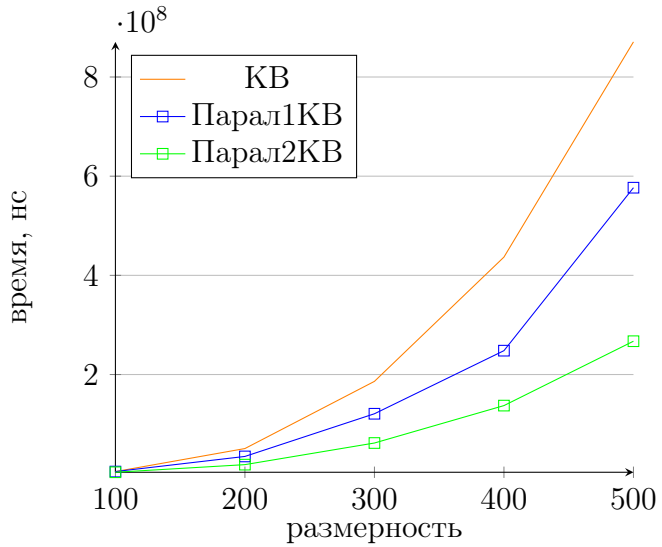

Рис. 4.4: Зависимость времени работы от размерности матриц (при 2 потоках)

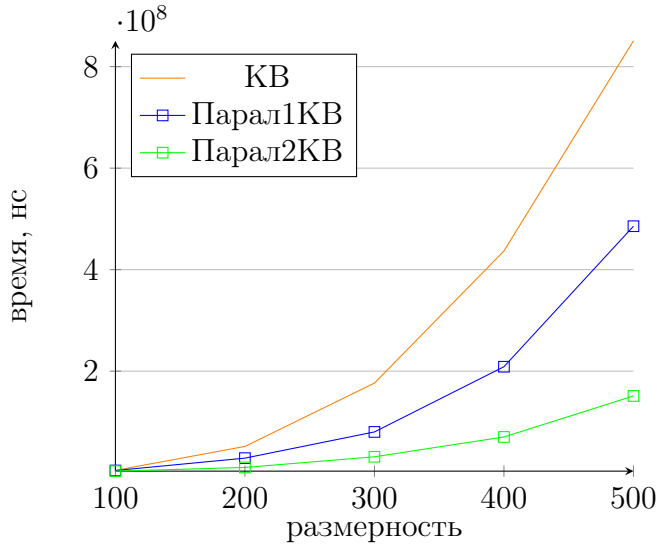

Рис. 4.5: Зависимость времени работы от размерности матриц (при 4 потоках)

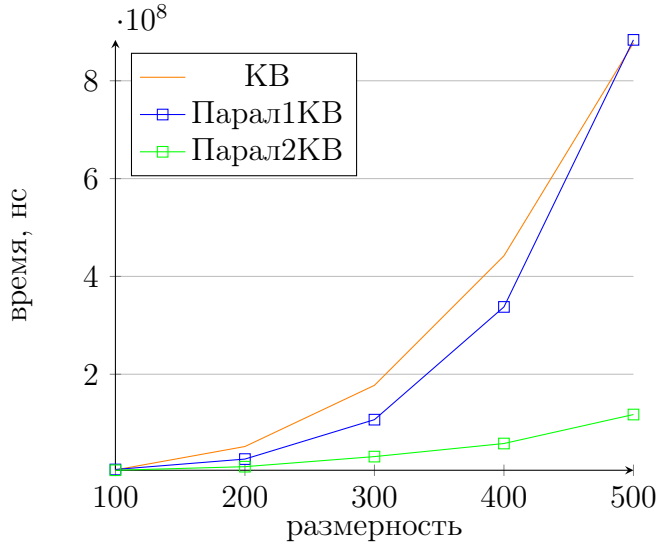

Рис. 4.6: Зависимость времени работы от размерности матриц (при 8 потоках)

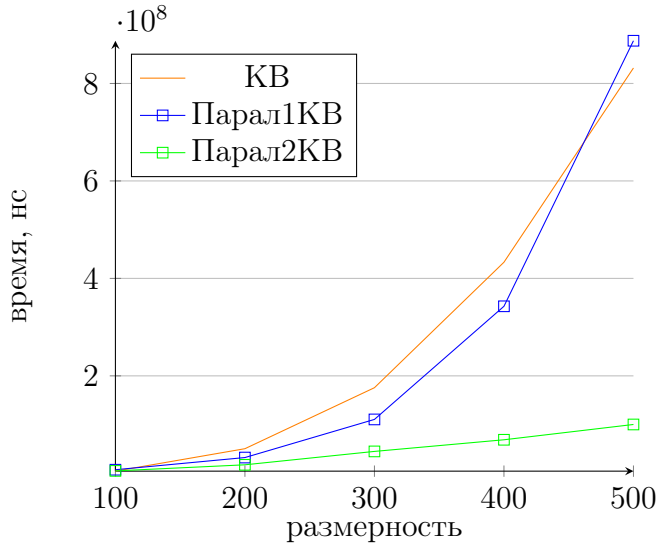

Рис. 4.7: Зависимость времени работы от размерности матриц (при 16 потоках)

Заключение

В ходе выполнения лабораторной работы:

1. были изучены алгоритмы сортировки пузырьком, вставками и выбором;
2. были реализованы алгоритмы сортировки пузырьком, вставками и выбором;
3. был проведён сравнительный анализа трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
4. был проведён сравнительный анализ алгоритмов на основе экспериментальных данных;
5. был подготовлен отчёт по лабораторной работе;
6. были получены практические навыки реализации алгоритмов на ЯП Kotlin.

Исследования показали, что...

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).