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 дисциплине "Анализ алгоритмов"

Тема Параллельное нахождение определителя матрицы

Студент Шацкий Р.Е.

Группа ИУ7-55Б

Оценка (баллы) _____

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

Москва — 2021 г.

Оглавление

Введение

Многопоточность — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременновыполнять 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 или 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.

Многопоточность (как доктрину программирования) не следует путать ни с многозадачностью, ни с многопроцессорностью, несмотря на то, что операционные системы, реализующие многозадачность, как правило, реализуют и многопоточность.

Достоинства:

- облегчение программы посредством использования общего адресного пространства;
- меньшие затраты на создание потока в сравнении с процессами;
- повышение производительности процесса за счёт распараллеливания процессорных вычислений;
- если поток часто теряет кэш, другие потоки могут продолжать использовать неиспользованные вычислительные ресурсы.

Недостатки:

- несколько потоков могут вмешиваться друг в друга при совместном использовании аппаратных ресурсов [?];
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения [?];

- проблема планирования потоков;
- специфика использования. Вручную настроенные программы на ассемблере, использующие расширения MMX или AltiVec и выполняющие предварительные выборки данных, не страдают от потерь кэша или неиспользуемых вычислительных ресурсов. Таким образом, такие программы не выигрывают от аппаратной многопоточности и действительно могут видеть ухудшенную производительность из-за конкуренции за общие ресурсы.

Однако несмотря на количество недостатков, перечисленных выше, многопоточная парадигма имеет большой потенциал на сегодняшний день, и при должном написании кода позволяет значительно ускорить однопоточные алгоритмы.

Цель лабораторной работы

Целью данной лабораторной работы является изучение и реализация параллельных вычислений.

Задачи лабораторной работы

В рамках выполнения работы необходимо решить следующие задачи:

- изучить понятие параллельных вычислений;
- реализовать последовательную и параллельную реализацию алгоритма нахождения определителя матрицы;
- сравнить временные характеристики реализованных алгоритмов экспериментально.

1 Аналитическая часть

В данном разделе дано определение определителя матрицы и рассмотрены основные способы его нахождения.

1.1 Описание задачи

Определитель матрицы или просто определитель играет важную роль в решении систем линейных уравнений. Определитель матрицы A обозначается как $\det A$, $\Delta A, |A|$

Сформулировать определение определителя матрицы можно на основе его свойств. Определителем вещественной матрицы называется функция $\det : \mathbb{R}^{n\times n} \to \mathbb{R}$, обладающая следующими свойствами:

1. $\det(A)$ - кососимметрическая функция строк (столбцов) матрицы A
2. $\det(A)$ - полилинейная функция строк(столбцов) матрицы A
3. $\det(A) = 1$, где E - единичная $n \times n$-матрица

1.2 Нахождение определителя матрицы

Ниже описаны способы нахождения определителя матрицы размеров 1×1, 2×2, 3×23, $n \times 1n$.

1.2.1 Матрица 1×1

Для матрицы первого порядка значение детерминанта равно единственному элементу этой матрицы:

$$\Delta = |a_{11}| = a_{11} \tag{1.1}$$

1.2.2 Матрица 2×2

Для матрицы 2×2 определитель вычисляется следующим образом:

$$\Delta = \begin{vmatrix} a & c \\ b & d \end{vmatrix} = ad - bc \tag{1.2}$$

Абсолютное значение определителя $|ad - bc|$ равно площади параллелограмма с вершинами $(0, 0), (a, b), (a + c, b + d), (c, d)$.

1.2.3 Матрица 3×3

Определитель матрицы 3×3 можно вычислить по формуле:

$$\Delta = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \cdot \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \cdot \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \cdot \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} \tag{1.3}$$

Определитель матрицы, составленной из векторов a, b, c представляет собой объём параллелепипеда, натянутого на вектора a, b, c.

1.2.4 Матрица $n \times n$

В общем случае, для матриц $n \times n$, где $n > 2$ определитель можно вычислить, применив следующую рекурсивную формулу:

$$\Delta = \sum_{j=1}^{n} (-1)^{1+j} \cdot a_{1j} \cdot M_j^{-1}, \text{ где}$$

M_j^{-1} - дополнительный минор к элементу a_{ij} (1.4)

В данной лабораторной работе стоит задача распараллеливания алгоритма нахождения определителя матрицы. Так как каждое слагаемое для вычисления итогового определителя вычисляется независимо от других и матрица не изменяется, для параллельного вычисления определителя было решено распределять задачу вычисления слагаемых между потоками.

1.3 Вывод

Обычный алгоритм нахождения определителя матрицы размера $n \times n$ независимо вычисляет слагаемые для нахождения итогового определителя, что дает возможность для реализации параллельного варианта алгоритма.

2 Конструкторская часть

Данный раздел содержит схемы алгоритмов, реализуемых в работе (Стандартный рекурсивный алгоритм нахождения определителя, алгоритм с использованием потоков и распределение слагаемых между потоками) и теоретический рассчет повышения эффективности исполнения алгоритма по времени.

2.1 Схемы алгоритмов

В данном пункте раздела представлены схемы реализуемых в работе алгоритмов.

2.1.1 Схема классического алгоритма

На рисунке 2.1 представлена схема рекурсивного алгоритм нахождения определителя.

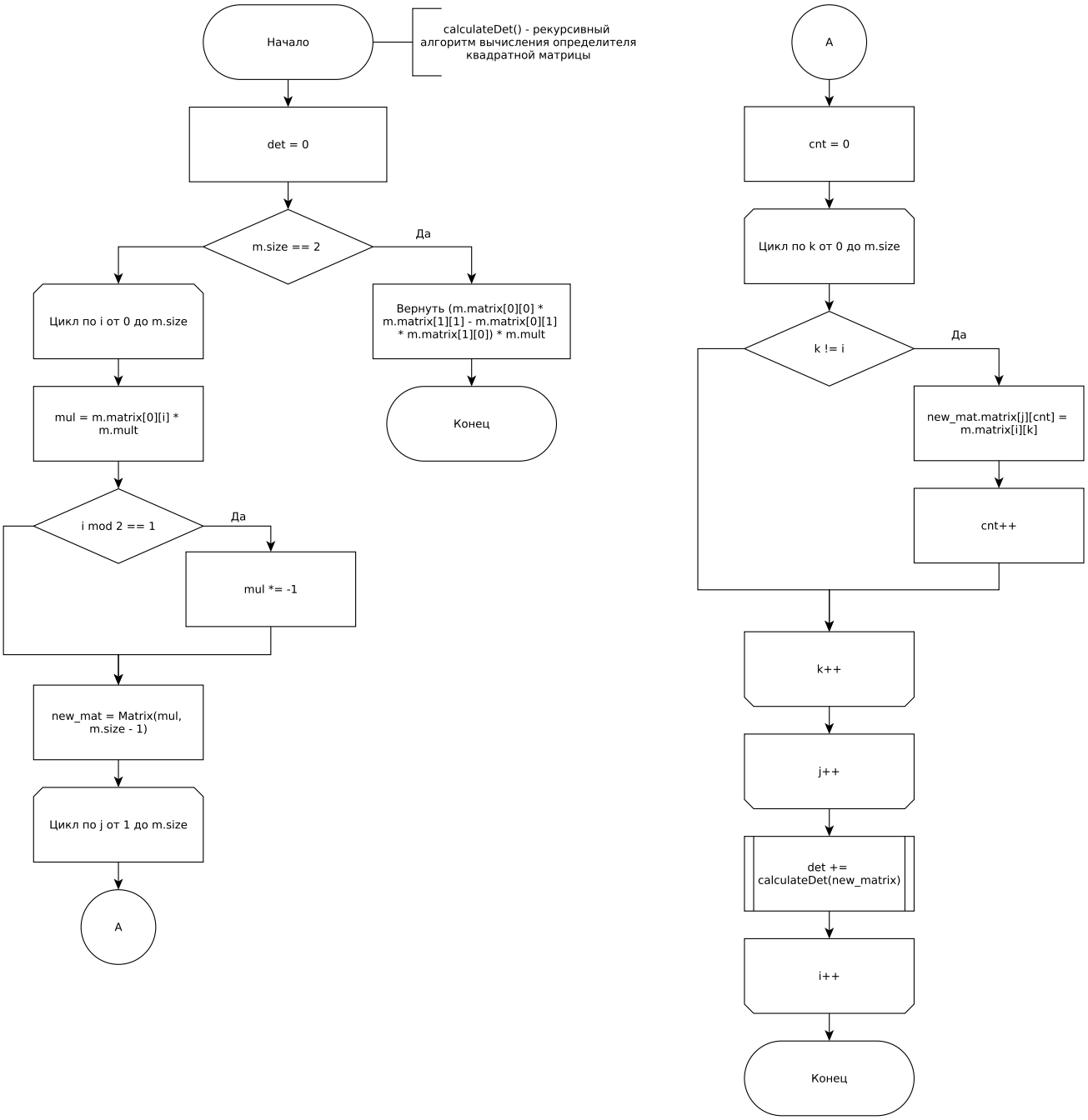

Рис. 2.1:

Схема рекурсивного алгоритм нахождения определителя

На рисунке 2.2 представлена схема алгоритма подсчета слагаемых итогового определителя матрицы при использовании потоков.

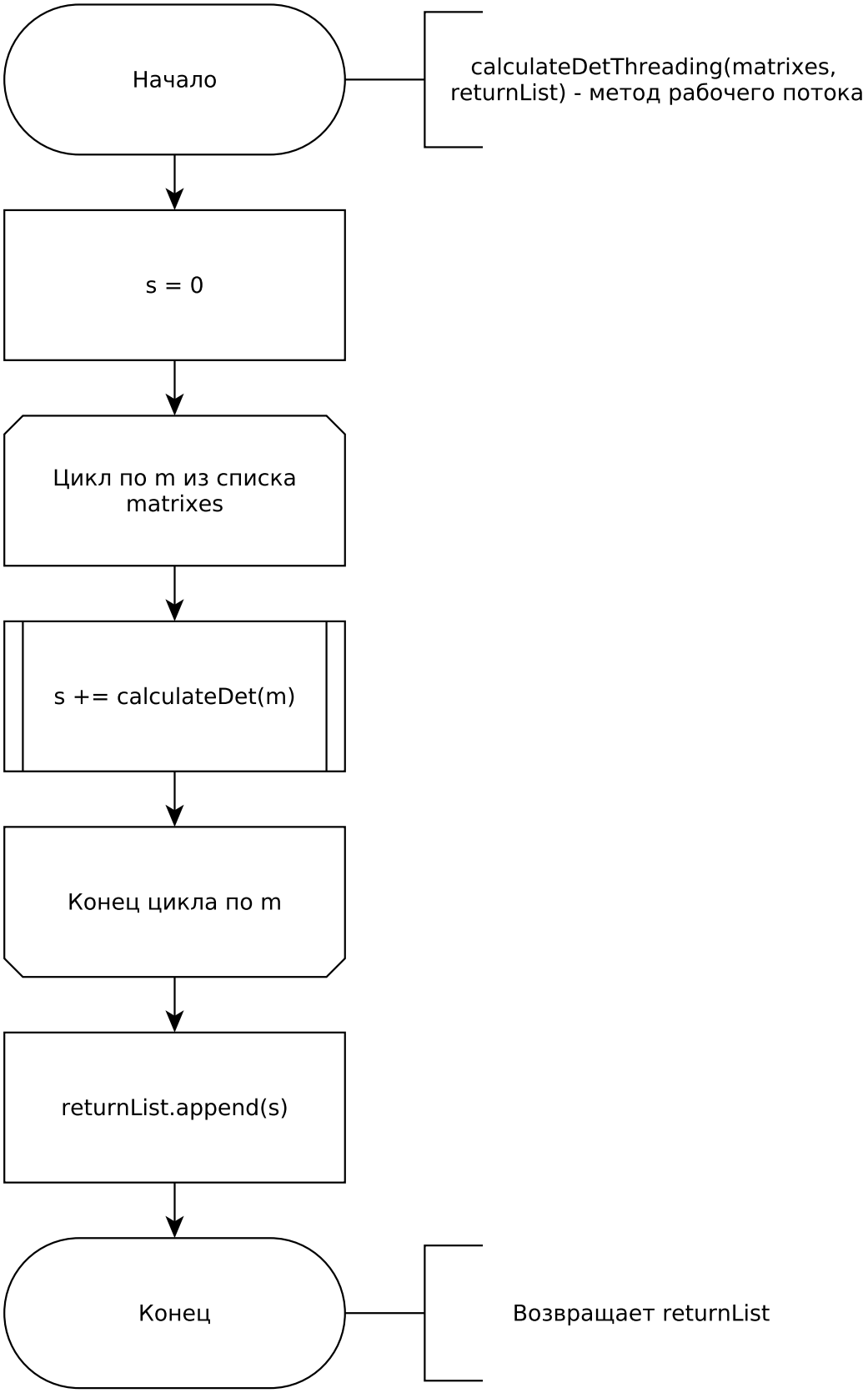

Рис. 2.2:

Схема рекурсивного алгоритм нахождения определителя

На рисунках 2.3 - 2.4 представлена схема алгоритма создания потоков, разделения задач между ними и нахождения итогового определителя матрицы.

Рис. 2.3:

Схема рекурсивного алгоритм нахождения определителя

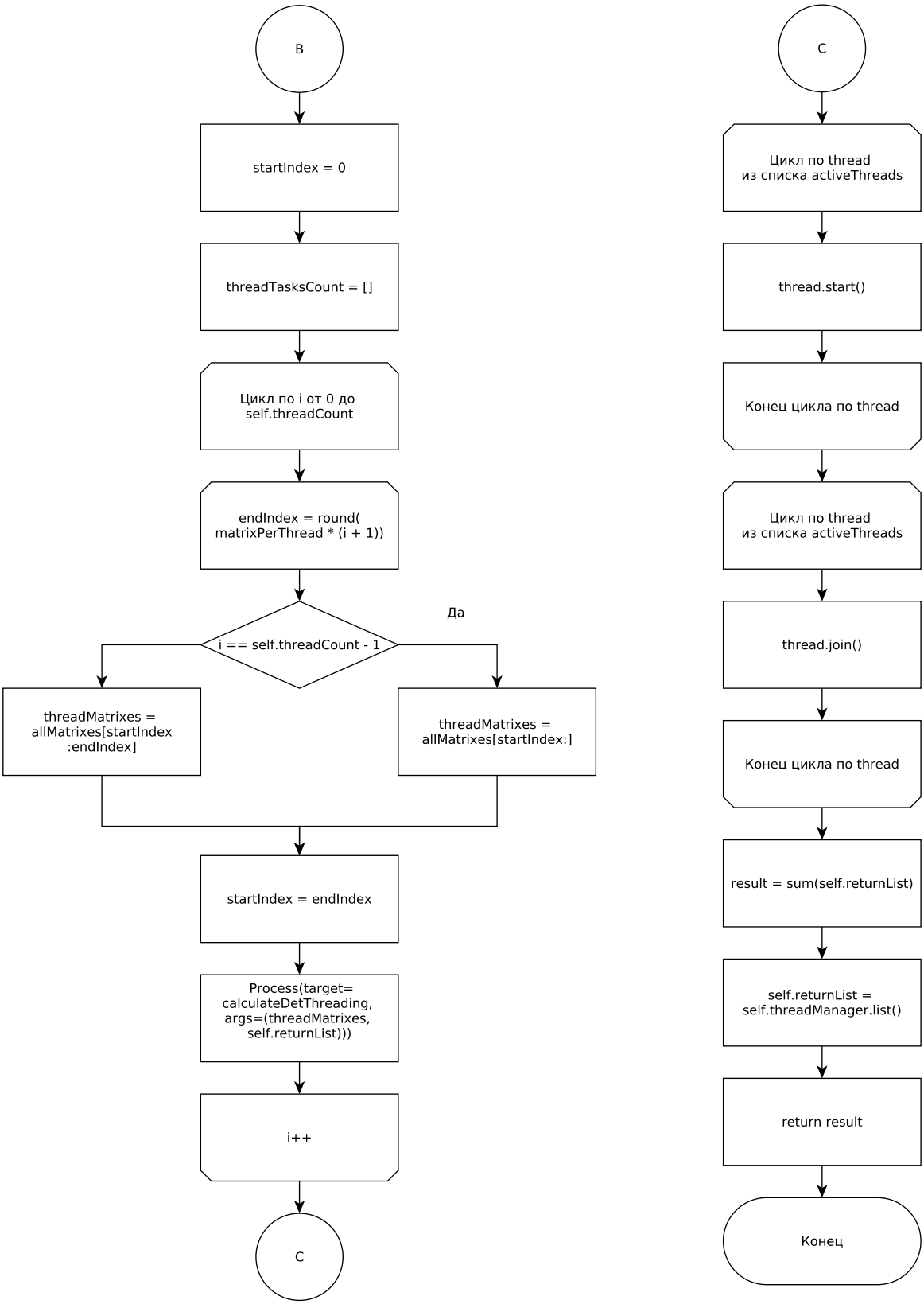

Рис. 2.4:

Схема рекурсивного алгоритм нахождения определителя

2.1.2 Теоретический рассчет эффективности по времени

При параллельном выполнении алгоритма изначальная матрица размером $n \times n$ делится на n матриц со значением $mult = (-1)^{j+1} \cdot a_{1j}$, где j - номер элемента в первом ряду, для которого находится значение минора. Далее "подматрицы"равномерно распределяются между потоками для вычисления значения миноров. По мере работы потоки записывают результат в общий массив, и после окончания работы всех потоков определитель исходной матрицы считается как сумма элементов в общем массиве.

При таком методе распараллеливания алгоритма для матрицы размером $n \times n$ эффективность алгоритма по времени исполнения должна повышаться примерно в k раз, где k - количество потоков, при $1 < k \leq n/2$ или $k = n$. Однако при $n/2 < k < n$, время исполнения алгоритма не увеличится более чем в $n/2$ раза, так как часть потоков будут искать два слагаемых итогового определителя, а часть - одно, в таком случае произойдет простой второй части потоков.

2.2 Вывод

В данном разделе на основе приведенных в аналитическом разделе теоретических данных были составлены схемы алгоритмов для реализации в технологической части. Были составлены схемы разделения вычисления определителя на потоки. Проведены теоретические расчеты эффективности параллельного алгоритма нахождения определителя.

3 Технологическая часть

Данный раздел содержит обоснование выбора языка и среды разработки, реализацию алгоритмов.

3.1 Требования к программному обеспечению

Требования, выдвигаемые к разрабатываемому ПО:

- выходные данные - сгенерированная матрица, время, затраченное на вычисление определителя при использовании от 0 до 15 потоков.

3.2 Средства реализации

Для реализации программы был выбран язык программирования Python [?]. Такой выбор обусловлен следующими причинами:

- имеется большой опыт разработки;
- удобные средства для работы с потоками;
- обладает информативной документацией.

3.3 Реализация алгоритмов

В листингах ?? - ?? представлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Рекурсивный алгоритм

```
1          def calculateSum(m: Matrix):
2      s = 0
3      if m.size == 2:
4          return (m.matrix[0][0] * m.matrix[1][1] - m.matrix[0][1] * m.matrix
               [1][0]) * m.mult
5
6      for i in range(m.size):
7          mul = m.matrix[0][i] * m.mult
8          if i % 2 == 1:
9              mul *= -1
10
11         size = m.size - 1
```

```
12          matrix = []
13
14          for j in range(1, m.size):
15              matrix.append([])
16              for k in range(m.size):
17                  if k != i:
18                      matrix[-1].append(m.matrix[j][k])
19
20          s += calculateSum(Matrix(size, mul, matrix))
21      return s
```

Листинг 3.2: Параллельный алгоритм с делением на потоки

```
1           def calculateSumThreading(matrixes, returnList: list):
2       t = time.time()
3       s = 0
4
5       for m in matrixes:
6           if m.size == 2:
7               returnList.append((m.matrix[0][0] * m.matrix[1][1] - m.matrix
                    [0][1] * m.matrix[1][0]) * m.mult)
8               continue
9
10          for i in range(m.size):
11              mul = m.matrix[0][i] * m.mult
12              if i % 2 == 1:
13                  mul *= -1
14
15              size = m.size - 1
16              matrix = []
17
18              for j in range(1, m.size):
19                  matrix.append([])
20                  for k in range(m.size):
21                      if k != i:
22                          matrix[-1].append(m.matrix[j][k])
23
24              s += calculateSum(Matrix(size, mul, matrix))
25              # allMatrixes.append(Matrix(size, mul, matrix))
26
27      returnList.append(s)
28      # print(f'process {getpid()}, time {time.time() - t}')
29
30  class Solver:
31      def __init__(self, matrix: Matrix = None):
32          self.m = matrix
33          if self.m is None:
34              self.m = Matrix(10).randomize()
35
36          self.threadCount = 1
```

```
37          self.threadManager = Manager()
38          self.returnList = self.threadManager.list()
39
40
41      def solve(self):
42          activeThreads = []
43          matrixPerThread = self.m.size / self.threadCount
44          allMatrixes = []
45
46          for i in range(self.m.size):
47              mul = self.m.matrix[0][i] * self.m.mult
48              if i % 2 == 1:
49                  mul *= -1
50
51              size = self.m.size - 1
52              matrix = []
53
54              for j in range(1, self.m.size):
55                  matrix.append([])
56                  for k in range(self.m.size):
57                      if k != i:
58                          matrix[-1].append(self.m.matrix[j][k])
59
60              allMatrixes.append(Matrix(size, mul, matrix))
61
62          startMatrix = 0
63          threadTasksCount = []
64
65          for i in range(self.threadCount):
66              endMatrix = round(matrixPerThread * (i+1))
67              # if endMatrix == startMatrix: break
68              if i == self.threadCount - 1: #
69                  threadMatrixes = allMatrixes[startMatrix:]
70              else:
71                  threadMatrixes = allMatrixes[startMatrix:endMatrix]
72              startMatrix = endMatrix
73              activeThreads.append(
74                  Process(target=calculateSumThreading, args=(threadMatrixes,
                        self.returnList,)))
75              threadTasksCount.append(len(threadMatrixes))
76
77          # print(f'threads - {len(activeThreads)}, tasks -', threadTasksCount
                )
78          for thread in activeThreads:
79              thread.start()
80          for thread in activeThreads:
81              thread.join()
82          self.returnList = self.threadManager.list()
83
84          return sum(self.returnList)
```

3.4 Тестирование

В таблице 3.1 представлены использованные для тестирования методом "черного ящика"данные, были рассмотрены все возможные тестовые случаи. Все тесты пройдены успешно.

3.5 Вывод

В данном разделе были реализованы и протестированы алгоритмы умножения матриц: классический, Копперсмита-Винограда и оптимизированный Копперсмита-Винограда.

Таблица 3.1: Проведенные тесты

Матрица 1	Строка 1	Ожидаемый результат
$\begin{pmatrix} 1 & 5 & 2 \\ 1 & 2 & 8 \\ 1 & 3 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 4 & 9 \\ 8 & 8 & 8 \\ 12 & 21 & 13 \end{pmatrix}$	$\begin{pmatrix} 65 & 86 & 75 \\ 113 & 188 & 129 \\ 49 & 70 & 59 \end{pmatrix}$
$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \end{pmatrix}$	$\begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 50 \\ 32 \end{pmatrix}$
$\begin{pmatrix} 12 \end{pmatrix}$	$\begin{pmatrix} 17 \end{pmatrix}$	$\begin{pmatrix} 204 \end{pmatrix}$
$\begin{pmatrix} -1 & -2 & 3 \\ 8 & -9 & 7 \\ -4 & -7 & 5 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & -9 \end{pmatrix}$	$\begin{pmatrix} 12 & 12 & -42 \\ 21 & 27 & -93 \\ 3 & -3 & -99 \end{pmatrix}$
$\begin{pmatrix} 8 & 7 \end{pmatrix}$	$\begin{pmatrix} 4 & 2 \end{pmatrix}$	Ошибка
$\begin{pmatrix} 1 & -1 & 2 \\ -4 & 7 & -5 \end{pmatrix}$	$\begin{pmatrix} 1 & 8 & 3 \\ 6 & -9 & 7 \\ 1 & 4 & 8 \end{pmatrix}$	$\begin{pmatrix} -3 & 25 & 12 \\ 33 & -115 & -3 \end{pmatrix}$
$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 1 & 8 & 6 \\ 3 & 5 & -4 \\ 6 & 6 & 6 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$
$\begin{pmatrix} 1 & 4 \\ 8 & -3 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 4 & 8 & -9 & 5 \\ -8 & 7 & 9 & 0 \\ 7 & 9 & -1 & 1 \\ 8 & 5 & 2 & 8 \end{pmatrix}$	$\begin{pmatrix} 4 & 8 & -9 & 5 \\ -8 & 7 & 9 & 0 \\ 7 & 9 & -1 & 1 \\ 8 & 5 & 2 & 8 \end{pmatrix}$
$\begin{pmatrix} 7 & 1 & 3 \\ -1 & -1 & -1 \\ 3 & 5 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 7 & 1 & 3 \\ -1 & -1 & -1 \\ 3 & 5 & 2 \end{pmatrix}$

4 Экспериментальная часть

В данном разделе сравниваются реализованные алгоритмы, дается сравнительная оценка затрат на память и время.

4.1 Пример работы программы

Пример работы программы представлен на рисунках 4.1-4.1.

Рис. 4.1: Ввод данных

Рис. 4.2: Результат работы программы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система — Windows [1] 10 64-bit;
- оперативная память — 16 Гб;
- процессор — Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz [2].

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов замерялось на автоматически генерируемых квадратных матрицах необходимого размера с использованием функции getrusage библиотеки resources. Усредненные результаты замеров процессорного времени приведены в таблице. Используемые обозначения: "Классич." - классический алгоритм умножения матриц, "Виноград" - алгоритм Копперсмита-Винограда, "Опт.Виноград" - оптимизированный алгоритм Копперсмита-Винограда.

Рис. 4.3: Сравнение времени работы алгоритмов умножения квадратных матриц четных размеров

Таблица 4.1: Время обработки строк разной длины в микросекундах

Размер	Классич.	Виноград	Опт.Виноград
10	7626	8432	6334
11	10114	11119	8247
30	198387	200491	144975
31	221763	221463	159228
50	914673	913305	651054
51	990919	975478	695120
70	2568877	2492211	1768562
71	2692695	2625663	1855013
90	5472335	5286995	3717559
91	5634068	5480780	3892148
110	10070443	9613919	6775491
111	10310690	9927475	7034633

Рис. 4.4: Сравнение времени работы алгоритмов умножения квадратных матриц нечетных размеров

4.4 Вывод

По результатам проведенных замеров видно, что оптимизированный алгоритм Копперсмита-Винограда работает в 1.2-1.5 раза быстрее классического алгоритма умножения матриц с увеличением этого коэффициента при увеличении размера матрицы. Важно заметить, что коэффициент незначительно меньше для матриц нечетных размеров, что говорит о более медленной работе оптимизированного алгоритма Винограда для таких матриц, в то время как классический алгоритм зависимости от четности размеров не имеет. Алгоритм Копперсмита-Винограда на четных матрицах меньше 50x50 работает медленнее классического алгоритма, на больших - чуть быстрее, однако разрыв составляет не более 1.05 раз. В случае с нечетным размером матрицы алгоритм Винограда на небольших матрицах проигрывает классическому алгоритму, но при увеличении размерности время работы обоих алгоритмов сопоставимо, с небольшим преимуществом алгоритма Винограда.

Заключение

В процессе выполнения лабораторной работы были изучены и реализованы классический алгоритм умножения и алгоритм Копперсмита-Винограда матриц, оптимизирован алгоритм Винограда.

Согласно проведенному анализу трудоемкости алгоритмов в соответствии с выбранной моделью вычислений, трудоемкость классического алгоритма составила приблизительно $11mnq$, алгоритма Копперсмита-Винограда - $16mnq$, оптимизированного алгоритма Копперсмита-Винограда - $9mnq$.

Было исследовано процессорное время выполнения выше обозначенных алгоритмов. В результате было выявлено, что на матрицах с количеством элементов в строках и столбцах, меньших 50, дольше всего работает алгоритм Копперсмита-Винограда, на больших - классический, причем время работы алгоритма Винограда незначительно меньше (разница не превышает 1.04 раз). Быстрее всего работает оптимизированный алгоритм Копперсмита-Винограда (в 1.2-1.5 раз быстрее других алгоритмов с увеличением разницы во времени работы с увеличением размеров матриц), однако заметна небольшая деградация времени работы для матриц с нечетным количеством строк и столбцов (как и у алгоритма Винограда-Копперсмита), тогда как классический алгоритм таким свойством не обладает.

Литература

[1] Сравните выпуски Windows 10 [Электронный ресурс]. Режим доступа: https://www.microsoft.com/ru-ru/windows/compare-windows-10-home-vs-pro. Дата обращения: 21.09.2021.

[2] Процессор Intel® Core™ i5-3550 [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/97150/intel-core-i57600-processor-6m-cache-up-to-4-10-ghz.html. Дата обращения: 21.09.2021.