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.

Содержание

Введение

Конвейер — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Сам термин «конвейер» пришёл из промышленности, где используется подобный принцип работы — материал автоматически подтягивается по ленте конвейера к рабочему, который осуществляет с ним необходимые действия, следующий за ним рабочий выполняет свои функции над получившейся заготовкой, следующий делает ещё что-то. Таким образом, к концу конвейера цепочка рабочих полностью выполняет все поставленные задачи, сохраняя высокий темп производства. Например, если на самую медленную операцию затрачивается одна минута, то каждая деталь будет сходить с конвейера через одну минуту. В процессорах роль рабочих исполняют функциональные модули, входящие в состав процессора.

Цель лабораторной работы

Целью данной работы является реализация асинхронного взаимодействия потоков на примере конвейерной обработки данных.

Задачи лабораторной работы

В рамках выполнения работы необходимо выполнить следующие задачи:

- Изучить асинхронное взаимодействие на примере конвейерной обработки данных.
- Привести схему конвейерных вычислений.
- Описать используемые структуры данных.
- Определить средства программной реализации.
- Реализовать и протестировать ПО.
- Провести сравнительный последовательной и конвейерной реализации по затрачиваемым ресурсам (времени работы).
- Изучить время, затрачиваемое на нахождение заявки в очереди к каждому этапу конвейера.

1 Аналитическая часть

В данном разделе рассматриваются принципы и идея конвейерной обработки данных, а также приводится описание решаемой задачи и выделенных стадий конвейерной обработки.

1.1 Описание конвейерной обработки данных

Конвейер[?] — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Идея 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.

Многие современные процессоры управляются тактовым генератором. Процессор внутри состоит из логических элементов и ячеек памяти — триггеров. Когда приходит сигнал от тактового генератора, триггеры приобретают своё новое значение, и «логике» требуется некоторое время для декодирования новых значений. Затем приходит следующий сигнал от тактового генератора, триггеры принимают новые значения, и так далее. Разбивая последовательности логических элементов на более короткие и помещая триггеры между этими короткими последовательностями, уменьшают время, необходимое логике для обработки сигналов. В этом случае длительность одного такта процессора может быть соответственно уменьшена.

1.2 Выделенные стадии конвейерной обработки

В данной работе в качестве алгоритма, реализованного для конвейеризации, используется шифрование паролей с сохранением в базу данных. Таким образом, было выделено три ленты конвейера:

1. Модификация исходных данных выбранным способом для обеспечения невозможности расшифровки без информации об используемом методе.

2. Двойное последовательное хеширование полученной строки для большей безопасности.

3. Загрузка полученного значение в базу данных.

1.3 Требования к программному обеспечению

На основе приведенного алгоритма можно выдвинуть требования к разрабатываемому ПО:

- выходные данные - время работы последовательного выполнения действий, время работы конвейера, минимальное, среднее, максимальное времена ожидания заявки в очереди каждой из конвейерных лент - вещественные числа;

- наличие обработки некорректного ввода.

1.4 Вывод

Был рассмотрен алгоритм нахождения определителя квадратной матрицы размера $n \times n$, он независимо вычисляет слагаемые для нахождения итогового определителя, что дает возможность для реализации параллельного варианта алгоритма. Выдвинуты требования к разрабатываемому ПО: работа с квадратными матрицами произвольного размера (целое число) с возможностью ввода элементов вещественного типа пользователем или генерацией матрицы заданного размера, на выходе - вещественное число, соответствующее вычисленному определителю матрицы, обработка некорректного ввода.

2 Конструкторская часть

Данный раздел содержит схемы конвейерной обработки данных, последовательного и конвейерного алгоритма.

2.1 Схемы алгоритмов

В данном пункте раздела представлены схемы реализуемых в работе алгоритмов.

На рисунке 2.1 представлена схема организации конвейерных вычислений на примере конвейера с тремя лентами.

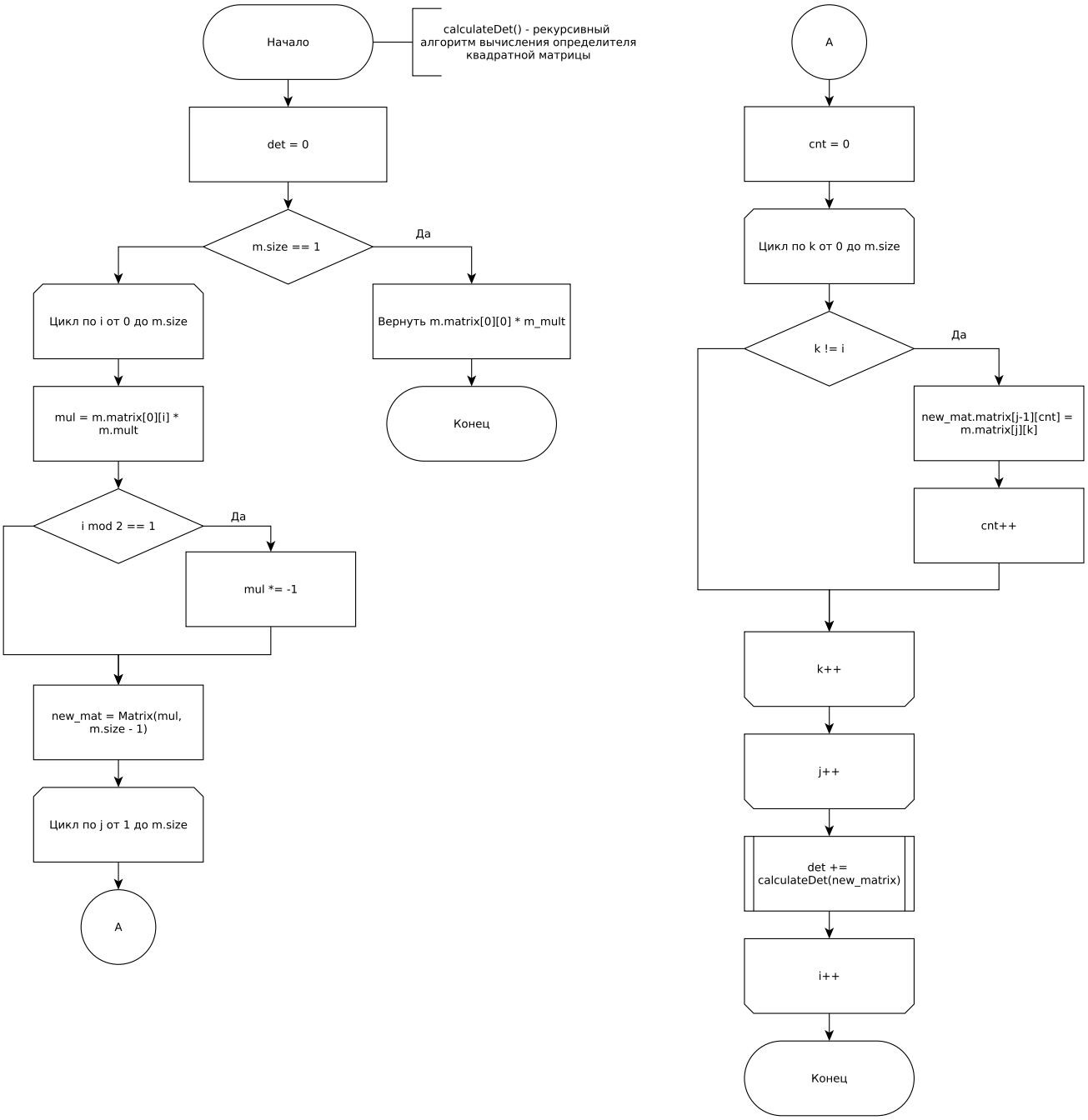

Рис. 2.1: Схема организации конвейера с тремя лентами

На рисунке 2.2 представлена схема последовательного алгоритма шифрования и сохранения паролей.

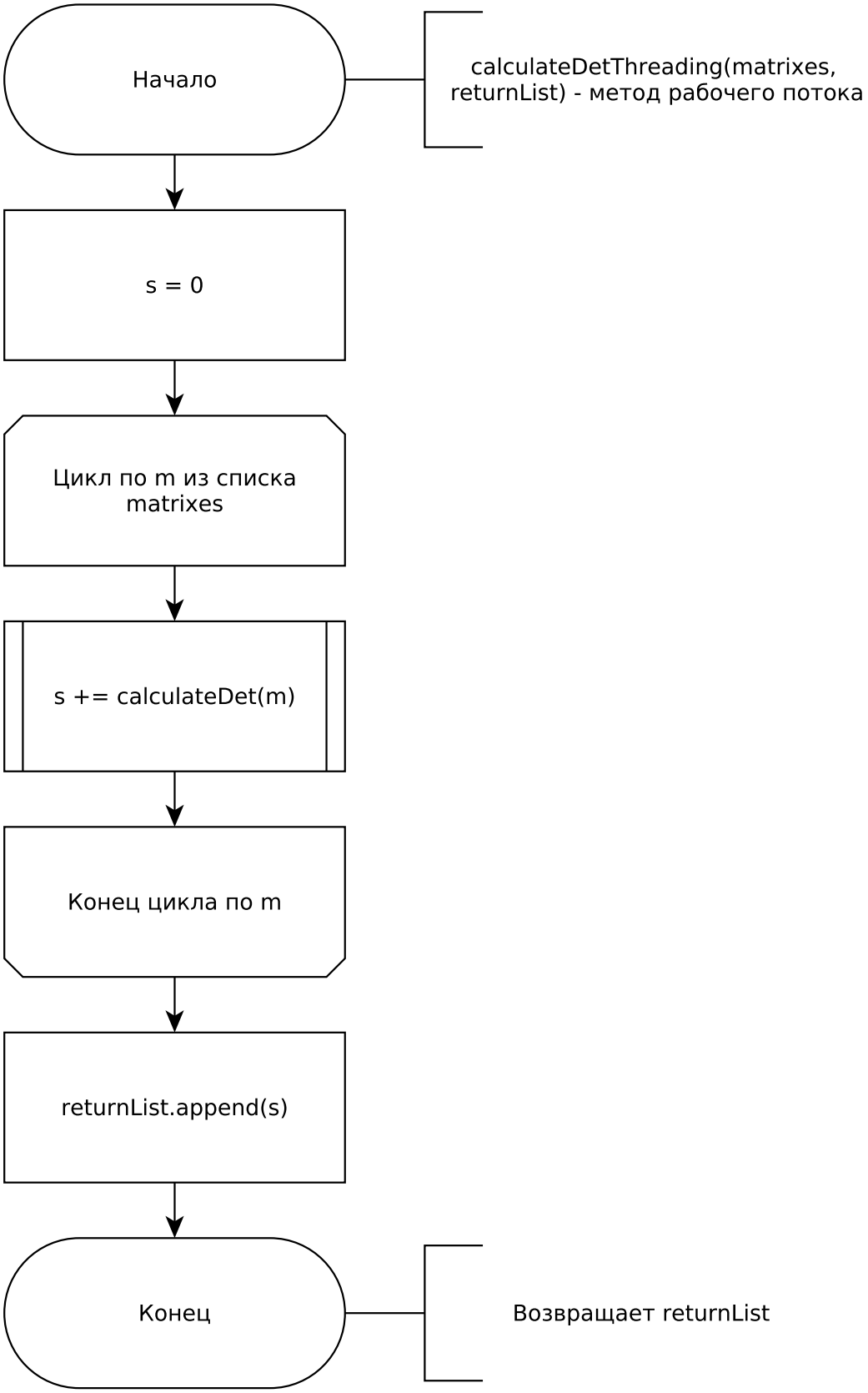

Рис. 2.2: Схема последовательного алгоритма

На рисунках 2.3 - ?? представлена схема реализация алгоритма шифрования на конвейере с тремя лентами.

Рис. 2.3: Схема алгоритма работы конвейерной обработки

2.2 Структуры данных

Для удобства работы были выделены следующие классы:

- UserStats со следующими полями:
  - login - строка, логин пользователя
- mul - множитель минора (целое число, 1 для четных столбцов, -1 для нечетных с учетом начала нумерации с нуля);
- size - размер матрицы (целое число).

Для возможности генерации матриц, заполненных случайными элементами, класс Matrix содержит метод randomize.

Для управления потоками выделен класс Dispatcher со следующими полями:

- m - матрица вещественных чисел, определитель которой необходимо вычислить;
- size - размер матрицы (целое число);
- threadCount - количество создаваемых потоков;
- returnList - список, разделяемый потоками, содержащий промежуточные вычисления определителей.

2.2.1 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.

2.3 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 Средства реализации

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

- удобные средства для работы с потоками;
- обладает информативной документацией.

3.2 Реализация алгоритмов

В листингах 3.1 - 3.2 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 класс Solver и функция calculateSumThreading

```
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          for thread in activeThreads:
78              thread.start()
79          for thread in activeThreads:
80              thread.join()
81          self.returnList = self.threadManager.list()
82
83          return sum(self.returnList)
```

3.3 Тестирование

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

Таблица 3.1: Проведенные тесты

Матрица	Определитель
$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$	0
$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	1
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 12 \end{pmatrix}$	-9
$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$	0

3.4 Вывод

В данном разделе были реализованы и протестированы алгоритмы нахождения определителя матрицы: обычный и параллельный.

4 Экспериментальная часть

В данном разделе сравниваются реализованные алгоритмы, дается сравнительная оценка затрат по времени.

4.1 Пример работы программы

Пример работы программы представлен на рисунках.

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

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

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов замерялось на автоматически генерируемых квадратных матрицах необходимого размера с использованием функции getrusage библиотеки resources.

Таблица 4.1: Время нахождения определителя матрицы при использовании разного количества потоков в микросекундах

Размерность	1 п.	2 п.	4 п.	8 п.	16 п.	32 п.
4	10	14	17	23	38	67
5	10	14	17	23	38	68
6	12	15	17	24	39	68
7	22	19	21	25	40	72
8	75	47	37	42	57	82
9	529	306	304	164	155	182

На рисунке 4.1 графически изображена зависимость времени работы алгоритма от размерности матрицы и количества потоков.

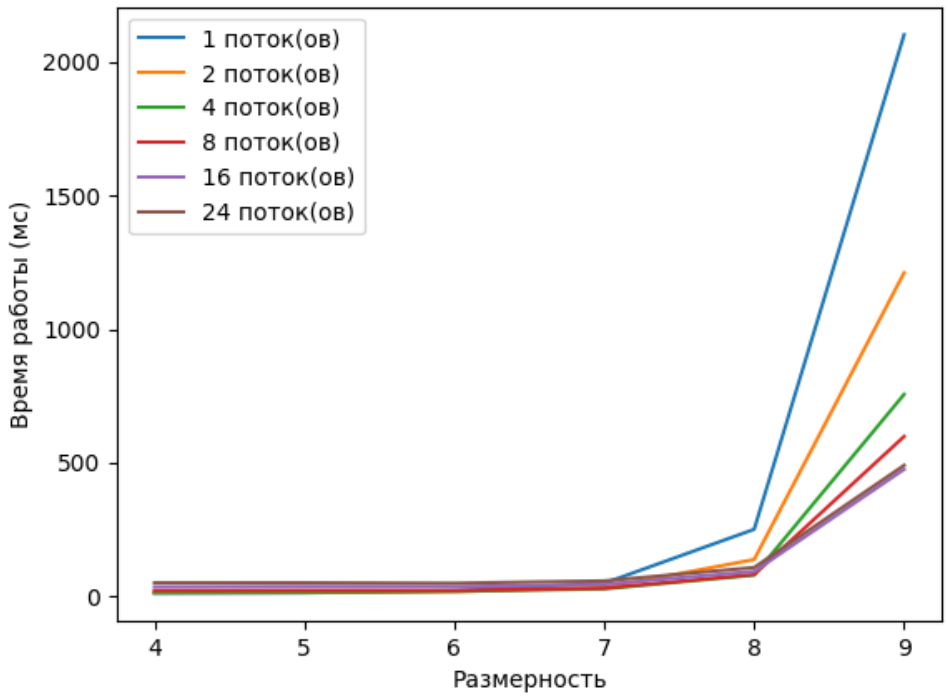

Рис. 4.1: Зависимость времени работы алгоритма от размерности матрицы и количества потоков

Поскольку процессор на устройстве содержит 4 физических и логических ядра, эффективность алгоритма улучшается только при создании 4 и менее потоков. Как видно по графику 4.1, с 2 потоками алгоритм выполнялся практически в 2 раза быстрее, чем с 1 потоком. Аналогичная ситуация наблюдается при сравнении 4 и 2 потоков.

4.4 Вывод

В данном разделе были проведены измерения времени, требуемого на исполнение алгоритма. По результатам измерений видно, что с увеличением количества потоков скорость выполнения алгоритма увеличивается. Однако как только количество потоков становится равно количеству логических ядер процессора, временные затраты на выполнение алгоритма увеличиваются из-за времени, затрачиваемого на создание новых потоков.

Заключение

В процессе выполнения лабораторной работы были изучены и реализованы последовательный и параллельный рекурсивные алгоритмы нахождения определителя матрицы.

Было экспериментально вычислено реальное время выполнения выше обозначенных алгоритмов. В результате было выявлено, что распараллеливание алгоритма нахождения определителя матрицы позволяет увеличить скорость его выполнения при использовании количества потоков меньшего или равного количеству логических ядер процессора.

Литература