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 (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

Отчет по лабораторной работе №4 по дисциплине анализ алгоритмов

Тема: Многопоточные вычисления

Студент: Серова М.Н.

Группа: ИУ7-55Б

Оценка (баллы): _____

Преподаватель: Волкова Л.Л.

Москва, 2021

Содержание

Введение

Цель: сравнить производительность последовательной и многопоточной версий алгоритма вычисления определителя матрицы.

Задачи:

1. Изучить алгоритм вычисления определителя квадратной матрицы.
2. Реализовать и протестировать:
   (a) последовательный алгоритм нахождения определителя матрицы;
   (b) многопоточный алгоритм нахождения определителя матрицы.
3. Провести сравнительный анализ алгоритмов по затрачиваемым ресурсам (процессорному времени работы).
4. Определить, всегда ли при задании количества потоков, равному удвоенному количеству логических ядер процессора, достигается наибольшая эффективность.

1 Аналитическая часть

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.

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

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

1.1 Нахождение определителя матрицы

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

1.1.1 Матрица 1×1

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

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

1.1.2 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 $(0, 0), (a, b), (a + c, b + d), (c, d)$.

1.1.3 Матрица 3×3

Определитель матрицы 3×3 можно вычислить по формуле:

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

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

1.1.4 Матрица $n \times n$

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

$$\Delta = \sum_{j=1}^{n} (-1)^{1+j} \cdot a_{1j} \cdot M_j^{-1}, \text{ где} M_j^{-1} - a_{ij} \tag{1.4}$$

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.

1.2 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.

2 Конструкторская часть

Данный раздел содержит требования к разрабатываемому ПО, схемы реализуемых в работе алгоритмов (стандартного рекурсивного алгоритма вычисления определителя матриц и алгоритма с использованием потоков в виде схем потока-диспетчера и рабочего потока) и теоретический расчет повышения эффективности исполнения алгоритма по времени.

2.1 Требования к программному обеспечению

Требования, выдвигаемые к разрабатываемому ПО:

- входные данные - размер матрицы (целое число), её элементы (вещественные числа);
- выходные данные - определитель матрицы (вещественное число).

2.2 Схемы алгоритмов

В данном пункте раздела представлены схемы реализуемых в работе алгоритмов.

На рисунке 2.1 представлена схема рекурсивного алгоритм нахождения определителя.

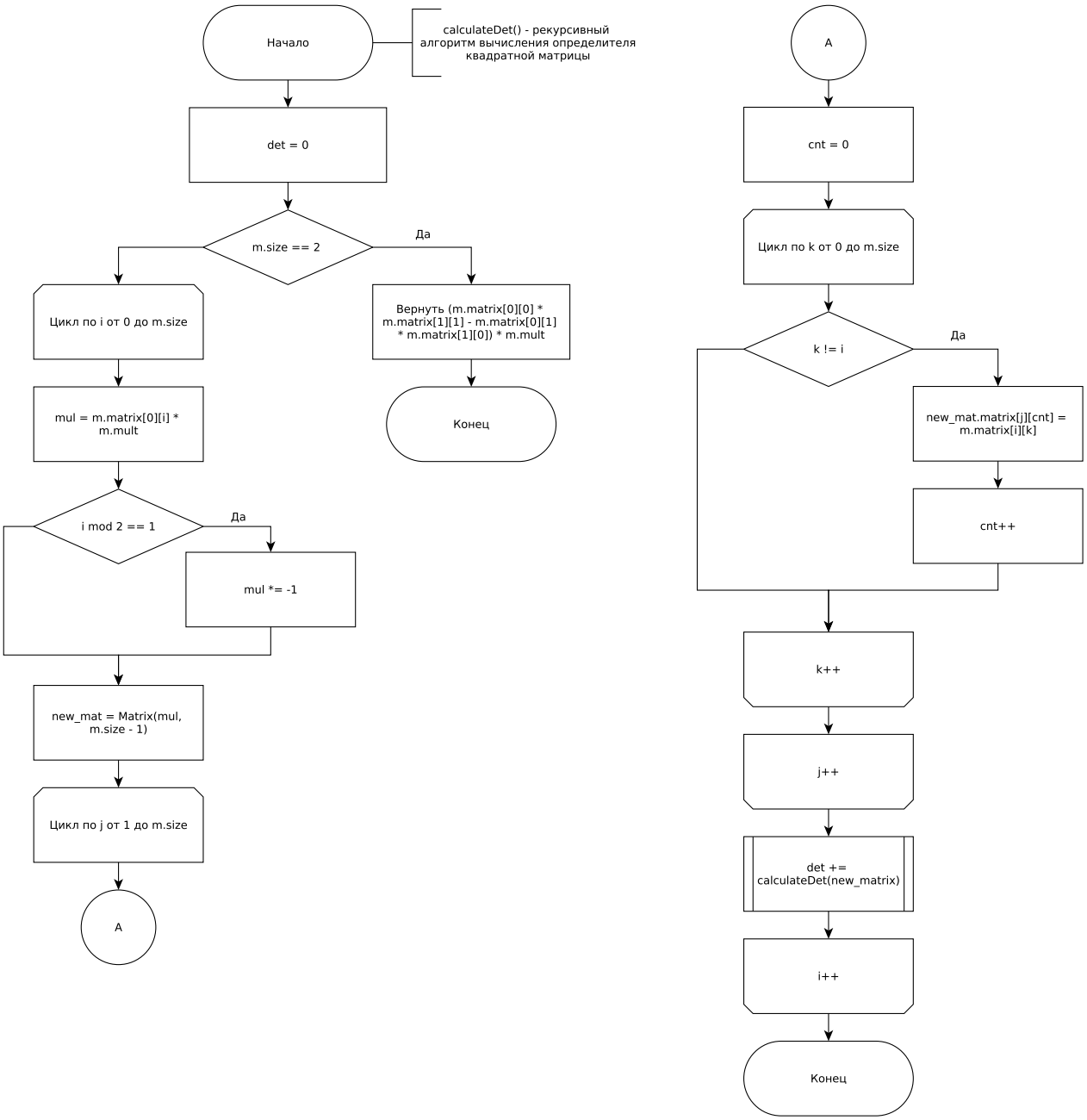

Рис. 2.1: Схема рекурсивного алгоритма нахождения определителя

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

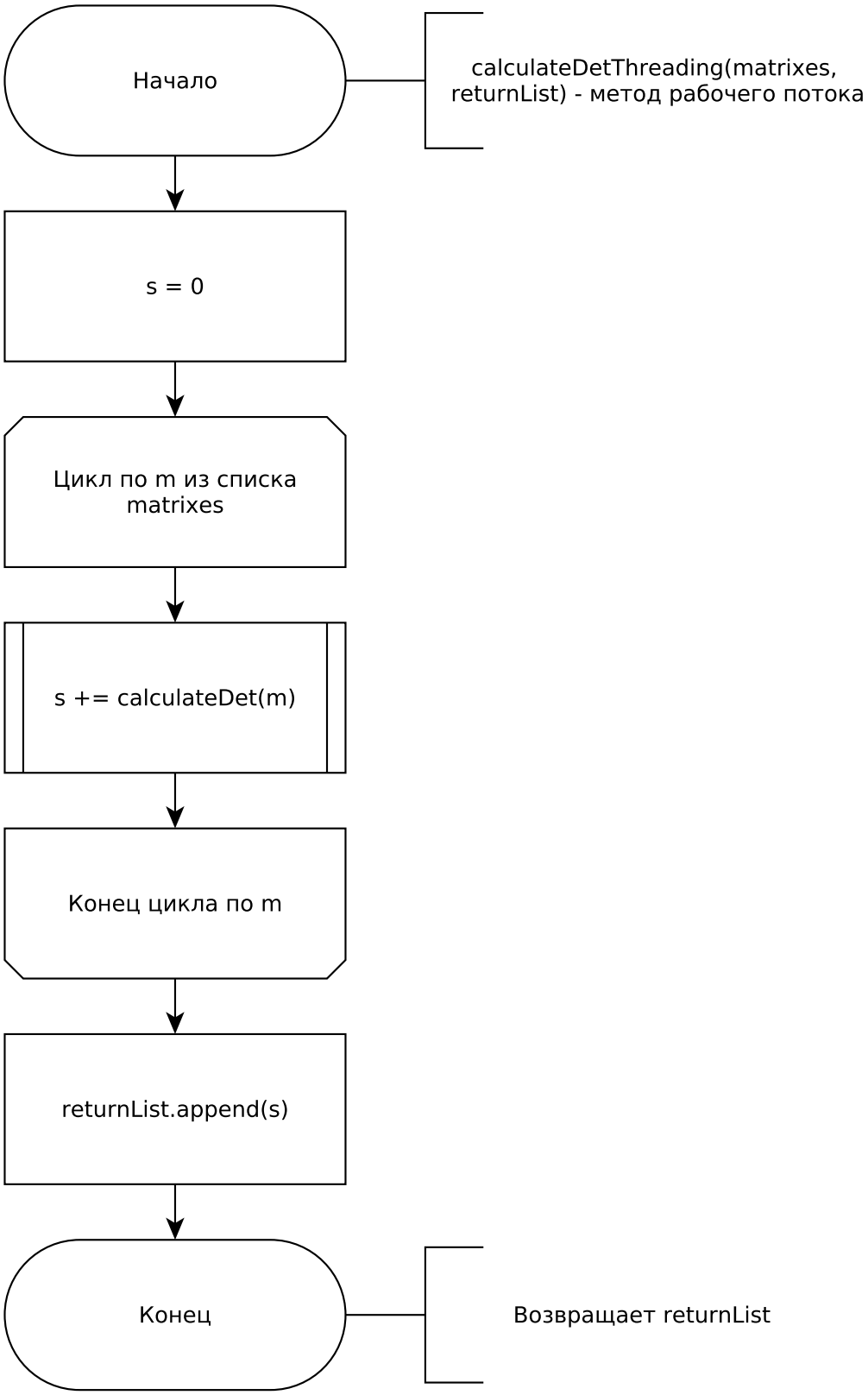

Рис. 2.2: Схема рекурсивного алгоритм нахождения определителя

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

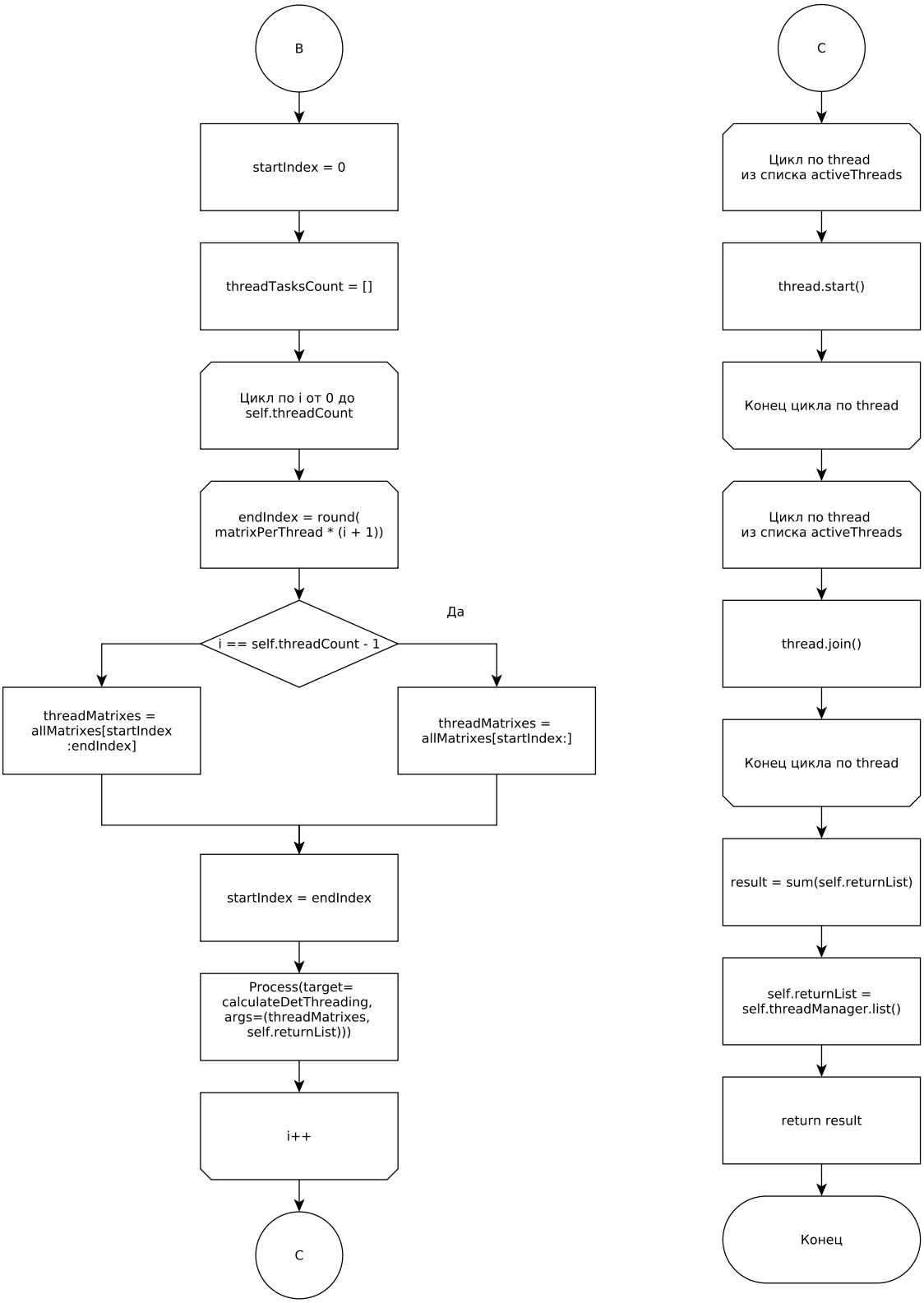

Рис. 2.4: Схема рекурсивного алгоритм нахождения определителя

2.2.1 Теоретический расчет 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, в том числе, разделение вычисления определителя на потоки.

Согласно проведенному теоретическому расчету эффективности по времени, эффективность версии алгоритма с потоками должна повышаться примерно в k раз, где k ($1 < k \leq n/2$ или $k = n$) - количество потоков. Если k $n/2 < k < n$, время исполнения алгоритма не увеличится более чем в $n/2$ раза.

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:

- имеется большой опыт разработки;
- имеет большое количество расширений и библиотек, в том числе библиотеку для работы с потоками, измерения времени, построения графиков;
- обладает информативной документацией;

3.2 Реализация алгоритмов

В листингах 3.1 - 3.3 представлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Рекурсивный алгоритм вычисления определителя матрицы

```
def calculateDet(m: Matrix):
  s = 0
  if m.size == 2:
    s = (m.matrix[0][0] * m.matrix[1][1] - m.matrix[0][1] * m.
        matrix[1][0]) * m.mult
  else:
    for i in range(m.size):
      mul = m.matrix[0][i] * m.mult
      if i % 2 == 1:
        mul *= -1
      size = m.size - 1
      matrix = []
      for j in range(1, m.size):
        matrix.append([])
        for k in range(m.size):
          if k != i:
            matrix[-1].append(m.matrix[j][k])
      s += calculateDet(Matrix(size, mul, matrix))
  return s
```

Листинг 3.2: Поток-диспетчер

```
class Solver:
  def __init__(self, size: int = 0, matrix: Matrix = None):
    self.m = matrix
    if size == 0:
      size = 9
    if self.m is None:
      self.m = Matrix(size).randomize()
    self.threadCount = 1
    self.threadManager = Manager()
    self.returnList = self.threadManager.list()

  def solve(self):
    activeThreads = []
    matrixPerThread = self.m.size / self.threadCount
    allMatrixes = []
    for i in range(self.m.size):
      mul = self.m.matrix[0][i] * self.m.mult
      if i % 2 == 1:
        mul *= -1
```

```
20          size = self.m.size - 1
21          matrix = []
22          for j in range(1, self.m.size):
23            matrix.append([])
24            for k in range(self.m.size):
25              if k != i:
26                matrix[-1].append(self.m.matrix[j][k])
27          allMatrixes.append(Matrix(size, mul, matrix))
28        startIndex = 0
29        threadTasksCount = []
30        for i in range(self.threadCount):
31          endIndex = round(matrixPerThread * (i + 1))
32          if i == self.threadCount - 1:  # last thread
33            threadMatrixes = allMatrixes[startIndex:]
34          else:
35            threadMatrixes = allMatrixes[startIndex:endIndex]
36          startIndex = endIndex
37          activeThreads.append(
38          Process(target=calculateDetThreading, args=(threadMatrixes,
               self.returnList,)))
39          threadTasksCount.append(len(threadMatrixes))
40        for thread in activeThreads:
41          thread.start()
42        for thread in activeThreads:
43          thread.join()
44        result = sum(self.returnList)
45        self.returnList = self.threadManager.list()
46        return result
```

Листинг 3.3: Рабочий поток

```
def calculateDetThreading(matrixes, returnList: list):
  s = 0
  for m in matrixes:
    s += calculateDet(m)
  returnList.append(s)
```

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 Выводы

В данном разделе были реализованы и протестированы алгоритмы рекурсивного вычисления определителя и вычисления определителя с использованием многопоточности.

4 Экспериментальная часть

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

4.1 Пример работы программы

Пример работы программы представлен на рисунках 4.1 - 4.2.

```
Введите любой символ для генерации матрицы заданного размера, нажмите Enter для
ввода матрицы:

Введите размеры квадратной матрицы (целое число): 9
Введите матрицу:
1 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 1
Рекурсивно вычисленный определитель 1.0
Без потоков, время выполнения = 0.4965999126434326
Количество потоков: 1, время выполнения = 0.5147125720977783, результат = 1.0
Количество потоков: 2, время выполнения = 0.2974381446838379, результат = 1.0
Количество потоков: 4, время выполнения = 0.18285012245178223, результат = 1.0
Количество потоков: 8, время выполнения = 0.14126157760620117, результат = 1.0
Количество потоков: 16, время выполнения = 0.14821529388427734, результат = 1.0
Количество потоков: 32, время выполнения = 0.1662731170654297, результат = 1.0
```

Рис. 4.1: Пример работы программы для вводимой матрицы

```
Введите любой символ для генерации матрицы заданного размера, нажмите Enter для ввода матрицы:
d
Введите размер матрицы для автоматической генерации: 9
Generated matrix  [[290, -462, -36, -561, -312, -123, -696, -148, -973], [-700, -749, -439, 692, -626, 864, 58
6, 992, 903], [-706, 716, -476, -341, 381, -291, -451, 23, -706], [961, -436, 443, -191, -280, -228, 745, -528
, 287], [-702, -553, -617, -32, -4, 537, -867, -606, -709], [146, -242, -468, -738, -298, 962, -817, 917, -874
], [864, -938, -886, -60, 591, 940, 517, -131, 949], [-737, 363, -934, 206, -635, 759, 485, 374, -24], [303, 3
67, 907, 457, -557, -349, -634, -376, 290]]
Рекурсивно вычисленный определитель 954348099616103725872662116
Без потоков, время выполнения = 0.8421685695648193
Количество потоков: 1, время выполнения = 0.8674774169921875, результат = 954348099616103725872662116
Количество потоков: 2, время выполнения = 0.4965970516204834, результат = 954348099616103725872662116
Количество потоков: 4, время выполнения = 0.3104851245880127, результат = 954348099616103725872662116
Количество потоков: 8, время выполнения = 0.21962666511535645, результат = 954348099616103725872662116
Количество потоков: 16, время выполнения = 0.22811508178710938, результат = 954348099616103725872662116
Количество потоков: 32, время выполнения = 0.27892494201660156, результат = 954348099616103725872662116
```

Рис. 4.2: Пример работы программы для вводимой матрицы

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- операционная система: Ubuntu 20.01 Linux x86_64 [**?**];
- оперативная память: 8 Гб;
- процессор: AMD Ryzen5 3500U [**?**]:
  - количество физических ядер: 4;
  - количество логических ядер: 8.

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов измерялось на автоматически генерируемых квадратных матрицах необходимого размера (элементы которых - вещественные числа в диапазоне [-1000, 1000]) с использованием функции time библиотеки time. Усредненные результаты 10 замеров реального времени работы приведены в таблице ниже.

На рисунке **??** представлен график зависимости времени работы алгоритмов от размера произвольно упорядоченных массивов на основе таблицы 4.1. Для произвольно упорядоченных массивов самым эффективным алгоритмом оказался алгоритм Шелла. Выигрыш сортировки Шелла по сравнению с алгоритмом сортировки выбором составляет от 1.5 до 16 раз; по сравнению с шейкерной сортировкой - от 2 до 30 раз. Выигрыш по времени увеличивается с увеличением размера входной последовательности. Наименее эффективной оказалась шейкерная сортировка, которая проигрывает сортировке выбором в среднем в 2 раза.

Таблица 4.1: Время обработки произвольных массивов разной длины в микросекундах

Размер	1 поток	2 потока	4 потока	8 потоков	16 потоков	32 потока
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

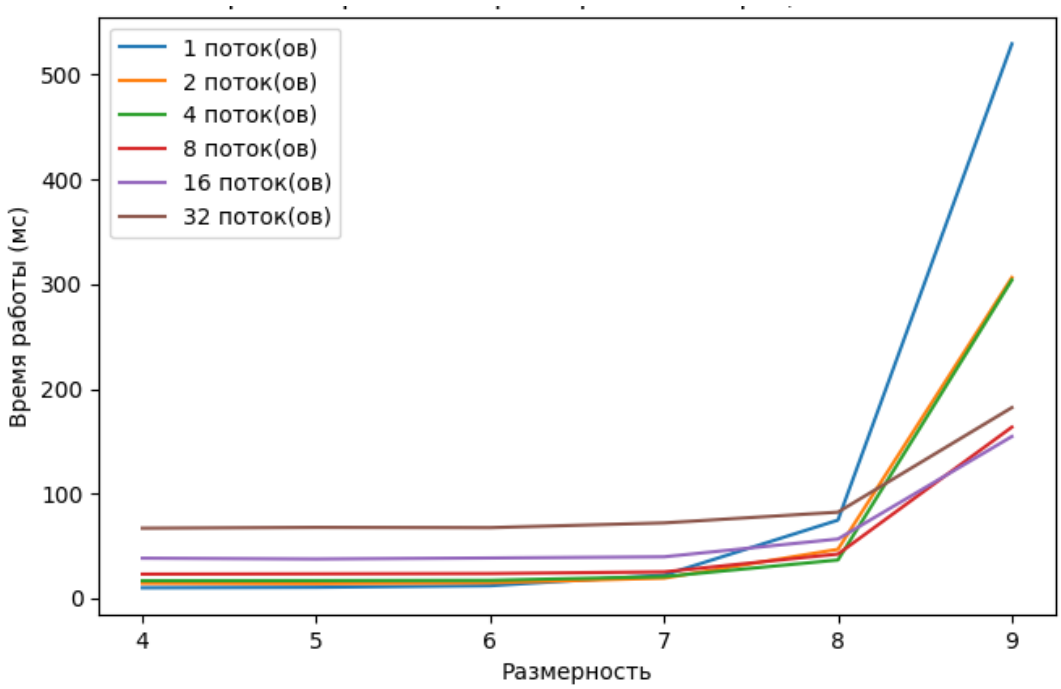

Рис. 4.3: Зависимость времени работы алгоритмов от размера квадратной матрицы

4.4 Выводы

По результатам проведенных замеров видно, что самым быстрым является алгоритм сортировки Шелла, причем его преимущество по сравнению с сортировкой выбором составил от 1.5 до 16 раз для произвольно упорядоченных массивов, от 2 до 56 раз для отсортированных массивов и от 2 до 22 раз для отсортированных в обратном порядке; по сравнению с сортировкой перемешиванием преимущество в среднем составило от 2 до 40 раз для

различно упорядоченных массивов. Время работы алгоритмов шейкерной сортировки и сортировки выбором для лучшего случая - упорядоченного массива - практически совпало с незначительным преимуществом сортировки выбором. Наименее эффективным оказался алгоритм шейкерной сортировки, проигрыш которого по сравнению с сортировкой выбором составил 2-3 раза. Худшим случаем сортировки Шелла оказался произвольно упорядоченный массив, что говорит о неподходящем выбранном шаге сравнения элементов.

Заключение

В процессе выполнения лабораторной работы были изучены и реализованы алгоритмы сортировки перемешиванием, выбором и сортировки Шелла.

Согласно проведенному анализу трудоемкости алгоритмов в соответствии с выбранной моделью вычислений, приблизительная трудоемкость шейкерной сортировки для лучшего случая равна $3n^2$, худшего - $\frac{15}{2}n^2$; сортировки выбором для лучшего случая - $\frac{5}{2}n^2$, для худшего - $3n^2$. Согласно проанализированным источникам, трудоемкость сортировки Шелла для лучшего случая равна $n \log n$, для худшего случая - n^2. Лучший случай для выбранных алгоритмов оказался общим, это случай обработки отсортированного массива. Худший случай совпал для шейкерной сортировки и сортировки выбором, это обработка отсортированного в обратном порядке массива. Худшим случаем для алгоритма сортировки Шелла является случай неудачного выбора шага сравнения. Таким образом, наиболее эффективной является сортировка Шелла, наименее - шейкерная сортировка.

Было исследовано процессорное время выполнения выше обозначенных алгоритмов. В результате было выявлено, что самым быстрым является алгоритм сортировки Шелла, причем его выигрыш по сравнению с сортировкой выбором составил от 1.5 до 16 раз для произвольно упорядоченных массивов, от 2 до 56 раз для отсортированных массивов и от 2 до 24 раз для отсортированных в обратном порядке; по сравнению с сортировкой перемешиванием преимущество составило от 2 до 30 раз для произвольно упорядоченных, от 2 до 56 раз для упорядоченных и от 2 до 60 раз для упорядоченных в обратном порядке массивов. Преимущество по сравнению с обоими алгоритмами увеличивается с увеличением размера массива. Время работы алгоритмов шейкерной сортировки и сортировки выбором для лучшего случая - упорядоченного массива - практически совпало с незначительным преимуществом сортировки выбором. Наименее эффективным оказался алгоритм шейкерной сортировки, проигрыш которого по сравнению с сортировкой выбором составил 2-3 раза. Худшим случаем сортировки Шелла оказался произвольно упорядоченный массив, что говорит о неподходящем выбранном шаге сравнения элементов.

Таким образом, практика подтверждает теорию, и самым эффективным

является алгоритм сортировки Шелла, наименее эффективным - шейкерной сортировки.

Список литературы