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

Многопоточность — 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 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 многопоточности:

- 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 многопоточности:

- несколько потоков могут вмешиваться друг в друга при совместном использовании аппаратных ресурсов [1];
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения [2];
- 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 иметь худшую 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 алгоритмы нахождения определителя матрицы;
- вычислить и сравнить трудоемкость реализуемых алгоритмов на основе выбранной модели вычислений;
- изучить понятие параллельных вычислений;
- реализовать последовательную и параллельную реализацию алгоритма нахождения определителя матрицы;
- сравнить временные характеристики реализованных алгоритмов экспериментально;
- дать оценку временных затрат алгоритмов на основе результатов проведенных экспериментов.

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:

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

$$a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} \quad (1.3)$$

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:

$$\Delta = \sum_{j=1}^{n} (-1)^{1+j} \cdot a_{1j} \cdot M_j^{-1}, \text{ где} M_j^{-1} - \text{дополнительный минор к элементу} a_{ij} \quad (1.4)$$

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.

Выдвинуты требования к разрабатываемому ПО:

- предусмотреть возможность задания произвольного размера квадратной матрицы (целое число);
- предусмотреть возможность ввода элементов вещественного типа;
- предусмотреть возможность генерации матрицы заданного размера со случайными элементами;
- вывод в формате вещественного числа, равного определителю заданной пользователем матрицы.

2 Конструкторская часть

Данный раздел содержит требования к разрабатываемому ПО, схемы алгоритмов, реализуемых в работе (Стандартный рекурсивный алгоритм нахождения определителя, алгоритм с использованием потоков и распределение слагаемых между потоками) 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), ее элементы (вещественные числа);
- выходные данные - определитель матрицы (вещественное число).

2.2 Схемы алгоритмов

В данном пункте раздела представлены схемы реализуемых в работе алгоритмов.

2.2.1 Схема классического алгоритма

На рисунке 2.1 представлена схема рекурсивного алгоритма нахождения определителя.

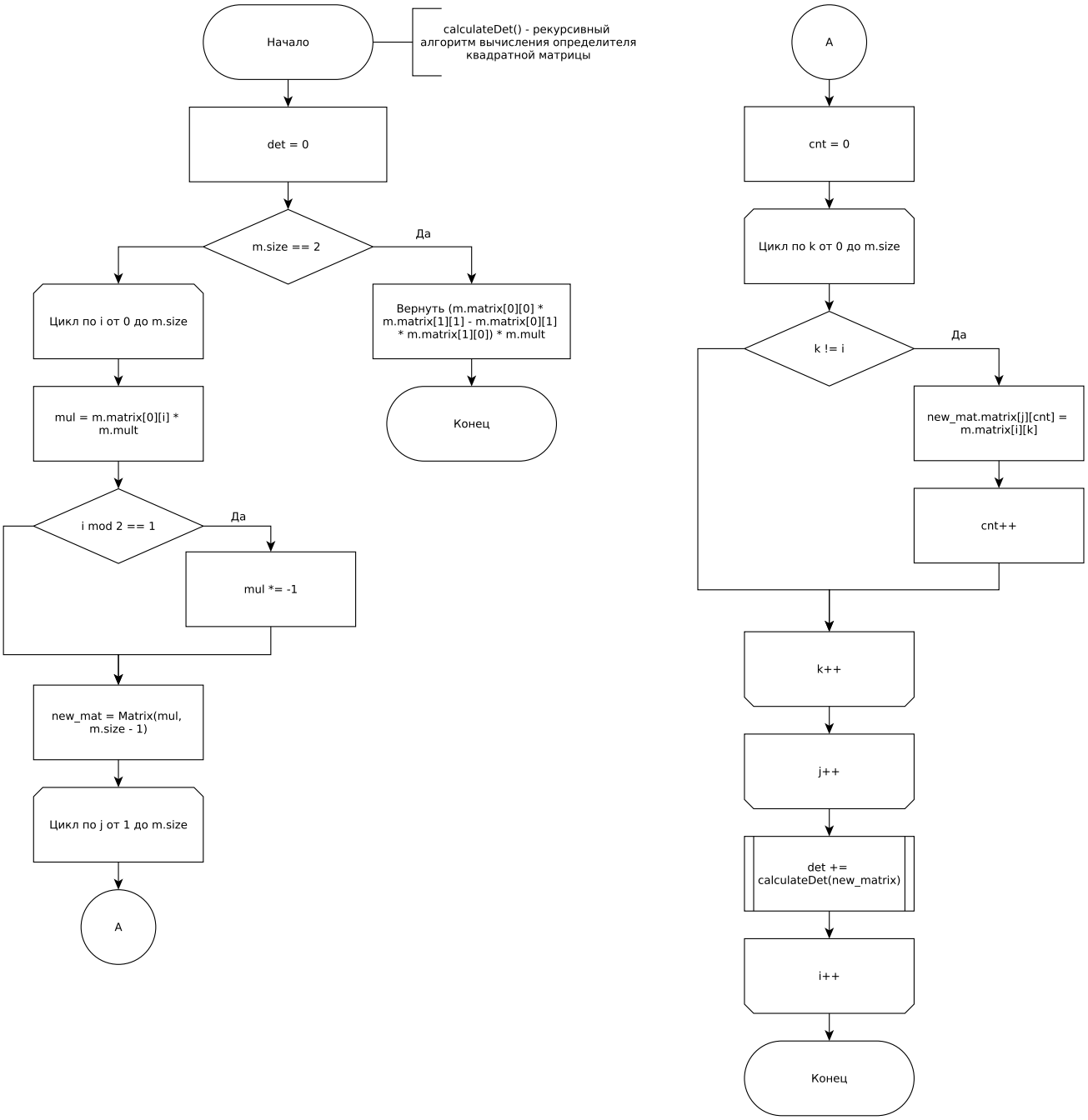

Рис. 2.1: Схема рекурсивного алгоритма нахождения определителя

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

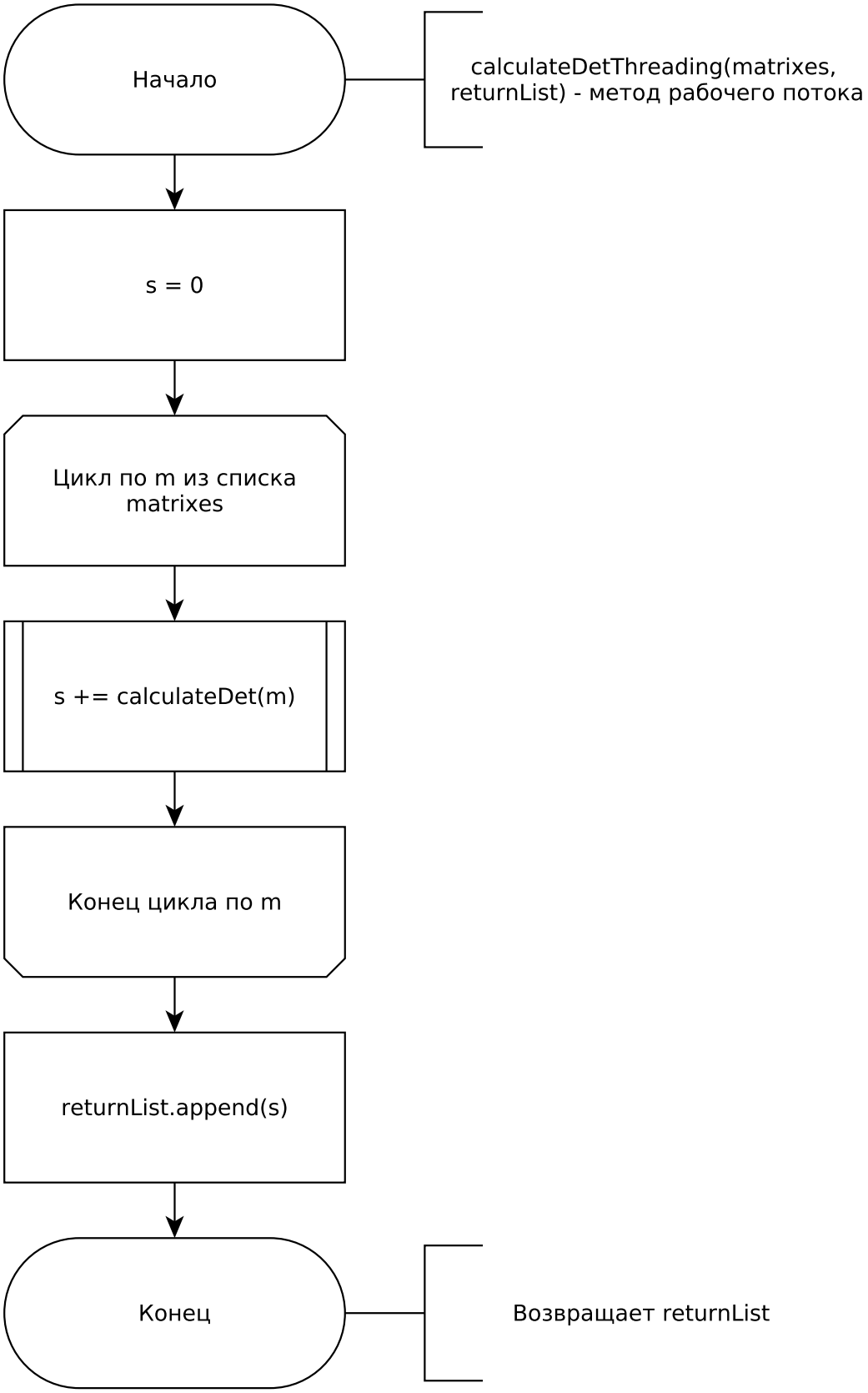

Рис. 2.2: Схема рекурсивного алгоритм нахождения определителя

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

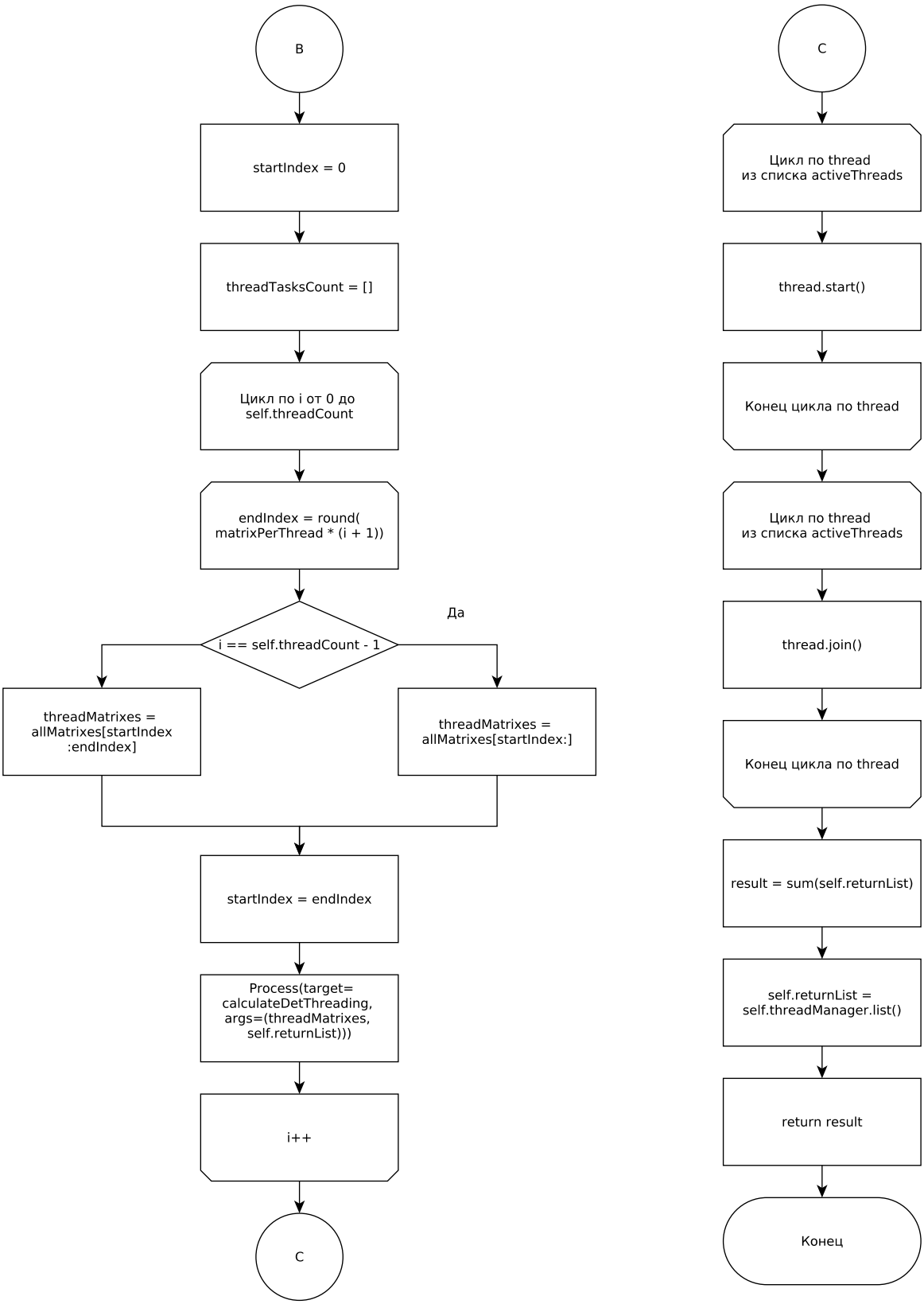

Рис. 2.4: Схема рекурсивного алгоритм нахождения определителя

2.3 Структуры данных

Для удобства работы с матрицами было решено создать класс Matrix со следующими полями:

- float[ ][ ] matrix - матрица вещественных чисел;
- int mul - множитель минора (целое число, 1 для четных столбцов, -1 для нечетных с учетом начала нумерации с нуля);
- int 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:

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

Таким образом, программа получает входные данные от пользователя (размер матрицы и её элементы) Затем начинается вычисление определителя для разного количества потоков, с целью сравнение времени и корректности вычислений. Для деления на потоки исходная матриц и требуемое количество потоков, передается в конструктор класса Dispatcher. Затем для каждого количества потоков вызывается метод dispatch, где происходит выделение "подматриц"потокам и запуск вычислительного процесса. Функция рабочего потока выполняет рекурсивное вычисление определителя для каждой переданной ему матрицы после чего вычисленное значение помещает в общий список. Сумма чисел в списке вычисляется в главном потоке по завершении работы всех дочерних потоков и возвращается как итоговый результат.

2.4 Структура ПО

На рисунке 2.5 изображена uml-диаграмма описанных в предыдущем пункте классов.

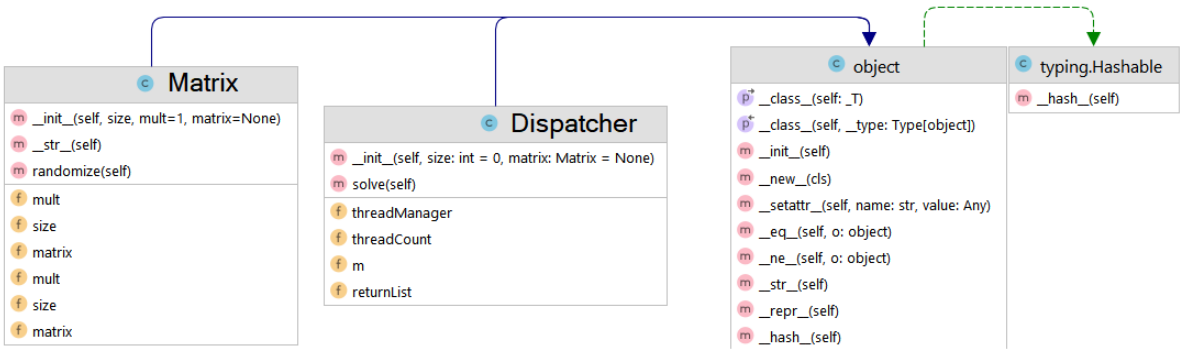

Рис. 2.5: Диаграмма классов в ПО

2.5 Классы эквивалентности

Для осуществления функционального тестирования ПО были выделены следующие классы эквивалентности:

- матрица, состоящая из одного элемента;
- нулевая матрица;
- единичная матрица;
- произвольная матрица, определитель которой равен нулю;
- произвольная матрица, определитель которой не равен нулю.

2.5.1 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.

2.6 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 [3]. 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 4.1-4.2.

```
D:\IdeaProjects\bmstu_aa\lab4\venv\Scripts\python.exe D:/IdeaProjects/bmstu_aa/lab4/src/main.py
Type anything for input size, press enter for input matrix:
5
Input matrix size: 5
Generated matrix  [[620, 274, 604, 503, 356], [528, -929, 918, 152, -211], [-860, -816, 277, 897, 688],
True det -413308567100185
No threads, time = 0.0
Threads 1, time = 0.4542369842529297, result = -413308567100185
Threads 2, time = 0.5055418014526367, result = -413308567100185
Threads 4, time = 0.5780351161956787, result = -413308567100185
Threads 8, time = 0.9136068820953369, result = -413308567100185
Threads 16, time = 1.792574405670166, result = -413308567100185
Threads 32, time = 3.551326036453247, result = -413308567100185

Process finished with exit code 0
```

Рис. 4.1: Пользовательский ввод матрицы

```
D:\IdeaProjects\bmstu_aa\lab4\venv\Scripts\python.exe D:/IdeaProjects/bmstu_aa/lab4/src/main.py
Type anything for input size, press enter for input matrix:

Input square matrix size (one int digit): 4
Input matrix:
1 2 3 4
5 6 7 8
8 7 6 5
4 3 2 1
True det 0.0
No threads, time = 0.0
Threads 1, time = 0.45787787437438965, result = 0.0
Threads 2, time = 0.5080811977386475, result = 0.0
Threads 4, time = 0.6610324382781982, result = 0.0
Threads 8, time = 0.9975094795227051, result = 0.0
Threads 16, time = 1.9495010375976562, result = 0.0
Threads 32, time = 3.816169261932373, result = 0.0

Process finished with exit code 0
```

Рис. 4.2: Автоматическая генерация матрицы заданного размера

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система — Windows [4] 10 64-bit;
- оперативная память — 16 Гб;
- процессор — Intel(R) Core(TM) i5-7600 CPU @ 3.50GHz [5].

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 4.3 графически изображена зависимость времени работы алгоритма от размерности матрицы и количества потоков.

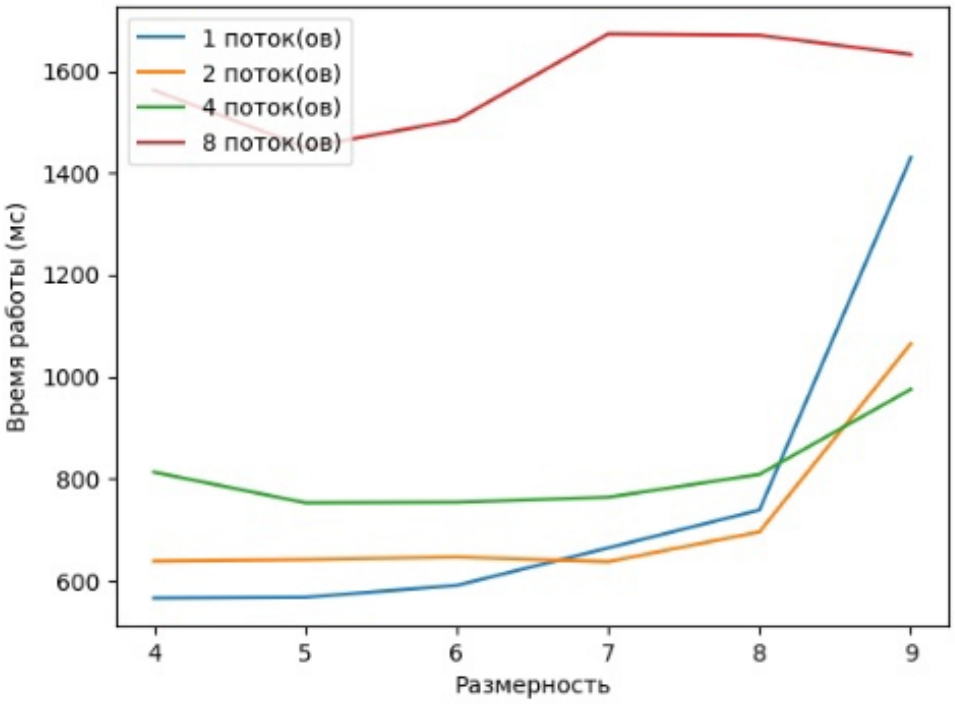

Рис. 4.3: Зависимость времени работы алгоритма от размерности матрицы и количества потоков

Поскольку процессор на устройстве содержит 4 физических и логических ядра, эффективность алгоритма улучшается только при создании 4 и менее потоков. Как видно по графику 4.3, 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

[1] Mario Nemirovsky D. M. T. Multithreading Architecture // Morgan and Claypool Publishers. 2013.

[2] Olukotun K. Chip Multiprocessor Architecture — Techniques to Improve Throughput and Latency // Morgan and Claypool Publishers. 2007. p. 154.

[3] About Python [Электронный ресурс]. Режим доступа: https://www.python.org/about/. Дата обращения: 03.10.2021.

[4] 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.

[5] 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.