



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 Распараллеливание алгоритма Винограда

Студент Пересторонин П.Г.

Группа ИУ7-53Б

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

Оглавление

Введение	2
1 Аналитическая часть	4
1.1 Описание задачи	4
2 Конструкторская часть	6
2.1 Разработка алгоритмов	6
2.1.1 Алгоритм Винограда	6
2.1.2 Параллельная реализация алгоритма Винограда . . .	8
3 Технологическая часть	13
3.1 Требования к ПО	13
3.2 Средства реализации	13
3.3 Листинг кода	13
3.4 Тестирование функций	21
4 Исследовательская часть	22
4.1 Технические характеристики	22
4.2 Время выполнения алгоритмов	22
Заключение	27
Литература	28

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, 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.

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 [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 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.

Задачи работы

В рамках выполнения работы необходимо решить следующие задачи:

- изучить понятие параллельных вычислений;
- реализовать последовательный и 2 параллельных алгоритма Винограда;
- сравнить временные характеристики реализованных алгоритмов экспериментально;
- на основании проделанной работы сделать отчет, в котором сделать выводы по проделанной работе.

1 Аналитическая часть

1.1 Описание задачи

Пусть даны две прямоугольные матрицы

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

тогда матрица C

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

где

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

будет называться произведением матриц A и B [3].

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: $V \cdot W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4$, что эквивалентно (1.4):

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

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умно-

жений - 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. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного [4].

В данной лабораторной работе стоит задача распараллеливания алгоритма Винограда по 2 схемам. Так как каждый элемент матрицы C вычисляется независимо от других и матрицы A и B не изменяются, то для параллельного вычисления произведения, достаточно просто равным образом распределить элементы матрицы C между потоками.

Вывод

Алгоритм перемножения матриц Винограда независимо вычисляет элементы матрицы-результата, что дает большое количество возможностей для реализации параллельного варианта алгоритма.

2 Конструкторская часть

2.1 Разработка алгоритмов

2.1.1 Алгоритм Винограда

На рисунке 2.1 представлена схема однопоточного алгоритма Винограда с выделением этапов, а на рисунке 2.2 представлены подробные схемы 1 и 2 этапов (предварительный расчёт для строк первой матрицы и столбцов второй).

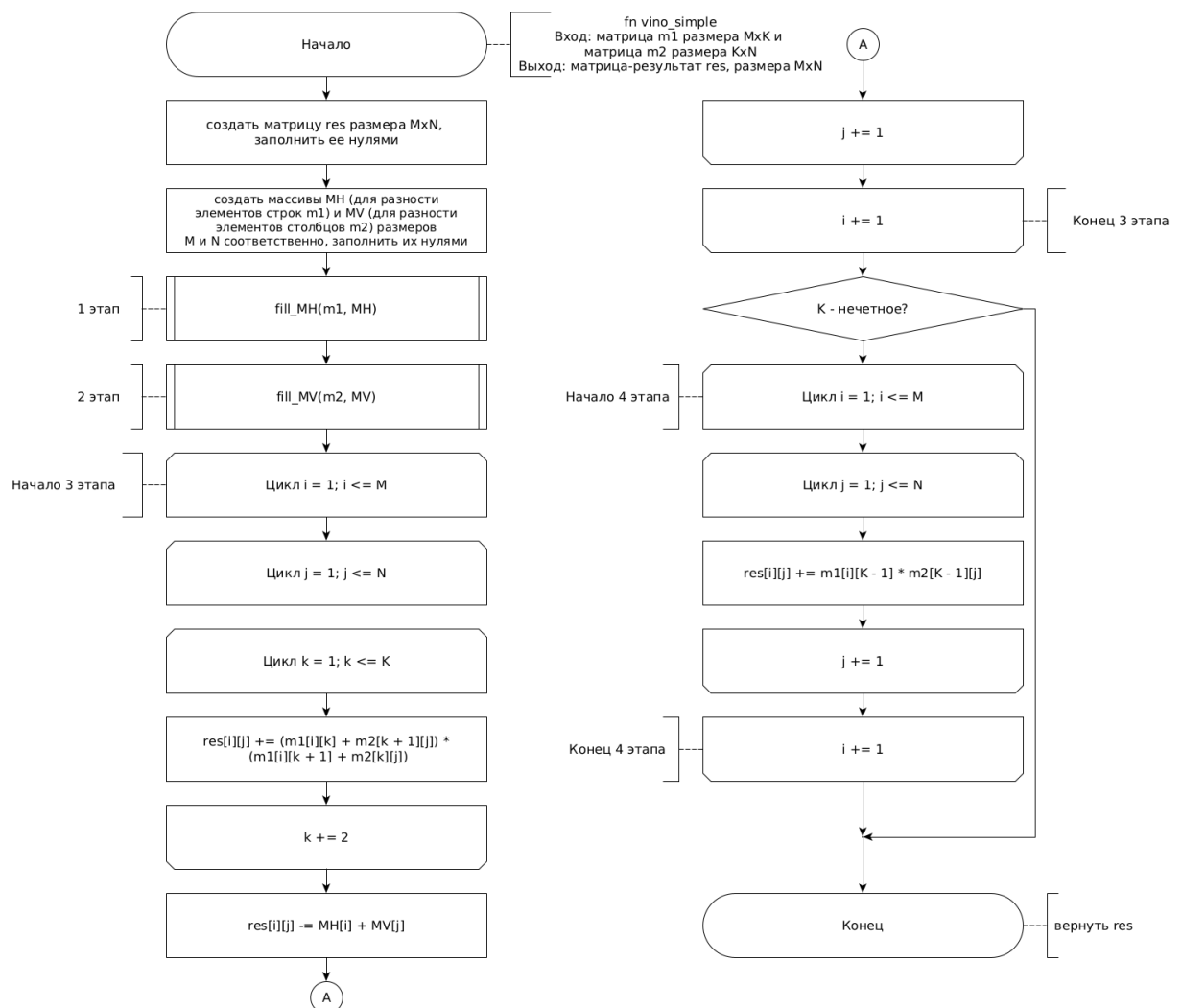


Рис. 2.1: Схема однопоточного алгоритма Винограда

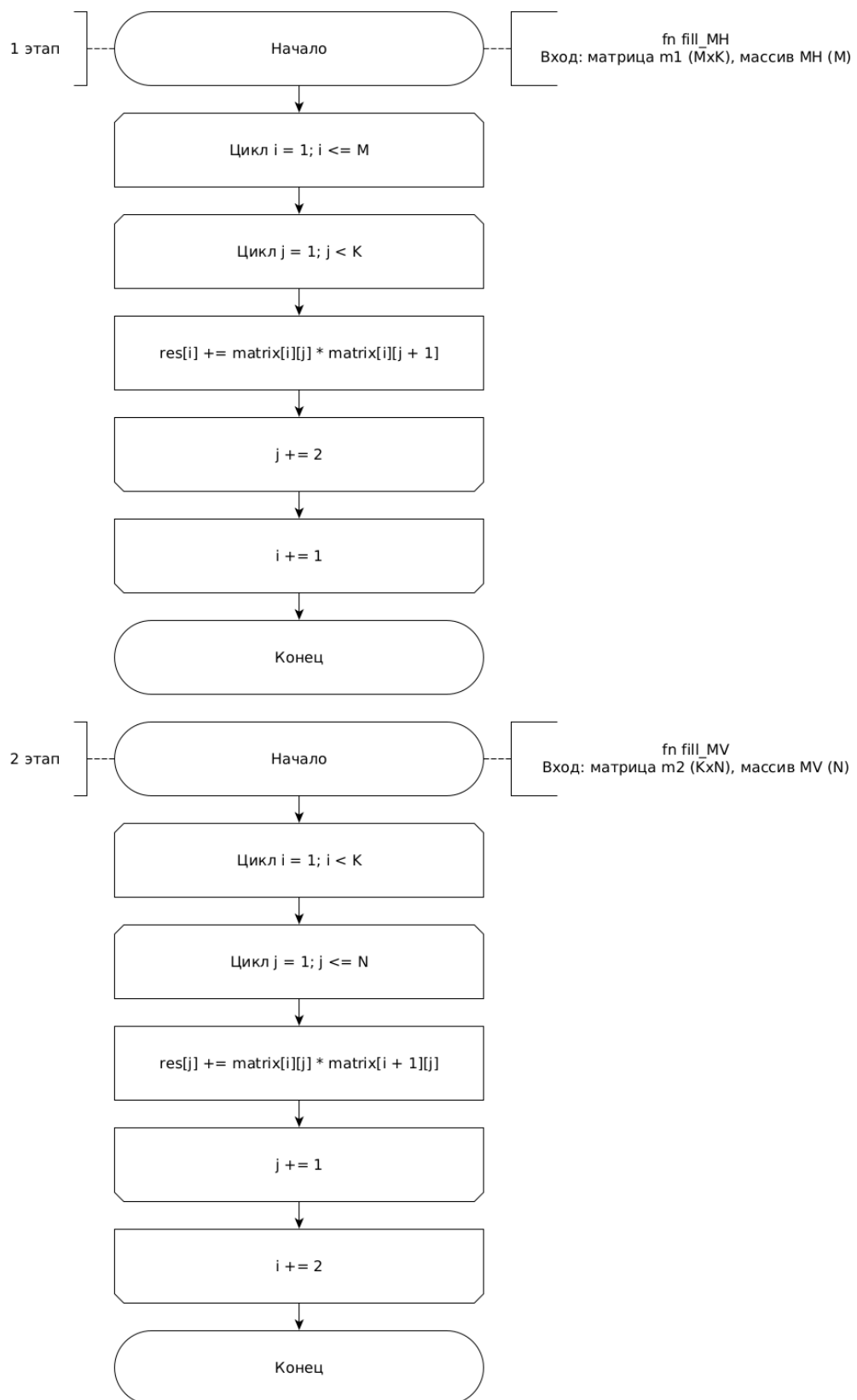


Рис. 2.2: Схема предварительного расчета для строк первой матрицы и столбцов второй

2.1.2 Параллельная реализация алгоритма Винограда

Пусть размеры перемножаемых матриц непосредственно равны $M \times K$ и $K \times N$.

Рассмотрим необходимые для распараллеливания замечания:

- каждый из выделенных этапов может быть выполнен независимо от других;
- в следствие независимости этапов, каждый из них может быть выполнен в любой момент, в том числе и параллельно с другими;
- каждый из выделенных этапов содержит цикл в некотором промежутке, который может быть разбит на некоторое множество меньших промежутков, в сумме составляющих исходный;
- трудоёмкости первого и второго этапов - величины одного порядка и относятся M/N ;
- трудоёмкость третьего этапа в N раз больше трудоёмкости первого этапа и в M раз больше трудоёмкости второго этапа, что, не позволяет распараллелить третий этап с первым и (или) вторым;
- четвертый этап требует обращения к матрице на каждой итерации цикла, что при распараллеливании приведёт к большому числу блокировок разделяемой памяти, и, в купе с затратами на порождение потоков, будет неэффективно.

На рисунке 2.3 представлена схема алгоритма функции, запускающая в требуемом количестве потоков функцию-аргумент, передавая ей равные по размеру промежутки из разбиения исходного. С помощью этой функции распараллеливаются этапы, описанные в рисунке 2.1.

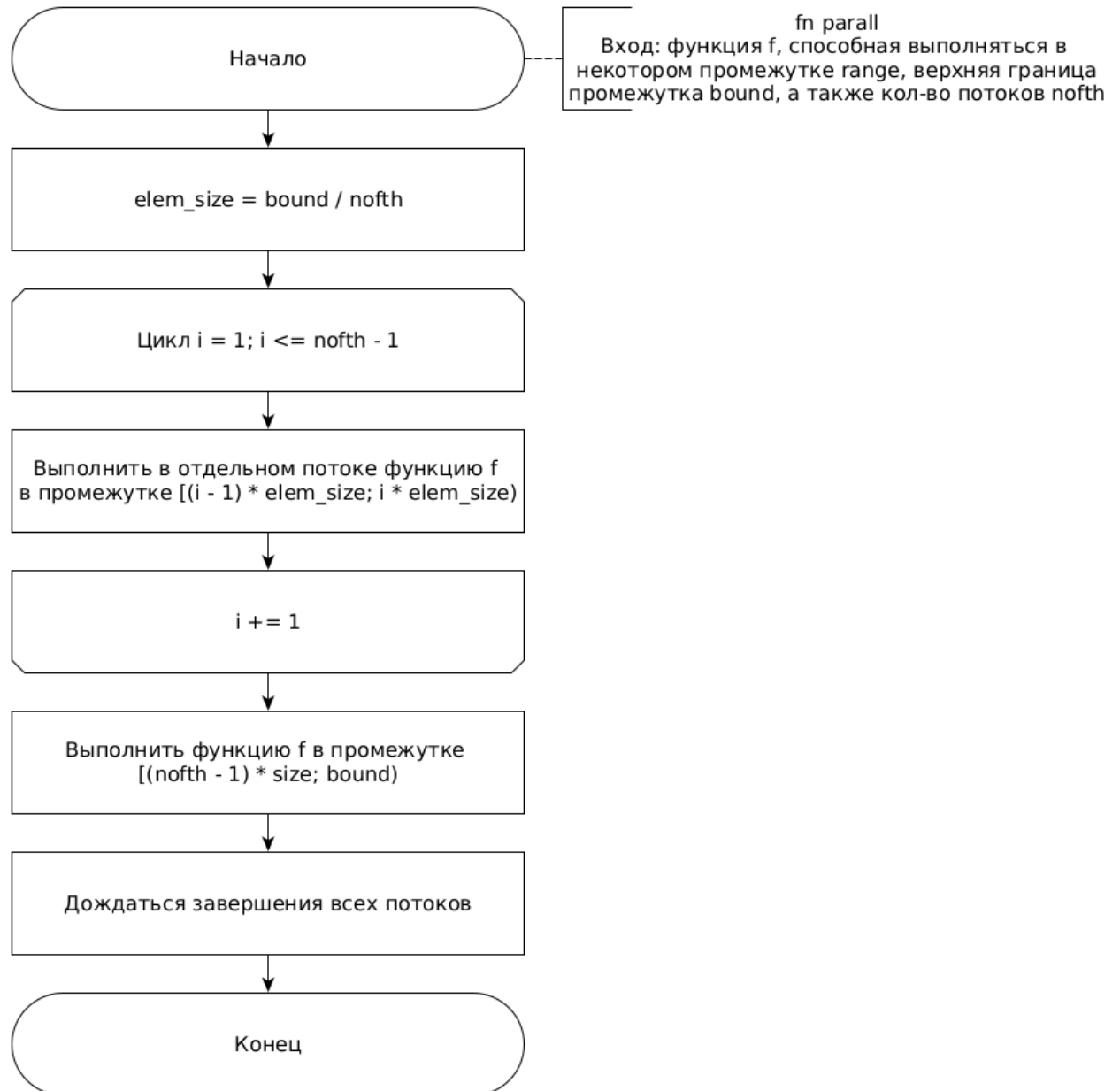


Рис. 2.3: Схема распараллеливания произвольной функции, способной выполняться в некоторых независимых промежутках

Исходя из всего, описанного выше, можно сделать вывод, что третий этап следует параллелить независимо от других, в то время как первый и второй этапы можно сделать как параллельно (где каждому из этапов достается половина от общего числа потоков), так и последовательно (где каждому из этапов достается общее число потоков). Так же очевидно, что

не следует параллелить четвертый этап.

На рисунке 2.4 представлена схема с параллельным выполнением первого и второго этапов, а на рисунке 2.5 представлена схема с последовательным выполнением первого и второго этапов.

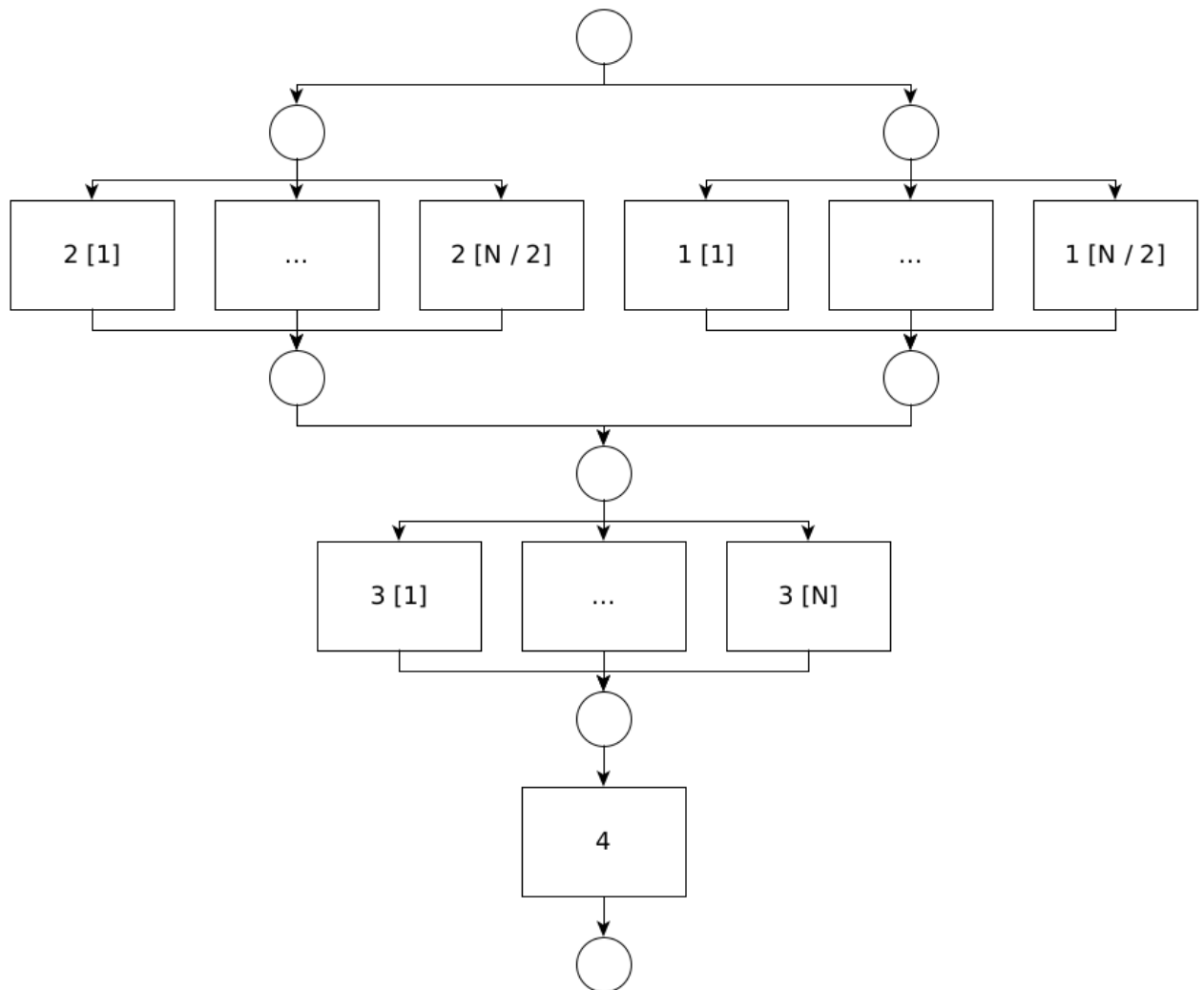


Рис. 2.4: Схема с параллельным выполнением первого и второго этапов

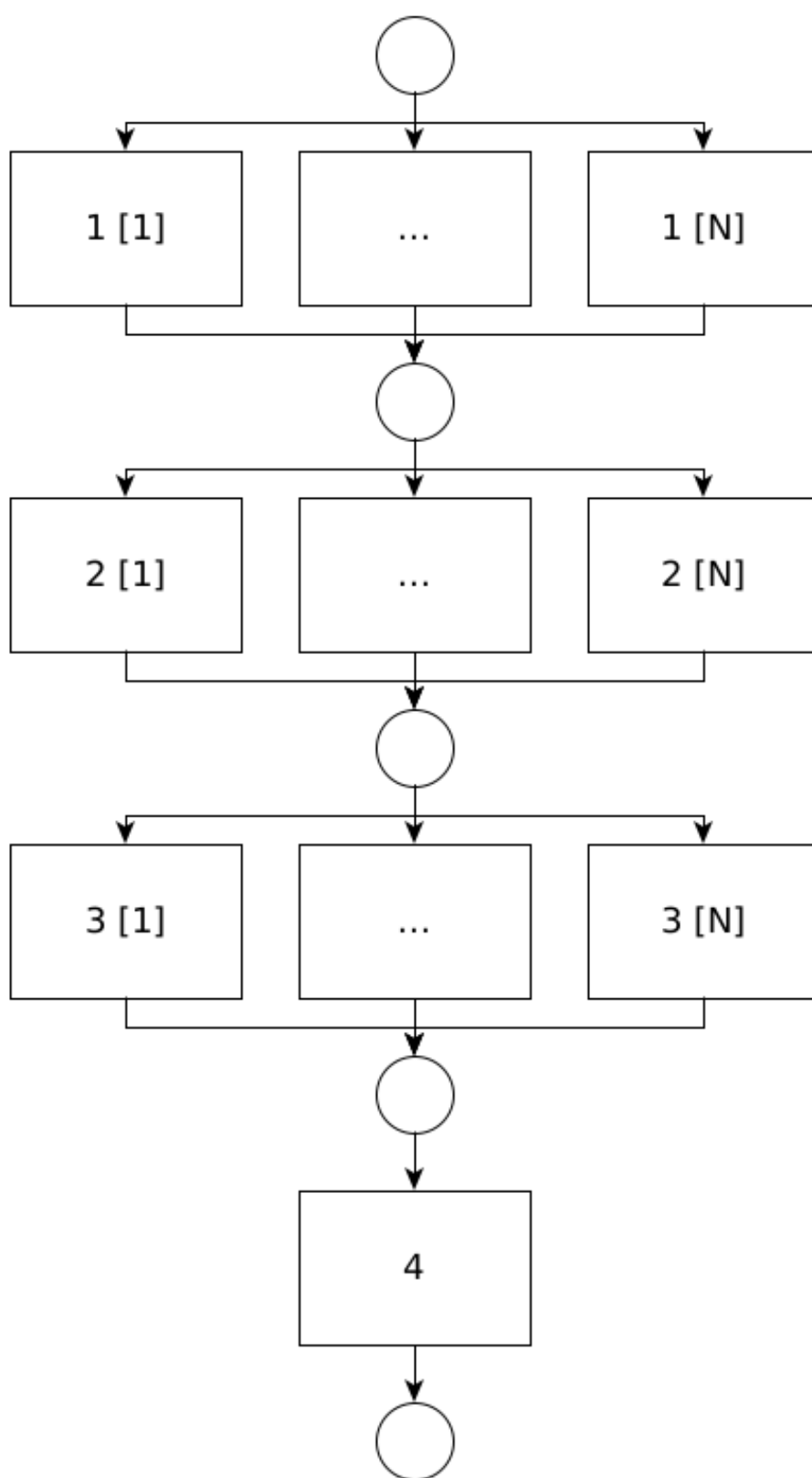


Рис. 2.5: Схема с последовательным выполнением первого и второго этапов

Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема алгоритма Винограда, а так же после разделения алгоритма на этапы были предложены 2 схемы параллельного выполнения данных этапов.

3 Технологическая часть

В данном разделе приведены средства программной реализации и листинг кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются размеры 2 матриц, а также их элементы;
- на выходе — матрица, которая является результатом умножения входных матриц.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран современный компилируемый ЯП Rust [5]. Данный выбор обусловлен популярностью языка и скоростью его выполнения, а также тем, что данный язык предоставляет широкие возможности для написания тестов [6].

3.3 Листинг кода

В листинге 3.1 приведена реализация простого алгоритма Винограда. В листингах 3.2 и 3.3 приведены реализации параллельных алгоритмов Винограда. В листингах 3.4 и 3.5 приведены вспомогательные однопоточные и многопоточные функции соответственно.

```
1 fn precompute_rows(matrix: &[Vec<MatInner>]) -> Vec<MatInner> {  
2     let mut res = vec![0; matrix.len()];  
3  
4     for i in 0..matrix.len() {  
5         for j in (0..(matrix[0].len() - 1)).step_by(2) {  
6             res[i] -= matrix[i][j] * matrix[i][j + 1];  
7         }  
8     }
```

```

8     }
9
10    res
11 }
12
13 fn precompute_cols(matrix: &[Vec<MatInner>]) -> Vec<MatInner> {
14     let mut res = vec![0; matrix[0].len()];
15
16     for i in (0..(matrix.len() - 1)).step_by(2) {
17         for j in 0..matrix[0].len() {
18             res[j] -= matrix[i][j] * matrix[i + 1][j];
19         }
20     }
21
22     res
23 }
24
25 fn precompute_values(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) -> (Vec<MatInner>,
26     Vec<MatInner>) {
27     (precompute_rows(m1), precompute_cols(m2))
28 }
29
30 pub fn vinograd_simple(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) -> Vec<Vec<MatInner>>
31 {
32     let mut matrix = get_result_matrix(m1, m2);
33     let precomputed = precompute_values(m1, m2);
34
35     let m = matrix.len();
36     let n = matrix[0].len();
37     let k_iteration = m2.len();
38
39     for i in 0..m {
40         for j in 0..n {
41             matrix[i][j] = precomputed.0[i] + precomputed.1[j];
42             for k in (0..(k_iteration - 1)).step_by(2) {
43                 matrix[i][j] += (m1[i][k] + m2[k + 1][j]) * (m1[i][k + 1] + m2[k][j]);
44             }
45         }
46     }
47
48     if m2.len() & 1 != 0 {
49         let max_k = m2.len() - 1;
50
51         for i in 0..matrix.len() {
52             for j in 0..matrix[0].len() {

```

```

54         matrix[i][j] += m1[i][max_k] * m2[max_k][j];
55     }
56 }
57 }
58
59 matrix
60 }

```

Листинг 3.1: Последовательный алгоритм Винограда

```

1 pub fn vinograd_parallel1(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) ->
    Vec<Vec<MatInner>> {
2     let matrix = get_result_matrix(m1, m2);
3     let precomputed = precompute_values_parallel(m1, m2);
4
5     let mut matrix = mult_parallel(matrix, m1, m2, &precomputed, NUMBER_OF_THREADS1 - 1);
6
7     if m2.len() & 1 != 0 {
8         matrix = odd_mult_sync(matrix, m1, m2);
9     }
10
11     matrix
12 }

```

Листинг 3.2: Параллельный алгоритм Винограда 1

```

1 pub fn vinograd_parallel2(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) ->
    Vec<Vec<MatInner>> {
2     let matrix = get_result_matrix(m1, m2);
3     let precomputed = (precompute_rows_parallel(&m1, NUMBER_OF_THREADS2 - 1),
4         precompute_cols_parallel(&m2, NUMBER_OF_THREADS2 - 1));
5
6     let mut matrix = mult_parallel(matrix, m1, m2, &precomputed, NUMBER_OF_THREADS2 - 1);
7
8     if m2.len() & 1 != 0 {
9         matrix = odd_mult_sync(matrix, m1, m2);
10    }
11
12    matrix
13 }

```

Листинг 3.3: Параллельный алгоритм Винограда 2

```

1 pub fn get_result_matrix(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) ->
    Vec<Vec<MatInner>> {
2     if m1.len() == 0 || m2.len() == 0 {
3         return Vec::new();
4     } else if m1[0].len() != m2.len() {
5         panic!("Bad_matrices!");

```



```

6     } else {
7         vec![vec![0; m2[0].len()]; m1.len()]
8     }
9 }
10
11 fn generate_row(size: usize) -> Vec<MatInner> {
12     let mut rng = rand::thread_rng();
13     (0..size).map(|_| rng.gen::<MatInner>() % MODULAR).collect()
14 }
15
16 pub fn generate_matrix(rows: usize, cols: usize) -> Vec<Vec<MatInner>> {
17     (0..rows).map(|_| generate_row(cols)).collect()
18 }
19
20 pub fn odd_mult_sync(mut matrix: Vec<Vec<MatInner>>, m1: &[Vec<MatInner>],
21     m2: &[Vec<MatInner>]) -> Vec<Vec<MatInner>> {
22     let max_k = m2.len() - 1;
23
24     for i in 0..m1.len() {
25         for j in 0..m2[0].len() {
26             matrix[i][j] += m1[i][max_k] * m2[max_k][j];
27         }
28     }
29
30     matrix
31 }

```

Листинг 3.4: Вспомогательные однопоточные функции

```

1 use crossbeam::thread as cthread;
2 use std::sync::{ Arc, Mutex };
3 use super::*;
4
5 fn precompute_rows_elementary(matrix_slice: &[Vec<MatInner>], res_slice:
6     Arc<Mutex<Vec<MatInner>>>,
7     range: std::ops::Range<usize>) {
8     let mut buf = vec![0; matrix_slice.len()];
9     for (ind, row) in matrix_slice.iter().enumerate() {
10         for i in (1..row.len()).step_by(2) {
11             buf[ind] += row[i - 1] * row[i];
12         }
13     }
14
15     let offset = range.start;
16     let mut res = res_slice.lock().unwrap();
17     for i in range {
18         res[i] = buf[i - offset];
19     }
20 }

```

```

20
21 pub fn precompute_rows_parallel(matrix: &[Vec<MatInner>], nofth: usize) -> Vec<MatInner>
22 {
23     if matrix.len() == 0 {
24         return Vec::new();
25     }
26
27     cthread::scope(|s| {
28         let res = Arc::new(Mutex::new(vec![0; matrix.len()]));
29         let mut threads = Vec::with_capacity(nofth);
30         let size = matrix.len() / (nofth + 1);
31
32         for i in 0..nofth {
33             let range = (i * size)..((i + 1) * size);
34             let res_cpy = res.clone();
35             threads.push(
36                 s.spawn(move |_| precompute_rows_elementary(&matrix[range.clone()],
37                     res_cpy, range)));
38         }
39
40         let range = (size * nofth)..matrix.len();
41         let res_cpy = res.clone();
42         threads.push(
43             s.spawn(|_| precompute_rows_elementary(&matrix[range.clone()], res_cpy,
44                 range)));
45
46         for th in threads {
47             th.join().unwrap();
48         }
49
50         Arc::try_unwrap(res).unwrap().into_inner().unwrap()
51     }).unwrap()
52 }
53
54 fn precompute_cols_elementary(matrix_slice: &[Vec<MatInner>], res_slice:
55     Arc<Mutex<Vec<MatInner>>>,
56     range: std::ops::Range<usize>) {
57     let mut buf = vec![0; range.end - range.start];
58
59     for (ind, j) in range.clone().enumerate() {
60         for i in (1..matrix_slice.len()).step_by(2) {
61             buf[ind] += matrix_slice[i][j] * matrix_slice[i - 1][j];
62         }
63     }
64
65     let mut res = res_slice.lock().unwrap();
66     let offset = range.start;

```

```

64     for i in range {
65         res[i] = buf[i - offset];
66     }
67 }
68
69 pub fn precompute_cols_parallel(matrix: &[Vec<MatInner>], nofth: usize) -> Vec<MatInner>
70 {
71     if matrix.len() == 0 {
72         return Vec::new();
73     }
74
75     cthread::scope(|s| {
76         let res = Arc::new(Mutex::new(vec![0; matrix[0].len()]));
77         let mut threads = Vec::with_capacity(nofth);
78         let size = matrix[0].len() / (nofth + 1);
79
80         for i in 0..nofth {
81             let range = (i * size)..((i + 1) * size);
82             let res_cpy = res.clone();
83             threads.push(s.spawn(move |_| precompute_cols_elementary(&matrix, res_cpy,
84                 range)));
85         }
86
87         let range = (size * nofth)..matrix[0].len();
88         precompute_cols_elementary(&matrix, res.clone(), range);
89
90         for th in threads {
91             th.join().unwrap();
92         }
93
94         Arc::try_unwrap(res).unwrap().into_inner().unwrap()
95     }).unwrap()
96 }
97
98 pub fn mult_parallel(matrix: Vec<Vec<MatInner>>, m1: &[Vec<MatInner>], m2:
99     &[Vec<MatInner>], precomputed: &(Vec<MatInner>, Vec<MatInner>), nofth: usize) ->
100     Vec<Vec<MatInner>> {
101     cthread::scope(|s| {
102         let mut threads = Vec::with_capacity(nofth);
103
104         let size = matrix.len() / (nofth + 1);
105         let matrix_guard = Arc::new(Mutex::new(matrix));
106
107         for i in 0..nofth {
108             let range = (i * size)..((i + 1) * size);
109             let guard_copy = matrix_guard.clone();
110             let precopy = &precomputed;

```

```

108         threads.push(s.spawn(move |_| multiplication_in_range(&precopy, guard_copy,
109             &m1, &m2, range)));
110     }
111
112     let range = (nofth * size)..m1.len();
113     let guard_copy = matrix_guard.clone();
114     multiplication_in_range(&precomputed, guard_copy, &m1, &m2, range);
115     for th in threads {
116         th.join().unwrap();
117     }
118
119     let matrix = Arc::try_unwrap(matrix_guard).unwrap().into_inner().unwrap();
120
121     matrix
122 }).unwrap()
123 }
124
125 pub fn multiplication_in_range(precomputed: &(Vec<MatInner>, Vec<MatInner>),
126     matrix_guard: Arc<Mutex<Vec<Vec<MatInner>>>>, m1: &[Vec<MatInner>],
127     m2: &[Vec<MatInner>], range: std::ops::Range<usize>) {
128     let k_iterations = m2.len();
129     let n = m2[0].len();
130
131     for i in range {
132         for j in 0..n {
133             let mut buffer = -precomputed.0[i] - precomputed.1[j];
134             for k in (0..(k_iterations - 1)).step_by(2) {
135                 buffer += (m1[i][k] + m2[k + 1][j]) * (m1[i][k + 1] + m2[k][j]);
136             }
137             let mut matrix = matrix_guard.lock().unwrap();
138             matrix[i][j] = buffer;
139         }
140     }
141 }
142
143 pub fn precompute_values_parallel(m1: &[Vec<MatInner>], m2: &[Vec<MatInner>]) ->
144     (Vec<MatInner>, Vec<MatInner>) {
145     cthread::scope(move |s| {
146         let nofth = NUMBER_OF_THREADS1 - 2;
147
148         let t1 = s.spawn(move |_| precompute_rows_parallel(&m1, nofth / 2));
149         let p2 = precompute_cols_parallel(&m2, nofth / 2 + nofth & 1);
150
151         let p1 = t1.join().unwrap();
152         (p1, p2)
153     }).unwrap()
154 }

```

```

154 pub fn _odd_mult_parallel(matrix: Vec<Vec<MatInner>>, m1: &[Vec<MatInner>],
155 m2: &[Vec<MatInner>], nofth: usize) -> Vec<Vec<MatInner>> {
156     cthread::scope(move |s| {
157         let mut threads = Vec::with_capacity(nofth);
158
159         let size = matrix.len() / (nofth + 1);
160         let matrix_guard = Arc::new(Mutex::new(matrix));
161
162         for i in 0..nofth {
163             let range = (i * size)..((i + 1) * size);
164             let guard_copy = matrix_guard.clone();
165             threads.push(s.spawn(move |_| _odd_mult_in_range(guard_copy, &m1, &m2,
166                 range)));
167         }
168
169         let range = (nofth * size)..m1.len();
170         let guard_copy = matrix_guard.clone();
171         _odd_mult_in_range(guard_copy, &m1, &m2, range);
172
173         for th in threads {
174             th.join().unwrap();
175         }
176
177         let matrix = Arc::try_unwrap(matrix_guard).unwrap().into_inner().unwrap();
178         matrix
179     }).unwrap()
180 }
181
182 fn _odd_mult_in_range(matrix_guard: Arc<Mutex<Vec<Vec<MatInner>>>>, m1: &[Vec<MatInner>],
183 m2: &[Vec<MatInner>], range: std::ops::Range<usize>) {
184     let max_k = m2.len() - 1;
185
186     for i in range {
187         for j in 0..m2[0].len() {
188             let buf = m1[i][max_k] * m2[max_k][j];
189             let mut matrix = matrix_guard.lock().unwrap();
190             matrix[i][j] += buf;
191         }
192     }
193 }

```

Листинг 3.5: Вспомогательные многопоточные функции

3.4 Тестирование функций

В таблице 3.1 приведены тесты для функций, реализующих однопоточный и многопоточный алгоритмы Винограда. Тесты пройдены успешно.

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 6 & 12 & 18 \\ 6 & 12 & 18 \\ 6 & 12 & 18 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 3 & 6 \\ 3 & 6 \end{pmatrix}$
(2)	(2)	(4)
$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$
(1 2)	(1 2)	Не могут быть перемножены

Таблица 3.1: Тестирование функций

Вывод

Правильный выбор инструментов разработки позволил эффективно реализовать алгоритмы, настроить модульное тестирование и выполнить исследовательский раздел лабораторной работы.

4 Исследовательская часть

4.1 Технические характеристики

- Операционная система: Manjaro [7] Linux [8] x86_64.
- Память: 8 ГБ.
- Процессор: Intel® Core™ i7-8550U[9].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

Результаты замеров приведены в таблицах 4.1 и 4.2. На рисунке 4.1 приведено сравнение простого алгоритма и параллельных алгоритмах при исполнении на 8 потоках, а на рисунке 4.2 приведен график зависимости времени работы 2 параллельных алгоритмов от кол-ва используемых потоков при перемножении матриц размера 500.

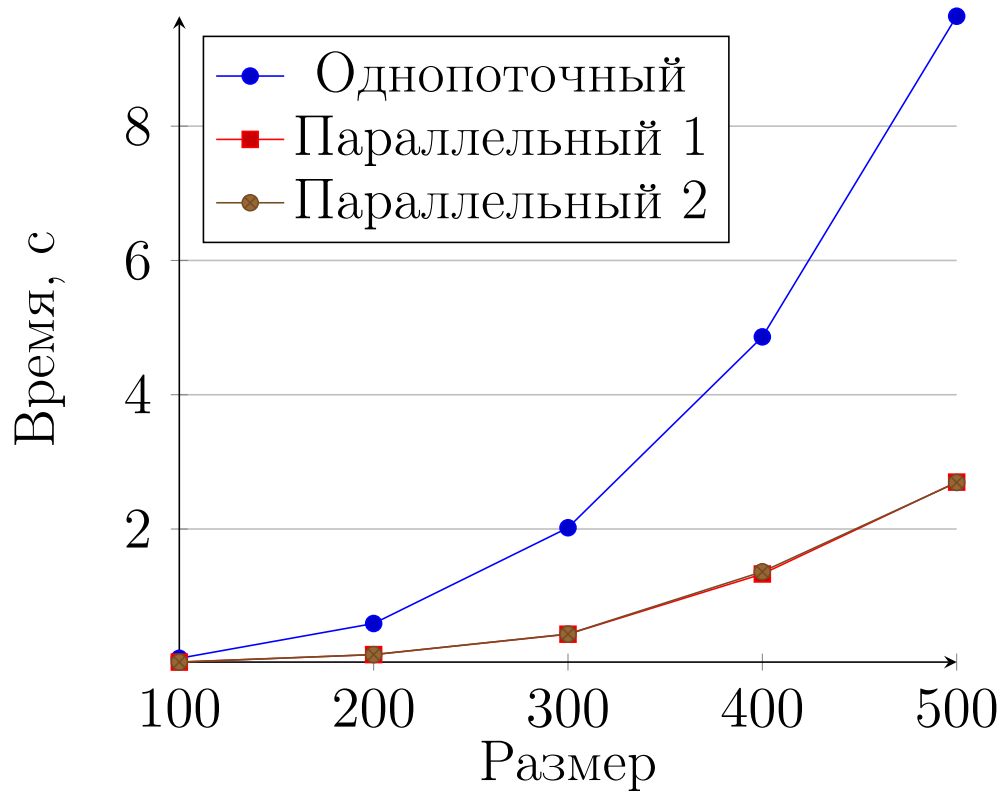


Рис. 4.1: Зависимость времени работы алгоритмов от размеров квадратной матрицы

Размер	Сравнение времени работы алгоритмов, с		
	Простой	Параллельный 1	Параллельный 2
100	0.074706872	0.018282067	0.018363864
200	0.592781543	0.129555696	0.129755026
300	2.018969587	0.433536846	0.436197555
400	4.861893087	1.330399187	1.363892051
500	9.637673585	2.698750384	2.692891268

Таблица 4.1: Время работы алгоритмов при различных размерах

Потоки	Время при кол-ве потоков, с	
	Параллельный 1	Параллельный 2
2	3.947089585	4.520112435
4	2.439386432	2.965838115
8	2.294140352	2.697487545
16	2.405290496	2.692021774
32	2.376067741	2.876883444

Таблица 4.2: Сравнение времени работы 2 параллельных алгоритмов при различном кол-ве потоков

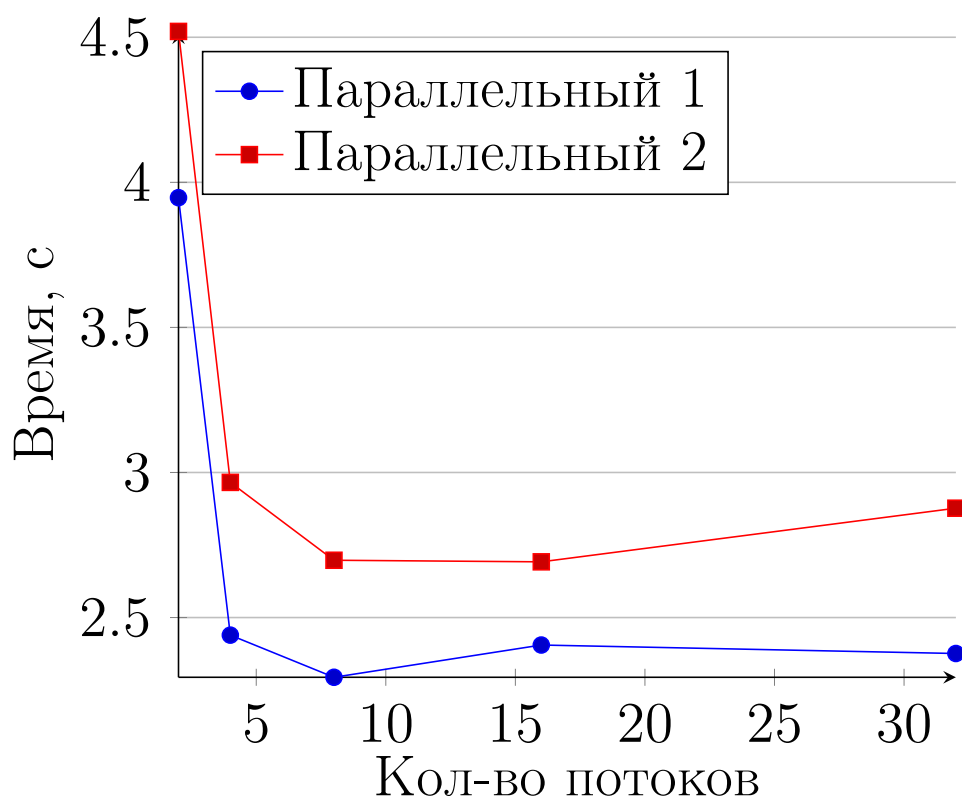


Рис. 4.2: Зависимость времени работы параллельных алгоритмов от кол-ва потоков на квадратных матрицах размера 500

Примечание: на одном потоке параллельные алгоритмы выполняются 7.379062295 секунд.

Вывод

Наилучшее время параллельные алгоритмы показали на 8 потоках (можно заметить, что минимум времени на графике 4.2 находится в точке 8), соответствующих количеству логических ядер ноутбука, на котором проводилось тестирование. На матрицах размеров 500 на 500 параллельные алгоритмы улучшают время однопоточной реализации примерно на 72%. Количество потоков, большее 8, в итоге немногим замедляет время необходимостью переключения между потоками, а так же большим количеством инициализаций.

Заключение

В рамках выполнения работы были решены следующие задачи:

- было изучено понятие параллельных вычислений;
- были реализованы последовательный и 2 параллельных алгоритма Винограда;
- было произведено сравнение временных характеристик реализованных алгоритмов экспериментально;
- на основании проделанной работы был сделан отчет с выводами по проделанной работе.

Несмотря на более сложный код, параллельные алгоритмы значительно выигрывают по времени аналогичные однопоточные реализации. Наиболее эффективны данные алгоритмы при количестве потоков, совпадающем с количеством логических ядер компьютера. Так, на матрицах 500 на 500 удалось улучшить время выполнения алгоритма на 72% (в сравнении с однопоточной реализацией).

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] Group-theoretic Algorithms for Matrix Multiplication / H. Cohn, R. Kleinberg, B. Szegedy et al. // Proceedings of the 46th Annual Symposium on Foundations of Computer Science. 2005. October. P. 379–388.
- [4] Погорелов Дмитрий. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.
- [5] Rust Programming Language [Электронный ресурс]. URL: <https://doc.rust-lang.org/std/index.html>.
- [6] Документация по ЯП Rust: бенчмарки [Электронный ресурс]. Режим доступа: <https://doc.rust-lang.org/1.7.0/book/benchmark-tests.html> (дата обращения: 10.10.2020).
- [7] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: <https://manjaro.org/> (дата обращения: 10.10.2020).
- [8] Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: <https://www.linux.org.ru/> (дата обращения: 10.10.2020).
- [9] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html> (дата обращения: 10.10.2020).