



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

Студент _____ Топорков Павел _____

Группа _____ ИУ7-53Б _____

Дисциплина _____ Анализ алгоритмов _____

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

подпись, дата

Фамилия, И.О.

Оценка _____

Москва — 2021 г.

Оглавление

Введение	3
1 Аналитическая часть	5
1.1 Описание задачи	5
2 Конструкторская часть	7
2.1 Разработка алгоритмов	7
3 Технологическая часть	14
3.1 Требования к ПО	14
3.2 Средства реализации	14
3.3 Листинг кода	14
3.4 Тестирование функций	20
4 Исследовательская часть	21
4.1 Пример работы	21
4.2 Технические характеристики	21
4.3 Время выполнения алгоритмов	22
Заключение	25
Литература	26

Введение

Целью данной лабораторной работы является изучить и реализовать параллельные вычисления.

Многопоточность — способность центрального процессора (ЦПУ) 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.

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.

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 в работу друг друга при совместном использовании аппаратных ресурсов;
- с программной точки зрения аппаратная поддержка многопоточности более трудоемка для программного обеспечения;
- проблема планирования потоков;

Задачи лабораторной работы:

- изучить понятие параллельных вычислений;
- реализовать последовательную и две параллельных реализации алгоритма перемножения матриц;
- сравнить временные характеристики реализованных алгоритмов экспериментально.

1 Аналитическая часть

В данном разделе представлены теоретические сведения о рассматриваемых алгоритмах.

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 . Стандартный алгоритм реализует данную формулу.

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

В данной работе стоит задача реализации алгоритма параллельного умножения матриц. Обычный алгоритм перемножения матриц независимо вычисляет элементы матрицы-результата, что дает большое количество возможностей для реализации параллельного варианта алгоритма.

2 Конструкторская часть

В данном разделе представлены схемы рассматриваемых алгоритмов.

2.1 Разработка алгоритмов

На рисунках 2.1 и 2.2 приведены схемы распараллеливания алгоритма простого умножения матриц.

На рисунках 2.3, 2.4, 2.5 приведены схемы алгоритмов простого умножения матриц без распараллеливания, с распараллеливанием по схеме 2.1 и с распараллеливанием по схеме 2.2 соответственно.

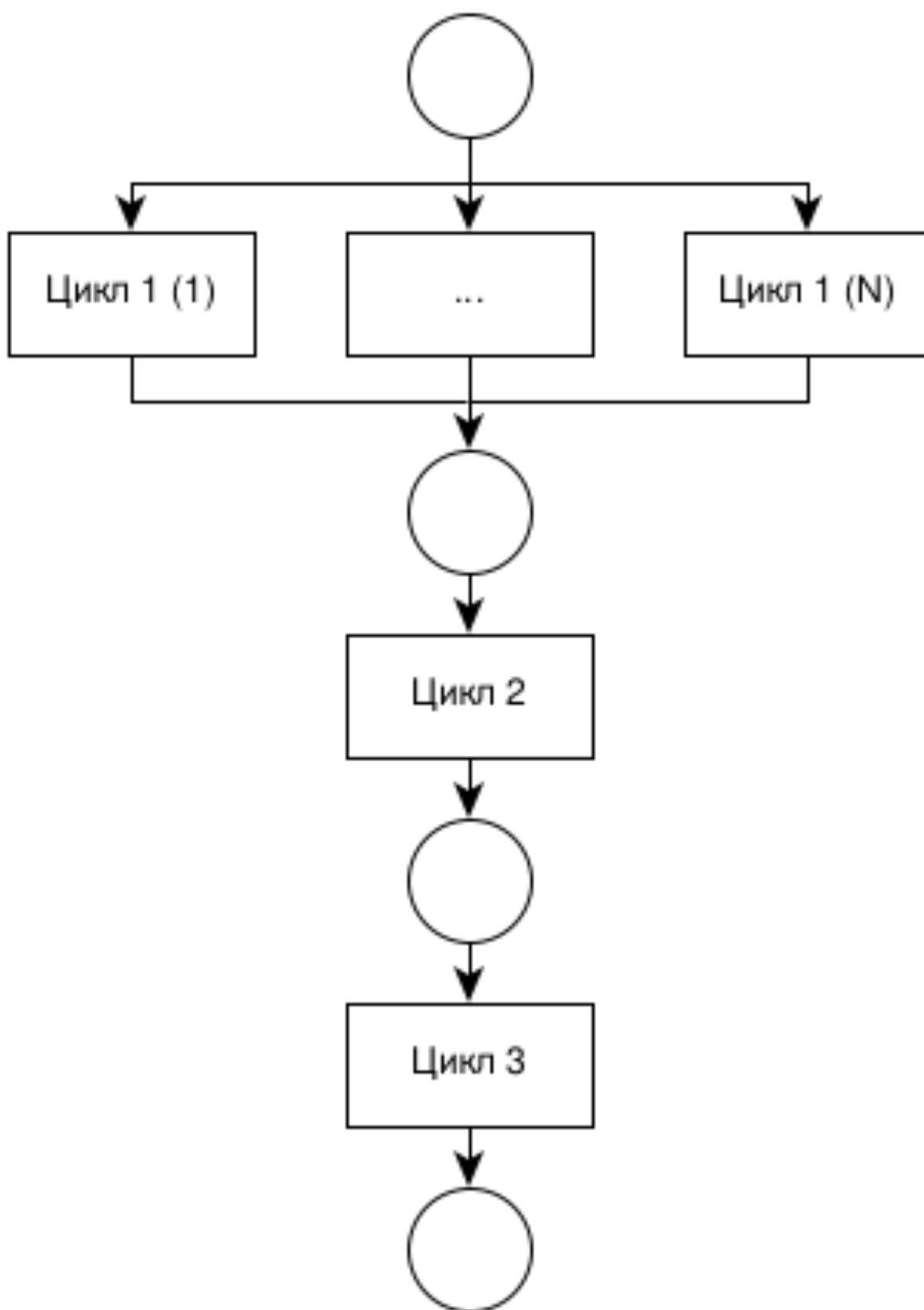


Рисунок 2.1 – Схема распараллеливания алгоритма, способ №1

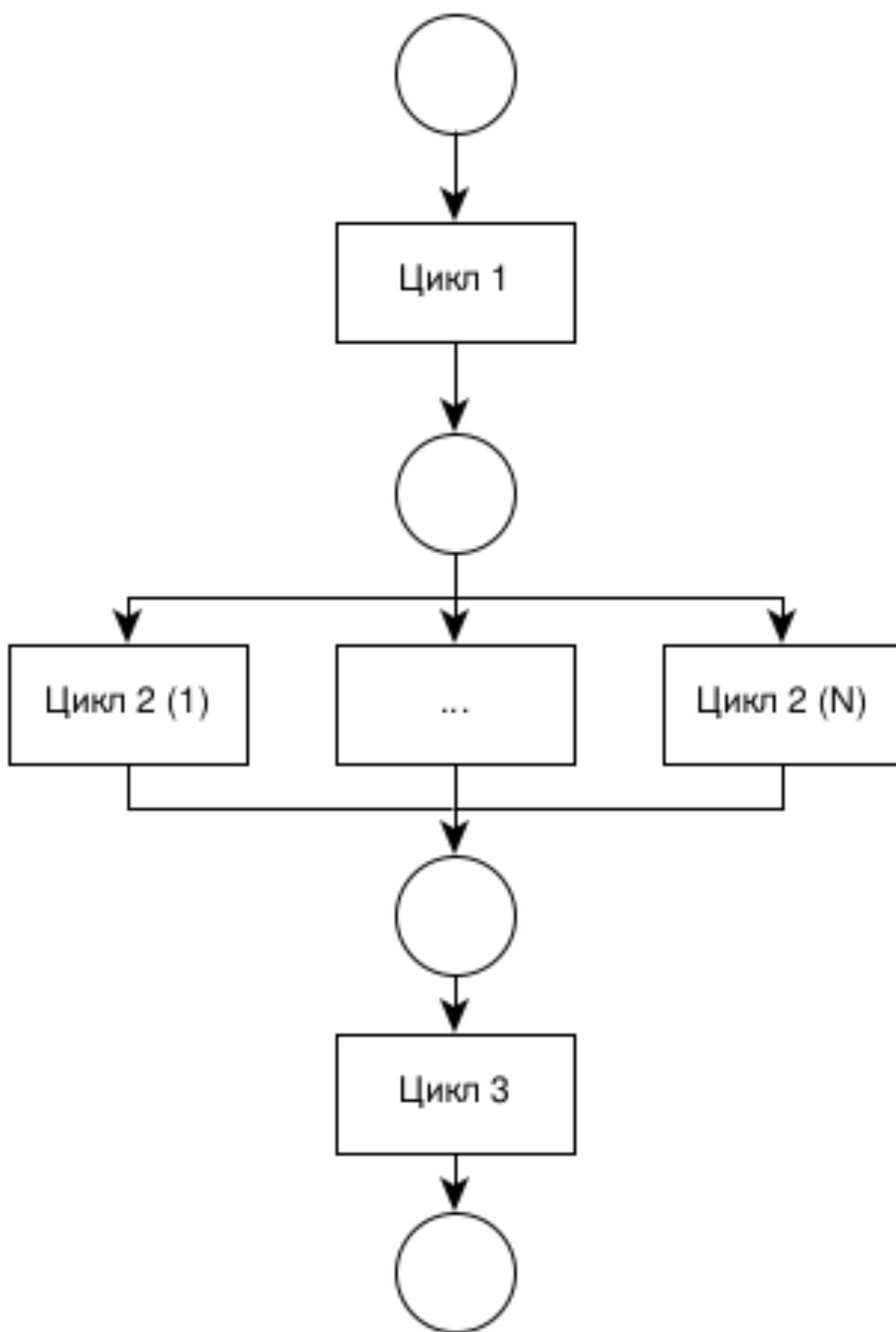


Рисунок 2.2 – Схема распараллеливания алгоритма, способ №2

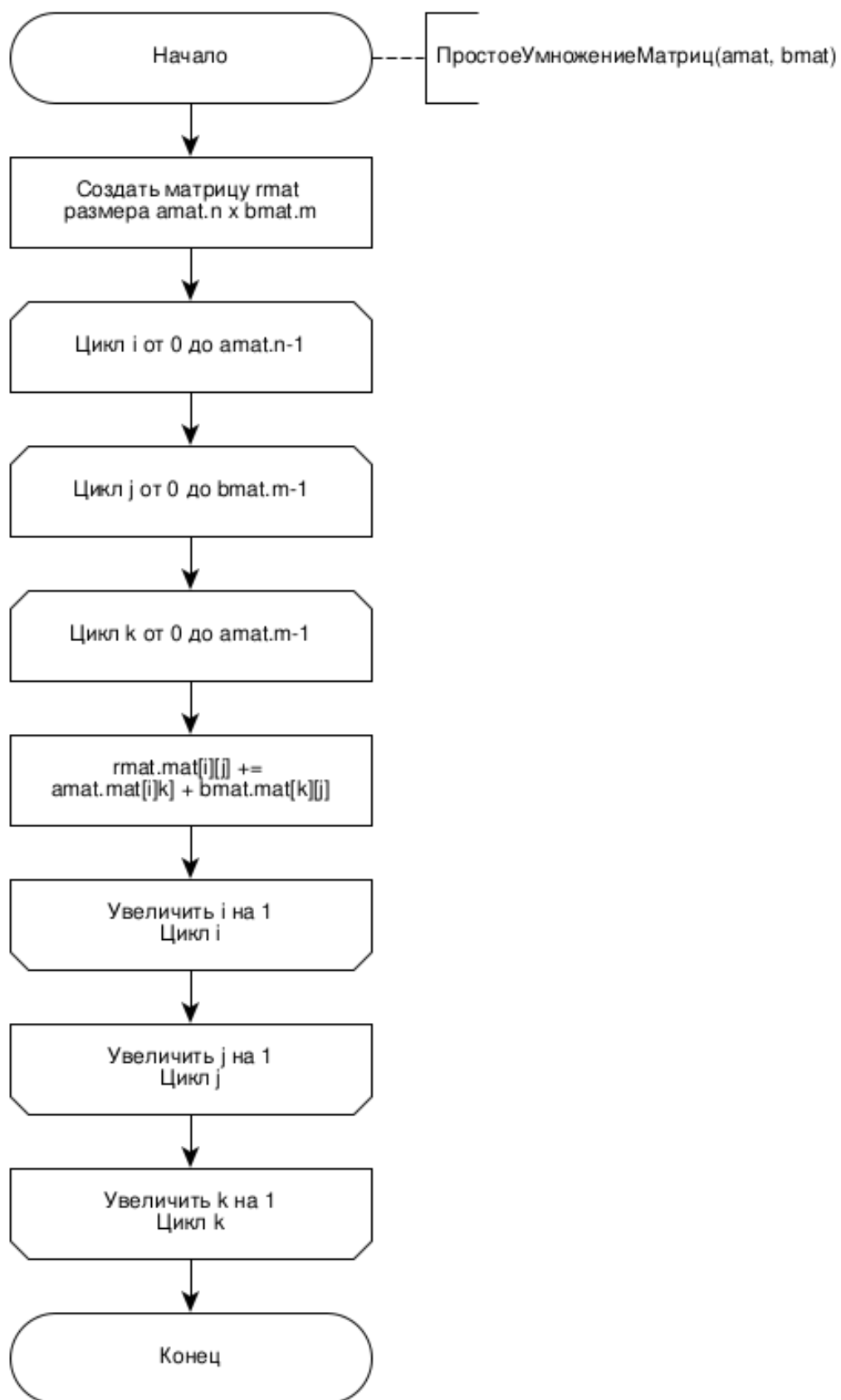


Рисунок 2.3 – Схема алгоритма простого умножения матриц

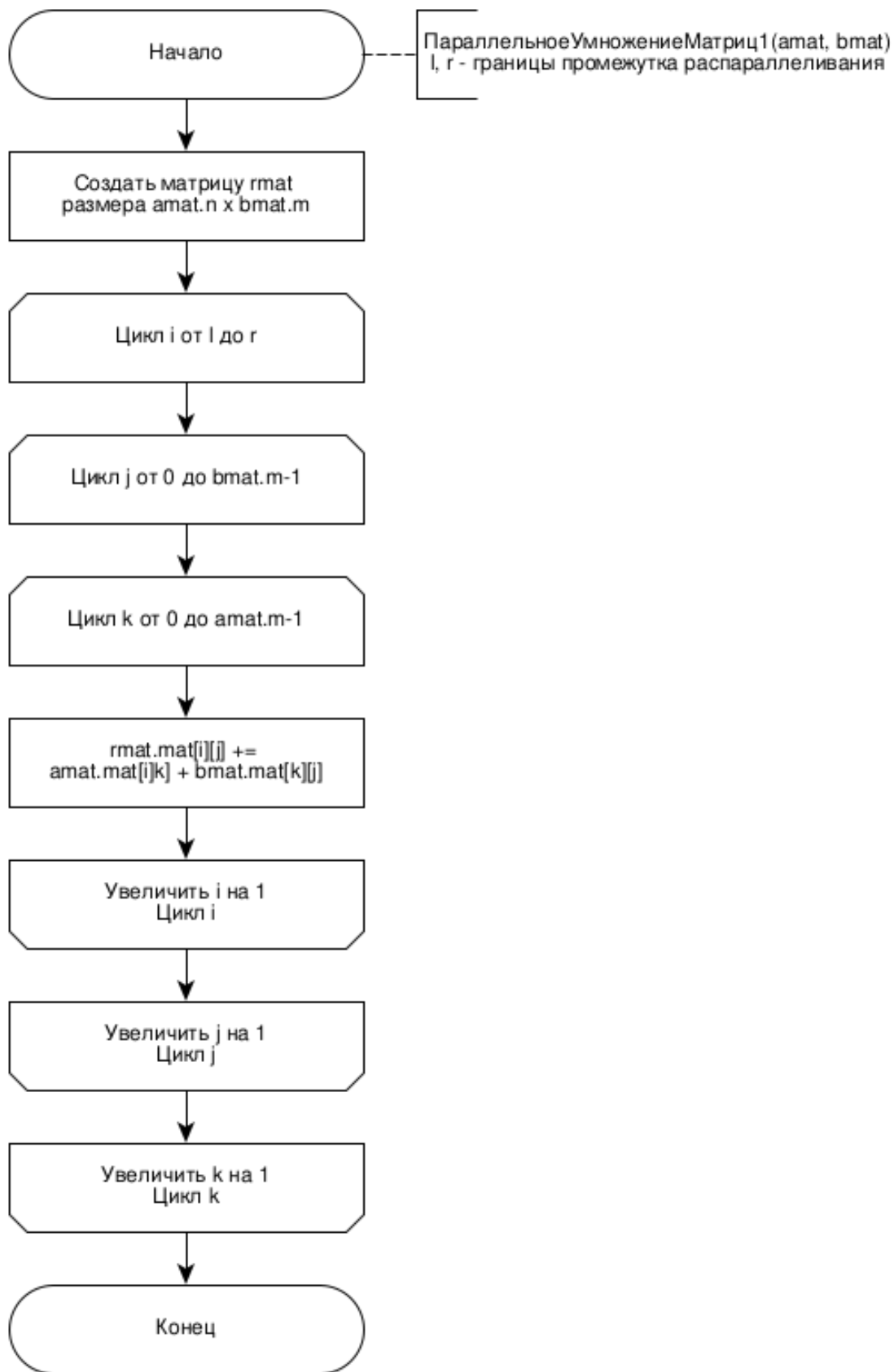


Рисунок 2.4 – Схема алгоритма распараллеленного простого умножения матриц, способ №1

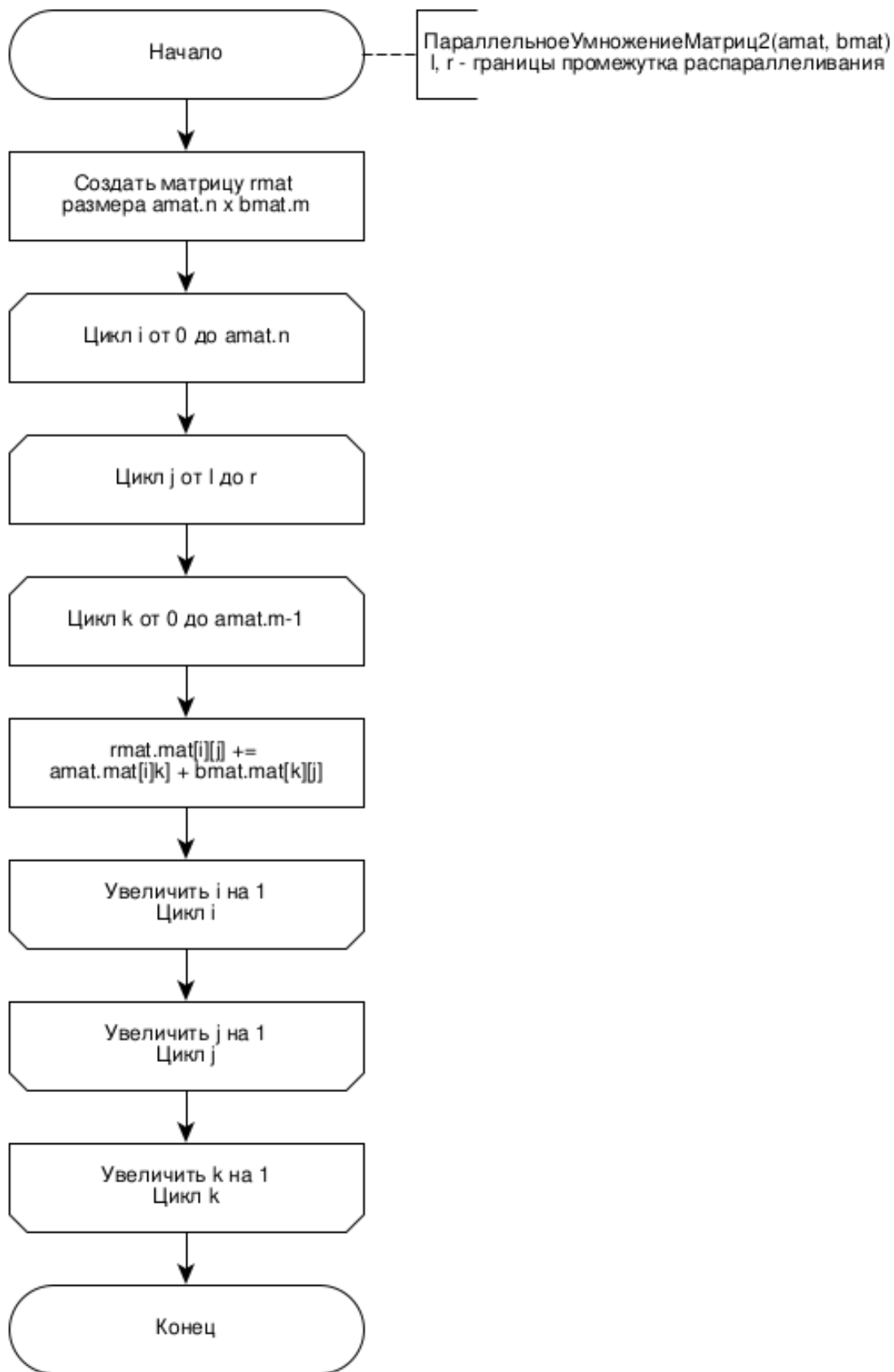


Рисунок 2.5 – Схема алгоритма распараллеленного простого умножения матриц, способ №2

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются размеры двух матриц а также их элементы;
- на выходе — матрица, являющаяся результатом умножения матриц, полученных на входе.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C [1]. Данный выбор обусловлен моими знаниями в области применения данного языка, а также запретом на использование в данной лабораторной работе моего основного языка - Golang [2].

3.3 Листинг кода

В листингах 3.1 и 3.2 приведены реализации алгоритмов умножения матриц и алгоритмов распределения потоков соответственно.

Листинг 3.1 – Реализация алгоритмов умножения матриц

```
1 #include "multiplication.h"
2
3 #define SWAP(t, a, b) \
4     do \
5     { \
6         t c = a; \
7         a = b; \
8         b = c; \
```

```

9     } while (0);
10
11 void print_matrix(int **matrix)
12 {
13     for (int i = 0; i < N; i++)
14     {
15         for (int j = 0; j < M; j++)
16         {
17             printf("%d_", matrix[i][j]);
18         }
19         printf("\n");
20     }
21 }
22
23 void read_matrix(int **matrix, FILE *file)
24 {
25     for (int i = 0; i < N; i++)
26     {
27         for (int j = 0; j < M; j++)
28         {
29             fscanf(file, "%d", &matrix[i][j]);
30         }
31     }
32 }
33
34 void init_matrix(int **matrix)
35 {
36     for (int i = 0; i < N; i++)
37     {
38         for (int j = 0; j < M; j++)
39         {
40             matrix[i][j] = 10;
41         }
42     }
43 }
44
45 int **create_matrix(int n, int m)
46 {
47     int **matrix = malloc(sizeof(int *) * n);
48
49     if (!matrix)
50     {
51         return NULL;
52     }
53
54     for (int i = 0; i < n; i++)
55     {
56         *(matrix + i) = malloc(sizeof(int) * m);

```

```

57
58     if (!(*matrix + i))
59     {
60         return NULL;
61     }
62 }
63
64 return matrix;
65 }
66
67 args_t *create_args(int n, int m, int k, int read_file)
68 {
69     args_t *args = malloc(sizeof(args));
70     if (!args)
71     {
72         return NULL;
73     }
74
75     if (!(args->m1 = create_matrix(n, m)))
76     {
77         return NULL;
78     }
79
80     if (read_file)
81     {
82         FILE *mtr_file = fopen("data/matrix1.txt", "r");
83
84         if (!mtr_file)
85         {
86             return NULL;
87         }
88
89         read_matrix(args->m1, mtr_file);
90         fclose(mtr_file);
91     }
92     else
93     {
94         init_matrix(args->m1);
95     }
96
97     if (!(args->m2 = create_matrix(m, k)))
98     {
99         return NULL;
100     }
101
102     if (read_file)
103     {
104         FILE *mtr_file = fopen("data/matrix2.txt", "r");

```



```

105
106     if (!mtr_file)
107     {
108         return NULL;
109     }
110
111     read_matrix(args->m2, mtr_file);
112     fclose(mtr_file);
113 }
114 else
115 {
116     init_matrix(args->m2);
117 }
118
119 if (!(args->res = create_matrix(n, k)))
120 {
121     return NULL;
122 }
123
124 return args;
125 }
126
127 void base_multiplication(args_t *args)
128 {
129     for (int i = 0; i < N; i++)
130     {
131         for (int j = 0; j < K; j++)
132         {
133             args->res[i][j] = 0;
134
135             for (int k = 0; k < M; k++)
136             {
137                 args->res[i][j] += args->m1[i][k] * args->m2[k][j];
138             }
139         }
140     }
141 }
142
143 void *parallel_multiplication1(void *args)
144 {
145     pthread_args_t *argsp = (args_t *)args;
146
147     int row_start = argsp->tid * (argsp->size / argsp->cnt_threads);
148     int row_end = (argsp->tid + 1) * (argsp->size / argsp->cnt_threads);
149
150     for (int i = row_start; i < row_end; i++)
151     {
152         for (int j = 0; j < K; j++)

```

```

153     {
154         argsp->mult_args->res[i][j] = 0;
155
156         for (int k = 0; k < M; k++)
157         {
158             argsp->mult_args->res[i][j] += argsp->mult_args->m1[i][k] *
159                 argsp->mult_args->m2[k][j];
160         }
161     }
162
163     return NULL;
164 }
165
166 void *parallel_multiplication2(void *args)
167 {
168     pthread_args_t *argsp = (args_t *)args;
169
170     int col_start = argsp->tid * (argsp->size / argsp->cnt_threads);
171     int col_end = (argsp->tid + 1) * (argsp->size / argsp->cnt_threads);
172
173     for (int i = 0; i < N; i++)
174     {
175         for (int j = col_start; j < col_end; j++)
176         {
177             argsp->mult_args->res[i][j] = 0;
178
179             for (int k = 0; k < M; k++)
180             {
181                 argsp->mult_args->res[i][j] += argsp->mult_args->m1[i][k] *
182                     argsp->mult_args->m2[k][j];
183             }
184         }
185     }
186
187     return NULL;
188 }

```

Листинг 3.2 – Реализация алгоритмов распределения потоков

```

1  #include "threads.h"
2
3  int start_threading(args_t *args, const int cnt_threads, const int type)
4  {
5      pthread_t *threads = malloc(cnt_threads * sizeof(pthread_t));
6
7      if (!threads)
8      {
9          return ALLOCATE_ERROR;

```

```

10     }
11
12     pthread_args_t *args_array = malloc(sizeof(pthread_args_t) * cnt_threads);
13
14     if (!args_array)
15     {
16         free(threads);
17
18         return ALLOCATE_ERROR;
19     }
20
21     for (int i = 0; i < cnt_threads; i++)
22     {
23         args_array[i].mult_args = args;
24         args_array[i].tid = i;
25         args_array[i].size = N;
26         args_array[i].cnt_threads = cnt_threads;
27
28         switch (type)
29         {
30             case 1:
31                 pthread_create(&threads[i], NULL, parallel_multiplication1, &args_array[i]);
32                 break;
33             case 2:
34                 pthread_create(&threads[i], NULL, parallel_multiplication2, &args_array[i]);
35                 break;
36         }
37     }
38
39     for (int i = 0; i < cnt_threads; i++)
40     {
41         pthread_join(threads[i], NULL);
42     }
43
44     free(args_array);
45     free(threads);
46
47     return OK;
48 }

```

3.4 Тестирование функций

В таблице 3.1 приведены функциональные тесты для функций, реализующих алгоритмы сортировки. Все тесты пройдены успешно.

Таблица 3.1 – Тестирование функций

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

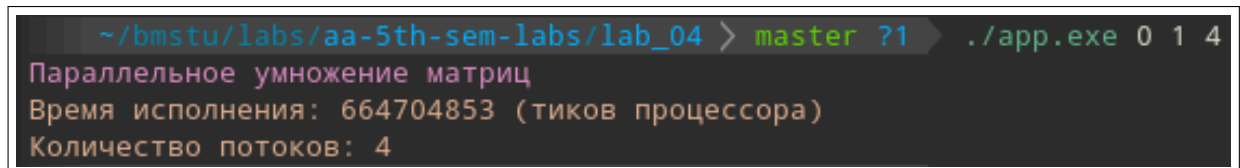
Вывод

Были разработаны и протестированы реализации алгоритмов: простое умножение матриц и распараллеленные версии простого умножения матриц.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.



```
~/bmstu/labs/aa-5th-sem-labs/lab_04 > master ?1 > ./app.exe 0 1 4
Параллельное умножение матриц
Время исполнения: 664704853 (тиков процессора)
Количество потоков: 4
```

Рисунок 4.1 – Демонстрация работы алгоритмов параллельного умножения

Параметры выполнения передаются через аргументы командной строки, где:

- 1-ый аргумент - значение 0 или 1 (генерировать данные случайно или считывать их из файла);
- 2-ой аргумент - значение 1, 2 или 3 (параллельное умножение по первой схеме, параллельное умножение по второй схеме или обычное умножение);
- 3-ий аргумент - количество потоков, на котором запускать программу.

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Manjaro [3] Linux [4] 20.1 64-битная.
- Память: 16 ГБ.
- Процессор: AMD Ryzen™ 7 3700U [5] ЦПУ @ 2.30ГГц

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

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи измерения процессорных тиков до старта работы реализации алгоритма и после завершения. Находилась разность полученных значений и полученная разность бралась за искомый результат.

В листинге 4.1 представлена реализация функции замера процессорных тиков.

Листинг 4.1 – Реализация алгоритмов замеров процессорного времени

```
1 #include "timer.h"
2
3 uint64_t tick(void)
4 {
5     uint32_t high, low;
6     __asm__ __volatile__(
7         "rdtsc\n"
8         "movl_%%edx,_%0\n"
9         "movl_%%eax,_%1\n"
10        : "=r"(high), "=r"(low) : "%rax", "%rbx", "%rcx", "%rdx");
11
12    uint64_t ticks = ((uint64_t)high << 32) | low;
13
14    return ticks;
15 }
```

Результаты замеров приведены в таблицах 4.1 и 4.2.

На рисунке 4.2 приведен график, иллюстрирующий зависимость времени работы алгоритмов параллельного умножения от количества потоков, на которых выполняются реализации.

На рисунке 4.3 приведен график, иллюстрирующий зависимость времени работы алгоритмов простого и параллельного умножения (на 4 потоках) от размеров матриц, на которых выполняются реализации.

Таблица 4.1 – Время работы реализаций алгоритмов параллельного умножения матриц при перемножении квадратных матриц 512×512

Количество потоков	Время умножения, тики	
	Схема 1	Схема 2
1	2438131077	2441008032
2	1240338135	1280537259
4	694732365	727681774
8	592549230	608055922
16	563097799	581367205
32	557056596	566872352

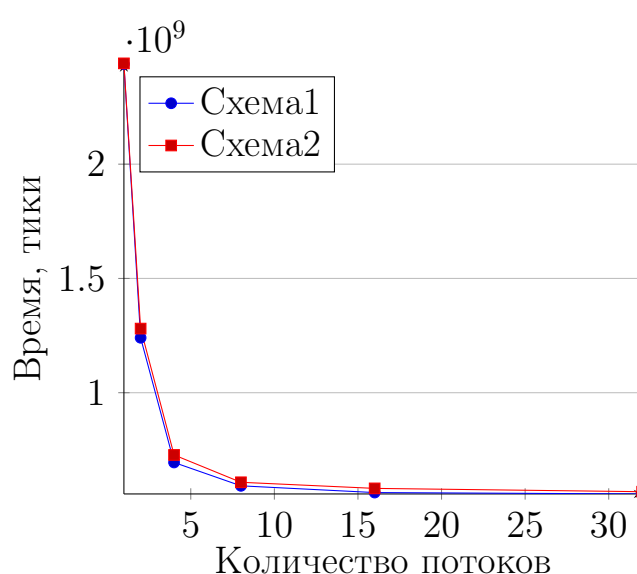


Рисунок 4.2 – Зависимость времени работы реализаций алгоритмов параллельного умножения матриц от количества потоков

Таблица 4.2 – Время работы реализаций алгоритмов простого и параллельного (на 4 потоках) умножения матриц при перемножении квадратных матриц

Размер	Время умножения, тики		
	Обычное умножение	Схема 1	Схема 2
64	7068061	2922587	2874701
128	57752540	19797411	19442130
256	266817411	135553973	139564874
512	2056951617	666832284	709346864
1024	15309810643	3627899434	3603030937

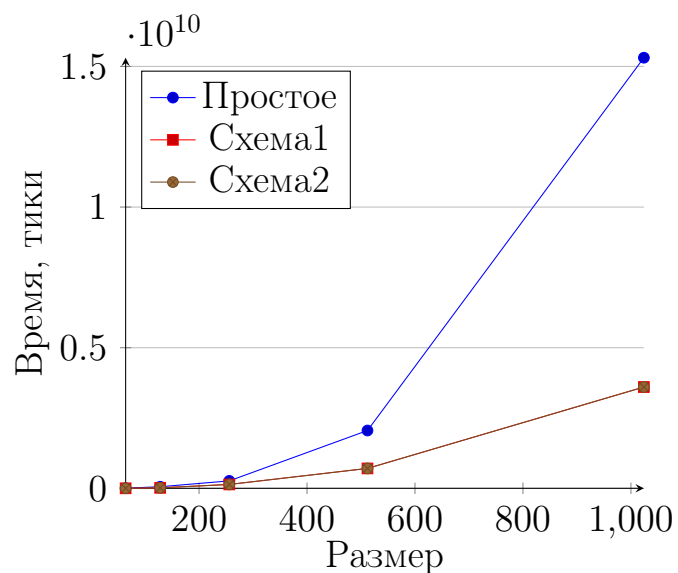


Рисунок 4.3 – Зависимость времени работы реализаций алгоритмов умножения матриц от размера матриц

Вывод

Наилучшее время параллельные алгоритмы показали на 8 потоках, что соответствует количеству логических ядер компьютера, на котором проводилось тестирование. На матрицах размером 512 на 512, параллельные алгоритмы улучшают время однопоточной реализации перемножения матриц в 4 раза. При количестве потоков, большее чем 8, параллельная реализация замедляет выполнение в сравнении с 8 потоками.

Заключение

В ходе выполнения работы была достигнута цель выполнены все поставленные задачи:

- изучить понятие параллельных вычислений;
- реализовать последовательную и две параллельных реализации алгоритма перемножения матриц;
- сравнить временные характеристики реализованных алгоритмов экспериментально.

Экспериментально были установлены различия в производительности реализаций параллельного и обычного алгоритмов умножения матриц. Параллельные реализации алгоритмов выигрывают по скорости у однопоточной реализации перемножения двух матриц. Наиболее эффективны данные алгоритмы при количестве потоков, совпадающем с количеством логических ядер компьютера. Так, например, на матрицах размером 512 на 512, удалось улучшить время выполнения алгоритма умножения матриц в 4 раза в сравнении с однопоточной реализацией.

Литература

- [1] C (programming language) – Wikipedia [Электронный ресурс]. Режим доступа: [https://en.wikipedia.org/wiki/C_\(programming_language\)](https://en.wikipedia.org/wiki/C_(programming_language)) (дата обращения: 29.10.2020).
- [2] The Go Programming Language [Электронный ресурс]. Режим доступа: <https://golang.org/> (дата обращения: 11.09.2020).
- [3] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: <https://manjaro.org/> (дата обращения: 14.09.2020).
- [4] Linux – Википедия [Электронный ресурс]. Режим доступа: <https://ru.wikipedia.org/wiki/Linux> (дата обращения: 14.09.2020).
- [5] Мобильный процессор AMD Ryzen™ 7 3700U с графикой Radeon™ RX Vega 10 [Электронный ресурс]. Режим доступа: <https://www.amd.com/ru/products/apu/amd-ryzen-7-3700u> (дата обращения: 14.09.2020).