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

по курсу «Анализ Алгоритмов»

на тему: «Параллельное умножение матриц»

Студент ИУ7-53Б _____ Миронов Г. А.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Волкова Л. Л.
(Подпись, дата) (И. О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

Введение

В данной лабораторной работе будут рассмотрены параллельные вычисления на примере умножения матриц.

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.

Целью данной работы является реализация и изучение следующих алгоритмов:

- обычное умножение матриц по строкам;
- параллельное умножение матриц по строкам;
- параллельное умножение матриц по столбцам.

Для достижения данной цели необходимо решить следующие задачи:

- изучить основные методы параллельных вычислений;
- реализовать каждый из указанных алгоритмов умножения матриц;
- сравнить временные характеристики реализованных алгоритмов экспериментально.

1 Аналитическая часть

1.1 Некоторые теоретические сведения

Для начала нужно ввести собственно понятие матрицы.

Матрица – объект, записываемый в виде прямоугольной таблицы элементов, которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы (формула 1.1).

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \tag{1.1}$$

Произведение матриц AB состоит из всех возможных комбинаций скалярных произведений вектор-строк матрицы A и вектор-столбцов матрицы B (рис. 1.1).

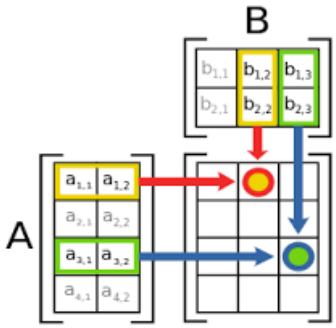

Рисунок 1.1 – – Произведение матриц

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первой матрице равно числу строк во второй.

1.2 Стандартный алгоритм умножения матриц

Пусть даны матрицы A (формула 1.1) размерностью $n \times m$ и B (формула 1.2) $m \times q$.

$$B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1q} \\ b_{21} & b_{22} & \dots & b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mq} \end{pmatrix} \tag{1.2}$$

Матрица $C = AB$ будет размерностью $n \times q$. Тогда каждый элемент матрицы C выражается формулой (1.3).

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \quad (i = 1, 2, \ldots n; j = 1, 2, \ldots q) \tag{1.3}$$

1.3 Параллельный алгоритм умножения

Так как каждый элемент матрицы C вычисляется независимо от других [1] и матрицы A и B не изменяются, для параллельного вычисления произведения достаточно равным образом распределить вычисление элементов матрицы C между потоками.

В связи с аппаратными ограничениями, производить данные вычисления для каждого элемента результирующей матрицы в отдельности не эффективно. Решением данной проблемы является группировка элементов результирующей матрицы по строкам или столбцам и параллельное вычисление результатов для каждой из данных групп.

1.4 Вывод

Стандартный алгоритм умножения матриц вычисляет элементы результирующей матрицы независимо друг от друга, что позволяет реализовать параллеьный вариант алгоритма.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы алгоритмов умножения матриц и модель вычислений.

2.1 Схемы

На рисунке 2.1 представлена схема стандартного алгоритма умножения матриц.

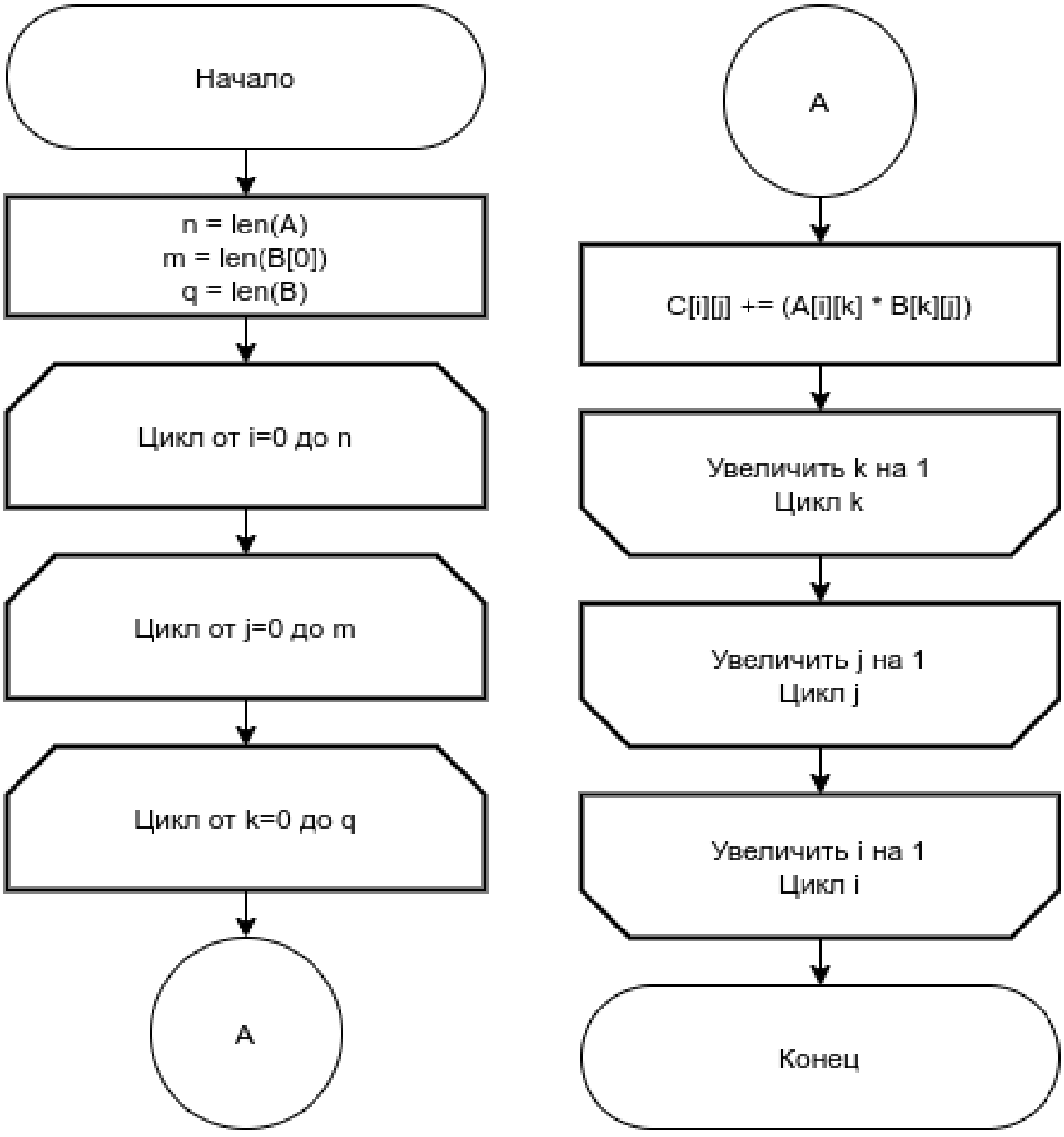

Рисунок 2.1 – Схема стандартного алгоритма умножения матриц

На рисунках 2.2 – 2.3 представлены схемы параллельных алгоритмов умножения матриц.

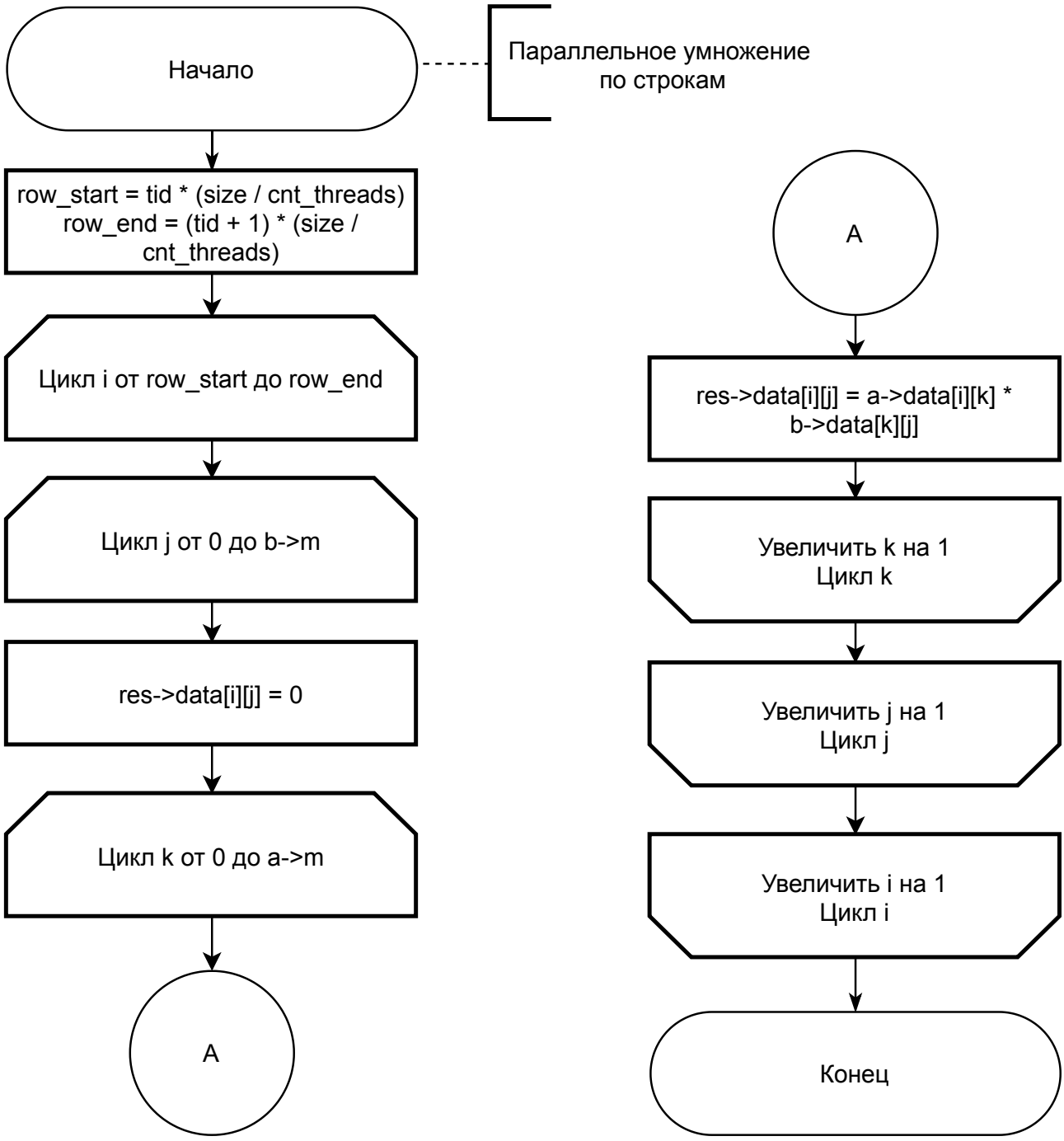

Рисунок 2.2 – Схема параллельного алгоритма по строкам

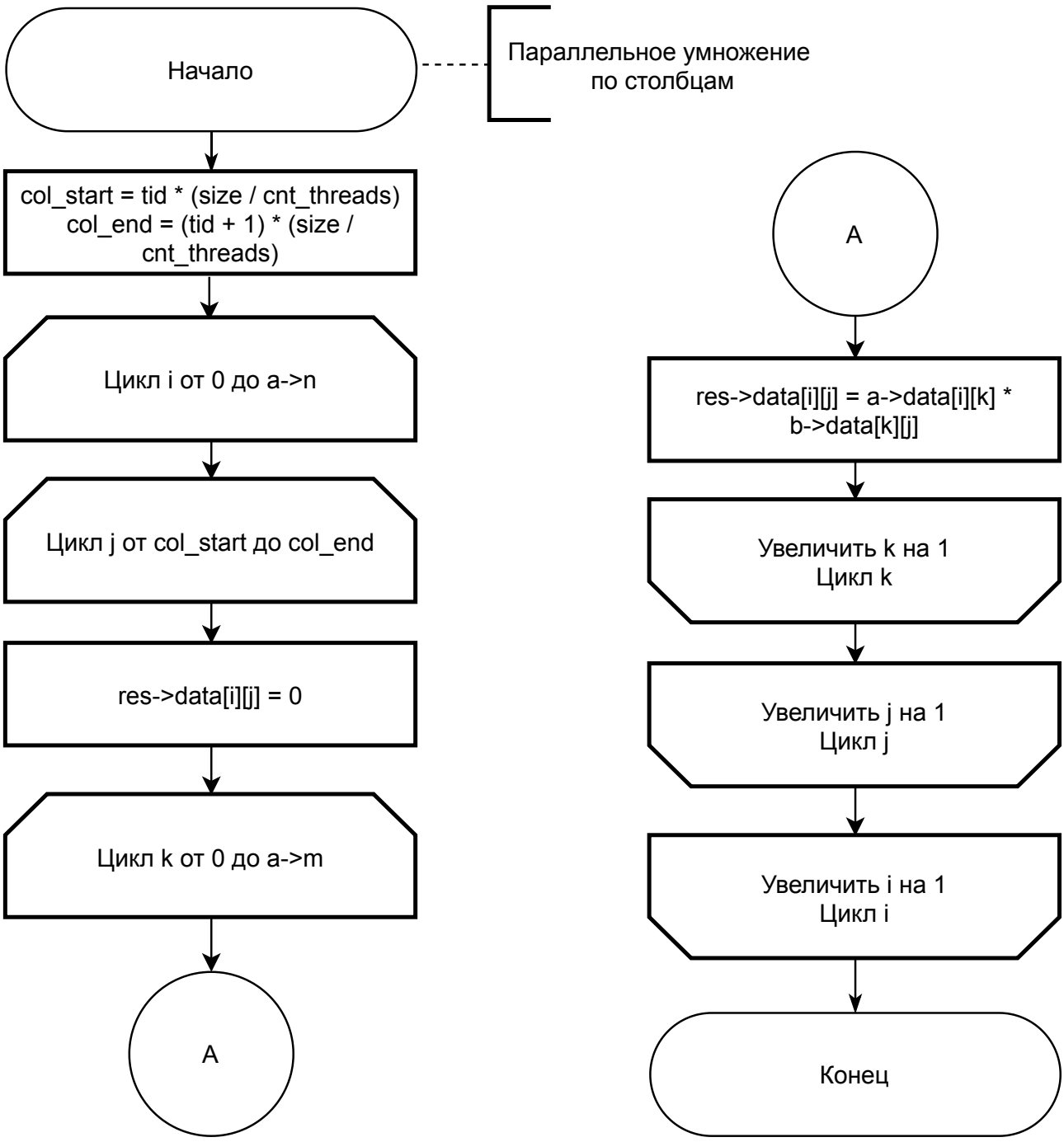

Рисунок 2.3 – Схема параллельного алгоритма по столбцам

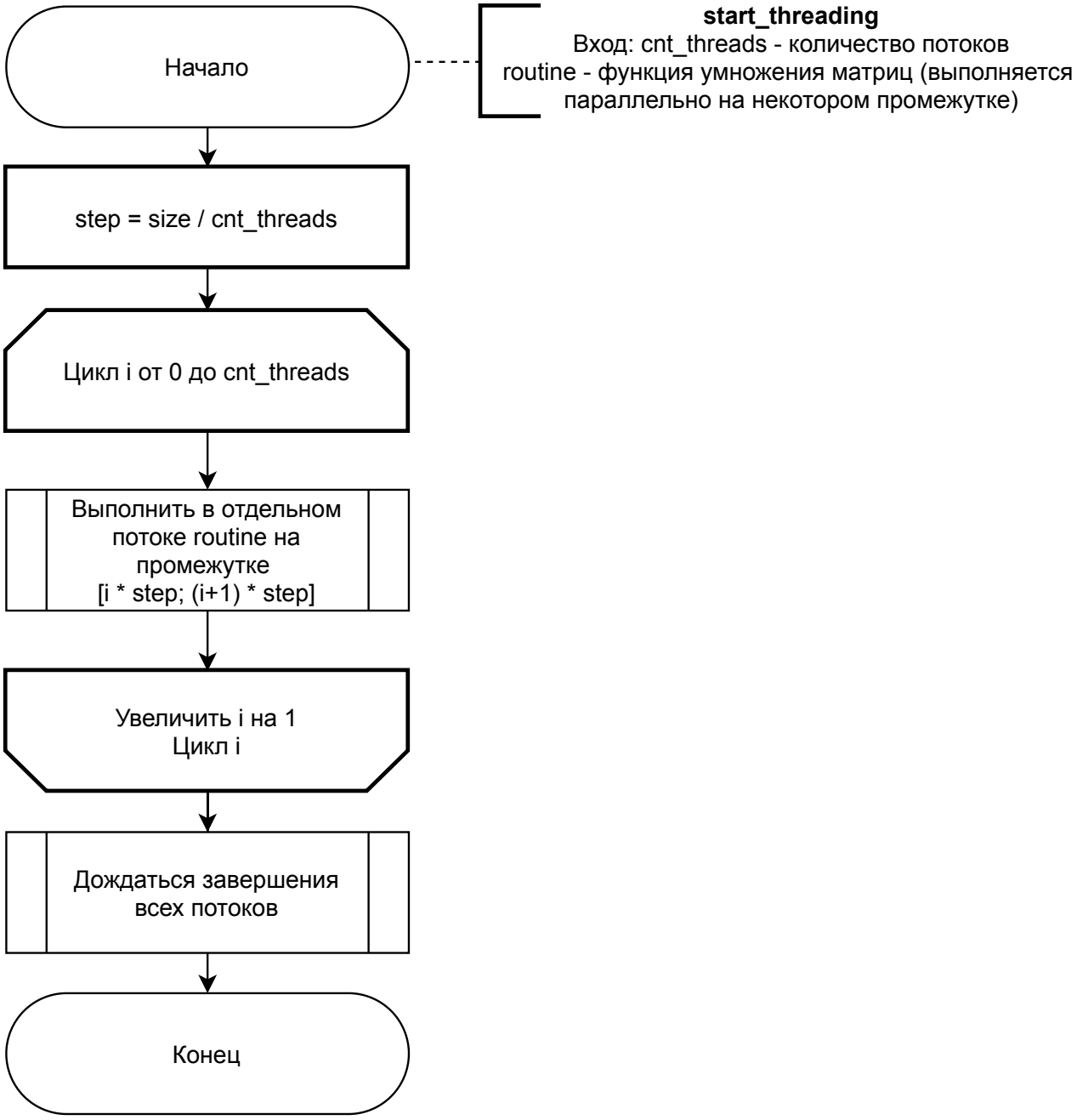

Рисунок 2.4 – Схема функции создания потоков

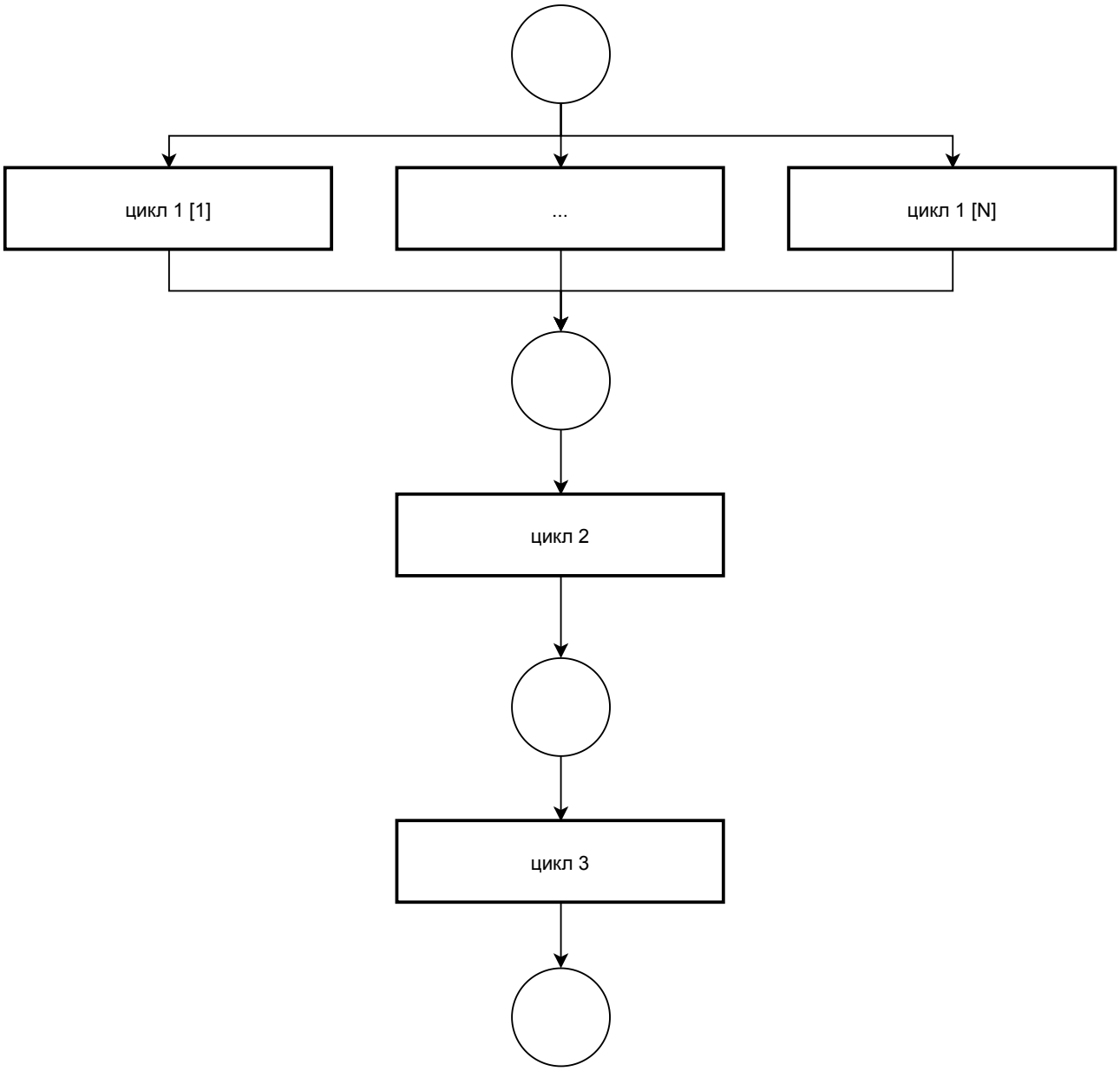

Рисунок 2.5 – Схема с параллельным выполнением первого цикла

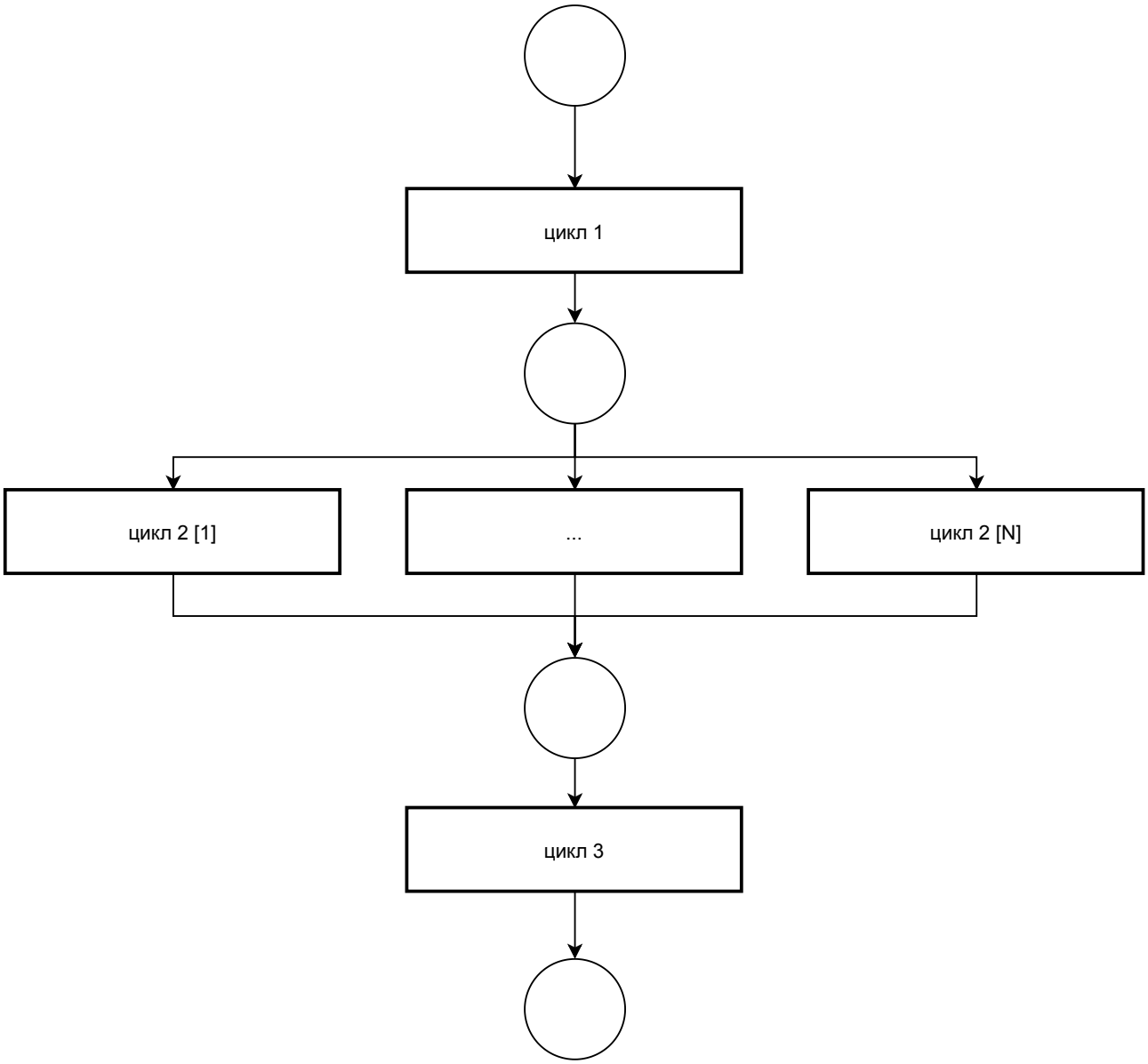

Рисунок 2.6 – Схема с параллельным выполнением второго цикла

2.2 Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема стандартного алгоритма умножения матриц (рис. 2.1),а так же были построены схемы двух вариантов параллельного выполнения данного алгоритма (рис. 2.4 – 2.2).

3 Технологическая часть

3.1 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык C [2]. Данный выбор обусловлен тем, что я имею некоторый опыт разработки на нем, а так же поддержкой данным языком нативных потоков посредством использования библиотеки pthreads [3].

3.2 Требования к программному обеспечению

Входными данными являются две матрицы A и B. Количество столбцов матрицы A должно быть равно количеству строк матрицы B.

На выходе получается результат умножения введенных пользователем матриц.

3.3 Сведения о модулях программы

Данная программа разбита на модули:

- `main.c` - файл, содержащий точку входа в программу;
- `matrix.h` - файл, содержащий определение структуры матрицы;
- `matrix.c` - файл, содержащий реализации основных функций для работы с матрицами;
- `process.c` - файл содержащий логику работы приложения. В этом файле происходит общение с пользователем и вызов алгоритмов;
- `parallel_process.c` - файл содержащий логику работы в параллельном режиме. В этом файле происходят замеры временных характеристик для алгоритмов с разными входными данными и количеством потоков;
- `multiplication.c` - файл, содержащий реализацию простого алгоритма умножения матриц;
- `parallel_multiplication.c` - файл, содержащий реализации параллельных алгоритмов умножения матриц;

- `threads.h` - файл, содержащий определение структуры аргументов, передаваемых функциям реализующим параллельные алгоритмы;
- `threads.c` - файл, содержащий реализацию функции распараллеливания вычислений;
- `timer.c` - файл, содержащий реализацию функции вычисления текущего количества тиков;
- `matrix_io.c` - файл, содержащий реализации различных функций ввода-вывода матриц;
- `io.c` - файл, содержащий реализации различных функций ввода-вывода;

На листингах 3.1 – 3.12 представлен код программы.

Листинг 3.1 – Основной файл программы main

```
1  #include <parallel_process.h>
2  #include <process.h>
3
4  int main(void) {
5  #ifdef __FILE_READING__
6      return process_file();
7  #else
8      return process_stdin();
9  #endif
10 }
```

Листинг 3.2 – Определение структур и методов для работы с матрицами matrix.h

```
1  typedef struct {
2      int64_t **data;
3      size_t n, m;
4  } matrix_t;
5
6  #define INIT_MATR_PTR(__ptr__)                                          \
7      do {                                                                \
8          if (!(__ptr__ = (matrix_t *) malloc(sizeof(matrix_t)))) \
               return ALLOCATION_ERROR; \
9          __ptr__->data = NULL;                                           \
10         __ptr__->n = __ptr__->m = 0;                                    \
11     } while (0)
12
13 typedef struct {
14     matrix_t *a;
15     matrix_t *b;
16     matrix_t *res;
17 } args_t;
18
19 uint8_t create_matrix(matrix_t *matrix, size_t n, size_t m);
20
21 void free_matrix(matrix_t *matrix);
22
23 void free_matrixes(size_t count, ...);
```

Листинг 3.3 – Реализация методов для работы с матрицами matrix.c

```
1  uint8_t create_matrix(matrix_t *matrix, size_t n, size_t m) {
2      if (!matrix) return NULL_PTR_ERROR;
3
4      matrix_t *temp = (matrix_t *) malloc(sizeof(matrix_t));
5      if (!temp) return ALLOCATION_ERROR;
6
7      temp->n = temp->m = 0;
8      if (!(temp->data = (int64_t **) malloc(n * sizeof(int64_t *))
           )) return ALLOCATION_ERROR;
9
10     for (size_t i = 0; i < n; ++i) {
11         if ((temp->data[i] = (int64_t *) malloc(m * sizeof(
               int64_t)))) continue;
12
13         free_matrix(temp);
14         return ALLOCATION_ERROR;
15     }
16
17     temp->n = n;
18     temp->m = m;
19
20     if (matrix->data) free_matrix(matrix);
21     memcpy(matrix, temp, sizeof(matrix_t));
22
23     return OK;
24 }
25
26 void free_matrix(matrix_t *matrix) {
27     if (!matrix || !(matrix->data)) return;
28
29     for (size_t i = 0; i < matrix->n; ++i)
30         if (matrix->data[i])
31             free(matrix->data[i]);
32
33     free(matrix->data);
34
35     matrix->data = NULL;
36     matrix->n = matrix->m = 0;
37 }
38
39 void free_matrixes(size_t count, ...) {
40     va_list ap;
41     va_start(ap, count);
42     for (size_t j = 0; j < count; j++)
43         free_matrix(va_arg(ap, matrix_t *));
44     va_end(ap);
45 }
```

Листинг 3.4 – Реализация интерактивного процесса process.c

```
#define THREAD_COUNT 1

uint8_t process_stdin() {
    setbuf(stdout, NULL);

    print_main_prompt();

    matrix_t *a, *b;
    INIT_MATR_PTR(a);
    INIT_MATR_PTR(b);

    uint8_t rc = get_from_stdin(a, b);
    if (rc) return rc;

    matrix_t *c;
    INIT_MATR_PTR(c);
    args_t args = {.a = a, .b = b, .res = c};

    uint64_t ticks = 0;
    if ((rc = base_multiplication(&args, &ticks))) return rc;
    print_multiplication_results(c, ticks, 1);

    if ((rc = start_threading(&args, THREAD_COUNT,
       parallel_multiplication_by_rows, &ticks))) return rc;
    print_multiplication_results(c, ticks, 1);

    if ((rc = start_threading(&args, THREAD_COUNT,
       parallel_multiplication_by_cols, &ticks))) return rc;
    print_multiplication_results(c, ticks, 1);

    free_matrixes(3, a, b, c);

    return rc;
}
```

Листинг 3.5 – Реализация замеров временных характеристик для разных алгоритмов parallel_process.c

```
1  #define THREAD_COUNT 2
2
3  static routine_t routines[] = {
4          parallel_multiplication_by_rows,
5          parallel_multiplication_by_cols,
6          NULL};
7
8  static inline uint8_t process(const char *filename, args_t *args)
       {
9      if (!filename || !args) return NULL_PTR_ERROR;
10
11     uint8_t rc = OK;
12     if ((rc = get_from_file(filename, args->a, args->b))) return
           rc;
13
14     uint64_t ticks = 0;
15     if ((rc = base_multiplication(args, &ticks))) return rc;
16     print_multiplication_results(args->res, ticks, 0);
17
18     for (size_t j = 1; j <= THREAD_COUNT; j *= 2) {
19         printf("%zu␣threads:\n", j);
20         for (size_t k = 0; routines[k]; ++k) {
21             if ((rc = start_threading(args, j, routines[k], &
                   ticks))) return rc;
22             print_multiplication_results(args->res, ticks, 0);
23         }
24     }
25
26     return rc;
27 }
28
29 uint8_t process_file() {
30     setbuf(stdout, NULL);
31     print_main_prompt();
32
33     matrix_t *a, *b, *c;
34     INIT_MATR_PTR(a);
35     INIT_MATR_PTR(b);
36     INIT_MATR_PTR(c);
37
38     args_t args = {.a = a, .b = b, .res = c};
39     for (size_t i = 0, rc = 0; files[i]; ++i) {
40         printf("\n%s\n", files[i]);
41         if ((rc = process(files[i], &args))) return rc;
42     }
43     free_matrixes(3, a, b, c);
44
45     return OK;
46 }
```

Листинг 3.6 – Определения структур аргументов для параллельных реализаций алгоритмов threads.h

```
1  #pragma once
2
3  #include <common.h>
4  #include <matrix.h>
5
6  typedef struct {
7      args_t *mult_args;
8      size_t tid;
9      size_t size;
10     size_t cnt_threads;
11 } pthread_args_t;
12
13 typedef void *(*routine_t)(void *);
14
15 uint8_t start_threading(args_t *args, const size_t cnt_threads,
       routine_t routine, uint64_t *ticks);
```

Листинг 3.7 – Реализация распараллеливания алгоритмов thread.c

```
1  uint8_t start_threading(args_t *args, const size_t cnt_threads,
       routine_t routine, uint64_t *ticks) {
2      uint8_t rc = OK;
3      if ((rc = create_matrix(args->res, args->a->n, args->b->m)))
           return rc;
4
5      pthread_t *threads = (pthread_t *) malloc(cnt_threads *
           sizeof(pthread_t));
6      if (!threads) return ALLOCATION_ERROR;
7
8      pthread_args_t *args_array = malloc(sizeof(pthread_args_t) *
           cnt_threads);
9      if (!args_array) {
10         free(threads);
11         return ALLOCATION_ERROR;
12     }
13
14     for (size_t i = 0; i < cnt_threads; i++) {
15         args_array[i].mult_args = args;
16         args_array[i].tid = i;
17         args_array[i].size = args->a->n;
18         args_array[i].cnt_threads = cnt_threads;
19     }
20
21     *ticks = 0;
22 #ifdef __TICKS_COUNT__
23     uint64_t total_ticks = 0;
24     for (int c = 0; c < REPEATS_COUNT; ++c) {
25         uint64_t start = tick();
26 #endif
27         for (size_t i = 0; i < cnt_threads; i++)
28             pthread_create(&threads[i], NULL, routine, &
                   args_array[i]);
29
30         for (size_t i = 0; i < cnt_threads; i++)
31             pthread_join(threads[i], NULL);
32 #ifdef __TICKS_COUNT__
33         uint64_t end = tick();
34         total_ticks += end - start;
35     }
36     *ticks = total_ticks / REPEATS_COUNT;
37 #endif
38
39     free(args_array);
40     free(threads);
41
42     return OK;
43 }
```

Листинг 3.8 – Реализация стандартного алгоритма умножения

```
1  static inline uint8_t check_args(args_t *args) {
2      return (args && args->a && args->b && args->res) ? OK :
           NULL_PTR_ERROR;
3  }
4
5  uint8_t base_multiplication(args_t *args, uint64_t *ticks) {
6      uint8_t rc = check_args(args);
7      if (rc) return rc;
8
9      matrix_t *a = args->a, *b = args->b, *res = args->res;
10     if ((rc = create_matrix(res, a->n, b->m))) return rc;
11
12     *ticks = 0;
13 #ifdef __TICKS_COUNT__
14     uint64_t total_ticks = 0;
15     for (int c = 0; c < REPEATS_COUNT; ++c) {
16         uint64_t start = tick();
17 #endif
18         for (int i = 0; i < a->n; i++) {
19             for (int j = 0; j < b->m; j++) {
20                 res->data[i][j] = 0;
21                 for (int k = 0; k < a->m; k++) {
22                     res->data[i][j] += a->data[i][k] * b->data[k
                           ][j];
23                 }
24             }
25         }
26 #ifdef __TICKS_COUNT__
27         uint64_t end = tick();
28         total_ticks += end - start;
29     }
30     *ticks = total_ticks / REPEATS_COUNT;
31 #endif
32
33     return rc;
34 }
```

Листинг 3.9 – Реализация функции вычисления текущего просеррного времени timer.c

```
1  inline uint64_t tick(void) {
2      return __rdtsc();
3  }
```

Листинг 3.10 – Реализация параллельных алгоритмов умножения

```
1  void *parallel_multiplication_by_rows(void *args) {
2      pthread_args_t *argsp = (pthread_args_t *) args;
3
4      int row_start = argsp->tid * (argsp->size / argsp->
           cnt_threads);
5      int row_end = (argsp->tid + 1) * (argsp->size / argsp->
           cnt_threads);
6
7      matrix_t *a = argsp->mult_args->a, *b = argsp->mult_args->b,
           *res = argsp->mult_args->res;
8      for (int i = row_start; i < row_end; i++) {
9          for (int j = 0; j < b->m; j++) {
10             res->data[i][j] = 0;
11             for (int k = 0; k < a->m; k++) {
12                 res->data[i][j] += a->data[i][k] * b->data[k][j];
13             }
14         }
15     }
16
17     return NULL;
18 }
19
20 void *parallel_multiplication_by_cols(void *args) {
21     pthread_args_t *argsp = (pthread_args_t *) args;
22
23     int col_start = argsp->tid * (argsp->size / argsp->
           cnt_threads);
24     int col_end = (argsp->tid + 1) * (argsp->size / argsp->
           cnt_threads);
25
26     matrix_t *a = argsp->mult_args->a, *b = argsp->mult_args->b,
           *res = argsp->mult_args->res;
27     for (int i = 0; i < a->n; i++) {
28         for (int j = col_start; j < col_end; j++) {
29             res->data[i][j] = 0;
30             for (int k = 0; k < a->m; k++) {
31                 res->data[i][j] += a->data[i][k] * b->data[k][j];
32             }
33         }
34     }
35
36     return NULL;
37 }
```

Листинг 3.11 – Реализация функций ввода-вывода io.c

```
1  #define ANSI_COLOR_YELLOW "\x1b[33m"
2  #define ANSI_COLOR_CYAN "\x1b[36m"
3  #define ANSI_COLOR_RESET "\x1b[0m"
4
5  inline void print_main_prompt() {
6      printf(ANSI_COLOR_CYAN "\nMATRIX␣MULTIPLICATION\n\n"
          ANSI_COLOR_RESET);
7  }
8
9  inline uint8_t get_from_stdin(matrix_t *const a, matrix_t *const
       b) {
10     uint8_t rc = OK;
11
12     printf(ANSI_COLOR_YELLOW "Please,␣input␣first␣matrix:\n"
          ANSI_COLOR_RESET);
13     if ((rc = read_matrix(stdin, a, 1))) return rc;
14
15     printf(ANSI_COLOR_YELLOW "Please,␣input␣second␣matrix:\n"
          ANSI_COLOR_RESET);
16     if ((rc = read_matrix(stdin, b, 1))) return rc;
17
18     return (a->m != b->n) ? INCORRECT_MATR_SIZES : rc;
19 }
20
21 inline uint8_t get_from_file(const char *const f, matrix_t *const
       a, matrix_t *const b) {
22     FILE *fin = fopen(f, "r");
23     if (!fin) return FILE_OPEN_ERROR;
24
25     uint8_t rc = OK;
26     if ((rc = read_matrix(fin, a, 0))) {
27         fclose(fin);
28         return rc;
29     }
30
31     if ((rc = read_matrix(fin, b, 0))) {
32         fclose(fin);
33         return rc;
34     }
35
36     fclose(fin);
37     return (a->m != b->n) ? INCORRECT_MATR_SIZES : rc;
38 }
39
40 inline void print_multiplication_results(matrix_t *res, const
       uint64_t ticks, int8_t printable_res) {
41     printf(ANSI_COLOR_YELLOW "Execution␣time:␣%" PRIu64 "␣(cpu␣
          ticks)\n" ANSI_COLOR_RESET, ticks);
42
43     if (printable_res) print_matrix(res);
44 }
```

Листинг 3.12 – Реализация функций ввода-вывода для матриц matrix_io.c

```
1  #define INPUT_ROWS_PROMPT "Please, input rows count: "
2  #define INPUT_COLS_PROMPT "Please, input cols count: "
3  #define INPUT_DATA_PROMPT "Please, input matrix data:\n"
4
5  inline static uint8_t input_matrix_dim(FILE *fin, const char *msg
       , size_t *value) {
6      if (msg) printf("%s", msg);
7      return (1 != fscanf(fin, "%zu", value)) ? INPUT_ERROR : OK;
8  }
9
10 inline static uint8_t input_matrix_data(FILE *fin, const char *
       msg, matrix_t *matr) {
11     if (msg) printf("%s", msg);
12     for (size_t i = 0; i < matr->n; ++i)
13         for (size_t j = 0; j < matr->m; ++j)
14             if (1 != fscanf(fin, "%" PRId64, &(matr->data[i][j]))
                   )
15                 return INPUT_ERROR;
16     return OK;
17 }
18
19 uint8_t read_matrix(FILE *fin, matrix_t *matrix, int8_t prompt) {
20     if (!matrix) return NULL_PTR_ERROR;
21
22     uint8_t rc = OK;
23
24     size_t n, m;
25     if ((rc = input_matrix_dim(fin, (prompt) ? INPUT_ROWS_PROMPT
           : NULL, &n))) return rc;
26     if ((rc = input_matrix_dim(fin, (prompt) ? INPUT_COLS_PROMPT
           : NULL, &m))) return rc;
27
28     matrix_t *temp;
29     INIT_MATR_PTR(temp);
30
31     if ((rc = create_matrix(temp, n, m))) return rc;
32     if ((rc = input_matrix_data(fin, (prompt) ? INPUT_DATA_PROMPT
           : NULL, temp))) return rc;
33
34     free_matrix(matrix);
35     memcpy(matrix, temp, sizeof(matrix_t));
36
37     return OK;
38 }
39
40 void print_matrix(matrix_t *matrix) {
41     for (size_t i = 0; i < matrix->n; ++i) {
42         for (size_t j = 0; j < matrix->m; ++j)
43             printf("%" PRId64 " ", matrix->data[i][j]);
44         printf("\n");
45     }
46     printf("\n");
47 }
```

3.4 Вывод

В данном разделе были реализованы вышеописанные алгоритмы.

Было разработано программное обеспечение, удовлетворяющее предъявляемым требованиям. Так же были представлены соответствующие листинги 3.1 – 3.12 с кодом программы.

4 Экспериментальная часть

В данном разделе будет проведено функциональное тестирование разработанного программного обеспечения. Так же будет произведено измерение временных характеристик каждого из реализованных алгоритмов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- Процессор: Intel Core™ i5-8250U [4] CPU @ 1.60GHz.
- Память: 32 GiB.
- Операционная система: Manjaro [5] Linux [6] 21.1.4 64-bit.

Исследование 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.

4.2 Тестирование

В данном разделе будет приведена таблица с тестами (таблица 4.1).

Таблица 4.1 – Таблица тестов

Первая матрица	Вторая матрица	Ожидаемый результат
$\begin{pmatrix}2\end{pmatrix}$	$\begin{pmatrix}2\end{pmatrix}$	$\begin{pmatrix}4\end{pmatrix}$
$\begin{pmatrix}2\\1\end{pmatrix}$	$\begin{pmatrix}1 & 2\end{pmatrix}$	$\begin{pmatrix}2 & 4\\1 & 2\end{pmatrix}$
$\begin{pmatrix}1 & 2 & 2\\1 & 2 & 2\end{pmatrix}$	$\begin{pmatrix}1 & 2\\1 & 2\\1 & 2\end{pmatrix}$	$\begin{pmatrix}5 & 10\\5 & 10\end{pmatrix}$
$\begin{pmatrix}1 & -2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{pmatrix}$	$\begin{pmatrix}-1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{pmatrix}$	$\begin{pmatrix}0 & 4 & 6\\4 & 12 & 18\\4 & 12 & 18\end{pmatrix}$

При проведении функционального тестирования, полученные результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

4.3 Временные характеристики

Для сравнения возьмем квадратные матрицы размерностью [32, 64, 128, ..., 1024]. Количество потоков будем брать из набора [1, 2, 4, 8, ..., $4 * M$] где M - количество логических ядер используемой ЭВМ.

Так как в общем случае вычисление произведения матриц является достаточно короткой задачей, воспользуемся усреднением массового эксперимента. Для этого вычислим среднее арифметическое число тактов процессора, затраченных на выполнение алгоритма, для n запусков. Сравнение произведем при $n = 100$.

Результаты замеров по результатам экспериментов для умножения по строкам приведены в Таблице 4.2.

Таблица 4.2 – Замер времени для матриц, размером от 32 до 1024 элементов

Размерность матрицы, эл.	**Количество потоков, ед**					
	1	**2**	**4**	**8**	**16**	**32**
32	3.3e+05	4.5e+05	3.5e+05	3e+05	3.3e+05	5.8e+05
64	2.4e+06	9.7e+06	5.7e+06	4.3e+06	2.1e+06	2.2e+06
128	2.1e+07	2.1e+07	1.1e+07	9e+06	6.3e+06	5.8e+06
256	1.8e+08	1.8e+08	9e+07	7.6e+07	4.5e+07	4.6e+07
512	1.6e+09	1.6e+09	8.1e+08	5.2e+08	4.8e+08	4.8e+08
1024	2e+10	2e+10	1.1e+10	5.3e+09	5.5e+09	4.9e+09

Результаты замеров по результатам экспериментов для умножения по столбцам приведены в Таблице 4.3.

Таблица 4.3 – Замер времени для матриц, размером от 32 до 1024 элементов

Размерность матрицы, эл.	**Количество потоков, ед**					
	1	**2**	**4**	**8**	**16**	**32**
32	4.3e+05	3.4e+05	2.8e+05	3.4e+05	5.6e+05	1.1e+06
64	4.3e+06	5.5e+06	4.8e+06	2.3e+06	2.1e+06	2.8e+06
128	2.1e+07	1.2e+07	9.2e+06	6.7e+06	5.9e+06	6.4e+06
256	1.8e+08	9.1e+07	7.1e+07	4.5e+07	4.6e+07	5.4e+07
512	1.6e+09	8.1e+08	4.7e+08	4.5e+08	4.6e+08	4.6e+08
1024	2e+10	1.1e+10	5.6e+09	5e+09	4.9e+09	4.9e+09

На Рисунке 4.1 отображены временные харкатеристики параллельных алгоритмов при размерностях квадратных матриц равных 1024 элемента.

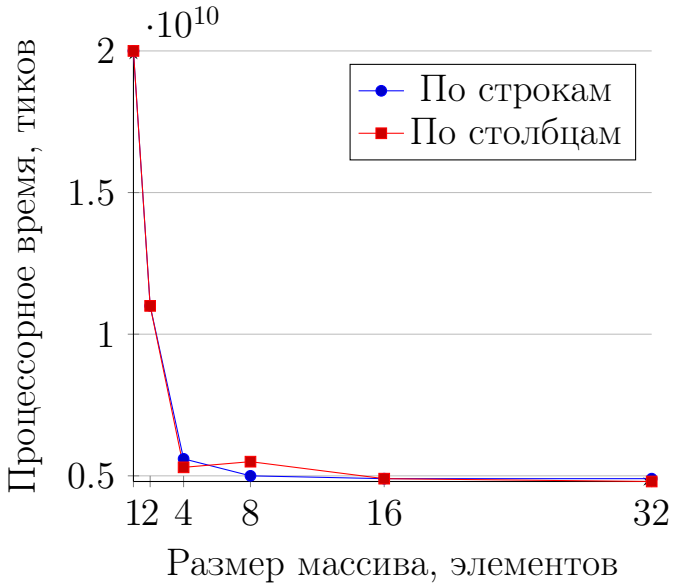

Рисунок 4.1 – Временные характеристики на разном количестве потоков при матрицах размером 1024x1024

Из Рисунка 4.1 следует, что наиболее эффективными параллельные алгоритмы становятся при приближении числа потоков к количеству логических ядер используемой ЭВМ (8 для использованной в ходе эксперимента). На Рисунке 4.2 приведены временные характеристики параллельных алгоритмов для числа потоков из набора [8, 16, 32].

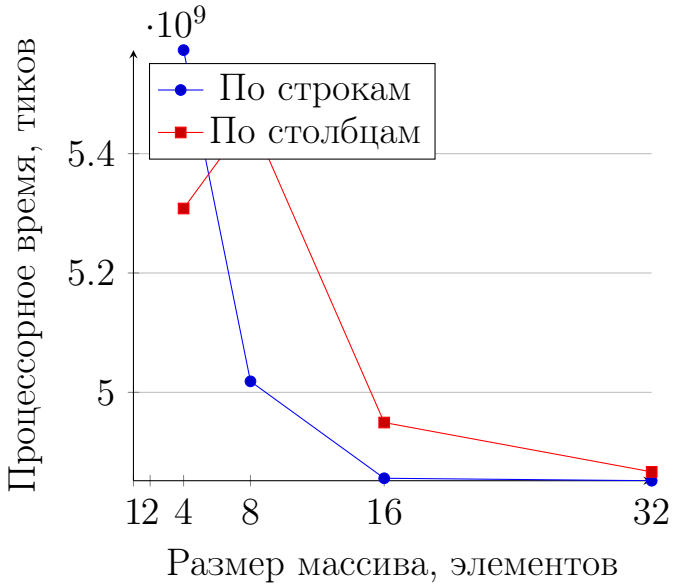

Рисунок 4.2 – Временные характеристики на разном количестве потоков при матрицах размером 1024x1024

Из Рисунка 4.2 можно сделать вывод, что наиболее эффективны параллельные алгоритмы при количестве потоков, 16 (Удвоенное число логических ядердля использованной в ходе эксперимента ЭВМ). Объяснить это можно особенностью реализации многопоточности в конкретной системе (например, реализации `pthreads.h` [3] через так называемые `Lightweight Threads (LWT)`).

В общем случае, наиболее эффективным будет являться конфигурация с числом потоков равным числу логических ядер используемов ЭВМ.

Сравним временные характеристики стандартного и параллельных алгоритмов при количестве потоков равном 8 (число логических ядер использованной ЭВМ) при разных размерностях квадратных матриц. Данные для сравнения получены из Таблиц 4.2 и 4.3.

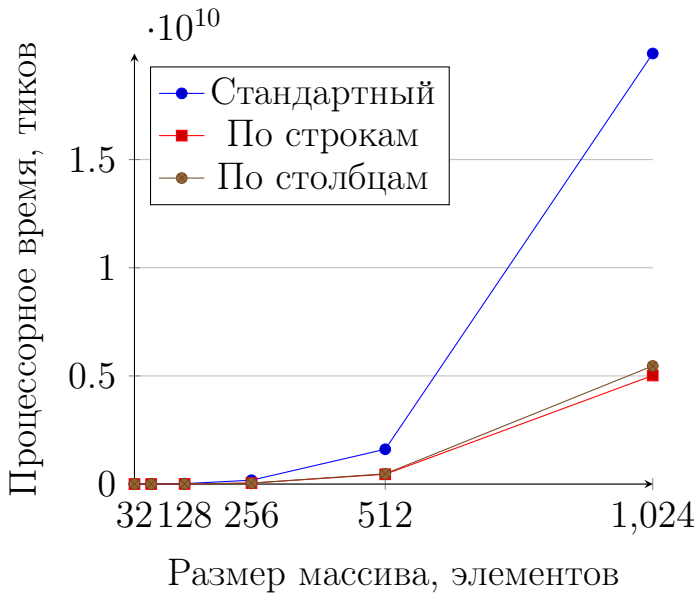

Рисунок 4.3 – Временные характеристики реализованных алгоритмов при количестве потоков равном 8

Как и ожидалось, параллельные алгоритмы оказались более эффективными по времени, чем стандартный алгоритм. При этом, параллеьльный алгоритм по строкам является наиболее эффективным из рассмотренных. Это связано с аппаратными особенностями работы с памятью - при умножении по столбцам на каждой итерации 2 цикла происходит обращение к новой области памяти, что приводит к большим временным затратам. В алгоритме умножения по строкам область памяти меняется только в первом цикле, поэтому он является более эффективным.

Однако, при работе с матрицами малых размерностей (менее 64 элементов) стандартный алгоритм оказался более эффективным (в 4 раза в сравнении с 32 поточной параллельной реализацией) по времени, так как параллельные реализации требуют дополнительных затрат по времени и памятидля реализацию многопоточности (создание потоков, реализация совместного доступа к ресурсам).

4.4 Вывод

В данном разделе было произведено сравнение количества затраченного времени вышеизложенных алгоритмов.

Наиболее эффективной по времени при работе с матрицами больших размерностей (более 64 элементов) оказалась параллельная реализация на 16 потоках, что может быть обусловлено особенностью конкретной реализации библиотеки `pthreads` [3] для данной операционной системы. В общем случае, наиболее эффективной является реализация с числом потоков равным числу логических ядер используемой ЭВМ.

При работе с матрицами малых размерностей (менее 64 элементов) стандартный алгоритм оказался более эффективным (в 4 раза в сравнении с 32 поточной параллельной реализацией) по времени, что связано с дополнительным затратами на реализацию многопоточности (создание потоков, реализация совместного доступа к ресурсам).

Заключение

В данной лабораторной работе были рассмотрены параллельные алгоритмы умножения матриц.

Среди рассмотренных алгоритмов наиболе эффективным по времени яляется параллельный алгоритм умножения матриц по строкам, так как в нем отсутствуют лишние обращения к памяти.

В связи с вышуказанным, параллельный алгоритм умножения по строкам является предпочтительным при обработке больших матриц в многопоточном окружении, однако, при работе с матрицами малых размерностей (меньше 64), стандартный алгоритм умножения становится более эффективным в связи с дополнительными затратами на организацию параллельности вычислений (создание потоков, организация совместного доступа к ресурсам).

В рамках выполнения работы решены следующие задачи.

- изучtены основные методы параллельных вычислений;
- реализован каждый из указанных алгоритмов умножения матриц;
- проведено сравнение временных характеристик реализованных алгоритмов экспериментально.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Group-theoretic Algorithms for Matrix Multiplication / H. Cohn [и др.] // Proceedings of the 46th Annual Symposium on Foundations of Computer Science. — 2005. — С. 379—388.
2. C language [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/c/language` (дата обращения: 07.10.2021).
3. pthreads(7) - Linux manual page [Электронный ресурс]. — `https://man7.org/linux/man-pages/man7/pthreads.7.html` (дата обращения: 07.10.2021).
4. Процессор Intel® Core™ i5-8250U [Электронный ресурс]. — Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/124967/intel-core-i5-8250u-processor-6m-cache-up-to-3-40-ghz.html` (дата обращения: 10.07.2021).
5. Manjaro - enjoy the simplicity [Электронный ресурс]. — Режим доступа: `https://manjaro.org/` (дата обращения: 10.07.2021).
6. LINUX.ORG.RU – Русская информация об ОС Linux [Электронный ресурс]. — Режим доступа: `https://www.linux.org.ru/` (дата обращения: 10.07.2021).