Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа №4

Исследование параллельных вычислений на основе параллельного алгоритма Винограда

Студент группы ИУ7-55Б,
Руднев К. К.,

Преподаватель,
Волкова Л. Л.,
Строганов Ю. В.

2019 г.

Оглавление

Введение

Цель работы: изучение возможности параллельных вычислений и использование такого подхода на практике, а также реализация параллельного алгоритма Винограда умножения матриц. В данной лабораторной работе рассматривается алгоритм Винограда и параллельный алгоритм Винограда.

Задачи:

- сравнить зависимость времени от числа параллельных потоков и размера матриц;
- разработать алгоритм для умножения матриц алгоритмом Винограда последовательно;
- разработать алгоритм для умножения матриц алгоритмом Винограда параллельно;
- провести сравнение стандартного и однопоточного параллельного алгоритмов.

Глава 1

Аналитическая часть

В рамках раздела будет дано аналитическое описание алгоритма для умножения матриц по Винограду, а также идеи его распаралелливания.

1.1 Описание алгоритмов

Пусть даны две прямоугольные матрицы A[MxQ] и B[QxN]:

$$A = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1Q} \\ a_{21} & a_{22} & ... & a_{1Q} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ a_{M1} & a_{M2} & ... & a_{MQ} \end{bmatrix}, B = \begin{bmatrix} b_{11} & b_{12} & ... & b_{1N} \\ b_{21} & b_{22} & ... & b_{1N} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ b_{Q1} & b_{Q2} & ... & b_{QN} \end{bmatrix}.$$

Тогда матрица C[MxN]:

$$C = \begin{bmatrix} c_{11} & c_{12} & ... & c_{1N} \\ c_{21} & c_{22} & ... & c_{1N} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ c_{M1} & c_{M2} & ... & c_{MN} \end{bmatrix}, \text{ в которой:}$$

$$c_{ij} = \sum_{r=1}^{m} a_{ir} * b_{rj} \quad (i = 1, 2, ...M; j = 1, 2, ...N). \quad (1.1)$$

называется их произведением [1].

1.1.1 Алгоритм Винограда

Подход алгоритма Винограда является иллюстрацией общей методологии, начатой в 1979-х годах на основе билинейных и трилинейных форм, благодаря которым большинство усовершенствований для умножения матриц были получены [2]. Пусть даны две прямоугольные матрицы A[1xQ] и B[Qx1]:

$$\mathrm{U} = \begin{bmatrix} u_1 & u_2 & \dots & u_Q \end{bmatrix}, \mathrm{V} = \begin{bmatrix} v_1 \\ v_2 \\ \cdot \\ \cdot \\ \cdot \\ v_Q \end{bmatrix}.$$

Тогда матрица C[1x1]:

$$C_{ij} = u_1 * v_1 + u_1 * v_1 + \dots + u_Q * v_Q = (u_1 + v_2) * (u_2 + v_1) + \dots + (u_{Q-2} + v_{Q-2}) * (u_Q + v_Q) - u_1 * u_2 - u_{Q-3} * u_{Q-2} - u_{Q-1} * u_Q - v_1 * v_2 - v_{Q-3} * v_{Q-2} - v_{Q-1} * v_Q \quad (1.2)$$

В случае, если матрица имеет нечетную размерность, необходимо отдельно обработать последний вектор:

$$C_{ij} = u_{in-1} * v_{n-1j} \quad (1.3)$$

1.1.2 Параллельный алгоритм Винограда

Трудоёмкость алгоритма Винограда имеет сложность O(MNQ) для умножения матрицы MxN на матрицу NxQ. Для улучшения алгоритма имеет смысл распараллелить основную часть алгоритма с тремя вложенными циклами. Опционально можно вычислять массивы mulh и mulv на разных потоках для сокращения времени. Также опционально можно производить обработку последнего вектора в отдельном потоке.

1.2 Параллельное программирование

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время дотупа к памяти является одинаковым (при одновременном доступе нескольких процесоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (symmetric multiprocessors, SMP).

Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время многоядерные процессоры, в которых каждое ядро представляет практически независимо функционирующее вычислительное устройство.

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью - создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельноее программирование, так и уже имеющиеся языки, расширенные некоторым набором операторов для параллельных вычислений.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (application programing interface, API) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows

Thread API. Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоёмким для использования и имеет низкоуровневый характер [3].

1.2.1 Организация взаимодействия параллельных потоков

Потоки исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие данные могут изменяться несколькими потоками, необходимы специальные усилия для организации правильного взаимодействия.

1.3 Вывод

Был рассмотрен алгоритм Винограда и возможность его оптимизации с помощью распараллеливания потоков. Была рассмотрена технология параллельного программирования и организация взаимодействия параллельных потоков.

Глава 2

Конструкторская часть

В рамках раздела будет представлена схема алгоритма Винограда, изображенная на рисунках 2.1-2.5. Также будет объявлено решение о распараллеливании этого алгоритма.

2.1 Разработка алгоритмов

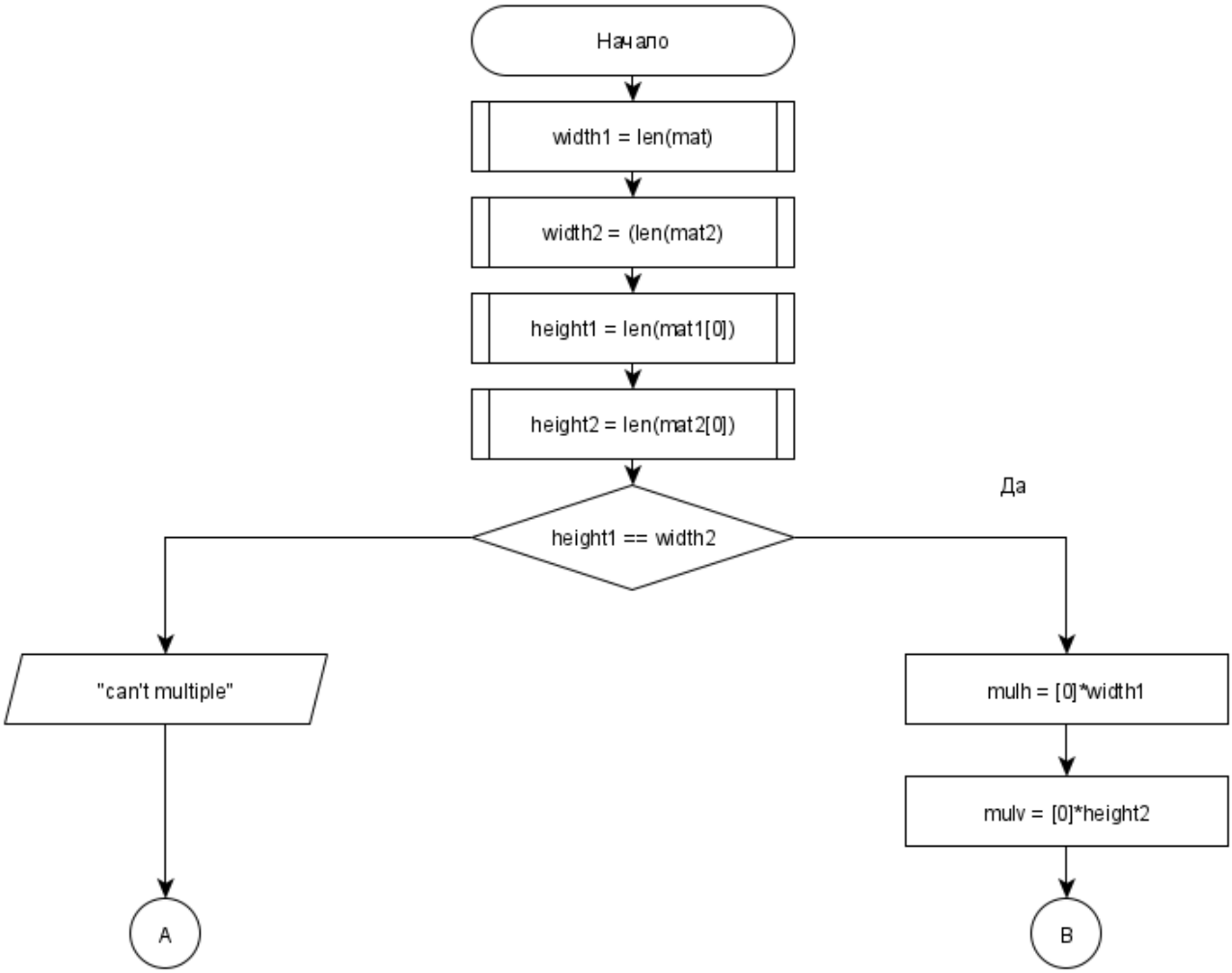

Рис. 2.1: Алгоритм Винограда умножения матриц. Часть 1

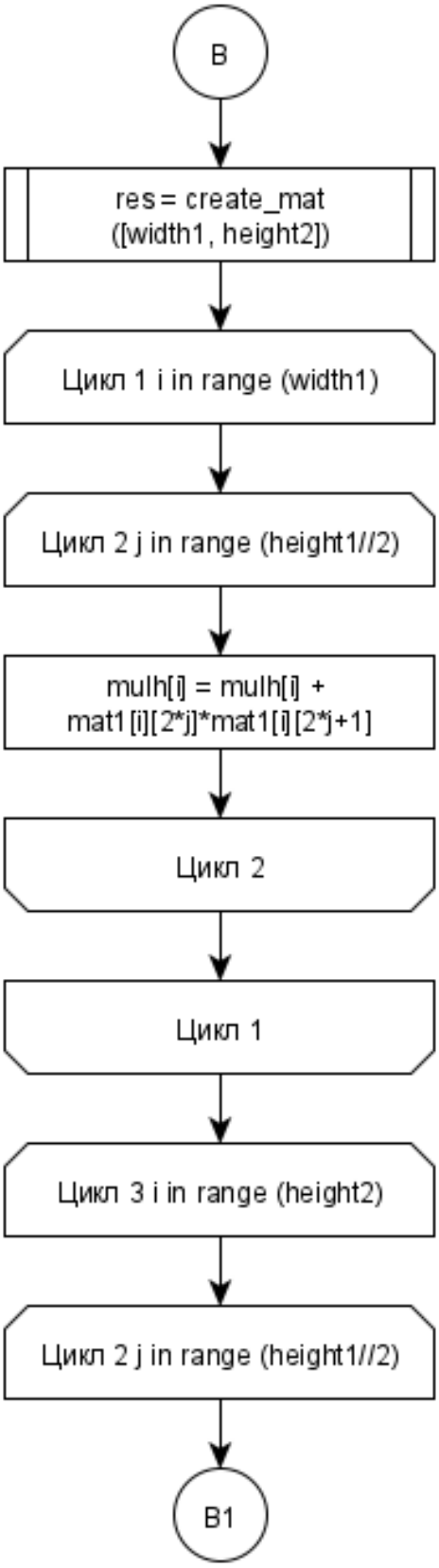

Рис. 2.2: Алгоритм Винограда умножения матриц. Часть 2

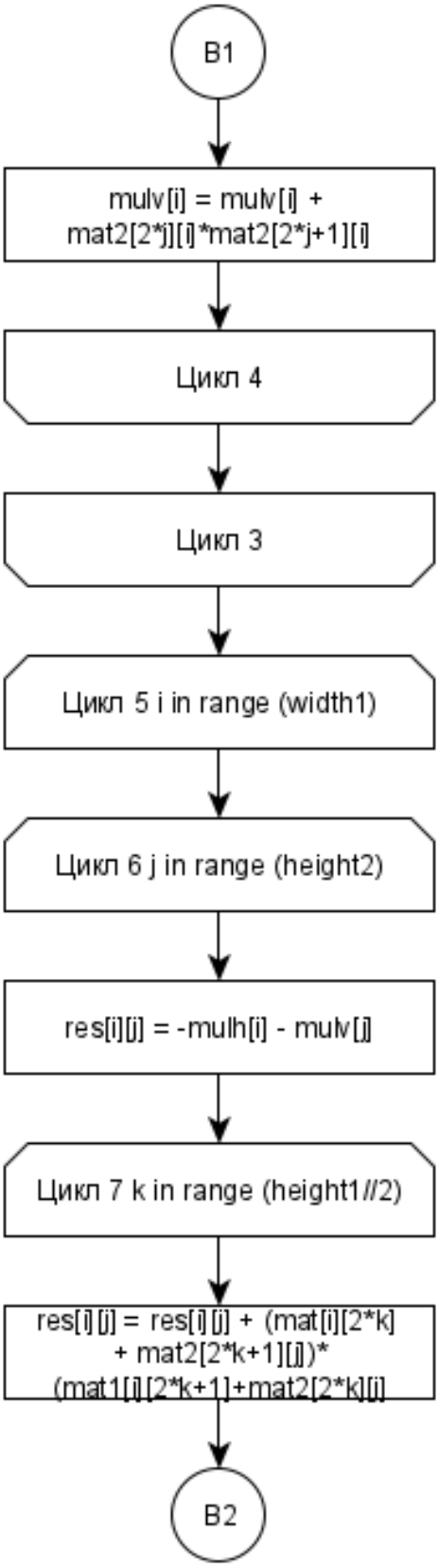

Рис. 2.3: Алгоритм Винограда умножения матриц. Часть 3

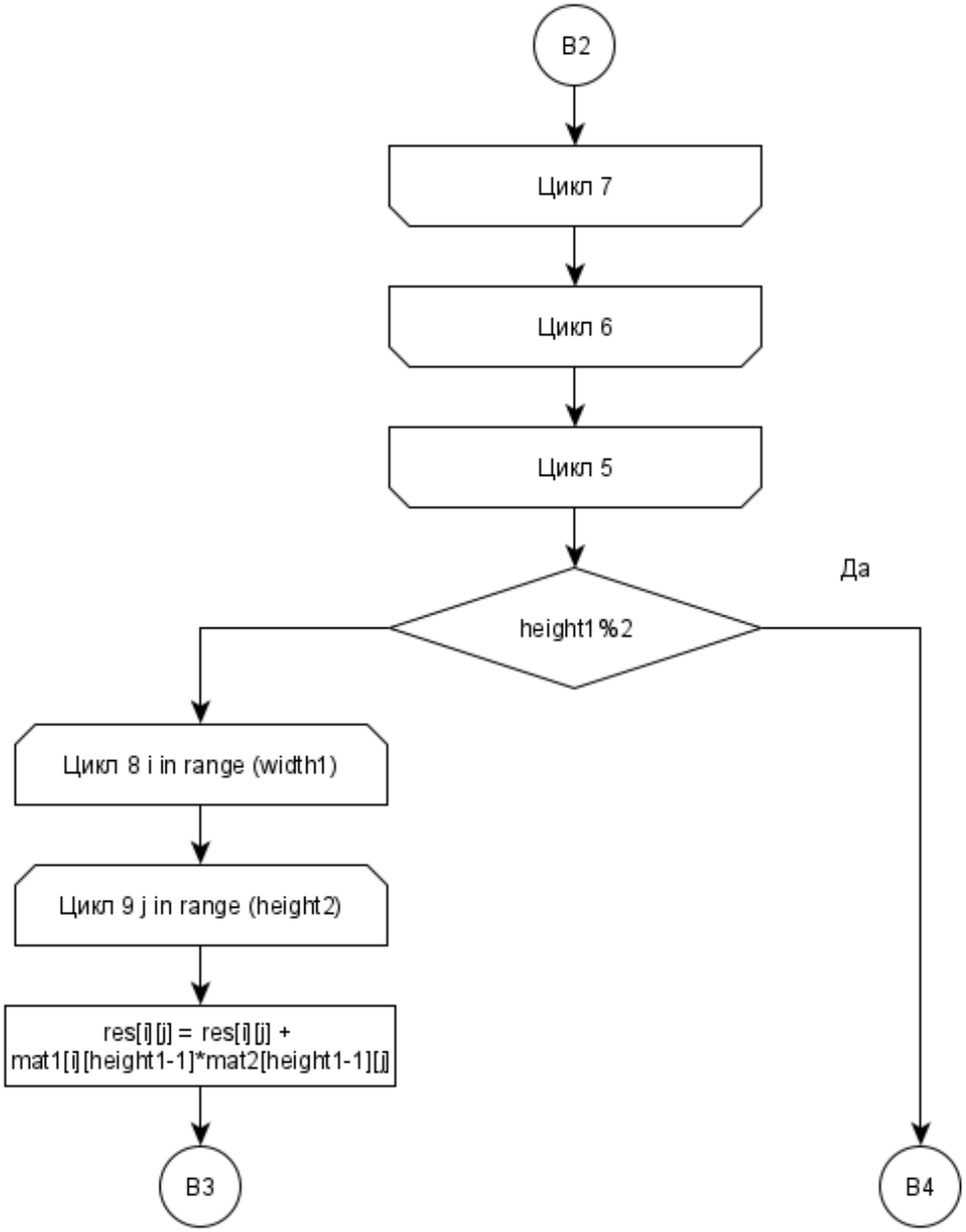

Рис. 2.4: Алгоритм Винограда умножения матриц. Часть 4

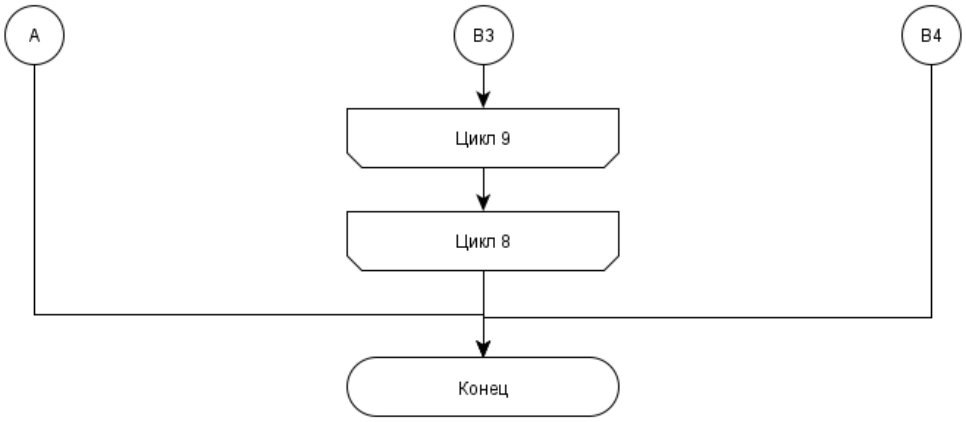

Рис. 2.5: Алгоритм Винограда умножения матриц. Часть 5

2.2 Распараллеливание алгоритма

Распараллеливание программы должно ускорять её выполнение. Это достигается за счет реализации в узким участках (например, в циклах с большим количеством независимых вычислений).

В предложенном алгоритме данным участком будет являться тройной цикл поиска результата. Данный блок и предлагается распараллелить. На рисунках 2.3-2.4 этот блок представлен циклами 5-7. Также было принято решение вычислять массивы mulv и mulh в отдельных потоках.

2.3 Вывод

В данном разделе была рассмотрена схема алгоритма Винограда и способ её распараллеливания.

Глава 3

Технологическая часть

Замеры времены были произведены на: Intel(R) Core(TM) i3-6006U, 2 ядра, 4 логических процессора.

3.1 Средства реализации

Для реализации алгоритмов использовался язык программирования C++ 11 и среда разработки QtCreator Community Edition 5.5. У данного языка имеются удобные библиотеки для написания мультипоточных приложений, чего будет достаточно для реализации текущей лабораторной работы, а среда разработки имеет бесплатную коммьюнити версию и подходящий компилятор, позволяющий использовать std::threads.

Замер времени реализован с помощью метода high_resolution_clock() класса std::chrono. Измеряется время исполнения кода чистого алгоритма (без учета времени на создание матриц, генерацию данных и т.п.).

3.2 Требования к программному обеспечению

На вход программа должна получать две матрицы, для которых вычисляется их произведение различными алгоритмами (параллельный алгоритм умножения матриц по Винограду, параллельный, но однопоточный алгоритм умножения матриц по Винограду). На выход программа должна выдавать результирующую матрицу всеми алгоритмами, должна корректно обрабатывать исключительные ситуации вроде невозможности произведения данных матриц.

3.3 Листинг кода

На листинге 3.1 представлены дополнительные функции и объявления, необходимые для реализации алгоритмов. На листинге 3.2 представлена реализация последовательного алгоритма Винограда. На листинге 3.3 представлена реализация параллельного алгоритма Винограда.

Листинг 3.1: Вспомогательные классы и объявления

```
#include <iostream>
#include <thread>
#include <vector>
#include <ctime>
#include <memory>

class Imatrix
```

```
{
public:
        virtual int get_rows() = 0;
        virtual int get_cols() = 0;
        virtual std::vector<std::vector<int>> get_mat() = 0;
};

class matrix:Imatrix
{
private:
        std::vector<std::vector<int>> mat;
        int rows;
        int cols;

public:
        matrix():
        rows(0), cols(0)
        {}

        matrix(int n, int m):
        rows(n), cols(m)
        {
                std::vector<std::vector<int>> tmp;
                srand(time(0));
                for (int i = 0; i < n; ++i)
                {
                        std::vector<int> row;
                        for (int j = 0; j < m; ++j)
                                row.push_back(rand() % 10 + 1);
                        tmp.push_back(row);
                }
                mat = tmp;
        }

        matrix(matrix *m1)
        {
                if (this != m1)
                {
                        mat = m1->get_mat();
                        rows = m1->get_rows();
                        cols = m1->get_cols();
                }
        }

        void zero_matrix()
        {
                std::vector<std::vector<int>> tmp;

                for (int i = 0; i < this->get_rows(); ++i)
                {
                        std::vector<int> row;
                        for (int j = 0; j < this->get_cols(); ++j)
```

```
                        row.push_back(0);
                    tmp.push_back(row);
                }

                this->set_mat(tmp);
        }

        std::vector<std::vector<int>> get_mat() override
        {
                return mat;
        }

        int get_rows() override
        {
                return rows;
        }

        int get_cols() override
        {
                return cols;
        }

        void set_mat(std::vector<std::vector<int>> m1)
        {
                mat = m1;
        }

        void set_rows(int r)
        {
                rows = r;
        }

        void set_cols(int c)
        {
                cols = c;
        }
};
```

Листинг 3.2: Последовательный алгоритм Винограда

```
matrix *wino_sequence()
{
        int m = m1->get_rows();
        int n = m1->get_cols();
        int q = m2->get_cols();
        int _2n = n/2;

        auto *result = new matrix();
        result->set_rows(m);
        result->set_cols(q);

        result->zero_matrix();
        std::vector<std::vector<int>> res = result->get_mat();
```

```
        std::vector<std::vector<int>> mat1 = m1->get_mat();
        std::vector<std::vector<int>> mat2 = m2->get_mat();

        auto *row_factor = new int[m];
        auto *column_factor = new int[q];

        for (int i = 0; i < m; ++i)
                row_factor[i] = 0;

        for (int i = 0; i < m; ++i)
                for (int j = 0; j < _2n; ++j)
                        row_factor[i] += mat1[i][2*j]*mat1[i][2*j+1];

        for (int i = 0; i < q; ++i)
                column_factor[i] = 0;

        for (int i = 0; i < q; ++i)
                for (int j = 0; j < _2n; ++j)
                        column_factor[i] += mat2[2*j][i]*mat2[2*j+1][i];

        for (int i = 0; i < m; ++i)
        {
                for (int j = 0; j < q; ++j)
                {
                        res[i][j] = -row_factor[i] - column_factor[j];
                        for (int k = 0; k < _2n; ++k)
                        {
                                res[i][j] += (mat1[i][2*k+1] + mat2[2*k][j])
                                        *(mat1[i][2*k] + mat2[2*k+1][j]);
                        }
                }
        }

        if (n % 2)
        {
                for (int i = 0; i < m; ++i)
                {
                        for (int j = 0; j < q; j++)
                        {
                                res[i][j] += mat1[i][n-1] * mat2[n-1][j];
                        }
                }
        }

        result->set_mat(res);
        return result;
}
```

Листинг 3.3: Параллельный алгоритм Винограда

```
matrix *wino_paral(int num_thread = 1)
{
    int m = m1->get_rows();
```

```
int n = m1->get_cols();
int q = m2->get_cols();
int _2n = n/2;

auto *result = new matrix();
result->set_rows(m);
result->set_cols(q);

result->zero_matrix();
std::vector<std::vector<int>> res = result->get_mat();
std::vector<std::vector<int>> mat1 = m1->get_mat();
std::vector<std::vector<int>> mat2 = m2->get_mat();

std::unique_ptr<int> row_factor(new int[m]);
std::unique_ptr<int> column_factor(new int[q]);

auto rows_func_thread = [](std::vector<std::vector<int>> m1, int*
    row_factor, int _2n)
{
        int r = m1.size();

        for (int i = 0; i < r; ++i)
                row_factor[i] = 0;

        for (int i = 0; i < r; ++i)
                for (int j = 0; j < _2n; ++j)
                        row_factor[i] += m1[i][2*j]*m1[i][2*j+1];
};
std::thread rows_thread(rows_func_thread, mat1, row_factor.get(), _2n);

auto cols_func_thread = [](std::vector<std::vector<int>> m2, int*
    column_factor, int _2n)
{
        int c = m2[0].size();

        for (int i = 0; i < c; ++i)
                column_factor[i] = 0;

        for (int i = 0; i < c; ++i)
                for (int j = 0; j < _2n; ++j)
                        column_factor[i] += m2[2*j][i]*m2[2*j+1][i];
};
std::thread cols_thread(cols_func_thread, mat2, column_factor.get(), _2n);

if (rows_thread.joinable())
        rows_thread.join();
if (cols_thread.joinable())
        cols_thread.join();

auto thread_counter = num_thread;
std::thread threads[thread_counter];
```

```
        auto winograd_thread = [](std::vector<std::vector<int>> mat1,
        std::vector<std::vector<int>> mat2, std::vector<std::vector<int>> &res,
        int* row_factor, int* column_factor, int number, int count)
        {
                int d = mat2.size()/2;
                for (unsigned int i = number; i < mat1.size(); i += count)
                {
                        for (unsigned int j = 0; j < mat2[0].size(); ++j)
                        {
                                res[i][j] = -row_factor[i] - column_factor[j];
                                for (int k = 0; k < d; ++k)
                                {
                                        res[i][j] += (mat1[i][2*k+1] + mat2[2*k][j]) *(
                                            mat1[i][2*k] + mat2[2*k+1][j]);
                                }
                        }
                }
        };

        for (int i = 0; i < thread_counter; ++i)
        threads[i] = std::thread(winograd_thread, mat1, mat2, std::ref(res),
        row_factor.get(), column_factor.get(), i, thread_counter);

        for (int i = 0; i < thread_counter; ++i)
                if (threads[i].joinable())
                        threads[i].join();

        if (n % 2)
        {
                for (int i = 0; i < m; ++i)
                {
                        for (int j = 0; j < q; j++)
                        {
                                res[i][j] += mat1[i][n-1] * mat2[n-1][j];
                        }
                }
        }

        result->set_mat(res);
        return result;
    }
```

3.4 Вывод

В рамках раздела были предъявлены требования к программному обеспечению. На основании их были разработаны и представлены конкретные реализации параллельных однопоточных и мультипоточных алгоритмов умножения матриц по Винограду.

Глава 4

Экспериментальная часть

В рамках раздела будут проведены эксперименты, связанные с временем выполнения последовательного и параллельного алгоритмов. Результаты проведённых экспериментов представлены на рисунках 4.1-4.19.

4.1 Сравнительный анализ на материале экспериментальных данных

Результаты проведения эксперимента по расчету времени выполнения однопоточного и последовательного алгоритмов на четных размерностях.

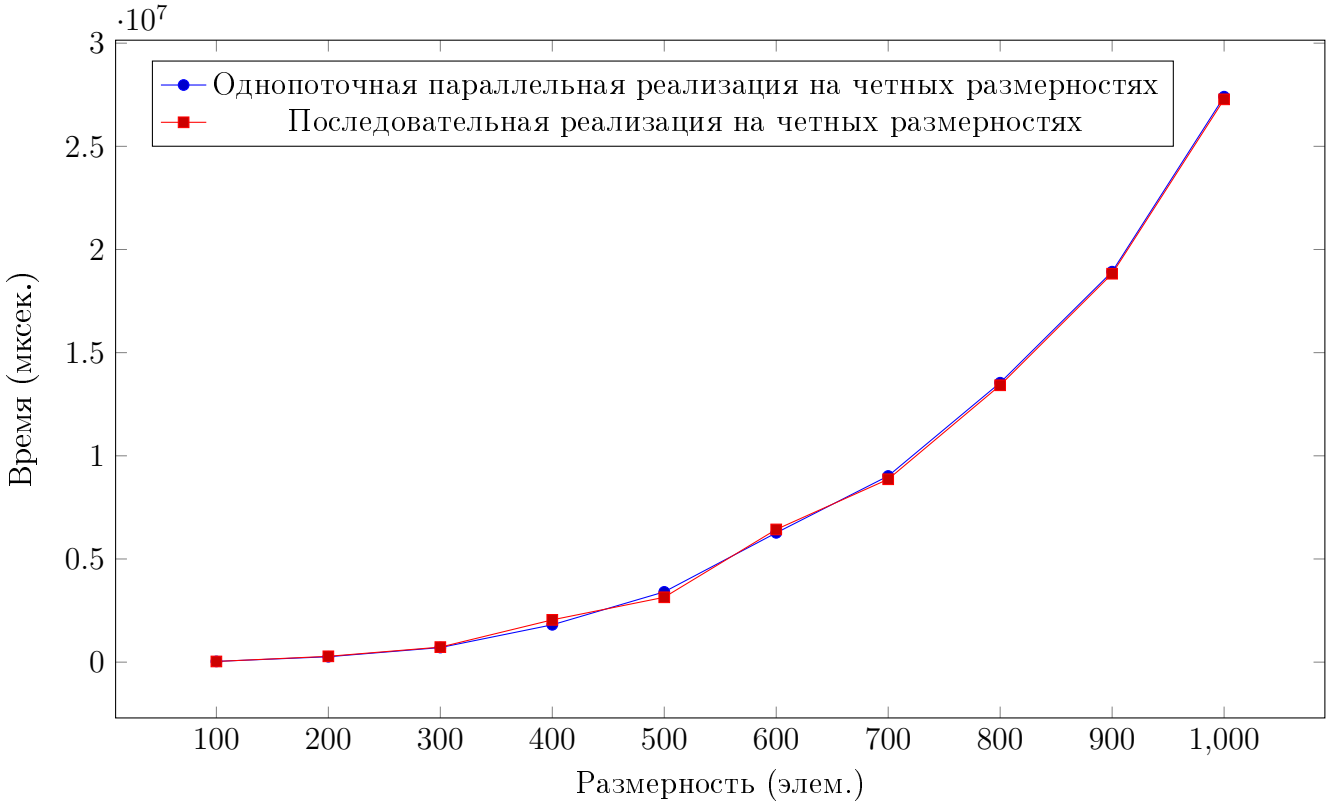

Рис. 4.1: Сравнение времени работы однопоточной параллельной и последовательной реализаций на четных размерностях

Результаты проведения эксперимента по расчету времени выполнения однопоточного и последовательного алгоритмов.

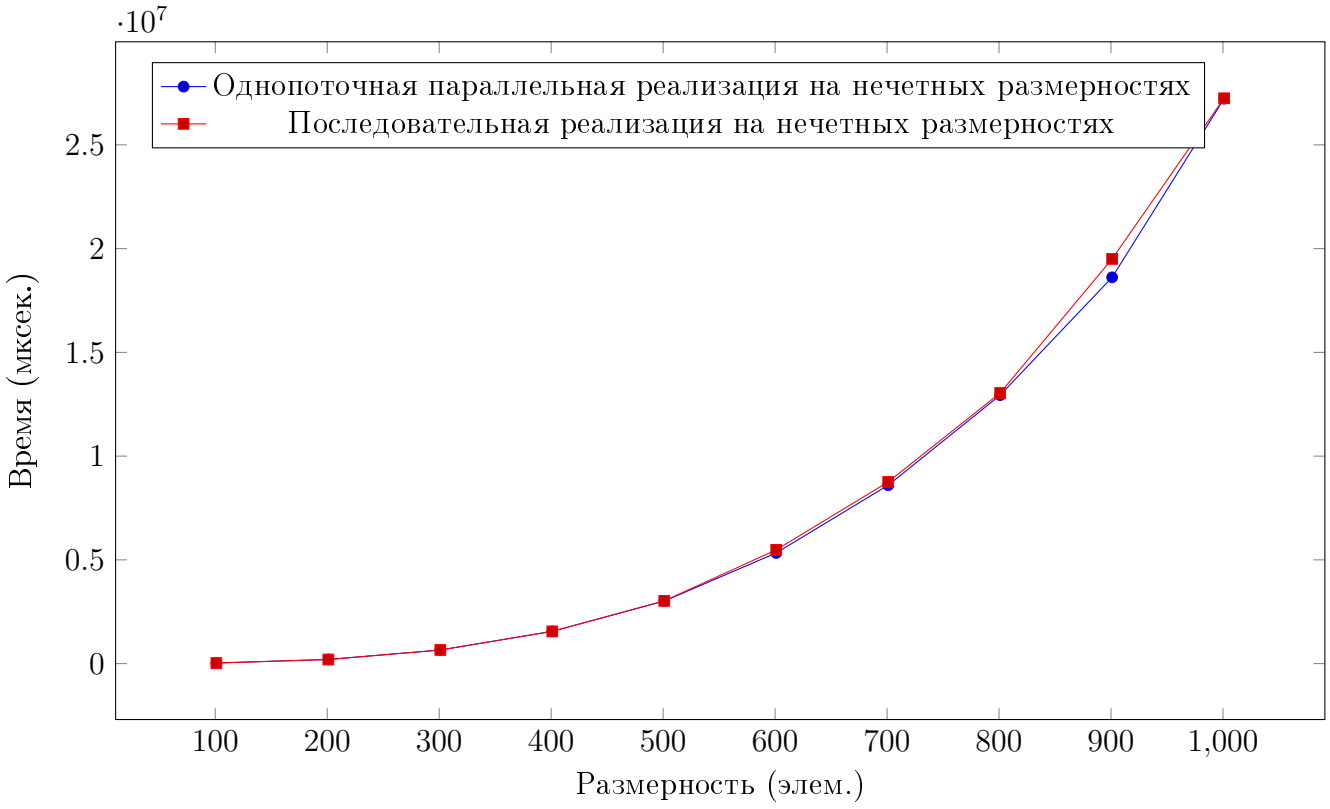

Рис. 4.2: Сравнение времени работы однопоточной параллельной и последовательной реализаций на нечетных размерностях

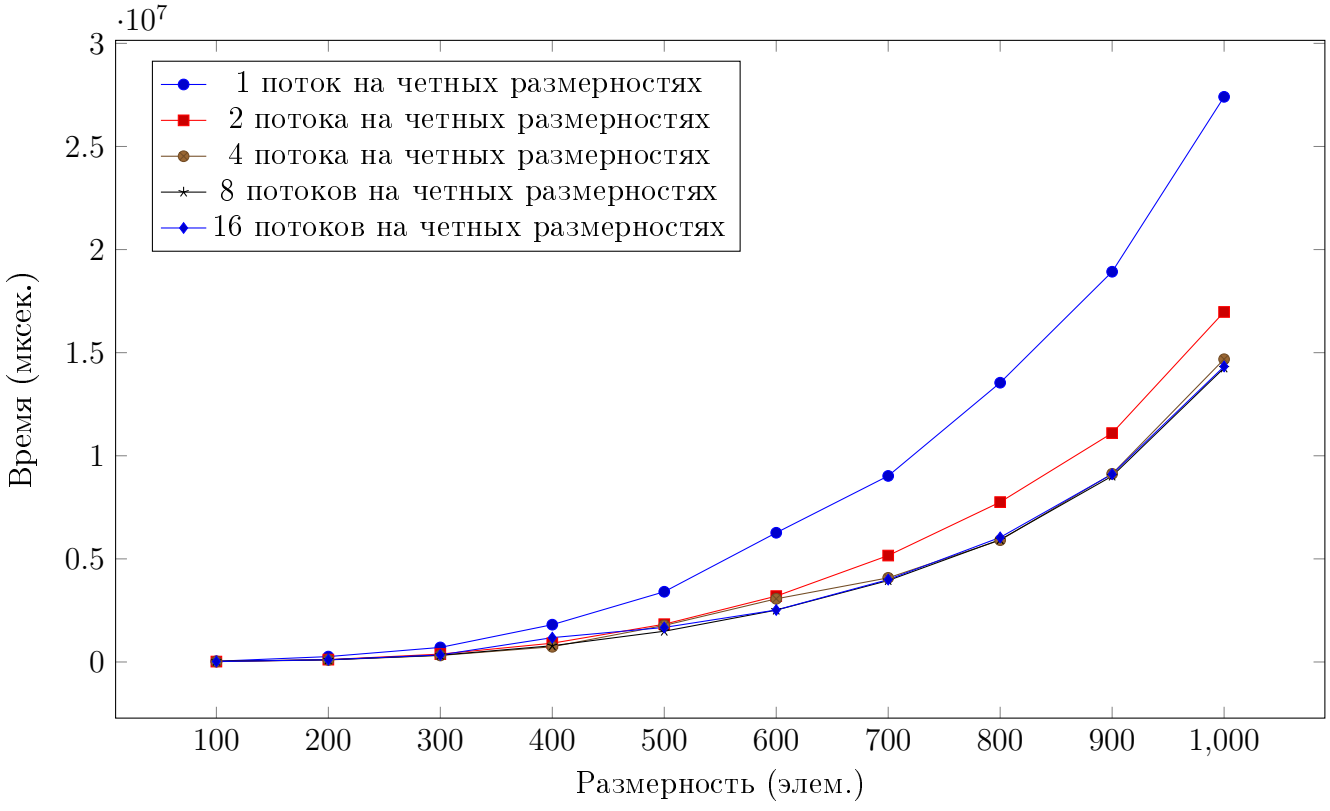

Рис. 4.3: Сравнение времени работы многопоточной параллельной реализации на четных размерностях

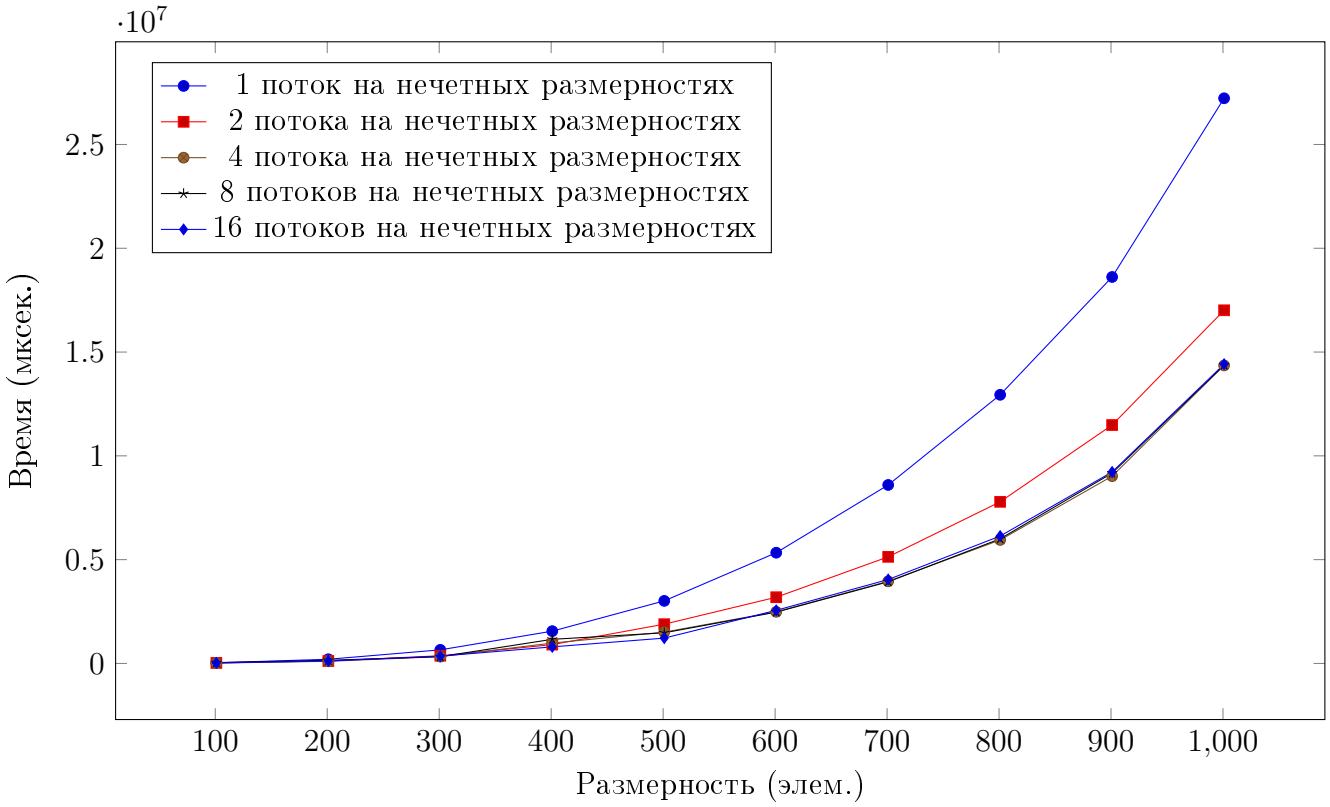

Рис. 4.4: Сравнение времени работы многопоточной параллельной реализации на нечетных размерностях

4.2 Вывод

Параллельная однопоточная и последовательная реализации похожи между собой по времени работы, составляя не более 2 % разницы, используя затраченное время на создание и ожидание потока.

Сравнивая мультипоточные реализации с разным числом потоков, были получены следующие результаты:

- увеличение числа потоков вплоть до 4 ведет к снижению времени выполнения (на 2 потоках разница до 100 %, на 4 потоках разница до 30 % по сравнению с 2 потоками);
- увеличение числа потоков свыше 4 ведет к незначительному снижению времени.

Заключение

В ходе лабораторной работы я изучил возможности параллельных вычислений и использовал такой подход на практике. Реализовал алгоритм Винограда умножения матриц с помощью параллельных вычислений. Было произведено сравнение работы обычного алгоритма Винограда и параллельной реализации при увеличении количества потоков.

Выяснилось, что однопоточная параллельная и последовательная реализации не имеют разницы (до 2 %) во времени выполнения, а увеличение потоков до 4 сокращает время работы на 70 % по сравнению с однопоточной реализацией на тестовых данных размерностью до 1001x1001. Однако дальнейшее увеличение количества потоков не дает значительного выигрыша во времени (разница до 5 %).

Литература

[1] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16

[2] Le Gall, F. (2012), "Faster algorithms for rectangular matrix multiplication Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), pp. 514–523

[3] Константин Баркалов, Владимир Воеводин, Виктор Гергель. Intel Parallel Programming [Электронный ресурс], - режим доступа https://www.intuit.ru/studies/courses/4447/983/lecture/14925