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: «Параллельные вычисления»

Студент: Ле Ни Куанг

Группа: ИУ7и-56Б

Преподаватель: Волкова Л. Л.
Строганов Ю. В.

Москва

2020

Оглавление

Введение

Параллельные вычисления - способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно).

Целью работы: изучение параллельных вычисления с использованием алгоритма Винограда. В данной лабораторной работе реализовать последовательный и параллельный алгоритм Винограда.

Задачи работы:

1. изучить алгоритм Винограда умножения матриц;
2. реализовать последовательный и параллельный алгоритм Винограда;
3. сравнить временные характеристики реализованных алгоритмов экспериментально.

1 Аналитический раздел

В данном разделе будет приведено описание алгоритмов и модель вычислений для оценок трудоемкости.

1.1 Описание алгоритмов

1.1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы A и B размерности l x m и m x n соответственно:

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1m} \\ a_{21} & a_{22} & \ldots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \ldots & a_{lm} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{pmatrix}$$

Тогда матрица C размерностью l x m:

$$C = \begin{pmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \ldots & c_{ln} \end{pmatrix}$$

в которой:

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

называется их произведением.

1.1.2 Алгоритм оптимизированный Винограда

Рассматривая результат умножения двух матриц очевидно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. 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:

$$V \cdot W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4 \tag{1.2}$$

Это равенство можно переписать в виде:

$$V \cdot W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4 \tag{1.3}$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем первое: вместо четырех умножений - 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 позволяет выполнять для каждого элемента лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

1.1.3 Многопоточность

К достоинствам многопоточной реализации той или иной системы перед многозадачной можно отнести следующее:

- Упрощение программы в некоторых случаях за счет использования общего адресного пространства.

- Меньшие относительно процесса временные затраты на создание потока.

К достоинствам многопоточной реализации той или иной системы перед однопоточной можно отнести следующее:

- Упрощение программы в некоторых случаях, за счет вынесения механизмов чередования выполнения различных слабо взаимосвязанных подзадач, требующих одновременного выполнения, в отдельную подсистему многопоточности.

- Повышение производительности процесса за счет распараллеливания процессорных вычислений и операций ввода-вывода.

Существует два виды параллелизма в алгоритмах и программах:

- Конечный параллелизм определяется информационной независимостью некоторых фрагментов в тексте программы.

- Массовый параллелизм определяется информационной независимостью итераций циклов программы.

В этой работе я использую массовый параллелизм.

1.2 Вывод

Были приведено описание алгоритмов, стандартный и последовательный Винограда, также про многопоточность и параллелизм.

2 Конструкторский раздел

2.1 Разработка алгоритмов

На рисунках показаны схемы алгоритмов последовательного и параллельного алгоритма Винограда.

2.1.1 Схема алгоритма оптимизированного Винограда

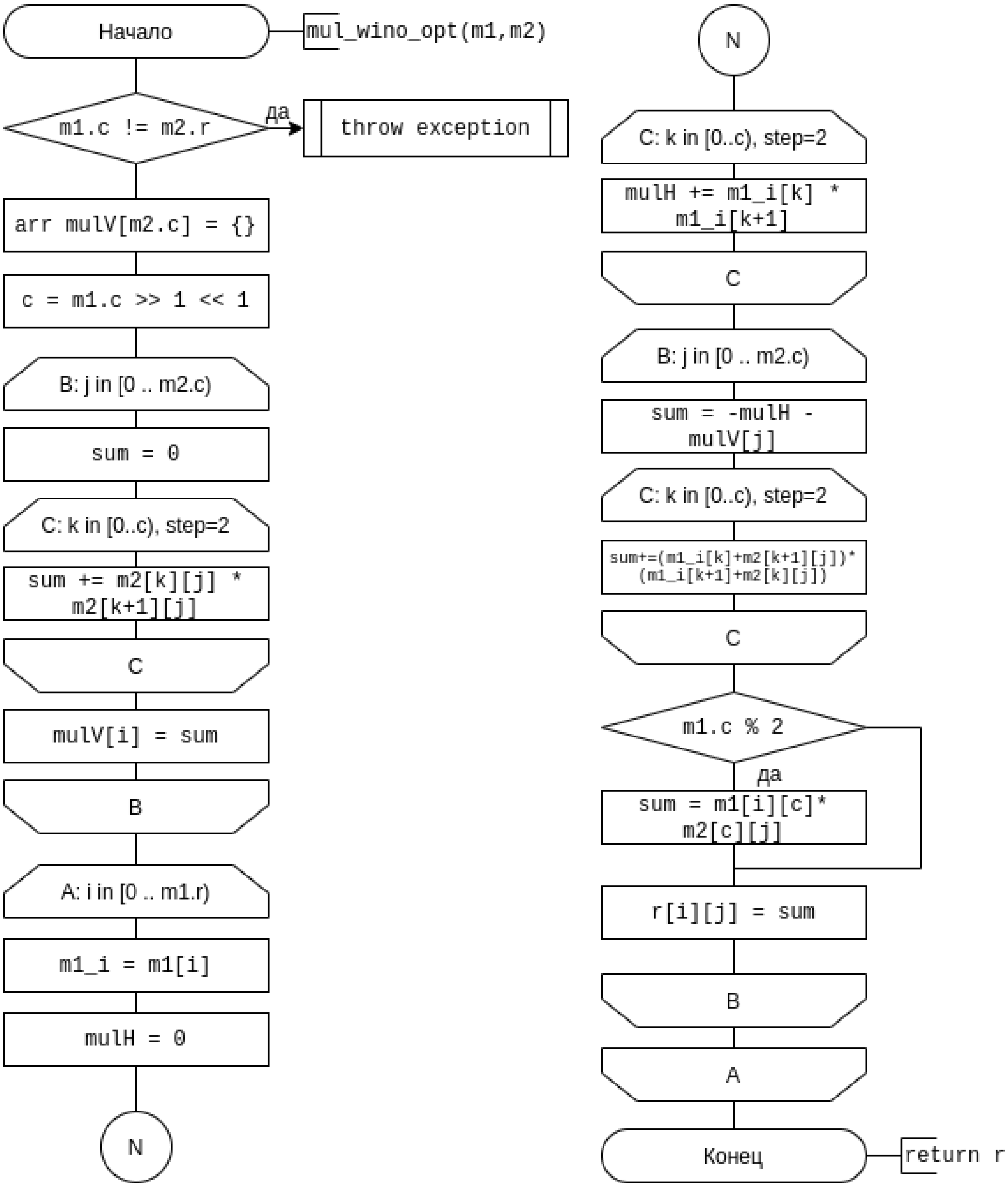

Рис. 2.1: Схема алгоритма оптимизированного Винограда

2.1.2 Схема параллельного оптимизированного алгоритма Винограда

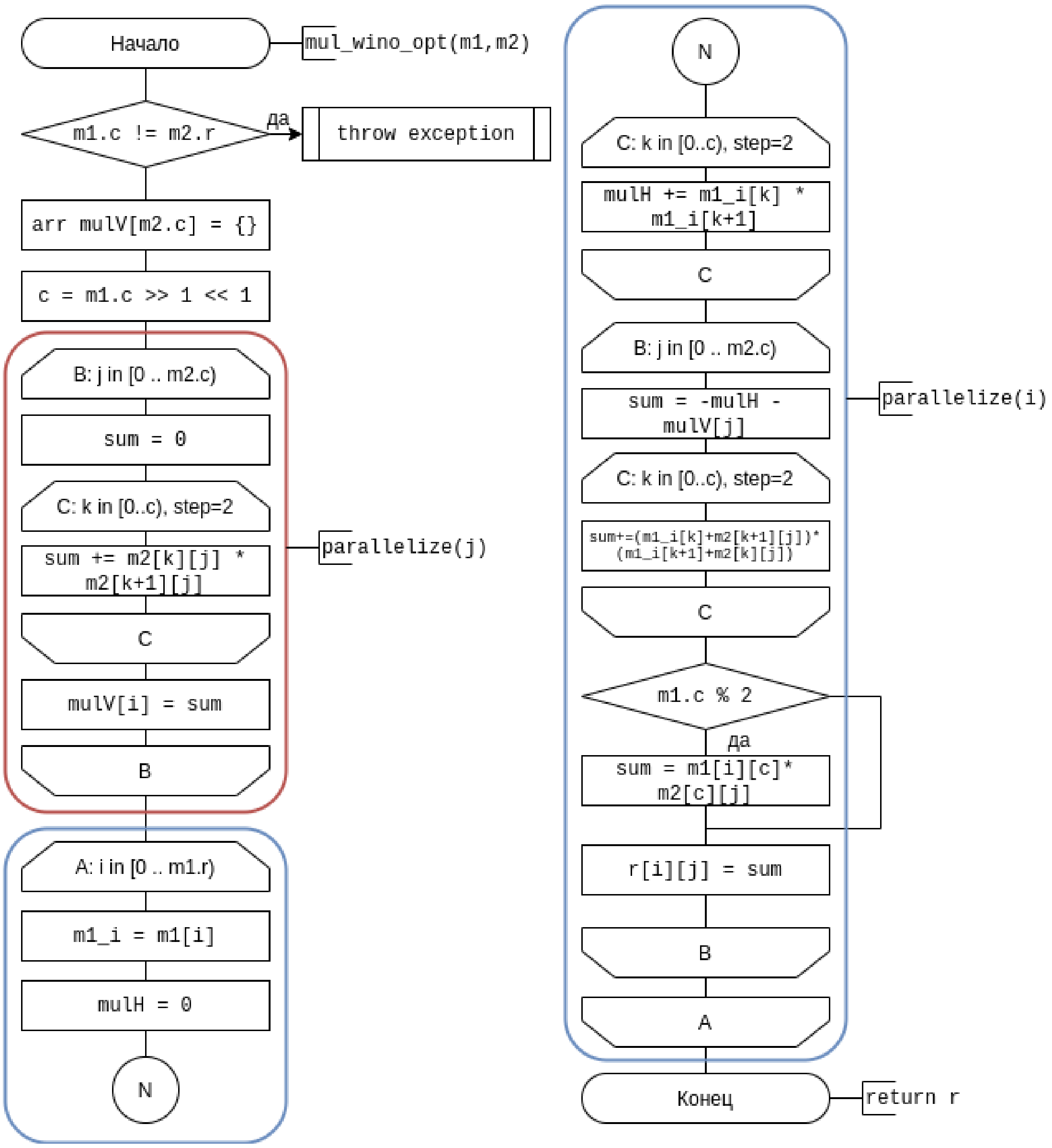

Рис. 2.2: Схема параллельного оптимизированного алгоритма Винограда

2.2 Вывод

В данном разделе было приведено описание схем алгоритмов последовательного и параллельного алгоритма Винограда.

3 Технологический раздел

3.1 Средства реализации

Язык программирования: C++

Библиотеки: google test, google benchmark

Редактор: VS Code

Я использую эти инструменты потому, что они мощные, широко используемые и хочу изучить фреймворк для тестирования и тестирования на C ++.

3.2 Листинг кода

Листинг 3.1: Шаблон для матричного типа

```
template <size_t R, size_t C, typename T = int>
class Matrix : public BaseMatrix
{
private:
    T data[R][C];
    // ...
}
```

Листинг 3.2: Оптимизированный алгоритм Винограда

```
template <size_t L, size_t M, size_t N, typename T>
Matrix<L,N,T> mul_winograd(
    Matrix<L,M,T> &m1,
    Matrix<M,N,T> &m2)
{
    Matrix<L,N,T> r;
    T sum, mulH;
    T mulV[N] = {};
    size_t M_ = M >> 1 << 1;


    for (int j = 0; j < N; j++)
    {
        sum = 0;
        for (int k = 0;  k < M_; k += 2)
```

```
                sum += m2[k][j] * m2[k+1][j];
            mulV[j] = sum;
        }

    T* m1_i = m1[0];

    for (int i = 0; i < L; i++, m1_i += M)
    {
        mulH = 0;
        for (int k = 0;  k < M_; k += 2)
            mulH += m1_i[k] * m1_i[k+1];

        for (int j = 0; j < N; j++)
        {
            sum = -mulH - mulV[j];
            for (int k = 0; k < M_; k += 2)
                sum += (m1_i[k] + m2[k+1][j])
                    * (m1_i[k+1] + m2[k][j]);

            if (M % 2)
                sum += m1_i[M_] * m2[M_][j];

            r[i][j] = sum;
        }
    }

    return r;
}
```

Листинг 3.3: Параллельный алгоритм Винограда

```
using f_parallel_t = std::function<void(size_t begin, size_t end)>;

void parallelize(f_parallel_t f, size_t loop_size, size_t n_thread)
{
    if (n_thread > loop_size)
        n_thread = loop_size;

    size_t block_size = loop_size / n_thread;
    size_t begin = 0;

    // + one main thread
    n_thread--;
    std::vector<std::thread> threads(n_thread);

    for (size_t i = 0; i < n_thread; i++, begin += block_size)
        threads[i] = std::thread(f, begin, begin + block_size);

    // main thread
```

```cpp
    f(begin, loop_size);

    for (auto& thread : threads)
        thread.join();
}


template <size_t L, size_t M, size_t N, typename T>
Matrix<L,N,T> mul_winograd_multithread(
    Matrix<L,M,T> &m1,
    Matrix<M,N,T> &m2,
    size_t n_thread = 1)
{
    Matrix<L,N,T> r;
    T mulV[N] = {};
    size_t M_ = M >> 1 << 1;

    auto fMulV = [&](size_t begin, size_t end) {
        for (int j = begin; j < end; j++)
        {
            T sum = 0;
            for (int k = 0;  k < M_; k += 2)
                sum += m2[k][j] * m2[k+1][j];
            mulV[j] = sum;
        }
    };

    auto fMulMat = [&](size_t begin, size_t end) {
        T* m1_i = m1[begin];
        for (int i = begin; i < end; i++, m1_i += M)
        {
            T mulH = 0;
            for (int k = 0;  k < M_; k += 2)
                mulH += m1_i[k] * m1_i[k+1];

            for (int j = 0; j < N; j++)
            {
                T sum = -mulH - mulV[j];
                for (int k = 0; k < M_; k += 2)
                    sum += (m1_i[k] + m2[k+1][j])
                        * (m1_i[k+1] + m2[k][j]);

                if (M % 2)
                    sum += m1_i[M_] * m2[M_][j];

                r[i][j] = sum;
            }
        }
```

```
    };

    // fMulV(0, N);
    // fMulMat(0, L);
    parallelize(fMulV, N, n_thread);
    parallelize(fMulMat, L, n_thread);

    return r;
}
```

3.3 Описание тестирования

В таблице 3.1 приведен функциональные тесты для алгоритмов умножения матриц.

Матрица 1	**Матрица 2**	**Ожидаемый результат**
$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & 3 \\ 4 & 4 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$
$\begin{pmatrix} 2 & 4 & 3 \\ 1 & -3 & 2 \end{pmatrix}$	$\begin{pmatrix} 2 & -3 \\ 4 & 4 \\ 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 26 & 19 \\ -6 & -9 \end{pmatrix}$
$\begin{pmatrix} 2 & -3 \\ 4 & 4 \\ 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 2 & 4 & 3 \\ 1 & -3 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 17 & 0 \\ 12 & 4 & 20 \\ 7 & -1 & 12 \end{pmatrix}$
$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 0 \end{pmatrix}$	Exception

Таблица 3.1: Функциональные тесты

3.4 Вывод

В этом разделе было рассмотрено код программы и описание тестирования.

4 Экспериментальный раздел

В данном разделе будет приведено пример работы программы, результаты тестирования и сравнение времени работы последовательного и параллельного алгоритма Винограда.

4.1 Примеры работы

На рисунке 4.1 и 4.2 приведен пример работы программы.

```
============ Program ============
A [2x3]
   2   4   3
   1  -3   2

B [3x2]
   2  -3
   4   4
   2   3

Coppersmith—Winograd optimized
AxB [2x2]
  26  19
  -6  -9

BxA [3x3]
   1  17   0
  12   4  20
   7  -1  12

Coppersmith—Winograd multithreading
AxB [2x2]
  26  19
  -6  -9

BxA [3x3]
   1  17   0
  12   4  20
   7  -1  12
```

Рис. 4.1: Примеры работы программы

```
=========== Benchmark ===========
2020-11-19 12:56:52
Running ./benchmark
Run on (4 X 3500 MHz CPU s)
CPU Caches:
  L1 Data 32K (x4)
  L1 Instruction 32K (x4)
  L2 Unified 256K (x4)
  L3 Unified 6144K (x1)
Load Average: 0.79, 1.34, 1.62
***WARNING*** CPU scaling is enabled, the benchmark real time measurements may
 be noisy and will incur extra overhead.
-------------------------------------------------------------------------------
Benchmark                                   Time             CPU   Iterations
-------------------------------------------------------------------------------
BM_Winograd<100>                      3407093 ns      3406441 ns          206
BM_Multithreading<100, 1>/real_time   3559595 ns      3559281 ns          197
BM_Multithreading<100, 2>/real_time   1978980 ns      1951829 ns          345
BM_Multithreading<100, 4>/real_time   1150360 ns      1062533 ns          608
BM_Multithreading<100, 8>/real_time   1339005 ns       834484 ns          499
BM_Winograd<200>                     27575684 ns     27575008 ns           25
BM_Multithreading<200, 1>/real_time  28160976 ns     28160329 ns           25
BM_Multithreading<200, 2>/real_time  15220214 ns     15149672 ns           46
BM_Multithreading<200, 4>/real_time   8280901 ns      7984681 ns           84
BM_Multithreading<200, 8>/real_time   8253001 ns      4163061 ns           76
BM_Winograd<300>                     87272007 ns     87269803 ns            8
BM_Multithreading<300, 1>/real_time  93495185 ns     93489594 ns            7
BM_Multithreading<300, 2>/real_time  49744438 ns     49701898 ns           14
BM_Multithreading<300, 4>/real_time  26377090 ns     26238940 ns           26
BM_Multithreading<300, 8>/real_time  27672882 ns     14662237 ns           25
BM_Winograd<400>                    231561069 ns    231557711 ns            3
BM_Multithreading<400, 1>/real_time 243034071 ns    243030604 ns            3
BM_Multithreading<400, 2>/real_time 129351382 ns    129307937 ns            5
BM_Multithreading<400, 4>/real_time  68763875 ns     68473684 ns           10
BM_Multithreading<400, 8>/real_time  69240058 ns     34514530 ns           10
BM_Winograd<500>                    430811602 ns    430801344 ns            2
BM_Multithreading<500, 1>/real_time 459220172 ns    459208547 ns            2
BM_Multithreading<500, 2>/real_time 244142588 ns    243669752 ns            3
BM_Multithreading<500, 4>/real_time 130122390 ns    129316936 ns            5
BM_Multithreading<500, 8>/real_time 132132422 ns     68566359 ns            5
BM_Winograd<600>                    747187461 ns    747163367 ns            1
BM_Multithreading<600, 1>/real_time 798074006 ns    798059755 ns            1
BM_Multithreading<600, 2>/real_time 422998262 ns    422845120 ns            2
BM_Multithreading<600, 4>/real_time 223707945 ns    223044773 ns            3
BM_Multithreading<600, 8>/real_time 232813010 ns    111770625 ns            3
```

Рис. 4.2: Примеры работы программы

4.2 Результаты тестирования

На рисунке 4.4 приведен результат теста с использованием фреймворка google test.

```
============ Testing ============
[==========] Running 4 tests from 2 test suites.
[----------] Global test environment set-up.
[----------] 2 tests from ZeroTest
[ RUN      ] ZeroTest.MulWinograd
[       OK ] ZeroTest.MulWinograd (0 ms)
[ RUN      ] ZeroTest.MulWinogradMultithreading
[       OK ] ZeroTest.MulWinogradMultithreading (1 ms)
[----------] 2 tests from ZeroTest (1 ms total)

[----------] 2 tests from NormalTest
[ RUN      ] NormalTest.MulWinograd
[       OK ] NormalTest.MulWinograd (0 ms)
[ RUN      ] NormalTest.MulWinogradMultithreading
[       OK ] NormalTest.MulWinogradMultithreading (0 ms)
[----------] 2 tests from NormalTest (0 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 2 test suites ran. (1 ms total)
[  PASSED  ] 4 tests.
```

Рис. 4.3: Результаты тестирования

4.3 Сравнение времени работы

В таблице 4.1 приведены замеры времени работы алгоритмов умножения матриц на квадратных матрицах, на основе них построены графики 4.4.

Размер	**Последо.**	**1 поток**	**2 поток**	**4 поток**	**8 поток**
100	3.40709e+06	3.5596e+06	1.97898e+06	1.15036e+06	1.33901e+06
200	2.75757e+07	2.8161e+07	1.52202e+07	8.2809e+06	8.253e+06
300	8.7272e+07	9.34952e+07	4.97444e+07	2.63771e+07	2.76729e+07
400	2.31561e+08	2.43034e+08	1.29351e+08	6.87639e+07	6.92401e+07
500	4.30812e+08	4.5922e+08	2.44143e+08	1.30122e+08	1.32132e+08
600	7.47187e+08	7.98074e+08	4.22998e+08	2.23708e+08	2.32813e+08
700	1.21963e+09	1.30366e+09	6.94993e+08	3.65427e+08	3.68879e+08
800	1.99084e+09	2.12634e+09	1.13247e+09	6.02996e+08	6.06262e+08

Таблица 4.1: Времени работы (ns)

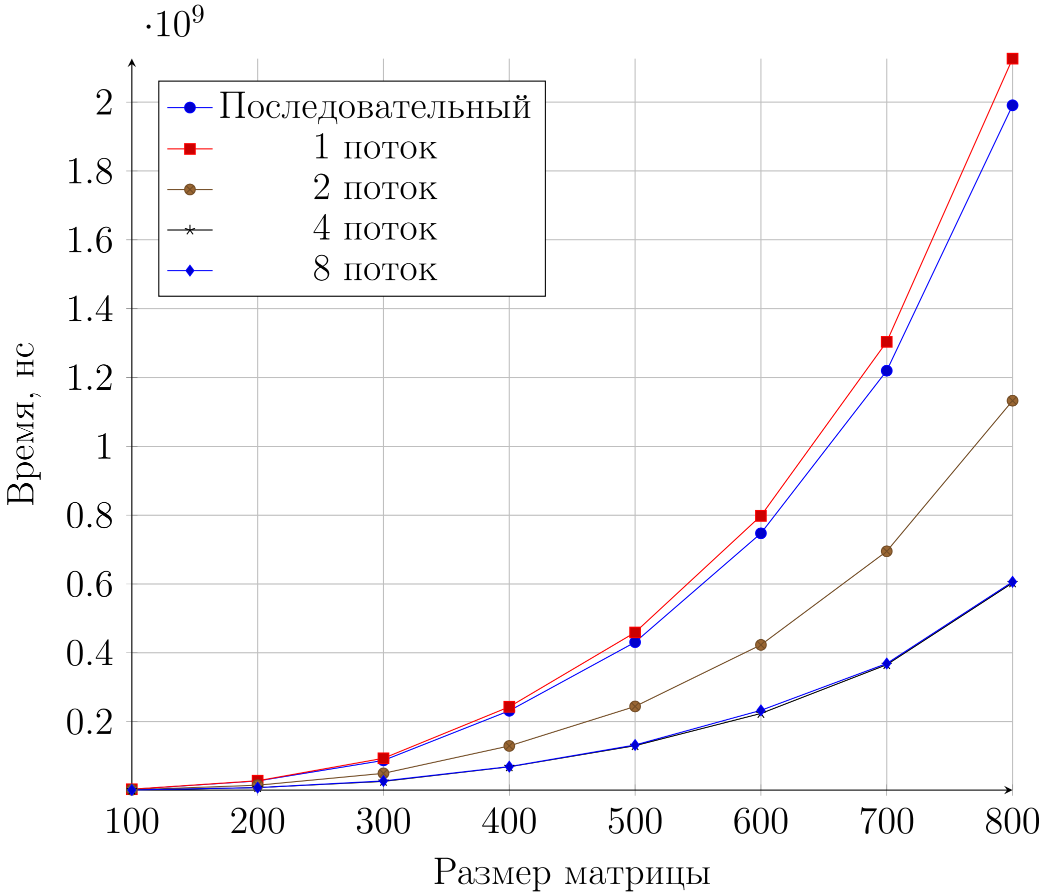

Рис. 4.4: Зависимость времени работы алгоритмов умножения матриц от размеры матрицы и количество потоков

4.4 Вывод

График показывает, что многопоточная версия более эффективна, когда количество потоков увеличивается, производительность пропорциональна количеству потоков до тех пор, пока она не станет равной количеству ядер процессора, и наиболее эффективна, когда количество потоков равно количеству ядер процессора. Затем, если количество потоков увеличивается, происходит небольшое уменьшение из-за необходимости управлять большим количеством потоков.

Заключение

В ходе лабораторной работы было изучено параллельных вычисления с использованием алгоритма Винограда, реализованны последовательный и параллельный алгоритм Винограда. Было сравнить временные характеристики последовательного и параллельного алгоритма Винограда и сделаны следующие выводы:

- производительность пропорциональна количеству потоков до тех пор, пока она не станет равной количеству ядер процессора;
- многопоточная версия наиболее эффективна когда количество потоков равно количеству ядер процессора;
- время выполнения с использованием 4 потоков всего 30% по сравнению с последовательным выполнением.

Литература

[1] Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб: БХВ-Петербург, 2002. — 608 с.

[2] C++ reference
https://en.cppreference.com/w/cpp/thread/thread

[3] Google Testing Framework
https://github.com/google/googletest

[4] Google Benchmark
https://github.com/google/benchmark