МГТУ им. Баумана

Лабораторная работа №4

По курсу: "Анализ алгоритмов"

Параллельное умножение матриц

Работу выполнила: Лаврова Анастасия, ИУ7-55Б

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

Москва, 2019

Оглавление

Введение **2**

1 Аналитическая часть **3**
1.1 Алгоритм Винограда 3
1.2 Параллельный алгоритм Винограда 3
1.3 Распараллеливание задачи 4
1.3.1 Вывод 4

2 Конструкторская часть **5**
2.1 Схемы алгоритмов 6

3 Технологическая часть **9**
3.1 Выбор ЯП 9
3.2 Реализация алгоритма 9

4 Исследовательская часть **14**
4.1 Постановка эксперимента 14
4.2 Сравнительный анализ на основе замеров времени работы алгоритмов 14
4.3 Вывод 16

Заключение **17**

Введение

Цель работы: изучение алгоритмов умножения матриц. В данной лабораторной работе рассматривается стандартный алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда. Также требуется изучить рассчет сложности алгоритмов, получить навыки в улучшении алгоритмов. Эти алгоритмы активно применяются во всех областях, применяющих линейную алгебру, таких как:

- компьютерная графика
- физика
- экономика

В ходе лабораторной работы предстоит:

- Изучение и реализация параллельного алгоритма Винограда для умножения матриц
- Сравнить зависимость времени работы алгоритма от числа параллельных потоков исполнения и размера матриц
- Провести сравнение стандартного и параллельного алгоритма.

1 Аналитическая часть

Матрица - математический объект, эквивалентный двумерному массиву. Числа располагаются в матрице по строкам и столбцам. Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

1.1 Алгоритм Винограда

Рассмотрим два вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$. Их скалярное произведение равно:

$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4$

Это равенство можно переписать в виде:

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

Менее очевидно, что 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. Это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

1.2 Параллельный алгоритм Винограда

Трудоемкость алгоритма Винограда имеет сложность $O(nmk)$ для умножения матриц $n1 \times m1$ на $n2 \times m2$. Чтобы улучшить алгоритм, следу-

ет распараллелить ту часть алгоритма, которая содержит 3 вложенных цикла.

Вычисление результата для каждой строки не зависит от результата выполнения умножения для других строк. Поэтому можно распараллелить часть кода, где происходят эти действия. Каждый поток будет выполнять вычисления определенных строк результирующей матрицы.

1.3 Распараллеливание задачи

В рамках данной лабораторной работы производилось распараллеливание задачи по потокам. В CPU для данной цели используются treads.

CPU – central processing unit – это универсальный процессор, также именуемый процессором общего назначения. Он оптимизирован для достижения высокой производительности единственного потока команд. Доступ к памяти с данными и инструкциями происходит преимущественно случайным образом. Для того, чтобы повысить производительность CPU еще больше, они проектируются специально таким образом, чтобы выполнять как можно больше инструкций параллельно. Например для этого в ядрах процессора используется блок внеочередного выполнения команд. Но несмотря на это, CPU все равно не в состоянии осуществить параллельное выполнение большого числа инструкций, так как расходы на распараллеливание инструкций внутри ядра оказываются очень существенными. Именно поэтому процессоры общего назначения имеют не очень большое количество исполнительных блоков.

1.3.1 Вывод

Были рассмотрены поверхностно стандартная и параллельная реализации алгоритма Винограда.

2 | Конструкторская часть

Требования к вводу:
На вход подаются две матрицы и их размерности
Требования к программе:
Корректное умножение двух матриц

На рис. 2.4 представлена IDEF0 диаграмма умножения двух матриц:

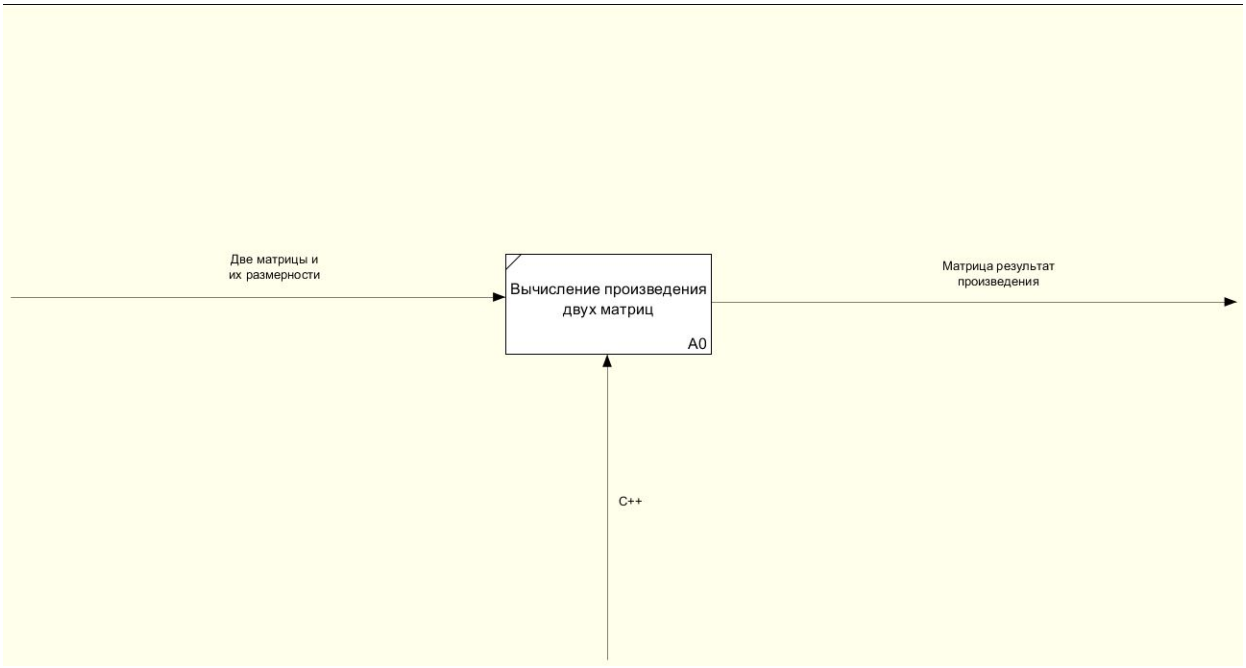

Рис. 2.1: IDEF0 диаграмма умножения двух матриц

2.1 Схемы алгоритмов

На рис. 2.4 представлена схема стандартного алгоритма Винограда:

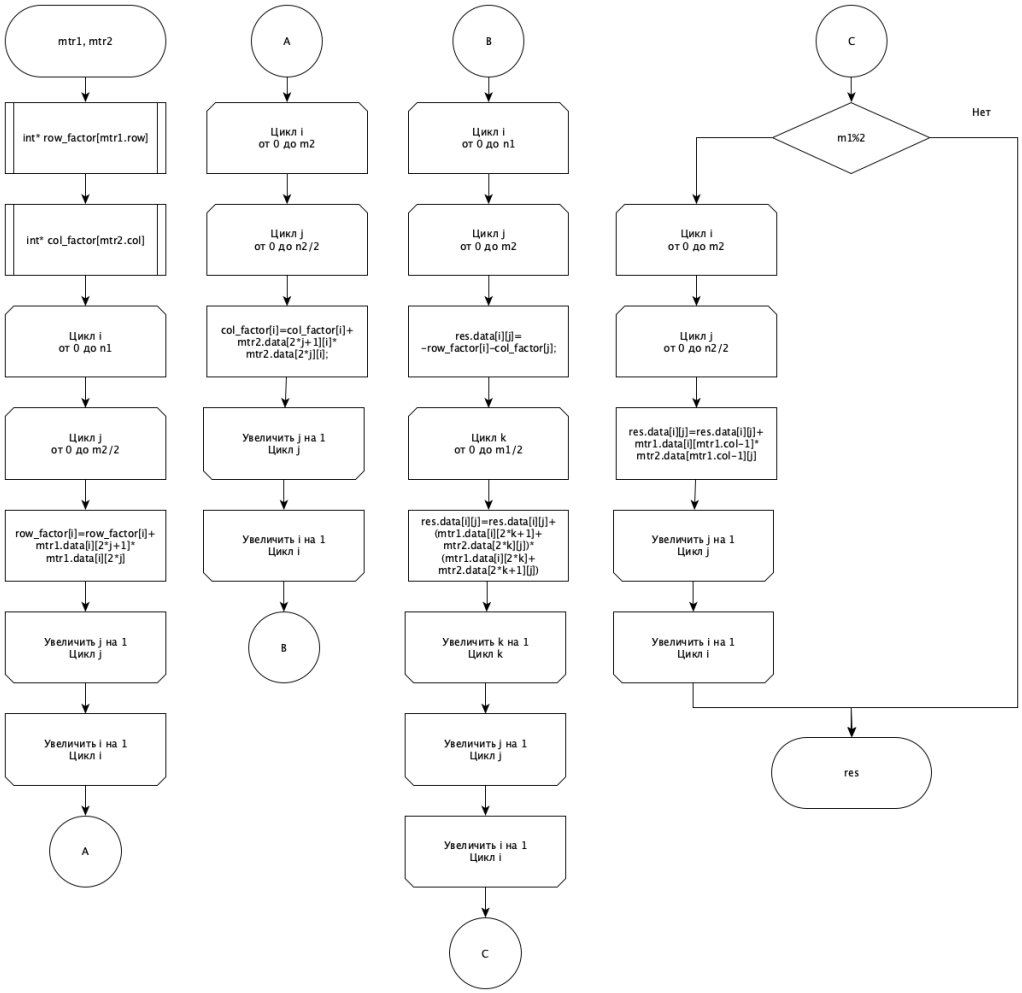

Рис. 2.2: Схема стандартного алгоритма Винограда

На рис. 2.4 представлена схема распараллеленого алгоритма Винограда:

На рис. 2.4 ппредставлена схема фнкции параллельного вычисления матрицы:

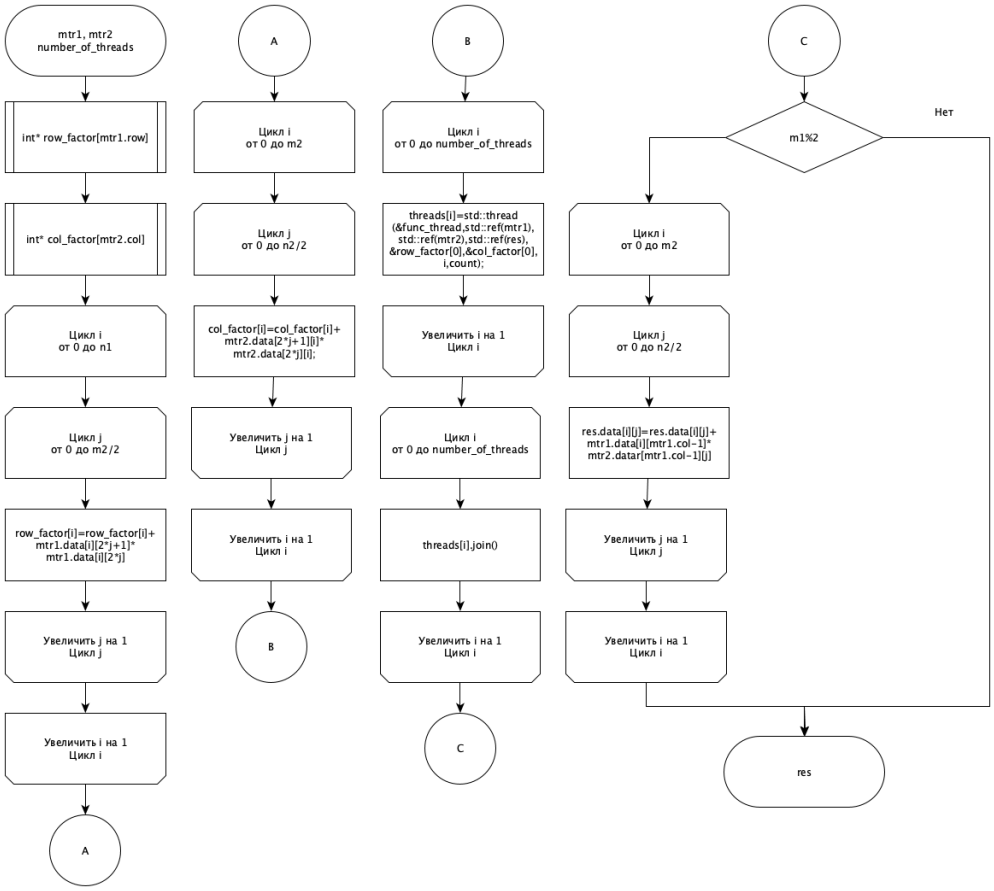

Рис. 2.3: Схема распараллеленого алгоритма Винограда

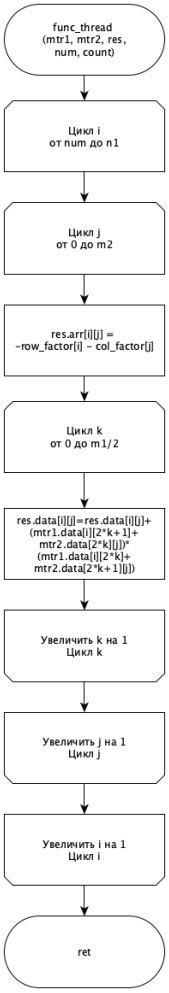

Рис. 2.4: Схема распараллеленого алгоритма Винограда

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрала язык программирования C++, так имею большой опыт работы с ним. Среда разработки - Visual Studio.

Для отключения оптимизации компилятора указан флаг «-o0».
Для работы с потоками используется библиотека поддержки потоков std::thread.
Наблюдатель joinable проверяет потенциальную возможность работы потока в параллельном контексте.
Операция join ожидает завершения потока.

Для замера процессорного времени используется функция, возвращающая количество тиков.

Листинг 3.1: Функция получения тиков

```
unsigned long long getTicks(void)
{
    unsigned long long d;
    __asm__ __volatile__ ("rdtsc" : "=A" (d) );
    return d;
}
```

3.2 Реализация алгоритма

Листинг 3.2: Стандартный алгоритм Винограда

```
Matrix Matrix::vinograd_mult(const Matrix &mtr1, const
    Matrix &mtr2)
{
    Matrix res(mtr1.row, mtr2.column);

    int row_factor[mtr1.row];
    int col_factor[mtr2.column];

    for(int i = 0; i < mtr1.row; i++)
    {
        row_factor[i] = 0;
        for(int j = 0; j < mtr1.column / 2; j++)
        {
            row_factor[i] = row_factor[i] + mtr1.data[i][2
                * j] * mtr1.data[i][2 * j + 1];
        }
    }

    for(int i = 0; i < mtr2.column; i++)
    {
        col_factor[i] = 0;
        for(int j = 0; j < mtr2.row / 2; j++)
        {
            col_factor[i] = col_factor[i] + mtr2.data[2 * j
                ][i] * mtr2.data[2 * j + 1][i];
        }
    }

    for(int i = 0; i < mtr1.row; i++)
    {
        for(int j = 0; j < mtr2.column; j++)
        {
            res.data[i][j] = -row_factor[i] - col_factor[j
                ];
            for(int k = 0; k < mtr1.column / 2; k++)
            {
                res.data[i][j] = res.data[i][j] +
                                 (mtr1.data[i][2 * k +
                                    1] + mtr2.data[2 * k
```

```
                                                ][j]) *
35                                             (mtr1.data[i][2 * k] +
                                                mtr2.data[2 * k +
                                                1][j]);
36              }
37          }
38      }
39
40      if(mtr1.column % 2)
41      {
42          for(int i = 0; i < mtr1.row; i++)
43              for(int j = 0; j < mtr2.column; j++)
44                  res.data[i][j] = res.data[i][j] + mtr1.data
                        [i][mtr1.column - 1] * mtr2.data[mtr1.
                        column - 1][j];
45      }
46
47      return res;
48  }
```

В листинге 3.3 представлен параллельный алгоритм Винограда:

Можно заметить, что вычисление результата для каждой строки происходит независимо от результата выполнения умножения для других строк. Поэтому возможно распараллелить участок кода, соответстующий строкам 24-32 листинга 3.2. Каждый поток будет выполнять вычисление некоторых строк результирующей матрицы. Это сделано потому, что проход по строкам матрицы является более эффективным с точки зрения организации данных в памяти.

Листинг 3.3: Параллельный алгоритм Винограда

```
1  Matrix Matrix::vinograd_mult_multithreading(const Matrix &
       mtr1, const Matrix &mtr2, int count)
2  {
3
4      Matrix res(mtr1.row, mtr1.column);
5
6      int row_factor[mtr1.row];
7      int col_factor[mtr1.column];
8
```

```
9       for(int i = 0; i < mtr1.row; i++)
10      {
11          row_factor[i] = 0;
12          for(int j = 0; j < mtr1.column / 2; j++)
13          {
14              row_factor[i] = row_factor[i] + mtr1.data[i][2
                    * j + 1] * mtr1.data[i][2 * j];
15          }
16      }
17
18      for(int i = 0; i < mtr1.column; i++)
19      {
20          col_factor[i] = 0;
21          for(int j = 0; j < mtr1.column / 2; j++)
22          {
23              col_factor[i] = col_factor[i] + mtr1.data[2 * j
                    + 1][i] * mtr1.data[2 * j][i];
24          }
25      }
26
27      std::thread threads[count];
28
29      for(int i = 0; i < count; i++)
30      {
31          threads[i] = std::thread(&func_thread, std::ref(
                mtr1), std::ref(mtr1), std::ref(res), &
                row_factor[0], &col_factor[0], i, count);
32      }
33
34      for(int i = 0; i < count; i++)
35      {
36          if(threads[i].joinable())
37          {
38              threads[i].join();
39          }
40      }
41
42      if(mtr1.column % 2)
43      {
44          for(int i = 0; i < mtr1.row; i++)
```

```
45              for(int j = 0; j < mtr1.column; j++)
46                  res.data[i][j] = res.data[i][j] + mtr1.data
                        [i][mtr1.column - 1] * mtr1.data[mtr1.
                        column - 1][j];
47      }
48
49      return res;
50  }
```

В листинге 3.4 представлено распараллеленное вычисление тройного цикла:

Листинг 3.4: Распараллеленный тройной цикл

```
1  void func_thread(const Matrix &mtr1, const Matrix &mtr2,
       Matrix &res, int* row_factor, int* col_factor, int num
       , int count)
2  {
3      for(int i = num; i < mtr1.row; i += count)
4      {
5          for(int j = 0; j < mtr2.column; j++)
6          {
7              res.data[i][j] = -row_factor[i] - col_factor[j
                   ];
8              for(int k = 0; k < mtr1.column / 2; k++)
9              {
10                 res.data[i][j] = res.data[i][j] + (mtr1.
                       data[i][2 * k + 1] + mtr2.data[2 * k][j
                       ]) * mtr1.data[i][2 * k] + mtr2.data[2 *
                        k + 1][j];
11             }
12         }
13     }
14 }
```

4 Исследовательская часть

4.1 Постановка эксперимента

Были проведены исследования зависимости времени работы трех алгоритмов от размеров перемножаемых матриц и количества использованных потоков. Замеры времени проводились для матриц четной размерности размером от 100 до 1000 с шагом 100 и матриц нечетной размерности размером от 101 до 1001 с шагом 100. Количество потоков - от 1 до 32, т.к. компьютер, на котором проводились вычисления, содержит 8 логических ядра.

Временные замеры проводятся путём многократного проведения эксперимента и деления результирующего времени на количество итераций эксперимента.

4.2 Сравнительный анализ на основе замеров времени работы алгоритмов

На рис. 4.1 представлен график времени работы алгоритма на матрицах четной размерности:

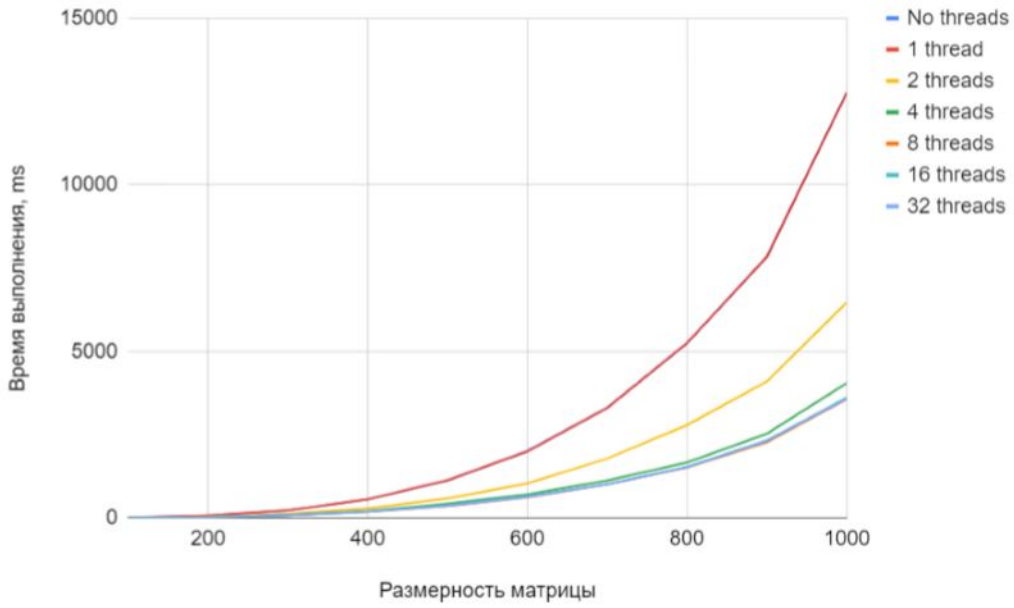

Рис. 4.1: График времени работы на матрицах четной размерности

На рис. 4.2 представлен график времени работы алгоритма на матрицах нечетной размерности:

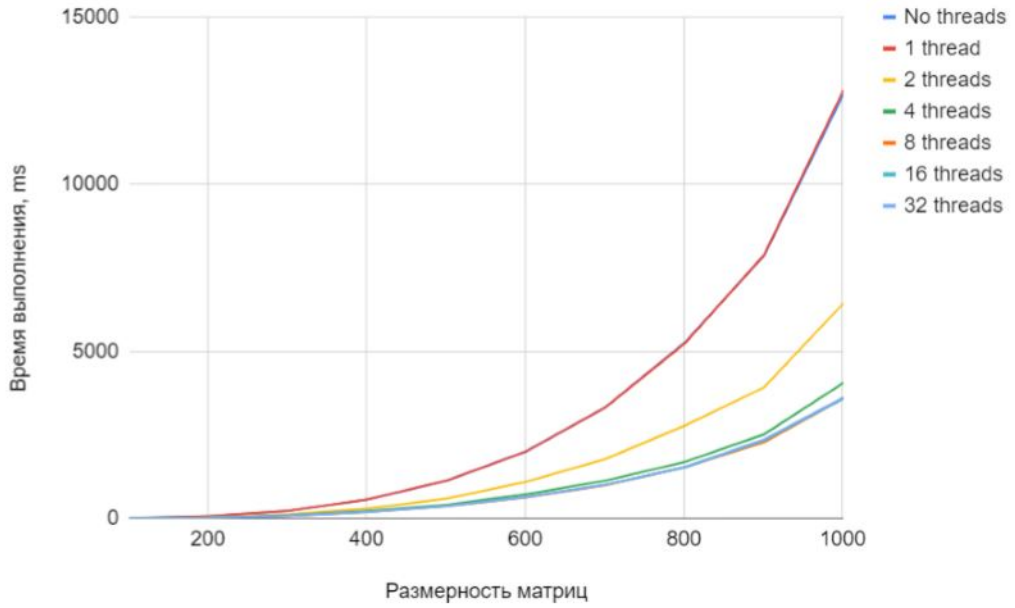

Рис. 4.2: График времени работы на матрицах нечетной размерности

4.3 Вывод

Эксперименты замера времени показали, что при последовательной и параллельной (с одним рабочим потоком) реализациях оптимизированного алгоритма Винограда совсем немного выигрывает последовательная реализация (в ней не тратится время на выделение рабочего потока).

При сравнении замеров времени для параллельной реализации алгоритма с 1, 2, 4, 8, 16 и 32 рабочими потоками выяснилось, что максимальная производительность достигается на 8-ми рабочих потоках , что равно количеству логических потоков компьютера, на котором производились замеры. Выполнение алгоритма на 8-ми рабочих потоках быстрее в 3,82 раз, по сравнению с выполнением на 1-м потоке для матриц размера 1000×1000. При большем количестве рабочих потоков происходит небольшое падение производительности (тратится время на создание новых рабочих потоков, но вычисления будут производиться с той же скоростью, что и при 8 рабочих потоках).

Заключение

В ходе работы был изучен параллельный алгоритм умножения матриц: алгоритм Винограда. Выполнено сравнение зависимости всех рассматриваемых алгоритмов от числа параллельных потоков и размера матриц. В ходе исследования было установлено, что многопоточный алгоритм Винограда выполняется быстрее, чем стандартный алгоритм.