Министерство образования и науки 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)

ОТЧЕТ
По лабораторной работе №4
По курсу «Анализ алгоритмов»
Тема: «Распараллеливание вычислений»

Студент: Зейналов З. Г.
Группа: ИУ7-51
Преподаватель: Волкова Л.Л.

Москва, 2019г.

Оглавление

Введение

При работе в высоконагруженной системе, обработке большого объема данных или при решении ресурсоемких задач неизбежно возникает вопрос повышения производительности системы. Добиться этого можно за счет:

- увеличения производительности оборудования,
- оптимизации алгоритмов обработки данных.

Увеличение производительности оборудования – путь дорогой и, в целом, не самый оптимальный. Оптимизация алгоритмов обработки данных выглядит предпочтительней, но и она имеет свой предел. Но и тот и другой путь не дадут революционного эффекта.

Однако если логика решаемой задачи позволяет распараллелить процессы обработки данных, то распараллеливание - это наиболее и эффективный способ кратного увеличения производительности системы[1].

Поток — это основная единица, которой операционная система выделяет время процессора. Каждый поток имеет приоритет планирования и набор структур, в которых система сохраняет контекст потока, когда выполнение потока приостановлено. Контекст потока содержит все сведения, позволяющие потоку безболезненно возобновить выполнение, в том числе набор регистров процессора и стек потока. Несколько потоков могут выполняться в контексте процесса. Все потоки процесса используют общий диапазон виртуальных адресов. Поток может исполнять любую часть программного кода, включая части, выполняемые в данный момент другим потоком[2].

Целью данной лабораторной работы является исследование многопоточности с использованием алгоритма Копперсмита-Винограда.

Задачи работы:

Задачами данной лабораторной работы являются:

1) Научиться писать многопоточные программы;
2) Применить полученные знания на практике с использованием алгоритма Копперсмита-Винограда в несколько потоков.
3) Провести замеры времени работы однопоточной и многопоточной реализаций и проанализировать полученные данные.

1. **Аналитическая часть**

В данном разделе будут представлены описания алгоритмов, формулы и оценки сложностей алгоритмов

1.1 Алгоритм умножения Копперсмита-Винограда

Алгоритм Копперсмита—Винограда — алгоритм умножения квадратных матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом (англ.). В исходной версии асимптотическая сложность алгоритма составляла $O(n^{2,3755})$, где n — размер стороны матрицы. Алгоритм Копперсмита—Винограда, с учетом серии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц[3].

Если рассмотреть результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим 2 вектора: $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$

Их скалярное произведение равно:

$$(V * W) = v1 * w1 + v2 * w2 + v3 * w3 + v4 * w4 \ (1)$$

Что эквивалентно

$$(V * M) = (v1 + w2) * (v2 + w1) + (v3 * w4) * (v4 + w3) -$$

$$-v1 * v2 - v3 * v4 - w1 * w2 - w3 * w4 \ (2)$$

Не очевидным остается тот факт, что выражение в правой части формулы 2 допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого стобца второй, что позволяет выполнять для каждого элемента лишь первые два умножения и последующие 5 сложений, а также два сложения.

2. Конструкторская часть

В данном разделе будут размещены схемы алгоритмов.

2.1 Разработка алгоритмов

На рисунках 1 – 5 приведены схемы алгоритма, демонстрирующего работу.

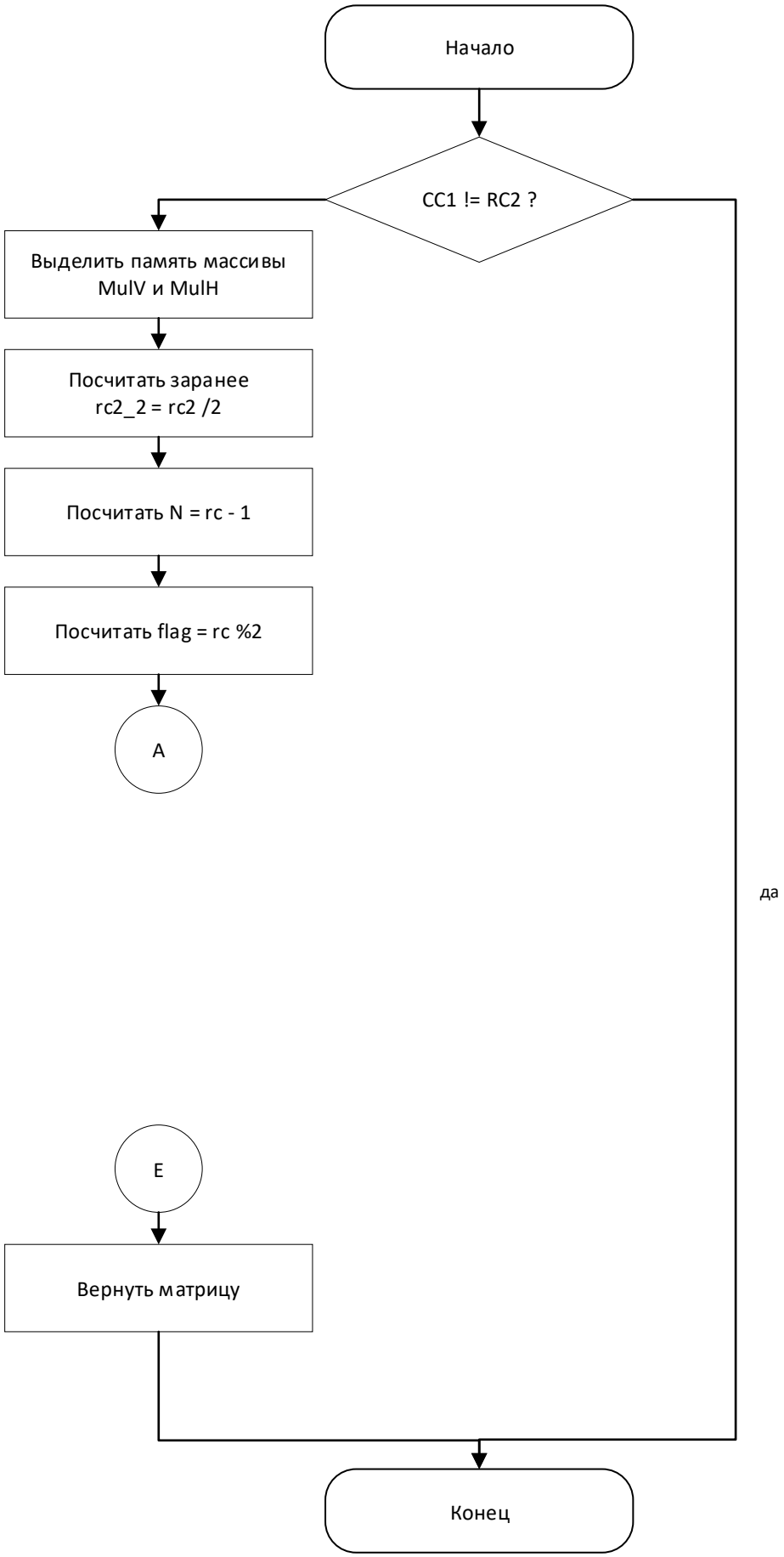

Рисунок 1. Оптимизированный алгоритм умножения Винограда (часть 1)

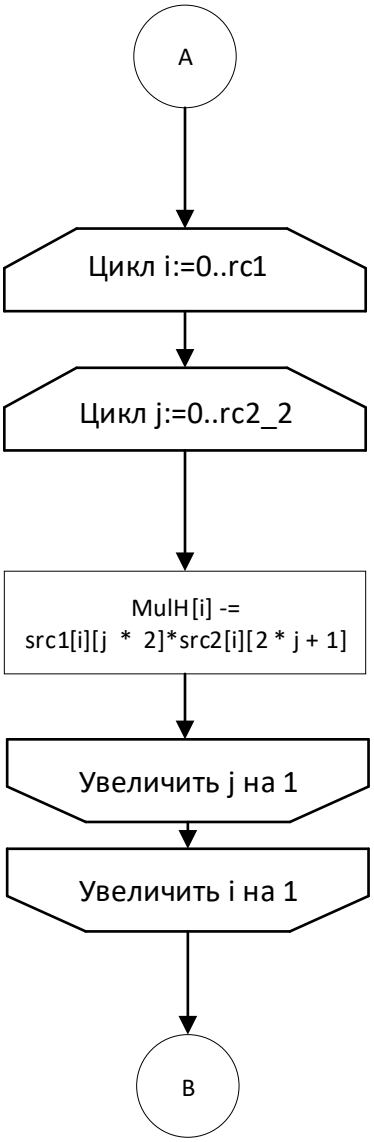

Рисунок 2. Оптимизированный алгоритм умножения Винограда (часть 2)

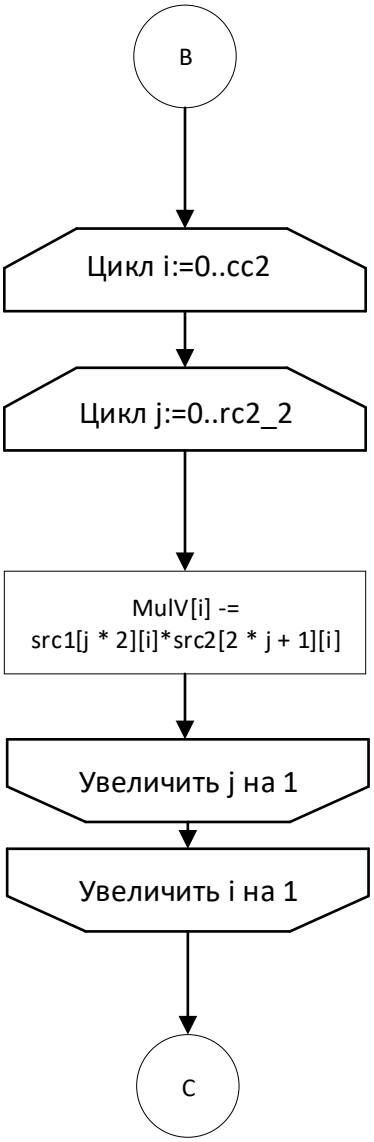

Рисунок 3. Оптимизированный алгоритм умножения Винограда (часть 3)

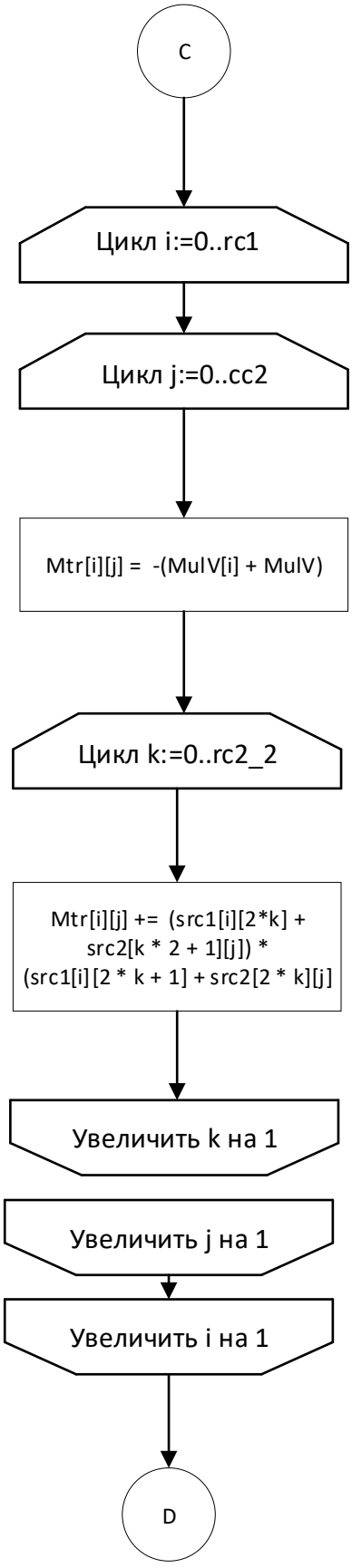

Рисунок 4. Оптимизированный алгоритм умножения Винограда (часть 4)

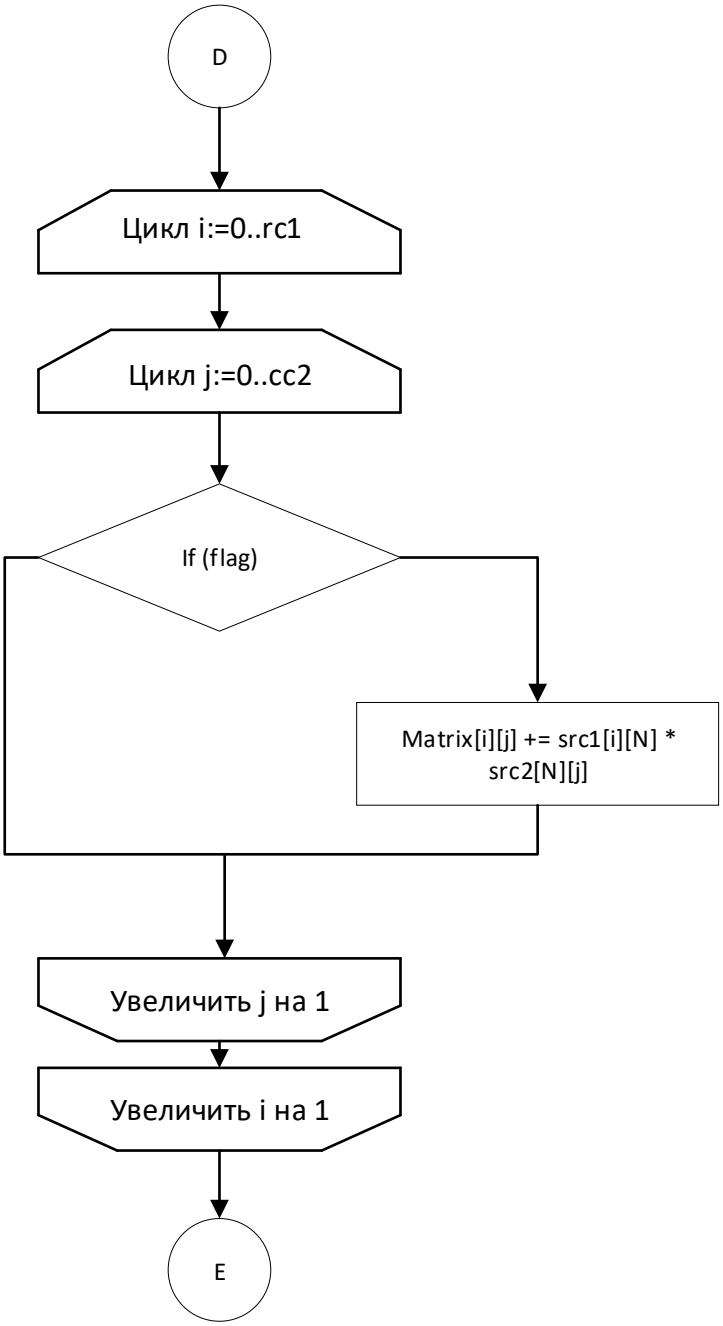

Рисунок 5. Оптимизированный алгоритм умножения Винограда (часть 5)

3 Технологическая часть

В данном разделе будут приведены Требования к программному обеспечению, средства реализации, листинг кода и примеры тестирования.

3.1 Средства реализации

В качестве языка программирования был выбран C++ (компилятор g++) в связи с его широким функционалом и быстротой работы, а так же благодаря привычному для меня синтаксису и семантики языка. Среда разработки - Qt. Для работы с потоками использовалась библиотека thread. Время работы процессора замеряется с помощью функции std::chrono().

3.2 Листинг кода

В листинге 1 представлена однопоточная реализация. Однако для многопоточной реализации алгоритма, функция была разбита на 3 части, представленные в листингах 2 – 4. В листинге 5 представлена уже двупоточная реализация. Разбиение на большее число потоков выглядит абсолютно аналогично.

Листинг 1. Реализация оптимизированного Алгоритма Винограда

```
int optimizedMultiplication(int ***result, int &rc, int &cc, int**
src1, int **src2, int rc1, int cc1, int rc2, int cc2)
{
    if (cc1 != rc2)
    {
        return ERR_MTR_SIZE;
    }
    if (src1 == NULL || src2 == NULL)
    {
        return ERR_EMPTY_MATRIX;
    }
    int rc2_2 = rc2 >> 1;                                    // opti
    rc = rc1;
    cc = cc2;
    int **tmpMtr = allocateMatrix(rc, cc);
    int *MulH = (int*)calloc(rc1, sizeof(int));
    int *MulV = (int*)calloc(cc2, sizeof(int));
    for (int i = 0; i < rc1; i++)
    {
        for (int j = 0; j < rc2_2; j++)                      // opti
        {
            MulH[i] -= src1[i][j * 2] * src1[i][2 * j + 1];  // opti
        }
    }
    for (int i = 0; i < cc2; i++)
    {
        for (int j = 0; j < rc2_2; j++)                      // opti
        {
            MulV[i] -= src2[j * 2][i] * src2[2 * j + 1][i];  // opti
        }
    }
    int N = rc2 - 1;                                         // opti
    bool flag = rc2 % 2;                                     // opti
    for (int i = 0; i < rc1; i++)
    {
        for (int j = 0; j < cc2; j++)
        {
            tmpMtr[i][j] = MulH[i] + MulV[j];
            for (int k = 0; k < rc2_2; k++)                  // opti
            {
                tmpMtr[i][j] += (src1[i][k * 2] + src2[2 * k + 1][j])
* (src1[i][2 * k + 1] + src2[2 * k][j]);                     // opti
            }
        }
    }
    for (int i = 0; i < rc1; i++)
    {
        for (int j = 0; j < cc2; j++)
        {
            if (flag)                                        // opti
            {
                tmpMtr[i][j] += src1[i][N] * src2[N][j];     // opti
            }
        }
    }
    *result = tmpMtr;
}
```

Листинг 1 - Часть вычисления вектора строки

```
void first_part(Matrix src1, int rows_begin ,int rc1, int rc2_2,
std::vector<int> &MulH)
{
    for (int i = rows_begin; i < rc1; i++)
    {
        for (int j = 0; j < rc2_2; j++)
        {
            MulH[i] -= src1.matrix[i][j * 2] * src1.matrix[i][2 * j +
1];
        }
    }
}
```

Листинг 3 - Часть вычисления вектора столбца

```
void second_part(Matrix src2, int columns_begin, int cc2, int rc2_2,
std::vector<int> &MulV)
{
    for (int i = columns_begin; i < cc2; i++)
    {
        for (int j = 0; j < rc2_2; j++)
        {
            MulV[i] -= src2.matrix[j * 2][i] * src2.matrix[2 * j +
1][i];
        }
    }
}
```

Листинг 4 -- Часть перемножения

```
void last_part(Matrix src1, Matrix src2, std::vector<int> &MulH,
std::vector<int> &MulV, int rows_begin, int rc1, int cc2, int rc2_2,
Matrix &mtr_res)
{
    int N = src2.rows_count - 1;
    bool flag =  src2.rows_count % 2;
    int rc = rc1;
    int cc = cc2;
    mtr_res.rows_count = rc1;
    mtr_res.columns_count = cc2;
    int **tmpMtr = allocateMatrix(rc, cc);
    for (int i = 0; i < rc1; i++)
    {
        for (int j = 0; j < cc2; j++)
        {
            tmpMtr[i][j] = MulH[i] + MulV[j];
            for (int k = 0; k < rc2_2; k++)
            {
                tmpMtr[i][j] += (src1.matrix[i][k * 2] + src2.matrix[2
* k + 1][j]) * (src1.matrix[i][2 * k + 1] + src2.matrix[2 * k][j]);
            }
        }
    }

    for (int i = 0; i < rc1; i++)
    {
        for (int j = 0; j < cc2; j++)
        {
            if (flag)
            {
                tmpMtr[i][j] += src1.matrix[i][N] * src2.matrix[N][j];
            }
        }
    }
    mtr_res.matrix = tmpMtr;
}
```

В листинге 5 представлена двупоточная реализация алгоритма. Реализации для 4, 8, 16 потоков имеют аналогичную реализацию.

Листинг 5 – двупоточная реализация алгоритма.

```
int multi2(Matrix mtr1, Matrix mtr2, Matrix &mtr_res)
{
    if (mtr1.columns_count != mtr2.rows_count)
    {
        return CANT_MULTIPLY;
    }

    int rows_count_first = mtr1.rows_count;
    int rows_count_second_2 = mtr2.rows_count / 2;
    int columns_count_second = mtr2.columns_count;

    mutex m1rc, m2rc_2, m2cc;
    mutex m1, m2, m_out;
    mutex row, column;
    mutex outout;

    std::vector<int> MulH(mtr1.rows_count);
    std::vector<int> MulV(mtr2.columns_count);

    thread thread_1_1(first_part, mtr1, 0,
                      rows_count_first,
                      rows_count_second_2,
                      ref(MulH));
    thread thread_2_1(second_part, mtr2, 0,
                      columns_count_second,
                      rows_count_second_2,
                      ref(MulV));

    if (thread_1_1.joinable() && thread_2_1.joinable())
    {
        thread_1_1.join();
        thread_2_1.join();
    }

    thread thread_3_1(last_part, mtr1,
                      mtr2,
                      ref(MulH),
                      ref(MulV), 0,
                      rows_count_first / 2,
                      columns_count_second,
                      rows_count_second_2,
                      ref(mtr_res));
    thread thread_3_2(last_part, mtr1,
                      mtr2,
                      ref(MulH),
                      ref(MulV), rows_count_first / 2,
                      rows_count_first,
                      columns_count_second,
                      rows_count_second_2,
                      ref(mtr_res));
    thread_3_1.join();
    thread_3_2.join();
   }
```

4 Экспериментальная часть

В данном разделе будут приведены постановка эксперимента и сравнительный анализ алгоритмов на основе экспериментальных данных. Анализ производился на персональном компьютере с процессором с 4 ядрами, 4 потоками.

4.1 Постановка эксперимента по замеру времени

Для произведения замеров времени выполнения реализаций алгоритмов будет использована следующая формула $t = \frac{Tn}{N}$, где t – время выполнения, N – количество замеров. Неоднократное измерение времени необходимо для построения более гладкого графика.

Количество замеров будет взято равным 100.

Тестирование будет проведено на одинаковых входных данных. 1) Матрицы размерностями от 100x100 до 500x500 с шагом 100.

4.2 Сравнительный анализ на материале экспериментальных данных

Ниже на рисунке 6 приведен график зависимости временных затрат от размеров входных данных.

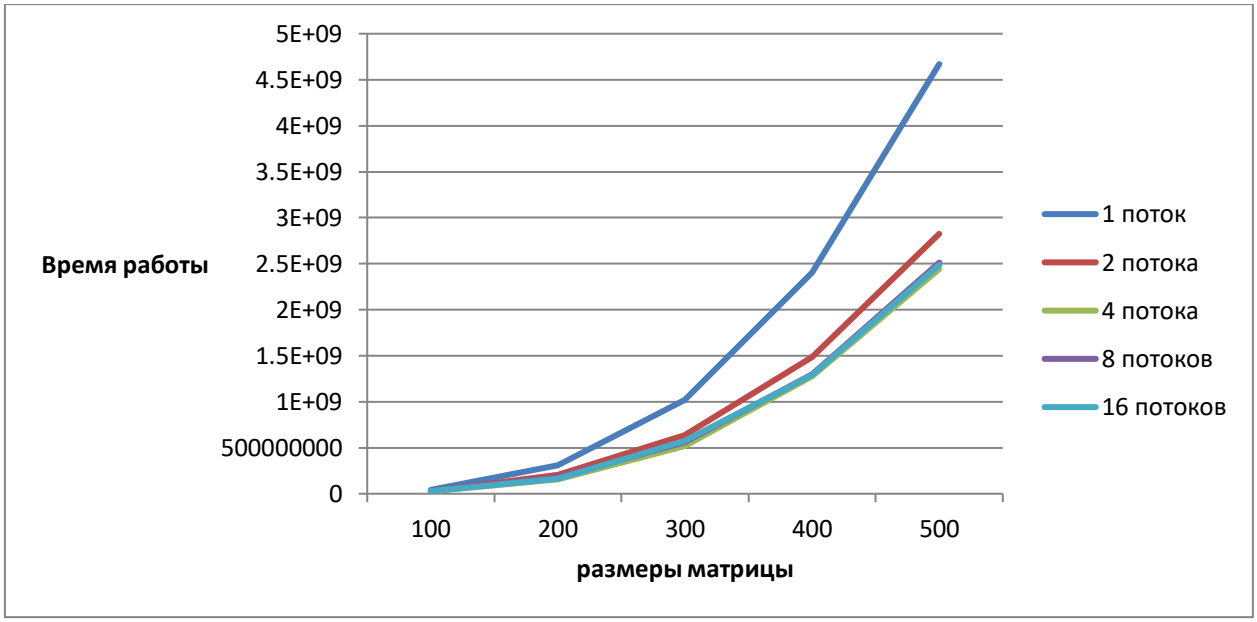

Рисунок 6 - Сравнительный временной анализ реализации алгоритмы с разным числом потоков

Вывод

Многопоточные программы работают быстрее однопоточных, причем разница во времени достигает почти 2 раз. Однако эффективность многопоточной реализации зависит от нагруженности процессора. Чем больше нагрузка, тем меньше эффективность нескольких потоков. Наиболее эффективным количеством потоков оказалось число равное 4, равное количеству ядер. При дальнейшем разбиении на потоки выигрыша по времени не наблюдается, в связи с издержками ожидания процессорного времени для потоков.

Заключение

В ходе работы была изучена и реализована многопоточная реализация алгоритма Копперсмита-Винограда. Использование мьютексов и семафоров в данной лабораторной работе не потребовалось в связи с тем, что разделяемая память не использовалась. Наиболее эффективным по времени количество потоков разбиения в программе должно соответствовать количеству ядер используемого компьютера.

Литература

1) Многопоточная обработка данных[Электронный ресурс]// Производительность. URL: https://infostart.ru/public/947222/
2) Потоки и работа с ними[Электронный ресурс]// Microsoft Docs. URL: https://docs.microsoft.com/ru-ru/dotnet/standard/threading/threads-and-threading

3) Henry Cohn, Robert Kleinberg, Balazs Szegedy, and Chris Umans. Group-theoretic Algorithms for Matrix Multiplication. arXiv:math.GR/0511460. *Proceedings of the 46th Annual Symposium on Foundations of Computer Science*, 23-25 October 2005, Pittsburgh, PA, IEEE Computer Society, стр. 379—388.