МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
КАФЕДРА МОЭВМ

ОТЧЕТ
по лабораторной работе №6
по дисциплине «Параллельные алгоритмы»
ТЕМА: УМНОЖЕНИЕ МАТРИЦ

Студент Степаненко Д. В.
Преподаватель Татаринов Ю. С.

Санкт-Петербург
2023

Цель.

Опираясь, на полученные знания использования библиотеки MPI для параллельных вычислений, распараллелить алгоритм перемножения матриц. Проанализировать его эффективность, используя различные подходы.

Постановка задачи (вариант 2).

Выполнить задачу умножения двух квадратных матриц A и B размера $m \times m$, результат записать в матрицу C. Реализовать последовательный и параллельный ленточный алгоритм 2 (горизонтальные и вертикальные полосы). Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Выполнение работы.

Первым делом в функции *main()* мы считываем аргумент – размер матрицы. Главное условие алгоритма при распараллеливании – размер матрицы должен быть кратен количеству запускаемых процессов. Далее для генерации случайных чисел подключается модуль time. На основе времени генерируются случайные числа в диапазоне от 0 до 9 и создаются матрицы. Матрицы представлены в виде одномерного массива (для удобной работы с топологиями).

Если количество запущенных процессов равно одному, то запускается последовательный алгоритм умножения матриц. Само перемножение происходит в функции *sequentialMatrixMultiplication()*. В ней происходит классическое перемножение строки на столбец и поэлементное суммирование.

Если процессов больше одного, то запускается параллельный алгоритм. Для более удобной работы со столбцами матрицы B, транспонируем ее. Вычислим некоторые константы:

procUseCount – количество ячеек в буфере, измеряемое в размерах матрицы

elemetsPerBlock – размер блока в ячейках (элементах)

Далее инициализуются буферы, в которые при помощи функции MPI_Scatter() рассылаются блоки матрицы A и B. После создается новая топология – одномерное кольцо. Это наиболее удобная топология, т.к. алгоритм подразумевает последовательный зацикленный обмен столбцами матрицы B между соседними процессами.

Следующий шаг – реализация основного цикла алгоритма: каждый процесс size раз перемножает строку матрицы A на строку транспонированной матрицы B, сохраняя значение в буфер C. Далее происходит циклический сдвиг, и процесс пересылает следующему значения буфера B с помощью функции MPI_Sendrecv_replace(). По окончании цикла собираются все буферы с результатом в полную матрицу C функцией MPI_Gather(). Далее выводится матрица C и освобождаются коммуникатор, ресурсы системы. Сеть Петри построенного алгоритма смотреть на рис. 2.

Таким образом, каждый процесс берет на себя задачу вычисления определенного блока матрицы C. Наглядное представление алгоритма представлено на рис. 1. В этом примере алгоритм работает на четырех процессах с квадратной матрицей размера 4.

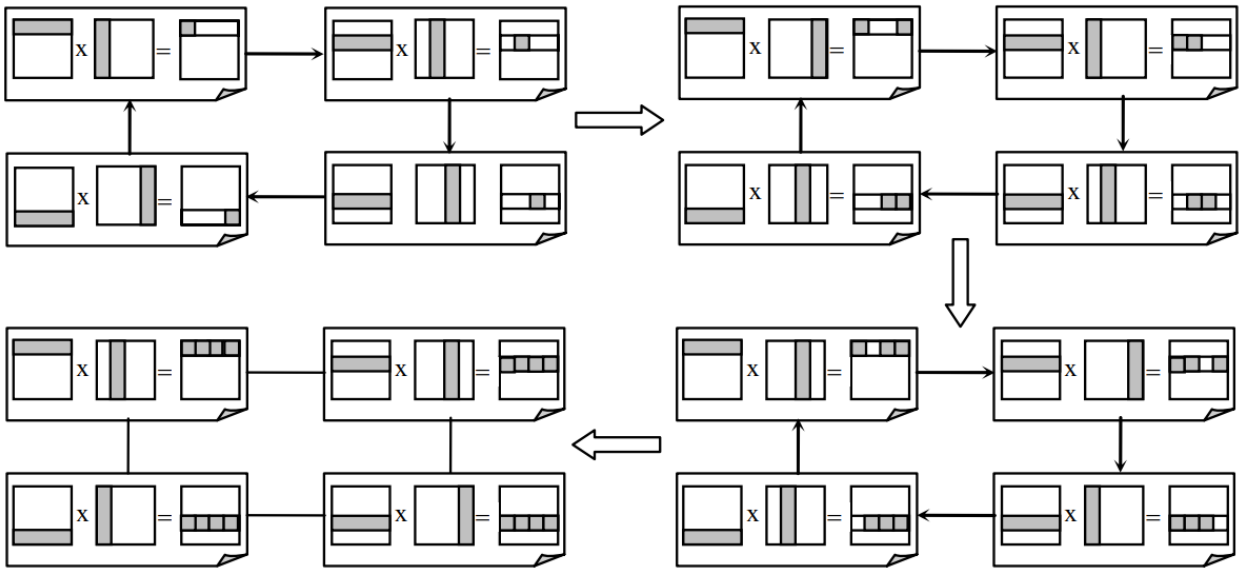

Рисунок 1 – Визуализация работы алгоритма

Задача умножения матриц хорошо масштабируема с использованием данного алгоритм. Вычисляемые блоки для каждого процесса имеют

одинаковые размеры благодаря условию, что размер матрицы кратен числу процессов. Когда размер матрицы больше числа процессов, вычисляемые блоки укрупняются путем объединения строк матрицы A и столбцов матрицы B. Вычислить количество элементов в одном блоке можно по формуле:

$$\frac{\text{длина матрицы}^2}{\text{количество процессов}}$$

Такой подход позволяет обеспечить равномерность распределения вычислительной нагрузки по процессорам вычислительной системы.

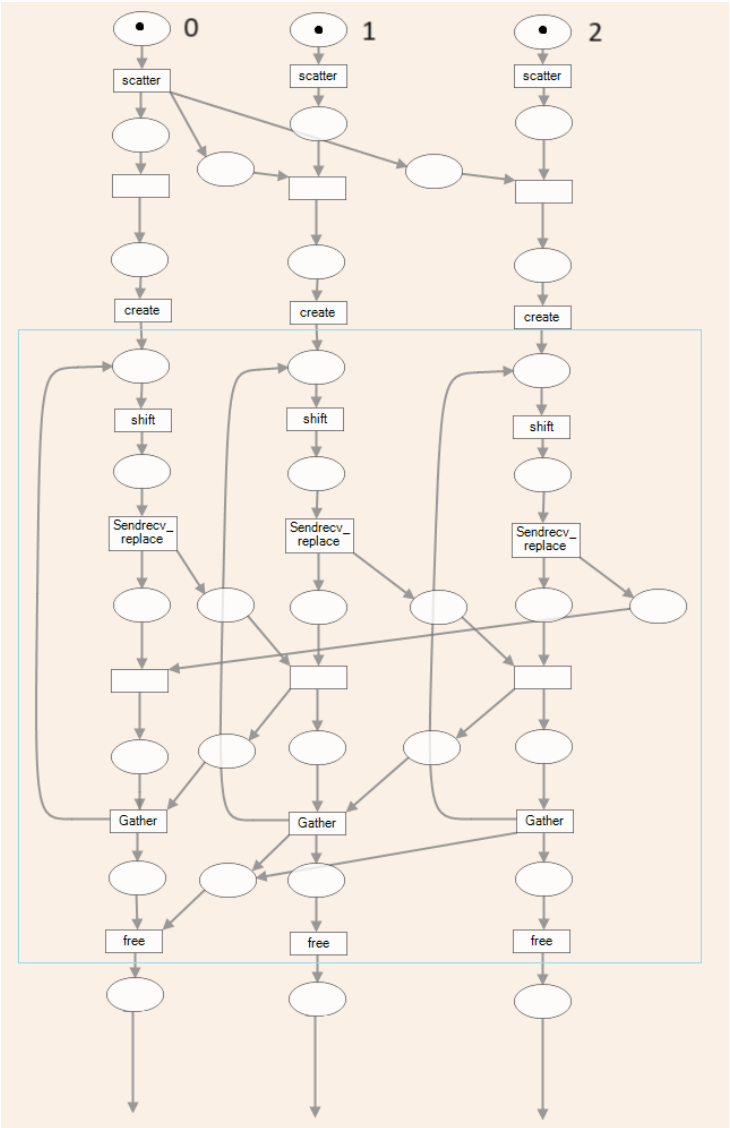

Рисунок 2 - Сеть Петри основной параллельной части программы для трех процессов

Листинг программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>


int *createMatrix (int size) {
    int *matrix;
    if (( matrix = malloc(size*size*sizeof(int))) == NULL) {
        printf("Malloc error");
        exit(1);
    }

    for (int i=0; i<size*size; i++) {
        matrix[i] = rand() % 10;
    }

    return matrix;
}

void sequentialMatrixMultiplication(int MatrixA[], int MatrixB[], int
MatrixC[], int size) {
    int index;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            index = i*size+j;
            MatrixC[index] = 0;
            for (int k = 0; k < size; k++) {
                MatrixC[index]       +=       MatrixA[i*size+k]       *
MatrixB[k*size+j];
            }
        }
    }
}

void printMatrix (int *matrix, int dim) {
    for (int i=0; i<dim; i++) {
        for (int j=0; j<dim; j++) {
            printf("%d ", matrix[i*dim+j]);
        }
        printf("\n");
    }
}

void transpose(int Matrix[], int size)
{
    int t;
    for(int i = 0; i < size; ++i){
        for(int j = i; j < size; ++j){
            t = Matrix[i*size+j];
            Matrix[i*size+j] = Matrix[j*size+i];
            Matrix[j*size+i] = t;
        }
    }
}
```

```
int main(int argc, char** argv) {
    int dim = atoi(argv[argc - 1]);
    int size, rank;

    //для генерации
    srand(time(NULL));

    //генерация матриц
    int *MatrixA  = createMatrix(dim);
    int *MatrixB  = createMatrix(dim);
    int *MatrixC  = createMatrix(dim);


    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    if (size == 1) {
        //последовательный алгоритм
        printf("Matrix A:\n");
        printMatrix(MatrixA, dim);

        printf("Matrix B:\n");
        printMatrix(MatrixB, dim);

        sequentialMatrixMultiplication(MatrixA,    MatrixB,    MatrixC,
dim);
        printf("Matrix C:\n");
        printMatrix(MatrixC, dim);
    }

    else if (size > 1 && dim%size==0) {
        //параллельный ленточный алгоритм
        if(rank ==0){
            printf("Matrix A:\n");
            printMatrix(MatrixA, dim);

            printf("Matrix B:\n");
            printMatrix(MatrixB, dim);

            transpose(MatrixB, dim);
        }

        MPI_Status Status;
        // сколько раз придется задействовать каждый буфер
        int procUseCount = dim/size;
        // размер всех ячеек из матрицы A и B для одного процесса
        int elemetsPerBlock = procUseCount*dim;

        // буферы
        int bufA[elemetsPerBlock];
        int bufB[elemetsPerBlock];
        int bufC[elemetsPerBlock];
        for (int i = 0; i < elemetsPerBlock; ++i){
            bufA[i] = 0;
```

```
            bufB[i] = 0;
            bufC[i] = 0;
        }

        //рассыдка каждому поцессу своей части матриц A и B
        MPI_Scatter(MatrixA,    elemetsPerBlock,    MPI_INT,    &bufA,
elemetsPerBlock, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Scatter(MatrixB,    elemetsPerBlock,    MPI_INT,    &bufB,
elemetsPerBlock, MPI_INT, 0, MPI_COMM_WORLD);

        //создание нового коммуникатора (одномерное кольцо)
        MPI_Comm ring_comm;
        int dims[1] = {size};
        int periods[1] = {1};
        int reorder = 1;
        MPI_Cart_create(MPI_COMM_WORLD,  1,  dims,  periods,  reorder,
&ring_comm);
        MPI_Comm_rank(ring_comm, &rank);

        int source, destination, index;
        for (int p=0; p < size; p++) {
            //расчет результата
            for (int i=0; i < procUseCount; i++) {
                for (int j=0; j < procUseCount; j++) {
                    index = i*dim+j+((rank+size-p)*procUseCount)%dim;
                    for (int k=0; k < dim; k++) {
                        bufC[index] += bufA[i*dim+k]*bufB[j*dim+k];
                    }
                }
            }
            //получение ранка процесса для передачи блока-столбца
            MPI_Cart_shift(ring_comm, 0, 1, &source, &destination);
            //пересылка столбца
            MPI_Sendrecv_replace(&bufB,    elemetsPerBlock,    MPI_INT,
destination, 0, source, 0, ring_comm, &Status);
        }

        //Сбор всех частей матрицы C в единую
        MPI_Gather(&bufC,    elemetsPerBlock,    MPI_INT,    MatrixC,
elemetsPerBlock, MPI_INT, 0, ring_comm);

        if (rank == 0) {
            printf("Matrix C:\n");
            printMatrix(MatrixC, dim);
        }
        MPI_Comm_free(&ring_comm);
    }
    else{
        if (rank == 0)
            printf("Incorrect  data:  check  the  divisibility  of  the
matrix size by the number of processes");
    }

    MPI_Finalize();
    return 0;
}
```

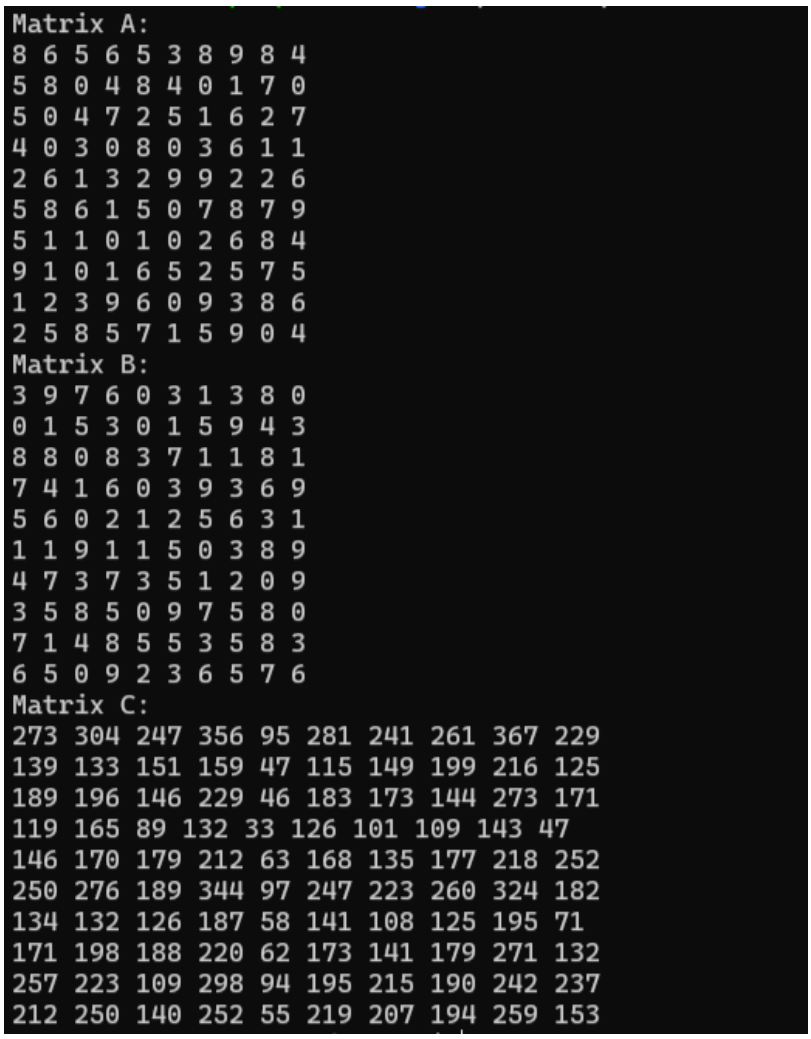

Рисунок 3 – Результат перемножения матриц размера 10 на 5 процессах

Для изменения времени будем использовать функцию MPI_Wtime(). Измерение времени проводим исключительно на алгоритме, который не включает в себя инициализацию матриц и их вывод в консоль. Для измерения ускорения воспользуемся формулой:

$$S_p(n) = T_1(n)/T_p(n)$$

Стоит отметить, что измерение времени более, чем на 12 процессах не имеет смысла. Происходит это потому, что резко возрастает время выполнения программы из-за конкуренции процессов за ресурс (кванты врмени). Таким образом, измерение времени проводится на 1, 2, 4, 5, 8, 10 процессах, которым кратен размер матрицы. Результат измерений смотреть в табл. 1 и рис. 4.

Таблица 1 – Результаты работы программы на разном количестве процессов.

Размерность матриц (m)	Последова-тельный алгоритм	2 процесса		4 процессов	
	время	время	ускорение	время	ускорение
10	0.0039	0.0448	0.0871	-	-
50	0.4150	0.3992	1.0395	-	-
100	3.1072	2.6470	1.177	1.8301	1.6979
200	25.2182	12.0968	2.0826	11.3507	2.2301
500	427.5432	200.4278	2.1330	147.7213	2.894
1000	4112.2851	1596.1093	2.5765	1007.1507	4.0831
10000	-	-	-	-	-

Размерность матриц (m)	5 процессов		8 процессов		10 процессов	
	время	ускорение	время	время	время	ускорение
10	0.1674	0.1674	-	-	0.2008	0.0194
50	0.3516	1.1803	-	-	0.3132	1.325
100	1.2738	2.4466	-	-	0.9063	3.4284
200	8.7486	2.8966	6.7500	2.7333	5.6014	4.5
500	126.3483	3.3837	-	-	68.8541	6.2092
1000	847.2246	4.8539	630.5995	6.5212	503.4883	8.1676
10000	-	-	-	-	-	-

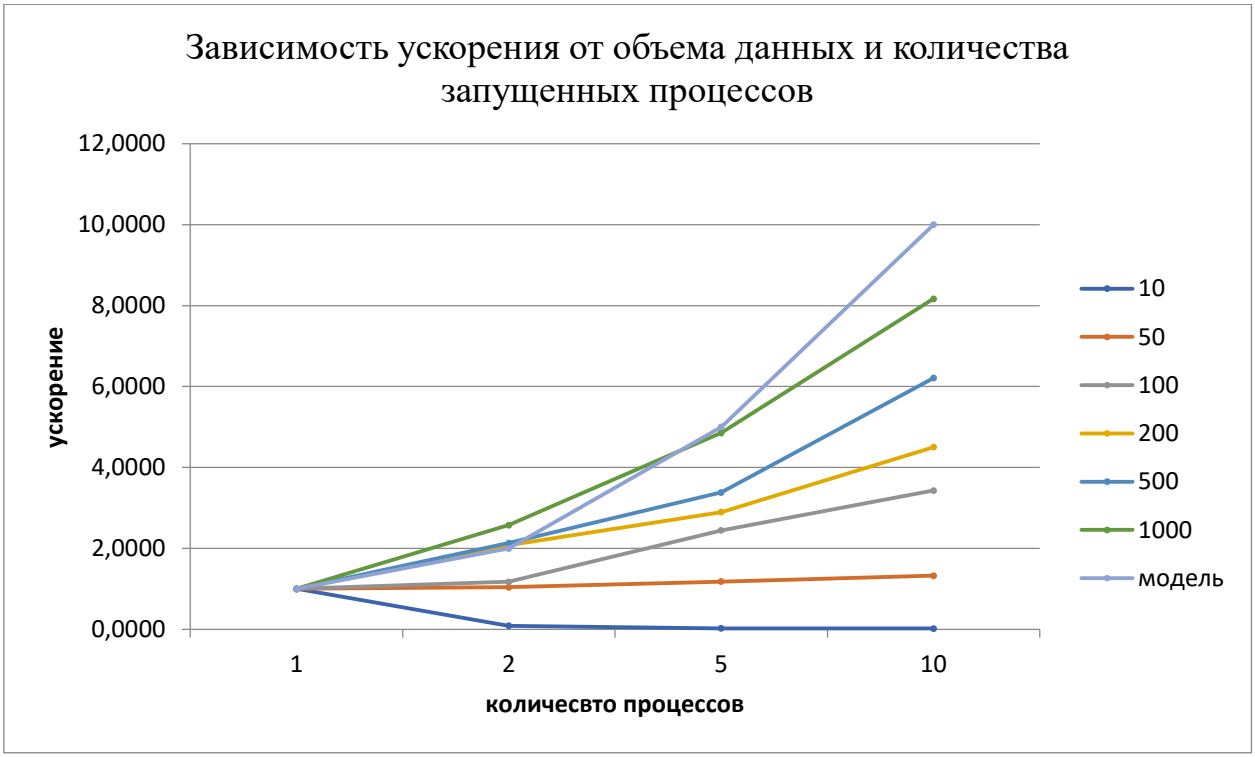

Рисунок 4 – График зависимости ускорения от объема данных и количества запущенных процессов

Время выполнения последовательного алгоритма напрямую зависит только от размера матрицы, т.к. количество операций m^3. Реализация параллельного алгоритма предполагает перемножения элементов линий матриц на каждом шаге. Так как каждый процесс используется m/p раз перемножая строку на столбец размера m, то сложность алгоритма без затрат на передачу данных будет равна m^3/p. Тогда ускорение представляется как:

$$S_p(m) = \frac{m^3}{\left(\frac{m^3}{p}\right)} = p$$

Сравним экспериментальные и теоретические оценки времени выполнения задачи на двух процессах. Для расчета теоретического времени воспользуемся формулой: $m^3/p*C$, где C – константа, учитывающая время длительности 1 такта процессора (на данной вычислительной системе равна 1/ 300000). Результаты смотреть на рис. 5.

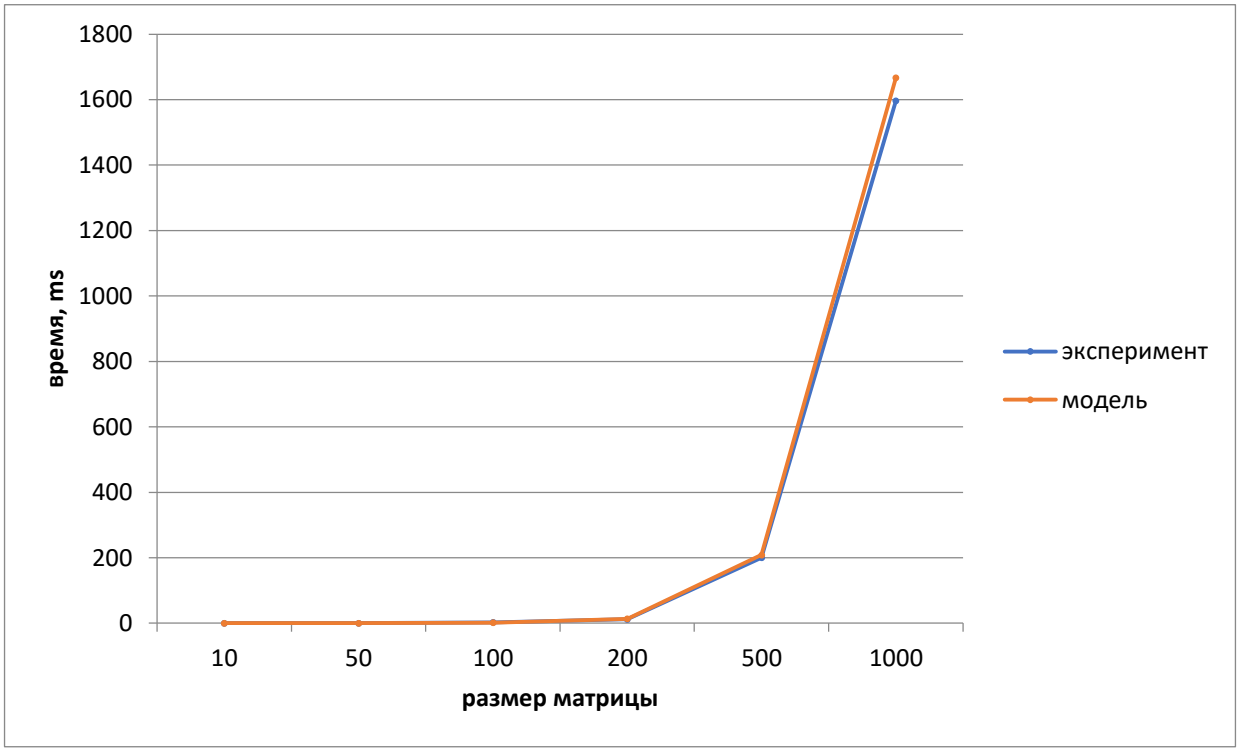

Рисунок 5 – График зависимости скорости выполнения программы от размера матрицы. Сравнение теоретического расчета и эмпирических значений.

Вывод.

В ходе выполнения лабораторной работы были реализованы 2 алгоритма перемножения матриц: последовательный и параллельный. Для реализации параллельного подхода использовался ленточный алгоритм с передачей столбцов. Были исследованы сложности двух алгоритмов: у последовательного $O(m^3)$, у параллельного – $O(m^3/p)$.

Также были проведены измерения времени работы программы в зависимости от количества запущенных процессов и объема данных (размера матрицы). На основе этих данных были найдены ускорения. По полученному графику было видно, что с увеличением объема данных показатели ускорения приближались к теоретическим. Это происходит потому, что затраты на передачу данных становятся меньше, по сравнению с затратами на вычисление блоков данных.

Далее мы сравнили экспериментальное и модельное время работы параллельного алгоритма на 2 процессах. С ростом количества данных экспериментальная модель приближается к теоретическим расчетам. Однако на малом числе процессов данные относительно друг друга сильно отличаются вследствие затрат на передачу буферов, которые не учитывались в теоретической модели.

Таким образом, можно сделать вывод, что для перемножения матриц размерности меньше 50 лучше использовать последовательный алгоритм, а для матриц большей – параллельный.