**­МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Параллельные алгоритмы»**

**Тема: Коллективные операции.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Дубровин Д.Н. |
| Преподаватель |  | Татаринов Ю.С. |

Санкт-Петербург

2025

## Цель работы.

Изучить коллективные операции в библиотеке MPI. Выполнить поставленную задачу, применив некоторые из этих операций на практике.

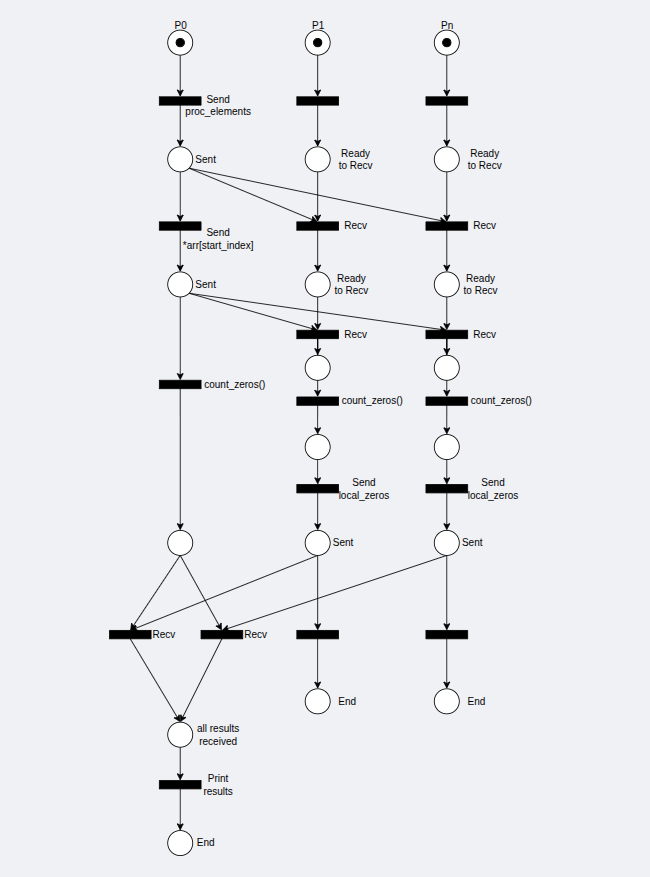
## Задание.

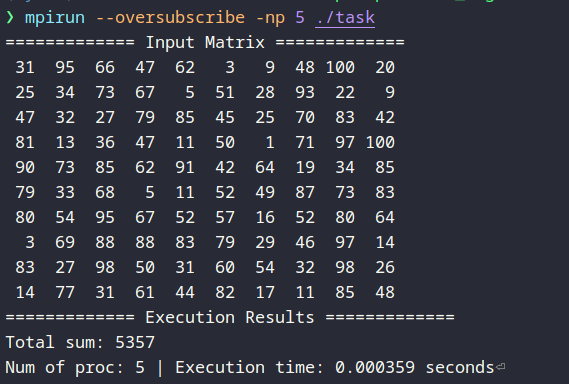
13. Написать и отладить параллельную программу вычисления суммы

элементов матрицы с использованием коллективных операций.

## Выполнение работы.

1. Программа реализует параллельную обработку матрицы с использованием MPI: распределяет строки матрицы между процессами с помощью MPI\_Scatterv, выполняет локальные суммирования над полученными частями матрицы, а затем агрегирует результаты со всех процессов на корневом узле с помощью MPI\_Reduce для получения финального результата.
2. Построим схему Петри:

Риc.1 - сеть Петри

  
Риc.2 — пример работы программы

3. Проведём запуск при малой входной матрице (10 x 10)

Таблица 1- результаты выполнения программы при различном количестве процессов.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| t, сек | 0.000012 | 0.000112 | 0.000074 | 0.000161 | 0.000929 | 0.000471 | 0.000397 | 0.001430 |

4. Построим графики времени выполнения и ускорения для 1-15 процессов.

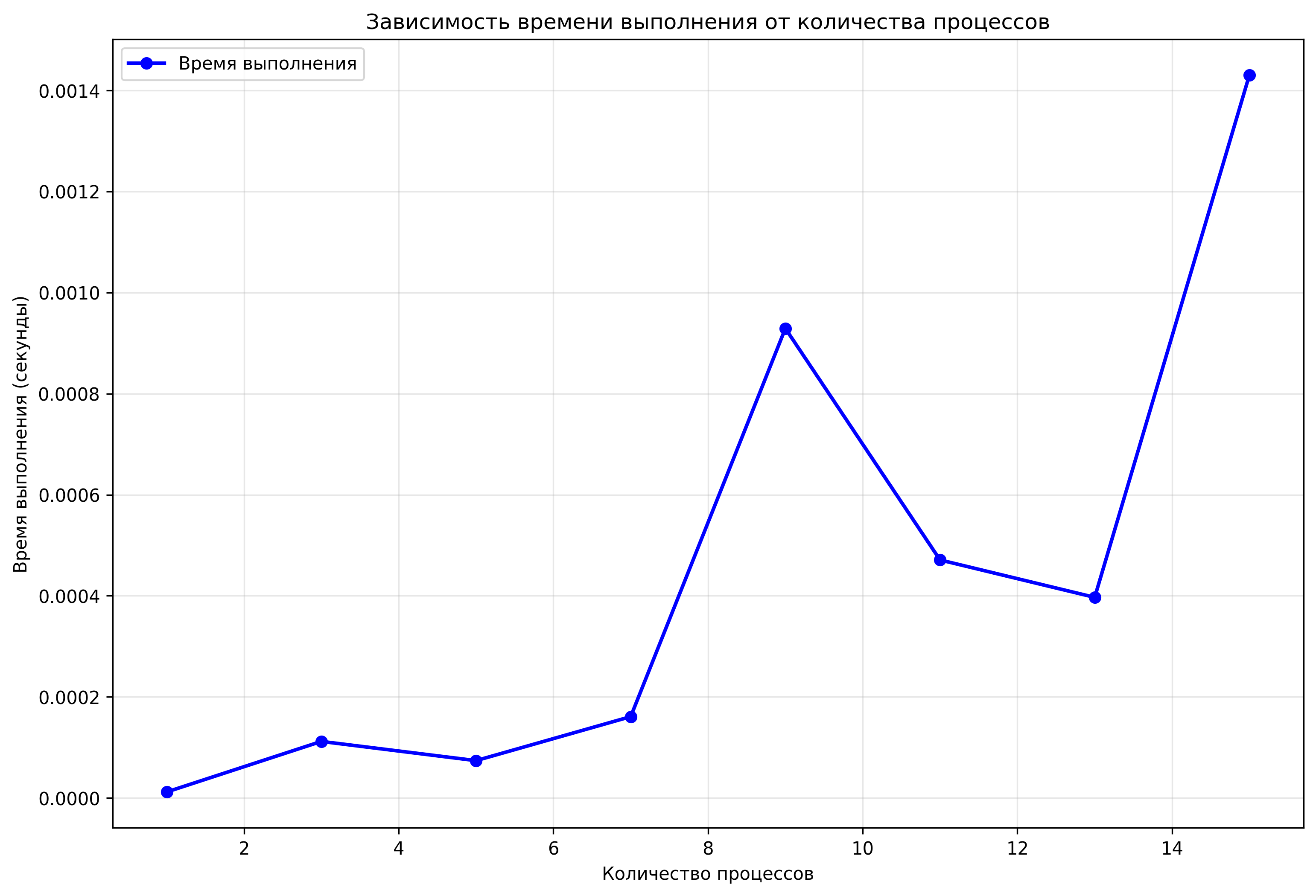
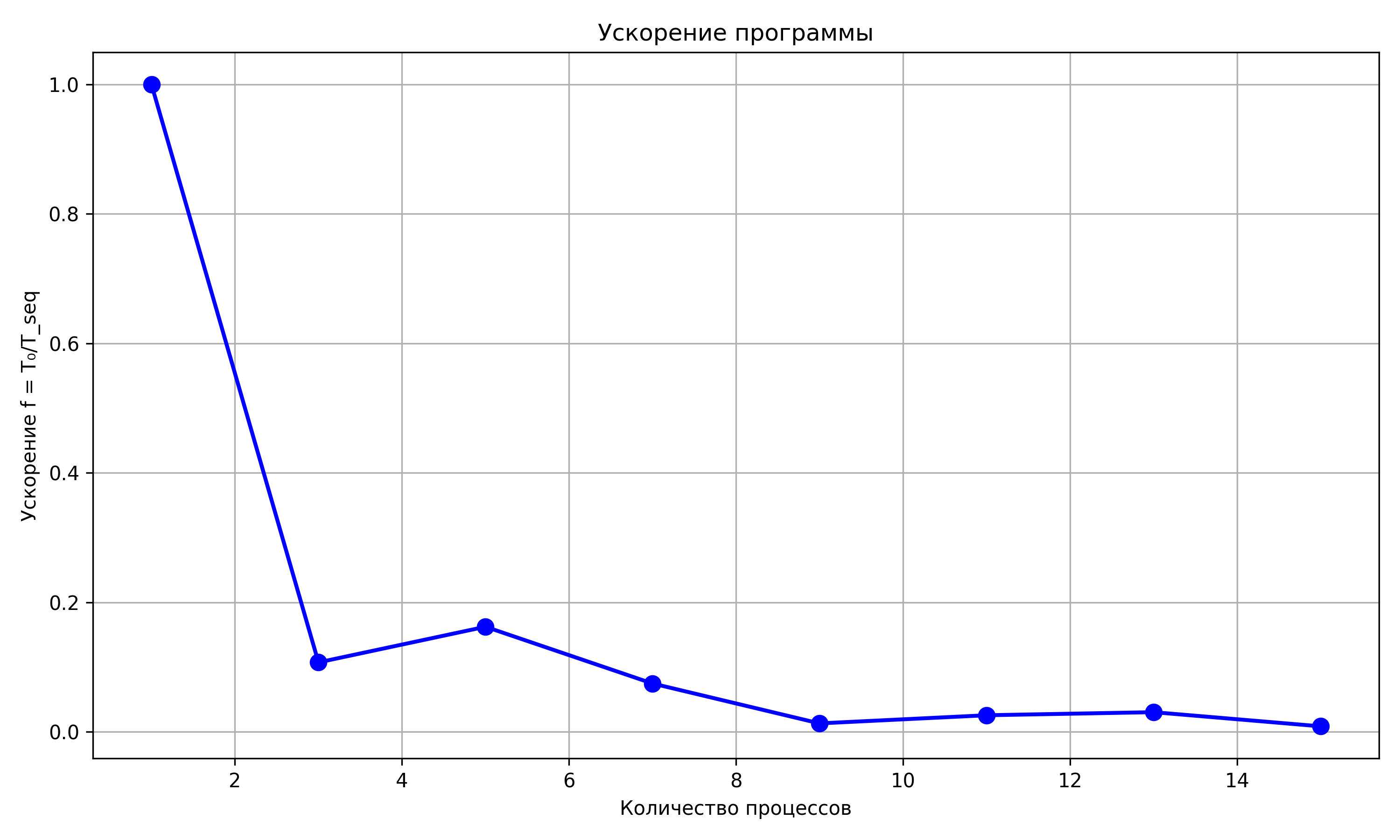


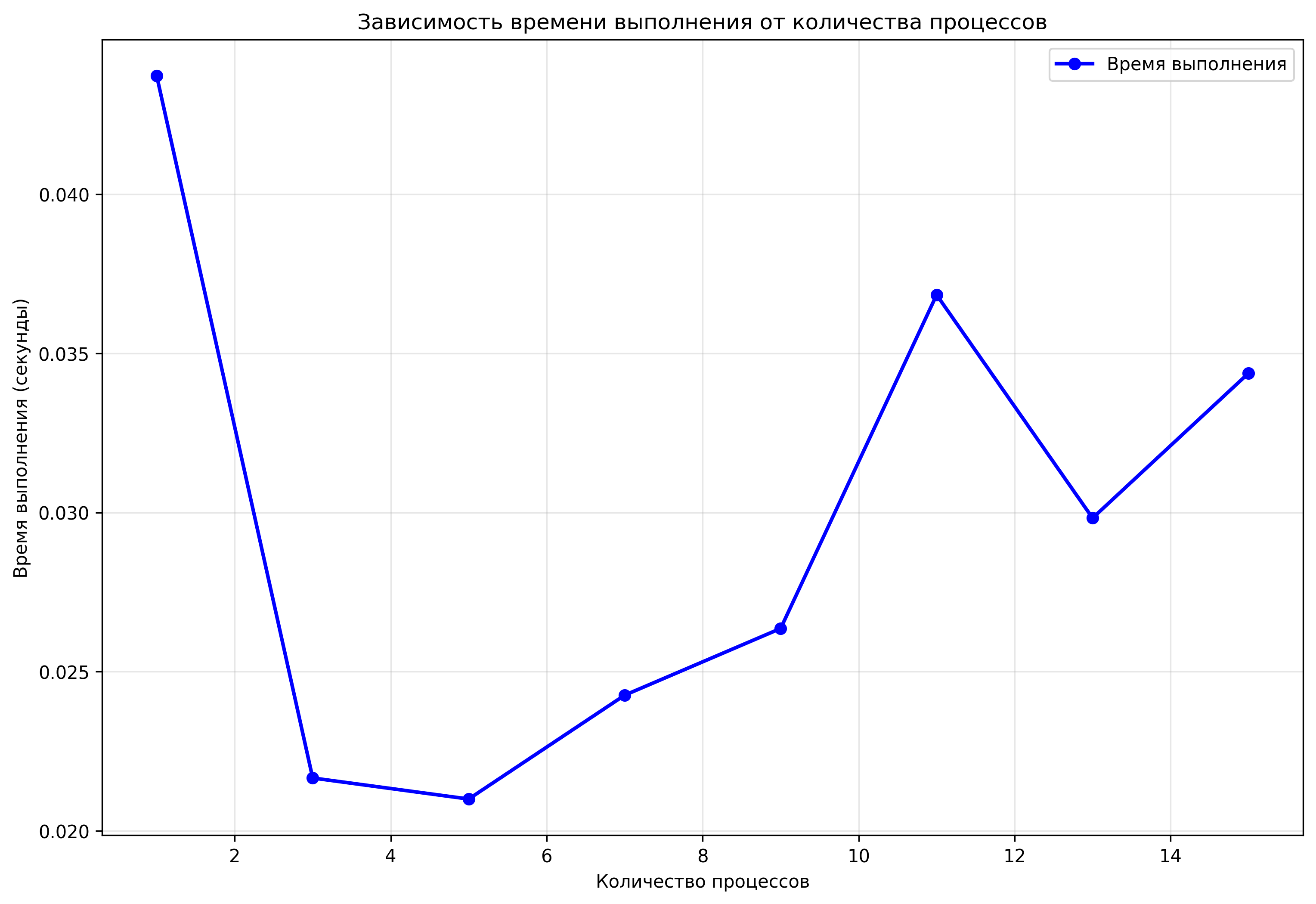
Рис.3 – график зависимости времени выполнения от числа процессов

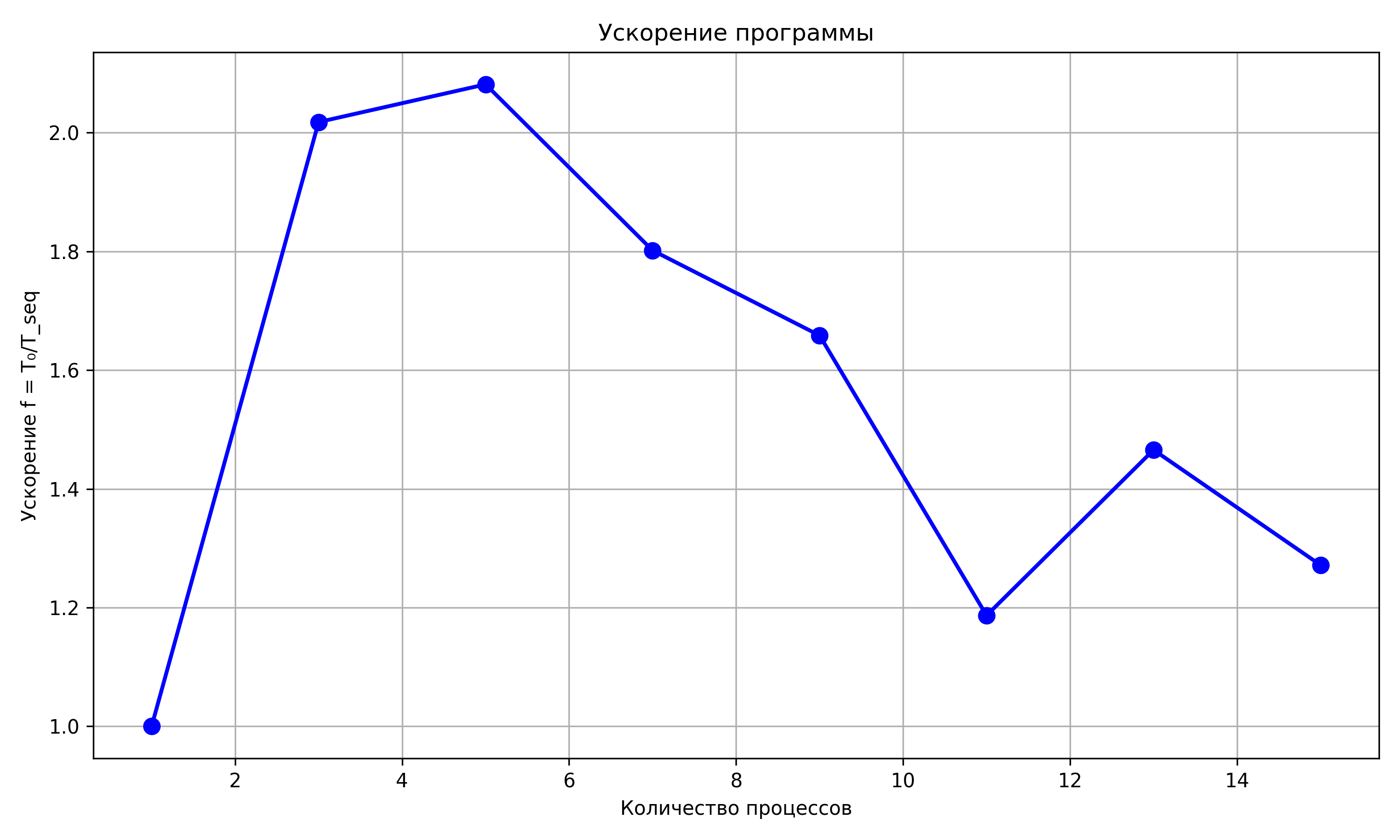
Рис.4 - график ускорения

5. Проведём запуск при большой входной матрице (4000 x 4000) c целью увидеть улучшенные результаты ускорения.

Таблица 2- результаты выполнения программы при различном количестве процессов.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N | 1 | 3 | 5 | 7 | 9 | 11 | 13 | 15 |
| t, сек | 0.043713 | 0.021664 | 0.021003 | 0.024264 | 0.026359 | 0.036835 | 0.029826 | 0.034377 |

Рис.5 – график зависимости времени выполнения от числа процессов

Рис.3 - график ускорения

**Анализ результатов.**

**Разработанный программный код см. в приложении А.**

## Выводы.

В результате выполнения работы была успешно разработана и реализована параллельная программа для обработки матриц с использованием MPI. Также была исследована зависимость ускорения программы от размера входной матрицы.

**Приложение А  
Исходный код программы**

Название файла: task.с

#include <stdio.h>

#include <mpi.h>

#include <stdlib.h>

#include <time.h>

#define BUFF\_SIZE 30

#define ERR\_MALLOC 1

#define ERR\_SEND\_DATA 2

#define ERR\_RECV\_RESULT 3

int count\_zeroes(int \*arr, int size);

int main(int argc, char \*\*argv)

{

int rank, size;

int success;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

if (rank == 0) {

double start\_time, end\_time, exec\_time;

start\_time = MPI\_Wtime();

int \*arr = malloc(BUFF\_SIZE \* sizeof(int));

if (!arr) MPI\_Abort(MPI\_COMM\_WORLD, 1);

srand(time(NULL));

puts("Initial array:");

for (int i = 0; i < BUFF\_SIZE; ++i) {

arr[i] = rand() % 7;

printf("%d ", arr[i]);

}

printf("\n");

int elements\_per\_proc = BUFF\_SIZE / size;

int remainder = BUFF\_SIZE % size;

int start\_index = elements\_per\_proc + (0 < remainder ? 1 : 0);

for (int i = 1; i < size; ++i) {

int proc\_elements = elements\_per\_proc + (i < remainder ? 1 : 0);

if (MPI\_Send(&proc\_elements,1, MPI\_INT, i, 0, MPI\_COMM\_WORLD)

!= MPI\_SUCCESS) {

free(arr); MPI\_Abort(MPI\_COMM\_WORLD, ERR\_SEND\_DATA);

};

if (MPI\_Send(&arr[start\_index], proc\_elements, MPI\_INT, i, 0, MPI\_COMM\_WORLD)

!= MPI\_SUCCESS) {

free(arr); MPI\_Abort(MPI\_COMM\_WORLD, ERR\_SEND\_DATA);

}

start\_index += proc\_elements;

}

int local\_slice\_size = elements\_per\_proc + (0 < remainder ? 1 : 0);

int local\_zeros = count\_zeroes(arr, local\_slice\_size);

printf("Process %d: zeros = %d\n", rank, local\_zeros);

int total\_zeros = local\_zeros;

for (int i = 1; i < size; ++i) {

int proc\_zeroes;

if (MPI\_Recv(&proc\_zeroes, 1, MPI\_INT, MPI\_ANY\_SOURCE, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status)

!= MPI\_SUCCESS) {

free(arr); MPI\_Abort(MPI\_COMM\_WORLD, ERR\_RECV\_RESULT);

}

printf("Process %d: zeros = %d\n", status.MPI\_SOURCE, proc\_zeroes);

total\_zeros += proc\_zeroes;

}

end\_time = MPI\_Wtime();

exec\_time = end\_time - start\_time;

puts("============= Execution Results =============");

printf("Total zeroes: %d\n", total\_zeros);

printf("Num of proc: %d | Execution time: %.6f seconds",

size, exec\_time);

free(arr);

} else {

int local\_slice\_size;

MPI\_Recv(&local\_slice\_size, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

int local\_arr[local\_slice\_size];

MPI\_Recv(local\_arr, local\_slice\_size, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, &status);

int local\_zeros = count\_zeroes(local\_arr, local\_slice\_size);

MPI\_Send(&local\_zeros, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}

int count\_zeroes(int \*arr, int size)

{

int zeros\_count = 0;

for (int i = 0; i < size; ++i) {

if (arr[i] == 0)

zeros\_count++;

}

return zeros\_count;

}