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:

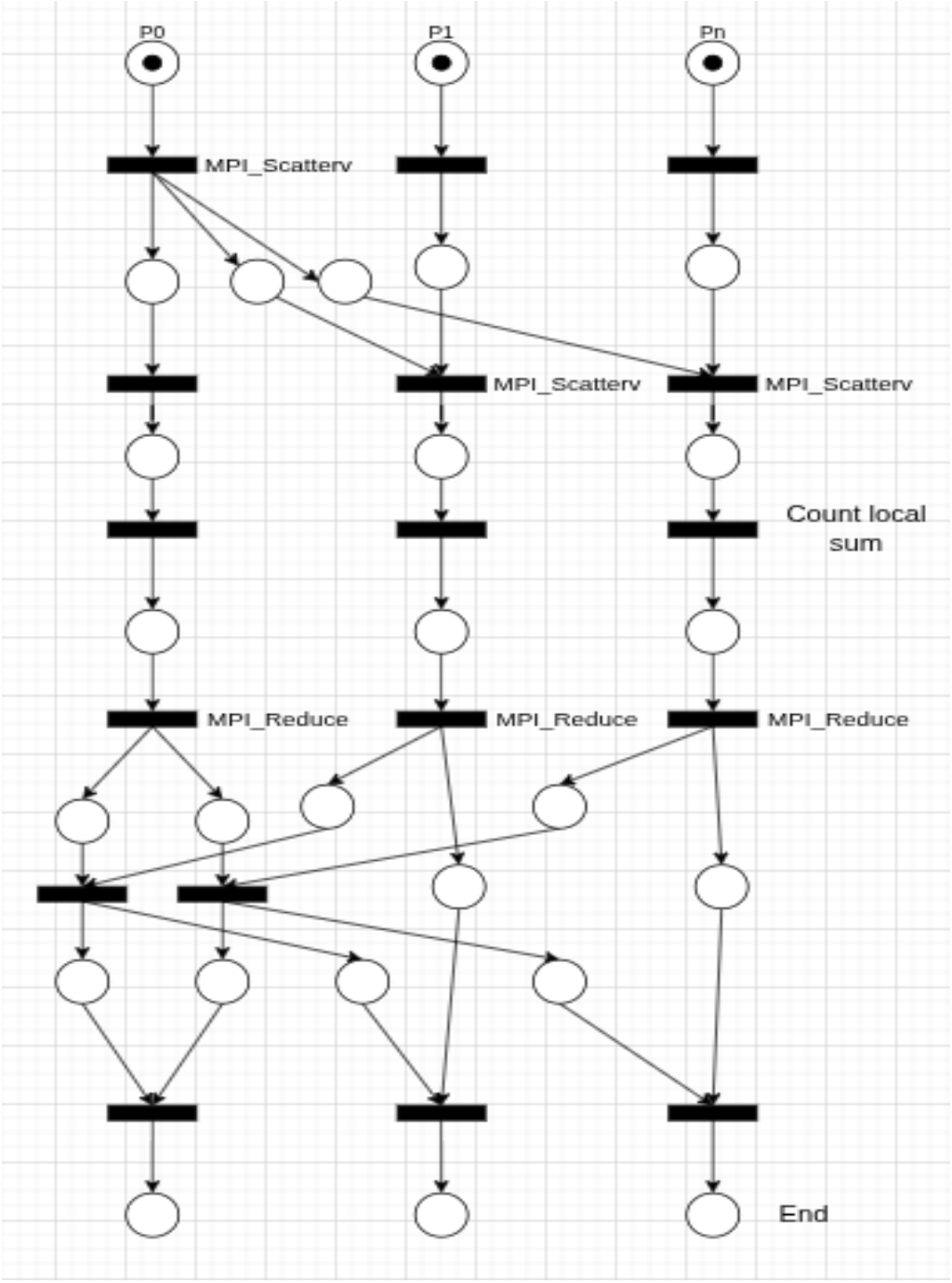

Рис.1 - сеть Петри

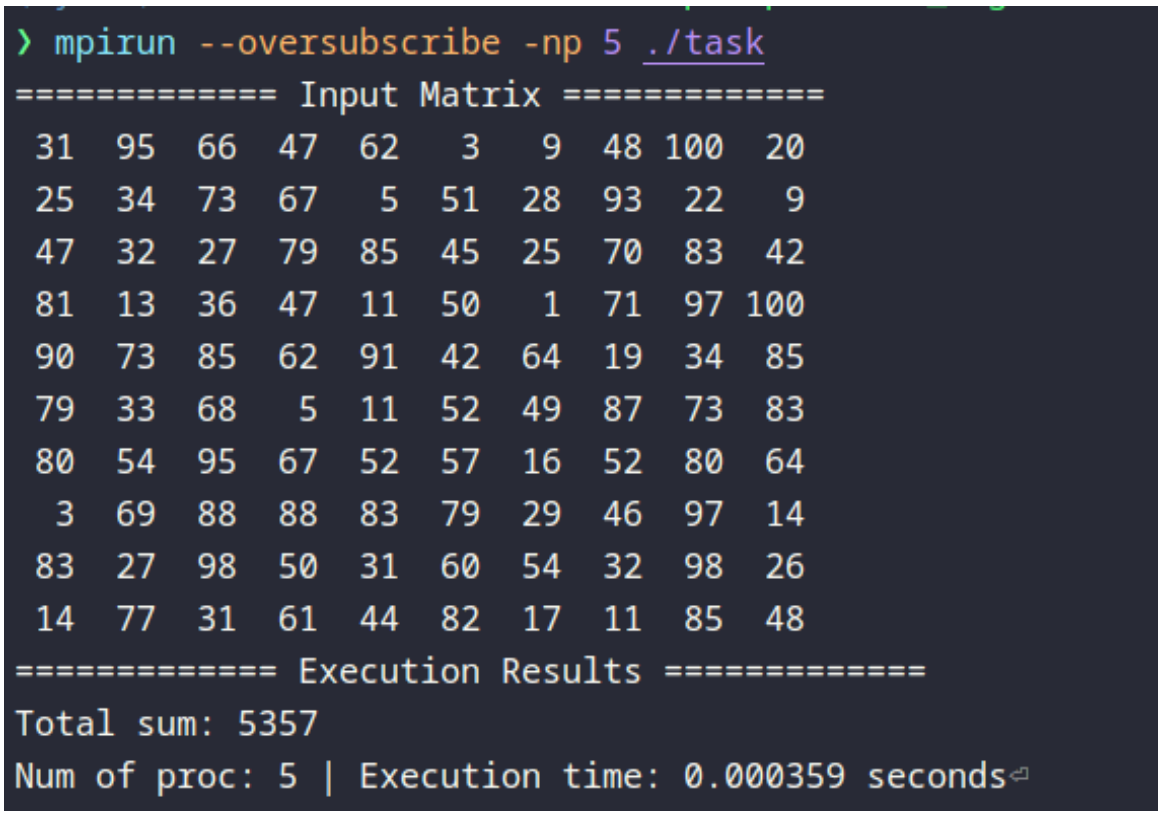

Рис.2 — пример работы программы

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.

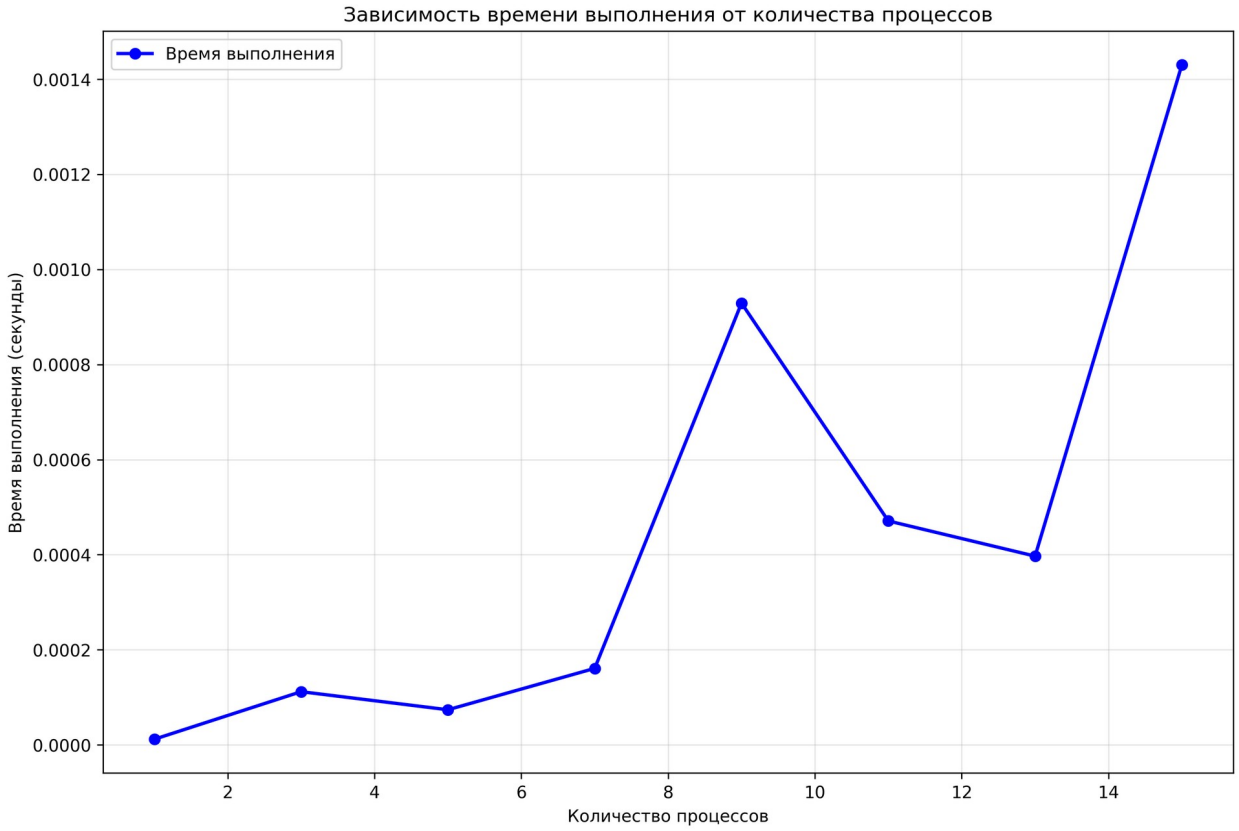

Рис.3 – график зависимости времени выполнения от числа процессов

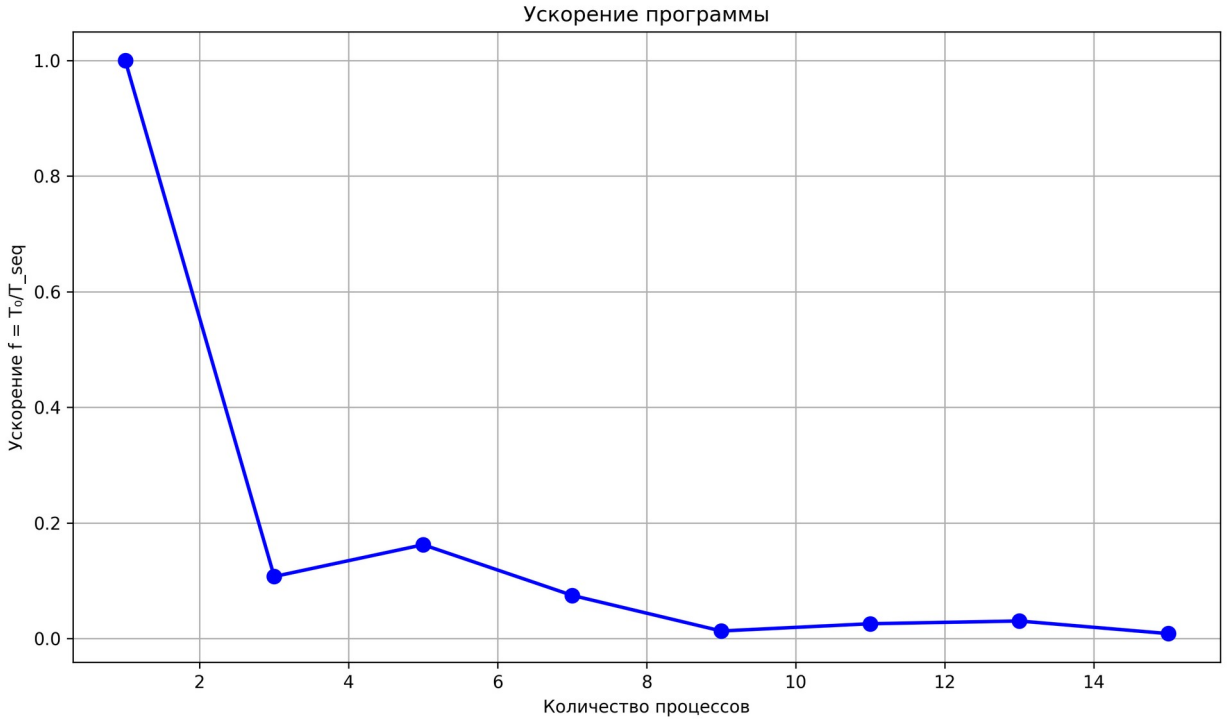

Рис.4 - график ускорения

5. Проведём запуск при большой входной матрице (4000 x 4000) с 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

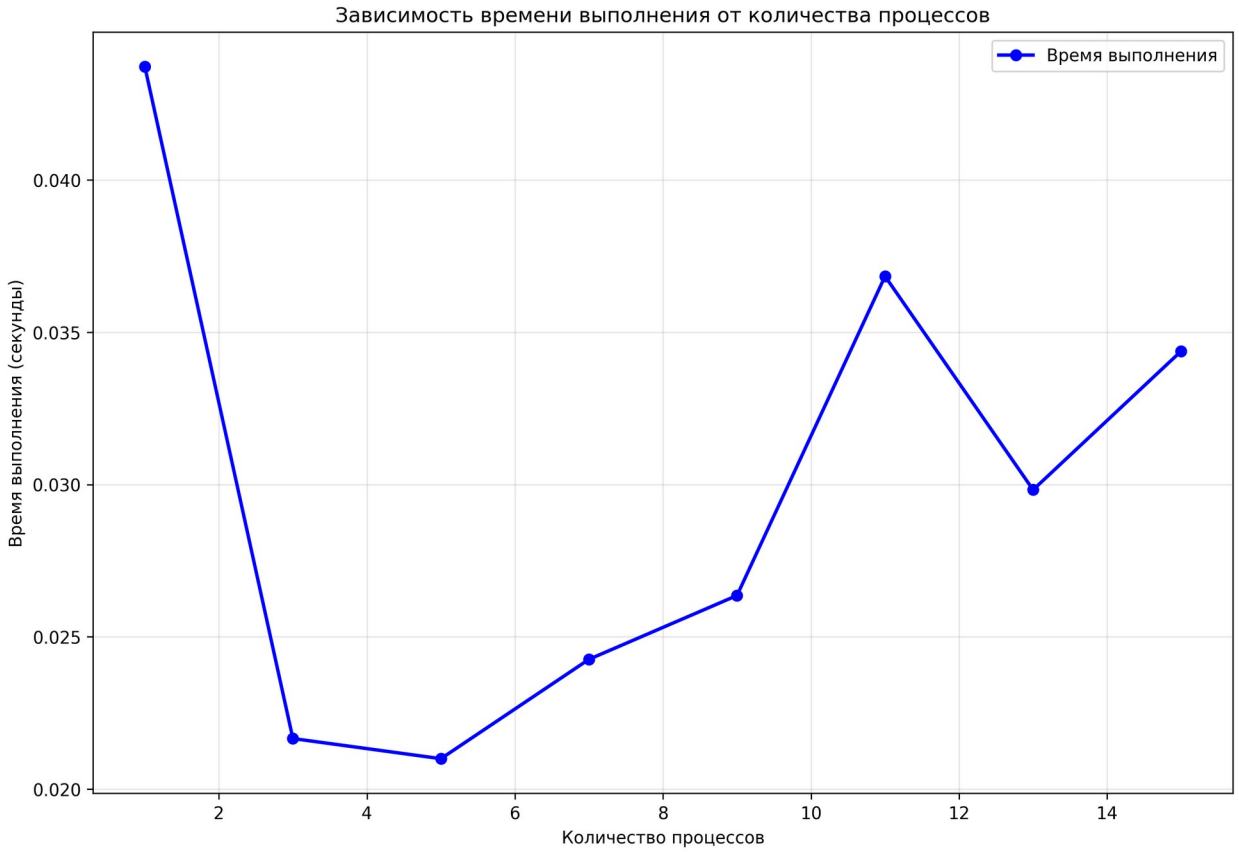

Рис.5 – график зависимости времени выполнения от числа процессов

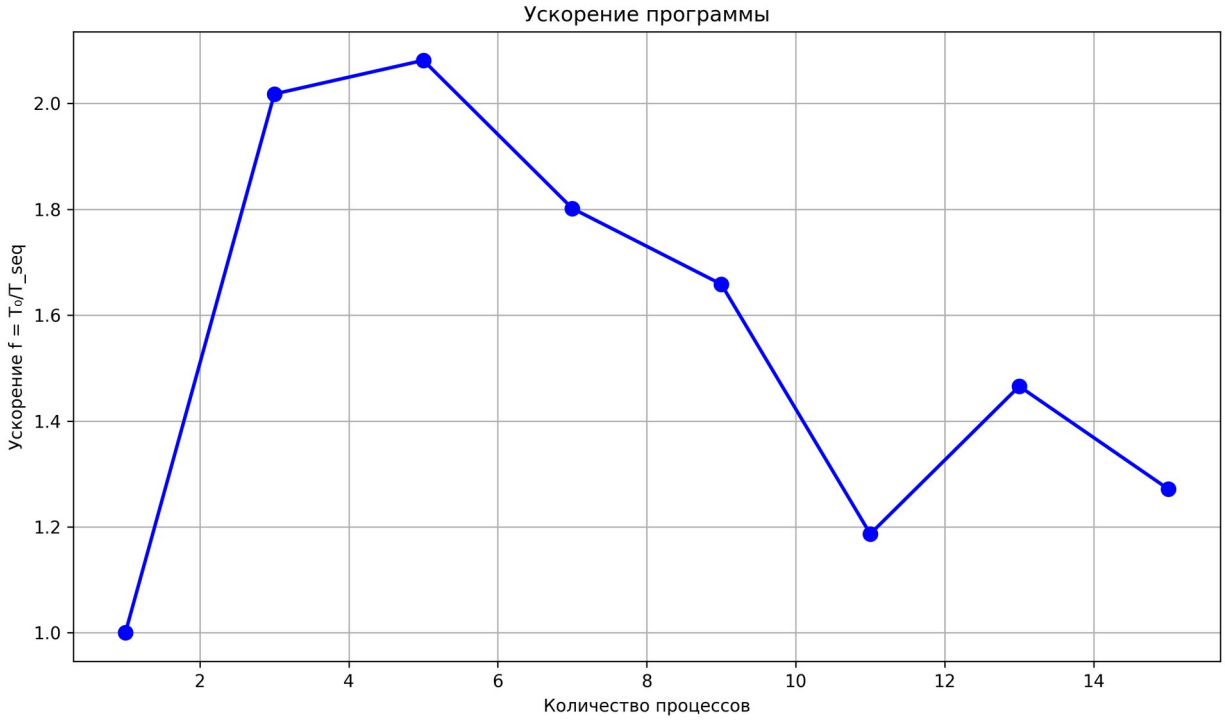

Рис.6 - график ускорения

Анализ результатов.

При запуске на малом массиве наблюдается замедление программы от числа запущенных процессов. На большом массиве программа продемонстрировала скачок ускорения при запуске на 3-ех процессах, после чего с увеличением числа процессов программа начала замедляться, но всё ещё показывала результат лучше чем при запуске на одном процессе.

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.c

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

#define ROWS 10
#define COLS 10

void init_matrix(int *matrix, int rows, int cols)
{
    for (int i = 0; i < rows * cols; ++i) {
        matrix[i] = rand() % 100 + 1;
    }
}

void print_matrix(int *matrix, int rows, int cols)
{
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            printf("%3d ", matrix[i * cols + j]);
        }
        printf("\n");
    }
}

int main(int argc, char *argv[])
{
    int rank, size;
    double start_time, end_time;

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

    int total_rows = ROWS, total_cols = COLS;
    int total_elements = total_rows * total_cols;

    int rows_per_proc = total_rows / size;
    int rem = total_rows % size;
```

```
int local_rows;
if (rank < rem) {
    local_rows = rows_per_proc + 1;
} else {
    local_rows = rows_per_proc;
}

int local_elements = local_rows * total_cols;

int *local_matrix = (int*)malloc(local_elements * sizeof(int));
int *full_matrix = NULL;

if (rank == 0) {
    full_matrix = (int*)malloc(total_elements * sizeof(int));
    srand(time(NULL));
    init_matrix(full_matrix, total_rows, total_cols);

    puts("============= Input Matrix =============");
    print_matrix(full_matrix, total_rows, total_cols);

    start_time = MPI_Wtime();

    int *sendcounts = (int*)malloc(size * sizeof(int));
    int *displs = (int*)malloc(size * sizeof(int));

    int offset = 0;
    for (int i = 0; i < size; ++i) {
        int rows_for_i = (i < rem) ? rows_per_proc + 1 : rows_per_proc;
        sendcounts[i] = rows_for_i * total_cols;
        displs[i] = offset * total_cols;
        offset += rows_for_i;
    }

    MPI_Scatterv(full_matrix, sendcounts, displs, MPI_INT,
                 local_matrix, local_elements, MPI_INT,
                 0, MPI_COMM_WORLD);

    free(sendcounts);
    free(displs);

} else {
    start_time = MPI_Wtime();
    MPI_Scatterv(NULL, NULL, NULL, MPI_INT,
                 local_matrix, local_elements, MPI_INT,
                 0, MPI_COMM_WORLD);
```

```
    }

    int local_sum = 0;
    for (int i = 0; i < local_elements; ++i) {
        local_sum += local_matrix[i];
    }

    int global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    end_time = MPI_Wtime();

    if (rank == 0) {
        puts("============== Execution Results ==============");
        printf("Total sum: %d\n", global_sum);
        printf("Num of proc: %d | Execution time: %.6f seconds",
                size, end_time - start_time);

        free(full_matrix);
    }

    free(local_matrix);
    MPI_Finalize();

    return 0;
}
```