МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Параллельные алгоритмы»

Тема: ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ ОБМЕНА ДАННЫМИ «ТОЧКА-ТОЧКА» В БИБЛИОТЕКЕ MPI.

Студент гр. 3388 _____ Дубровин Д.Н.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2025

Цель работы.

Изучить соединения типа Точка-точка в библиотеке MPI. Выполнить поставленную задачу, реализовав программу, протестировать программу на различном колличестве процессов, построить необходимые графики и сеть Петри.

Задание.

7. Обработка элементов массива. Процесс 0 генерирует массив и раздает его другим процессам для обработки (например, поиска нулевых элементов), после чего собирает результат.

Выполнение работы.

1. MPI-программа реализует распределённый подсчёт нулевых элементов в массиве. Процесс-координатор (ранг 0) генерирует массив случайных чисел размером BUFF_SIZE, затем равномерно распределяет данные между процессами с учётом остатка через вычисление elements_per_proc и remainder. Для коммуникации используются блокирующие операции MPI_Send и MPI_Recv с динамическим выделением памяти под локальные массивы. Каждый процесс независимо выполняет функцию count_zeroes для своей части данных, а результаты агрегируются на процессе-координаторе с использованием временных меток MPI_Wtime для измерения времени работы с последующим выводом результатов.
2. Построим схему Петри:

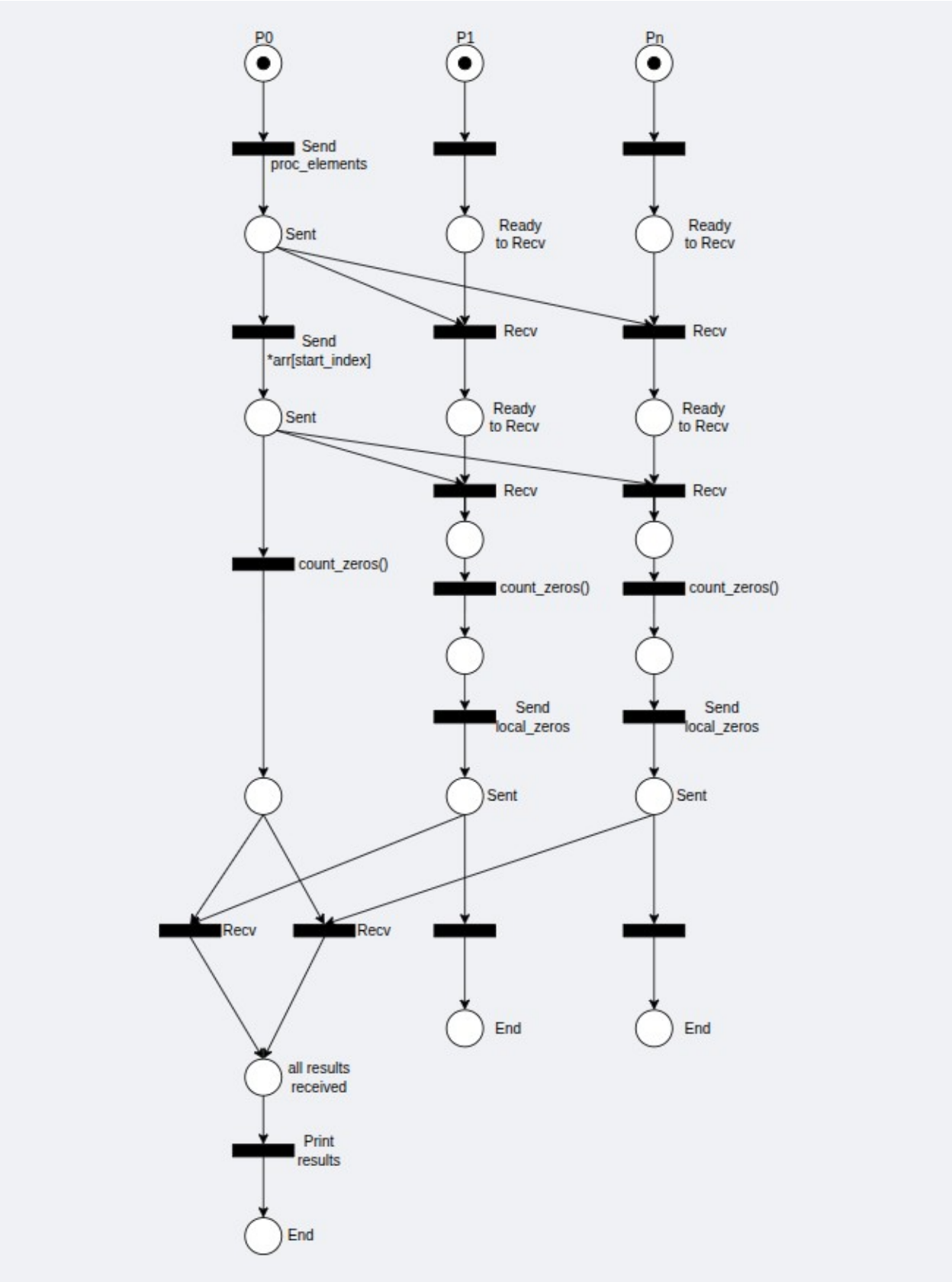

Рис.1 - сеть Петри

3. Проведём запуск при малом входном массиве (30 элементов)

Таблица 1- результаты выполнения программы при различном количестве процессов.

N	1	3	5	7	9	11	13	15
t, сек	0.000022	0.000062	0.000088	0.000151	0.000138	0.000187	0.000219	0.000258

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

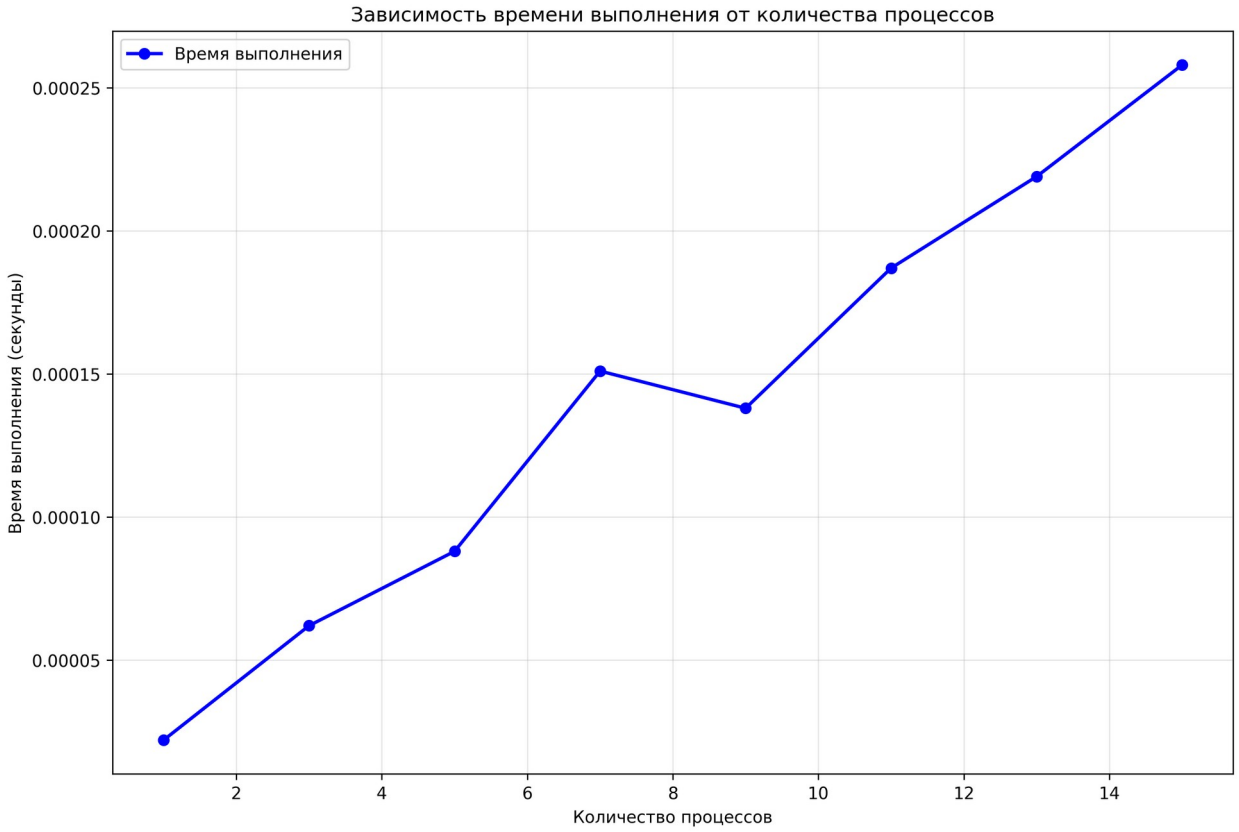

Рис.2 – график зависимости времени выполнения от числа процессов

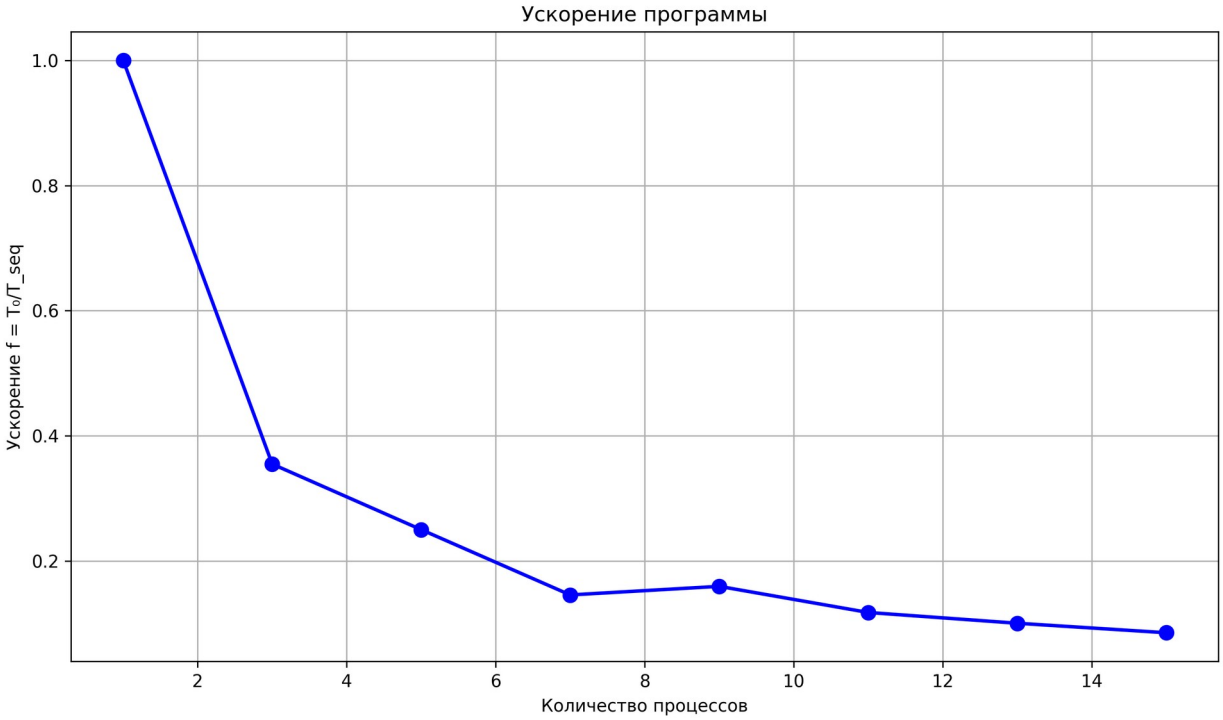

Рис.3 - график ускорения

5. Проведём запуск при большом входном массиве (1000000 элементов) в надежде увидеть улучшенные результаты ускорения.

Таблица 2- результаты выполнения программы при различном количестве процессов.

N	1	3	5	7	9	11	13	15
t, сек	0.070875	0.068095	0.067455	0.103476	0.074557	0.072628	0.094673	0.148950

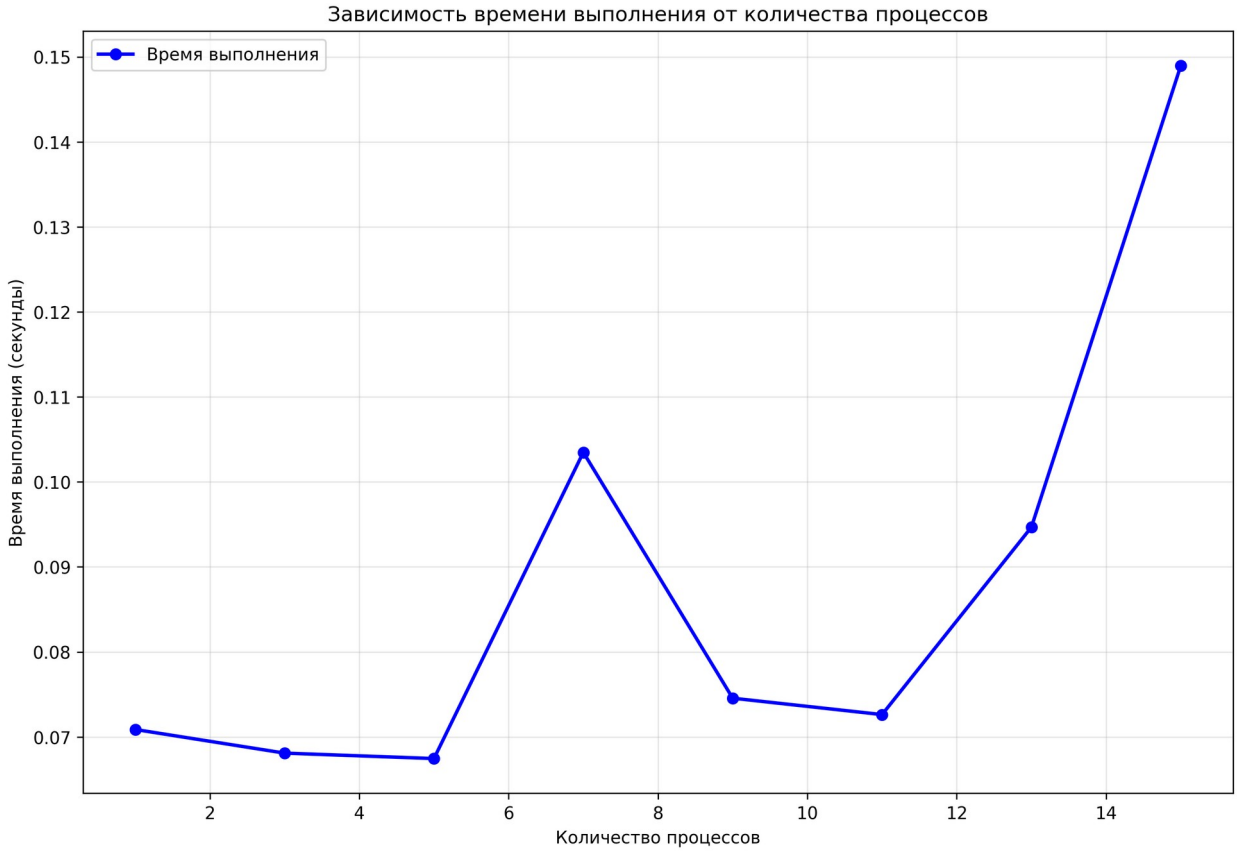

Рис.2 – график зависимости времени выполнения от числа процессов

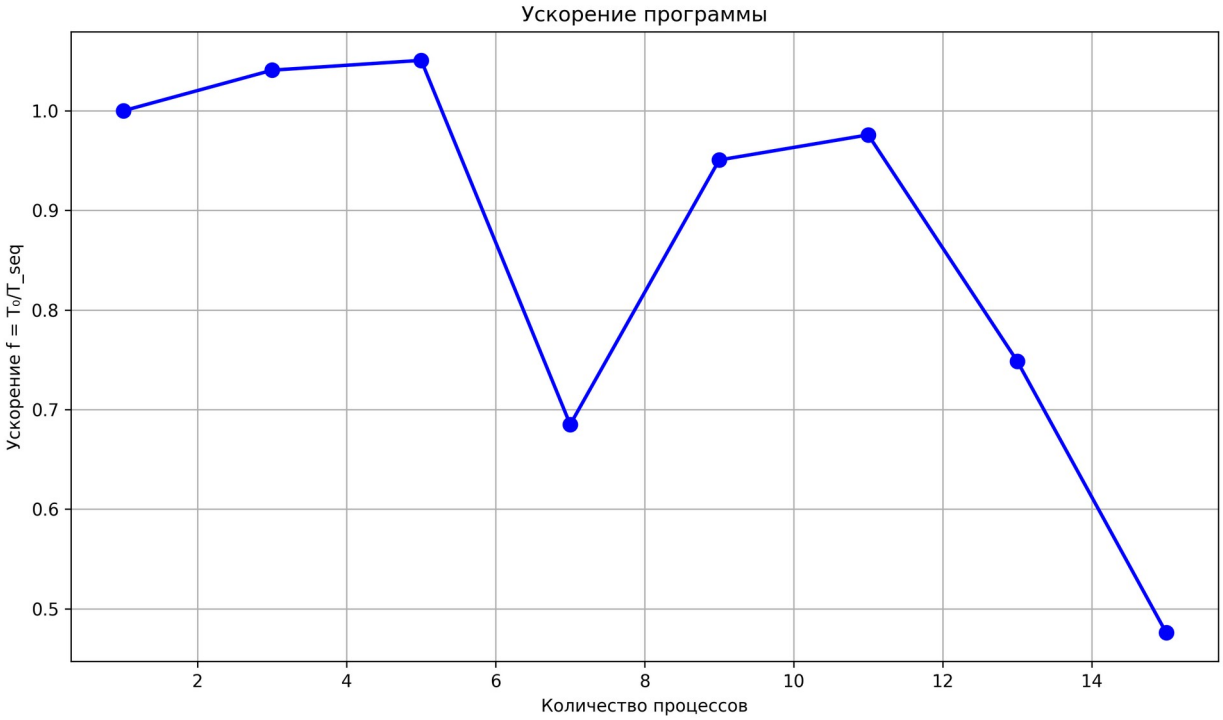

Рис.3 - график ускорения

Анализ результатов.

Можно заметить, что при больших и малых объёмах начальных данных программа не показала ускорения от числа запущенных процессов, что обусловлено простотой и низкими вычислительными требованиями первоначальной задачи. Распараллеливание же вызывает больше вычислительных затрат, чем даёт приемущества. Но при стоит отметить, что распараллеливание задачи при большом объёме данных не вызвало такого резкого спада ускорения как при малых входных данных.

Разработанный программный код см. в приложении А.

Выводы.

В ходе выполнения лабораторной работы была успешно реализована распределённая программа для подсчёта нулевых элементов в массиве с использованием библиотеки MPI и соединений типа "точка-точка". Исследована зависимость ускорения программы от объёма входных данных.

Приложение А

Исходный код программы

Название файла: task.c

```
#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;
    }
```