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.

Студентка гр. 2384 _____ Соц Е.А.

Преподаватель _____ Татаринов Ю.С.

Санкт-Петербург

2024

Цель работы

Изучить и применить основные функции библиотеки MPI для распределенных вычислений на примере задачи поиска нулевых элементов в массиве. Разработать программу, которая распределяет элементы массива между процессами для параллельной обработки, обеспечивая корректное распределение элементов, даже если размер массива не делится на количество процессов без остатка.

Задание

Вариант 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. Разработанная программа lab2.c:

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

void generate_array(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = rand() % 3; // Генерация случайных чисел
    }
}

int main(int argc, char **argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int array_size = 10; // Размер массива по умолчанию
    if (argc > 1) {
        array_size = atoi(argv[1]); // Считывание размера массива
    }

       int *array = (int *)malloc(array_size * sizeof(int)); //
Выделение памяти для массива во всех процессах
    int local_count = 0;
    int *results = NULL;

    if (rank == 0) {
        // Процесс 0 генерирует массив
        generate_array(array, array_size);
        // Вывод сгенерированного массива
        printf("Generated array: ");
        for (int i = 0; i < array_size; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");

        // Рассылка частей массива другим процессам
        int local_size = array_size / size;
        int remainder = array_size % size;
        for (int i = 1; i < size; i++) {
             int start_index = i * local_size + (i < remainder ? i
: remainder);
                 int end_index = start_index + local_size + (i <
remainder ? 1 : 0);
             MPI_Send(array + start_index, end_index - start_index,
MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        // Обработка своей части массива
        int start_index = 0;
```

```
        int end_index = local_size + (0 < remainder ? 1 : 0);
        printf("Process 0: ");
        for (int i = start_index; i < end_index; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");

        for (int i = start_index; i < end_index; i++) {
            if (array[i] == 0) {
                local_count++;
            }
        }
    } else {
        // Другие процессы получают свои части массива
        int local_size = array_size / size;
        int remainder = array_size % size;
         int start_index = rank * local_size + (rank < remainder ?
rank : remainder);
             int end_index = start_index + local_size + (rank <
remainder ? 1 : 0);
           MPI_Recv(array + start_index, end_index - start_index,
MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Вывод элементов, которые получил текущий процесс
        printf("Process %d: ", rank);
        for (int i = start_index; i < end_index; i++) {
            printf("%d ", array[i]);
        }
        printf("\n");

        // Поиск нулевых элементов в локальной части массива
        for (int i = start_index; i < end_index; i++) {
            if (array[i] == 0) {
                local_count++;
            }
        }
    }

    // Сбор результатов с помощью MPI_Send и MPI_Recv
    if (rank == 0) {
        results = (int *)malloc(size * sizeof(int));
        results[0] = local_count;
        for (int i = 1; i < size; i++) {
                        MPI_Recv(&results[i],  1,  MPI_INT,  i,  1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        // Вывод результатов
        int total_count = 0;
        for (int i = 0; i < size; i++) {
            total_count += results[i];
        }
                printf("Total  number  of  zero  elements:  %d\n",
total_count);

        free(results);
```

```
    } else {
        // Другие процессы отправляют результаты процессу 0
        MPI_Send(&local_count, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }

       free(array); // Освобождение памяти для массива во всех
процессах
    MPI_Finalize();
    return 0;
}
```

Программа начинается с инициализации MPI и получения идентификатора процесса (rank) и общего количества процессов (size). Процесс с rank 0 генерирует массив случайных чисел размером array_size. Этот размер может быть передан через аргументы командной строки. Процесс 0 делит массив на части и рассылает их другим процессам. Каждый процесс получает свою часть массива. Затем каждый процесс подсчитывает количество нулевых элементов в своей части массива. После этого процесс 0 собирает результаты подсчета нулей от всех процессов. Для этого используется последовательность вызовов MPI_Send и MPI_Recv. Процесс 0 суммирует результаты и выводит общее количество нулевых элементов в массиве. Программа освобождает выделенную память и завершает работу MPI.

Таким образом, коммуникация точка-точка в программе заключается в использовании функций MPI_Send и MPI_Recv для обмена данными между процессами, что позволяет распределить задачу поиска нулевых элементов массива между несколькими процессами и собрать результаты в одном месте.

Протестируем программу:

```
katya@katya:~/ПА/lb2$ mpirun -np 4 ./lb2 12
Generated array: 1 1 0 1 2 1 1 0 0 1 2 1
Process 0: 1 1 0
Process 2: 1 0 0
Process 1: 1 2 1
Process 3: 1 2 1
Total number of zero elements: 3
katya@katya:~/ПА/lb2$ mpirun -np 13 ./lb2 100
Generated array: 1 1 0 1 2 1 1 0 0 1 2 1 2 1 2 1 0 0 1 1 2 2 0 0 2
2 2 1 1 1 2 0 0 0 2 0 1 1 1 1 0 0 0 2 2 1 2 2 2 0 2 1 1 2 2 0 2 2
```

```
1 1 0 0 2 0 2 2 1 0 1 2 0 0 0 0 2 0 2 2 0 2 1 0 0 2 2 0 0 2 2 1 0
0 2 0 1 1 1 0 0 2
Process 0: 1 1 0 1 2 1 1 0
Total number of zero elements: 35
Process 1: 0 1 2 1 2 1 2 1
Process 2: 0 0 1 1 2 2 0 0
Process 3: 2 2 2 1 1 1 2 0
Process 4: 0 0 2 0 1 1 1 1
Process 5: 0 0 0 2 2 1 2 2
Process 6: 2 0 2 1 1 2 2 0
Process 7: 2 2 1 1 0 0 2 0
Process 8: 2 2 1 0 1 2 0 0
Process 9: 0 0 2 0 2 2 0
Process 10: 2 1 0 0 2 2 0
Process 11: 0 2 2 1 0 0 2
Process 12: 0 1 1 1 0 0 2
katya@katya:~/ПА/lb2$ mpirun -np 2 ./lb2 5
Generated array: 1 1 0 1 2
Process 0: 1 1 0
Total number of zero elements: 1
Process 1: 1 2
katya@katya:~/ПА/lb2$ mpirun -np 4 ./lb2 10
Generated array: 1 1 0 1 2 1 1 0 0 1
Process 0: 1 1 0
Process 1: 1 2 1
Process 2: 1 0
Process 3: 0 1
Total number of zero elements: 3
```

Исходя из листинга, видно, что распределение элементов массива происходят в порядке нумерования процессов. В целом, это довольно предсказуемое поведение, так как элементы передаются последовательно по процессам, и не может быть ситуации, когда процесс n получит элемент раньше процесса с меньшим рангом.

2. Измененная программа, предназначенная для измерения времени with_time.c:

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

void generate_array(int *array, int size) {
    for (int i = 0; i < size; i++) {
        array[i] = rand() % 3; // Генерация случайных чисел
    }
}

int main(int argc, char **argv) {
    int rank, size;
```

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

    int array_size = 10; // Размер массива по умолчанию
    if (argc > 1) {
        array_size = atoi(argv[1]); // Считывание размера массива
    }

       int *array = (int *)malloc(array_size * sizeof(int)); //
Выделение памяти для массива во всех процессах
    int local_count = 0;
    int *results = NULL;

    if (rank == 0) {
        // Процесс 0 генерирует массив
        generate_array(array, array_size);

    }

    // Запуск таймера перед рассылкой массива
    double start_time = MPI_Wtime();

    if (rank == 0) {
        // Рассылка частей массива другим процессам
        int local_size = array_size / size;
        int remainder = array_size % size;
        for (int i = 1; i < size; i++) {
             int start_index = i * local_size + (i < remainder ? i
: remainder);
                 int end_index = start_index + local_size + (i <
remainder ? 1 : 0);
             MPI_Send(array + start_index, end_index - start_index,
MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        // Обработка своей части массива
        int start_index = 0;
        int end_index = local_size + (0 < remainder ? 1 : 0);
        for (int i = start_index; i < end_index; i++) {
            if (array[i] == 0) {
                local_count++;
            }
        }
    } else {
        // Другие процессы получают свои части массива
        int local_size = array_size / size;
        int remainder = array_size % size;
         int start_index = rank * local_size + (rank < remainder ?
rank : remainder);
             int end_index = start_index + local_size + (rank <
remainder ? 1 : 0);
           MPI_Recv(array + start_index, end_index - start_index,
MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        // Поиск нулевых элементов в локальной части массива
```

```
        for (int i = start_index; i < end_index; i++) {
            if (array[i] == 0) {
                local_count++;
            }
        }
    }

    // Сбор результатов с помощью MPI_Send и MPI_Recv
    if (rank == 0) {
        results = (int *)malloc(size * sizeof(int));
        results[0] = local_count;
        for (int i = 1; i < size; i++) {
                        MPI_Recv(&results[i], 1, MPI_INT, i, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        // Остановка таймера после сбора результатов
        double end_time = MPI_Wtime();

        // Вывод результатов
        int total_count = 0;
        for (int i = 0; i < size; i++) {
            total_count += results[i];
        }
                printf("Total number of zero elements: %d\n",
total_count);

        // Вывод времени работы программы
        printf("Time: %f seconds\n", end_time - start_time);

        free(results);
    } else {
        // Другие процессы отправляют результаты процессу 0
        MPI_Send(&local_count, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }

       free(array); // Освобождение памяти для массива во всех
процессах
    MPI_Finalize();
    return 0;
}
```

Время считывается от момента начала рассылки массива всем процессам до момента, когда сборы закончились.

Скрипт, запускающий программу, меняя в ней длину массива time.sh:

```
#!/bin/bash

mpicc -o time with_time.c

for num_processes in 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16; do
    echo "$num_processes processes"

    for data_size in 1 10 100 1000 10000; do
```

```
      echo "$data_size len"

         mpirun -np $num_processes ./time $data_size


    done

    echo ""
done
```

Листинг запущенной программы:

```
katya@katya:~/ПА/lb2$ ./time.sh
2 processes
1 len
Total number of zero elements: 0
Time: 0.000032 seconds
10 len
Total number of zero elements: 3
Time: 0.000035 seconds
100 len
Total number of zero elements: 35
Time: 0.000023 seconds
1000 len
Total number of zero elements: 354
Time: 0.000025 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000074 seconds

3 processes
1 len
Total number of zero elements: 0
Time: 0.000159 seconds
10 len
Total number of zero elements: 3
Time: 0.000114 seconds
100 len
Total number of zero elements: 35
Time: 0.000100 seconds
1000 len
Total number of zero elements: 354
Time: 0.000093 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000078 seconds

4 processes
1 len
Total number of zero elements: 0
Time: 0.000081 seconds
10 len
Total number of zero elements: 3
Time: 0.000034 seconds
100 len
Total number of zero elements: 35
```

```
Time: 0.000100 seconds
1000 len
Total number of zero elements: 354
Time: 0.000037 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000070 seconds

5 processes
1 len
Total number of zero elements: 0
Time: 0.000088 seconds
10 len
Total number of zero elements: 3
Time: 0.000183 seconds
100 len
Total number of zero elements: 35
Time: 0.000139 seconds
1000 len
Total number of zero elements: 354
Time: 0.000129 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000068 seconds

6 processes
1 len
Total number of zero elements: 0
Time: 0.000094 seconds
10 len
Total number of zero elements: 3
Time: 0.000125 seconds
100 len
Total number of zero elements: 35
Time: 0.000100 seconds
1000 len
Total number of zero elements: 354
Time: 0.000107 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000071 seconds

7 processes
1 len
Total number of zero elements: 0
Time: 0.000140 seconds
10 len
Total number of zero elements: 3
Time: 0.000172 seconds
100 len
Total number of zero elements: 35
Time: 0.000163 seconds
1000 len
Total number of zero elements: 354
Time: 0.000130 seconds
10000 len
```

```
Total number of zero elements: 3324
Time: 0.000101 seconds

8 processes
1 len
Total number of zero elements: 0
Time: 0.000108 seconds
10 len
Total number of zero elements: 3
Time: 0.000114 seconds
100 len
Total number of zero elements: 35
Time: 0.000062 seconds
1000 len
Total number of zero elements: 354
Time: 0.000054 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000188 seconds

9 processes
1 len
Total number of zero elements: 0
Time: 0.000137 seconds
10 len
Total number of zero elements: 3
Time: 0.000154 seconds
100 len
Total number of zero elements: 35
Time: 0.000142 seconds
1000 len
Total number of zero elements: 354
Time: 0.000225 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000116 seconds

10 processes
1 len
Total number of zero elements: 0
Time: 0.000147 seconds
10 len
Total number of zero elements: 3
Time: 0.000139 seconds
100 len
Total number of zero elements: 35
Time: 0.000119 seconds
1000 len
Total number of zero elements: 354
Time: 0.000171 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000164 seconds

11 processes
1 len
```

```
Total number of zero elements: 0
Time: 0.000175 seconds
10 len
Total number of zero elements: 3
Time: 0.000492 seconds
100 len
Total number of zero elements: 35
Time: 0.000159 seconds
1000 len
Total number of zero elements: 354
Time: 0.000204 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000178 seconds

12 processes
1 len
Total number of zero elements: 0
Time: 0.000236 seconds
10 len
Total number of zero elements: 3
Time: 0.000185 seconds
100 len
Total number of zero elements: 35
Time: 0.000174 seconds
1000 len
Total number of zero elements: 354
Time: 0.000170 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000196 seconds

13 processes
1 len
Total number of zero elements: 0
Time: 0.000482 seconds
10 len
Total number of zero elements: 3
Time: 0.001008 seconds
100 len
Total number of zero elements: 35
Time: 0.000573 seconds
1000 len
Total number of zero elements: 354
Time: 0.000851 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000792 seconds

14 processes
1 len
Total number of zero elements: 0
Time: 0.002932 seconds
10 len
Total number of zero elements: 3
Time: 0.000461 seconds
```

```
100 len
Total number of zero elements: 35
Time: 0.000674 seconds
1000 len
Total number of zero elements: 354
Time: 0.000714 seconds
10000 len
Total number of zero elements: 3324
Time: 0.001726 seconds

15 processes
1 len
Total number of zero elements: 0
Time: 0.000941 seconds
10 len
Total number of zero elements: 3
Time: 0.000749 seconds
100 len
Total number of zero elements: 35
Time: 0.001170 seconds
1000 len
Total number of zero elements: 354
Time: 0.003904 seconds
10000 len
Total number of zero elements: 3324
Time: 0.007206 seconds

16 processes
1 len
Total number of zero elements: 0
Time: 0.005520 seconds
10 len
Total number of zero elements: 3
Time: 0.000665 seconds
100 len
Total number of zero elements: 35
Time: 0.001267 seconds
1000 len
Total number of zero elements: 354
Time: 0.004591 seconds
10000 len
Total number of zero elements: 3324
Time: 0.000642 seconds
```

По результатам измерений можно построить график времени выполнения программы, зависящий от объема массива и количества процессов, на которых будет выполняться локальный поиск:

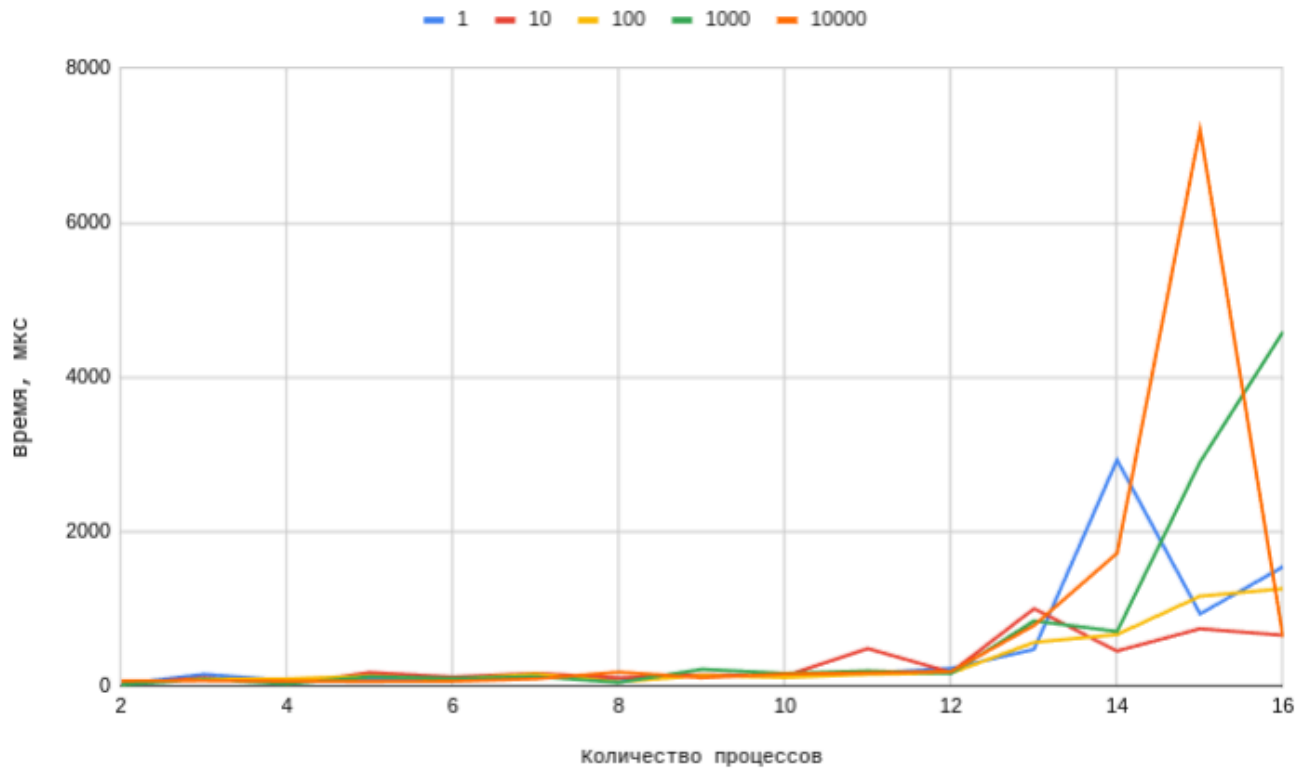


Рисунок 1 – Изменение времени от количества процессов для разных входных данных

На полученном графике четко видно: чем больше длина массива, тем дольше будет выполняться программа (ломаные, соответствующие бОльшим массивам, лежат выше ломаных, соответствующих меньшим). В целом, это вполне логично, так как чем больше длина массива, тем больше данных нужно передать другим процессам.

Также видно, что чем больше количество процессов, тем больше времени идет на выполнение программы. Объясняется это следующим: чем больше процессов задействовано, тем больше времени нужно для “взаимодействия” им между собой.

Можно заметить аномалию: функция графика возрастает, а потом резко падает (15 процессов для массива 10000) или время работы программы на 14 процессах для массива из одного элемента больше, чем для массивов бОльших длин. Все это можно объяснить значительными накладными расходами: затраты на координацию между процессами и

передачу данных начинают доминировать над преимуществами распараллеливания. Также использование скрипта увеличивает время исполнения программы.

Теперь можно проанализировать замедление/ускорение работы программы:

Для того, чтобы провести анализ ускорения или замедления работы программы, построим график, используя формулу ускорения: Sp(n) = T1(n)/Tp(n) , где T1(n) – среднее время выполнения алгоритма на одном процессе, Tp(n) - время выполнения алгоритма на n процессах.

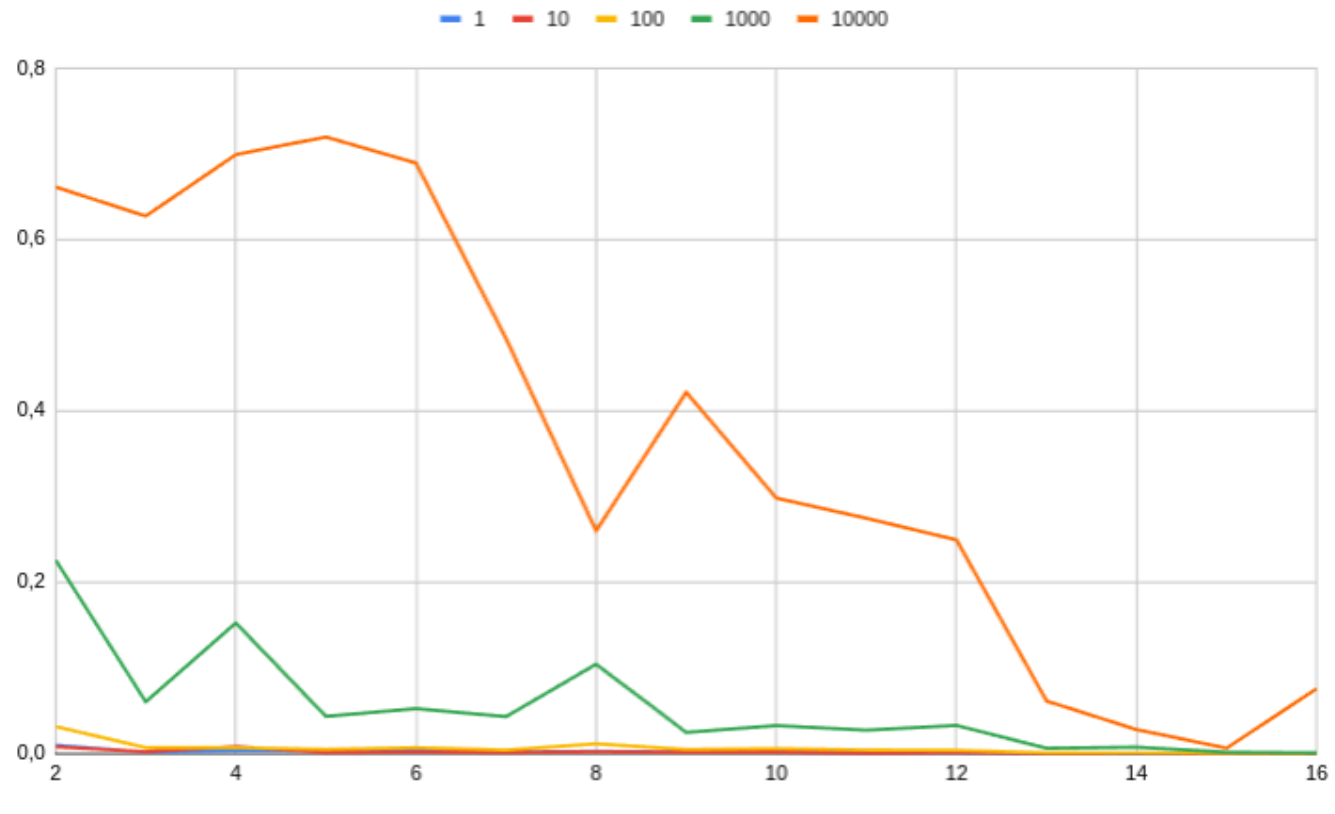
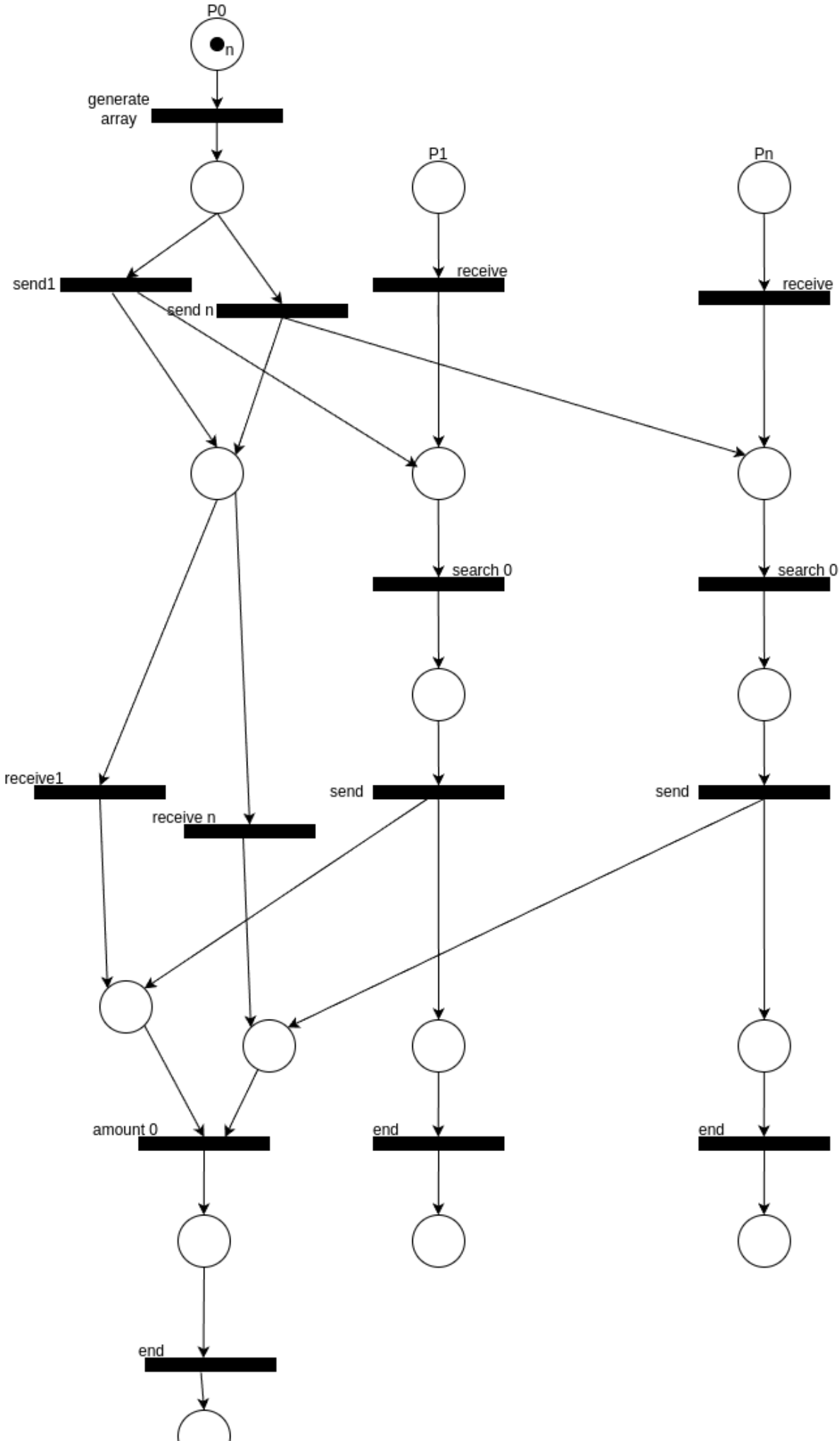


Рисунок 2 – Ускорение/замедление программы при данных разного размера для разного количества процессов

Из графика видно, что при увеличении количества процессов программа замедляется. Это происходит потому, что много времени уходит на их “взаимодействие”. Глядя на график, можно сказать, что последовательная программа выполняется быстрее. Точки перелома можно обосновать, опять же, накладными расходами.

Сети Петри:

Полученные сети Петри отражают всю работу программы: генерацию массива, отправка частей массива по процессам, локальный поиск нуля, отправкаа результатов в нулевой процесс, сбор этих результатов нулевым процессом и завершение. Также по сетям явно видно принцип “Точка-точка”.


P0
n
generate
array
P1
Pn
send1
send n
receive
receive
search 0
search 0
receive1
receive n
send
send
amount 0
end
end
end

Вывод

В ходе выполнения лабораторной работы была написана программа, которая распараллеливает задачу нахождения нулей в массиве между процессами: каждый процесс ищет ноль локально, на своей части массива. Программа была реализована с помощью функций обмена данными “точка-точка” в библиотеке MPI: MPI_Send и MPI_Recv. В ходе работы был построен график зависимости времени от количества процессов и входных данных, который показывает, что чем больше количество процессов/размер входных данных, тем больше времени нужно на это. Также был построен график замедления, отражающий, что время выполнения параллельной программы больше по сравнению с последовательной программой.