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

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Параллельные вычисления»
Тема: «Передача данных по процессам»

Студент гр. 1307 _____ Голубев М.А.

Преподаватель _____ Манжиков Л.П.

Санкт-Петербург
2025

Цель работы.

Освоить функции передачи данных между процессами.

Задание 1.

17) В прямоугольной матрице, состоящей из нулей и единиц, инвертировать четные столбцы;

Задание 2.

В полученной матрице (по результатам выполнения задания 1) найти:

17) Количество разных рядом стоящих элементов по столбцам;

Для решения задачи был написан код, в котором были определены параметры матрицы.

Вторая программа объединяет в себе сразу 2 задания.

Текст программы lab3_2.c.

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

#define DEBUG 0 // 1 для включения отладочной печати

void generate_random_matrix(int *matrix, int rows, int cols)
{
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   srand(time(NULL) + rank);
   for (int i = 0; i < rows * cols; i++)
   {
      matrix[i] = rand() % 2;
   }
}

void invert_even_columns(int *matrix, int rows, int cols)
{
   for (int j = 1; j < cols; j += 2)
      for (int i = 0; i < rows; i++)
      {
         matrix[i * cols + j] = 1 - matrix[i * cols + j];
      }
}
}

int count_different_neighbors_in_column(int *matrix, int rows, int cols, int col_index)
{
   int count = 0;
```

```
    for (int i = 0; i < rows - 1; i++)
    {
        if (matrix[i * cols + col_index] != matrix[(i + 1) * cols + col_index])
        {
            count++;
        }
    }
    return count;
}

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

int main(int argc, char **argv)
{
    int rank, size;
    int rows = 4;
    int cols = 6;
    int *matrix = NULL;
    int *inverted_matrix = NULL;
    int *column_counts = NULL;
    int global_total_count = 0;

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

    if (rank == 0)
    {
        matrix = (int *)malloc(rows * cols * sizeof(int));
        inverted_matrix = (int *)malloc(rows * cols * sizeof(int));
        column_counts = (int *)malloc(cols * sizeof(int));

        if (matrix == NULL || inverted_matrix == NULL || column_counts == NULL)
        {
            fprintf(stderr, "Ошибка выделения памяти в процессе 0.\n");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }

        generate_random_matrix(matrix, rows, cols);
        for (int i = 0; i < rows * cols; i++)
        {
            inverted_matrix[i] = matrix[i];
        }
        printf("Исходная матрица:\n");
        print_matrix(matrix, rows, cols);

        invert_even_columns(inverted_matrix, rows, cols);
        printf("\nИнвертированная матрица:\n");
        print_matrix(inverted_matrix, rows, cols);
        printf("\n");
```

```
    }

    MPI_Bcast(&rows, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&cols, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank != 0)
    {
        inverted_matrix = (int *)malloc(rows * cols * sizeof(int));
        if (inverted_matrix == NULL)
        {
            fprintf(stderr, "Ошибка выделения памяти в процессе %d.\n", rank);
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
    }

    MPI_Bcast(inverted_matrix, rows * cols, MPI_INT, 0, MPI_COMM_WORLD);

    int local_cols = cols / size;
    int remainder_cols = cols % size;
    int start_col = rank * local_cols;

    if (rank < remainder_cols)
    {
        local_cols++;
        start_col += rank;
    }
    else
    {
        start_col += remainder_cols;
    }

    int local_count = 0;
    for (int j = 0; j < local_cols; j++)
    {
        local_count += count_different_neighbors_in_column(inverted_matrix, rows, cols, start_col + j);
    }

    MPI_Reduce(&local_count, &global_total_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0)
    {
        printf("Общее количество различных соседей по столбцам: %d\n", global_total_count);
        free(matrix);
        free(inverted_matrix);
        free(column_counts);
    }
    else
    {
        free(inverted_matrix);
    }

    MPI_Finalize();
    return 0;
}
```

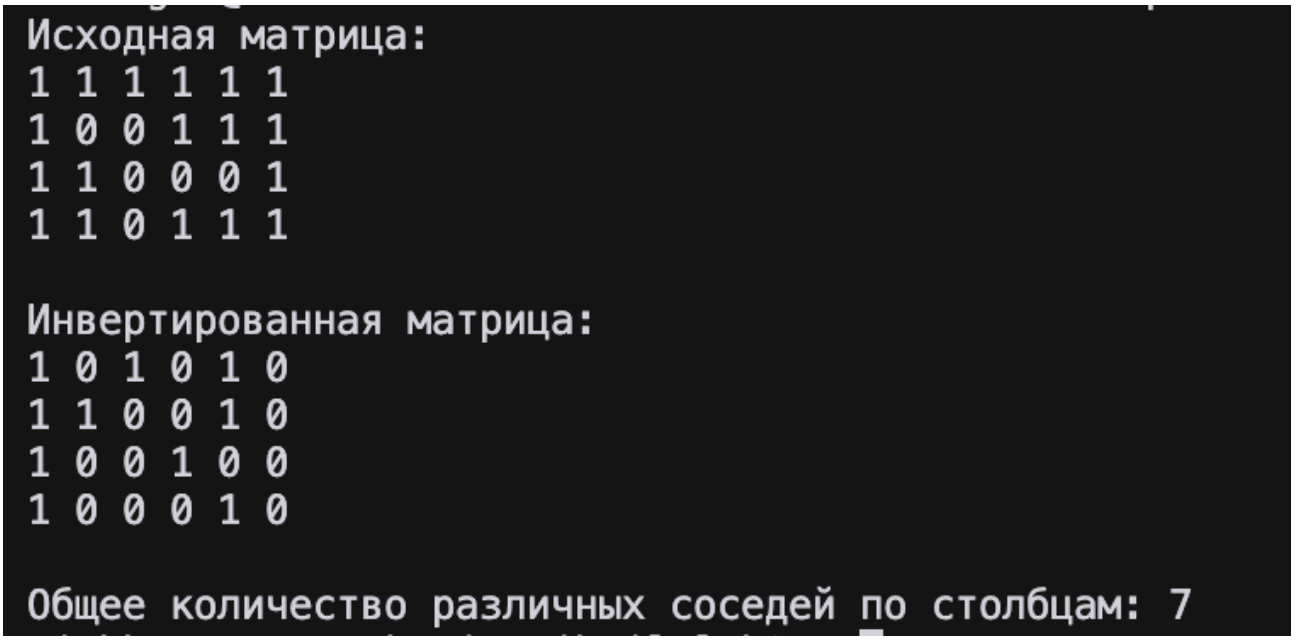

Рисунок 1. Запуск программы на 7-ми процессах.

Выводы.

В процессе выполнения лабораторной работы мы познакомились с основными принципами и возможностями MPI — программного интерфейса для параллельных вычислений. В ходе работы мы научились инициализировать, определять количество и упорядочивать процессы, передавать и получать данные, а также распределять вычисления и собирать результаты. В результате мы приобрели практические навыки работы с базовыми функциями MPI, которые станут основой для создания более сложных параллельных алгоритмов. Кроме того, мы получили ценный опыт в разработке, тестировании и отладке параллельных программ.