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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Параллельные вычисления»
Тема: «Коллективные функции»

Студент гр. 1307 _____ Голубев М.А.

Преподаватель _____ Манжиков Л.П.

Санкт-Петербург
2025

Цель работы.

Освоить функции коллективной обработки данных.

Задание 1 (по вариантам).

Решить задание 1 из лаб. работы 2 с применением коллективных функций.

Задание 2 (по вариантам).

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

Решить задание 1 или 2 из лаб. работы 3 с применением коллективных функций.

Текст программы lab4_1.cpp.

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

#define ROWS 6
#define COLS 4

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

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 matrix[ROWS][COLS];
    int local_rows = ROWS / (size - 1);
    int local_matrix[local_rows][COLS];

    if (rank == 0)
    {
        printf("Исходная матрица:\n");
        for (int i = 0; i < ROWS; i++)
        {
            for (int j = 0; j < COLS; j++)
            {
                matrix[i][j] = rand() % 2;
            }
        }
        print_matrix(matrix, ROWS, COLS);
    }

    MPI_Scatter(matrix, local_rows * COLS, MPI_INT, local_matrix, local_rows *
COLS, MPI_INT, 0, MPI_COMM_WORLD);

    for (int i = 0; i < local_rows; i++)
    {
        for (int j = 1; j < COLS; j += 2)
        {
            local_matrix[i][j] = 1 - local_matrix[i][j];
        }
    }

    MPI_Gather(local_matrix, local_rows * COLS, MPI_INT, matrix, local_rows * COLS,
MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0)
    {
        printf("Матрица после инверсии четных столбцов:\n");
        print_matrix(matrix, ROWS, COLS);
    }

    MPI_Finalize();
    return 0;
}
```

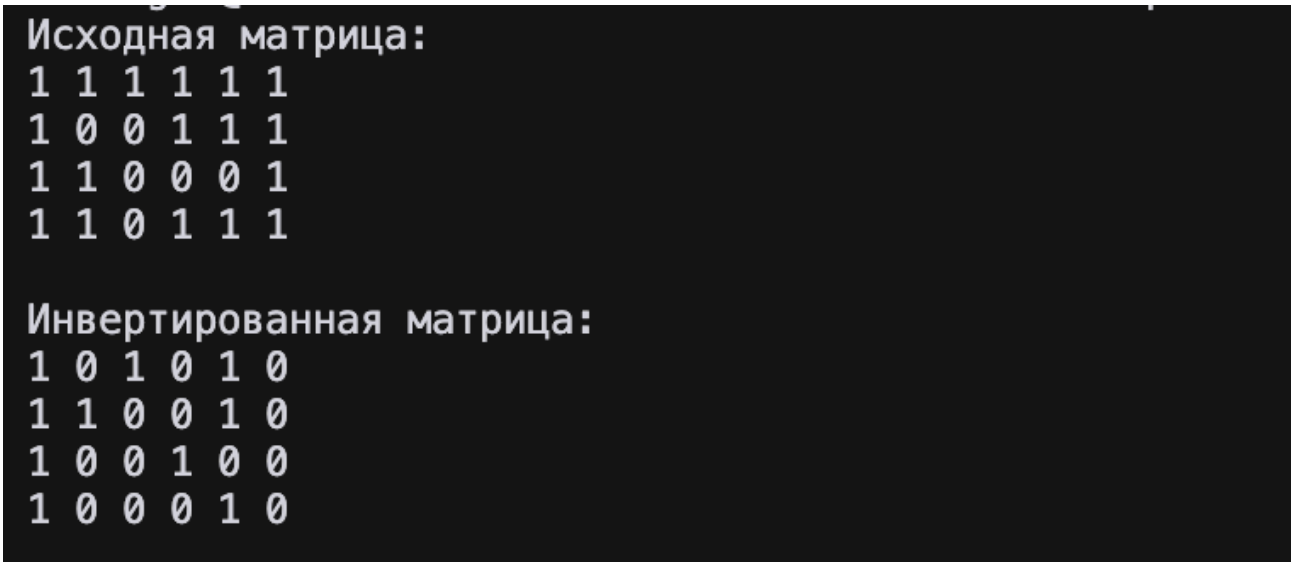

Рисунок 1. Запуск программы на 7-и процессах.

Текст программы lab4_2.cpp.

```
#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;
}
```

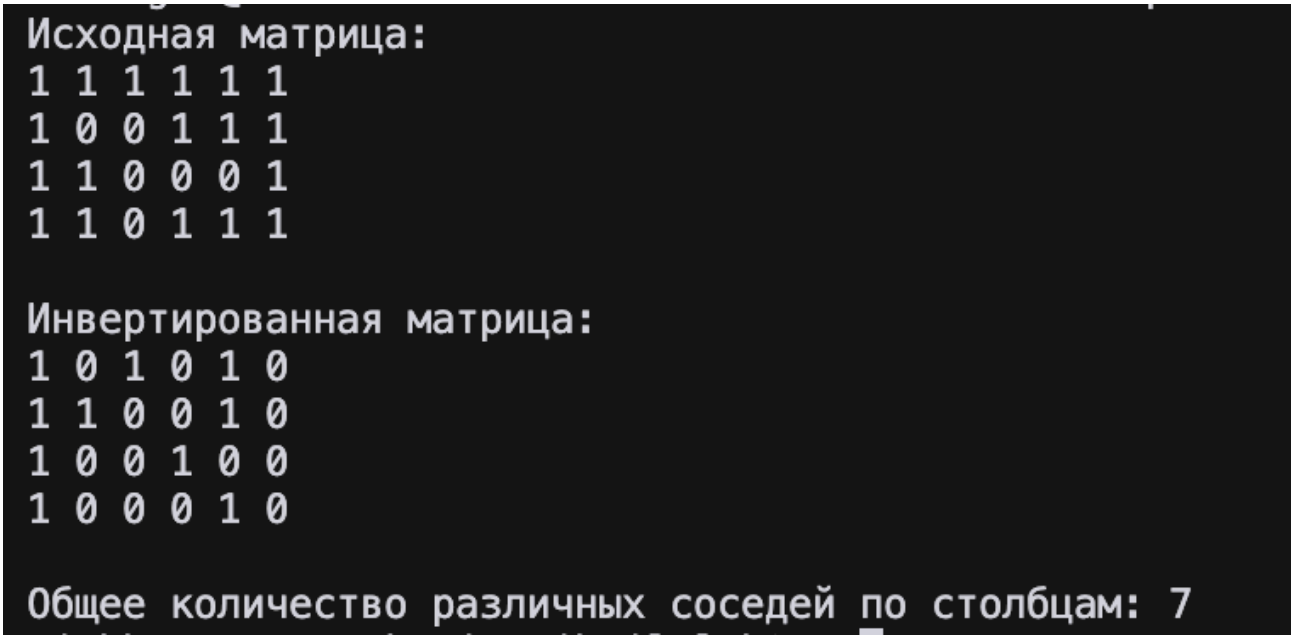

Рисунок 1. Запуск программы на 7-ми процессах.

Выводы.

В процессе выполнения лабораторной работы мы успешно освоили функции коллективной обработки данных в MPI. Мы разобрались в их принципах и приобрели практические навыки их применения для реализации параллельных алгоритмов. Мы научились использовать коллективные операции для обмена данными между процессами и эффективного распределения вычислительной нагрузки. Также мы узнали, что для корректного выполнения параллельной программы необходимо синхронизировать процессы при работе с коллективными функциями.