

**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.

Задание 2 из лаб. работы №2 - создать n процессов и найти минимальный положительный элемент массива.

Текст программы lab4\_1.cpp.

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

#define DEFAULT_ARR_SIZE 10

int main(int argc, char **argv)
{
    int rank, size;
    int local_min = INT_MAX;
    int global_min = INT_MAX;
    int array_size = 20;

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

    if (rank == 0)
    {
        if (argc > 1)
        {
            array_size = atoi(argv[1]);
        }
    }
}
```

```

        if (array_size <= 0)
        {
            printf("Размер массива не задан, используется размер по умолчанию
(20).\n");

            array_size = DEFAULT_ARR_SIZE;
        }
    }

    printf("Используемый размер массива: %d\n", array_size);
}

MPI_Bcast(&array_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

int *array = NULL;
if (rank == 0)
{
    array = (int *)malloc(array_size * sizeof(int));
    srand(time(NULL));
    printf("Массив: ");
    for (int i = 0; i < array_size; ++i)
    {
        array[i] = rand() % 100 - 50;
        printf("%d ", array[i]);
    }
    printf("\n");
}

int local_size = array_size / size;
int *local_array = (int *)malloc(local_size * sizeof(int));

MPI_Scatter(array, local_size, MPI_INT, local_array, local_size, MPI_INT, 0,
MPI_COMM_WORLD);

for (int i = 0; i < local_size; ++i)
{
    if (local_array[i] > 0 && local_array[i] < local_min)
    {
        local_min = local_array[i];
    }
}

MPI_Reduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

```

```

    if (rank == 0)
    {
        if (global_min != INT_MAX)
        {
            printf("Минимальный положительный элемент: %d\n", global_min);
        }
        else
        {
            printf("В массиве нет положительных элементов.\n");
        }
        free(array);
    }

    free(local_array);
    MPI_Finalize();
    return 0;
}

```

```

● mishhgun@MacBook-Air-Mihail lab2 % docker run mpich-2-2
Используемый размер массива: 20
Массив: -7 21 -36 -21 -20 -22 28 -49 -8 36 -39 47 46 41 0 -42 -49 17 36 -2
Минимальный положительный элемент: 21

```

Рисунок 1. Запуск программы на 8-ми процессах.

Задание 2 из лаб. работы №3 - в прямоугольной матрице, состоящей из 0 и 1 найти количество разных рядом стоящих элементов по столбцам.

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;
}

```

**Исходная матрица:**

```

1 1 1 1 1 1
1 0 0 1 1 1
1 1 0 0 0 1
1 1 0 1 1 1

```

**Инвертированная матрица:**

```

1 0 1 0 1 0
1 1 0 0 1 0
1 0 0 1 0 0
1 0 0 0 1 0

```

**Общее количество различных соседей по столбцам: 7**

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.