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 _____ Тростин М. Ю.

Преподаватель _____

Санкт-Петербург

2025

Цели и задачи

Цель: Освоить функции коллективной обработки данных.

Задание 1 (по вариантам).

Решить задание 2 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.

Выполнение работы

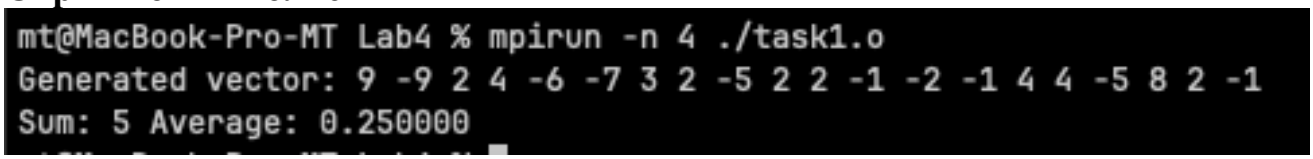
Задание 1 (по вариантам).

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

В данном задании «главный процесс» (с rank = 0) генерирует вектор в котором количество элементов зависит от количества запущенных процессов (на каждый процесс генерируется по 5 элементов). При помощи функции MPI_Scatter каждый процесс получает свои части массива. С помощью функции MPI_Reduce главный процесс получает массив суммы элементов. Осталось лишь, посчитав итоговую сумму и разделив её на количество сгенерированных элементов, получить среднее значение в массиве

Исходный код доступен в Приложении А

Скриншот выполнения



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.

По аналогии с Заданием 1, здесь главный процесс генерирует матрицу и, при помощи функции MPI_Scatter, распределяет строки матрицы по процессам, а также, при помощи MPI_Bcast, сообщает всем процессам последний элемент матрицы. Каждый процесс модифицирует свою строку, чтобы уравнять количество положительных и отрицательных тем же способом, что был описан в Лабораторной работе 3 и считает количество элементов, больше последнего элемента матрицы. Затем, используя MPI_Gather, главный процесс может собрать матрицу из модифицированных строк, а при помощи MPI_Reduce, посчитать итоговое количество элементов больших последнего элемента матрицы.

Исходный код доступен в Приложении Б

Скриншот выполнения

```
mt@MacBook-Pro-MT Lab4 % mpirun -n 4 ./task2.o
Generated matrix:
-5 3 1 7 2 6
8 3 -1 -6 -2 9
5 -8 8 1 -9 -8
9 -2 -1 -8 -5 -6
Reference: -6
#0 Count: 6 Modidfied row: -5 -3 -1 7 2 6
#1 Count: 5 Modidfied row: 8 3 -1 -6 -2 9
#2 Count: 3 Modidfied row: 5 -8 8 1 -9 -8
#3 Count: 4 Modidfied row: 9 2 1 -8 -5 -6
Global count: 18
Global matrix:
-5 -3 -1 7 2 6
8 3 -1 -6 -2 9
5 -8 8 1 -9 -8
9 2 1 -8 -5 -6
```

Выводы

В рамках выполнения данной работы были освоены функции коллективной обработки данных

ПРИЛОЖЕНИЕ А

Задание 1. Файл task1.cpp

```
#include <stdio.h>
#include <mpi.h>
#include <random>
#include <math.h>

#define vector_t int *
#define element_t int
#define MPI_ELEMENT_T MPI_INT

#define error_t unsigned char
#define ERR_VAL 1

#define SLICE_SIZE 5

int getSum(vector_t v) {
    int sum = 0;
    for (int i = 0; i < SLICE_SIZE; i++) sum += v[i];
    return sum;
}

vector_t emptyVector(size_t size) {
    vector_t vector = (vector_t)malloc(size * sizeof(element_t));
    for (int i = 0; i < size; i++) vector[i] = 0;

    return vector;
}

vector_t getRandomVector(size_t length, unsigned int seedOffset = 0) {
    vector_t vector = emptyVector(length);
    srand((unsigned int)time(NULL) + seedOffset);
    for (int i = 0; i < length; i++) vector[i] = rand() % 19 - 9;
    return vector;
}

void printVector(vector_t vector, int size) {
    for (int i = 0; i < size; i++) printf("%d ", vector[i]);
    printf("\n");
}

int main(int argc, char ** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2) {
        if (rank == 0) printf("Expected more than one process\n");
        MPI_Finalize();
```

```
        return ERR_VAL;
    }

    size_t vectorSize = SLICE_SIZE * size;
    vector_t vector = emptyVector(vectorSize);
    if (rank == 0) {
        vector = getRandomVector(vectorSize);
        printf("Generated vector: ");
        printVector(vector, vectorSize);
    }

    vector_t slice = emptyVector(SLICE_SIZE);
    MPI_Scatter(vector, SLICE_SIZE, MPI_ELEMENT_T, slice, SLICE_SIZE,
MPI_ELEMENT_T, 0, MPI_COMM_WORLD);

    vector_t sliceSum = emptyVector(SLICE_SIZE);
    MPI_Reduce(slice, sliceSum, SLICE_SIZE, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        int sum = 0;
        for (int i = 0; i < SLICE_SIZE; i++) sum += sliceSum[i];
        float average = (float)sum / vectorSize;
        printf("Sum: %d Average: %f\n", sum, average);
    }


    MPI_Finalize();
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Задание 2. Файл task2.cpp

```
#include <stdio.h>
#include <mpi.h>
#include <random>
#include <math.h>

#define matrix_t int **
#define row_t int *
#define element_t int
#define MPI_ELEMENT_T MPI_INT

#define error_t unsigned char
#define ERR_VAL 1

#define ROW_SIZE 6

matrix_t emptyMatrix(int rowCount, int rowSize) {
    matrix_t matrix = (matrix_t)malloc(rowCount * sizeof(row_t));
    for (int i = 0; i < rowCount; i++) {
        matrix[i] = (row_t)malloc(rowSize * sizeof(element_t));
        for (int j = 0; j < rowSize; j++) matrix[i][j] = 0;
    }

    return matrix;
}

row_t emptyRow(int rowSize) {
    row_t row = (row_t)malloc(rowSize * sizeof(element_t));
    for (int i = 0; i < rowSize; i++) row[i] = 0;

    return row;
}

matrix_t generateMatrix(int rowCount, int rowSize) {
    srand((unsigned int)time(NULL));
    matrix_t matrix = emptyMatrix(rowCount, rowSize);
    for (int i = 0; i < rowCount; i++) {
        for (int j = 0; j < rowSize; j++) {
            element_t el = rand() % 9 + 1;
            matrix[i][j] = rand() % 2 == 0 ? el : -el;
        }
    }

    return matrix;
}

row_t flat(matrix_t matrix, int rowCount, int rowSize) {
    row_t row = emptyRow(rowCount * rowSize);
    for (int i = 0; i < rowCount; i++) {
```

```
        for (int j = 0; j < rowSize; j++) {
            row[i * rowSize + j] = matrix[i][j];
        }
    }

    return row;
}

matrix_t deflat(row_t row, int rowCount, int rowSize) {
    matrix_t matrix = emptyMatrix(rowCount, rowSize);
    for (int i = 0; i < rowCount; i++) {
        for (int j = 0; j < rowSize; j++) {
            matrix[i][j] = row[i * rowSize + j];
        }
    }

    return matrix;
}

void printMatrix(matrix_t matrix, int rowCount, int rowSize) {
    for (int i = 0; i < rowCount; i++) {
        for (int j = 0; j < rowSize; j++) printf("%d ", matrix[i][j]);
        printf("\n");
    }
}

void printRow(row_t row, int rowSize) {
    for (int i = 0; i < rowSize; i++) printf("%d ", row[i]);
    printf("\n");
}

row_t balanceRow(row_t row, int rowSize) {
    row_t brow = emptyRow(rowSize);
    int beamScales = 0;
    for (int i = 0; i < rowSize; i++) {
        brow[i] = row[i];
        beamScales += row[i] > 0 ? 1 : row[i] == 0 ? 0 : -1;
    }

    for (int i = 0; i < rowSize; i++) {
        if (beamScales == 0) break;
        if (brow[i] == 0) continue;
        if ((brow[i] > 0) == (beamScales > 0)) {
            brow[i] = -brow[i];
            beamScales += beamScales > 0 ? -2 : 2;
        }
    }

    return brow;
}
```

```
int countToReference(row_t row, int rowSize, element_t reference) {
    int count = 0;
    for (int i = 0; i < rowSize; i++) {
        if (row[i] > reference) {
            count++;
        }
    }

    return count;
}

int main(int argc, char ** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2) {
        if (rank == 0) printf("Expected more than one process\n");
        MPI_Finalize();
        return ERR_VAL;
    }

    matrix_t matrix = emptyMatrix(size, ROW_SIZE);
    element_t reference = 0;

    if (rank == 0) {
        matrix = generateMatrix(size, ROW_SIZE);
        reference = matrix[size - 1][ROW_SIZE - 1];

        printf("Generated matrix:\n");
        printMatrix(matrix, size, ROW_SIZE);

        printf("Reference: %d\n", reference);
    }

    row_t absoluteRow = flat(matrix, size, ROW_SIZE);
    row_t dedicatedRow = emptyRow(ROW_SIZE);


    MPI_Scatter(absoluteRow, ROW_SIZE, MPI_ELEMENT_T, dedicatedRow,
ROW_SIZE, MPI_ELEMENT_T, 0, MPI_COMM_WORLD);
    MPI_Bcast(&reference, 1, MPI_ELEMENT_T, 0, MPI_COMM_WORLD);



    row_t balancedRow = balanceRow(dedicatedRow, ROW_SIZE);
    int count = countToReference(balancedRow, ROW_SIZE, reference);
```

```
    printf("#%d Count: %d Modidfied row: ", rank, count);
    printRow(balancedRow, ROW_SIZE);

    row_t modAbsoluteRow = emptyRow(size * ROW_SIZE);
    int globalCount = 0;
    MPI_Reduce(&count, &globalCount, 1,  MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);
    MPI_Gather(balancedRow, ROW_SIZE, MPI_ELEMENT_T, modAbsoluteRow,
ROW_SIZE, MPI_ELEMENT_T, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Global count: %d\n", globalCount);
        matrix_t globalMatrix = deflat(modAbsoluteRow, size, ROW_SIZE);

        printf("Global matrix:\n");
        printMatrix(globalMatrix, size, ROW_SIZE);
    }

    MPI_Finalize();
    return 0;
}
```