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 между процессами»

Студент гр. 1307 _____ Тростин М. Ю.

Преподаватель _____

Санкт-Петербург

2025

Цели и задачи
Освоить функции передачи данных между процессами

1) В прямоугольной матрице с произвольными ненулевыми числами уровнять число отрицательных и положительных элементов путем конвертации минимального числа элементов. Пример на множестве: Дано: -5,6,7,8,-4,4,2,8. Получаем: -5,-6,7,8,-4,4,2,-8
2) Среднее арифметическое элементов больших последнего элемента

Выполнение работы

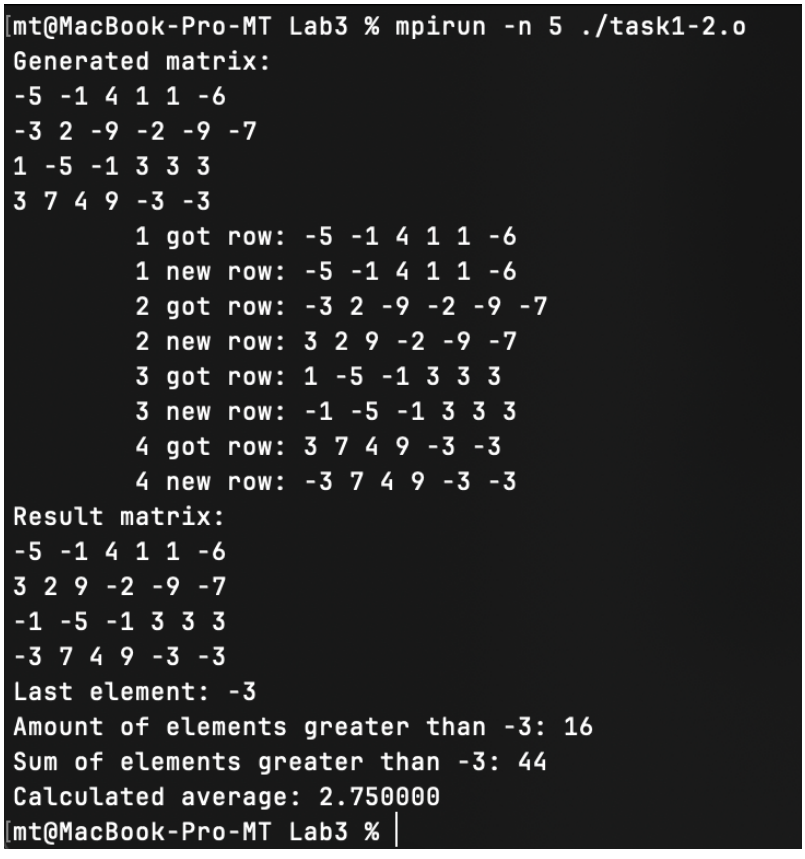
В данном задании «главный процесс» (с rank = 0) в начале генерирует случайную матрицу с ненулевыми числами. Количество строк зависит от количества запущенных процессов: на каждый другой процесс генерируется по одной строке. После генерации, главный процесс отправляет каждому другому процессу соответствующую строку.

Получив данные, процессы уравнивают количество отрицательных и положительных чисел в своей строке и отправляют получившуюся строку назад главному процессу. Таким образом, главный процесс восстанавливает новую матрицу и может получить последний элемент матрицы. Главный процесс вновь отправляет сообщения, теперь уже, с последним числом матрицы. Каждый процесс считает количество и сумму элементов больших того, что отправил главный процесс и отправляет эти значения главному процессу.

Собрав все данные, главный процесс может разделить общую сумму на общее количество, получив таким образом среднее арифметическое элементов больших последнего элемента.

Исходный код доступен в Приложении А

Скриншот выполнения



Выводы

В рамках выполнения данной работы были освоены функции передачи данных между процессами

ПРИЛОЖЕНИЕ А

Задание 1. Файл task1-2.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 error_t int
#define ERR_INVAL_ARG 1

#define ROW_SIZE 6

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

    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");
    }
}

error_t mainProcess(int size) {
    matrix_t matrix = generateMatrix(size - 1, ROW_SIZE);
    printf("Generated matrix:\n");
    printMatrix(matrix, size - 1, ROW_SIZE);

    for (int i = 1; i < size; i++) {
        row_t row = matrix[i - 1];
        error_t errc = MPI_Send(row, ROW_SIZE, MPI_INT, i, i,
MPI_COMM_WORLD);
        if (errc != MPI_SUCCESS) return errc;
    }

    for (int i = 1; i < size; i++) {
```

```
        MPI_Status s;
        element_t localRow[ROW_SIZE];
        error_t errc;

        errc = MPI_Recv(&localRow, ROW_SIZE, MPI_INT, i, size + i,
MPI_COMM_WORLD, &s);
        if (errc != MPI_SUCCESS) return errc;

        for (int j = 0; j < ROW_SIZE; j++) matrix[i - 1][j] =
localRow[j];
    }

    printf("Result matrix:\n");
    printMatrix(matrix, size - 1, ROW_SIZE);

    element_t reference = matrix[size - 2][ROW_SIZE - 1];
    printf("Last element: %d\n", reference);

    for (int i = 1; i < size; i++) {
        error_t errc = MPI_Send(&reference, 1, MPI_INT, i, 2 * size +
i, MPI_COMM_WORLD);
        if (errc != MPI_SUCCESS) return errc;
    }

    int sum = 0;
    int count = 0;
    for (int i = 1; i < size; i++) {
        MPI_Status s;
        int localSum, localCount;
        error_t errc;

        errc = MPI_Recv(&localSum, 1, MPI_INT, i, 3 * size + i,
MPI_COMM_WORLD, &s);
        if (errc != MPI_SUCCESS) return errc;

        errc = MPI_Recv(&localCount, 1, MPI_INT, i, 4 * size + i,
MPI_COMM_WORLD, &s);
        if (errc != MPI_SUCCESS) return errc;

        sum += localSum;
        count += localCount;
    }

    printf("Amount of elements greater than %d: %d\n", reference,
count);
    printf("Sum of elements greater than %d: %d\n", reference, sum);

    switch(count) {
    case 0:
        printf("Calculated average: N/A\n");
```

```
        break;
    default:
        printf("Calculated average: %f\n", (float)(sum) / count);
        break;
    }

    for (int i = 0; i < size - 1; i++) free(matrix[i]);
    free(matrix);
    return 0;
}


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

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

error_t calculatorProcess(int rank, int size) {
    MPI_Status s;
    element_t row[ROW_SIZE];
    error_t errc = MPI_Recv(&row, ROW_SIZE, MPI_INT, 0, rank,
MPI_COMM_WORLD, &s);
    if (errc != MPI_SUCCESS) return errc;

    printf("\t%d got row: ", rank);
    for (int i = 0; i < ROW_SIZE; i++) printf("%d ", row[i]);
    printf("\n");

    balanceSigns(row, ROW_SIZE);

    printf("\t%d new row: ", rank);
    for (int i = 0; i < ROW_SIZE; i++) printf("%d ", row[i]);
    printf("\n");

    errc = MPI_Send(&row, ROW_SIZE, MPI_INT, 0, size + rank,
MPI_COMM_WORLD);
    if (errc != MPI_SUCCESS) return errc;

    int reference;
```

```
    errc = MPI_Recv(&reference, 1, MPI_INT, 0, 2 * size + rank,
MPI_COMM_WORLD, &s);
    if (errc != MPI_SUCCESS) return errc;

    int sum = 0;
    int count = 0;
    for (int i = 0; i < ROW_SIZE; i++) {
        if (row[i] > reference) {
            sum += row[i];
            count++;
        }
    }

    errc = MPI_Send(&sum, 1, MPI_INT, 0, 3 * size + rank,
MPI_COMM_WORLD);
    if (errc != MPI_SUCCESS) return errc;

    errc = MPI_Send(&count, 1, MPI_INT, 0, 4 * size + rank,
MPI_COMM_WORLD);
    if (errc != MPI_SUCCESS) return errc;

    return 0;
}

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_INVAL_ARG;
    }

    error_t errc = !rank ? mainProcess(size) : calculatorProcess(rank,
size);

    MPI_Finalize();
    return errc;
}
```