МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

Умножение матрицы на матрицу в MPI 2D решетка

студента 2 курса, группы 23201

Смирнова Гордея Андреевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.С. Матвеев

Новосибирск 2025

СОДЕРЖАНИЕ

ЦЕЛЬ

Научиться писать базовые параллельные программы с помощью MPI.

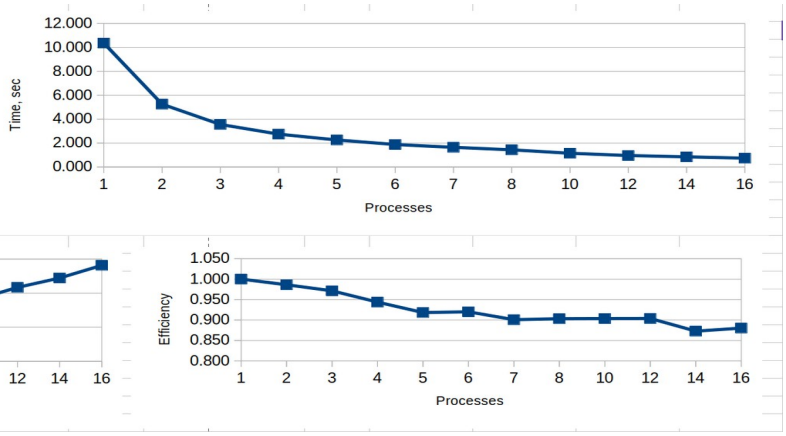
ЗАДАНИЕ

1. Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке.
2. Исследовать производительность параллельной программы в зависимости от размера матрицы и размера решетки.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер.

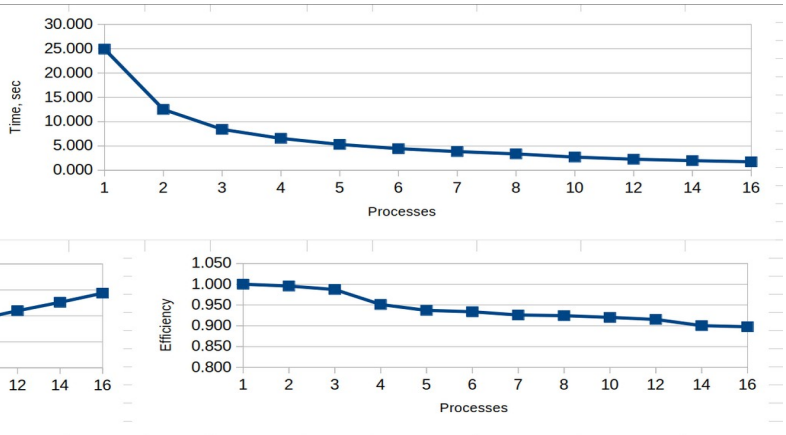
ОПИСАНИЕ РАБОТЫ

На языке C++ была написана MPI-программа для вычисления произведения матриц с использованием 2D-топологии MPI-процессов. В качестве дополнительного задания был реализован алгоритм, работающий с матрицами не только размера кратного размерам сетки, но и произвольного размера. Ниже приведены результаты замеров производительности при умножении матриц 1500x1500, 2000x2000, 2500x2500 соответственно, на числе процессов от 1-8 и 10, 12, 14, 16, а также профилирование на 16 процессах.

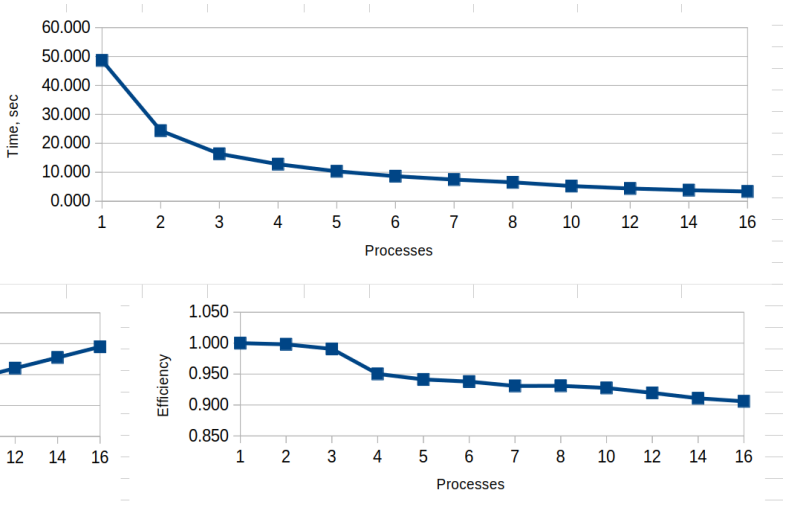
Processes	Time, sec	Acceleration	Efficiency
1	10.359	1.000	1.000
2	5.252	1.972	0.986
3	3.555	2.914	0.971
4	2.744	3.775	0.944
5	2.256	4.591	0.918
6	1.877	5.520	0.920
7	1.643	6.305	0.901
8	1.434	7.226	0.903
10	1.147	9.035	0.904
12	0.955	10.845	0.904
14	0.848	12.219	0.873
16	0.736	14.083	0.880

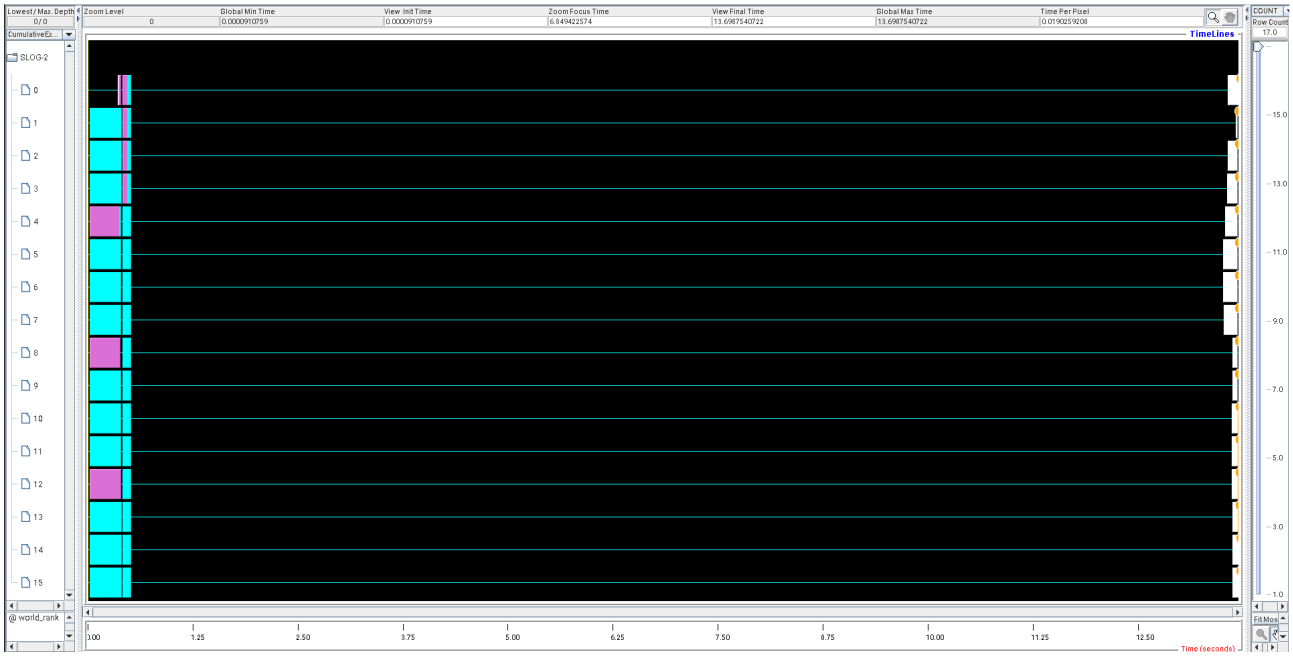


Processes	Time, sec	Acceleration	Efficiency
1	24.909	1.000	1.000
2	12.509	1.991	0.996
3	8.411	2.961	0.987
4	6.547	3.805	0.951
5	5.316	4.685	0.937
6	4.446	5.603	0.934
7	3.843	6.482	0.926
8	3.368	7.395	0.924
10	2.707	9.202	0.920
12	2.268	10.983	0.915
14	1.976	12.604	0.900
16	1.734	14.362	0.898



Processes	Time, sec	Acceleration	Efficiency
1	48.626	1.000	1.000
2	24.360	1.996	0.998
3	16.362	2.972	0.991
4	12.793	3.801	0.950
5	10.332	4.706	0.941
6	8.642	5.627	0.938
7	7.463	6.515	0.931
8	6.529	7.448	0.931
10	5.242	9.275	0.928
12	4.407	11.035	0.920
14	3.813	12.752	0.911
16	3.355	14.494	0.906





По результатам профилирования установлено, что вычисления (т.е. вызов метода locMultiply) занимает абсолютное большинство времени, что говорит об эффективности распараллеливания.

ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были изучены функции библиотеки MPI, позволяющие работать с декартовыми топологиями и параллелить вычислительные задачи не только в одномерном пространстве процессов, но и в 2D решетке.

ПРИЛОЖЕНИЯ

Приложение 1: *MatrixMultiplier.h*

```
#pragma once

#include <mpi.h>
#include <vector>

typedef std::vector<float> fvector;

class MatrixMultiplier {
 public:
    explicit MatrixMultiplier(int size, int rank, int N1, int N2, int N3);
    void locMultiply();
    void gatherMatC();
    void printC() const;

 private:
    void createGrid();
    void distributeMatA();
    void distributeMatB();
    static void fillMatrix(fvector& mat, int rows, int columns);

    const int gridSize, procRank;
    const int N1, N2, N3;
    int procCoords[2];

    fvector globA, globB, globC;
    fvector locA, locB, locC;
    int locN1, locN3;
    std::vector<int> allLocN1, allLocN3;

    MPI_Comm COMM_GRID;
    MPI_Comm COMM_ROW;
    MPI_Comm COMM_COLUMN;
};
```

Приложение 2: *MatrixMultiplier.cpp*

```
#include "MatrixMultiplier.h"
#include <iostream>
#include <vector>

MatrixMultiplier::MatrixMultiplier(int size, int rank, int N1, int N2, int
N3)
: gridSize(size), procRank(rank), N1(N1), N2(N2), N3(N3) {
    createGrid();
    if (procRank == 0) {
        fillMatrix(globA, N1, N2);
        fillMatrix(globB, N2, N3);
```

```
        globC = fvector(N1 * N3);
    }
    distributeMatA();
    distributeMatB();
}

void MatrixMultiplier::createGrid() {
    int dims[2] = {0, 0};
    MPI_Dims_create(gridSize, 2, dims);

    int periods[2] = {0, 0};
    int reorder = 0;

    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &COMM_GRID);
    MPI_Cart_coords(COMM_GRID, procRank, 2, procCoords);

    int remainDims[2] = {0, 1};
    MPI_Cart_sub(COMM_GRID, remainDims, &COMM_ROW);

    remainDims[0] = 1;
    remainDims[1] = 0;
    MPI_Cart_sub(COMM_GRID, remainDims, &COMM_COLUMN);
}

void MatrixMultiplier::distributeMatA() {
    int colCommSize;
    MPI_Comm_size(COMM_COLUMN, &colCommSize);
    int rowCommSize = gridSize / colCommSize;

    allLocN1 = std::vector<int>(gridSize);

    std::vector<int> sendcounts(colCommSize);
    std::vector<int> displs(colCommSize);
    const int baseSize = N1 / colCommSize;
    const int remainder = N1 % colCommSize;
    int shift = 0;
    for (int i = 0; i < colCommSize; ++i) {
        int locRowsCount = baseSize + (i < remainder);
        for (int j = 0; j < rowCommSize; ++j) {
            allLocN1[i * rowCommSize + j] = locRowsCount;
        }
        sendcounts[i] = locRowsCount * N2;
        displs[i] = shift;
        shift += sendcounts[i];
    }

    locN1 = allLocN1[procRank];
    const int locSize = locN1 * N2;
    locA = fvector(locSize);

    if (procCoords[1] == 0) {
        MPI_Scatterv(globA.data(), sendcounts.data(), displs.data(),
MPI_FLOAT,
            locA.data(), locSize, MPI_FLOAT, 0, COMM_COLUMN);
```

```
    }

    MPI_Bcast(locA.data(), locSize, MPI_FLOAT, 0, COMM_ROW);
}

void MatrixMultiplier::distributeMatB() {
    int rowCommSize;
    MPI_Comm_size(COMM_ROW, &rowCommSize);
    int colCommSize = gridSize / rowCommSize;

    allLocN3 = std::vector<int>(gridSize);

    std::vector<int> sendcounts(rowCommSize);
    std::vector<int> displs(rowCommSize);
    const int baseSize = N3 / rowCommSize;
    const int remainder = N3 % rowCommSize;
    int shift = 0;
    for (int i = 0; i < rowCommSize; ++i) {
        sendcounts[i] = baseSize + (i < remainder);
        for (int j = 0; j < colCommSize; ++j) {
            allLocN3[j * rowCommSize + i] = sendcounts[i];
        }
        displs[i] = shift;
        shift += sendcounts[i];
    }

    locN3 = allLocN3[procRank];
    const int locSize = locN3 * N2;
    locB = fvector(locSize);

    if (procCoords[0] == 0) {
        MPI_Datatype col;
        MPI_Type_vector(N2, 1, N3, MPI_FLOAT, &col);
        MPI_Type_create_resized(col, 0, sizeof(float), &col);
        MPI_Type_commit(&col);

        MPI_Scatterv(globB.data(), sendcounts.data(), displs.data(), col,
            locB.data(), locSize, MPI_FLOAT, 0, COMM_ROW);

        MPI_Type_free(&col);
    }

    MPI_Bcast(locB.data(), locSize, MPI_FLOAT, 0, COMM_COLUMN);
}

void MatrixMultiplier::locMultiply() {
    locC = fvector(locN1 * locN3);
    for (int i = 0; i < locN1; ++i) {
        for (int j = 0; j < locN3; ++j) {
            for (int k = 0; k < N2; ++k) {
                locC[i * locN3 + j] += locA[i * N2 + k] * locB[j * N2 + k];
            }
        }
    }
```

```
}

void MatrixMultiplier::gatherMatC() {
    int rowCommSize;
    MPI_Comm_size(COMM_ROW, &rowCommSize);

    std::vector<int> bigRowSendcounts(gridSize),
smallRowSendcounts(gridSize);
    std::vector<int> bigRowRecvcounts(gridSize),
smallRowRecvcounts(gridSize);
    std::vector<int> displs(gridSize);

    int baseDispls = 0;
    for (int i = 0; i < gridSize; ++i) {
        const int recvcount = allLocN3[i];
        if (recvcount == allLocN3[0]) {
            bigRowSendcounts[i] = allLocN1[i] * allLocN3[i];
            bigRowRecvcounts[i] = recvcount;
            smallRowSendcounts[i] = 0;
            smallRowRecvcounts[i] = 0;
        } else {
            bigRowSendcounts[i] = 0;
            bigRowRecvcounts[i] = 0;
            smallRowSendcounts[i] = allLocN1[i] * allLocN3[i];
            smallRowRecvcounts[i] = recvcount;
        }
        if (i == 0) continue;
        if (i % rowCommSize == 0) {
            baseDispls += N3 * allLocN1[i - 1];
            displs[i] = baseDispls;
        } else {
            displs[i] = displs[i - 1] +
                bigRowRecvcounts[i - 1] + smallRowRecvcounts[i - 1];
        }
    }

    MPI_Datatype bigRowBlock, smallRowBlock;
    MPI_Type_vector(allLocN1[0], allLocN3[0],
        N3, MPI_FLOAT, &bigRowBlock);
    MPI_Type_vector(allLocN1[0], allLocN3[0] - 1,
        N3, MPI_FLOAT, &smallRowBlock);
    MPI_Type_create_resized(bigRowBlock, 0, sizeof(float), &bigRowBlock);
    MPI_Type_create_resized(smallRowBlock, 0, sizeof(float),
&smallRowBlock);
    MPI_Type_commit(&bigRowBlock);
    MPI_Type_commit(&smallRowBlock);

    MPI_Gatherv(locC.data(), bigRowSendcounts[procRank], MPI_FLOAT,
        globC.data(), bigRowRecvcounts.data(), displs.data(),
        bigRowBlock, 0, MPI_COMM_WORLD);
    MPI_Gatherv(locC.data(), smallRowSendcounts[procRank], MPI_FLOAT,
        globC.data(), smallRowRecvcounts.data(), displs.data(),
        smallRowBlock, 0, MPI_COMM_WORLD);
```

```
    MPI_Type_free(&bigRowBlock);
    MPI_Type_free(&smallRowBlock);
}

void MatrixMultiplier::printC() const {
    if (procRank == 0) {
        for (int i = 0; i < N1; ++i) {
            for (int j = 0; j < N3; ++j) {
                std::cout << globC[i * N3 + j] << " ";
            }
            std::cout << std::endl;
        }
    }
}

void MatrixMultiplier::fillMatrix(fvector& mat, int rows, int columns) {
    size_t size = rows * columns;
    mat.resize(size);
    for (size_t i = 0; i < size; i++) {
        mat[i] = static_cast<float>(i % columns);
    }
}
```

Приложение 3: *main.cpp*

```
#include <mpi.h>
#include <cfloat>
#include <cstdlib>
#include <iostream>
#include <string>
#include <stdexcept>
#include <algorithm>
#include "MatrixMultiplier.h"

#define LOOPS 8

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 (argc < 4) {
        if (rank == 0) std::cerr <<
            "Usage: " << argv[0] << " N1 N2 N3" << std::endl;
        MPI_Finalize();
        return EXIT_FAILURE;
    }

    int N1 = std::atoi(argv[1]);
    int N2 = std::atoi(argv[2]);
    int N3 = std::atoi(argv[3]);
```

```
    if (N1 ⩽ 0 || N2 ⩽ 0 || N3 ⩽ 0) throw std::invalid_argument(
        "All N must be positive");

    double duration = DBL_MAX;
    for (int i = 0; i < LOOPS; i++) {
        const double start = MPI_Wtime();
        MatrixMultiplier mm(size, rank, N1, N2, N3);
        mm.locMultiply();
        mm.gatherMatC();
        mm.printC();
        const double end = MPI_Wtime();
        duration = std::min(duration, end - start);
    }

    if (rank == 0) std::cout <<
        "Time taken: " << duration << " sec" << std::endl;

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```