###### 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 ЛАБОРАТОРНОЙ РАБОТЫ

«Умножение матрицы на матрицу в MPI 2D решетка»

студента 2 курса, 22208 группы

**Лебедева Антона Андреевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Кудинов

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ГРАФИКИ 4](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 7](#_Toc18443924)

[Приложение Листинг 8](#_Toc18443925)

# ЦЕЛЬ

Разработать и исследовать параллельную программу умножения матрицы на матрицу с применением одной из библиотек, реализующих стандарты MPI.

# ЗАДАЧА

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.**ГРАФИКИ**

**Графики разбиения**

Мы видим, что самым эффективным является разбиение 4x4, так как данные равномерно распределены между процессами.

**ЗАКЛЮЧЕНИЕ**

# В результате выполнения работы была освоена работа с коммуникаторами MPI, распараллелено решение умножения матрицы на матрицу, замерено ускорение и эффективность программы, проведено исследование по использованию целесообразности использования разного разбиения матрицы C по процессов.

Было выяснено, что распараллеливание данной задачи является крайне эффективным. При тестировании разных разбиений матрицы на 16-ти процессах мы определили, что 4x4 является наиболее эффективным.

# ЛИСТИНГ

# Вариант 1

#include <mpi.h>

#include <iostream>

const std::size\_t N1 = 1600;

const std::size\_t N2 = 1600;

const std::size\_t N3 = 1600;

double\* MatrixBuilder(const std::size\_t n1, const std::size\_t n2) {

    double\* matrix = new double[n1 \* n2];

    std::fill\_n(matrix, n1 \* n2, 3.0);

    return matrix;

}

void CreateComm(int\* dims, MPI\_Comm\* distr\_rows\_comm, MPI\_Comm\* distr\_cols\_comm) {

    int periods[2] = {0};

    MPI\_Comm\* comm\_cart = new MPI\_Comm;

    MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, true, comm\_cart);

    int distrDims[2] = {1, 0};

    MPI\_Cart\_sub(\*comm\_cart, distrDims, distr\_rows\_comm);

    distrDims[0] = 0;

    distrDims[1] = 1;

    MPI\_Cart\_sub(\*comm\_cart, distrDims, distr\_cols\_comm);

    delete comm\_cart;

}

int\* Sendcounts(const std::size\_t n1, const std::size\_t n2, const int size) {

    int\* sendcounts = new int[size];

    for (std::size\_t r = 0; r < size; ++r) {

        sendcounts[r] = ((n1 / size) + (int)(n1 % size > r)) \* n2;

    }

    return sendcounts;

}

std::size\_t OffSet(const std::size\_t n, const int size, const int rank) {

    std::size\_t offset = 0;

    for (int i = 0; i < rank; ++i) {

        offset += (n / size) + (int)(n % size > i);

    }

    return offset;

}

int\* Displs(const std::size\_t n1, const std::size\_t n2, const int size) {

    int\* displs = new int[size];

    for (std::size\_t r = 0; r < size; ++r) {

        displs[r] = OffSet(n1, size, r) \* n2;

    }

    return displs;

}

double\* ScatterRows(MPI\_Comm\* distr\_comm, double\* m, int\* dims, const int len) {

    double\* partM = new double[len \* N2];

    int\* sendcounts = Sendcounts(N1, N2, dims[0]);

    int\* displs = Displs(N1, N2, dims[0]);

    MPI\_Scatterv(m, sendcounts, displs, MPI\_DOUBLE, partM, len \* N2, MPI\_DOUBLE, 0, \*distr\_comm);

    delete[] sendcounts;

    delete[] displs;

    return partM;

}

double\* ScatterCols(MPI\_Comm\* distr\_comm, double\* m, int\* dims, const int len) {

    double\* partM = new double[len \* N2];

    MPI\_Datatype cols, colsFinal;

    MPI\_Type\_vector(N2, len, N3, MPI\_DOUBLE, &cols);

    MPI\_Type\_commit(&cols);

    MPI\_Type\_create\_resized(cols, 0, len \* sizeof(double), &colsFinal);

    MPI\_Type\_commit(&colsFinal);

    int rank;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Scatter(m, 1, colsFinal, partM, len \* N2, MPI\_DOUBLE, 0, \*distr\_comm);

    MPI\_Type\_free(&colsFinal);

    MPI\_Type\_free(&cols);

    return partM;

}

double\* MultMatrix(double\* partA, double\* partB, const std::size\_t rowsA, const std::size\_t colsB) {

    double\* partMatrixC = new double[rowsA \* colsB]{0};

    for (int i = 0; i < rowsA; i++) {

        for (int k = 0; k < N2; k++) {

            for (int j = 0; j < colsB; j++) {

                partMatrixC[i \* colsB + j] += partA[i \* N2 + k] \* partB[k \* colsB + j];

            }

        }

    }

    return partMatrixC;

}

int\* DisplsC(int\* dims) {

    int countPart = 0;

    int\* displs = new int[dims[0] \* dims[1]]{0};

    for (int i = 0; i < dims[0]; ++i) {

        for (int j = 0; j < dims[1]; ++j) {

            displs[i \* dims[1] + j] = countPart;

            countPart += 1;

        }

        countPart += ((N1 / dims[0]) + (int)(N1 % dims[0] > i) - 1) \* dims[1];

    }

    return displs;

}

int\* SendcountsC(int\* dims) {

    int\* sendcounts = new int[dims[0] \* dims[1]]{0};

    for (int i = 0; i < dims[0]; ++i) {

        for (int j = 0; j < dims[1]; ++j) {

            sendcounts[i \* dims[1] + j] = 1;

        }

    }

    return sendcounts;

}

double\* GatherC(int\* dims, const int sizeA, const int sizeB, double\* partC) {

    int rank;

    double\* mC;

    MPI\_Datatype part, partFinale;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Type\_vector(sizeA, sizeB, N3, MPI\_DOUBLE, &part);

    MPI\_Type\_commit(&part);

    MPI\_Type\_create\_resized(part, 0, sizeB \* sizeof(double), &partFinale);

    MPI\_Type\_commit(&partFinale);

    int\* displs = DisplsC(dims);

    int\* sendcounts = SendcountsC(dims);

    if (rank == 0) {

        mC = new double[N1 \* N3];

    }

    MPI\_Gatherv(partC, sizeA \* sizeB, MPI\_DOUBLE, mC, sendcounts, displs, partFinale, 0, MPI\_COMM\_WORLD);

    MPI\_Type\_free(&partFinale);

    MPI\_Type\_free(&part);

    return mC;

}

void CheckingSolution(double\* mA, double\* mB, double\* mC) {

    double\* tempC = MultMatrix(mA, mB, N1, N3);

    bool equal = true;

    for (int i = 0; i < N3; i++) {

        for (int j = 0; j < N1; j++) {

            if (mC[i \* N3 + j] - tempC[i \* N3 + j] != 0) {

                equal = false;

            }

        }

    }

    printf("Matrix Equally:(%d)\n", (int)equal);

    delete[] tempC;

}

int main(int argc, char\*\* argv) {

    double startTime, endTime;

    int\* dims = new int[2]{0};

    int rank, size;

    double\* matrixA = nullptr;

    double\* matrixB = nullptr;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    matrixA = MatrixBuilder(N1, N2);

    matrixB = MatrixBuilder(N2, N3);

    startTime = MPI\_Wtime();

    MPI\_Dims\_create(size, 2, dims);

    MPI\_Comm\* distr\_rows\_comm = new MPI\_Comm;

    MPI\_Comm\* distr\_cols\_comm = new MPI\_Comm;

    CreateComm(dims, distr\_rows\_comm, distr\_cols\_comm);

    int drows\_rank, dcols\_rank;

    MPI\_Comm\_rank(\*distr\_rows\_comm, &drows\_rank);

    int lenPartA = (N1 / dims[0]) + (int)(N1 % dims[0] > drows\_rank);

    double\* partA = ScatterRows(distr\_rows\_comm, matrixA, dims, lenPartA);

    MPI\_Comm\_rank(\*distr\_cols\_comm, &dcols\_rank);

    int lenPartB = (N3 / dims[1]) + (int)(N3 % dims[1] > dcols\_rank);

    double\* partB = ScatterCols(distr\_cols\_comm, matrixB, dims, lenPartB);

    double\* partC = MultMatrix(partA, partB, lenPartA, lenPartB);

    double\* matrixC = GatherC(dims, lenPartA, lenPartB, partC);

    if (rank == 0) {

endTime = MPI\_Wtime();

        CheckingSolution(matrixA, matrixB, matrixC);

        std::cout << "Time: " << endTime - startTime << std::endl;

        delete[] matrixA;

        delete[] matrixB;

        delete[] matrixC;

    }

    delete distr\_rows\_comm;

    delete distr\_cols\_comm;

    delete[] partA;

    delete[] partB;

    delete[] partC;

    delete[] dims;

    MPI\_Finalize();

    return 0;

}