###### 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

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью OpenMP»

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)

[ПРОФИЛИРОВАНИЕ 6](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 8](#_Toc18443924)

[Приложение Листинг 9](#_Toc18443925)

# ЦЕЛЬ

Разработать и исследовать параллельные программы решения СЛАУ методом простой итерации с применением одной из библиотек, реализующих стандарты MPI.

# ЗАДАЧА

# Написать последовательную программу, реализующую итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b, распараллелить с помощью MPI.

# Реализовать два варианта программы:

# Вариант 1: векторы x и b дублируются в каждом MPI-процессе

# Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A

# Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1,2, 4, 8, 16. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.

# Выполнить профилирование двух вариантов программы с помощью MPE при использовании 16-и ядер.

# ГРАФИКИ

**ПРОФИЛИРОВАНИЕ**

**ЗАКЛЮЧЕНИЕ**

# ЛИСТИНГ

# Вариант 1

#include <mpi.h>

#include <cmath>

#include <cstdlib>

#include <cstring>

#include <ctime>

#include <iostream>

#include <vector>

using matrix\_cont = std::vector<std::vector<double>>;

const std::size\_t N = 1300;

const double e = 0.000001;

std::size\_t FindLrows(int size, int rank) {

    return (N / size + ((int)(N % size) > rank));

}

std::size\_t FindBegin(int size, int rank) {

    std::size\_t begin = 0;

    for (int i = 0; i < rank; ++i) {

        begin += (N / size) + ((int)(N % size) > i);

    }

    return begin;

}

matrix\_cont MatrixBuilder(int size, int rank) {

    std::size\_t matrixBegin = FindBegin(size, rank);

    std::size\_t lrows = FindLrows(size, rank);

    matrix\_cont matrix(lrows, std::vector<double>(N));

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

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

            if (j == i + matrixBegin) {

                matrix[i][j] = 2.0;

            } else {

                matrix[i][j] = 1.0;

            }

        }

    }

    return matrix;

}

void RandomVectorX(double\* x) {

    srand(time(NULL));

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

        x[i] = rand() % 10;

    }

}

int\* FindLrowsV(int size) {

    int\* array = new int[size];

    for (int j = 0; j < size; ++j) {

        array[j] = (N / size) + (int)((int)(N % size) > j);

    }

    return array;

}

int\* FindBeginV(int size) {

    int\* array = new int[size];

    for (int j = 0; j < size; j++) {

        array[j] = FindBegin(size, j);

    }

    return array;

}

void AMultX(matrix\_cont& matrix, double\* x, double\* b) {

    int size, rank;

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

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

    std::size\_t lrows = FindLrows(size, rank);

    std::size\_t matrixBegin = FindBegin(size, rank);

    int\* lrowsV = FindLrowsV(size);

    int\* beginV = FindBeginV(size);

    double b1[lrows];

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

        b1[i] = b[i + matrixBegin];

    }

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

        for (std::size\_t j = matrixBegin; j < matrixBegin + lrows; ++j) {

            b1[i] += x[j] \* matrix[i][j];

        }

    }

    MPI\_Allgatherv(b1, lrows, MPI\_DOUBLE, b, lrowsV, beginV, MPI\_DOUBLE, MPI\_COMM\_WORLD);

    delete[] lrowsV;

    delete[] beginV;

}

void Sub(double\* x1, double\* b) {

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

        x1[i] -= b[i];

    }

}

void MultT(double\* x1, double t) {

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

        x1[i] \*= t;

    }

}

double Module(double\* u) {

    double a = 0;

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

        a += (u[i] \* u[i]);

    }

    return sqrt(a);

}

void SearchX(matrix\_cont& matrix, double\* x, double\* b) {

    std::size\_t count = 0;

    double x0[N] = {0};

    double x1[N] = {0};

    double u = 0;

    double bNorm = 0;

    double uOld = 0;

    double t = 0.001;

    int size, rank;

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

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

    AMultX(matrix, x0, x1);

    Sub(x1, b);

    bNorm = Module(b);

    do {

        if (u > uOld && (count % 5 == 0)) {

            t = (-t);

        }

        uOld = u;

        MultT(x1, t);

        Sub(x0, x1);

        AMultX(matrix, x0, x1);

        Sub(x1, b);

        u = Module(x1);

        u = u / bNorm;

        count++;

    } while (u > e);

    Sub(x0, x);

    if (rank == 0) {

        std::cout << "u:" << u << std::endl;

        std::cout << "NormalX: " << Module(x0) << std::endl;

        std::cout << rank << " Vector found" << std::endl;

    }

}

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

    double startTime, endTime;

    int rank, size;

    double x[N];

    double b[N] = {0};

    MPI\_Init(&argc, &argv);

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

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

    matrix\_cont matrix = MatrixBuilder(size, rank);

    if (rank == 0) {

        startTime = MPI\_Wtime();

        RandomVectorX(x);

    }

    MPI\_Bcast(x, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    AMultX(matrix, x, b);

    SearchX(matrix, x, b);

    if (rank == 0) {

        endTime = MPI\_Wtime();

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

    }

    MPI\_Finalize();

    return 0;

}

# Вариант 2

#include <mpi.h>

#include <algorithm>

#include <cmath>

#include <cstdlib>

#include <ctime>

#include <iostream>

const std::size\_t N = 1300;

const double e = 0.000001;

double t = 0.001;

std::size\_t FindLrows(int size, int rank) {

    return (N / size + ((int)(N % size) > rank));

}

std::size\_t FindBegin(int size, int rank) {

    std::size\_t begin = 0;

    for (int i = 0; i < rank; ++i) {

        begin += ((N / size) + (int)((int)(N % size) > i));

    }

    return begin;

}

double\*\* MatrixBuilder(int size, int rank) {

    std::size\_t matrixBegin = FindBegin(size, rank);

    std::size\_t lrows = FindLrows(size, rank);

    double\*\* matrix = new double\*[lrows];

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

        matrix[i] = new double[N];

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

            if (j == i + matrixBegin) {

                matrix[i][j] = 2.0;

            } else {

                matrix[i][j] = 1.0;

            }

        }

    }

    return matrix;

}

void RandomVectorX(double\* x) {

    srand(time(NULL));

    int size, rank;

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

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

    std::size\_t lrows = FindLrows(size, rank);

    srand(time(NULL));

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

        x[i] = rand() % 10;

    }

}

int\* FindLrowsV(int size) {

    int\* array = new int[size];

    for (int j = 0; j < size; ++j) {

        array[j] = (N / size) + (int)((int)(N % size) > j);

    }

    return array;

}

int\* FindBeginV(int size) {

    int\* array = new int[size];

    for (int j = 0; j < size; j++) {

        array[j] = FindBegin(size, j);

    }

    return array;

}

void AMultX(double\*\* matrix, double\* xOld, double\* b) {

    int size, rank;

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

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

    std::size\_t lrows = FindLrows(size, rank);

    double\* x = new double[lrows];

    int\* lrowsV = FindLrowsV(size);

    int\* beginV = FindBeginV(size);

    std::copy\_n(xOld, lrows, x);

    for (int p = 0; p < size; ++p) { *// полный подсчёт по всем процессам*

        for (std::size\_t i = 0; i < lrows; ++i) { *// проходимся по всем строкам матрицы*

            for (int j = 0; j < lrowsV[rank]; ++j) {

                b[i] += x[j] \* matrix[i][j + beginV[rank]];

            }

        }

        double\* x1 = new double[lrowsV[(rank + 1) % size]]();

        MPI\_Sendrecv(x, lrowsV[rank], MPI\_DOUBLE, (rank + size - 1) % size, 123, x1, lrowsV[(rank + 1) % size], MPI\_DOUBLE, (rank + 1) % size, 123, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        std::swap(x, x1);

        std::rotate(lrowsV, lrowsV + 1, lrowsV + size);

        std::rotate(beginV, beginV + 1, beginV + size);

        delete[] x1;

    }

    delete[] lrowsV;

    delete[] beginV;

}

void Sub(double\* x, double\* b, std::size\_t lrows) {

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

        x[i] -= b[i];

    }

}

void MultT(double\* x, std::size\_t lrows) {

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

        x[i] \*= t;

    }

}

double Module(double\* u, std::size\_t& lrows, int& size) {

    double a = 0;

    double sum = 0;

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

        a += (u[i] \* u[i]);

    }

    if (size == 1) {

        return sqrt(a);

    }

    MPI\_Allreduce(&a, &sum, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

    return sqrt(sum);

}

void SearchX(double\*\* matrix, double\* x, double\* b) {

    std::size\_t count = 0;

    double u = 0;

    double bNorm = 0;

    double uOld = 0;

    int size, rank;

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

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

    std::size\_t lrows = FindLrows(size, rank);

    double\* x0 = new double[lrows]();

    double\* x1 = new double[lrows]();

    bNorm = Module(b, lrows, size);

    AMultX(matrix, x0, x1);

    Sub(x1, b, lrows);

    do {

        if (u > uOld && (count % 5 == 0)) {

            t = (-t);

        }

        uOld = u;

        MultT(x1, lrows);

        Sub(x0, x1, lrows);

        AMultX(matrix, x0, x1);

        Sub(x1, b, lrows);

        u = Module(x1, lrows, size);

        u = u / bNorm;

        count++;

    } while (u > e);

    Sub(x0, x, lrows);

    bNorm = Module(x0, lrows, size);

    if (rank == 0) {

        std::cout << "u:" << u << std::endl;

        std::cout << "NormalX: " << bNorm << std::endl;

        std::cout << rank << " Vector found" << std::endl;

    }

}

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

    double startTime, endTime;

    int size, rank;

    MPI\_Init(&argc, &argv);

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

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

    std::size\_t lrows = FindLrows(size, rank);

    double b[lrows] = {0};

    double\* x = new double[lrows]();

    double\*\* matrix = MatrixBuilder(size, rank);

    if (rank == 0) {

        startTime = MPI\_Wtime();

    }

    RandomVectorX(x);

    AMultX(matrix, x, b);

    SearchX(matrix, x, b);

    if (rank == 0) {

        endTime = MPI\_Wtime();

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

    }

    MPI\_Finalize();

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

        delete[] matrix[i];

    }

    delete[] matrix;

    delete[] x;

    return 0;

}