###### 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 MPI»

студента 2 курса, 22202 группы

**Бальчинова А.С.**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

В.А. Перепёлкин

Новосибирск 2024

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ОПИСАНИЕ РАБОТЫ 4-6](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 6](#_Toc18443924)

[Приложение. 7](#_Toc18443925)-11

# ЦЕЛЬ

Ознакомиться со стандартом MPI, изучить некоторые его функции и проверить эффективность его применения на практике.

# ЗАДАНИЕ

1. Написать программу на языке C или C++, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A – матрица размером N×N, x и b – векторы длины N. Тип элементов – double.
2. Программу распараллелить с помощью MPI с разрезанием матрицы A по строкам на близкие по размеру, возможно не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. 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. Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

1. 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.
2. Выполнить профилирование двух вариантов программы с помощью MPE при использовании 16-и ядер.
3. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы

# ОПИСАНИЕ РАБОТЫ

В ходе работы был реализован метод простой итерации, написана последовательная программа, затем распараллелена.

Было написано два варианта распараллеливания итерационного алгоритма.

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

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

Затем были произведены замеры времени работы итерационного алгоритма при использовании различного числа процессорных ядер.

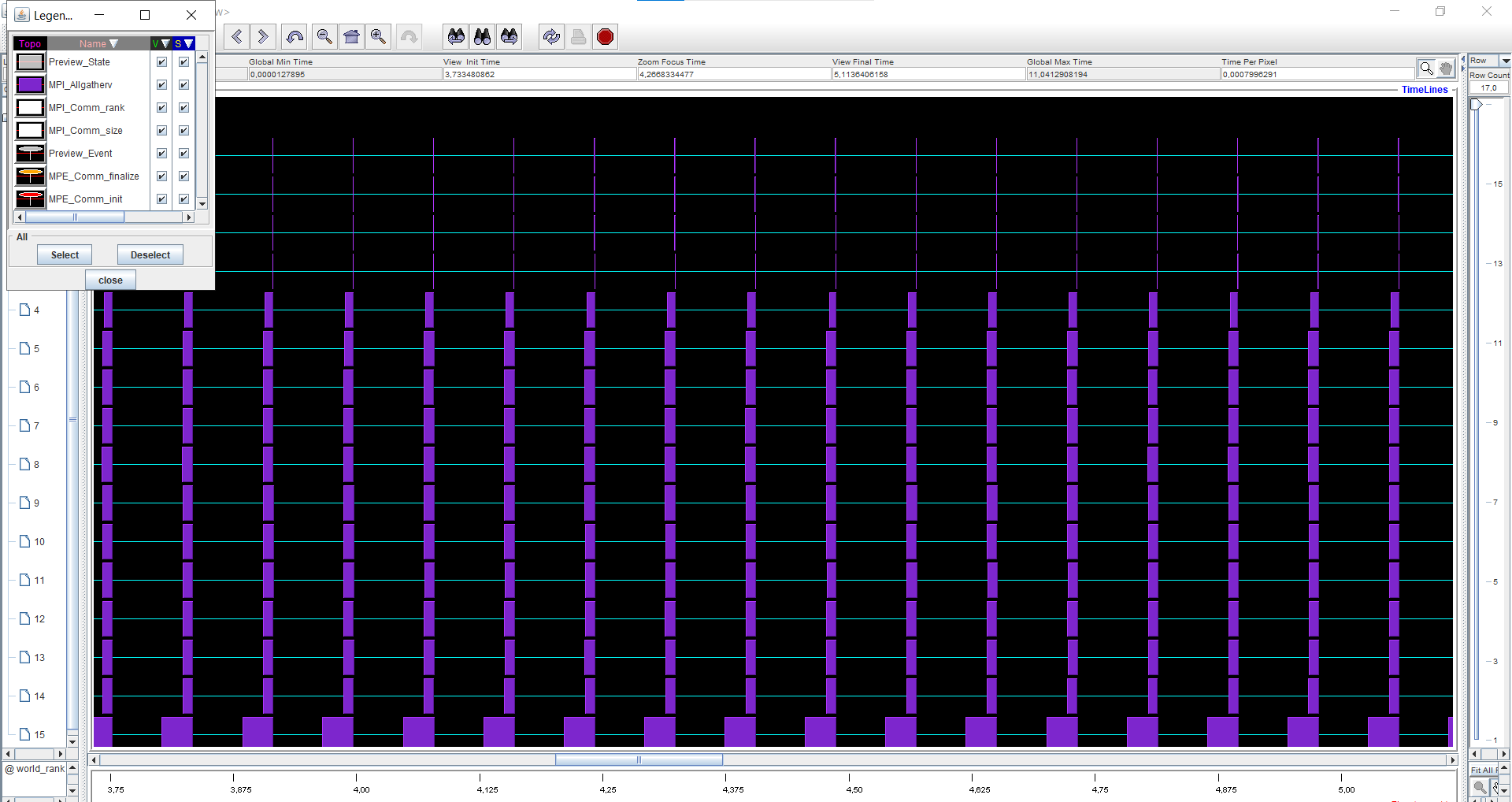
Построены графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

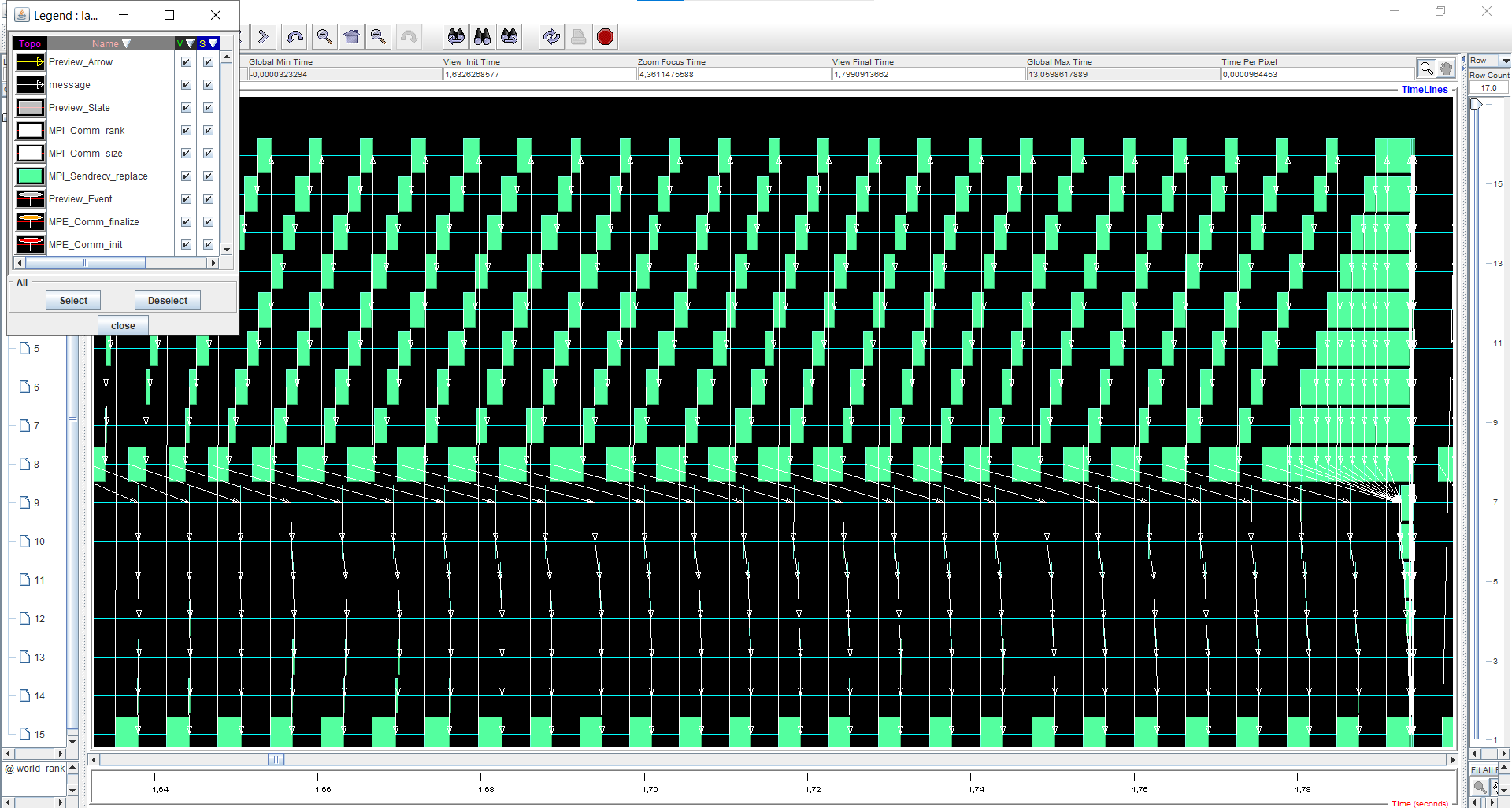
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Число потоков | 1 | 2 | 4 | 8 | 16 |
| Время выполнения 1 | 38,838 | 20,984 | 13,203 | 12,267 | 10,697 |
| Ускорение S | 1 | 1,851 | 2,941 | 3,166 | 3,631 |
| Эффективность E, % | 100 | 92.5 | 73.5 | 39,6 | 22,7 |
| Время выполнения 2 | 38,829 | 21,007 | 13,656 | 12,738 | 11,603 |
| Ускорение, раз | 1 | 1,848 | 2,843 | 3,048 | 3,346 |
| Эффективность, % | 100 | 92,4 | 71,0 | 38,1 | 20,9 |

# 

# Как можно видеть из графиков, ускорение в обоих вариантах растёт примерно одинаково, хотя и заметно, что вариант 1 слегка превосходит второй.

# Падение эффективности распараллеливания примерно одинаковое.





# ЗАКЛЮЧЕНИЕ

В ходе анализа времени работы программы, а так же зависимости ускорения и эффективности программы от числа потоков, было выяснено, что увеличение числа ядер после определённого значения уже не даёт значительного прироста скорости и не является оправданным.

# ПРИЛОЖЕНИЕ 1 (Вариант 1)

#include <iostream>

#include <cmath>

#include <mpi.h>

#define N 17280

using namespace std;

int num\_of\_proc;

int proc\_rank;

double \*buffer;

int \*dataStarts;

int \*dataLengths;

double Norm(const double \*u, const int size) {

    double res = 0;

    for (int i = 0; i < size; i++) {

        res += u[i] \* u[i];

    }

    return sqrt(res);

}

void mult(const double \*matrix, const double \*vect, double \*result, int size) {

    for (int i = 0; i < dataLengths[proc\_rank]; i++) {

        double sum = 0;

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

            sum += matrix[i \* size + j] \* vect[j];

        }

        result[i] = sum;

    }

}

void mult(double \*a, const int size, const double tau) {

    for (int i = 0; i < size; i++) {

        a[i] \*= tau;

    }

}

void sub(const double \*a, const double \*b, double \*c, const int size) {

    for (int i = 0; i < size; i++) {

        c[i] = a[i] - b[i];

    }

}

struct Context {

    double \*A;

    double \*x;

    double \*b;

    double tau;

    double \*vMult;    //A\*x^n

    double \*vSub;     //A\*x^n - b

    const int size;

    const double epsilon;

    Context(const int n, const double epsilon) : size(n), epsilon(epsilon) {

        A = new double[size \* dataLengths[proc\_rank]];

        x = new double[size];

        b = new double[size];

        tau = 0.0001;

        vMult = new double[dataLengths[proc\_rank]];

        vSub = new double[dataLengths[proc\_rank]];

        for (int i = 0; i < size; i++) {

            x[i] = 0;

            b[i] = size + 1;

        }

        for (int i = 0; i < dataLengths[proc\_rank]; i++) {

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

                if (dataStarts[proc\_rank] + i == j) {

                    A[i \* size + j] = 2.0;

                }

                else {

                    A[i \* size + j] = 1.0;

                }

            }

        }

    }

    ~Context() {

        delete[] A;

        delete[] x;

        delete[] b;

        delete[] vMult;

        delete[] vSub;

    }

};

void gatherVector(double \*from, double \*to) {

    MPI\_Allgatherv(from, dataLengths[proc\_rank],  MPI\_DOUBLE, to, dataLengths, dataStarts, MPI\_DOUBLE, MPI\_COMM\_WORLD);

}

bool isCloseEnough(Context &cont) {

    mult(cont.A, cont.x, cont.vMult, cont.size);

    sub(cont.vMult, cont.b, cont.vSub, dataLengths[proc\_rank]);

    gatherVector(cont.vSub, buffer);

    double res = Norm(buffer, cont.size) / Norm(cont.b, cont.size);

    return res < cont.epsilon;

}

void next(Context &cont) {

    mult(cont.A, cont.x, cont.vMult, cont.size);

    sub(cont.vMult, cont.b, cont.vSub, dataLengths[proc\_rank]);

    mult(cont.vSub, dataLengths[proc\_rank], cont.tau);

    sub(cont.x, cont.vSub, cont.vMult, dataLengths[proc\_rank]);

    gatherVector(cont.vMult, cont.x);

}

void InitRowsPerProcess(int n) {

    dataStarts = new int[n];

    dataLengths = new int[n];

    int perProcess = n / num\_of\_proc;

    int ost = n % num\_of\_proc;

    int start = 0;

    for (int i = 0; i < ost; i++) {

        dataStarts[i] = start;

        dataLengths[i] = perProcess + 1;

        start += perProcess + 1;

    }

    for (int i = ost; i < num\_of\_proc; i++) {

        dataStarts[i] = start;

        dataLengths[i] = perProcess;

        start += perProcess;

    }

}

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

    double begin, end;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_of\_proc);

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

    const double epsilon = pow(10, -9);

    InitRowsPerProcess(N);  //Разрезаем матрицу

    Context cont = Context(N, epsilon);

    buffer = new double[N];

    begin = MPI\_Wtime();

    while (!isCloseEnough(cont)) {

        next(cont);

    }

    end = MPI\_Wtime();

    cout << "Time diff = " << (end - begin) << "[s]" <<endl;

    delete[] dataStarts;

    delete[] dataLengths;

    delete[] buffer;

    MPI\_Finalize();

}

# ПРИЛОЖЕНИЕ 2 (Вариант 2)

#include <iostream>

#include <cmath>

#include <mpi.h>

#define N 17280

using namespace std;

int num\_of\_proc;

int proc\_rank;

int \*dataStarts;

int \*dataLengths;

void shiftData(double \*data) {  //Сдвиг данных по кольцу

    int rankFrom = (proc\_rank + num\_of\_proc - 1) % num\_of\_proc;

    int rankTo = (proc\_rank + 1) % num\_of\_proc;

    MPI\_Sendrecv\_replace(data, dataLengths[0], MPI\_DOUBLE, rankTo, 123, rankFrom, 123, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

double Norm(double \*u) {

    double res = 0;

    int currRank = proc\_rank;

    for (int i = 0; i < num\_of\_proc; i++) {

        for (int j = 0; j < dataLengths[currRank]; j++) {

            res += u[j] \* u[j];

        }

        shiftData(u);

        currRank = (currRank + num\_of\_proc - 1) % num\_of\_proc;

    }

    return sqrt(res);

}

void mult(const double \*matrix, double \*vect, double \*result, const int size) {

    int currRank = proc\_rank;

    for (int i = 0; i < dataLengths[proc\_rank]; i++) {

        result[i] = 0;

    }

    for (int k = 0; k < num\_of\_proc; k++) {

        int offset = dataStarts[currRank];

        for (int i = 0; i < dataLengths[proc\_rank]; i++) {

            double sum = 0;

            for (int j = offset; j < offset + dataLengths[currRank]; j++) {

                sum += matrix[i \* size + j] \* vect[j - offset];

            }

            result[i] += sum;

        }

        shiftData(vect);

        currRank = (currRank + num\_of\_proc - 1) % num\_of\_proc;

    }

}

void mult(double \*a, const int size, const double tau) {

    for (int i = 0; i < size; i++) {

        a[i] \*= tau;

    }

}

void sub(const double \*a, const double \*b, double \*c, const int size) {

    for (int i = 0; i < size; i++) {

        c[i] = a[i] - b[i];

    }

}

struct Context {

    double \*A;

    double \*x;

    double \*b;

    double tau;

    double \*multRes;

    double \*subRes;

    const int size;

    const double epsilon;

    Context(const int n, const double epsilon) : size(n), epsilon(epsilon) {

        A = new double[size \* dataLengths[proc\_rank]];

        x = new double[dataLengths[0]];

        b = new double[dataLengths[0]];

        tau = 0.0001;

        multRes = new double[dataLengths[0]];

        subRes = new double[dataLengths[0]];

        for (int i = 0; i < dataLengths[proc\_rank]; i++) {

            x[i] = 0;

            b[i] = size + 1;

        }

        for (int i = 0; i < dataLengths[proc\_rank]; i++) {

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

                if (dataStarts[proc\_rank] + i == j) {

                    A[i \* size + j] = 2.0;

                } else {

                    A[i \* size + j] = 1.0;

                }

            }

        }

    }

    ~Context() {

        delete[] A;

        delete[] x;

        delete[] b;

        delete[] multRes;

        delete[] subRes;

    }

};

bool isCloseEnough(Context &cont) {

    mult(cont.A, cont.x, cont.multRes, cont.size);

    sub(cont.multRes, cont.b, cont.subRes, dataLengths[proc\_rank]);

    double res = Norm(cont.subRes) / Norm(cont.b);

    return res < cont.epsilon;

}

void next(Context &cont) {

    mult(cont.A, cont.x, cont.multRes, cont.size);

    sub(cont.multRes, cont.b, cont.subRes, dataLengths[proc\_rank]);

    mult(cont.subRes, dataLengths[proc\_rank], cont.tau);

    sub(cont.x, cont.subRes, cont.x, dataLengths[proc\_rank]);

}

void InitRowsPerProcess(int n) {

    dataStarts = new int[num\_of\_proc];

    dataLengths = new int[num\_of\_proc];

    int perProcess = n / num\_of\_proc;

    int ost = n % num\_of\_proc;

    int start = 0;

    for (int i = 0; i < ost; i++) {

        dataStarts[i] = start;

        dataLengths[i] = perProcess + 1;

        start += perProcess + 1;

    }

    for (int i = ost; i < num\_of\_proc; i++) {

        dataStarts[i] = start;

        dataLengths[i] = perProcess;

        start += perProcess;

    }

}

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

    double begin, end;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_of\_proc);

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

    const double epsilon = pow(10, -9);

    InitRowsPerProcess(N);

    Context cont = Context(N, epsilon);

    begin = MPI\_Wtime();

    while (!isCloseEnough(cont)) {

        next(cont);

    }

    end = MPI\_Wtime();

    cout << "Time diff = " << (end - begin) << "[s]" <<endl;

    delete[] dataStarts;

    delete[] dataLengths;

    MPI\_Finalize();

}