###### 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 метода Якоби в трехмерной области»

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 4-8](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ 8](#_Toc18443924)

[Приложение. 9](#_Toc18443925)-13

**ЦЕЛЬ**

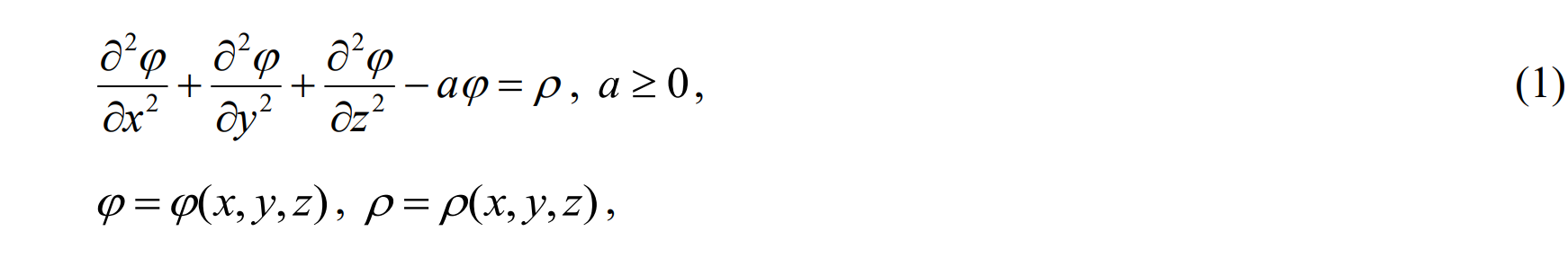
Практическое освоение методов распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области.

# ЗАДАНИЕ

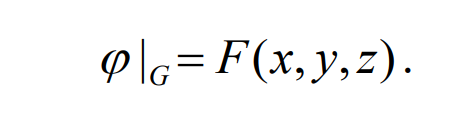
1. Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.
2. Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16. Размеры сетки и порог сходимости подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер.

ОПИСАНИЕ РАБОТЫ

Требуется решить уравнение (т.е. найти функцию φ):

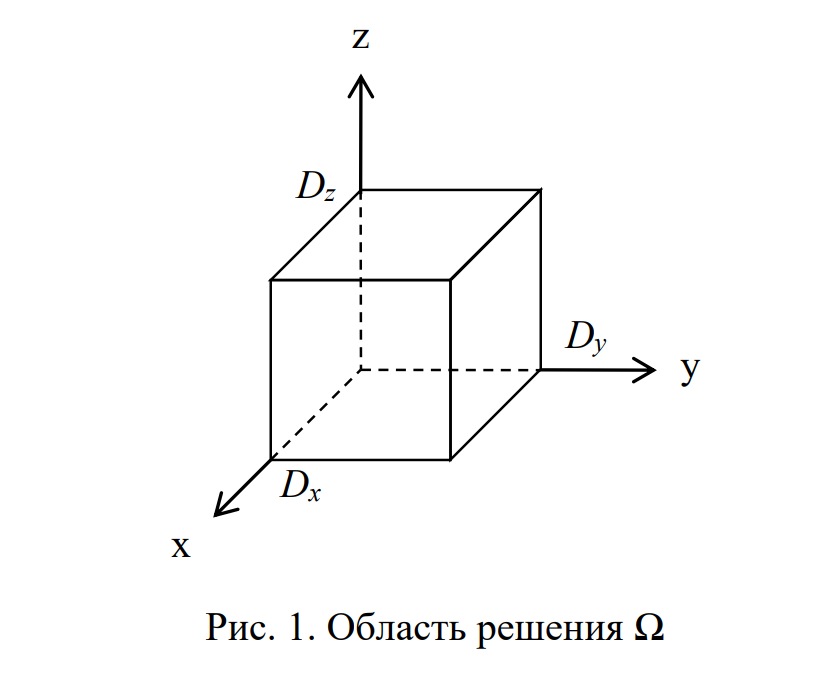


в области Ω с краевыми условиями 1-го рода (т.е. на границе G известны значения искомой функции φ):



Область Ω имеет вид прямоугольного параллелепипеда с размерами

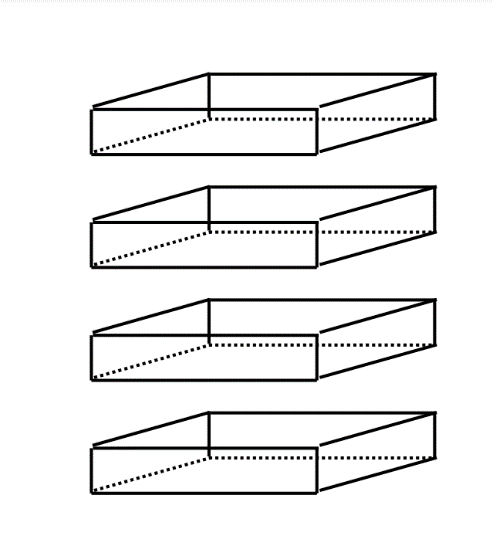
Dx × Dy × Dz.



Для численного решения задачи необходимо перейти к ее дискретному аналогу.

При параллельной реализации алгоритма порядок действий выглядит следующим образом: на каждой итерации все процессы одновременно вычисляют значения в своей подобласти (этап вычислений), после чего происходит обмен граничными значениями (этап обменов).

В ходе работы было реализовано решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области.



Декомпозиция на “линейке”.

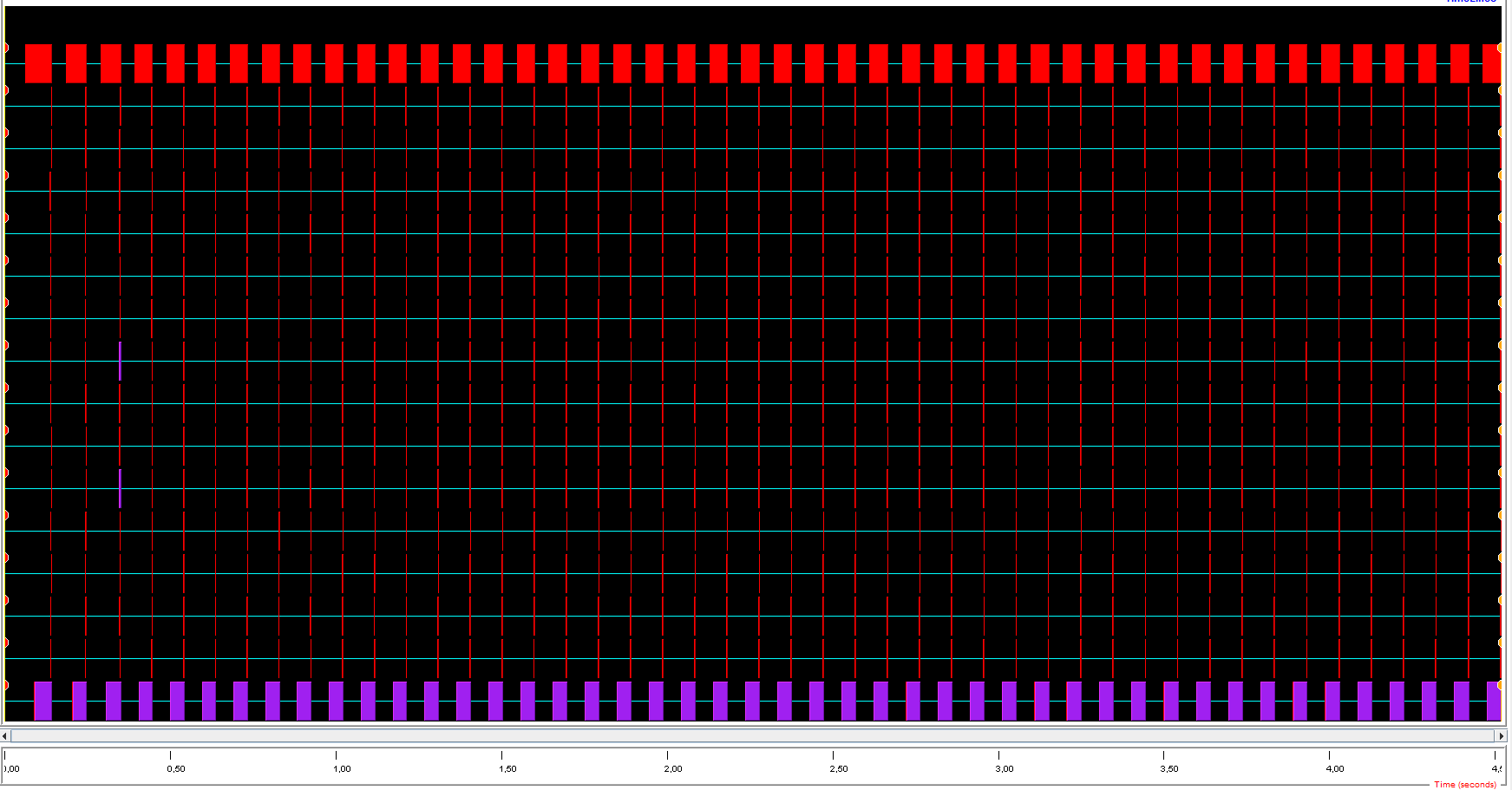
Затем были произведены замеры времени работы итерационного алгоритма при использовании различного числа процессорных ядер.

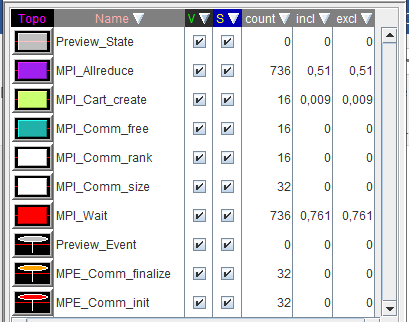
Построены графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

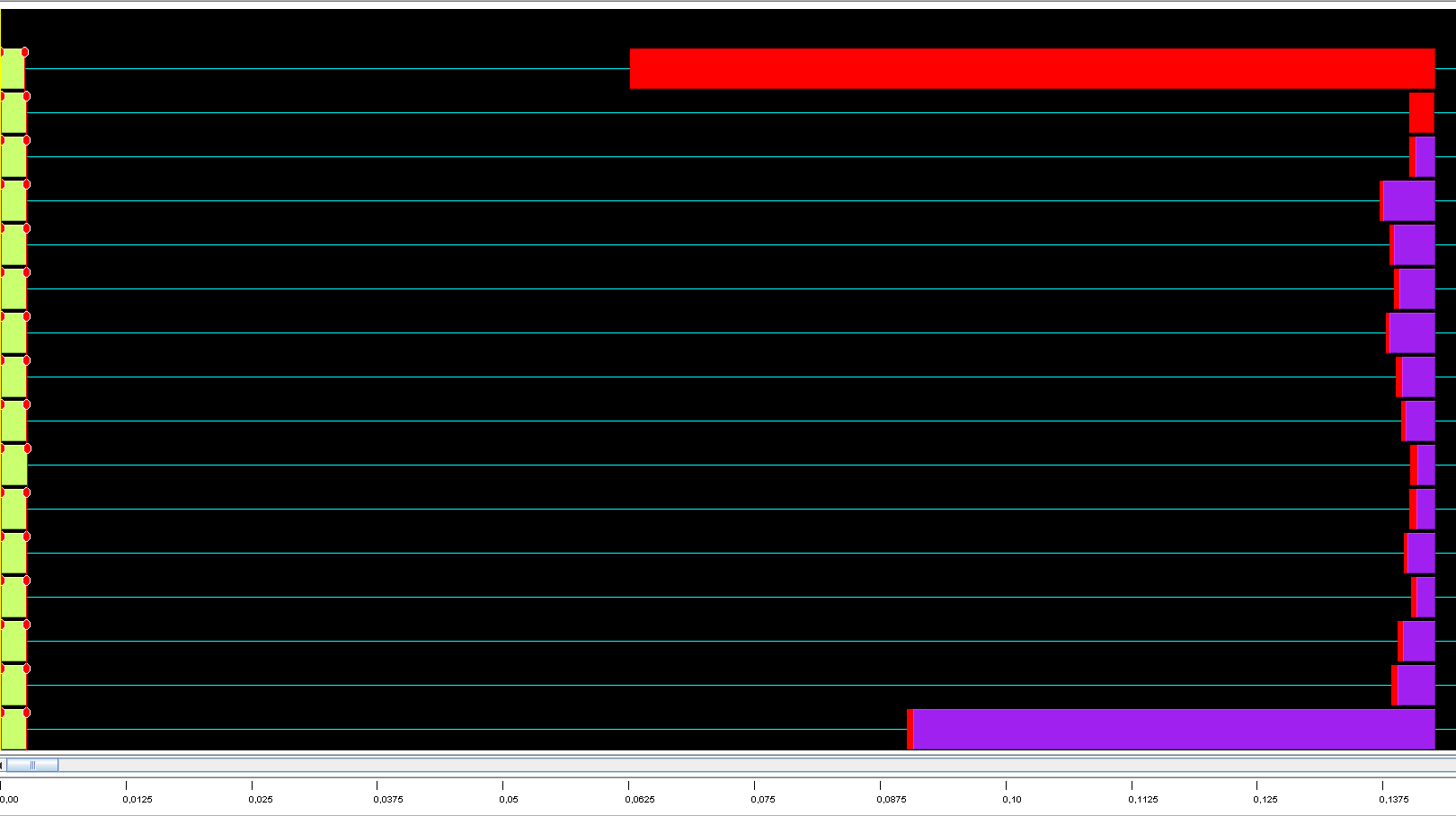
Размеры матриц: Nx = Ny = 324, Nz = 384

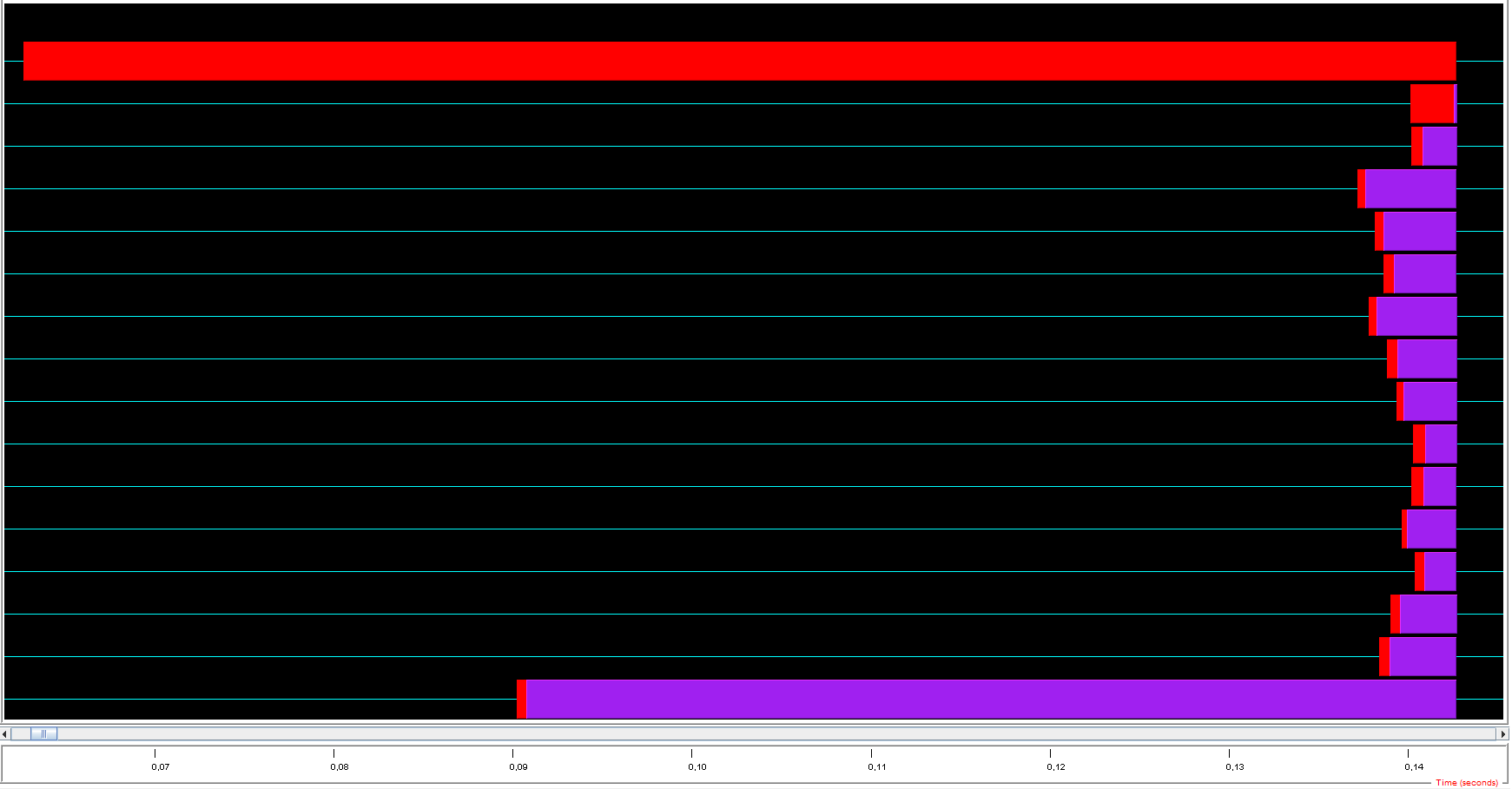
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Число процессов | 1 | 2 | 4 | 8 | 16 |
| Время выполнения | 31,07 | 29,198 | 15,193 | 7,86 | 4,52 |
| Ускорение S | 1 | 1,0641 | 2,045 | 3,9522 | 6,874 |
| Эффективность E, % | 100 | 53,2 | 51,125 | 49,403 | 42,96 |

**Профилирование MPE**

****

****

****

****

# ЗАКЛЮЧЕНИЕ

В процессе выполнения лабораторной работы мы глубже познакомились с MPI, научились применять некоторые его функции на практике, реализовали решение уравнения методом Якоби в трехмерной области, а также исследовали производительность параллельной программы в зависимости от числа процессов.

ПРИЛОЖЕНИЕ

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <math.h>

#include <stdbool.h>

#include <mpi.h>

#define hx2 hx\*hx

#define hy2 hy\*hy

#define hz2 hz\*hz

const double epsilon = 1e-8;

const double a = 1e5;

int Nx = 256, Ny = 256, Nz = 384;

double Dx = 2.0, Dy = 2.0, Dz = 2.0;

double Ox = -1.0, Oy = -1.0, Oz = -1.0;

double hx, hy, hz;

double coeff;

double phi(double x, double y, double z){

    return x\*x + y\*y + z\*z;

}

double rho(double x, double y, double z){

    return 6.0 - a \* phi(x, y, z);

}

double max(double a, double b){

    return a > b ? a : b;

}

MPI\_Comm CreateLinearComm() {

    int commsize;

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

    int ndims = 1;

    int dims[1] = {commsize};

    int periods[1] = {false};

    bool reorder = true;

    MPI\_Comm linear\_comm;

    MPI\_Cart\_create(MPI\_COMM\_WORLD, ndims, dims, periods, reorder, &linear\_comm);

    return linear\_comm;

}

void splitGrid(int num\_of\_proc, int procrank) {

    Nz /= num\_of\_proc;

    Dz /= num\_of\_proc;

    Oz += procrank \* Dz;

    hx = Dx / (Nx - 1);

    hy = Dy / (Ny - 1);

    hz = Dz / (Nz - 1);

    coeff = 1 / (2 / (hx2) + 2 / (hy2) + 2 / (hz2) + a);

}

void initGrid(double (\*grid)[Nx][Ny], bool neighbor\_below, bool neighbor\_above) {

    for (int k = 0; k < Nz; k++){

        for (int i = 0; i < Nx; i++){

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

                double x = Ox + i \* hx;

                double y = Oy + j \* hy;

                double z = Oz + k \* hz;

                if (i == 0 || i == Nx-1 || j == 0 || j == Ny-1 || k == 0 && !neighbor\_below || k == Nz-1 && !neighbor\_above){

                    grid[k][i][j] = phi(x, y, z);

                }

                else{

                    grid[k][i][j] = 0.0;

                }

            }

        }

    }

}

void initBuff(double (\*buff)[Nx][Ny]){

    for (int i = 1; i < Nx - 1; i++){

        for (int j = 1; j < Ny - 1; j++){

            buff[0][i][j] = 0.0;

            buff[1][i][j] = 0.0;

        }

    }

}

void exchangeBoundaries(double (\*grid)[Nx][Ny], double (\*buff)[Nx][Ny], MPI\_Comm comm, MPI\_Request\* request){

    int count = Nx \* Ny;

    int sendcounts[2] = {count, count};

    int recvcounts[2] ={count, count};

    int sdispls[2] = {0, (Nz - 1) \* count};

    int rdispls[2] = {0, count};

    MPI\_Ineighbor\_alltoallv(grid, sendcounts, sdispls, MPI\_DOUBLE, buff, recvcounts, rdispls, MPI\_DOUBLE, comm, request);

}

void updateInnerValues(double (\*curr)[Nx][Ny], double (\*next)[Nx][Ny], double \*delta){

    for (int k = 1; k < Nz - 1; k++){

        for (int i = 1; i < Nx - 1; i++){

            for (int j = 1; j < Ny - 1; j++){

                double x = Ox + i \* hx;

                double y = Oy + j \* hy;

                double z = Oz + k \* hz;

                next[idx(k,i,j)] = coeff \* (

                        (curr[k][i + 1][j] + curr[k][i - 1][j]) / (hx2) +

                        (curr[k][i][j + 1] + curr[k][i][j - 1]) / (hy2) +

                        (curr[k + 1][i][j] + curr[k - 1][i][j]) / (hz2) -

                        rho(x, y, z));

                \*delta = max(\*delta, fabs(next[k][i][j] - curr[k][i][j]));

            }

        }

    }

}

void updateBoundaryValues(double (\*curr)[Nx][Ny], double (\*next)[Nx][Ny], double (\*buff)[Nx][Ny], double \*delta, bool neighbor\_below, bool neighbor\_above) {

    for (int i = 1; i < Nx - 1; i++){

        for (int j = 1; j < Ny - 1; j++){

            double x = Ox + i \* hx;

            double y = Oy + j \* hy;

            if (neighbor\_above){

                int k = Nz - 1;

                double z = Oz + k \* hz;

                next[k][i][j] = coeff \* (

                        (curr[k][i + 1][j] + curr[k][i - 1][j]) / (hx2) +

                        (curr[k][i][j + 1] + curr[k][i][j - 1]) / (hy2) +

                        (buff[1][i][j] + curr[k - 1][i][j]) / (hz2) -

                        rho(x, y, z));

                \*delta = max(\*delta, fabs(next[k][i][j] - curr[k][i][j]));

            }

            if (neighbor\_below){

                int k = 0;

                double z = Oz + k \* hz;

                next[k][i][j] = coeff \* (

                     (curr[k][i + 1][j] + curr[k][i - 1][j]) / (hx \* hx) +

                     (curr[k][i][j + 1] + curr[k][i][j - 1]) / (hy \* hy) +

                     (curr[k + 1][i][j] + buff[0][i][j]) / (hz \* hz) -

                     rho(x, y, z));

                \*delta = max(\*delta, fabs(next[k][i][j] - curr[k][i][j]));

            }

        }

    }

}

void solveJacobi() {

    MPI\_Comm linear\_comm = CreateLinearComm();

    int num\_of\_proc, procrank;

    MPI\_Comm\_size(linear\_comm, &num\_of\_proc);

    MPI\_Comm\_rank(linear\_comm, &procrank);

    splitGrid(num\_of\_proc, procrank);

    bool neighbor\_below = procrank != 0;

    bool neighbor\_above = procrank != num\_of\_proc - 1;

    double(\*curr)[Nx][Ny] = malloc(sizeof(double[Nz][Nx][Ny]));

    InitGrid(curr, neighbor\_below, neighbor\_above);

    double(\*buff)[Nx][Ny] = malloc(sizeof(double[2][Nx][Ny]));

    InitBuff(buff);

    double delta;

    double(\*next)[Nx][Ny];

    do {

        next = malloc(sizeof(double[Nz][Nx][Ny]));

        delta = 0;

        updateBoundaryValues(curr, next, buff, &delta, neighbor\_below, neighbor\_above);

        MPI\_Request exchange\_request;

        exchangeBoundaries(curr, buff, linear\_comm, &exchange\_request);

        updateInnerValues(curr, next, &delta);

        MPI\_Wait(&exchange\_request, MPI\_STATUS\_IGNORE);

        MPI\_Allreduce(MPI\_IN\_PLACE, &delta, 1, MPI\_DOUBLE, MPI\_MAX, linear\_comm);

        free(curr);

        curr = next;

    } while (delta >= epsilon);

    free(curr);

    free(buff);

    MPI\_Comm\_free(&linear\_comm);

}

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

    MPI\_Init(&argc, &argv);

    double begin = MPI\_Wtime();

    solveJacobi();

    double end = MPI\_Wtime();

    printf("Time taken: %lf [s]", end-begin);

    MPI\_Finalize();

    return 0;

}