###### 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 №4

«Параллельная реализация метода Якоби в трехмерной области»

студентки 2 курса, … группы

**…**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

…

Новосибирск 2021

**Содержание**

[Задание 3](#_Toc72775694)

[Описание алгоритма 4](#_Toc72775695)

[График зависимости времени работы программы и ускорения от числа ядер 5](#_Toc72775696)

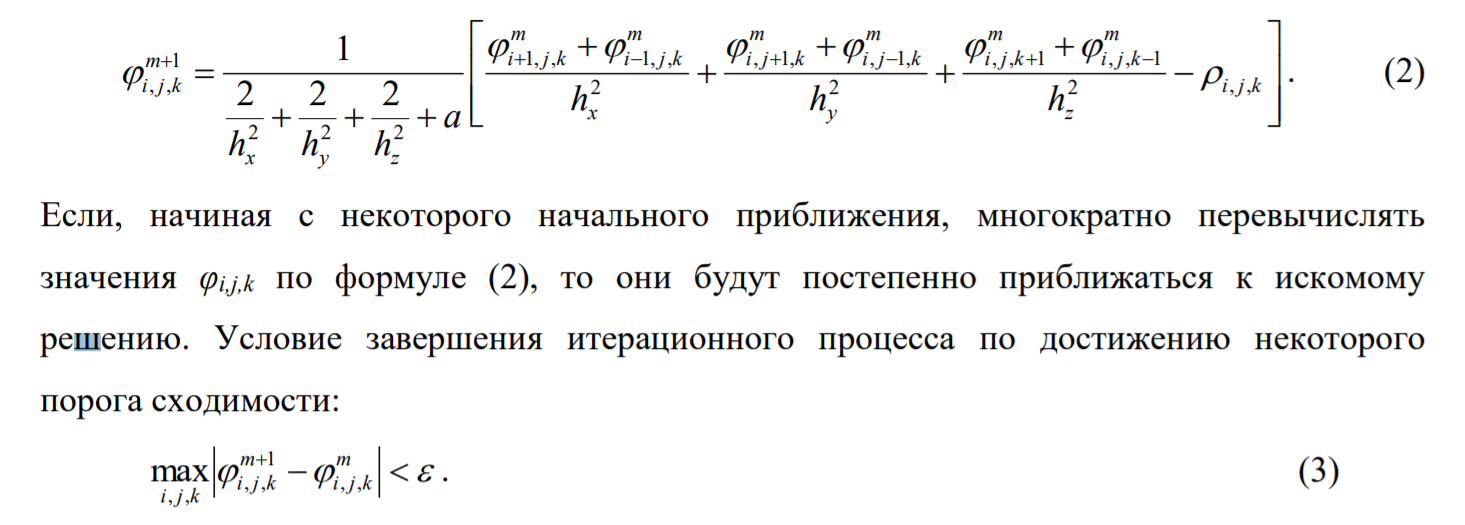
[Профилирование 6](#_Toc72775697)

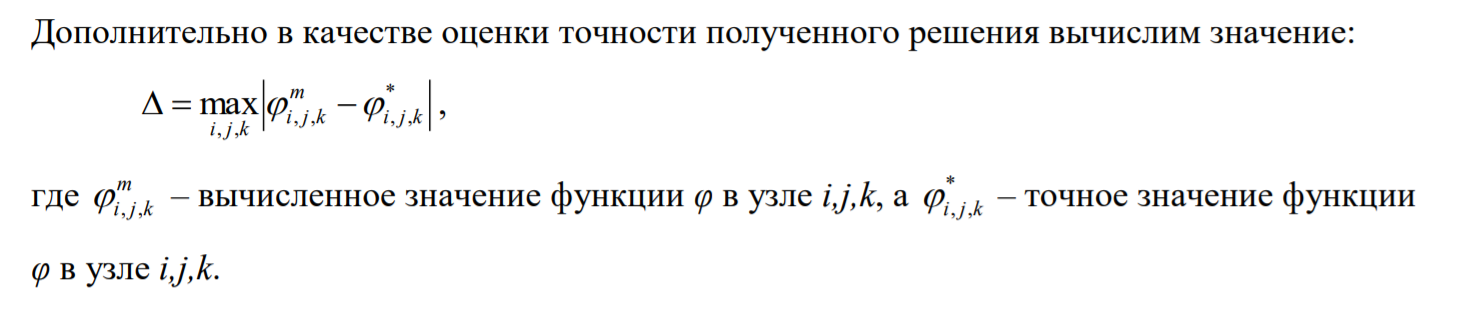
[Листинг программы 7](#_Toc72775698)

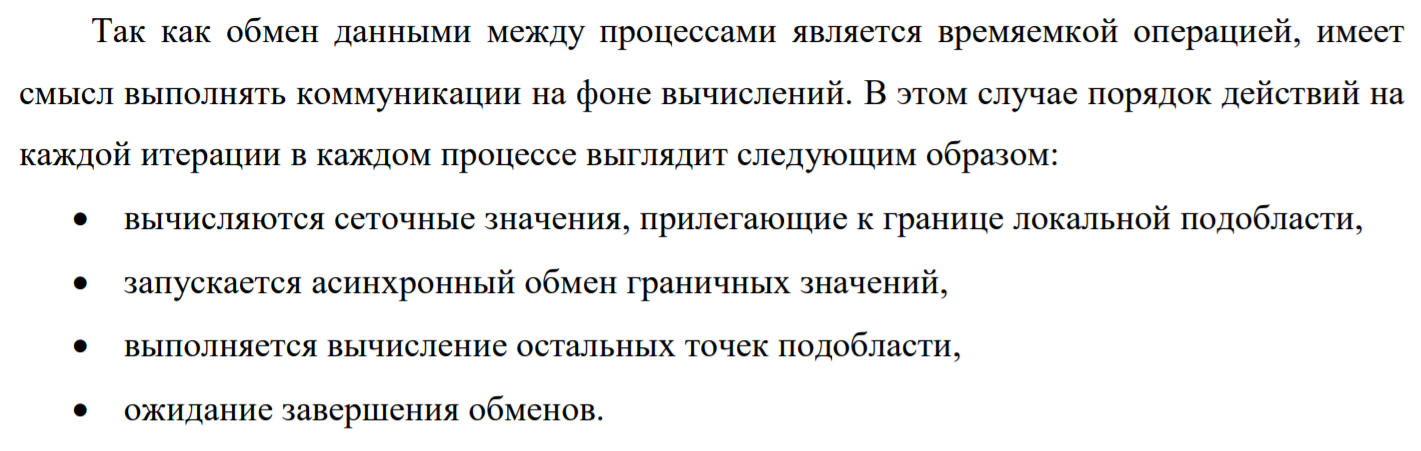
# Задание

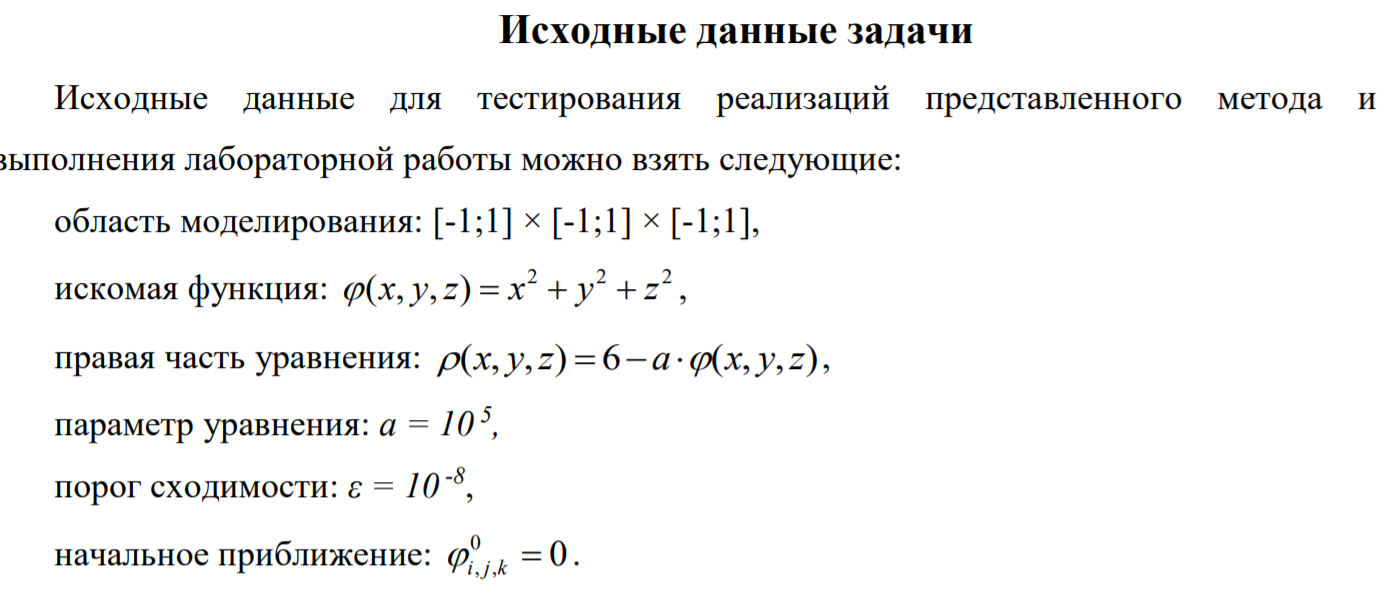
* Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.
* Измерить время работы 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.
* Выполнить профилирование программы с помощью MPE при использовании 16-и ядер. По профилю убедиться, что коммуникации происходят на фоне счета.

# Описание алгоритма



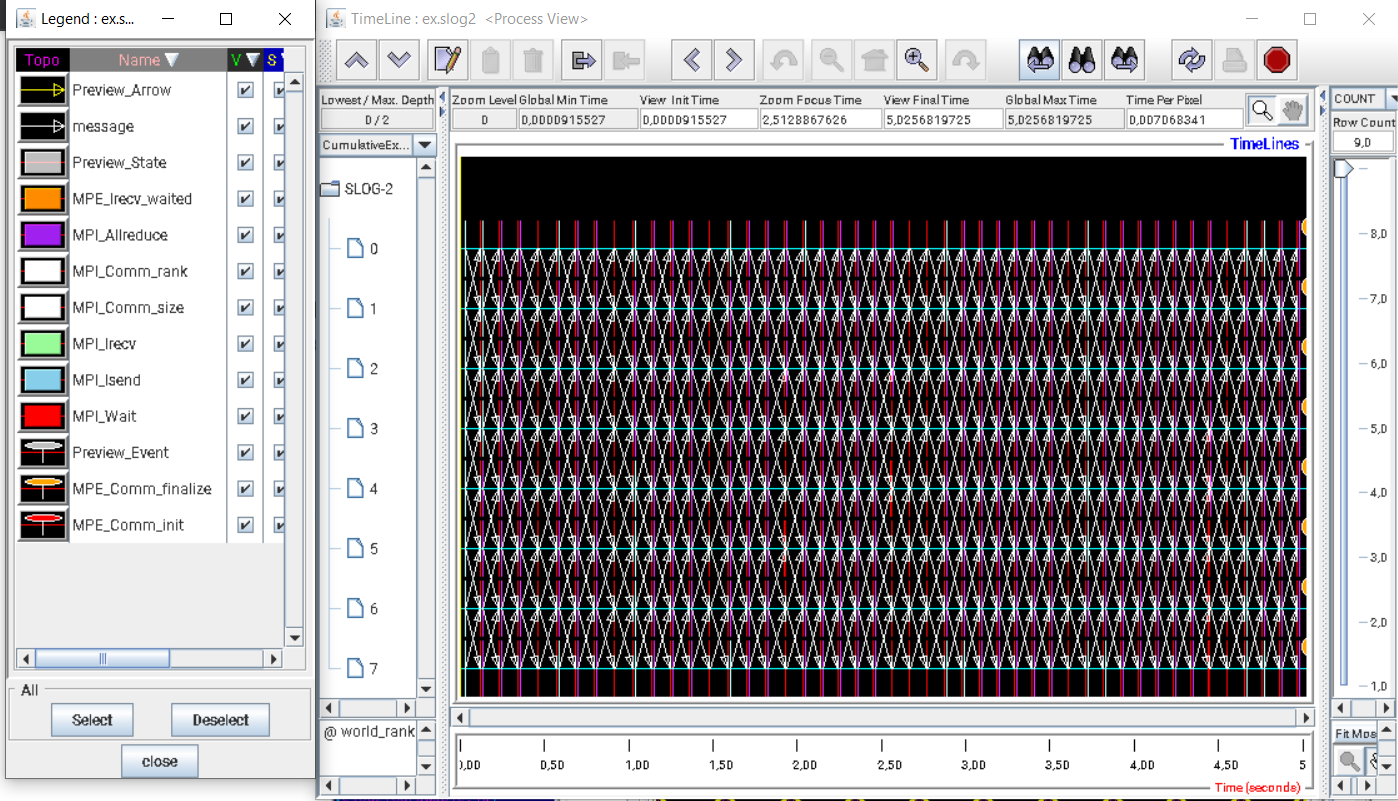


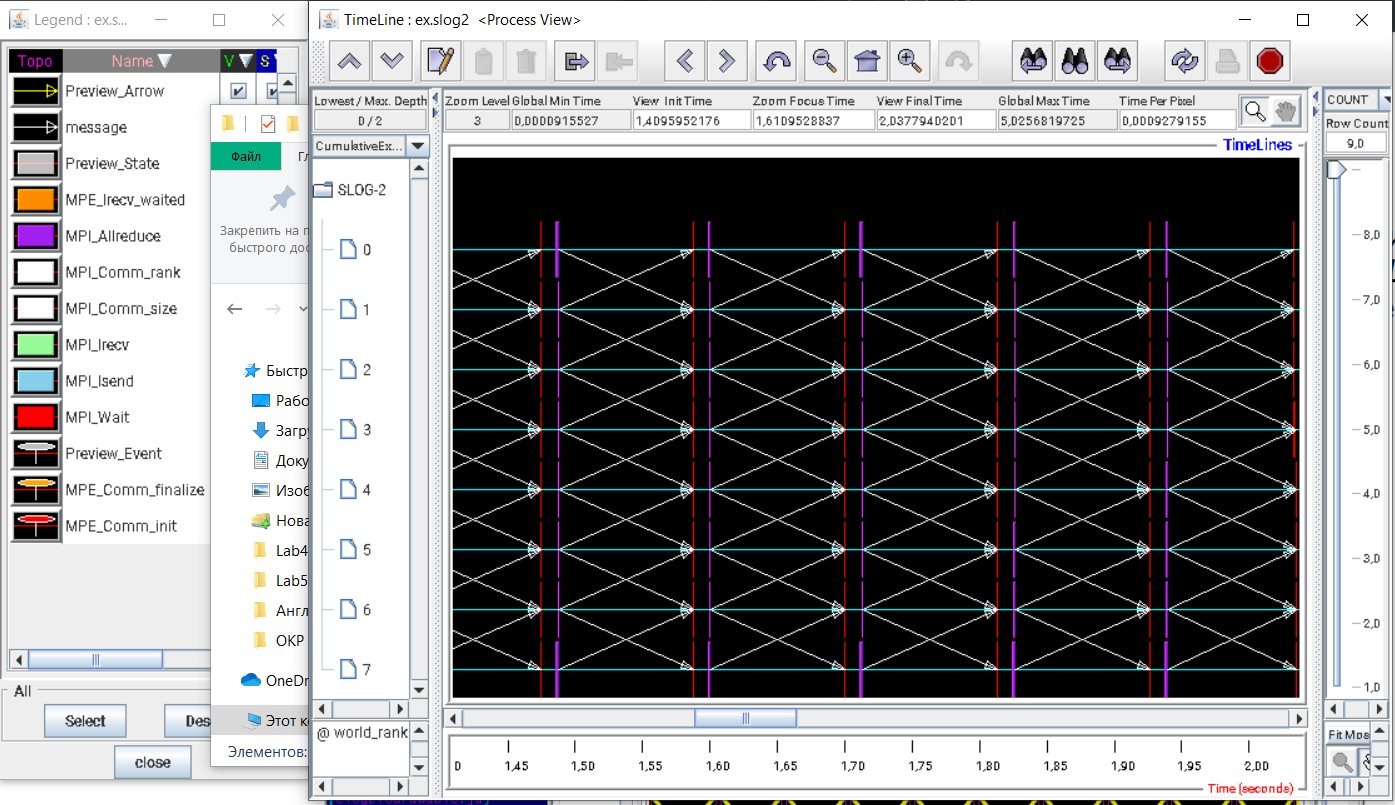




# График зависимости времени работы программы и ускорения от числа ядер

# Профилирование





# Листинг программы

#include <mpi.h>

#include <iostream>

#include <math.h>

#include <cstdlib>

#include <ctime>

#include <string.h>

using namespace std;

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

return x \* x + y \* y + z \* z;

}

const double PARAMETER\_A = 10e5;

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

return 6 - PARAMETER\_A \* realFunction(x, y, z);

}

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

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

MPI\_Init(&argc, &argv);

int rank, size;

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

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

const int Nx = 1600;

const int Ny = 100;

const int Nz = 100;

const double Hx = 2.0 / (Nx - 1);

const double Hy = 2.0 / (Ny - 1);

const double Hz = 2.0 / (Nz - 1);

const double Hx2 = Hx \* Hx;

const double Hy2 = Hy \* Hy;

const double Hz2 = Hz \* Hz;

const double EPSILON = 10e-12;

const double factor = 1 / (2 / Hx2 + 2 / Hy2 + 2 / Hz2 + PARAMETER\_A);

const int X = Nx / size;

const int Y = Ny;

const int Z = Nz;

double\* phi = new double[X \* Y \* Z];

double\* phiNext = new double[X \* Y \* Z];

for (int i = 0; i < X; ++i)

for (int j = 0; j < Y; j++)

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

if ((i + rank \* X == 0) || (i + rank \* X == Nx - 1)

|| (j == 0) || (j == Ny - 1)

|| (k == 0) || (k == Nz - 1)) {

phiNext[i \* Y \* Z + j \* Z + k] = phi[i \* Y \* Z + j \* Z + k] = realFunction((i + rank \* (X - 1)) \* Hx - 1, (j \* Hy) - 1,

(k \* Hz) - 1);

}

else

phiNext[i \* Y \* Z + j \* Z + k] = phi[i \* Y \* Z + j \* Z + k] = 0;

}

MPI\_Request leftBorderSendRequest;

MPI\_Request rightBorderSendRequest;

MPI\_Request leftBorderRecvRequest;

MPI\_Request rightBorderRecvRequest;

double\* leftBorder = new double[Z \* Y];

double\* rightBorder = new double[Z \* Y];

double maxDeltaPhi;

double commonMaxDeltaPhi = 1;

while (commonMaxDeltaPhi > EPSILON) {

if (rank != size - 1) {

MPI\_Isend(&(phi[Y \* Z \* (X - 1) - 1]), Y \* Z, MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD, &rightBorderSendRequest);

MPI\_Irecv(rightBorder, Y \* Z, MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD, &rightBorderRecvRequest);

}

if (rank != 0) {

MPI\_Isend(&(phi[0]), Y \* Z, MPI\_DOUBLE, rank - 1, 0, MPI\_COMM\_WORLD, &leftBorderSendRequest);

MPI\_Irecv(leftBorder, Y \* Z, MPI\_DOUBLE, rank - 1, 0, MPI\_COMM\_WORLD, &leftBorderRecvRequest);

}

maxDeltaPhi = 0;

for (int i = 1; i < X - 1; ++i)

for (int j = 1; j < Y - 1; ++j)

for (int k = 1; k < Z - 1; ++k) {

phiNext[i \* Y \* Z + j \* Z + k] =

factor \* ((phi[(i + 1) \* Y \* Z + j \* Z + k] + phi[(i - 1) \* Y \* Z + j \* Z + k]) / Hx2 +

(phi[i \* Y \* Z + (j + 1) \* Z + k] + phi[i \* Y \* Z + (j - 1) \* Z + k]) / Hy2 +

(phi[i \* Y \* Z + j \* Z + (k + 1)] + phi[i \* Y \* Z + j \* Z + (k - 1)]) / Hz2 -

rightFunction((i + rank \* (X - 1)) \* Hx - 1, j \* Hy - 1, k \* Hz - 1));

double deltaPhi = fabs(phiNext[i \* Y \* Z + j \* Z + k] - phi[i \* Y \* Z + j \* Z + k]);

if (deltaPhi > maxDeltaPhi)

maxDeltaPhi = deltaPhi;

}

if (rank != 0) {

MPI\_Wait(&leftBorderSendRequest, MPI\_STATUS\_IGNORE);

MPI\_Wait(&leftBorderRecvRequest, MPI\_STATUS\_IGNORE);

}

if (rank != size - 1) {

MPI\_Wait(&rightBorderSendRequest, MPI\_STATUS\_IGNORE);

MPI\_Wait(&rightBorderRecvRequest, MPI\_STATUS\_IGNORE);

}

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

for (int k = 1; k < Z - 1; ++k) {

if (rank != 0) {

int i = 0;

phiNext[i \* Y \* Z + j \* Z + k] =

factor \* ((phi[(i + 1) \* Y \* Z + j \* Z + k] + leftBorder[j \* Z + k]) / Hx2 +

(phi[i \* Y \* Z + (j + 1) \* Z + k] + phi[i \* Y \* Z + (j - 1) \* Z + k]) / Hy2 +

(phi[i \* Y \* Z + j \* Z + (k + 1)] + phi[i \* Y \* Z + j \* Z + (k - 1)]) / Hz2 -

rightFunction((i + rank \* (X - 1)) \* Hx - 1, j \* Hy - 1, k \* Hz - 1));

double deltaPhi = fabs(phiNext[i \* Y \* Z + j \* Z + k] - phi[i \* Y \* Z + j \* Z + k]);

if (deltaPhi > maxDeltaPhi)

maxDeltaPhi = deltaPhi;

}

if (rank != size - 1) {

int i = X - 1;

phiNext[i \* Y \* Z + j \* Z + k] =

factor \* ((rightBorder[j \* Z + k] + phi[(i - 1) \* Y \* Z + j \* Z + k]) / Hx2 +

(phi[i \* Y \* Z + (j + 1) \* Z + k] + phi[i \* Y \* Z + (j - 1) \* Z + k]) / Hy2 +

(phi[i \* Y \* Z + j \* Z + (k + 1)] + phi[i \* Y \* Z + j \* Z + (k - 1)]) / Hz2 -

rightFunction((i + rank \* (X - 1)) \* Hx - 1, j \* Hy - 1, k \* Hz - 1));

double deltaPhi = fabs(phiNext[i \* Y \* Z + j \* Z + k] - phi[i \* Y \* Z + j \* Z + k]);

if (deltaPhi > maxDeltaPhi)

maxDeltaPhi = deltaPhi;

}

}

}

memcpy(phi, phiNext, X \* Y \* Z \* sizeof(double));

MPI\_Allreduce(&maxDeltaPhi, &commonMaxDeltaPhi, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

}

double currMax = 0;

for (int i = 1; i < X - 1; ++i)

for (int j = 1; j < Y - 1; ++j)

for (int k = 1; k < Z - 1; ++k) {

double currDelta = fabs(

phi[i \* Y \* Z + j \* Z + k] -

realFunction((i + rank \* (X - 1)) \* Hx - 1, j \* Hy - 1, k \* Hz - 1));

if (currDelta > currMax)

currMax = currDelta;

}

double max;

MPI\_Allreduce(&currMax, &max, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

double time = end.tv\_sec-start.tv\_sec + 0.000000001\*(end.tv\_nsec-start.tv\_nsec);

double fulltime = 0;

MPI\_Allreduce(&time, &fulltime, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

if (rank == 0) {

cout << "Time taken: " << fulltime << endl;

cout << "Difference between result and real function: " << max << endl;

}

delete[] phi;

delete[] phiNext;

delete[] rightBorder;

delete[] leftBorder;

MPI\_Finalize();

return 0;

}