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, группы 22206

***Тропина Никиты Васильевича***

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

ассистент,

А.А.Ажбаков

Новосибирск 2023

# ЦЕЛИ

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

,

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. По профилю убедиться, что коммуникации происходят на фоне счета

# ОПИСАНИЕ РАБОТЫ

Для распараллеливания задачи использовалась декартова топология «линейка». Область разрезается на слои по оси Oz. Порядок действий алгоритма в каждом процессе выглядит следующим образом:

* вычисляются сеточные значения, прилегающие к границе локальной подобласти,
* запускается асинхронный обмен граничных значений,
* выполняется вычисление остальных точек подобласти,
* ожидание завершения обменов.

**Исходные данные задачи**

Исходные данные для тестирования реализации представленного метода и выполнения лабораторной работы взяты следующие:

область моделирования: [-1;1] × [-1;1] × [-1;1],

искомая функция:,

правая часть уравнения: ρ(x, y,z) = 6−a⋅ϕ(x, y,z),

параметр уравнения: a = ,

порог сходимости: ε = ,

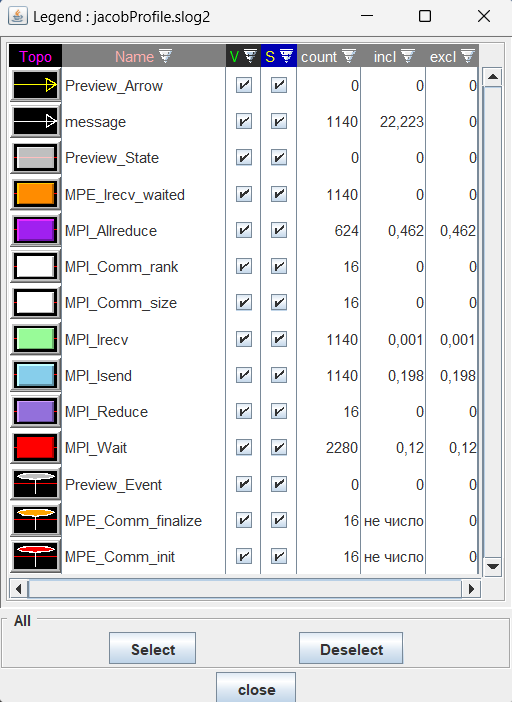
начальное приближение: .

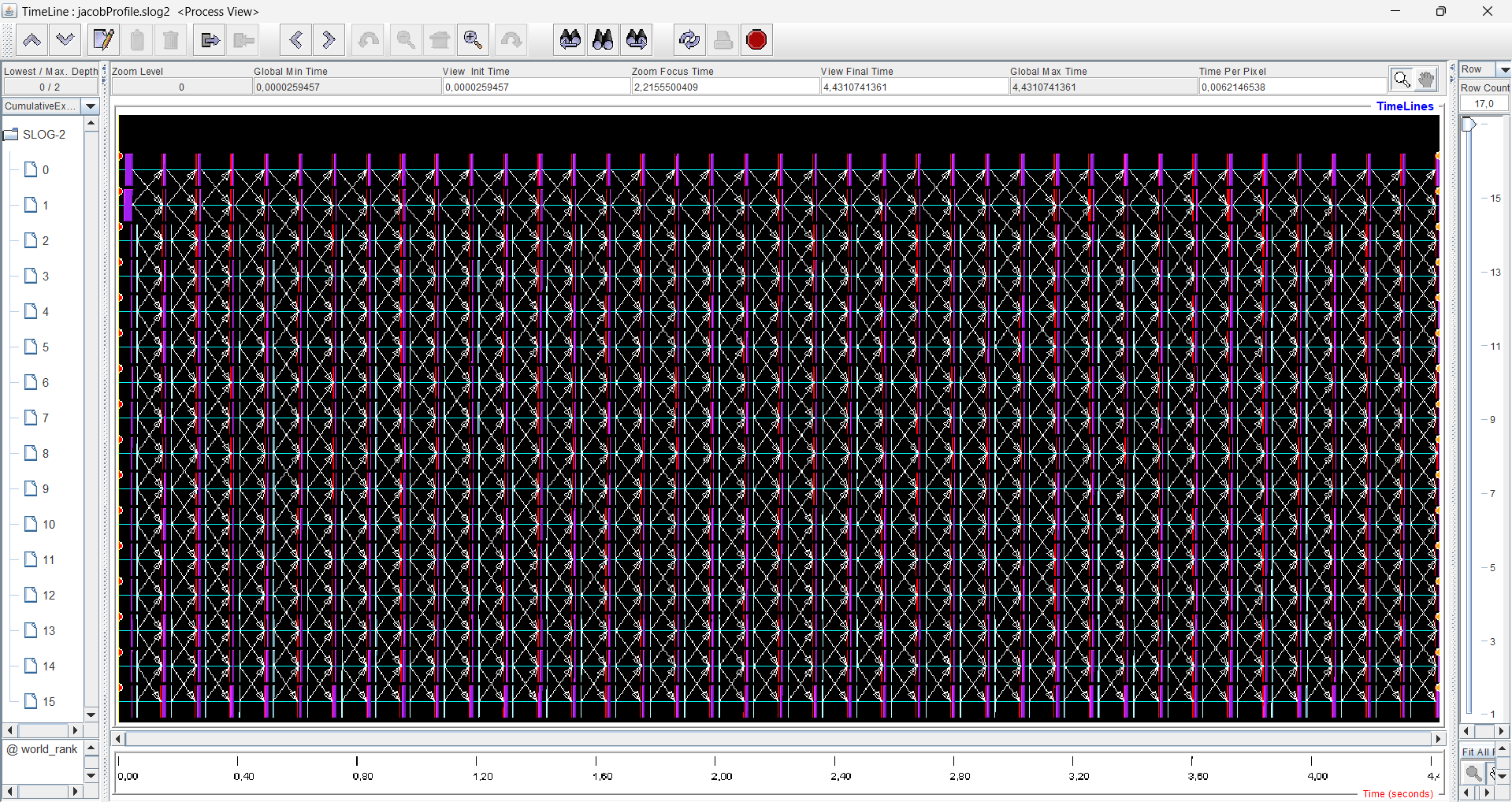
**Измерение времени**

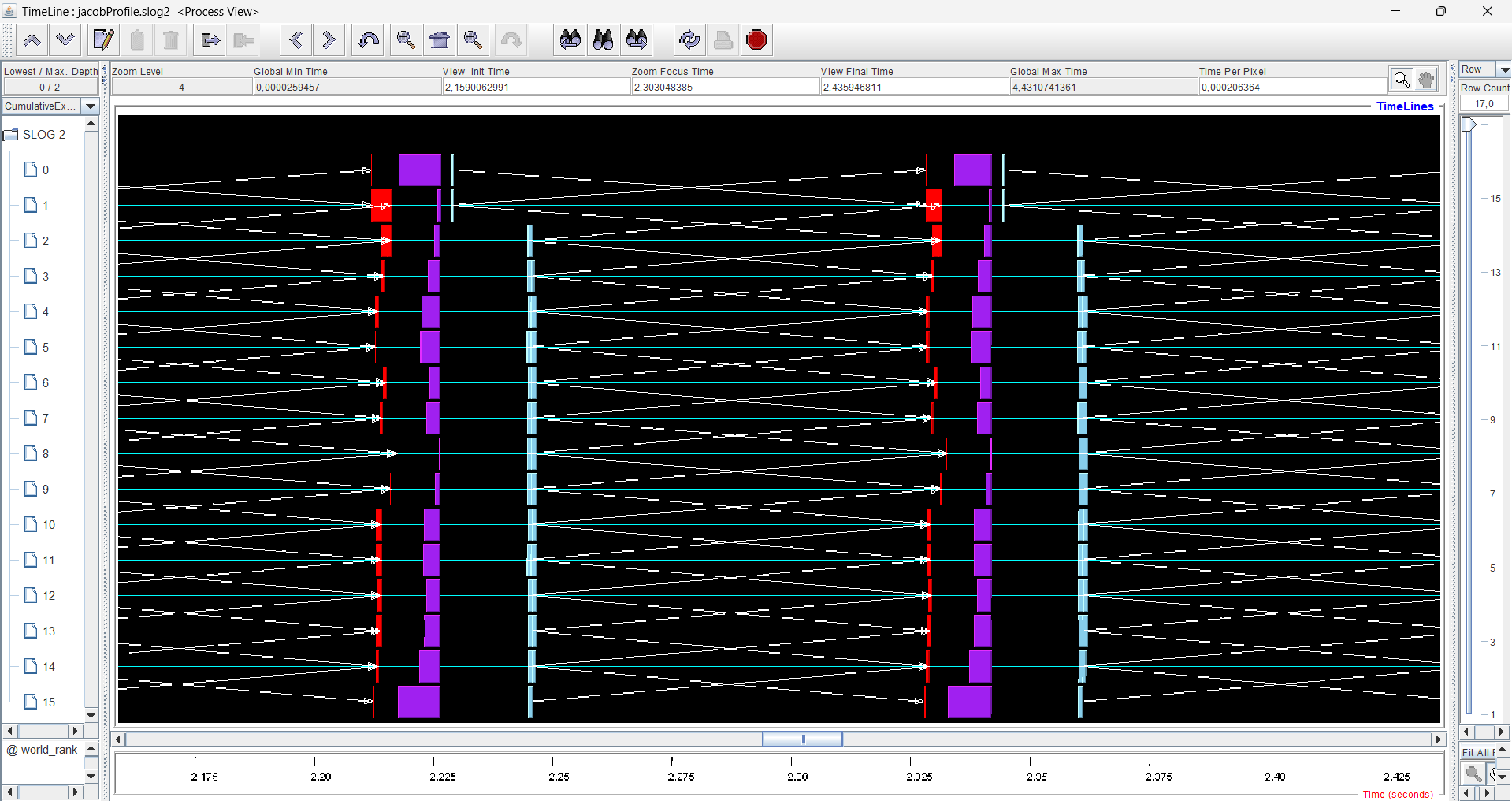
Выполнение программы производилось на вычислительном кластере НОЦ «Газпромнефть-НГУ» с помощью скрипта системы пакетной обработки SLURM (Приложение 1).   
 Для измерения времени использовалась функция MPI\_Wtime().

Результаты измерений представлены на диаграммах:

**Профилирование**



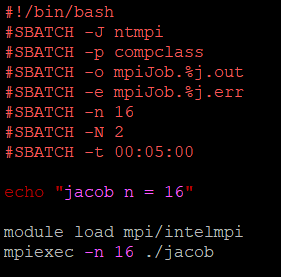




# ЗАКЛЮЧЕНИЕ

В ходе работы были построены графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых потоков. Также выполнено профилирование. Эффективность и ускорение распараллеливания программы оказались ниже, сравнивая с программами предыдущих лабораторных.

# ПРИЛОЖЕНИЕ 1



# ПРИЛОЖЕНИЕ 2

#include <iostream>

#include <cmath>

#include "mpi.h"

#define epsilon 1e-8

#define alpha 1e+5

#define Nx 160

#define Ny 160

#define Nz 160

#define Dx 2

#define Dy 2

#define Dz 2

#define X0 -1

#define Y0 -1

#define Z0 -1

double Hx = (Dx / (double)(Nx - 1));

double Hy = (Dy / (double)(Ny - 1));

double Hz = (Dz / (double)(Nz - 1));

#define factor (1.0 / ((2.0 / (Hx \* Hx)) + (2.0 / (Hy \* Hy)) + (2.0 / (Hz \* Hz)) + alpha))

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

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

}

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

return 6 - alpha \* phi(x, y, z);

}

double newCoor(double startCoor, double ort, double step) {

return startCoor + ort \* step;

}

void setFunctionValuesInRegion(double\* functionValues, double\* nextFunctionValues, int layerHeight, int rank, int nProcesses) {

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

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

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

if (i == 0 || j == 0 || i == Nx - 1 || j == Ny - 1) {

functionValues[k \* Ny \* Nx + j \* Nx + i] = phi(newCoor(X0, i, Hx), newCoor(Y0, j, Hy),

newCoor(Z0, k + layerHeight \* rank, Hz));

}

else {

functionValues[k \* Ny \* Nx + j \* Nx + i] = 0;

}

nextFunctionValues[k \* Ny \* Nx + j \* Nx + i] = functionValues[k \* Ny \* Nx + j \* Nx + i];

}

}

}

if (rank == 0) {

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

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

functionValues[j \* Nx + i] = phi(newCoor(X0, i, Hx), newCoor(Y0, j, Hy), Z0);

nextFunctionValues[j \* Nx + i] = functionValues[j \* Nx + i];

}

}

}

if (rank == nProcesses - 1) {

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

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

functionValues[(layerHeight - 1) \* Ny \* Nx + j \* Nx + i] = phi(newCoor(X0, i, Hx), newCoor(Y0, j, Hy), Z0 + Dz);

nextFunctionValues[(layerHeight - 1) \* Ny \* Nx + j \* Nx + i] = functionValues[(layerHeight - 1) \* Ny \* Nx + j \* Nx + i];

}

}

}

}

double calculateNextInnerValue(double\* functionValues, int x, int y, int z) {

double xSum = (functionValues[z \* Ny \* Nx + y \* Nx + x + 1] + functionValues[z \* Ny \* Nx + y \* Nx + x + 1]) / (Hx \* Hx);

double ySum = (functionValues[z \* Ny \* Nx + (y + 1) \* Nx + x] + functionValues[z \* Ny \* Nx + (y - 1) \* Nx + x]) / (Hy \* Hy);

double zSum = (functionValues[(z + 1) \* Ny \* Nx + y \* Nx + x] + functionValues[(z - 1) \* Ny \* Nx + y \* Nx + x]) / (Hz \* Hz);

double resultSum = xSum + ySum + zSum - roFunction(newCoor(X0, x, Hx), newCoor(Y0, y, Hy), newCoor(Z0, z, Hz));

return factor \* resultSum;

}

double calculateNextDownBoundValue(double\* functionValues, double\* downLayer, int x, int y, int layerHeight, int rank) {

double xSum = (functionValues[y \* Nx + x + 1] + functionValues[y \* Nx + x + 1]) / (Hx \* Hx);

double ySum = (functionValues[(y + 1) \* Nx + x] + functionValues[(y - 1) \* Nx + x]) / (Hy \* Hy);

double zSum = (functionValues[Ny \* Nx + y \* Nx + x] + downLayer[y \* Nx + x]) / (Hz \* Hz);

double resultSum = xSum + ySum + zSum - roFunction(newCoor(X0, x, Hx), newCoor(Y0, y, Hy), newCoor(Z0, layerHeight \* rank, Hz));

return factor \* resultSum;

}

double calculateNextTopBoundValue(double\* functionValues, double\* topLayer, int x, int y, int layerHeight, int rank) {

double xSum = (functionValues[(layerHeight - 1) \* Ny \* Nx + y \* Nx + x + 1] +

functionValues[(layerHeight - 1) \* Ny \* Nx + y \* Nx + x - 1]) / (Hx \* Hx);

double ySum = (functionValues[(layerHeight - 1) \* Ny \* Nx + (y + 1) \* Nx + x] +

functionValues[(layerHeight - 1) \* Ny \* Nx + (y - 1) \* Nx + x]) / (Hy \* Hy);

double zSum = (topLayer[y \* Nx + x] + functionValues[(layerHeight - 2) \* Ny \* Nx + y \* Nx + x]) / (Hz \* Hz);

double resultSum = xSum + ySum + zSum - roFunction(newCoor(X0, x, Hx), newCoor(Y0, y, Hy), newCoor(Z0, (layerHeight + 1) \* rank - 1, Hz));

return factor \* resultSum;

}

int isAccuracyAchieved(int accuracy) {

int isAchieved;

MPI\_Allreduce(&accuracy, &isAchieved, 1, MPI\_INTEGER, MPI\_SUM, MPI\_COMM\_WORLD);

return isAchieved;

}

double computeFunction(double\* functionValues, double\* nextFunctionValues, int layerHeight, int rank, int nProcesses) {

MPI\_Request requests[4];

double\* downLayer = (double\*)malloc(sizeof(double) \* Nx \* Ny);

double\* topLayer = (double\*)malloc(sizeof(double) \* Nx \* Ny);

int accuracy = 1;

double maxDifferenceTmp = 0;

while (isAccuracyAchieved(accuracy)) {

accuracy = 0;

maxDifferenceTmp = 0;

memcpy(functionValues, nextFunctionValues, sizeof(double) \* Nx \* Ny \* layerHeight);

if (rank != 0) {

MPI\_Isend(&functionValues[0], Nx \* Ny, MPI\_DOUBLE, rank - 1, 1, MPI\_COMM\_WORLD, &requests[0]);

MPI\_Irecv(downLayer, Nx \* Ny, MPI\_DOUBLE, rank - 1, 2, MPI\_COMM\_WORLD, &requests[1]);

}

if (rank != nProcesses - 1) {

MPI\_Isend(&functionValues[(layerHeight - 1) \* Nx \* Ny], Nx \* Ny, MPI\_DOUBLE, rank + 1, 2, MPI\_COMM\_WORLD, &requests[2]);

MPI\_Irecv(topLayer, Nx \* Ny, MPI\_DOUBLE, rank + 1, 1, MPI\_COMM\_WORLD, &requests[3]);

}

for (int z = 1; z < layerHeight - 1; z++) {

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

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

nextFunctionValues[z \* Ny \* Nx + y \* Ny + x] = calculateNextInnerValue(functionValues, x, y, z);

double difference = fabs(nextFunctionValues[z \* Ny \* Nx + y \* Ny + x] - functionValues[z \* Ny \* Nx + y \* Ny + x]);

if (difference > epsilon) {

accuracy = 1;

}

maxDifferenceTmp = difference > maxDifferenceTmp ? difference : maxDifferenceTmp;

}

}

}

if (rank != 0) {

MPI\_Wait(&requests[0], MPI\_STATUS\_IGNORE);

MPI\_Wait(&requests[1], MPI\_STATUS\_IGNORE);

}

if (rank != nProcesses - 1) {

MPI\_Wait(&requests[2], MPI\_STATUS\_IGNORE);

MPI\_Wait(&requests[3], MPI\_STATUS\_IGNORE);

}

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

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

if (rank != 0) {

nextFunctionValues[y \* Nx + x] = calculateNextDownBoundValue(functionValues, downLayer, x, y, layerHeight, rank);

}

if (rank != nProcesses - 1) {

nextFunctionValues[(layerHeight - 1) \* Ny \* Nx + y \* Nx + x] = calculateNextTopBoundValue(functionValues, topLayer,

x, y, layerHeight, rank);

}

double difference = fabs(nextFunctionValues[Nx \* y + x] - functionValues[Nx \* y + x]);

if (difference > epsilon) {

accuracy = 1;

}

maxDifferenceTmp = difference > maxDifferenceTmp ? difference : maxDifferenceTmp;

}

}

}

free(downLayer);

free(topLayer);

double maxDifference = 0;

MPI\_Reduce(&maxDifferenceTmp, &maxDifference, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

return maxDifference;

}

double\* jacobiMethod(int rank, int nProcesses) {

int layerHeight = Nz / nProcesses;

double\* functionValues = (double\*)malloc(sizeof(double) \* Nx \* Ny \* layerHeight);

double\* finalFunctionValues = (double\*)malloc(sizeof(double) \* Nx \* Ny \* layerHeight);

setFunctionValuesInRegion(functionValues, finalFunctionValues, layerHeight, rank, nProcesses);

double maxDifference = computeFunction(functionValues, finalFunctionValues, layerHeight, rank, nProcesses);

//printf("%f", maxDifference);

free(functionValues);

return finalFunctionValues;

}

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

{

int rank;

int nProcesses;

MPI\_Init(&argc, &argv);

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

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

double start = MPI\_Wtime();

double\* functionValues = jacobiMethod(rank, nProcesses);

double finish = MPI\_Wtime();

if (rank == 0) {

printf("TIME: %f", finish - start);

}

MPI\_Finalize();

return 0;

}