Министерство образования и науки Российской Федерации

Новосибирский национальный исследовательский государственный университет

Основы параллельного программирования

Отчет по лабораторной работе № 4

Студент: Ланин Даниил Михайлович

Преподаватель: Артюхов Алексей Андреевич

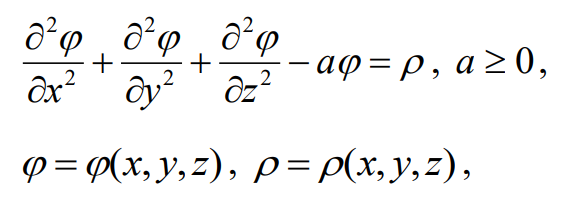
Новосибирск, 2023 г.

1. **Цель работы**

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

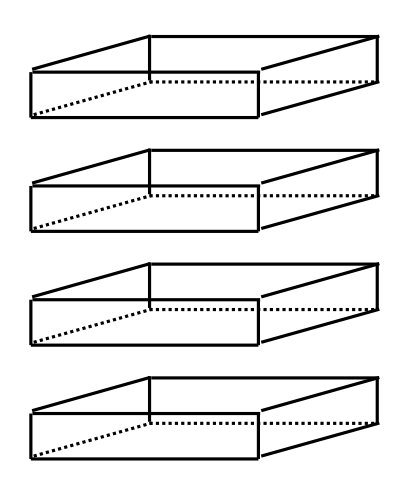
1. **Краткое описание подхода к организации решения прикладной задачи параллельными взаимодействующими процессами**

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

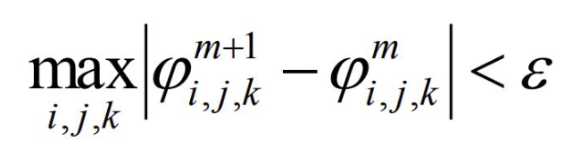


в области Ω с краевыми условиями 1-го рода (т.е. на границе G известны значения искомой функции φ).

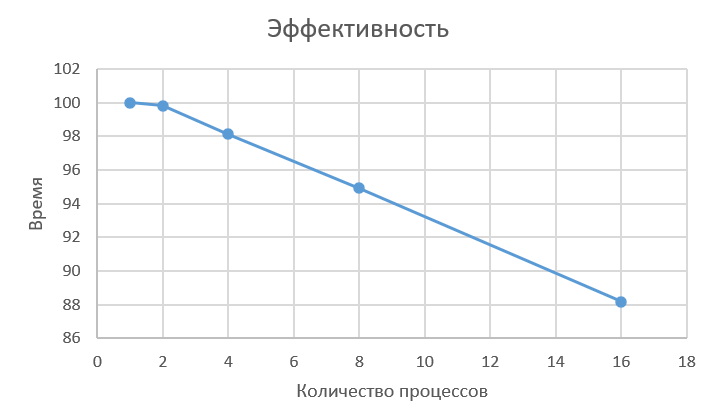
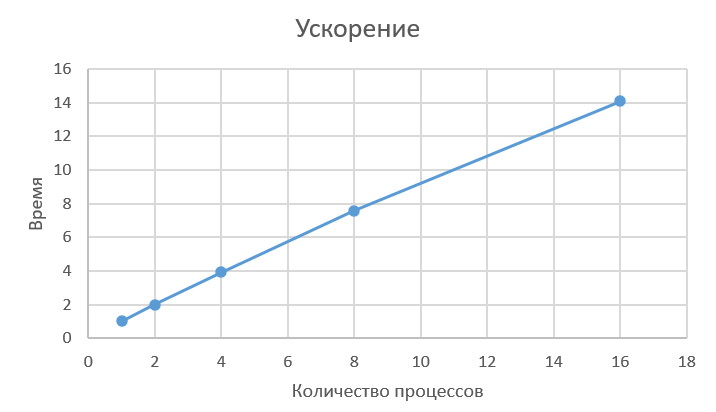
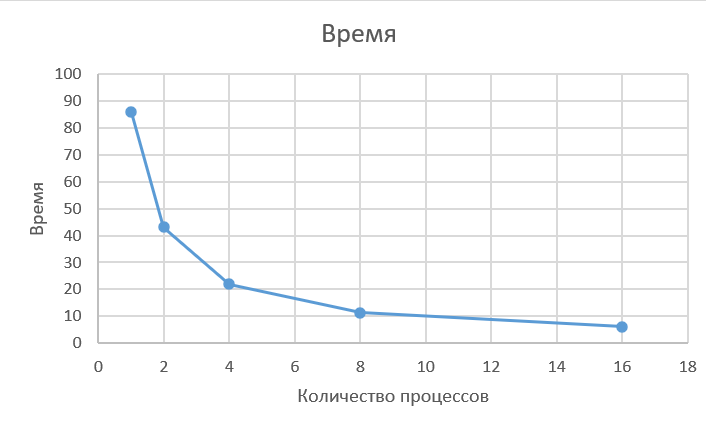
Для решения использовать итерационный метод Якоби.  
Для расспаралеливания алгоритма, использовал декомпозицию области Ω “На линейке”



Окончание алгоритма является критерий сходимости:

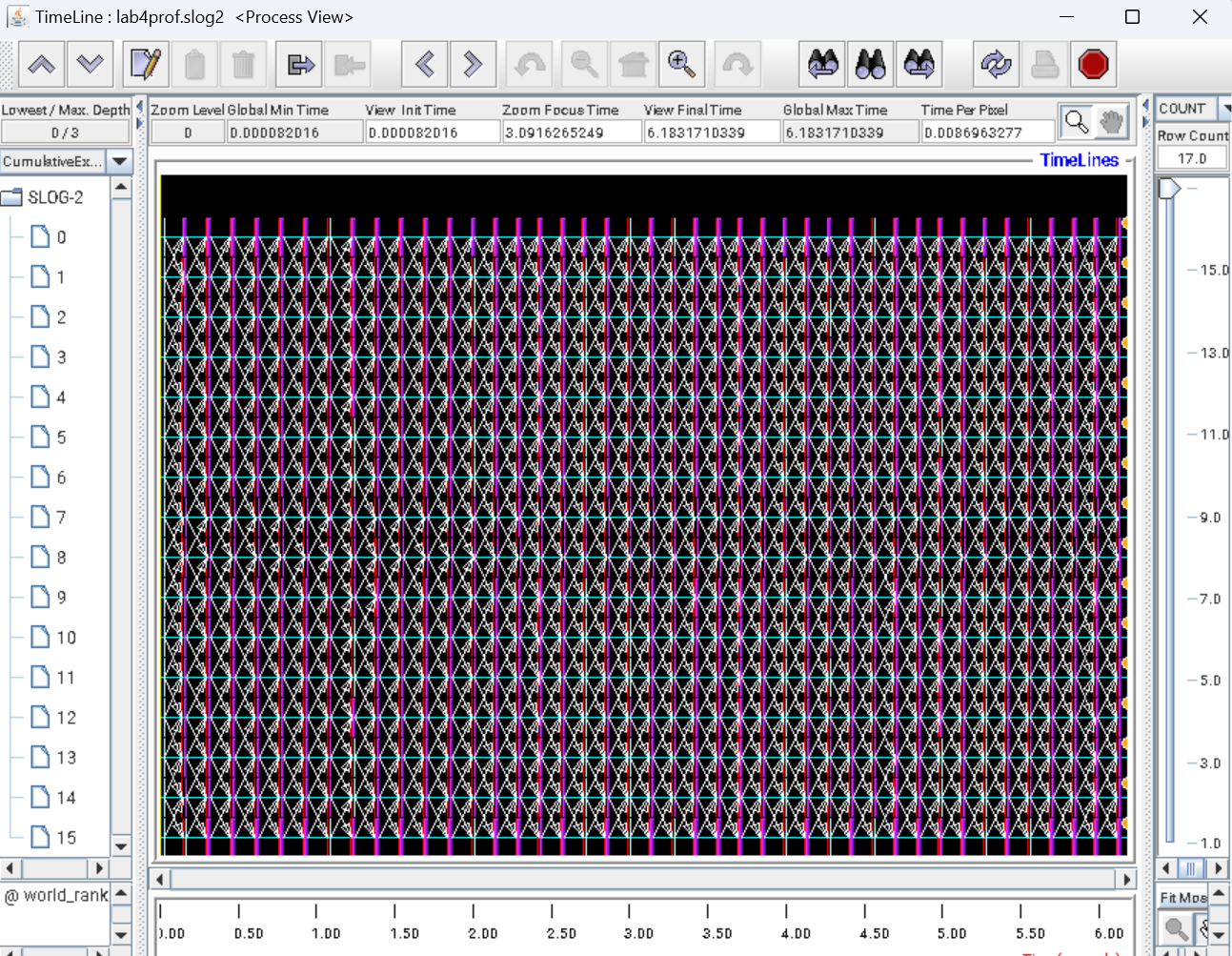


1. **Исследование производительности програмы**

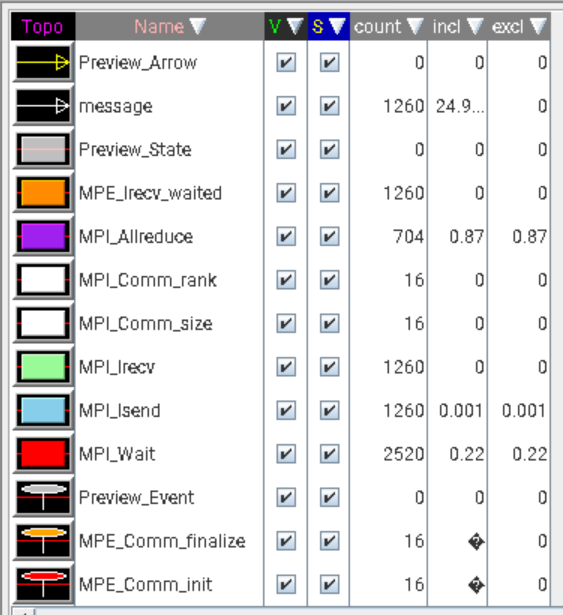
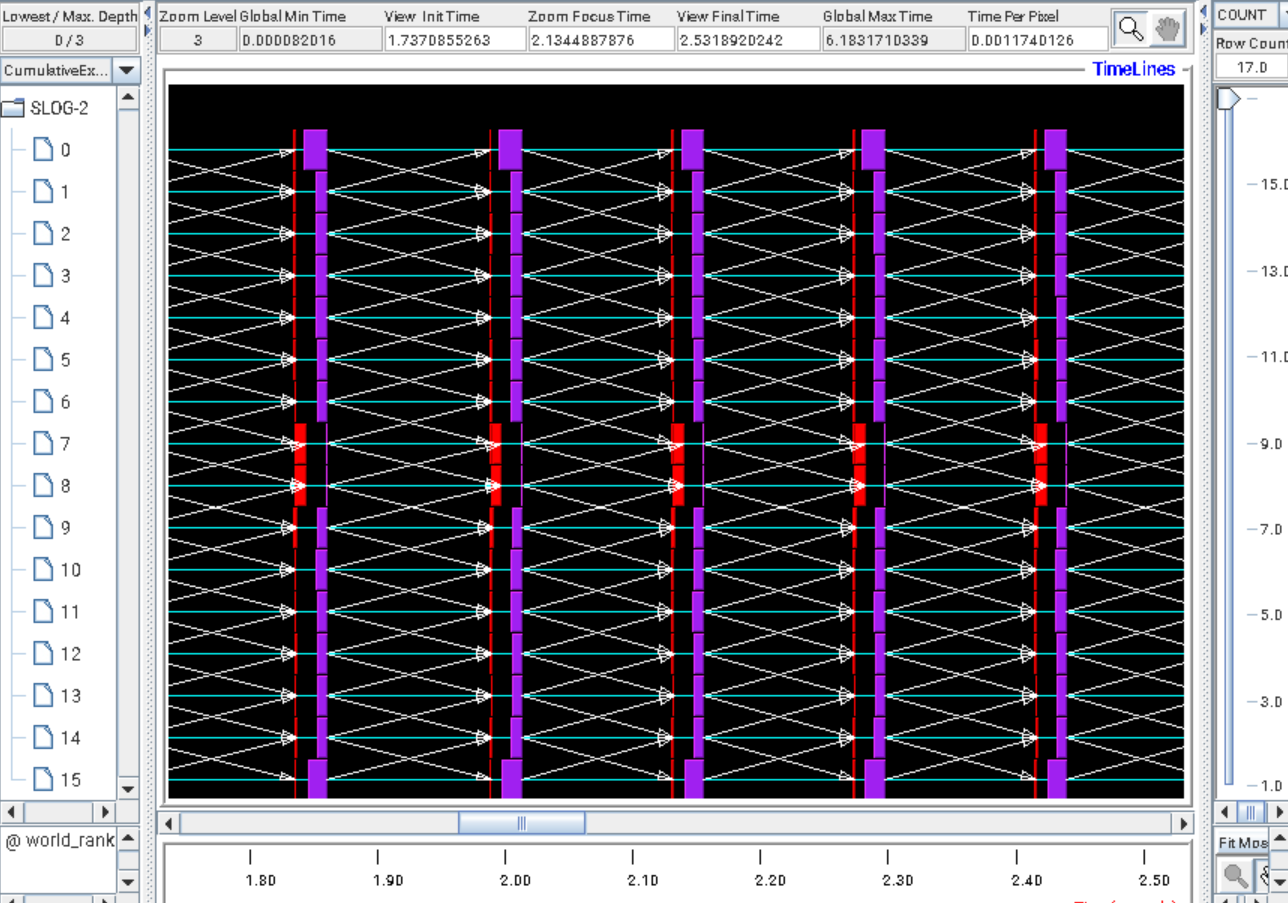


1. **Профилирование**

Общий вид профилирования:



Профилирование в приближённом виде:



1. **Заключение**

В ходе лабораторной работы я освоил метод распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трёхмерной области посредством MPI.

Критической частью работы было освоение неблокирующих операций обмена данными в MPI - MPI\_Isend и MPI\_Irecv. Эти операции позволяют начать передачу или получение данных и продолжить выполнение программы без ожидания завершения операции передачи данных. Важно отметить, что данные могут быть переданы или получены в любой момент после вызова MPI\_Isend или MPI\_Irecv, корректность данных гарантирована только после вызова функций MPI\_Wait или MPI\_Test, которые проверяют завершение соответствующей операции.

Этот подход требует внимательного проектирования и управления потоками выполнения, но позволяет добиться значительного улучшения производительности для больших численных расчетов.

1. **Исходники**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <mpi.h>

#define eps 1e-9

#define a 1e5

#define X0 -1

#define Y0 -1

#define Z0 -1

#define Dx 2.0

#define Dy 2.0

#define Dz 2.0

#define Nx 400

#define Ny 350

#define Nz 300

#define hx (Dx / (Nx - 1.0))

#define hy (Dy / (Ny - 1.0))

#define hz (Dz / (Nz - 1.0))

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 - a \* phi(x, y, z);

}

double JacobiMethod(int rank, int layerHeight, double\* prevPhi, int x, int y, int z) {

    double xComp = (prevPhi[Nx \* Ny \* z + Nx \* y + (x - 1)] + prevPhi[Nx \* Ny \* z + Nx \* y + (x + 1)]) / (hx \* hx);

    double yComp = (prevPhi[Nx \* Ny \* z + Nx \* (y - 1) + x] + prevPhi[Nx \* Ny \* z + Nx \* (y + 1) + x]) / (hy \* hy);

    double zComp = (prevPhi[Nx \* Ny \* (z - 1) + Nx \* y + x] + prevPhi[Nx \* Ny \* (z + 1) + Nx \* y + x]) / (hz \* hz);

    double mult = 1.0 / (2.0 / (hx \* hx) + 2.0 / (hy \* hy) + 2.0 / (hz \* hz) + a);

    return mult \* (xComp + yComp + zComp - rho(X0 + x \* hx, Y0 + y \* hy, Z0 + (z + layerHeight \* rank) \* hz));

}

double JacobiMethodTopEdge(int rank, int layerHeight, double\* prevPhi, int x, int y, const double\* upLayer) {

    double xComp = (prevPhi[Nx \* Ny \* (layerHeight - 1) + Nx \* y + (x - 1)] + prevPhi[Nx \* Ny \* (layerHeight - 1) + Nx \* y + (x + 1)]) / (hx \* hx);

    double yComp = (prevPhi[Nx \* Ny \* (layerHeight - 1) + Nx \* (y - 1) + x] + prevPhi[Nx \* Ny \* (layerHeight - 1) + Nx \* (y + 1) + x]) / (hy \* hy);

    double zComp = (prevPhi[Nx \* Ny \* (layerHeight - 2) + Nx \* y + x] + upLayer[Nx \* y + x]) / (hz \* hz);

    double mult = 1.0 / (2.0 / (hx \* hx) + 2.0 / (hy \* hy) + 2.0 / (hz \* hz) + a);

    return mult \* (xComp + yComp + zComp - rho(X0 + x \* hx, Y0 + y \* hy, Z0 + ((layerHeight - 1) + layerHeight \* rank) \* hz));

}

double JacobiMethodBottomEdge(int rank, int layerHeight, double\* prevPhi, int y, int x, const double\* downLayer) {

    double xComp = (prevPhi[Nx \* y + (x - 1)] + prevPhi[Nx \* y + (x + 1)]) / (hx \* hx);

    double yComp = (prevPhi[Nx \* (y - 1) + x] + prevPhi[Nx \* (y + 1) + x]) / (hy \* hy);

    double zComp = (downLayer[Nx \* y + x] + prevPhi[Nx \* Ny + Nx \* y + x]) / (hz \* hz);

    double mult = 1.0 / (2.0 / (hx \* hx) + 2.0 / (hy \* hy) + 2.0 / (hz \* hz) + a);

    return mult \* (xComp + yComp + zComp - rho(X0 + x \* hx, Y0 + y \* hy, Z0 + (layerHeight \* rank) \* hz));

}

void CalculateCenter(double layerHeight, double\* prevPhi, double\* Phi, int rank, char\* flag) {

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

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

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

                Phi[Nx \* Ny \* z + Nx \* y + x] = JacobiMethod(rank, layerHeight, prevPhi, x, y, z);

                if (fabs(Phi[Nx \* Ny \* z + Nx \* y + x] - prevPhi[Nx \* Ny \* z + Nx \* y + x]) > eps) {

                    (\*flag) = 0;

                }

            }

        }

    }

}

void CalculateEdges(int layerHeight, double\* prevPhi, double\* Phi, int rank, char\* flag, const double\* downLayer, const double\* upLayer, int size) {

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

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

            if (rank != 0) {

                Phi[Nx \* y + x] = JacobiMethodBottomEdge(rank, layerHeight, prevPhi, x, y, downLayer);

            }

            if (rank != size - 1) {

                Phi[Nx \* Ny \* (layerHeight - 1) + Nx \* y + x] = JacobiMethodTopEdge(rank, layerHeight, prevPhi, x, y, upLayer);

            }

            if (fabs(Phi[Nx \* y + x] - prevPhi[Nx \* y + x]) > eps) {

                (\*flag) = 0;

            }

        }

    }

}

void CalculateMaxDifference(int rank, int layerHeight, double\* Phi) {

    double max = 0;

    double diff = 0;

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

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

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

                diff = fabs(Phi[z \* Nx \* Ny + y \* Nx + x] - phi(X0 + x \* hx, Y0 + y \* hy, Z0 + (z + layerHeight \* rank) \* hz));

                if (diff > max) {

                    max = diff;

                }

            }

        }

    }

    double tmp = 0;

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

    max = tmp;

    if (rank == 0) {

        printf("Max difference: %lf\n", max);

    }

}

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

    int rank, size;

    double timeStart, timeFinish;

    MPI\_Init(&argc, &argv);

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

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

    int layerHeight = Nz / size;

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

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

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

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

    if (rank == 0) timeStart = MPI\_Wtime();

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

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

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

                if (y == 0 || x == 0 || y == Ny - 1 || x == Nx - 1) {

                    Phi[Nx \* Ny \* z + Nx \* y + x] = phi(X0 + x \* hx, Y0 + y \* hy, Z0 + (z + layerHeight \* rank) \* hz);

                    prevPhi[Nx \* Ny \* z + Nx \* y + x] = phi(X0 + x \* hx, Y0 + y \* hy, Z0 + (z + layerHeight \* rank) \* hz);

                }

                else {

                    Phi[Nx \* Ny \* z + Nx \* y + x] = 0;

                    prevPhi[Nx \* Ny \* z + Nx \* y + x] = 0;

                }

            }

        }

    }

    if (rank == 0) {

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

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

                Phi[0 + Nx \* y + x] = phi(X0 + x \* hx, Y0 + y \* hy, Z0);

                prevPhi[0 + Nx \* y + x] = phi(X0 + x \* hx, Y0 + y \* hy, Z0);

            }

        }

    }

    if (rank == size - 1) {

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

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

                Phi[Nx \* Ny \* (layerHeight - 1) + Nx \* y + x] = phi(X0 + x \* hx, Y0 + y \* hy, Z0 + Dz);

                prevPhi[Nx \* Ny \* (layerHeight - 1) + Nx \* y + x] = phi(X0 + x \* hx, Y0 + y \* hy, Z0 + Dz);

            }

        }

    }

    double\* tmp;

    int counter = 0;

    MPI\_Request requests[4];

    char isDiverged = 1;

    do {

        isDiverged = 1;

        tmp = prevPhi;

        prevPhi = Phi;

        Phi = tmp;

        if (rank != 0) {

            MPI\_Isend(&prevPhi[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 != size - 1) {

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

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

        }

        CalculateCenter(layerHeight, prevPhi, Phi, rank, &isDiverged);

        if (rank != 0) {

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

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

        }

        if (rank != size - 1) {

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

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

        }

        CalculateEdges(layerHeight, prevPhi, Phi, rank, &isDiverged, downLayer, upLayer, size);

        char tmpFlag;

        MPI\_Allreduce(&isDiverged, &tmpFlag, 1, MPI\_CHAR, MPI\_LAND, MPI\_COMM\_WORLD);

        isDiverged = tmpFlag;

        counter++;

    } while (!isDiverged);

    if (rank == 0) timeFinish = MPI\_Wtime();

    int tmpCounter;

    MPI\_Allreduce(&counter, &tmpCounter, 1, MPI\_INT, MPI\_MAX, MPI\_COMM\_WORLD);

    counter = tmpCounter;

    CalculateMaxDifference(rank, layerHeight, Phi);

    if (rank == 0) {

        printf("Number of iterations: %d\n", counter);

        printf("Time: %lf\n", (timeFinish - timeStart));

    }

    MPI\_Finalize();

    return 0;

}