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, 18209 группы

Большим Максима Антонович

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Матвеев А.С.

Новосибирск 2020

СОДЕРЖАНИЕ

ЦЕЛЬ **3**

ЗАДАНИЕ **3**

КОД ПРОГРАММЫ **3**

ГРАФИК ВРЕМЕНИ РАБОТЫ **12**

ЦЕЛЬ

Освоить метод распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области посредством MPI, используя асинхронную пересылку сообщений.

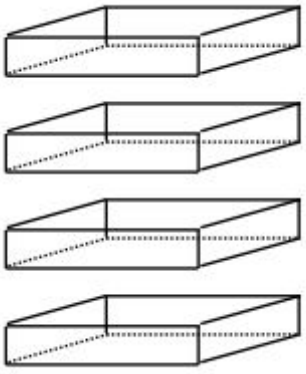
ЗАДАНИЕ

Реализовать в дискретном аналоге решение дифференциального уравнения вида:

$$\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}+\frac{\partial^2\varphi}{\partial z^2}-a\varphi=\rho\,,\ a\geq 0\,,$$

$$\varphi=\varphi(x,y,z)\,,\ \rho=\rho(x,y,z)\,,$$

в области Ω с краевыми условиями 1-го рода. Для решения использовать итерационный метод Якоби. Распараллелить алгоритм на регулярных сетках методом декомпозиции области Ω “на линейке”.



Окончанием алгоритма является критерий сходимости:

$$\max_{i,j,k}\left|\varphi_{i,j,k}^{m+1}-\varphi_{i,j,k}^{m}\right|<\varepsilon$$

КОД ПРОГРАММЫ

```
#include <cstdio>
#include <cmath>
#include <mpi.h>

#define in 60
#define jn 60
#define kn 60
#define a 1

double *(function[2]);
double *(buffer_layer[2]);
MPI_Request send_req[2];
MPI_Request recv_req[2];

double Fi;
double Fj;
double Fk;
double X = 2.0;
double Y = 2.0;
double Z = 2.0;
double e = 10e-8;
double constant;

int layer0 = 1;
int layer1 = 0;

double phi(double x, double y, double z);
double rho(double x, double y, double z);
```

```
void fill_layers(const int *perThreads, const int *offsets, int
threadRank, int J, int K, double hx, double hy, double hz);

void send_data(int J, int K, int threadRank, int threadCount,
const int *perThreads);

void receive_data(int threadRank, int threadCount);

void calc_center(int &f, int J, int K, double hx, double hy,
double hz, int threadRank, const int *perThreads, const int
*offsets, double owx, double owy, double owz);

void calc_edges(int &f, int J, int K, double hx, double hy,
double hz, int threadRank, int threadCount, const int
*perThreads, const int *offsets, double owx, double owy, double
owz);

void find_max_diff(int J, int K, double hx, double hy, double
hz, int threadRank, const int *perThreads, const int *offsets);


int main(int argc, char **argv)

{

    MPI_Init(&argc, &argv);


    int threadRank;

    int threadCount;

    int F, tmpF;


    MPI_Comm_size(MPI_COMM_WORLD, &threadCount);

    MPI_Comm_rank(MPI_COMM_WORLD, &threadRank);


    if (threadRank == 0)

        printf("Thread count: %d\n", threadCount);


    int *perThreads = new int[threadCount]();

    int *offsets = new int[threadCount]();


    for (int i = 0, height = kn + 1, tmp = threadCount - (height
% threadCount), currentLine = 0; i < threadCount; ++i)
```

```
    {
        offsets[i] = currentLine;
        perThreads[i] = i < tmp ? (height / threadCount) :
(height / threadCount + 1);
        currentLine += perThreads[i];
    }

    int I = perThreads[threadRank];
    int J = (jn + 1);
    int K = (kn + 1);

    function[0] = new double[I * J * K]();
    function[1] = new double[I * J * K]();

    buffer_layer[0] = new double[K * J]();
    buffer_layer[1] = new double[K * J]();

    double hx = X / in;
    double hy = Y / jn;
    double hz = Z / kn;

    double owx = pow(hx, 2);
    double owy = pow(hy, 2);
    double owz = pow(hz, 2);
    constant = 2 / owx + 2 / owy + 2 / owz + a;

    fill_layers(perThreads, offsets, threadRank, J, K, hx, hy,
hz);

    double start = MPI_Wtime();
```

```
    do
    {
        F = 1;
        //layer swap
        layer0 = 1 - layer0;
        layer1 = 1 - layer1;

        send_data(J, K, threadRank, threadCount, perThreads);
        calc_center(F, J, K, hx, hy, hz, threadRank, perThreads,
offsets, owx, owy, owz);
        receive_data(threadRank, threadCount);

        calc_edges(F, J, K, hx, hy, hz, threadRank, threadCount,
perThreads, offsets, owx, owy, owz);

        MPI_Allreduce(&F, &tmpF, 1, MPI_INT, MPI_MIN,
MPI_COMM_WORLD);
        F = tmpF;
    } while (F == 0);

    double finish = MPI_Wtime();

    if (threadRank == 0)
        printf("Time: %lf\n", finish - start);

    find_max_diff(J, K, hx, hy, hz, threadRank, perThreads,
offsets);

    delete[] buffer_layer[0];
    delete[] buffer_layer[1];
```

```
    delete[] function[0];
    delete[] function[1];
    delete[] offsets;
    delete[] perThreads;

    MPI_Finalize();
    return 0;
}

double rho(double x, double y, double z)
{
    return 6 - a * phi(x, y, z);
}

double phi(double x, double y, double z)
{
    return x * x + y * y + z * z;
}



void calc_edges(int &f, int J, int K, double hx, double hy,
double hz, int threadRank, int threadCount, const int
*perThreads, const int *offsets, double owx, double owy, double
owz)
{
    for (int j = 1; j < jn; ++j)
        for (int k = 1; k < kn; ++k)
        {

            if (threadRank != 0)
            {
```

```
                int i = 0;

                Fi = (function[layer0][(i + 1) * J * K + j * K +
k] + buffer_layer[0][j * K + k]) / owx;

                Fj = (function[layer0][i * J * K + (j + 1) * K +
k] + function[layer0][i * J * K + (j - 1) * K + k]) / owy;

                Fk = (function[layer0][i * J * K + j * K + (k +
1)] + function[layer0][i * J * K + j * K + (k - 1)]) / owz;

                function[layer1][i * J * K + j * K + k] =

                        (Fi + Fj + Fk - rho((i +
offsets[threadRank]) * hx, j * hy, k * hz)) / constant;


                if (fabs(function[layer1][i * J * K + j * K + k]
- phi((i + offsets[threadRank]) * hx, j * hy, k * hz)) > e)

                    f = 0;

            }


            if (threadRank != threadCount - 1)

            {

                int i = perThreads[threadRank] - 1;

                Fi = (buffer_layer[1][j * K + k] +
function[layer0][(i - 1) * J * K + j * K + k]) / owx;

                Fj = (function[layer0][i * J * K + (j + 1) * K +
k] + function[layer0][i * J * K + (j - 1) * K + k]) / owy;

                Fk = (function[layer0][i * J * K + j * K + (k +
1)] + function[layer0][i * J * K + j * K + (k - 1)]) / owz;

                function[layer1][i * J * K + j * K + k] =

                        (Fi + Fj + Fk - rho((i +
offsets[threadRank]) * hx, j * hy, k * hz)) / constant;


                if (fabs(function[layer1][i * J * K + j * K + k]
- phi((i + offsets[threadRank]) * hx, j * hy, k * hz)) > e)

                    f = 0;

            }
```

```
        }
}


void fill_layers(const int *perThreads, const int *offsets, int
threadRank, int J, int K, double hx, double hy, double hz)
{
    for (int i = 0, startLine = offsets[threadRank]; i <=
perThreads[threadRank] - 1; i++, startLine++)
        for (int j = 0; j <= jn; j++)
            for (int k = 0; k <= kn; k++)
            {
                if ((startLine != 0) && (j != 0) && (k != 0) &&
(startLine != in) && (j != jn) && (k != kn))
                {
                    function[0][i * J * K + j * K + k] = 0;
                    function[1][i * J * K + j * K + k] = 0;
                } else
                {
                    function[0][i * J * K + j * K + k] =
phi(startLine * hx, j * hy, k * hz);
                    function[1][i * J * K + j * K + k] =
phi(startLine * hx, j * hy, k * hz);
                }
            }
}


void send_data(int J, int K, int threadRank, int threadCount,
const int *perThreads)
{
    if (threadRank != 0)
    {
        MPI_Isend(&(function[layer0][0]), K * J, MPI_DOUBLE,
threadRank - 1, 0, MPI_COMM_WORLD, &send_req[0]);
```

```
        MPI_Irecv(buffer_layer[0], K * J, MPI_DOUBLE, threadRank
- 1, 1, MPI_COMM_WORLD, &recv_req[1]);

    }

    if (threadRank != threadCount - 1)

    {

        MPI_Isend(&(function[layer0][(perThreads[threadRank] -
1) * J * K]), K * J, MPI_DOUBLE, threadRank + 1, 1,

                  MPI_COMM_WORLD, &send_req[1]);

        MPI_Irecv(buffer_layer[1], K * J, MPI_DOUBLE, threadRank
+ 1, 0, MPI_COMM_WORLD, &recv_req[0]);

    }

}


void receive_data(int threadRank, int threadCount)

{

    if (threadRank != 0)

    {

        MPI_Wait(&recv_req[1], MPI_STATUS_IGNORE);

        MPI_Wait(&send_req[0], MPI_STATUS_IGNORE);

    }

    if (threadRank != threadCount - 1)

    {

        MPI_Wait(&recv_req[0], MPI_STATUS_IGNORE);

        MPI_Wait(&send_req[1], MPI_STATUS_IGNORE);

    }

}


void calc_center(int &f, int J, int K, double hx, double hy,
double hz, int threadRank, const int *perThreads, const int
*offsets, double owx, double owy, double owz)

{

    for (int i = 1; i < perThreads[threadRank] - 1; ++i)
```

```
        for (int j = 1; j < jn; ++j)
            for (int k = 1; k < kn; ++k)
            {
                Fi = (function[layer0][(i + 1) * J * K + j * K +
k] + function[layer0][(i - 1) * J * K + j * K + k]) / owx;
                Fj = (function[layer0][i * J * K + (j + 1) * K +
k] + function[layer0][i * J * K + (j - 1) * K + k]) / owy;
                Fk = (function[layer0][i * J * K + j * K + (k +
1)] + function[layer0][i * J * K + j * K + (k - 1)]) / owz;
                function[layer1][i * J * K + j * K + k] =
                        (Fi + Fj + Fk - rho((i +
offsets[threadRank]) * hx, j * hy, k * hz)) / constant;

                if (fabs(function[layer1][i * J * K + j * K + k]
- phi((i + offsets[threadRank]) * hx, j * hy, k * hz)) > e)
                    f = 0;

            }
}

void find_max_diff(int J, int K, double hx, double hy, double
hz, int threadRank, const int *perThreads, const int *offsets)
{
    double max = 0;
    double F1 = 0;

    for (int i = 1; i < perThreads[threadRank] - 2; i++)
        for (int j = 1; j < jn; j++)
            for (int k = 1; k < kn; k++)
                if ((F1 = fabs(function[layer1][i * J * K + j *
K + k] - phi((i + offsets[threadRank]) * hx, j * hy, k * hz))) >
max)
                    max = F1;
```

```
    double tmpMax = 0;

    MPI_Allreduce(&max, &tmpMax, 1, MPI_DOUBLE, MPI_MAX,
MPI_COMM_WORLD);


    if (threadRank == 0)

    {

        max = tmpMax;

        printf("Max diff = %.9lf\n", max);

    }

}
```

ГРАФИК ВРЕМЕНИ РАБОТЫ

Проведём тесты решения уравнения

$$\varphi(x,y,z) = x^2 + y^2 + z^2$$

при параметрах $a = 1$, $\varepsilon = 10^{-8}$, и правой частью

$$\rho(x,y,z) = 6 - a \cdot \varphi(x,y,z)$$

Вывод

В ходе лабораторной работы я освоил метод распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области посредством MPI. Программа реализована и выдаёт стабильный результат.