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

«УМНОЖЕНИЕ МАТРИЦЫ НА МАТРИЦУ В MPI 2D РЕШЕТКА»

студента 2 курса, группы 19212

Хомченко Станислава Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Ажбаков Артём Альбертович

ЦЕЛЬ

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

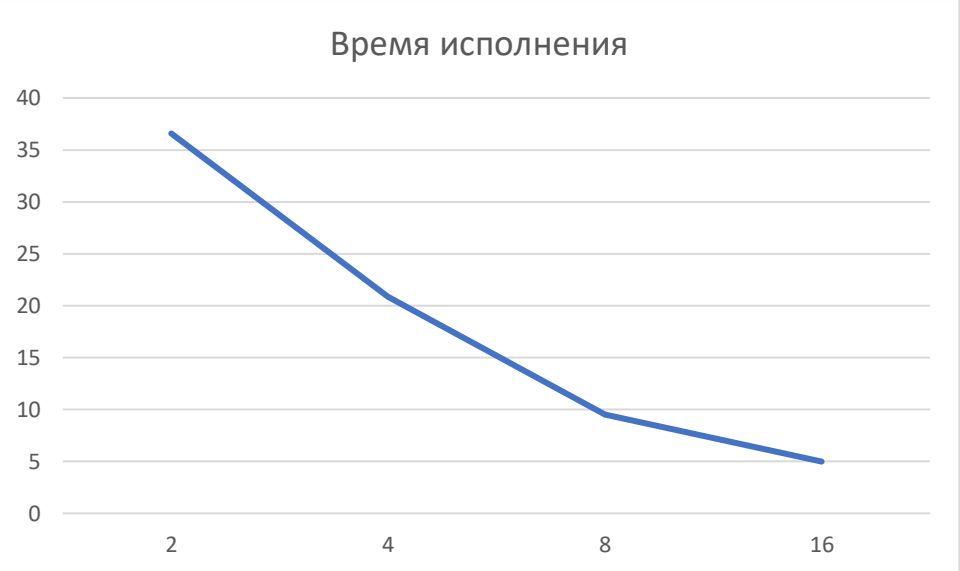
ЗАДАНИЕ

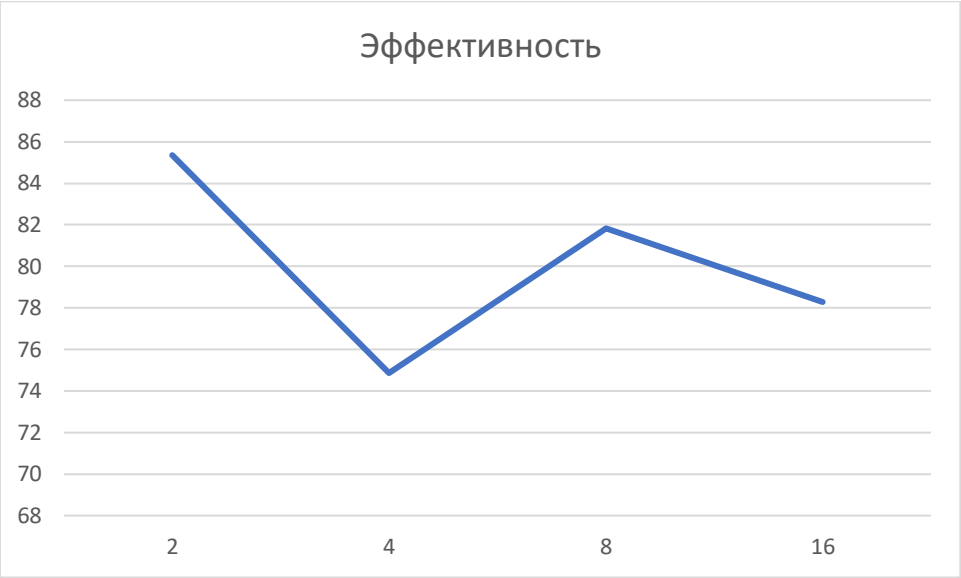
- Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.
- Измерить время работы 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 РАБОТЫ

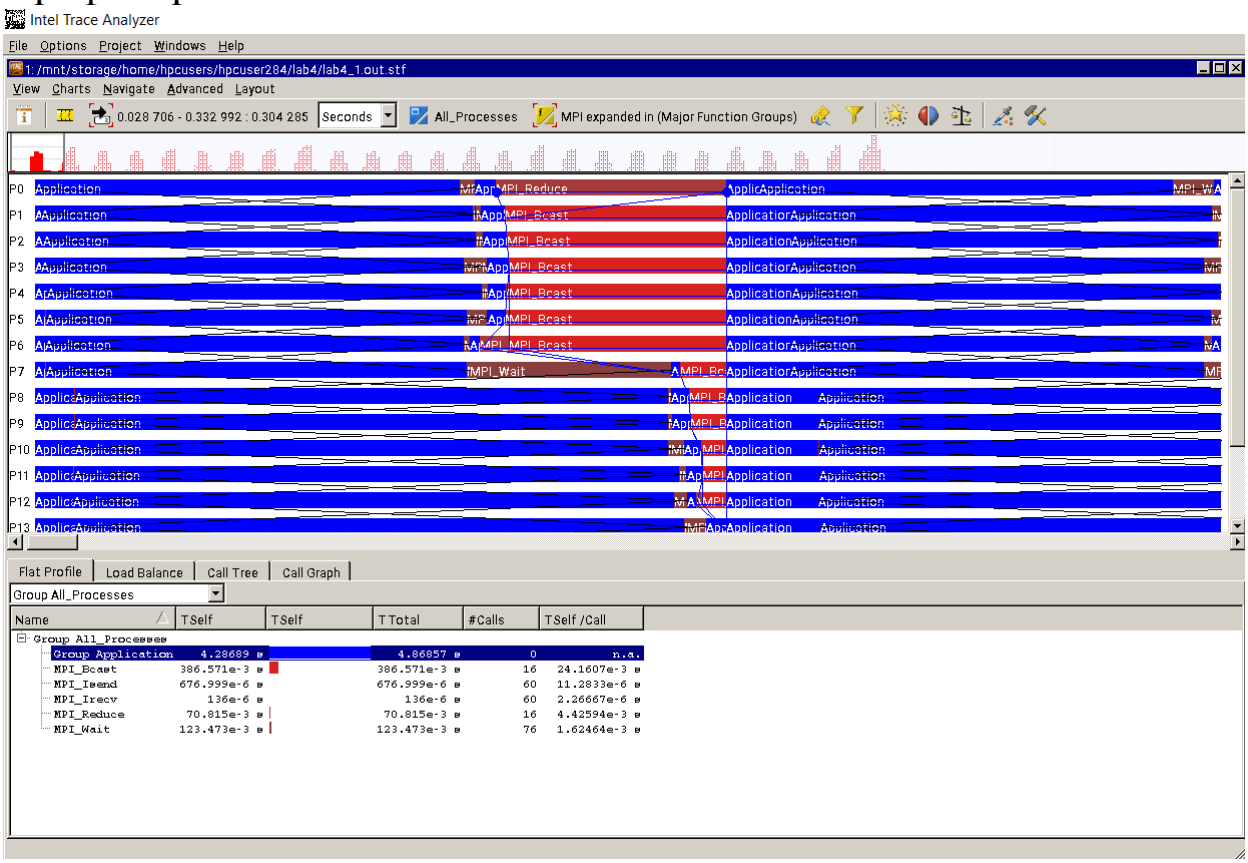
Время исполнения программы на Си на 1 ядре составляет 62,448422 sec.
ускорение: Sp = T1 / Tp, где T1 – время работы программы на 1 ядре. Tp - время работы параллельной программы на p ядрах.
Эффективность: Ep = (Sp / p) * 100%

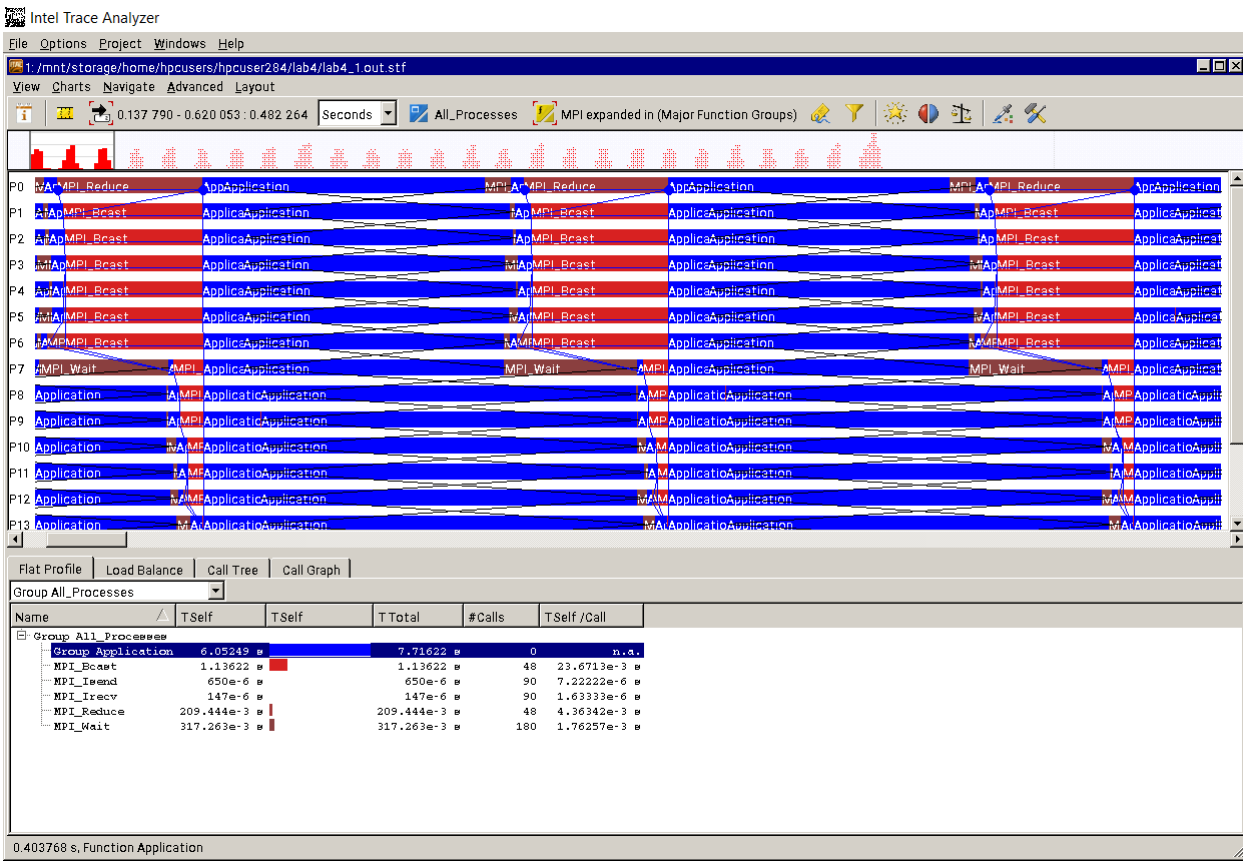
Ядра	Время исполнения	Ускорение	Эффективность
2	36,582585	1,707053288	85,35266439
4	20,853838	2,994576922	74,86442304
8	9,537773	6,547484617	81,84355772
16	4,984913	12,52748483	78,2967802

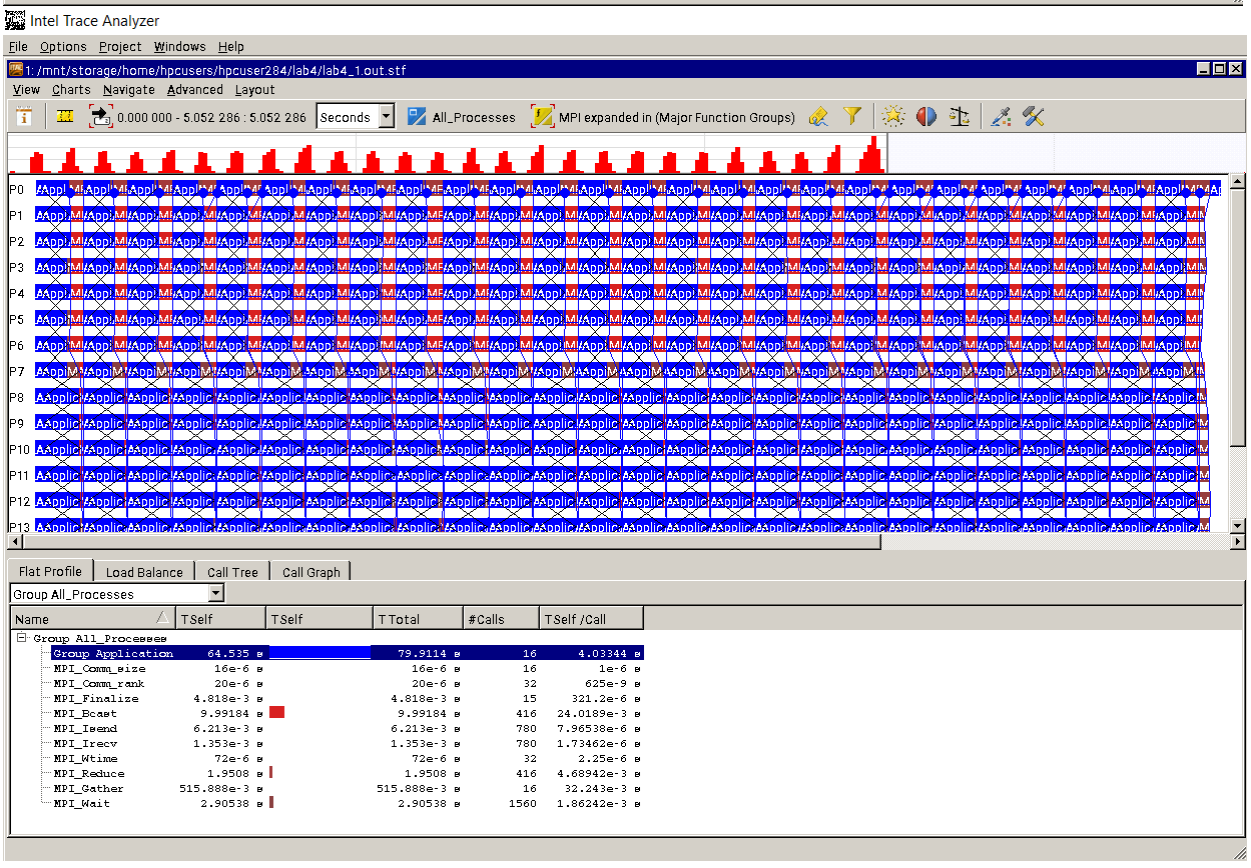




Профилирование




Intel Trace Analyzer
File Options Project Windows Help
1: /mnt/storage/home/hpcusers/hpcuser284/lab4/lab4_1.out.stf
View Charts Navigate Advanced Layout
0.137 790 - 0.620 053 : 0.482 264
Seconds
All_Processes
MPI expanded in (Major Function Groups)
Flat Profile
Load Balance
Call Tree
Call Graph
Group All_Processes
Name
TSelf
TSelf
TTotal
#Calls
TSelf /Call
Group All_Processes
Group Application 6.05249 s 7.71622 s 0 n.a.
MPI_Bcast 1.13622 s 1.13622 s 48 23.6713e-3 s
MPI_Isend 650e-6 s 650e-6 s 90 7.22222e-6 s
MPI_Irecv 147e-6 s 147e-6 s 90 1.63333e-6 s
MPI_Reduce 209.444e-3 s 209.444e-3 s 48 4.36342e-3 s
MPI_Wait 317.263e-3 s 317.263e-3 s 180 1.76257e-3 s
0.403768 s, Function Application


Intel Trace Analyzer
File Options Project Windows Help
1: /mnt/storage/home/hpcusers/hpcuser284/lab4/lab4_1.out.stf
View Charts Navigate Advanced Layout
0.000 000 - 5.052 286 : 5.052 286
Seconds
All_Processes
MPI expanded in (Major Function Groups)
Flat Profile
Load Balance
Call Tree
Call Graph
Group All_Processes
Name
TSelf
TSelf
TTotal
#Calls
TSelf /Call
Group All_Processes
Group Application 64.535 s 79.9114 s 16 4.03344 s
MPI_Comm_size 16e-6 s 16e-6 s 16 1e-6 s
MPI_Comm_rank 20e-6 s 20e-6 s 32 625e-9 s
MPI_Finalize 4.818e-3 s 4.818e-3 s 15 321.2e-6 s
MPI_Bcast 9.99184 s 9.99184 s 416 24.0189e-3 s
MPI_Isend 6.213e-3 s 6.213e-3 s 780 7.96538e-6 s
MPI_Irecv 1.353e-3 s 1.353e-3 s 780 1.73462e-6 s
MPI_Wtime 72e-6 s 72e-6 s 32 2.25e-6 s
MPI_Reduce 1.9508 s 1.9508 s 416 4.68942e-3 s
MPI_Gather 515.888e-3 s 515.888e-3 s 16 32.243e-3 s
MPI_Wait 2.90538 s 2.90538 s 1560 1.86242e-3 s

ЗАКЛЮЧЕНИЕ

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

Приложение 1. *Код программы*

```
#include <cstring>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <limits>
#include <mpi.h>
#include <algorithm>

double phi(double x, double y, double z) {
    return pow(x, x) + pow(y, y) + pow(z, z); //искомая функция phi(x, y, z)
= x^2 + y^2 + z^2
}

double ro(double x, double y, double z, const double a) {
    return 6 - a * phi(x,y,z); //правая часть уравнения ro(x, y, z) = 6 - a *
phi(x, y, z)
}

double updLayer(
        int base_z, int height, double *omega_part, double *tmp_omega_part,
        double hx, double hy, double hz,
        const int N, const double x0, const double y0, const double z0, const
double a) {

    int abs_z = base_z + height; //модуль

    if (abs_z == 0 || abs_z == N - 1) { // если граница области
        //копируем этот слой в новый массив на старое место, не пересчитывая
        memcpy(tmp_omega_part + height * N * N, omega_part + height * N * N,
N * N * sizeof(double));
        return 0;
    }
    //иначе пересчитываем каждый элемент слоя с помощью итерационной формулы
    double max_delta = 0;
    double z = z0 + abs_z * hz;

    for (int i = 0; i < N; i++) {
        double x = x0 + i * hx;

        for (int j = 0; j < N; j++) {
            double y = y0 + j * hy;

            int cell = height * N * N + i * N + j; //номер клетки в слое

            if (i == 0 || i == N - 1 || j == 0 || j == N - 1) { //если
элемент находится на границе слоя, то не пересчитываем его
                tmp_omega_part[cell] = omega_part[cell];
                continue;
            }
            //иначе пересчитываем по формуле Якоби
            tmp_omega_part[cell] = ((omega_part[height * N * N + (i + 1) * N
+ j]
                                     + omega_part[height * N * N + (i - 1) *
N + j]) / (hx * hx)
                                    + (omega_part[height * N * N + i * N + (j
+ 1)]
                                       + omega_part[height * N * N + i * N +
(j - 1)]) / (hy * hy)
                                    + (omega_part[(height + 1) * N * N + i *
N + j]
                                       + omega_part[(height - 1) * N * N + i
* N + j]) / (hz * hz)
```

```
                                    - ro(x, y, z, a)) /
                                   ((2 / (hx * hx)) + (2 / (hy * hy)) + (2 /
(hz * hz)) + a);

            max_delta = std::max(max_delta, std::abs(tmp_omega_part[cell] -
omega_part[cell]));
        }
    }

    return max_delta;
}

double JacobiMethod(const double epsilon, const double a, const int N,
                    const double x0, const double y0, const double z0,
                    const double x1, const double y1, const double z1) {

    //ищем функцию phi: d^2(phi)/d^2(x) + d^2(phi)/d^2(y) + d^2(phi)/d^2(z) -
a*phi = ro, [a >= 0]
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (N % size) {
        if(rank == 0) {
            std::cout << "Invalid number of processes" << std::endl;
            return -1;
        }
    }

    double start_time = MPI_Wtime();

    double hx = (x1 - x0) / (N - 1);     //
    double hy = (y1 - y0) / (N - 1);     // //расстояния между соседними
узлами - шаги сетки
    double hz = (z1 - z0) / (N - 1);     //

    int part_height = N / size; //высота слоя

    int part_base_z = rank * part_height - 1;  //координата для текущего
процесса

    // трехмерный массив для сохранения значений в точках слоя
    // +2 -- верхнее и нижнее внутреннее граничное
    double *omega = new double[(part_height + 2) * N * N];
    double *tmp_omega = new double[(part_height + 2) * N * N];

    int iterationsCounter = 0;

    // _Шаг алгоритма 1_
    // Задать значения искомой функции на границе области omega: phi_{i,j,k}
= F(x_i, y_j, z_k)
    // при i = 0, i = N_x, j = 0, j = N_y, k = 0, k = N_z
    // _Шаг алгоритма 2_
    // Задать начальное приближение во внутренней части области omega:
phi_{i,j,k}^0
    // для i=1..Nx-1, j=0..Ny-1, k=0..Nz-1.

    for (int i = 0; i < part_height + 2; i++) {
        int omega_z = i + part_base_z;  // Oz области омега
        double real_z = z0 + hz * omega_z;

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

            double x = x0 + hx * j;  // Ox
```

```
            for (int k = 0; k < N; k++) {

                double y = y0 + hy * k; // Oy

                if (omega_z == 0 || omega_z == N - 1 || j == 0 || j == N - 1
|| k == 0 || k == N - 1) { // значение функции на границе области [1 шаг]

                    omega[i * N * N + j * N + k] = phi(x, y, real_z);
                } else {
                    // начальное приближение во внутренней части области [2
шаг]
                    omega[i * N * N + j * N + k] = 0;
                }
            }
        }
    }


    // _Шаг алгоритма 3_
    // Многократно вычисляем очередное приближение искомой функции по формуле
phi_{i, j, k}^(m+1)
    // пока не достигнуто условие: max|phi_{i, j, k}^(m+1) - phi_{i, j,
k}^(m)| < eps. [по i, j, k]
    double max_delta_shared; // порог сх-ти
    do {
        //вычисляются сеточные значение, прилегающие к границе локальной
подобласти
        double max_delta = 0;
        double tmp_delta = updLayer(part_base_z, 1, omega, tmp_omega, hx, hy,
hz, N, x0, y0, z0, a);
        max_delta = std::max(max_delta, tmp_delta);

        tmp_delta = updLayer(part_base_z, part_height, omega, tmp_omega, hx,
hy, hz, N, x0, y0, z0, a);
        max_delta = std::max(max_delta, tmp_delta);
        //запускается асинхронный обмен граничных значений
        MPI_Request rq[4];
        //отправлить верхний пограничный слой предыдущим процессам
        if (rank != 0) {
            MPI_Isend(tmp_omega + N * N, N * N, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, &rq[0]);
            MPI_Irecv(tmp_omega, N * N, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, &rq[2]);
        }
        //отправить ниний пограничный слой следующим процессам
        if (rank != size - 1) {
            MPI_Isend(tmp_omega + part_height * N * N, N * N, MPI_DOUBLE,
                      rank + 1, 0, MPI_COMM_WORLD, &rq[1]);
            MPI_Irecv(tmp_omega + (part_height + 1) * N * N, N * N,
MPI_DOUBLE,
                      rank + 1, 0, MPI_COMM_WORLD, &rq[3]);
        }

        //выполняется вычисление остальных точек подобласти

        // пересчет всех элементов неграничных слоев
        for (int i = 2; i < part_height; i++) {
            double tmpdelta = updLayer(part_base_z, i, omega, tmp_omega, hx,
hy, hz, N, x0, y0, z0, a);
            max_delta = std::max(max_delta, tmpdelta);
        }

        //ожидание завершения обменов
```

```
        if (rank != 0) {
            MPI_Wait(&rq[0], MPI_STATUS_IGNORE);
            MPI_Wait(&rq[2], MPI_STATUS_IGNORE);
        }

        if (rank != size - 1) {
            MPI_Wait(&rq[1], MPI_STATUS_IGNORE);
            MPI_Wait(&rq[3], MPI_STATUS_IGNORE);
        }

        // полностью пересчитанная область для этого процесса
        memcpy(omega, tmp_omega, (part_height + 2) * N * N  *
sizeof(double));

        // максимальная дельта ВСЕХ процессов и отправка всем
        MPI_Reduce(&max_delta, &max_delta_shared, 1, MPI_DOUBLE, MPI_MAX, 0,
MPI_COMM_WORLD);
        MPI_Bcast(&max_delta_shared, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        iterationsCounter++;
    } while (max_delta_shared >= epsilon);
    // по достижению некоторого порога сх-ти -- завершение итерационного
процесса

    delete[] tmp_omega;

    double *fullResult = nullptr;
    if (rank == 0) {
        fullResult = new double[N * N * N];
    }

    MPI_Gather(omega + N * N, part_height * N * N, MPI_DOUBLE, fullResult,
               part_height * N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD); // сбор
подсчитанной части

    double end_time = MPI_Wtime();

    // ищем максимальное отклонение max |phi_{i,j,k}^m - phi_{i,j,k}^*|
    if (rank == 0) {

        double max_delta = 0; // подсчет максимального отклонения от
стандартного ответа

        for (int layer = 0; layer < N; layer++){
            double z = z0 + layer * hz;

            for (int j = 0; j < N; j++) {
                double x = x0 + j * hx;

                for (int k = 0; k < N; k++) {
                    double y = y0 + k * hy;

                    max_delta = std::max(max_delta, std::abs(fullResult[layer
* N * N + j * N + k] - phi(x, y, z)));
                }
            }
        }
        // оценка точности полученного решения:
        std::cout << "Answer: delta = " << max_delta << std::endl;
        printf("Time: %lf\n", end_time - start_time);
        std::cout << iterationsCounter << " cycle iterations" << std::endl;

        delete[] fullResult;
    }
```

```
    delete[] omega;

    return end_time - start_time;
}


void JacobiMethodTest(const int repeats, const double epsilon, const double
a, const int N,
    const double x0, const double y0, const double z0,
    const double x1, const double y1, const double z1) {

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double current_time, best_time = std::numeric_limits<double>::max();

    for (int i = 1; i <= repeats; i++) {
        if (rank == 0) {
        std::cout << "Try " << i << "/" << repeats << std::endl;
        }
        current_time = JacobiMethod(epsilon, a, N, x0, y0, z0, x1, y1, z1);
        if (rank == 0) {
            best_time = (current_time < best_time) ? current_time :
best_time;
        }
    }

    if (rank == 0) {
        printf("Best time: %lf sec\n", best_time);
    }
}

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    const int repeats = 1;
    const double epsilon = 1e-8;
    const double a = 1e5;
    const int N = 240;
    const double x0 = -1;
    const double y0 = -1;
    const double z0 = -1;
    const double x1 = 1;
    const double y1 = 1;
    const double z1 = 1;

    JacobiMethodTest(repeats, epsilon, a, N, x0, y0, z0, x1, y1, z1);
    MPI_Finalize();
    return 0;
}
```