###### 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 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**

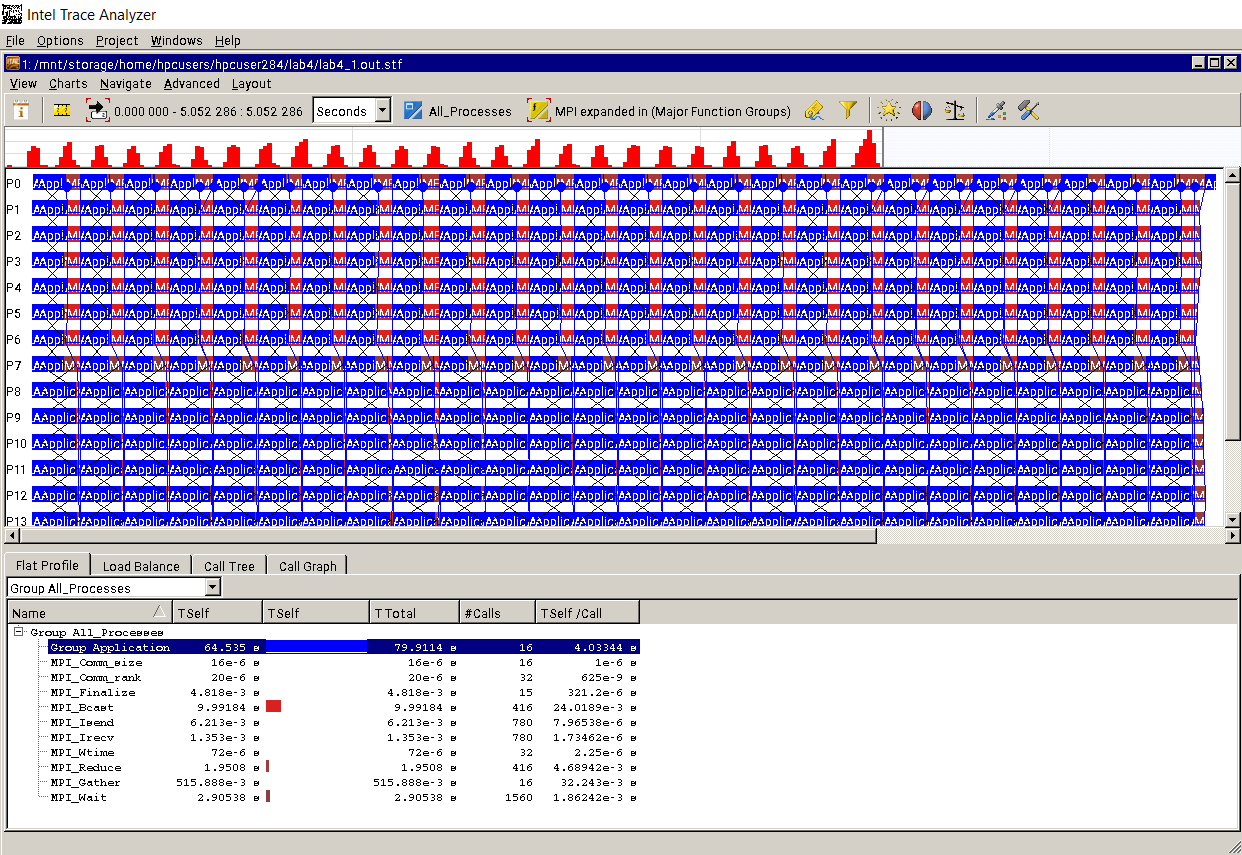
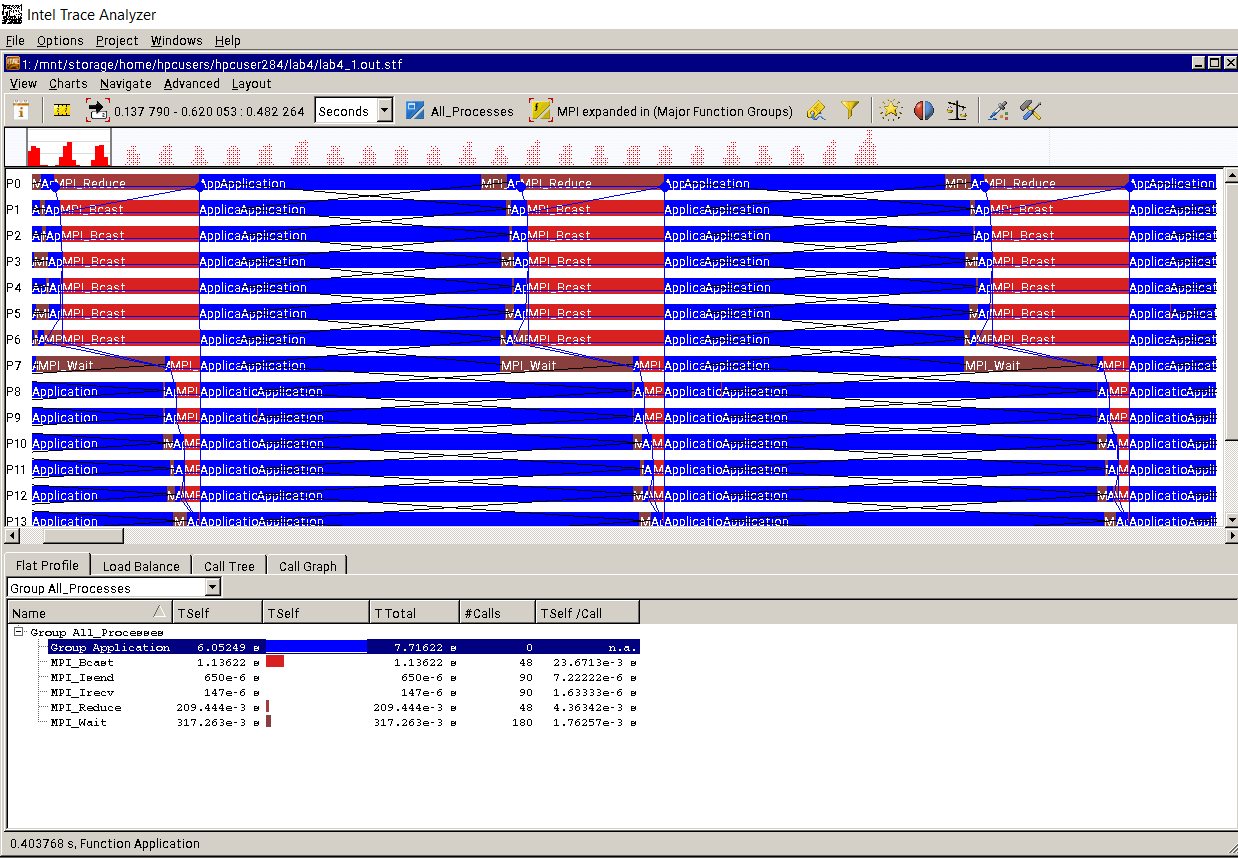
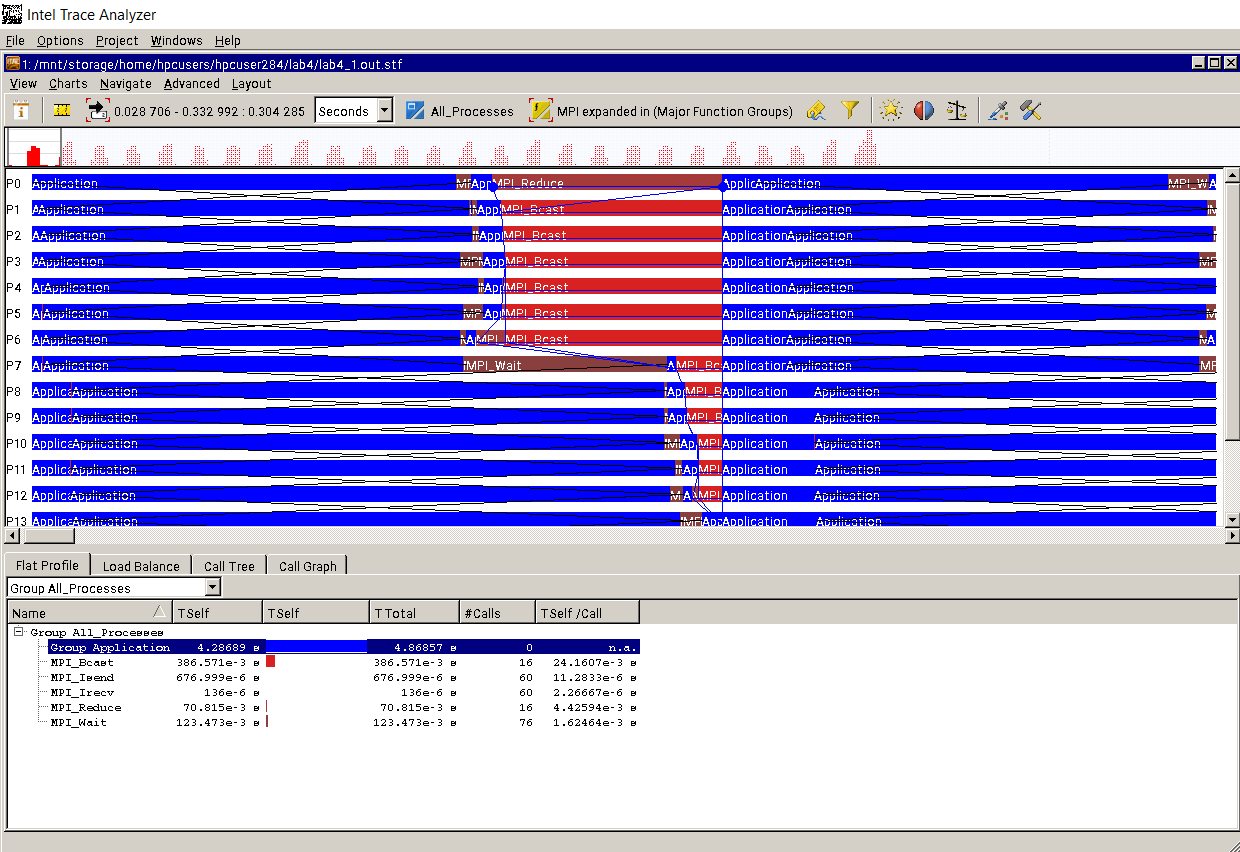
* Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.
* Измерить время работы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

# **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;  
}