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 Бородина Артёма Максимовича 2 курса, 19205 группы

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
к.т.н, доцент
А.Ю. Власенко

Новосибирск 2021

СОДЕРЖАНИЕ

ЦЕЛЬ

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

ЗАДАНИЕ

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

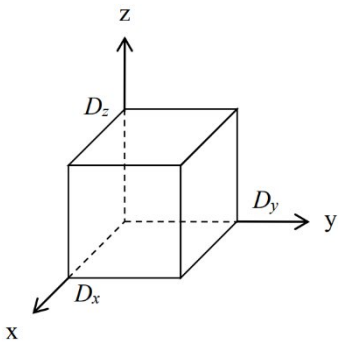
$$\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}+\frac{\partial^2\varphi}{\partial z^2}-a\varphi=\rho\,,\ a\geq 0\,,$$

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

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

$$\varphi\,|_G=F(x,y,z)\,.$$

Область Ω имеет вид прямоугольного параллелепипеда с размерами Dx × Dy × Dz.



Нужно использовать одномерную декомпозицию области.

ОПИСАНИЕ РАБОТЫ

1. Была написана программа (Приложение 1), с использованием MPI, реализующая метод Якоби в трехмерной области, где обмены граничными значениями подобластей выполняются на фоне счета.

2. Выбраны размеры сетки, при которых время работы программы на 1-ом ядре занимало примерно 30 секунд.

3. Сделано профилирование программы на 16-ти ядрах (Приложение 2). Было подтверждено что обмен граничными значениями подобластей выполняются на фоне счета. На скриншотах можно увидеть, как соседние процессы обмениваются своими граничными областями.

4. Проведены замеры времени, построены графики ускорения и эффективности (Приложение 3). По графикам можно что с увеличением количества процессов, эффективность стремительно падает. Это может происходить вследствие того, что работа каждого процесса зависит от его соседей, из-за этого замедление одного процесса повлечет замедление его соседей что отрицательно скажется на времени работы программы.

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были получены знания о разделения нагрузки на множество процессов для эффективной реализации метода Якоби в 3-х мерной области.

Приложение 1. *Код программы*

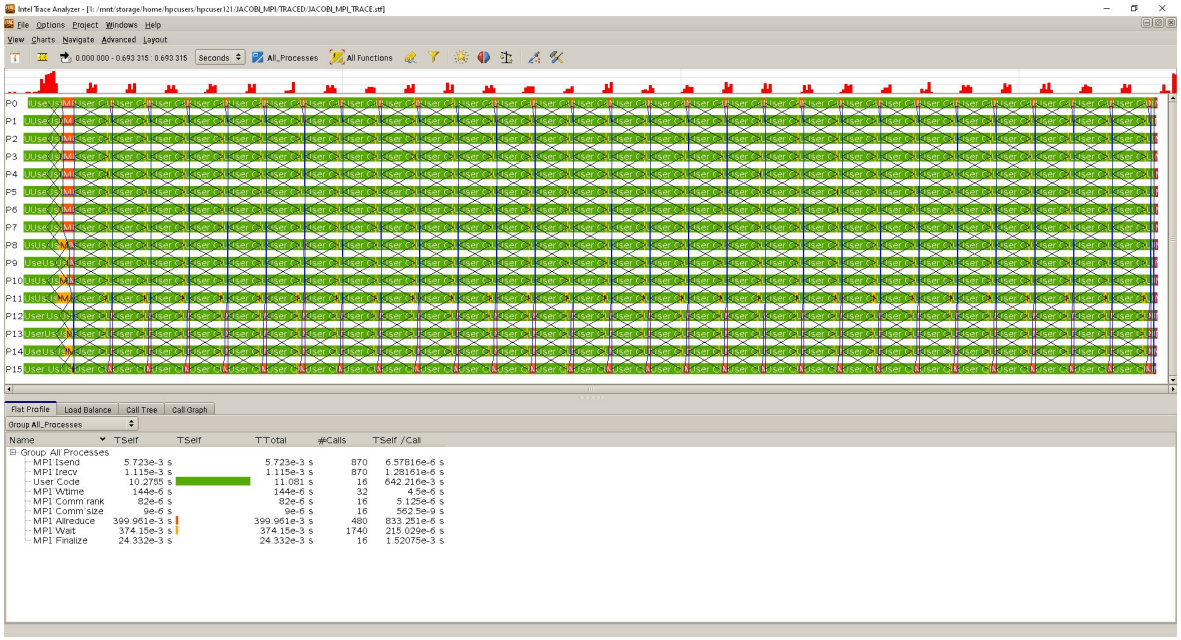
```
#include <iostream>
#include <cmath>
#include "mpi.h"

#define funcElement(i, x, y, z) functionIterations[i][(x) * Y * Z + (y) * Z + z]
#define calculateP p(((x + displs[rank]) * hx) - 1, (y * hy) - 1, (z * hz) - 1, a)
#define calculateF f(((x + displs[rank]) * hx) - 1, (y * hy) - 1, (z * hz) - 1)

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

inline double p(double x, double y, double z, double a) {
  return 6 - a * f(x, y, z);
}

int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  int procsCount, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &procsCount);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (argc < 4 && rank == 0) {
    std::cout << "Program needs 3 arguments (grid size): Nx, Ny, Nz" << std::endl;
    MPI_Finalize();
    return 0;
  }
  const int Nx = atoi(argv[1]);
  const int Ny = atoi(argv[2]);
  const int Nz = atoi(argv[3]);

  if (rank == 0) {
    std::cout << "procsCount: " << procsCount << std::endl;
    std::cout << "Grid size: " << Nx << "x" << Ny << "x" << Nz << std::endl;
  }

  int sizesPerThreads[procsCount], displs[procsCount];
  std::fill(sizesPerThreads, sizesPerThreads + procsCount, Nx / procsCount);
  for (int x = 0; x < Nx % procsCount; ++x) {
    sizesPerThreads[x] += 1;
  }
  displs[0] = 0;
  for (int x = 1; x < procsCount; ++x) {
    displs[x] = displs[x - 1] + sizesPerThreads[x - 1];
  }

  const int X = sizesPerThreads[rank];
  const int Y = Ny;
  const int Z = Nz;

  double *(functionIterations[2]);
  functionIterations[0] = new double[X * Y * Z];
  functionIterations[1] = new double[X * Y * Z];
  std::fill(functionIterations[0], functionIterations[0] + X * Y * Z, 0);
  std::fill(functionIterations[1], functionIterations[1] + X * Y * Z, 0);

  auto leftBorder = new double[Z * Y];
  auto rightBorder = new double[Z * Y];

  const double hx = 2.0 / (Nx - 1);
```

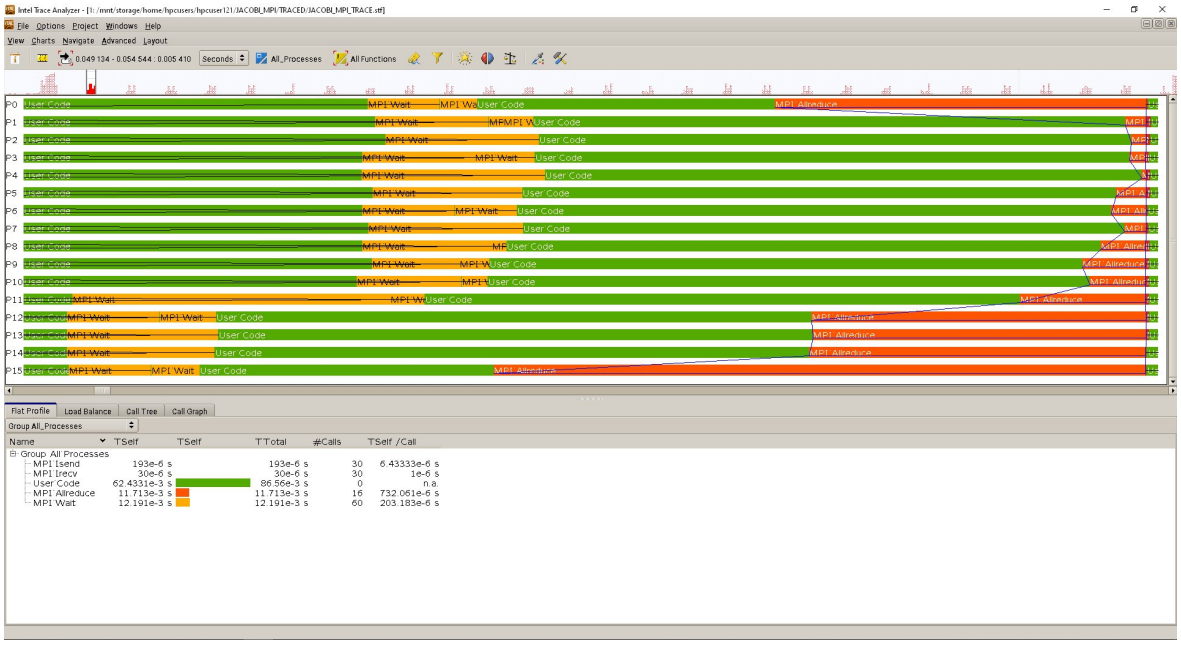
```
const double hy = 2.0 / (Ny - 1);ь ь    const double hx2 = hx * hx;
const double hy2 = hy * hy;
const double hz2 = hz * hz;
const double a = 1e5;
const double factor = 1 / (2 / hx2 + 2 / hy2 + 2 / hz2 + a);

/// Fill borders on [-1;1]x[-1;1]x[-1;1]
for (int x = 0, localX = displs[rank]; x < X; ++x, ++localX) {
  for (int y = 0; y < Y; y++) {
    for (int z = 0; z < Z; z++) {
      if ((localX == 0) || (y == 0) || (z == 0) || (localX == Nx - 1) || (y == Ny - 1) || (z == Nz - 1)) {
        funcElement(0, x, y, z) = f((localX * hx) - 1, (y * hy) - 1, (z * hz) - 1);
        funcElement(1, x, y, z) = f((localX * hx) - 1, (y * hy) - 1, (z * hz) - 1);
      }
    }
  }
}

int newIter = 0, prevIter = 1;
double phix, phiy, phiz;

MPI_Request sendLeftBorder, sendRightBorder;
MPI_Request recvLeftBorder, recvRightBorder;

const double EPSILON = 1e-8;
int criteria = 1;

double time = -MPI_Wtime();
while (criteria) {
  int tmpCriteria = 0;
  newIter = 1 - newIter;
  prevIter = 1 - prevIter;

  /// Send borders
  if (rank != 0) {
    MPI_Isend(&(funcElement(prevIter, 0, 0, 0)), Y * Z, MPI_DOUBLE,
          rank - 1, 0, MPI_COMM_WORLD, &sendLeftBorder);
    MPI_Irecv(leftBorder, Y * Z, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD, &recvLeftBorder);
  }
  if (rank != procsCount - 1) {
    MPI_Isend(&(funcElement(prevIter, (sizesPerThreads[rank] - 1), 0, 0)), Y * Z, MPI_DOUBLE,
          rank + 1, 1, MPI_COMM_WORLD, &sendRightBorder);
    MPI_Irecv(rightBorder, Y * Z, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, &recvRightBorder);
  }

  /// Calculate subregions
  for (int x = 1; x < X - 1; ++x) {
    for (int y = 1; y < Y - 1; ++y) {
      for (int z = 1; z < Z - 1; ++z) {
        phix = (funcElement(prevIter, x - 1, y, z) + funcElement(prevIter, x + 1, y, z)) / hx2;
        phiy = (funcElement(prevIter, x, y - 1, z) + funcElement(prevIter, x, y + 1, z)) / hy2;
        phiz = (funcElement(prevIter, x, y, z - 1) + funcElement(prevIter, x, y, z + 1)) / hz2;
        double element = factor * (phix + phiy + phiz - calculateP);
        funcElement(newIter, x, y, z) = element;

        tmpCriteria = fabs(element - calculateF) > EPSILON ? 1 : 0;
      }
    }
  }

  /// Wait for borders
  if (rank != 0) {
    MPI_Wait(&sendLeftBorder, MPI_STATUS_IGNORE);
```

```
      MPI_Wait(&recvLeftBorder, MPI_STATUS_IGNORE);
    }
    if (rank != procsCount - 1) {
      MPI_Wait(&sendRightBorder, MPI_STATUS_IGNORE);
      MPI_Wait(&recvRightBorder, MPI_STATUS_IGNORE);
    }

    /// Calculate borders
    for (int y = 1; y < Y - 1; ++y) {
      for (int z = 1; z < Z - 1; ++z) {
        /// Left border
        if (rank != 0) {
          int x = 0;
          phix = (leftBorder[y * Z + z] + funcElement(prevIter, x + 1, y, z)) / hx2;
          phiy = (funcElement(prevIter, x, y - 1, z) + funcElement(prevIter, x, y + 1, z)) / hy2;
          phiz = (funcElement(prevIter, x, y, z - 1) + funcElement(prevIter, x, y, z + 1)) / hz2;
          double element = factor * (phix + phiy + phiz - calculateP);
          funcElement(newIter, x, y, z) = element;

          tmpCriteria = fabs(element - calculateF) > EPSILON ? 1 : 0;
        }

        /// Right border
        if (rank != procsCount - 1) {
          int x = X - 1;
          phix = (funcElement(prevIter, x - 1, y, z) + rightBorder[y * Z + z]) / hx2;
          phiy = (funcElement(prevIter, x, y - 1, z) + funcElement(prevIter, x, y + 1, z)) / hy2;
          phiz = (funcElement(prevIter, x, y, z - 1) + funcElement(prevIter, x, y, z + 1)) / hz2;
          double element = factor * (phix + phiy + phiz - calculateP);
          funcElement(newIter, x, y, z) = element;

          tmpCriteria = fabs(element - calculateF) > EPSILON ? 1 : 0;
        }
      }
    }
    MPI_Allreduce(&tmpCriteria, &criteria, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
  }
  time += MPI_Wtime();

  double tmpMax = 0, abs;
  for (int x = 1; x < X - 1; ++x) {
    for (int y = 1; y < Y - 1; ++y) {
      for (int z = 1; z < Z - 1; ++z) {
        if ((abs = fabs(funcElement(newIter, x, y, z) - calculateF)) > tmpMax) {
          tmpMax = abs;
        }
      }
    }
  }

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

  if (rank == 0) {
    std::cout << "Time: " << time << std::endl;
    std::cout << "Max difference: " << max << std::endl;
  }

  delete[] functionIterations[0];
  delete[] functionIterations[1];
  delete[] leftBorder;
  delete[] rightBorder;
```

```
    MPI_Finalize();
    return 0;
}
```

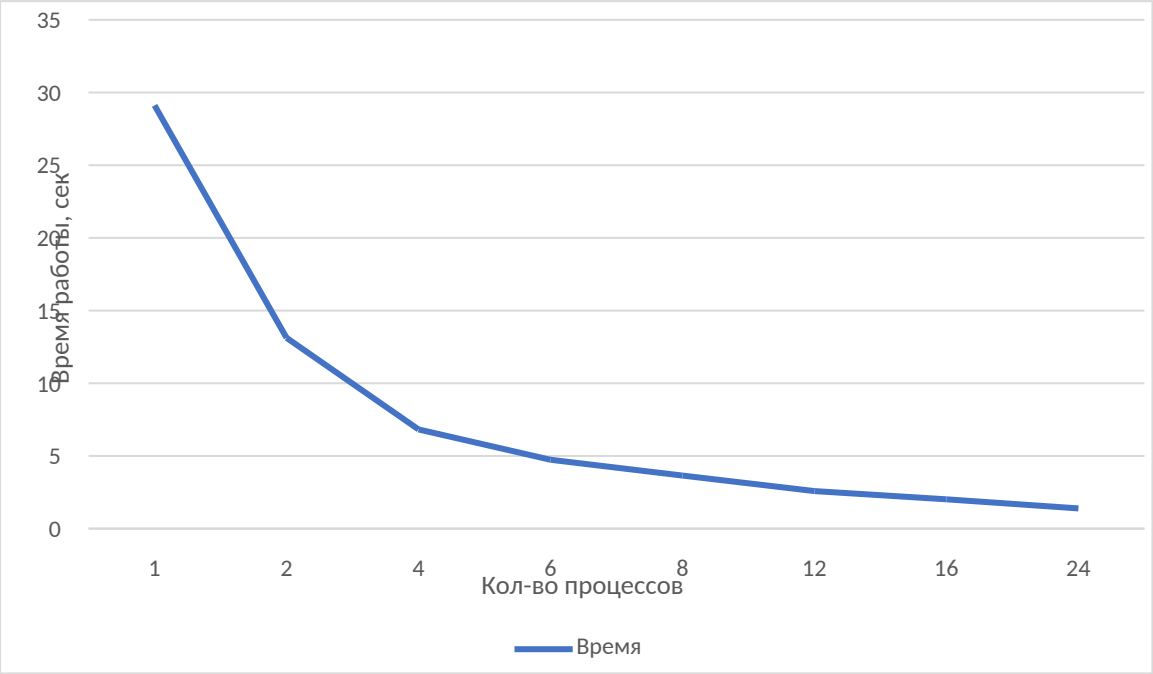
Приложение 2. *Профилирование на 16 ядрах*



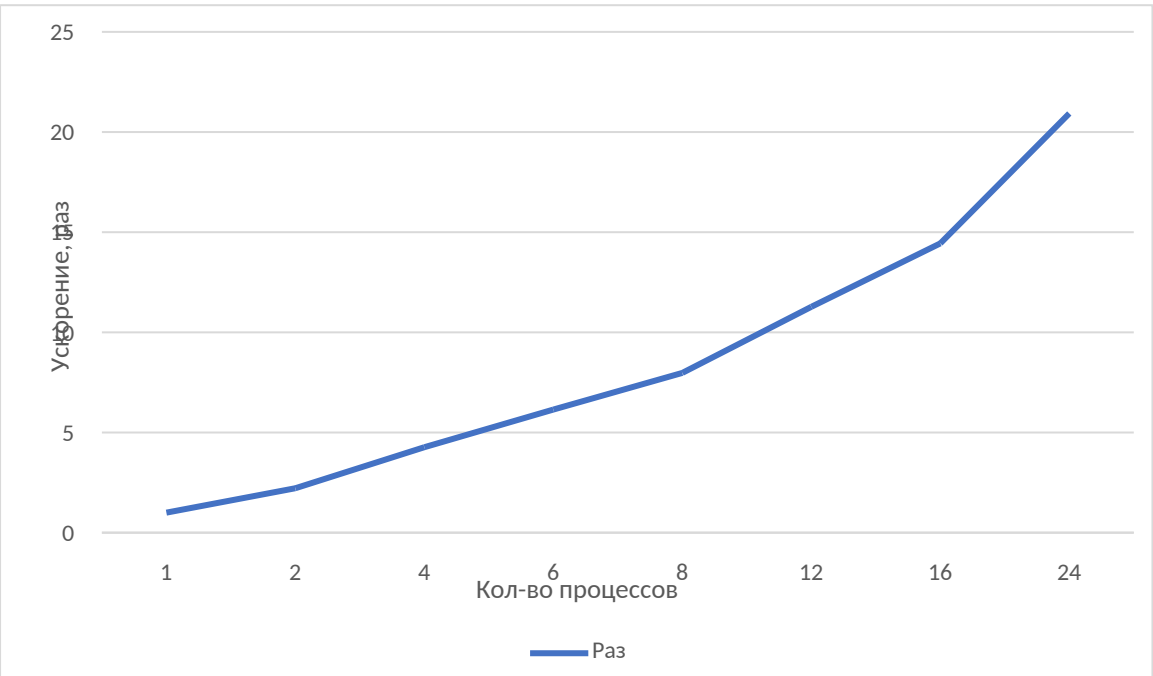


Приложение 3. *Замеры времени*

Время работы, сек								
	1	2	4	6	8	12	16	24
	29,11	13,13	6,82	4,74	3,65	2,58	2,02	1,39



Ускорение, раз								
	1	2	4	6	8	12	16	24
	1	2,22	4,27	6,15	7,98	11,27	14,44	20,93



Эффективность на процесс, %								
	1	2	4	6	8	12	16	24
	1	1,11	1,07	1,02	1,00	0,94	0,90	0,87