###### 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, 22208 группы

**Лебедева Антона Андреевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Кудинов

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc18443921)

[ЗАДАНИЕ 3](#_Toc18443922)

[ГРАФИКИ 4](#_Toc18443923)

[ПРОФИЛИРОВАНИЕ 6](#_Toc18443923)

[ЗАКЛЮЧЕНИЕ](#_Toc18443924) 8

[Приложение Листинг](#_Toc18443925) 9

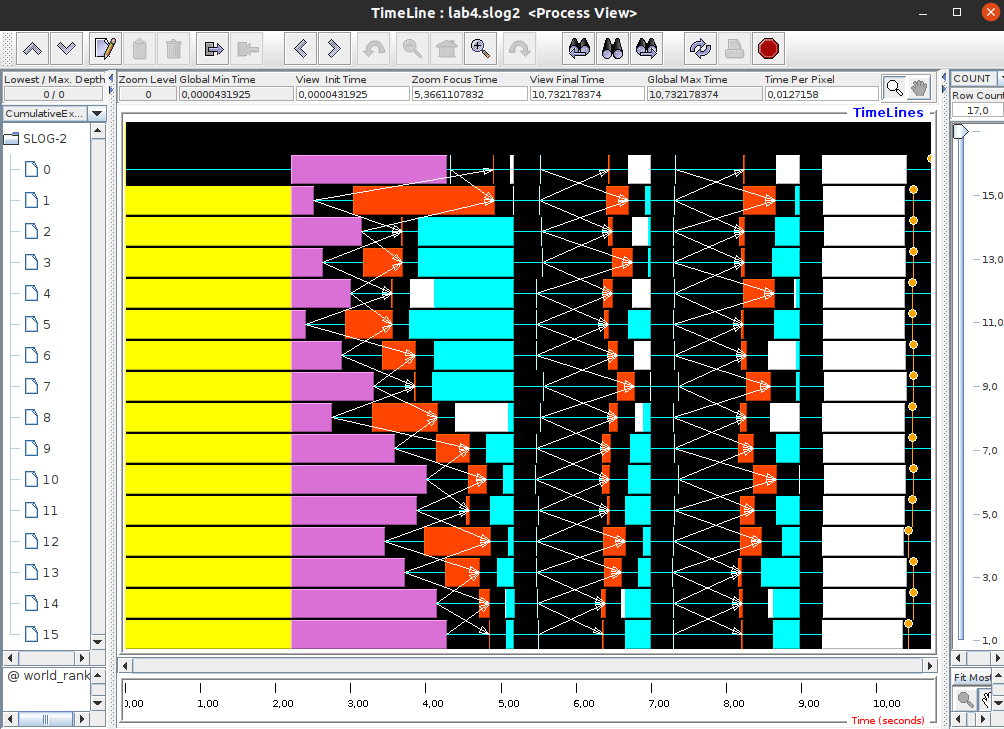
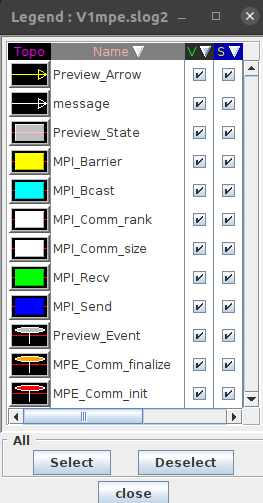
# ЦЕЛЬ

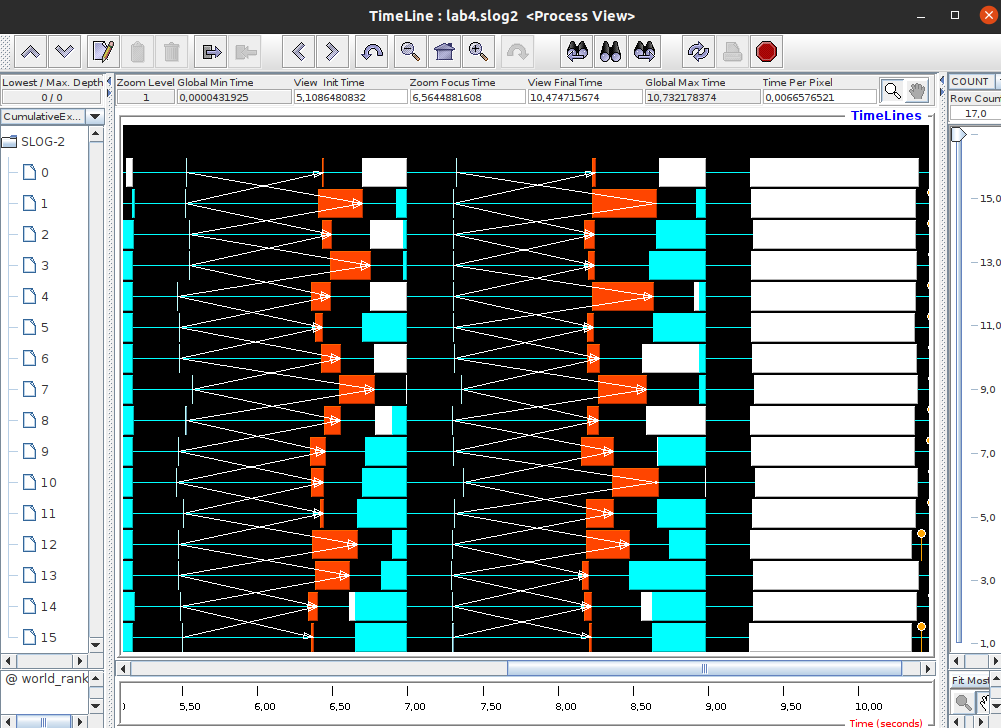
Практическое освоение методов распараллеливания численных алгоритмов  
на регулярных сетках на примере реализации метода Якоби в трехмерной области.

# ЗАДАЧА

1. 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.
2. 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.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер. По профилю убедиться, что коммуникации происходят на фоне счета.**ГРАФИКИ**

**Профилирование**

****



**ЗАКЛЮЧЕНИЕ**

В результате выполнения данной работы был освоен метод распараллеливания численных алгоритмов на регулярных сетках в трехмерной области. Изучены неблокирующие операции и асинхронная отправка. Мы выяснили, что подобные операции позволяют использовать вычислительные мощности максимально эффективно за счёт наименьшего времени простоя процессов.

# ЛИСТИНГ

#include <mpi.h>

#include <algorithm>

#include <iostream>

constexpr double

    x1 = -2,

    x2 = 1,

    y1 = -3,

    y2 = 1,

    z1 = -5,

    z2 = 1;

constexpr double a = 100000;

constexpr double epsilon = 1e-8;

constexpr int

    Nx = 1000,

    Ny = 700,

    Nz = 700,

    size2D = Nx \* Ny;

constexpr double

    Dx = x2 - x1,

    Dy = y2 - y1,

    Dz = z2 - z1;

constexpr double

    hx = Dx / (Nx - 1),

    hy = Dy / (Ny - 1),

    hz = Dz / (Nz - 1);

double Fi(double x, double y, double z) {

    x = x1 + x \* hx;

    y = y1 + y \* hy;

    z = z1 + z \* hz;

    return x \* x + y \* y + z \* z;

}

double ro(double x, double y, double z) {

*// auto fi = Fi(x, y, z);*

*// x = x1 + x \* hx;*

*// y = y1 + y \* hy;*

*// z = z1 + z \* hz;*

    return 6 - a \* Fi(x, y, z);

}

bool isLimit(int x, int y, int z, int height, int width, int depth) {

    return x == 0 || x == height - 1 || y == 0 || y == width - 1 || z == 0 || z == depth - 1;

}

void fillMatrix(double\* matrix) {

    for (int z = 0; z < Nz; ++z) {

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

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

                if (isLimit(x, y, z, Nx, Ny, Nz)) {

                    matrix[x + Nx \* y + size2D \* z] = Fi(x, y, z);

                } else {

                    matrix[x + Nx \* y + size2D \* z] = 0;

                }

            }

        }

    }

}

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

    int rank, size;

    double startTime, endTime;

    MPI\_Init(&argc, &argv);

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

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

*// Вычисление размеров подматриц для каждого процесса*

    int\* sizes = new int[size];

    std::fill(sizes, sizes + size, Nz / size);

    for (int i = 0, count = Nz % size; count > 0; ++i, --count) {

        sizes[i]++;

    }

    std::for\_each(sizes, sizes + size, [](int& n) { n \*= size2D; });

*// Вычисление смещений для каждого процесса*

    int\* displs = new int[size];

    displs[0] = 0;

    for (int i = 1; i < size; ++i) {

        displs[i] = displs[i - 1] + sizes[i - 1];

    }

*// Создание и заполнение полной матрицы (только на нулевом процессе)*

    double\* fullMatrix = nullptr;

    if (!rank) {

        fullMatrix = new double[Nx \* Ny \* Nz];

        fillMatrix(fullMatrix);

    }

    if (!rank) {

        startTime = MPI\_Wtime();

    }

*// Распределение подматриц между процессами*

    double\* submatrix = new double[sizes[rank] + 2 \* size2D];

    MPI\_Scatterv(fullMatrix, sizes, displs, MPI\_DOUBLE, submatrix + size2D,

                 sizes[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    if (!rank) {

        std::copy(fullMatrix, fullMatrix + sizes[0], submatrix + size2D);

    }

    int next = rank + 1;

    int prev = rank - 1;

    bool flag = true;

    bool\* flags;

    double\* tmpMatrix = new double[sizes[rank] + 2 \* size2D];

*// просто для красоты*

    if (rank == 0) {

        prev = -1;

        flags = new bool[size];

    }

    if (rank == size - 1) {

        next = -1;

    }

    while (flag) {

        MPI\_Request trash = MPI\_REQUEST\_NULL;

        MPI\_Request rNext, rPrev;

*// Отправляем и получаем граничные значения между процессами*

        if (rank != 0) {

            MPI\_Isend(submatrix + size2D, size2D, MPI\_DOUBLE, prev, 0,

                      MPI\_COMM\_WORLD, &trash);

            MPI\_Irecv(submatrix, size2D, MPI\_DOUBLE, prev, 1, MPI\_COMM\_WORLD, &rPrev);

        }

        if (rank != size - 1) {

            MPI\_Isend(submatrix + sizes[rank], size2D, MPI\_DOUBLE, next, 1,

                      MPI\_COMM\_WORLD, &trash);

            MPI\_Irecv(submatrix + sizes[rank] + size2D, size2D, MPI\_DOUBLE, next,

                      0, MPI\_COMM\_WORLD, &rNext);

        }

*// Если мы в процессе с рангом 0, то нет предыдущего процесса*

        if (!rank) {

            rPrev = MPI\_REQUEST\_NULL;

        }

*// Если мы в последнем процессе, то нет следующего процесса*

        if (rank == size - 1) {

            rNext = MPI\_REQUEST\_NULL;

        }

        double alpha = 1 / (2 / (hx \* hx) + 2 / (hy \* hy) + 2 / (hz + hz) + a);

*// Смещение для текущей подматрицы*

        int displ = 0;

        for (int i = 0; i < rank; ++i) {

            displ += sizes[i];

        }

*// Вычисляем центральные значения*

        for (int i = size2D; i < sizes[rank] + size2D; ++i) {

            int realI = i + displ - size2D;

            int realZ = realI / (size2D),

                realY = (realI - realZ \* size2D) / Nx,

                realX = realI - realZ \* size2D - realY \* Nx;

            int curZ = i / size2D - 1,

                curY = realY,

                curX = realX;

            if (!isLimit(curX, curY, curZ, Nx, Ny, sizes[rank] / size2D) &&

                !isLimit(realX, realY, realZ, Nx, Ny, Nz)) {

                auto tmpX = (submatrix[i - 1] + submatrix[i + 1]) / hx \* hx,

                     tmpY = (submatrix[i - Nx] + submatrix[i + Nx]) / hy \* hy,

                     tmpZ = (submatrix[i - size2D] + submatrix[i + size2D]) / hz \* hz;

                auto tmpRo = ro(realX, realY, realZ);

                tmpMatrix[i] = alpha \* (tmpX + tmpY + tmpZ - tmpRo);

            }

        }

*// Ожидаем завершения всех неблокирующих операций*

        MPI\_Request requestArray[2];

        requestArray[0] = rNext;

        requestArray[1] = rPrev;

        if (MPI\_Waitall(2, requestArray, MPI\_STATUS\_IGNORE) != MPI\_SUCCESS) {

            std::cout << "Rank: " << rank << " error on waitall" << std::endl;

        }

*// Вычисляем граничные значения*

        for (int i = size2D; i < sizes[rank] + size2D; ++i) {

            int realI = i + displ - size2D;

            int realZ = realI / (size2D);

            int realY = (realI - realZ \* size2D) / Nx;

            int realX = realI - realZ \* size2D - realY \* Nx;

            if (isLimit(realX, realY, realZ, Nx, Ny, Nz)) {

*// Если элемент является граничным, оставляем его значение без изменений*

                tmpMatrix[i] = submatrix[i];

                continue;

            }

*// Вычисление координаты z для элемента в подматрице*

            int curZ = i / size2D - 1;

            if (curZ == 0 || curZ == (sizes[rank] / size2D) - 1) {

*// Вычисление значений направлений x, y, z для элемента*

                auto tmpX = (submatrix[i - 1] + submatrix[i + 1]) / hx \* hx;

                auto tmpY = (submatrix[i - Nx] + submatrix[i + Nx]) / hy \* hy;

                auto tmpZ = (submatrix[i - size2D] + submatrix[i + size2D]) / hz \* hz;

                auto tmpRo = ro(realX, realY, realZ);

*// Вычисление нового значения для элемента в временной подматрице*

                tmpMatrix[i] = alpha \* (tmpX + tmpY + tmpZ - tmpRo);

            }

        }

*// Проверяем условие завершения*

        flag = false;

        for (int i = size2D; i < sizes[rank] + size2D; ++i) {

            if (std::abs(submatrix[i] - tmpMatrix[i]) > epsilon) {

                flag = true;

                break;

            };

        }

*// Собираем флаги из всех процессов и проверяем, есть ли еще необходимость в итерации*

        MPI\_Gather(&flag, 1, MPI\_CXX\_BOOL, flags, 1, MPI\_CXX\_BOOL, 0, MPI\_COMM\_WORLD);

        if (!rank) {

            flag = false;

            for (int i = 0; i < size; ++i) {

                if (flags[i] == true) {

                    flag = true;

                    break;

                }

            }

        }

*// Рассылаем флаг всем процессам*

        MPI\_Bcast(&flag, 1, MPI\_CXX\_BOOL, 0, MPI\_COMM\_WORLD);

*// Копируем временную подматрицу в основную подматрицу*

        std::copy(tmpMatrix, tmpMatrix + sizes[rank] + 2 \* size2D, submatrix);

    }

    MPI\_Gatherv(submatrix + size2D, sizes[rank], MPI\_DOUBLE, fullMatrix,

                sizes, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    if (!rank) {

        endTime = MPI\_Wtime();

        std::cout << "Time: " << endTime - startTime << std::endl;

    }

    delete[] displs;

    delete[] sizes;

    delete[] fullMatrix;

    delete[] submatrix;

    delete[] tmpMatrix;

    MPI\_Finalize();

}