**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, 19202 группы

**Ивакина Александра Олеговича**

Направление 09.03.01 – «Информатика и вычислительная техника»

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

Новосибирск 2021

# СОДЕРЖАНИЕ

## ЦЕЛЬ

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

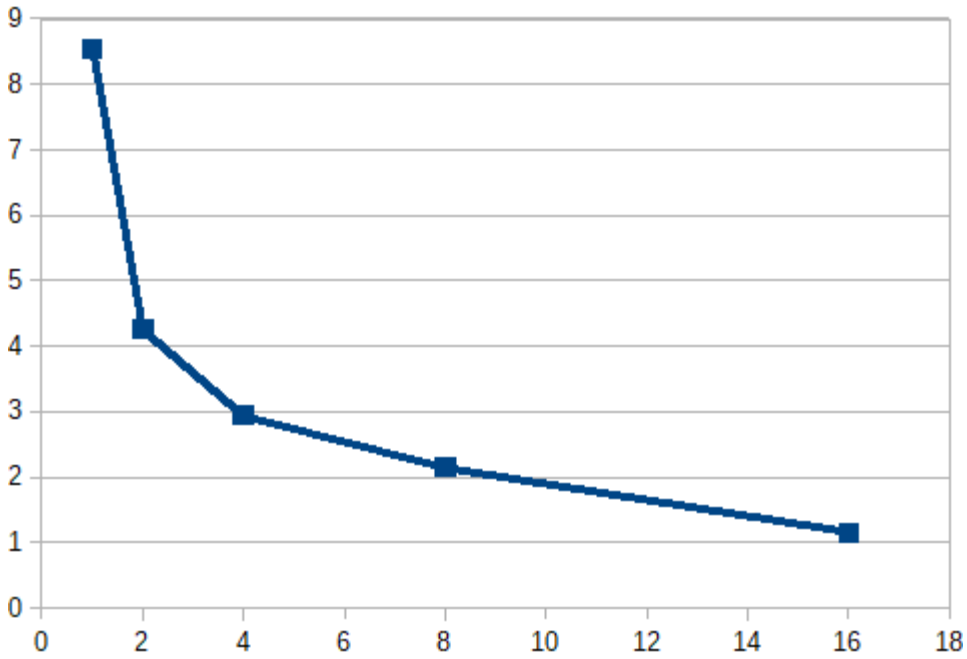
## ЗАДАНИЕ

Написать параллельную программу с использованием MPI, реализующую решение уравнения методом Якоби в трёхмерной области. Измерить время работы программы на разном числе ядер.

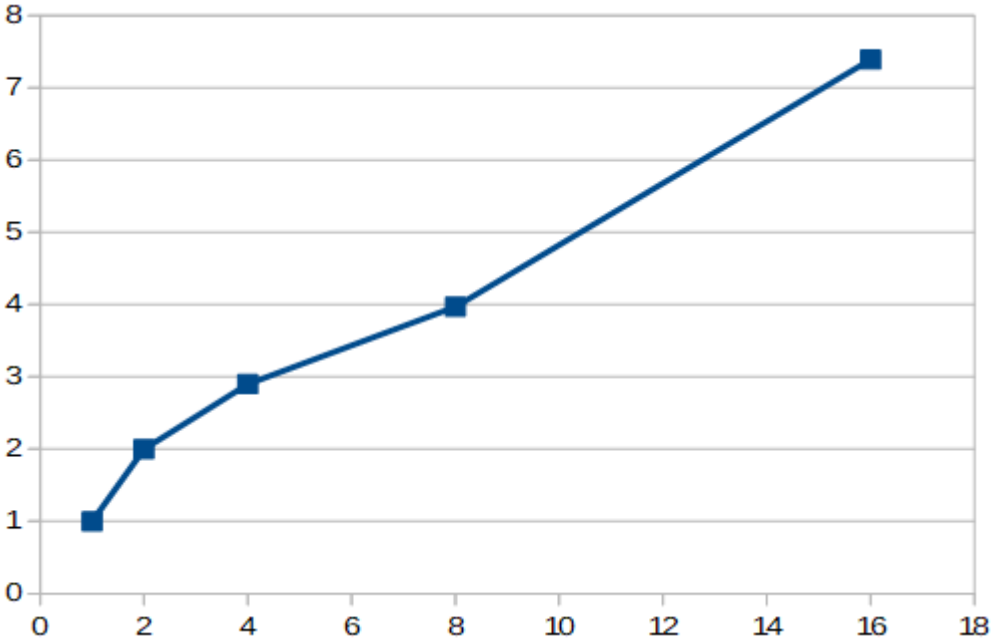
# ОПИСАНИЕ РАБОТЫ

Была написана параллельная программа, реализующая решение уравнения методом Якоби. Граничные значения блока вычислений процесса отправлялись соседним процессам и получались от них. Эти результаты ожидались одновременно с вычислением значений внутри блоков.

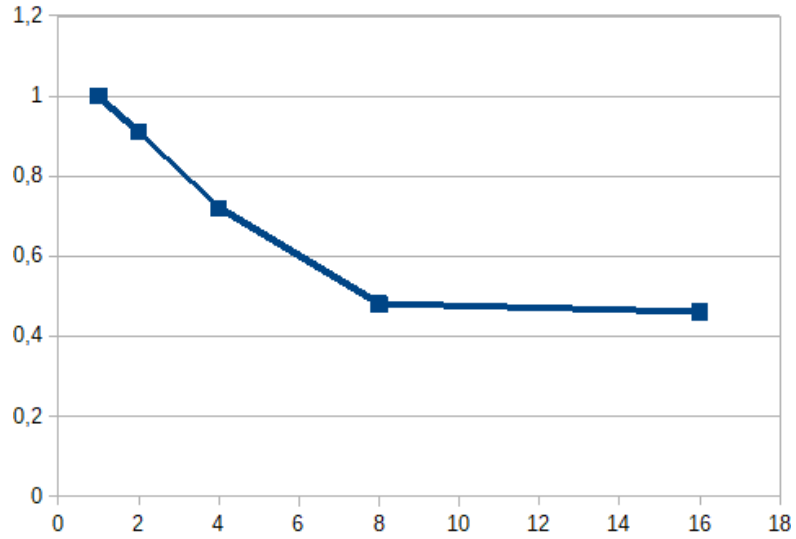
Зависимость времени работы программы от числа ядер:



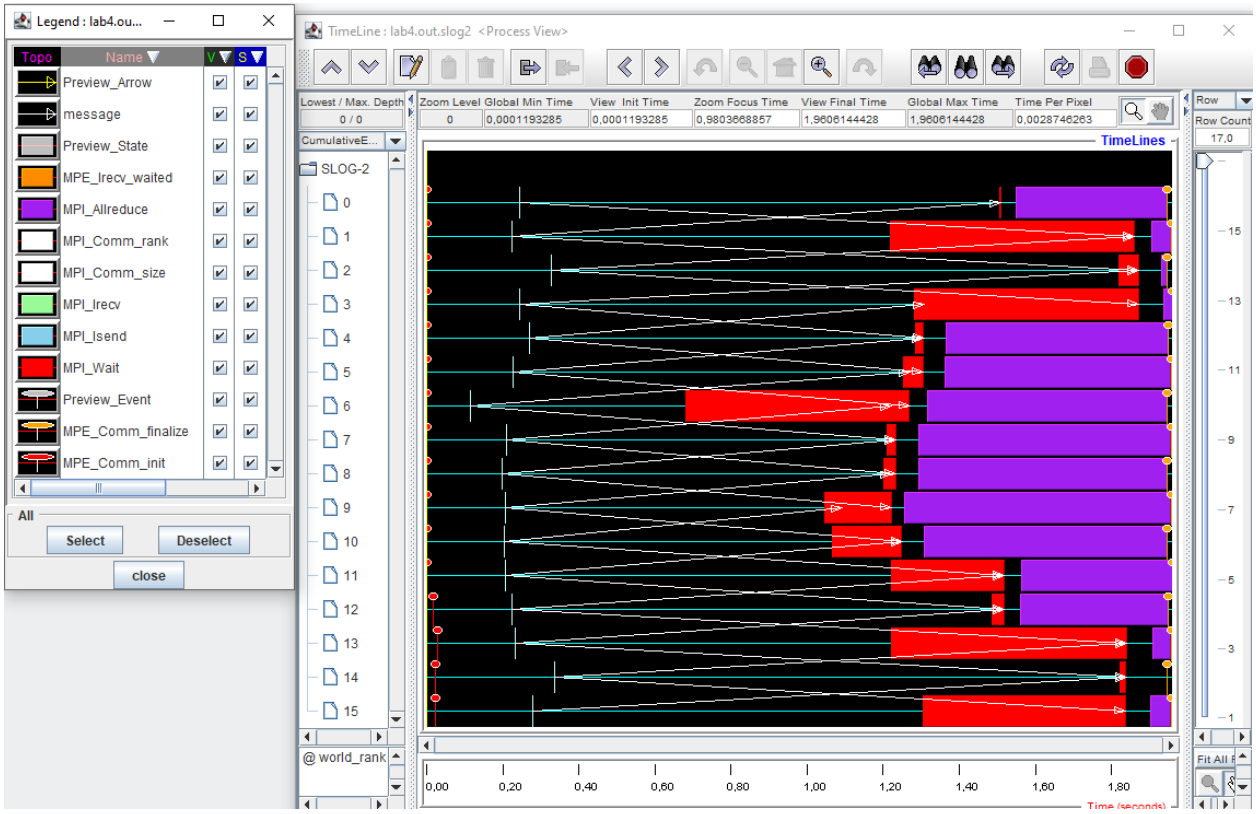

Зависимость ускорения от числа ядер:



Зависимость эффективности от числа ядер:




Профилирование:



# ЗАКЛЮЧЕНИЕ

В результате проделанной работы были освоены методы распараллеливания численных алгоритмов на регулярных сетках.

## **Приложение 1.** Код программы

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <mpi.h>

MPI_Request send_request[2];
MPI_Request recieve_request[2];

int old = 0;
int new = 1;

double a = 100000;
double epsilon = 0.00000001;

double get_coordinates(int proc_amount, int coord) {
return -1 + coord * (2 / (proc_amount));
}

double phi(int Nx, int Ny, int Nz, double x, double y, double z)
{
int coord_x = get_coordinates(Nx, x);
int coord_y = get_coordinates(Ny, y);
int coord_z = get_coordinates(Nz, z);

return coord_x * coord_x + coord_y * coord_y + coord_z * coord_z;
}

double right_part(int Nx, int Ny, int Nz, double x, double y,
double z) {
return 6 - a * phi(Nx, Ny, Nz, x, y, z);
}

int fill_matrix(double** matrix, int Nx, int Ny, int Nz, int
rank, int proc_num) {
for (int i = 0; i < Nz; i++) {
for (int j = 0; j < Ny; j++) {
for (int k = 0; k < Nx; k++) {
if (k == 0 || j == 0 || k == Nx - 1 || j == Ny - 1) {
matrix[0][i * Nx * Ny + j * Nx + k] = phi(Nx, Ny, Nz, i, j, k);
matrix[1][i * Nx * Ny + j * Nx + k] = phi(Nx, Ny, Nz, i, j, k);
}
else if ((i == 0 && rank == 0) || ((i == Nz / proc_num - 1) &&
rank == proc_num - 1)) {
matrix[0][i * Nx * Ny + j * Nx + k] = phi(Nx, Ny, Nz, i, j, k);
matrix[1][i * Nx * Ny + j * Nx + k] = phi(Nx, Ny, Nz, i, j, k);
}
else {
matrix[0][i * Nx * Ny + j * Nx + k] = 0;
```

```
46.matrix[1][i * Nx * Ny + j * Nx + k] = 0;
47.}
48.}
49.}
50.}
51.}
52.
53.void send_recv_jacobi(double** send_matrix, double** recv_matrix,
int Nx, int Ny, int Nz, int rank, int proc_num) {
54.if (rank != proc_num - 1) {
55.MPI_Isend(send_matrix[old], Nx * Ny, MPI_DOUBLE, rank + 1, 1,
MPI_COMM_WORLD, &send_request[1]);
56.MPI_Irecv(recv_matrix[1], Nx * Ny, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD, &recieve_request[0]);
57.}
58.if (rank != 0) {
59.MPI_Isend(send_matrix[old], Nx * Ny, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, &send_request[0]);
60.MPI_Irecv(recv_matrix[0], Nx * Ny, MPI_DOUBLE, rank - 1, 1,
MPI_COMM_WORLD, &recieve_request[1]);
61.}
62.}
63.
64.void count_center(double** matrix, int Nx, int Ny, int Nz, int*
keep_calc, int rank, int proc_num) {
65.double hx = 2 / (Nx - 1);
66.double hy = 2 / (Ny - 1);
67.double hz = 2 / (Nz - 1);
68.
69.for (int i = 1; i < Nz - 1; i++) {
70.for (int j = 1; j < Ny - 1; j++) {
71.for (int k = 1; k < Nx - 1; k++) {
72.matrix[new][i * Nx * Ny + j * Ny + k] = ((matrix[old][(i - 1) *
Nx * Ny + j * Nx + k] + matrix[old][(i + 1) * Nx * Ny + j * Nx + k])
/ (hx * hx)
73.+ (matrix[old][i * Nx * Ny + (j + 1) * Nx + k] + matrix[old][i *
Nx * Ny + (j - 1) * Nx + k]) / (hy * hy)
74.+ (matrix[old][i * Nx * Ny + j * Nx + k + 1] + matrix[old][i * Nx
* Ny + j * Nx + k - 1]) / (hz * hz)
75.- right_part(Nx, Ny, Nz, i, j, k)) /
76.(2 / (hx * hx) + 2 / (hy * hy) +
77.2 / (hz * hz) + a);
78.if (abs(matrix[new][i * Nx * Ny + j * Nx + k] - matrix[old][i *
Nx * Ny + j * Nx + k]) > epsilon) {
79.*keep_calc = 1;
80.}
81.}
82.}
83.}
84.
85.}
86.
87.void wait_data(int rank, int proc_num) {
```

```
88.if (rank != proc_num - 1) {
89.MPI_Wait(&send_request[1], MPI_STATUS_IGNORE);
90.MPI_Wait(&recieve_request[0], MPI_STATUS_IGNORE);
91.}
92.if (rank != 0) {
93.MPI_Wait(&send_request[0], MPI_STATUS_IGNORE);
94.MPI_Wait(&recieve_request[1], MPI_STATUS_IGNORE);
95.}
96.}
97.
98.void count_edges(double** matrix, double** recv_matrix, int Nx,
int Ny, int Nz, int* keep_calc, int rank, int proc_num) {
99.double hx = 2 / (Nx - 1);
100.double hy = 2 / (Ny - 1);
101.double hz = 2 / (Nz - 1);
102.
103.if (rank != proc_num - 1) {
104.int i = proc_num - 1;
105.for (int j = 1; j < Ny - 1; j++) {
106.for (int k = 1; k < Nx - 1; k++) {
107.matrix[new][i * Nx * Ny + j * Ny + k] = ((matrix[old][(i - 1) *
Nx * Ny + j * Nx + k] + recv_matrix[1][j * Ny + k]) / (hx * hx)
108.+ (matrix[old][i * Nx * Ny + (j + 1) * Nx + k] + matrix[old][i *
Nx * Ny + (j - 1) * Nx + k]) / (hy * hy)
109.+ (matrix[old][i * Nx * Ny + j * Nx + k + 1] + matrix[old][i *
Nx * Ny + j * Nx + k - 1]) / (hz * hz)
110.- right_part(Nx, Ny, Nz, i, j, k)) /
111.(2 / (hx * hx) + 2 / (hy * hy) +
112.2 / (hz * hz) + a);
113.if (abs(matrix[new][i * Nx * Ny + j * Nx + k] - matrix[old][i *
Nx * Ny + j * Nx + k]) > epsilon) {
114.*keep_calc = 1;
115.}
116.}
117.}
118.}
119.if (rank != 0) {
120.int i = 0;
121.for (int j = 1; j < Ny - 1; j++) {
122.for (int k = 1; k < Nx - 1; k++) {
123.matrix[new][i * Nx * Ny + j * Nx + k] = ((recv_matrix[0][i * Nx
* Ny + j * Nx + k] + matrix[old][(i + 1) * Nx * Ny + j * Nx + k]) /
(hx * hx)
124.+ (matrix[old][i * Nx * Ny + (j + 1) * Nx + k] + matrix[old][i *
Nx * Ny + (j - 1) * Nx + k]) / (hy * hy)
125.+ (matrix[old][i * Nx * Ny + j * Nx + k + 1] + matrix[old][i *
Nx * Ny + j * Nx + k - 1]) / (hz * hz)
126.- right_part(Nx, Ny, Nz, i, j, k)) /
127.(2 / (hx * hx) + 2 / (hy * hy) +
128.2 / (hz * hz) + a);
129.if (abs(matrix[new][i * Nx * Ny + j * Nx + k] - matrix[old][i *
Nx * Ny + j * Nx + k]) > epsilon) {
130.*keep_calc = 1;
```

```
}
}
}
}

}

int main(int argc, char** argv) {
// SETTING UP
MPI_Init(&argc, &argv);
int proc_num;
MPI_Comm_size(MPI_COMM_WORLD, &proc_num);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int Nx = 480;
int Ny = 480;
int Nz = 480 / proc_num;
double** matrix = (double**)malloc(2 * sizeof(double*));
double** rec_matrix = (double**)malloc(2 * sizeof(double*));
for (int i = 0; i < 2; i++) {
matrix[i] = (double*)malloc(Nx * Ny * Nz * sizeof(double));
rec_matrix[i] = (double*)malloc(Nx * Ny * Nz * sizeof(double));
}

fill_matrix(matrix, Nx, Ny, Nz, rank, proc_num);

double start = MPI_Wtime();

int keep_calc = 1;
while (keep_calc) {
keep_calc = 0;
// sending and recieving data
send_recv_jacobi(matrix, rec_matrix, Nx, Ny, Nz, rank,
proc_num);

// calculating middle values
count_center(matrix, Nx, Ny, Nz, &keep_calc, rank, proc_num);

// waiting for data
wait_data(rank, proc_num);

// calculating edge values
count_edges(matrix, rec_matrix, Nx, Ny, Nz, &keep_calc, rank,
proc_num);

old = 1 - old;
new = 1 - new;

int keep_calc_tmp;
MPI_Allreduce(&keep_calc, &keep_calc_tmp, 1, MPI_INT, MPI_MAX,
MPI_COMM_WORLD);
keep_calc = keep_calc_tmp;
```

```
}

double finish = MPI_Wtime();
if (rank == 0) {
printf("Time taken: %lf\n", finish - start);
}

for (int i = 0; i < 2; i++) {
free(matrix[i]);
free(rec_matrix[i]);
}
free(matrix);
free(rec_matrix);

MPI_Finalize();
return 0;
}
```