###### 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**

«Программирование многопоточных приложений. POSIX  
Threads»

студента 2 курса, 21203 группы

**Неретина Степана Ивановича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЗАДАНИЕ 3](#_heading=h.gjdgxs)

[ОПИСАНИЕ РАБОТЫ 4](#_heading=h.30j0zll)

Вывод9

[Приложение 1. Исходный коод](#_heading=h.1fob9te)  10

# ЗАДАНИЕ

Реализовать некоторый аналог вычислительного кластера,  
принимающего при старте некоторый список задач для каждой “ноды кластера” - процесса MPI программы.  
Реализовать систему автобалансировки задач между всеми нодами в случае, когда у какой-то из доступных закончились собственные,т.е. в тот момент, когда вычислительная эффективность кластера начинает уменьшаться. Общение нод, т.е. процессов, реализовать  
посредством стандартного MPI.  
Работа ноды кластера должна быть реализована следующим  
образом: в процессе должно быть запущено 2 POSIX потока:  
● поток “ исполнитель”, исполняющий поступившие к нему задачи;  
● поток “ коммуникатор ”, коммуницирующий со всеми остальными  
нодами

# ОПИСАНИЕ РАБОТЫ

# Написан код (Приложение 1, 2), который реализует поставленную задачу

# Пример вывода

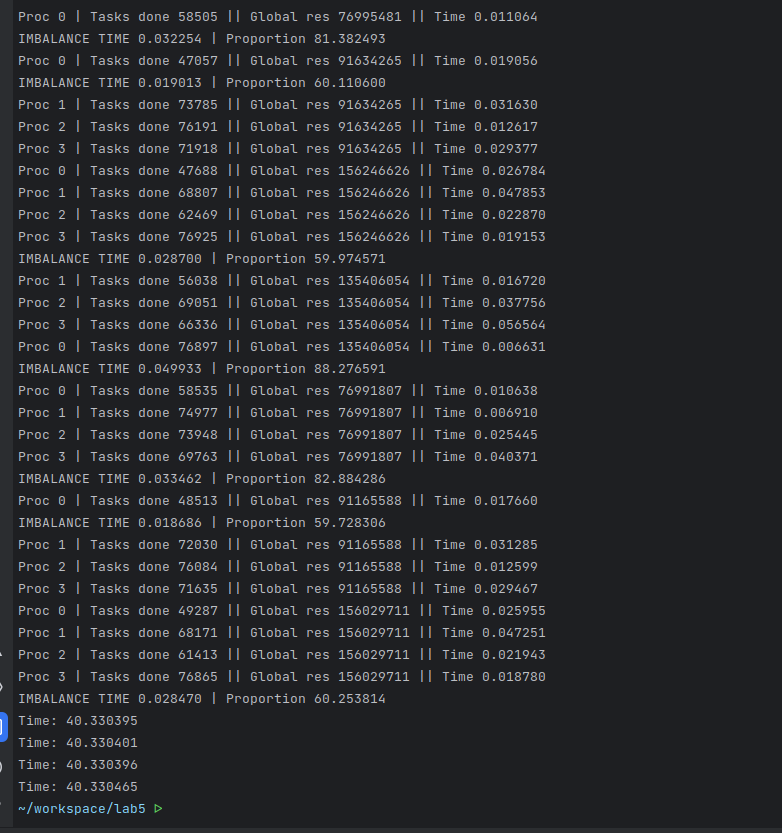
4 процесса

MAX\_TASKS 150000 \* size(size=4)

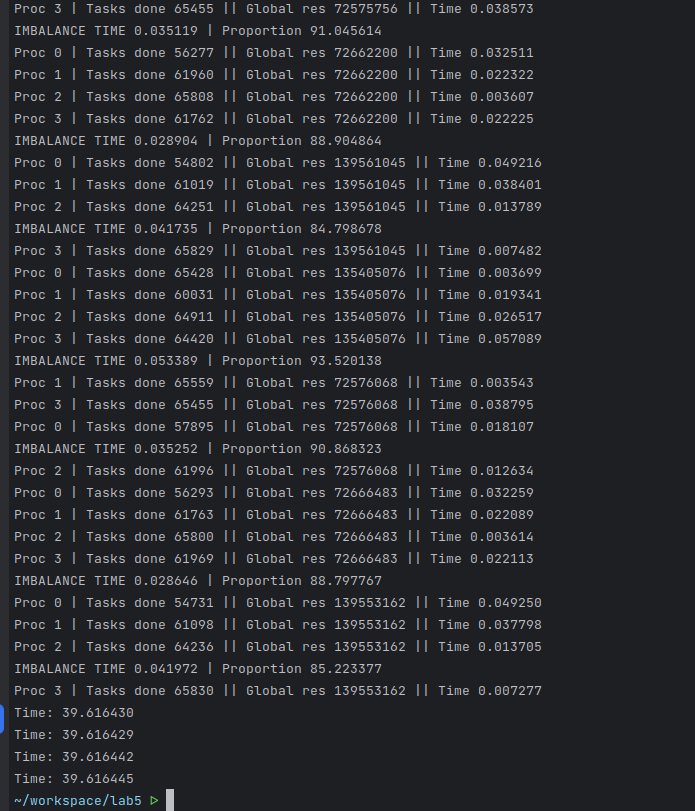
ITERATIONS\_COUNT 1000

L 2

30%



10%



**Вывод**

В ходе выполнения данной лабораторной работы мной были освоены  
концепции POSIX Threads, научился динамически распределять работу между  
процессами и разделение ответственности между потоками.

# **Приложение 1. *Параллельная программа***

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <mpi.h>

#include <math.h>

#define MAX\_TASKS 15000

#define ITERATIONS\_COUNT 1000

#define L 2

#define REQUEST\_TASK 228

#define STOP\_CODE 229

#define TO\_RECEIVER 1

#define SEND\_WORK\_TAG 2

int \*tasks;

int size;

int rank;

int offset;

double \*times;

pthread\_mutex\_t mutex;

struct Job\_Requester{

int count\_tasks;

int local\_res;

};

struct Job\_Requester do\_job(int length) {

int res = 0;

int count\_tasks = 0;

struct Job\_Requester req;

while (1) {

pthread\_mutex\_lock(&mutex);

if (offset >= length) {

pthread\_mutex\_unlock(&mutex);

break;

}

int current\_offset = offset++;

pthread\_mutex\_unlock(&mutex);

int weight = tasks[current\_offset];

// }

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

res += (int)sqrt(j);

}

count\_tasks++;

}

req.local\_res = res;

req.count\_tasks = count\_tasks;

return req;

}

void fill\_tasks(int iter\_count) {

for (int i = 0; i < size \* MAX\_TASKS; i++) {

tasks[i] = abs(50 - i % 100) \* abs(rank - iter\_count % size) \* L;

}

}

struct Job\_Requester do\_request\_tasks() {

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

if (i == rank) continue;

int req\_code = REQUEST\_TASK;

int help\_length;

//

MPI\_Send(&req\_code, 1, MPI\_INT, i, TO\_RECEIVER, MPI\_COMM\_WORLD);

MPI\_Recv(&help\_length, 1, MPI\_INT, i, SEND\_WORK\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

if (help\_length > 0) {

MPI\_Recv(tasks, help\_length, MPI\_INT, i, SEND\_WORK\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

pthread\_mutex\_lock(&mutex);

offset = 0;

pthread\_mutex\_unlock(&mutex);

}

return do\_job(help\_length);

}

}

void find\_max\_min(const double\* arr, int len, double\* max, double\* min) {

// Initialize max and min to the first element of the array

\*max = \*min = arr[0];

// Iterate over the array and update max and min as needed

for (int i = 1; i < len; i++) {

if (arr[i] > \*max) {

\*max = arr[i];

}

if (arr[i] < \*min) {

\*min = arr[i];

}

}

}

void do\_tasks() {

double start\_time = MPI\_Wtime();

struct Job\_Requester res1 = do\_job(size \* MAX\_TASKS);

struct Job\_Requester res2 = do\_request\_tasks();

double end\_time = MPI\_Wtime();

int general\_do\_task\_counter = res1.count\_tasks + res2.count\_tasks;

int general\_proc\_local\_res = res1.local\_res + res2.local\_res;

int general\_proc\_global\_res = 0;

MPI\_Allreduce(&general\_proc\_local\_res, &general\_proc\_global\_res, 1, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD);

double work\_time = end\_time - start\_time;

printf("Proc %d | Tasks done %d || Global res %d || Time %f\n", rank, general\_do\_task\_counter, general\_proc\_global\_res, work\_time);

MPI\_Gather(&work\_time, 1, MPI\_DOUBLE, times, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if(rank == 0){

double max\_time;

double min\_time;

find\_max\_min(times, size, &max\_time, &min\_time);

double imbalance\_time = max\_time - min\_time;

printf("IMBALANCE TIME %f | Proportion %f\n", imbalance\_time, imbalance\_time / max\_time \* 100);

}

}

void\* worker\_func(void\* arg) {

for (int i = 0; i < ITERATIONS\_COUNT; i++) {

pthread\_mutex\_lock(&mutex);

offset = 0;

fill\_tasks(i);

pthread\_mutex\_unlock(&mutex);

do\_tasks();

}

int stop\_code = STOP\_CODE;

MPI\_Barrier(MPI\_COMM\_WORLD);

MPI\_Send(&stop\_code, 1, MPI\_INT, rank, TO\_RECEIVER, MPI\_COMM\_WORLD);

//printf("SEND %d %d\n", stop\_code, rank);

return NULL;

}

void\* receiver\_func(void\* arg) {

while (1) {

int req\_code\_buf;

MPI\_Status status\_worker\_requester;

MPI\_Recv(&req\_code\_buf, 1, MPI\_INT, MPI\_ANY\_SOURCE, TO\_RECEIVER, MPI\_COMM\_WORLD, &status\_worker\_requester);

if (req\_code\_buf == STOP\_CODE){

break;

}

size\_t length = size \* MAX\_TASKS;

int new\_offset = offset + (int)((int)(length - offset) \* 0.3);

int tasks\_length = new\_offset - offset;

MPI\_Send(&tasks\_length, 1, MPI\_INT, status\_worker\_requester.MPI\_SOURCE, SEND\_WORK\_TAG, MPI\_COMM\_WORLD);

if (tasks\_length > 0 ) {

pthread\_mutex\_lock(&mutex);

int old\_offset = offset;

offset = new\_offset;

pthread\_mutex\_unlock(&mutex);

MPI\_Send(&tasks[old\_offset], tasks\_length, MPI\_INT, status\_worker\_requester.MPI\_SOURCE, SEND\_WORK\_TAG, MPI\_COMM\_WORLD);

}

}

return NULL;

}

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

int provided;

MPI\_Init\_thread(&argc, &argv, MPI\_THREAD\_MULTIPLE, &provided);

double start\_time = MPI\_Wtime();

if (provided != MPI\_THREAD\_MULTIPLE) {

printf("Invalid thread level support\n");

exit(-1);

}

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

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

tasks = (int\*) malloc(size \* MAX\_TASKS \* sizeof(int));

if(rank == 0){

times =(double\*) malloc(size \* sizeof(double));

}

pthread\_t worker, receiver;

pthread\_mutex\_init(&mutex, NULL);

pthread\_create(&worker, NULL, worker\_func, NULL);

pthread\_create(&receiver, NULL, receiver\_func, NULL);

pthread\_join(worker, NULL);

pthread\_join(receiver, NULL);

if (tasks != NULL)

free(tasks);

if (rank == 0)

free(times);

double end\_time = MPI\_Wtime();

printf("Time: %f\n", end\_time - start\_time);

MPI\_Finalize();

return 0;

}