Министерство образования и науки Российской Федерации

Новосибирский национальный исследовательский государственный университет

Основы параллельного программирования

Отчет по лабораторной работе № 5

Студент: Ланин Даниил Михайлович

Преподаватель: Артюхов Алексей Андреевич

Новосибирск, 2023 г.

1. **Цель работы**

Освоить разработку многопоточных программ с использованием POSIX Threads API. Познакомиться с задачей динамического распределения работы между процессорами.

1. **Краткое описание подхода к организации решения прикладной задачи параллельными взаимодействующими процессами**

Для динамического распределения работы между процессорами в данном коде используется подход с использованием параллельных потоков и MPI для обмена данными между процессами.

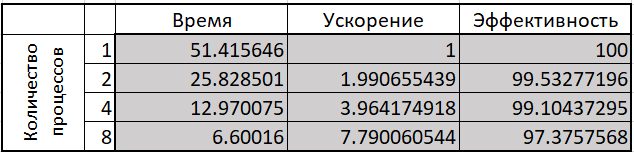
В функции **main** создаются два потока: **worker\_thread** и **server\_thread**. Поток **worker\_thread** выполняет задачи, а поток **server\_thread** принимает запросы от других процессов и выдает им задачи. Каждый поток работает в бесконечном цикле, пока не выполнятся все задачи.

В функции **worker\_thread** каждый процесс выбирает нераспределенную задачу из своего диапазона задач, либо запрашивает задачу у другого процесса, если у него закончились задачи. После получения задачи вызывается функция **calculate\_task**, которая выполняет задачу и помечает ее как выполненную.

В функции **server\_thread** процесс принимает запросы от других процессов и выдает им свободные задачи. Если у самого процесса закончились задачи, он отправляет специальное сообщение о завершении работы.

1. **Исследование производительности программы**

Вычисления производились на 320 задачах, где сложность задачи вычисляется по формуле: i \* 1000 mcs, где i – номер задачи.



**Вывод программы:**

**На 2 процессах:**

Time left: **25.828501**

All tasks difficulty: **51.360000**

All difficulty / N: **25.680000**

Difference: **0.148501**

**На 4 процессах:**

Time left: **12.970075**

All tasks difficulty: **51.360000**

All difficulty / N: **12.840000**

Difference: **0.130075**

**На 8 процессах:**

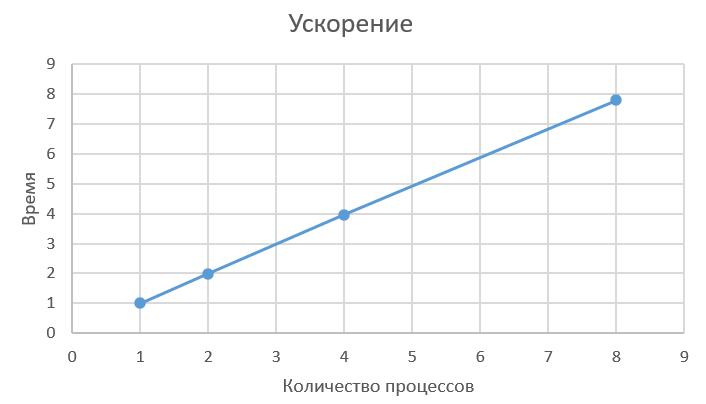
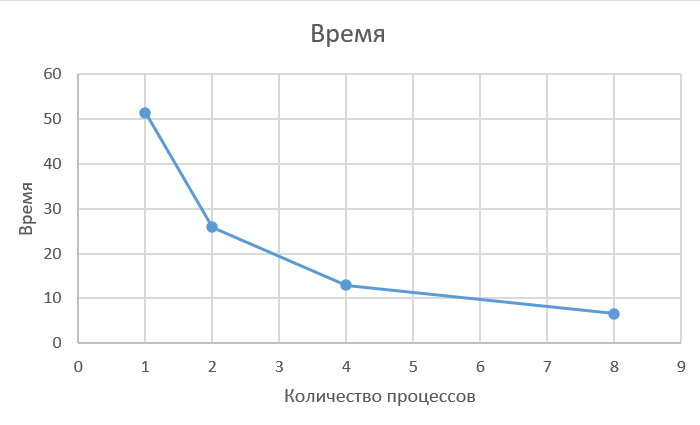
Time left: **6.600160**

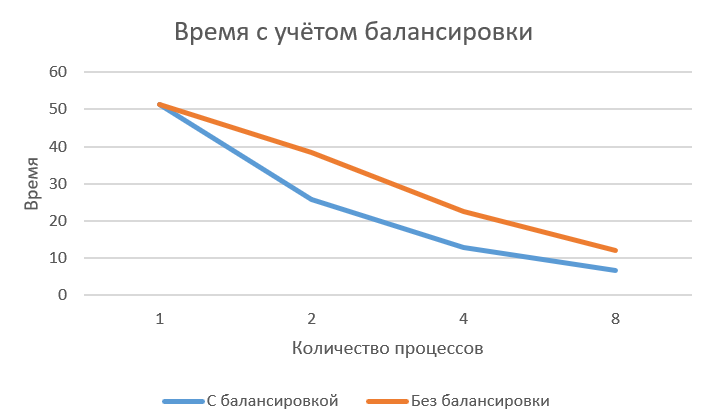
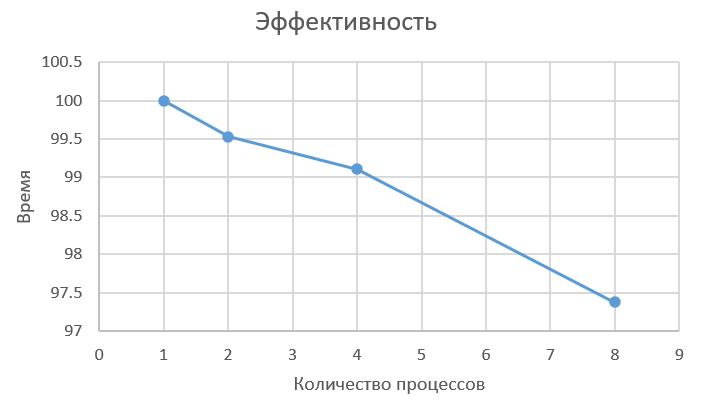
All tasks difficulty: **51.360000**

All difficulty / N: **6.420000**

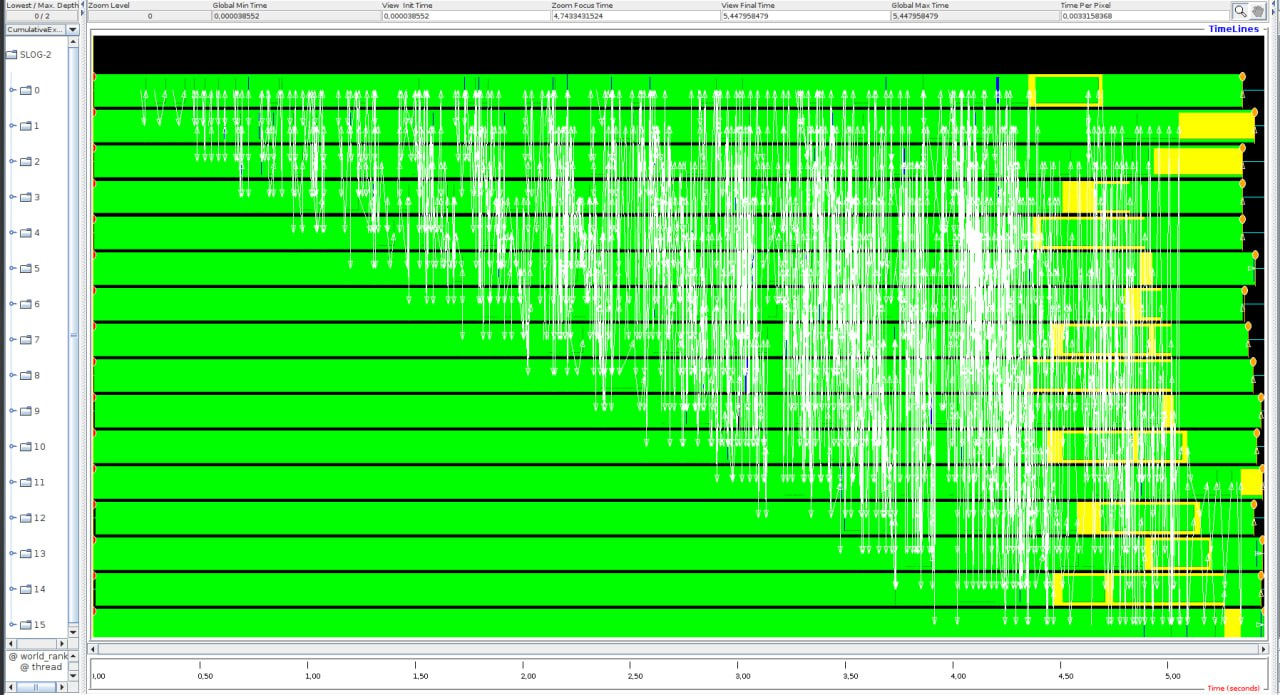
Difference: **0.180160**

По данным вывода программы не трудно заметить, что время, затраченное на выполнение всех задач и (общая сложность задач/количество процессов) почти не различаются, что показывает равномерное распределение задач.

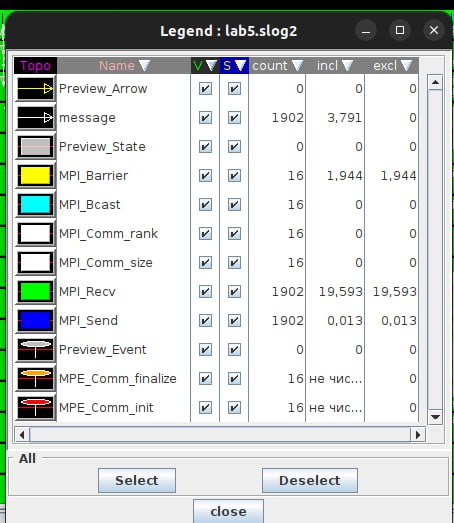
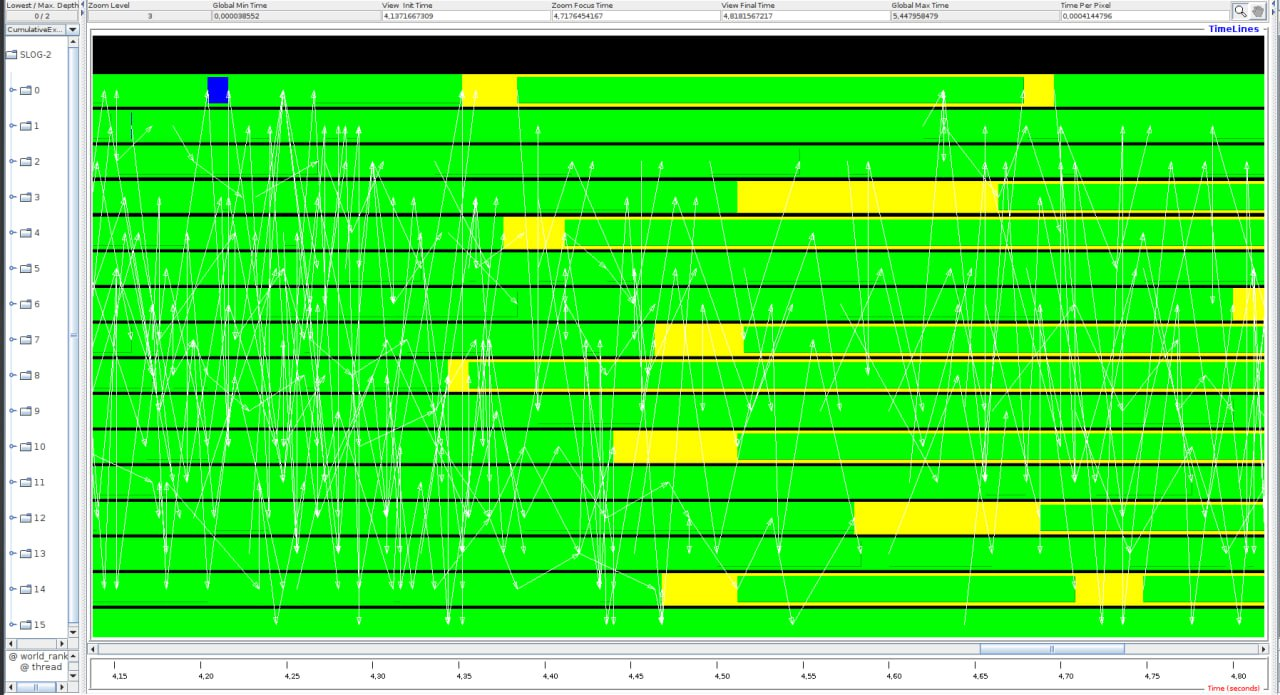




1. **Профилирование**

Общий вид профилирования:

Профилирование в конце вычислений. Не трудно заметить что из-за того что последние задачи самые время затратные, некоторые процессы начинают ожидать завершения других задач.



1. **Заключение**

В данной работе было представлено решение прикладной задачи с использованием параллельных потоков и MPI для взаимодействия между процессами. Каждый процесс выполняет некоторое количество задач, при этом при необходимости запрашивая задачи у других процессов. Использование мьютекса обеспечивает синхронизацию доступа к общим данным.

Результаты выполнения задач и время их выполнения выводятся в консоль. Выводится также информация о суммарной сложности задач, средней сложности на процесс и разнице между ожидаемым и фактическим временем выполнения.

Реализованное решение позволяет эффективно распараллеливать выполнение задач и эффективно использовать вычислительные ресурсы многопроцессорной системы.

1. **Исходники**

#include <stdio.h>

#include <pthread.h>

#include <mpi/mpi.h>

#include <stdlib.h>

#include <time.h>

#include <unistd.h>

#include <math.h>

#define N\_TASKS 320

#define REQUEST\_TAG 0

#define TASK\_TAG 1

typedef struct Task {

    int taskNumber;

    int difficulty;

    char completed;

} Task;

pthread\_mutex\_t mutex;

Task\* task\_list;

void calculate\_task(Task\* task, int rank) {

    pthread\_mutex\_lock(&mutex);

    if (task->completed == 0) {

        // printf("Worker %d: Start task %d(%d)\n", rank, task->taskNumber, task->difficulty);

        task->completed = 1;

        pthread\_mutex\_unlock(&mutex);

        usleep(task->difficulty); //time in mcs

    }

}

int request\_task(int request\_rank) {

    int task\_id = -1;

    int request = 1;

    MPI\_Send(&request, 1, MPI\_INT, request\_rank, REQUEST\_TAG, MPI\_COMM\_WORLD);

    MPI\_Recv(&task\_id, 1, MPI\_INT, request\_rank, TASK\_TAG, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

    return task\_id;

}

void\* worker\_thread(void\* args) {

    int rank = \*((int\*)args);

    int size = \*((int\*)args + 1);

    while (1) {

        Task\* task\_to\_do = NULL;

        for (int i = rank \* N\_TASKS / size; i <  rank \* N\_TASKS / size + N\_TASKS / size; ++i) {

            pthread\_mutex\_lock(&mutex);

            if (!task\_list[i].completed) {

                task\_to\_do = &task\_list[i];

                pthread\_mutex\_unlock(&mutex);

                break;

            }

            pthread\_mutex\_unlock(&mutex);

        }

        if (task\_to\_do == NULL) {

            // printf("Worker %d: local tasks complete, start request...\n", rank);

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

                if (i != rank) {

                    // printf("Worker %d: connect to %d\n", rank, i);

                    int task\_id = request\_task(i);

                    if (task\_id != -1) {

                        task\_to\_do = &task\_list[task\_id];

                        break;

                    }

                }

            }

            if (task\_to\_do == NULL) {

                // printf("Worker %d: BARRIER\n", rank);

                MPI\_Barrier(MPI\_COMM\_WORLD);

                MPI\_Send(NULL, 0, MPI\_INT, rank, REQUEST\_TAG, MPI\_COMM\_WORLD);

                break;

            }

        }

        calculate\_task(task\_to\_do, rank);

    }

    return NULL;

}

void\* server\_thread(void\* args) {

    int rank = \*((int\*)args);

    int size = \*((int\*)args + 1);

    int request;

    MPI\_Status status;

    while (1) {

        MPI\_Recv(&request, 1, MPI\_INT, MPI\_ANY\_SOURCE, REQUEST\_TAG, MPI\_COMM\_WORLD, &status);

        // printf("Server %d: %d connected\n", rank, status.MPI\_SOURCE);

        if (status.MPI\_SOURCE == rank) {

            break;

        }

        int task\_id = -1;

        for (int i = rank \* N\_TASKS / size; i <  rank \* N\_TASKS / size + N\_TASKS / size; ++i) {

            pthread\_mutex\_lock(&mutex);

            if (task\_list[i].completed == 0) {

                task\_id = i;

                task\_list[i].completed = 1;

                pthread\_mutex\_unlock(&mutex);

                break;

            }

            pthread\_mutex\_unlock(&mutex);

        }

        // printf("Server %d: send %d to %d\n", rank, task\_id, status.MPI\_SOURCE);

        MPI\_Send(&task\_id, 1, MPI\_INT, status.MPI\_SOURCE, TASK\_TAG, MPI\_COMM\_WORLD);

    }

    return NULL;

}

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

    int rank, size;

    int provided;

    pthread\_attr\_t attrs;

    pthread\_t threads[2];

    double start\_time, end\_time;

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

    if (provided != MPI\_THREAD\_MULTIPLE)

    {

        printf("Can't init thread\n");

        MPI\_Finalize();

        return -1;

    }

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

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

    int check = 0;

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

    if (check != 0)

    {

        printf("Can't init mutex\n");

        MPI\_Finalize();

        return -1;

    }

    check = pthread\_attr\_init(&attrs);

    if (check != 0)

    {

        printf("Can't init attrs\n");

        MPI\_Finalize();

        return -1;

    }

    check = pthread\_attr\_setdetachstate(&attrs, PTHREAD\_CREATE\_JOINABLE);

    if (check != 0)

    {

        printf("Can't set attrs\n");

        MPI\_Finalize();

        return -1;

    }

    task\_list = (Task\*)malloc(sizeof(Task) \* N\_TASKS);

    if (rank == 0) {

        srand(21212);

        for (int i = 0; i < N\_TASKS; ++i) {

            task\_list[i].taskNumber = i;

            task\_list[i].difficulty = (i + 1) \* 1000;

            task\_list[i].completed = 0;

        }

    }

    MPI\_Bcast(task\_list, N\_TASKS\*sizeof(Task), MPI\_BYTE, 0, MPI\_COMM\_WORLD);

    int\* threadArgs = (int\*)malloc(sizeof(int) \* 2);

    threadArgs[0] = rank;

    threadArgs[1] = size;

    pthread\_create(&threads[0], &attrs, worker\_thread, (void\*)threadArgs);

    pthread\_create(&threads[1], &attrs, server\_thread, (void\*)threadArgs);

    start\_time = MPI\_Wtime();

    pthread\_join(threads[0], NULL);

    pthread\_join(threads[1], NULL);

    end\_time = MPI\_Wtime();

    if (rank == 0) {

        printf("Tasks completed! Time left: %lf\n", end\_time - start\_time);

        int sumDifficulties = 0;

        for (int i = 0; i < N\_TASKS; ++i) {

            sumDifficulties += task\_list[i].difficulty;

        }

        printf("All tasks difficulty: %lf\n", sumDifficulties / 1000000.0);

        printf("All difficulty / N: %lf\n", sumDifficulties / 1000000.0 / size);

        printf("Difference: %lf\n", fabs(sumDifficulties / 1000000.0 / size - end\_time + start\_time));

    }

    free(threadArgs);

    free(task\_list);

    pthread\_mutex\_destroy(&mutex);

    pthread\_attr\_destroy(&attrs);

    MPI\_Finalize();

    return 0;

}