###### 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, 20204 группы

**Дронова Дениса Юрьевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

А. Ю. Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[«Программирование многопоточных приложений. POSIX Threads.» 1](#_Toc104628154)

[ЦЕЛЬ 3](#_Toc104628155)

[ЗАДАНИЕ 3](#_Toc104628156)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc104628157)

[ЗАКЛЮЧЕНИЕ 5](#_Toc104628158)

[Приложение 1. Код программы 6](#_Toc104628159)

# ЦЕЛЬ

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

# ЗАДАНИЕ

1. Реализовать балансировщик нагрузки для выполнения несколькими процессами некоторого списка задач.
2. Каждый процесс должен выполняться в несколько потоков, где один поток будет получать новые задачи, а другой – выполнять их.
3. Программа не должна зависеть от числа компьютеров.
4. Оценить эффективность распределения работы с различным числом задач на каждом процессе, различным весом задач.

# ОПИСАНИЕ РАБОТЫ

Реализована программа с использованием MPI и POSIX Threads, с использованием следующего подхода: создание на каждом процессе потока, отвечающего за распределение задач и основного потока, исполняющего их. Каждый процесс при выполнении своих задач опрашивает остальные на предмет наличия недовыполненных. В качестве задачи взял выполнение подсчета суммы синусов чисел. Были подобраны параметры так, чтобы время исполнения не было меньше 30 секунд, itersCount = 5. Измерены времена выполнения программы и распределение задач между процессами с различными параметрами: много маленьких задач и немного сложных задач, но так, что общий объем работы остается неизменным. На основе измерений построил таблицу.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Weight | Tasks per process | Total time | Disbalance(%, sec.) | |
| 1 | 10000000 | 55 | 7% | 0,9 |
| 1% | 0,1 |
| 0,40% | 0,03 |
| 3% | 0,4 |
| 8% | 1 |
| 100 | 100000 | 46 | 7,80% | 0,85 |
| 19% | 1,45 |
| 0,62% | 0,04 |
| 0,51% | 0,05 |
| 7% | 0,81 |
| 1000 | 10000 | 45 | 0,10% | 0,01 |
| 1,20% | 0,08 |
| 0,24% | 0,01 |
| 4% | 0,45 |
| 0,34% | 0,036 |
| 1000 | 10000 | 46 | 7% | 0,84 |
| 7% | 0,52 |
| 0,57% | 0,04 |
| 6% | 0,7 |
| 7% | 0,79 |
| 100000 | 100 | 62 | 19% | 2 |
| 26% | 1,82 |
| 15% | 1,9 |
| 8% | 1,45 |
| 13% | 2,15 |

# **ЗАКЛЮЧЕНИЕ**

После анализа измерений становится ясно, что при очень большом весе задач и малом их количестве на каждом процессе доля и время дисбаланса самые высокие, а также программа работает дольше. При очень малом весе задач и большом их количестве на каждом процессе также наблюдаются большие величины времени и доли дисбаланса. Можно сделать вывод, что эти два параметра должны быть сбалансированы между собой для того, чтобы можно было наблюдать хорошую эффективность. Также наблюдалось, что при итерировании по itersCount сначала наиболее легкие задачи получал первый процесс, а последний – самые сложные, что в последующем менялось в обратную сторону. Из-за того, что процесс с самыми легкими задачами заканчивал их исполнение быстрее остальных, он больше всех обращался к другим процессам за добавочными. Также сделан вывод о том, что наиболее эффективным способом отдачи процессом задач является деление оставшихся пополам.

# **Приложение 1. Код программы**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <math.h>

#define WEIGHT\_FACTOR 10000

#define TASKS\_PER\_PROCESS 1000

#define MIN\_TASKS\_TO\_SHARE 10

#define ITERATIONS\_COUNT 5

#define REQUEST 0

#define REPLY 1

double global\_result = 0.0;

double global\_result\_sum = 0.0;

int procRank, procSize;

int \*tasks;

int tasks\_remaining;

pthread\_mutex\_t mutex\_tasks;

pthread\_mutex\_t mutex\_tasks\_remaining;

pthread\_t thread\_receiver;

void setTasksWeight(int \*tasks, int count, int iterCounter)

{

    pthread\_mutex\_lock(&mutex\_tasks);

    for (int i = 0; i < count; ++i)

    {

        tasks[i] = abs(50 - i % 100) \* abs(procRank - (iterCounter % procSize)) \* WEIGHT\_FACTOR;

    }

    pthread\_mutex\_unlock(&mutex\_tasks);

}

void completeTasks(int \*tasks, int \*tasks\_executed)

{

    pthread\_mutex\_lock(&mutex\_tasks\_remaining);

    for (int i = 0; tasks\_remaining; ++i, --tasks\_remaining)

    {

        pthread\_mutex\_unlock(&mutex\_tasks\_remaining);

        pthread\_mutex\_lock(&mutex\_tasks);

        int task\_weight = tasks[i];

        pthread\_mutex\_unlock(&mutex\_tasks);

        for (int j = 0; j < task\_weight; ++j)

        {

            global\_result += sin(j);

        }

        ++(\*tasks\_executed);

        pthread\_mutex\_lock(&mutex\_tasks\_remaining);

    }

    pthread\_mutex\_unlock(&mutex\_tasks\_remaining);

}

void \*receiverThreadStart(void \*args)

{

    int tasks\_to\_share;

    int rankRequestedTasks;

    while (true)

    {

        // receiving process rank that requests tasks

        MPI\_Recv(&rankRequestedTasks, 1, MPI\_INT, MPI\_ANY\_SOURCE, REQUEST, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        if (rankRequestedTasks == procRank)

            break;

        pthread\_mutex\_lock(&mutex\_tasks\_remaining);

        if (tasks\_remaining >= MIN\_TASKS\_TO\_SHARE)

        {

            tasks\_to\_share = tasks\_remaining / 2;

            tasks\_remaining -= tasks\_to\_share;

            // sending number of tasks that shares

            MPI\_Send(&tasks\_to\_share, 1, MPI\_INT, rankRequestedTasks, REPLY, MPI\_COMM\_WORLD);

            pthread\_mutex\_lock(&mutex\_tasks);

            // sending tasks

            MPI\_Send(tasks + tasks\_remaining - 1, tasks\_to\_share, MPI\_INT, rankRequestedTasks, REPLY, MPI\_COMM\_WORLD);

            pthread\_mutex\_unlock(&mutex\_tasks);

        }

        else

        {

            tasks\_to\_share = 0;

            MPI\_Send(&tasks\_to\_share, 1, MPI\_INT, rankRequestedTasks, REPLY, MPI\_COMM\_WORLD);

        }

        pthread\_mutex\_unlock(&mutex\_tasks\_remaining);

    }

    return NULL;

}

void \*workerThreadStart(void \*args)

{

    tasks = (int \*)malloc(sizeof(int) \* TASKS\_PER\_PROCESS);

    double start\_time, total\_time;

    double minTime, maxTime;

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

    {

        setTasksWeight(tasks, TASKS\_PER\_PROCESS, iterCounter);

        pthread\_mutex\_lock(&mutex\_tasks\_remaining);

        tasks\_remaining = TASKS\_PER\_PROCESS;

        pthread\_mutex\_unlock(&mutex\_tasks\_remaining);

        int tasks\_executed = 0;

        int tasks\_received;

        MPI\_Barrier(MPI\_COMM\_WORLD);

        start\_time = MPI\_Wtime();

        completeTasks(tasks, &tasks\_executed);

        // Start requesting tasks from other processes

        //(sequental iterating)

        for (int rank\_iter = 0; rank\_iter < procSize; ++rank\_iter)

        {

            if (rank\_iter == procRank)

                continue;

            // Notificating other process that current process is free

            // and want to execute extra tasks (sending other process current procRank)

            MPI\_Send(&procRank, 1, MPI\_INT, rank\_iter, REQUEST, MPI\_COMM\_WORLD);

            // Receive number of extra tasks

            MPI\_Recv(&tasks\_received, 1, MPI\_INT, rank\_iter, REPLY, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            if (tasks\_received > 0)

            {

                // Receive tasks

                MPI\_Recv(tasks, tasks\_received, MPI\_INT, rank\_iter, REPLY, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

                pthread\_mutex\_lock(&mutex\_tasks\_remaining);

                tasks\_remaining = tasks\_received;

                pthread\_mutex\_unlock(&mutex\_tasks\_remaining);

                completeTasks(tasks, &tasks\_executed);

            }

        }

        total\_time = MPI\_Wtime() - start\_time;

        MPI\_Allreduce(&total\_time, &minTime, 1, MPI\_DOUBLE, MPI\_MIN, MPI\_COMM\_WORLD);

        MPI\_Allreduce(&total\_time, &maxTime, 1, MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

        // Print info about this iteration

        if (procRank == 0)

        {

            printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

            printf("Iteration %d\n", iterCounter + 1);

            printf("Disbalance time: %lf sec.\n", maxTime - minTime);

            printf("Disbalance percentage: %lf %%\n", (maxTime - minTime) / maxTime \* 100);

        }

        MPI\_Barrier(MPI\_COMM\_WORLD);

        for (int rank\_iter = 0; rank\_iter < procSize; rank\_iter++)

        {

            if (procRank == rank\_iter)

            {

                printf("\tProcess #%d\n", procRank + 1);

                printf("\t\texecuted tasks:\t%d\n", tasks\_executed);

                printf("\t\tglobal result:\t%lf\n", global\_result);

                printf("\t\titeration time:\t%lf sec.\n", total\_time);

            }

            MPI\_Barrier(MPI\_COMM\_WORLD);

        }

    }

    // Terminate Thread 'Receiver'

    MPI\_Send(&procRank, 1, MPI\_INT, procRank, 0, MPI\_COMM\_WORLD);

    MPI\_Allreduce(&global\_result, &global\_result\_sum, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

    free(tasks);

    return NULL;

}

void createThreads()

{

    pthread\_attr\_t attr;

    pthread\_attr\_init(&attr);

    pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE);

    // creating joinable thread for receiving requests from worker thread

    pthread\_create(&thread\_receiver, &attr, receiverThreadStart, NULL);

    pthread\_attr\_destroy(&attr);

    // starting work on main thread

    workerThreadStart(NULL);

    // waitng for recieiver thread to finish

    pthread\_join(thread\_receiver, NULL);

}

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

{

    int required = MPI\_THREAD\_MULTIPLE;

    int provided;

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

    if (provided != required)

    {

        fprintf(stderr, "Provided level doesn't match the required!\n");

        MPI\_Finalize();

        return EXIT\_FAILURE;

    }

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

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

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

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

    MPI\_Barrier(MPI\_COMM\_WORLD);

    double start\_time = MPI\_Wtime();

    createThreads();

    double total\_time = MPI\_Wtime() - start\_time;

    pthread\_mutex\_destroy(&mutex\_tasks);

    pthread\_mutex\_destroy(&mutex\_tasks\_remaining);

    if (procRank == 0)

    {

        printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

        printf("Total time:\t%lf sec.\n", total\_time);

        printf("Global result sum:\t%lf\n", global\_result\_sum);

        printf("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

    }

    MPI\_Finalize();

    return EXIT\_SUCCESS;

}