**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 с помощью POSIX Threads»

Студента 2 курса, 21211 группы

**Петрова Сергея Евгеньевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Антон Юрьевич Кудинов

**СОДЕРЖАНИЕ**

## ЦЕЛЬ

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

## ЗАДАНИЕ

*Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Необходимо организовать параллельную обработку заданий на нескольких компьютерах.*

## ТРЕБОВАНИЯ

- *Задания могут иметь различный вычислительный вес;*
- *Считается, что вес задания нельзя узнать, пока оно не выполнено;*
- *После того, как все задания из списка выполнены, появляется новый список заданий;*
- *Количество заданий существенно превосходит количество процессоров;*
- *Программа не должна зависеть от числа компьютеров;*
- *Для организации взаимодействия между компьютерами использовать MPI.*
- *Для организации потоков использовать POSIX Threads.*
- *Программа должна содержать 3 потока:*
    - *Поток, который выполняет задания*
    - *Поток, который запрашивает задания у других компьютеров, когда задания в очереди закончились;*
    - *Поток, который отправляет задания по запросам других компьютеров;*

## ОПИСАНИЕ РАБОТЫ

### Описание выполненной работы

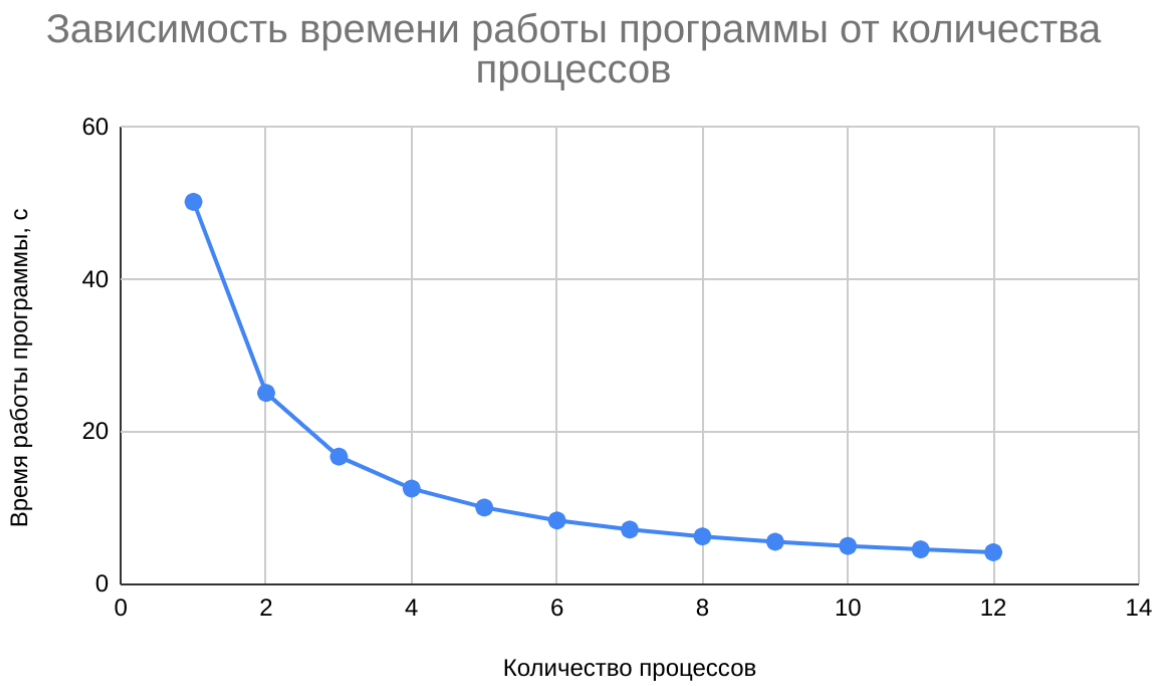
1. *Написал программу, которая организовывает параллельную обработку заданий на нескольких компьютерах;*
2. *Реализовал инициализацию очереди задач. Веса всех задач на узле пропорциональны номеру компьютера в коммуникаторе по умолчанию. Суммарный вес всех задач со всех узлов - 50 секунд;*
3. *Запустил программу при использовании 1-12 ядер;*
4. *Составил графики зависимости времени работы, ускорения работы и эффективности распараллеливания программы от количества MPI процессов;*

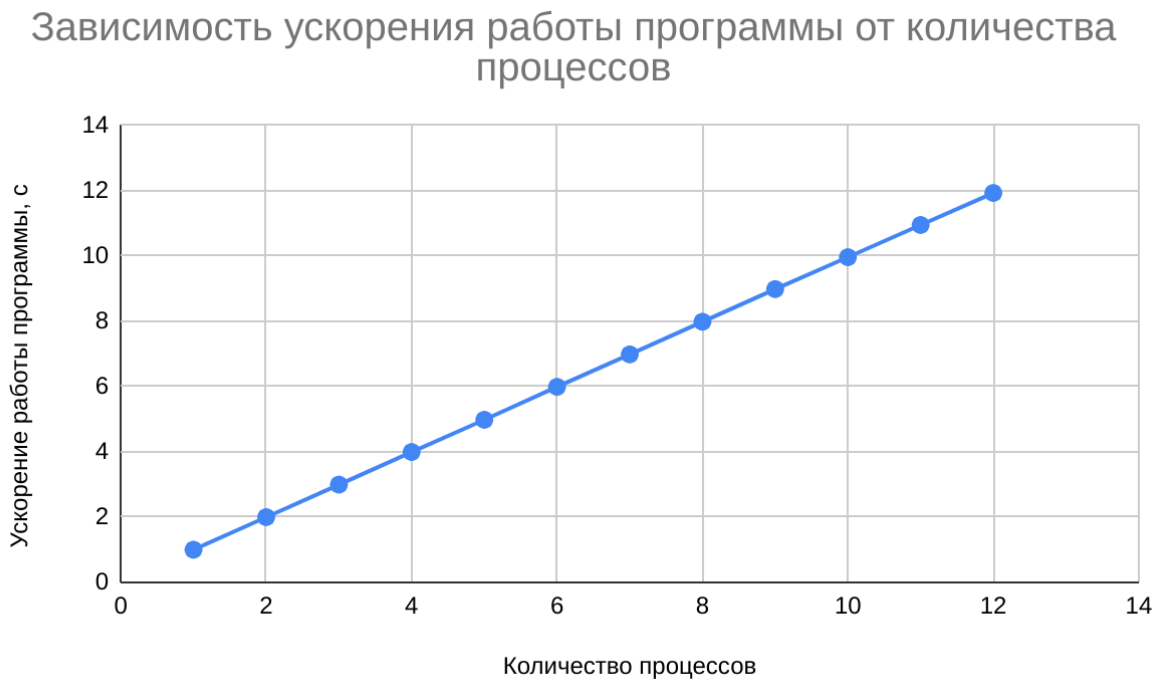
### Команды для компиляции и запуска

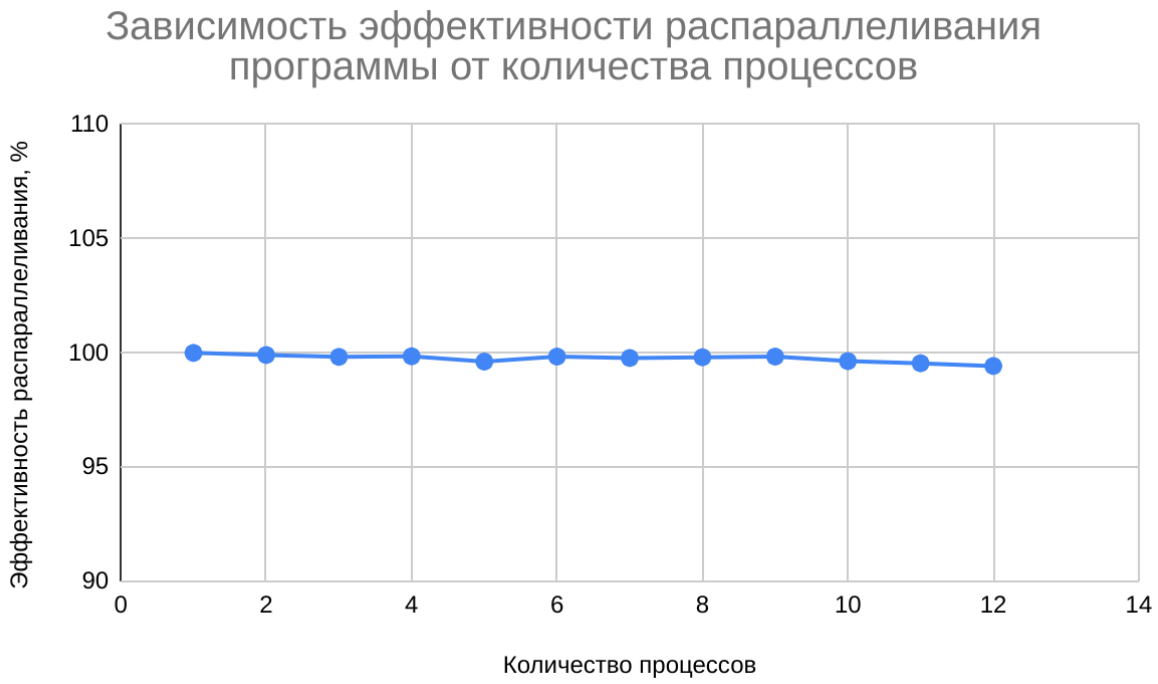
*Команда для компиляции и запуска MPI программы*

```
opp@comrade:~/211/s.petrov1/lab5$ ./build.sh
opp@comrade:~/211/s.petrov1/lab5$ ./run.sh <MPI process count>
```

**Результаты измерения**




Зависимость эффективности распараллеливания программы от количества процессов
Эффективность распараллеливания, %
110
105
100
95
90
0
2
4
6
8
10
12
14
Количество процессов

## ЗАКЛЮЧЕНИЕ

*В ходе выполнения лабораторной работы:*

- *Освоил разработку многопоточных программ с использованием POSIX Threads API;*
- *Познакомился с задачей динамического распределения работы между процессорами;*

*Анализируя результаты исследований, можно сделать следующие выводы:*

- *Благодаря пересылке задач время работы программы оценивается средним значением суммарного веса задач на отдельном узле, а не максимальным значением суммарного веса задач на отдельном узле;*

## ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

*color.h*

```
#ifndef COLOR_H
#define COLOR_H

#define FNORM       "\x1B[0m"
#define FBLACK      "\x1B[30m"
#define FRED        "\x1B[31m"
#define FGREEN      "\x1B[32m"
#define FYELLOW     "\x1B[33m"
#define FBLUE       "\x1B[34m"
#define FMAGENTA    "\x1B[35m"
#define FCYAN       "\x1B[36m"
#define FWHITE      "\x1B[37m"

#endif // COLOR_H
```

*task_queue.h*

```
#ifndef TASK_QUEUE_H
#define TASK_QUEUE_H

#include <stdbool.h>

#define SUCCESS 0
#define ERROR   (-1)

struct task_t {
    int id;
    int process_id;
    int weight;
};

struct task_queue_t;

struct task_queue_t *task_queue_create(int capacity);
bool task_queue_is_empty(const struct task_queue_t *queue);
bool task_queue_is_full(const struct task_queue_t *queue);
int task_queue_push(struct task_queue_t *queue, struct task_t task);
int task_queue_pop(struct task_queue_t *queue, struct task_t *task);
void task_queue_destroy(struct task_queue_t **queue);

#endif // TASK_QUEUE_H
```

*task_queue.c*

```
#include "task_queue.h"
#include <stdlib.h>

struct task_queue_t {
    struct task_t *data;
    int capacity;
    int count;
    int pop_index;
};
```

```c
struct task_queue_t *task_queue_create(int capacity) {
    struct task_queue_t *queue = malloc(sizeof(struct task_queue_t));
    if (queue == NULL) {
        return NULL;
    }

    struct task_t *data = malloc(sizeof(struct task_t) * capacity);
    if (data == NULL) {
        return NULL;
    }

    queue->data = data;
    queue->capacity = capacity;
    queue->count = 0;
    queue->pop_index = 0;

    return queue;
}

bool task_queue_is_empty(const struct task_queue_t *queue) {
    return queue->count == 0;
}

bool task_queue_is_full(const struct task_queue_t *queue) {
    return queue->count == queue->capacity;
}

int task_queue_push(struct task_queue_t *queue, struct task_t task) {
    if (queue == NULL) {
        return ERROR;
    }

    if (task_queue_is_full(queue)) {
        return ERROR;
    }

    int push_index = (queue->pop_index + queue->count) % queue->capacity;
    queue->data[push_index] = task;
    queue->count++;

    return SUCCESS;
}

int task_queue_pop(struct task_queue_t *queue, struct task_t *task) {
    if (queue == NULL) {
        return ERROR;
    }

    if (task_queue_is_empty(queue)) {
        return ERROR;
    }

    *task = queue->data[queue->pop_index];
    queue->pop_index = (queue->pop_index + 1) % queue->capacity;
    queue->count--;

    return SUCCESS;
}
```

```
void task_queue_destroy(struct task_queue_t **queue) {
    if (*queue == NULL) {
        return;
    }

    if ((*queue)->data == NULL) {
        return;
    }

    free((*queue)->data);
    free(*queue);

    *queue = NULL;
}
```

*main.c*

```
#include <mpich/mpi.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "color.h"
#include "task_queue.h"

#define TASK_COUNT              2000
#define REQUEST_TAG             0
#define RESPONSE_TAG            1
#define EMPTY_QUEUE_RESPONSE    (-1)
#define TERMINATION_SIGNAL      (-2)

int process_count;
int process_id;
int proc_sum_weight = 0;
bool termination = false;
struct task_queue_t *task_queue;

pthread_mutex_t mutex;
pthread_cond_t worker_cond;
pthread_cond_t receiver_cond;

void *worker_start(void *args);
void *receiver_start(void *args);
void *sender_start(void *args);

int main(int argc, char *argv[]) {
    int required = MPI_THREAD_MULTIPLE;
    int provided;
    double start_time;
    double end_time;
    pthread_t worker_thread;
    pthread_t receiver_thread;
    pthread_t sender_thread;

    // Initialize MPI environment
    MPI_Init_thread(&argc, &argv, required, &provided);
```

```
    if (provided != required) {
        return EXIT_FAILURE;
    }
    MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
    MPI_Comm_size(MPI_COMM_WORLD, &process_count);

    // Create task queue
    task_queue = task_queue_create(TASK_COUNT);

    // Initialize mutex and condition variable
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&worker_cond, NULL);
    pthread_cond_init(&receiver_cond, NULL);

    // Start worker and sender thread
    start_time = MPI_Wtime();
    pthread_create(&worker_thread, NULL, worker_start, NULL);
    pthread_create(&receiver_thread, NULL, receiver_start, NULL);
    pthread_create(&sender_thread, NULL, sender_start, NULL);

    pthread_join(worker_thread, NULL);
    pthread_join(receiver_thread, NULL);
    pthread_join(sender_thread, NULL);
    end_time = MPI_Wtime();

    // Print result
    MPI_Barrier(MPI_COMM_WORLD);
    printf(FGREEN"Summary weight %d: %lf\n"FNORM,
           process_id,
           proc_sum_weight * 1E-6);
    MPI_Barrier(MPI_COMM_WORLD);
    if (process_id == 0) {
        printf(FGREEN"Time: %lf\n"FNORM, end_time - start_time);
    }

    // Clear resources
    task_queue_destroy(&task_queue);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&worker_cond);
    pthread_cond_destroy(&receiver_cond);
    MPI_Finalize();

    return EXIT_SUCCESS;
}

static inline void init_tasks() {
    // Total sum of task weights does not change
    // For each process, tasks have a weight: min_weight * (process_id + 1)
    // min_weight = (TOTAL_SUM_WEIGHT * n) / (TASK_COUNT * S_n)
    // n - process count
    // S_n - sum of an arithmetic progression 1, 2, ... , n

    const int TOTAL_SUM_WEIGHT = 50000000;
    int min_weight = 2 * TOTAL_SUM_WEIGHT / (TASK_COUNT * (process_count + 1));
    int task_id = 1;

    for (int i = 0; i < TASK_COUNT; ++i) {
        // Create task
        struct task_t task = {
```

```
            .id = task_id,
            .process_id = process_id,
            .weight = min_weight * (i % process_count + 1)
        };

        if (i % process_count == process_id) {
            task_queue_push(task_queue, task);
            task_id++;
            proc_sum_weight += task.weight;
        }
    }
}

static inline void execute_tasks() {
    while (true) {
        struct task_t task;

        pthread_mutex_lock(&mutex);
        if (task_queue_is_empty(task_queue)) {
            pthread_mutex_unlock(&mutex);
            break;
        }
        task_queue_pop(task_queue, &task);
        pthread_mutex_unlock(&mutex);

        printf(FBLUE"Worker %d executing task %d of process %d"
                    " and weight %d\n"FNORM,
               process_id,
               task.id,
               task.process_id,
               task.weight);
        usleep(task.weight);
    }
}

void *worker_start(void *args) {
    init_tasks();

    // Worker start synchronization
    MPI_Barrier(MPI_COMM_WORLD);

    while (true) {
        execute_tasks();

        pthread_mutex_lock(&mutex);
        while (task_queue_is_empty(task_queue) && !termination) {
            pthread_cond_signal(&receiver_cond);
            pthread_cond_wait(&worker_cond, &mutex);
        }

        if (termination) {
            pthread_mutex_unlock(&mutex);
            break;
        }
        pthread_mutex_unlock(&mutex);
    }

    printf(FBLUE"Worker %d finished\n"FNORM, process_id);
    pthread_exit(NULL);
```

```
}

void *receiver_start(void *args) {
    int termination_signal = TERMINATION_SIGNAL;

    while (!termination) {
        int received_tasks = 0;
        struct task_t task;

        pthread_mutex_lock(&mutex);
        while (!task_queue_is_empty(task_queue)) {
            pthread_cond_wait(&receiver_cond, &mutex);
        }
        pthread_mutex_unlock(&mutex);

        for (int i = 0; i < process_count; ++i) {
            if (i == process_id) {
                continue;
            }

            printf(FYELLOW"Receiver %d sent request to process %d\n"FNORM,
                   process_id,
                   i);
            MPI_Send(&process_id, 1, MPI_INT, i, REQUEST_TAG, MPI_COMM_WORLD);
            MPI_Recv(&task,
                     sizeof(task),
                     MPI_BYTE,
                     i,
                     RESPONSE_TAG,
                     MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);

            if (task.id != EMPTY_QUEUE_RESPONSE) {
                printf(FYELLOW"Receiver %d received task %d from process %d\n"FNORM,
                       process_id,
                       task.id,
                       i);

                pthread_mutex_lock(&mutex);
                task_queue_push(task_queue, task);
                pthread_mutex_unlock(&mutex);

                received_tasks++;
            } else {
                printf(FYELLOW"Receiver %d received empty queue response "
                              "from process %d\n"FNORM,
                       process_id,
                       i);
            }
        }

        if (received_tasks == 0) {
            pthread_mutex_lock(&mutex);
            termination = true;
            pthread_mutex_unlock(&mutex);
        }

        pthread_mutex_lock(&mutex);
        pthread_cond_signal(&worker_cond);
```

```
        pthread_mutex_unlock(&mutex);
    }

    // Receiver destruction synchronization
    MPI_Barrier(MPI_COMM_WORLD);

    printf(FYELLOW"Receiver %d sent termination signal\n"FNORM, process_id);
    MPI_Send(&termination_signal,
             1,
             MPI_INT,
             process_id,
             REQUEST_TAG,
             MPI_COMM_WORLD);

    printf(FYELLOW"Receiver %d finished\n"FNORM, process_id);
    pthread_exit(NULL);
}

void *sender_start(void *args) {
    while (true) {
        int receive_process_id;
        struct task_t task;

        printf(FMAGENTA"Sender %d waiting for request\n"FNORM, process_id);
        MPI_Recv(&receive_process_id,
                 1,
                 MPI_INT,
                 MPI_ANY_SOURCE,
                 REQUEST_TAG,
                 MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

        if (receive_process_id == TERMINATION_SIGNAL) {
            printf(FMAGENTA"Sender %d received termination signal\n"FNORM,
                   process_id);
            break;
        }

        printf(FMAGENTA"Sender %d received request from process %d\n"FNORM,
               process_id,
               receive_process_id);

        pthread_mutex_lock(&mutex);
        if (!task_queue_is_empty(task_queue)) {
            task_queue_pop(task_queue, &task);
            printf(FMAGENTA"Sender %d sent task %d of process %d "
                           "to process %d\n"FNORM,
                   process_id,
                   task.id,
                   task.process_id,
                   receive_process_id);
        } else {
            task.id = EMPTY_QUEUE_RESPONSE;
            task.weight = 0;
            task.process_id = process_id;
            printf(FMAGENTA"Sender %d sent empty queue response "
                           "to process %d\n"FNORM,
                   process_id,
                   receive_process_id);
```

```
        }
        pthread_mutex_unlock(&mutex);

        MPI_Send(&task,
                 sizeof(task),
                 MPI_BYTE,
                 receive_process_id,
                 RESPONSE_TAG,
                 MPI_COMM_WORLD);
    }

    printf(FMAGENTA"Sender %d finished\n"FNORM, process_id);
    pthread_exit(NULL);
}
```

*CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.16.3)

project(cluster C)

find_package(MPI REQUIRED)
find_package(Threads REQUIRED)

add_library(task_queue STATIC src/task_queue.c)
target_include_directories(task_queue
        PUBLIC include)

add_executable(cluster src/main.c)
target_include_directories(cluster
        PUBLIC include
        PUBLIC ${MPI_C_INCLUDE_DIRS})
target_link_libraries(cluster
        PUBLIC task_queue
        PUBLIC MPI::MPI_C
        PUBLIC Threads::Threads)
```

*build.sh*

```
#!/bin/bash

cmake -B build -S ./
cmake --build build --target cluster
```

*run.sh*

```
#!/bin/bash

if [ $# -ne 1 ]; then
     echo "USAGE: run.sh <MPI process count>"
     exit 1
fi

mpiexec -np="$1" build/cluster
```

*test.sh*

```
#!/bin/bash

./build.sh

for (( i = 1; i <= 12; ++i ))
do
     echo "MPI process count: $i"
     ./run.sh "$i" > logs/cluster_"$i".log
     echo "Create logs/cluster_$i.log"
done
```