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 ПРАКТИЧЕСКОЙ РАБОТЫ

«Балансировка нагрузки процессов MPI при помощи и POSIX потоков»

студента 2 курса, группы 20205

Муратова Максима Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент
Власенко А. Ю.

Новосибирск, 2022

СОДЕРЖАНИЕ

ЦЕЛИ

1. При помощи потоков POSIX, организовать систему перераспределения нагрузки между процессами MPI;
2. Оценить эффективность системы балансировки нагрузки.

ЗАДАНИЕ

1. Процессы выполняют задачи из списков. Каждая задача неделима и может быть выполнена независимо от других задач, а также обладает весом – количеством времени, необходимого для выполнения задачи. Изначально процессам выдаются заведомо неодинаковые по суммарному весу задачи из списка. Всего списков N, в каждом из них по M задач;
2. Каждая задача – посчитать частичную сумму ряда $S_n = \sum_{k=1}^{n} \sin\sqrt{k}$ при некотором *n*, который является весом этой задачи;
3. Число MPI процессов – 8;
4. i задача в j списке имеет вес $w_{ij} = k + [k * \sqrt[3]{k+i+1}\sqrt{j+1}]$ с учётом того, что и задачи, и списки нумеруются с нуля.
5. N = 8, M = 24 000, k = 500.

ОПИСАНИЕ РАБОТЫ

На каждом процессе есть 3 потока:

1. Поток исполнения, который выполняет задачи;
2. Поток-отправитель, который может делегировать задачи процесса другому процессу по необходимости или отправить информацию о количестве оставшихся задач;
3. Поток-получатель, который принимает задачи и ставит их на выполнение.

Помимо них, есть ещё 2 потока, которые работают только на одном, выделенном процессе:

1. Поток управления, который занимается генерацией, раздачей и перераспределением задач процессам, принимает от процессов сигналы об завершении ими обработки задач;
2. Поток журналирования, который собирает с процессов данные об их времени выполнения и записывает результаты обработки списка: сколько задач было выполнено, общий результат, наибольшее и наименьшее времена обработки списка по каждому процессу отдельно и дисбаланс $D_i = \frac{max(t_{ij}) - min(t_{ik})}{max(t_{ij})} \cdot 100\%$.

На каждом сгенерированном списке поток управления делит его на одинаковые по количеству задач части, назначает эти части соответствующим спискам и обновляет номер списка задач.

Алгоритм генерации списка задач построен так, что первые задачи имеют меньший вес, чем последние, поэтому процессы с меньшим рангом получают более лёгкие задачи, чем процессы с большим рангом. Когда процесс завершает выполнение задач, он уведомляет об этом поток управления. Тот инициирует сбор с информации, сколько задач осталось выполнить каждому процессу, после чего просит у процесса с наибольшим количеством оставшихся задач передать часть своих задач освободившемуся потоку. Обычно передаётся половина от оставшихся зачач.

По окончании обработки списка, каждый процесс фиксирует, сколько задач он фактически выполнил, каков суммарный результат и сколько времени было потрачено на выполнение задач из предоставленной процессу

части списка. Эти данные собираются потоком журналирования, который записывает общие итоги обработки списка задач.

По окончании обработки списков поток управления посылает потокам-отправителям и получателям специальные сообщения, потоки-получатели сигнализируют о завершении работы свои потоки-исполнители, которые подводят итоги всей своей работы. Поток журналирования завершается сразу, потому что он заранее знает, сколько записей ему необходимо сделать.

Диаграммы последовательности взаимодействий потоков описаны в приложении 1.

Для сравнения программа была запущена дважды при одних и тех же условиях: количестве MPI процессов, числе и размере списков, весов заданий, но в первый раз без балансировки нагрузки, а во второй – с балансировкой. В первом случае наибольший дисбаланс составил 43,47%, а во втором – 9,35%. Результаты тестирования доступны в приложении 2.

Исходный код программы (приложение 3) распределён между следующими файлами:

- `main.cpp` – исходный код основной программы;
- `field.h` – заголовочный файл, который содержит описание функций потоков и их аргументы;
- `field.cpp` – реализация функций потоков.

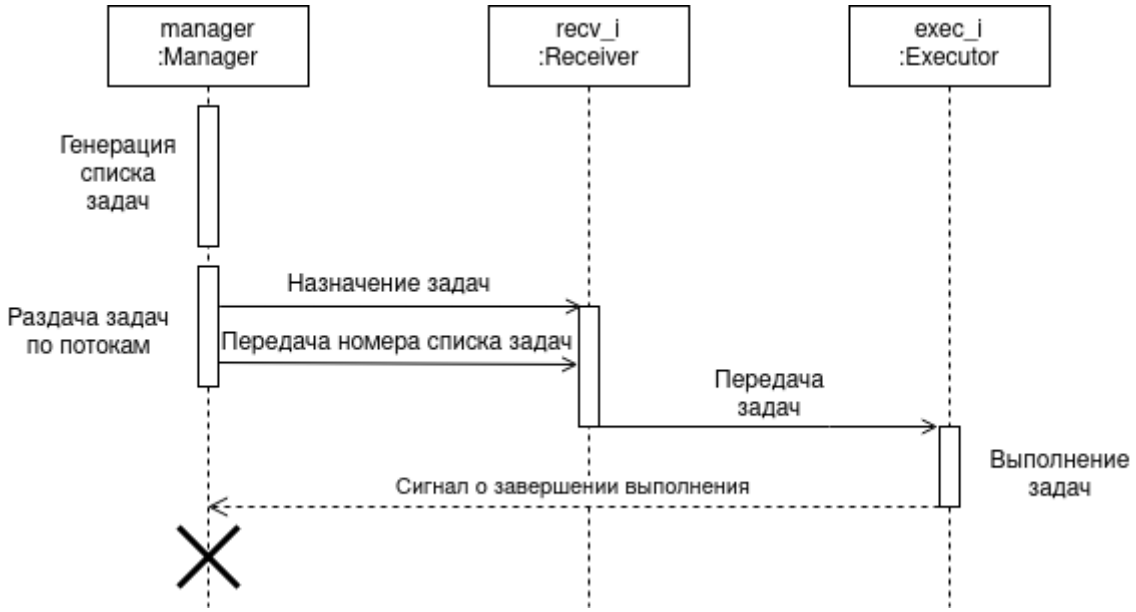
Программа собирается утилитой `make`, конфигурация `Makefile` описана в приложении 4.

ЗАКЛЮЧЕНИЕ

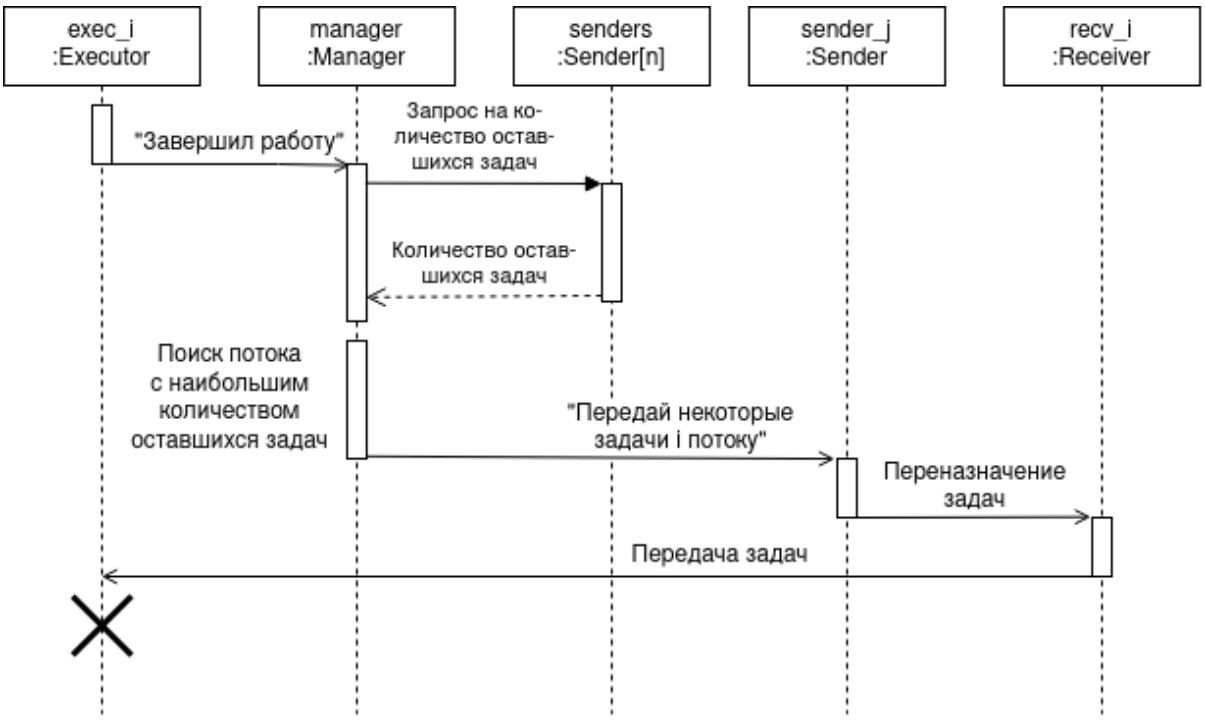
1. Благодаря системе балансировки POSIX-потоками дисбаланс был снижен с 43,47% до 9,35%, то есть более, чем в 4 раза.

Приложение 1. *Диаграммы последовательностей*

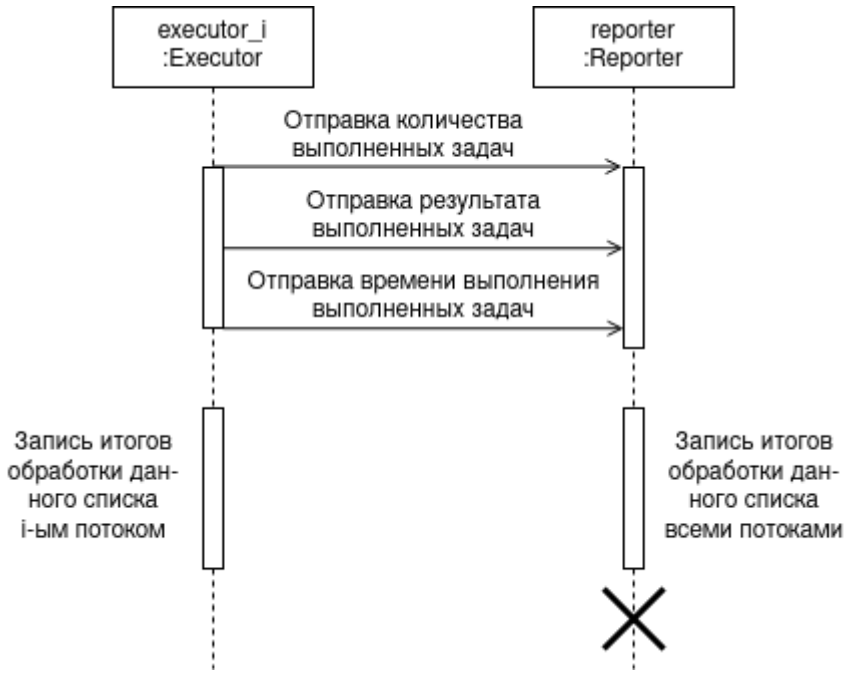
1. Диаграмма первоначальной раздачи заданий для отдельно взятого процесса



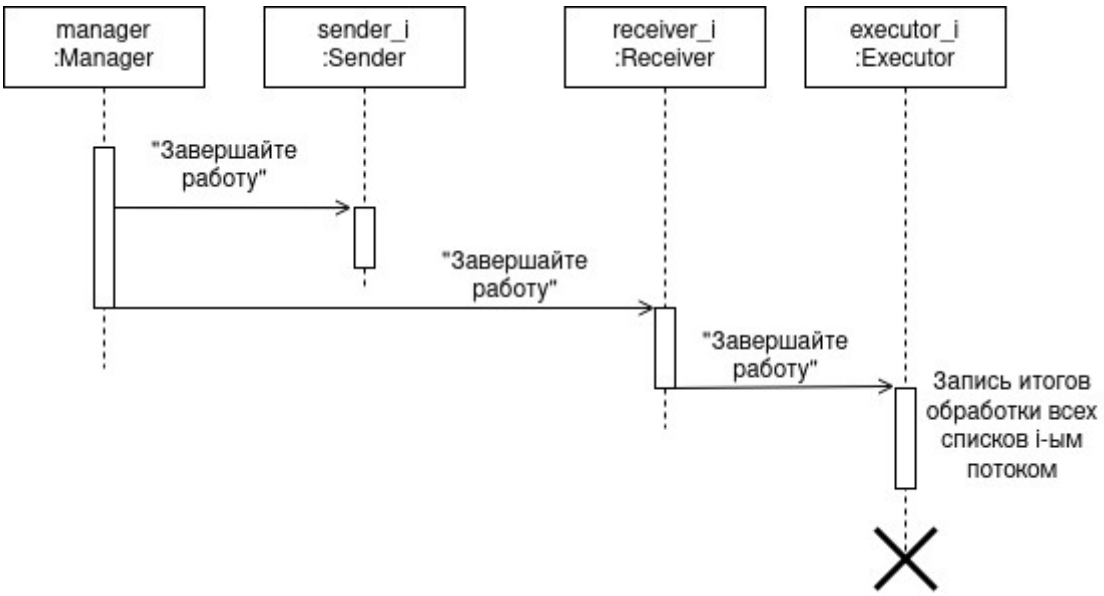
2. Диаграмма перебалансировки нагрузки



3. Диаграммы записи результатов обработки очередного списка



4. Диаграмма завершения работы



Приложение 2. *Результаты тестирования*

Тестирование с балансировкой нагрузки

Номер списка	Общий результат	Наименьшее время, с	Наибольшее время, с	Дисбаланс, %
0	18798.4	2.27574	2.49591	8.82135
1	-166807	3.21928	3.48952	7.74412
2	-15155.5	3.97878	4.22068	5.73141
3	-237226	4.59119	4.79633	4.27697
4	165001	5.09362	5.41732	5.97515
5	-297424	5.58285	6.15927	9.35846
6	-179358	6.00648	6.45144	6.89715
7	-78799.9	6.35063	6.85873	7.40802

Тестирование без балансировки нагрузки

Номер списка	Общий результат	Наименьшее время, с	Наибольшее время, с	Дисбаланс, %
0	18798.4	2.09698	3.67972	43.0126
1	-166807	2.92527	5.14996	43.1982
2	-15155.5	3.54276	6.2608	43.4137
3	-237226	4.09372	7.21896	43.292
4	165001	4.55051	8.04976	43.4702
5	-297424	4.98401	8.81351	43.4503
6	-179358	5.39494	9.49296	43.169
7	-78799.9	5.75706	10.118	43.1006

Приложение 3. *Исходный код программы*

main.cpp

```
#include <pthread.h>
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <string>
#include <signal.h>
#include <unistd.h>
#include "threads.h"

#define CORRECT_ARGC 2

std::ofstream *out, *report;

void interrupt(int signum) {
        if(signum == SIGINT) {
                out->close();
                if(report != NULL) {
                        report->close();
                        delete report;
                }
                abort();
        }
}

int main(int argc, char **argv) {
        if(argc!= CORRECT_ARGC + 1) {
                std::cerr << "Wrong argc " << argc << '\n';
                return 1;
        }
        signal(SIGINT, interrupt);

        int list_count = atoi(argv[1]);
        int list_size = atoi(argv[2]);

        int real;
        MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &real);
        if(real != MPI_THREAD_MULTIPLE) {
                perror("Failed to get thread access");
                MPI_Finalize();
                return 1;
        }

        pthread_attr_t attrs;
        if(pthread_attr_init(&attrs) != 0) {
                perror("Cannot initialize attributes");
```

```
        MPI_Finalize();
        return 1;
};
if(pthread_attr_setdetachstate(&attrs,
     PTHREAD_CREATE_JOINABLE) != 0) {
        perror("Error in setting attributes");
        MPI_Finalize();
        return 1;
}

int process_count, rank;
MPI_Comm_size(MPI_COMM_WORLD, &process_count);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
const int root_rank = process_count / 2;
int max_task_count = list_size * (rank + 1) / process_count -
     list_size * rank / process_count;
task_t *task_buffer = new task_t[max_task_count];

std::ofstream out("process" + std::to_string(rank) + ".txt");
std::ofstream *report = NULL;
if(root_rank == rank)
        report = new std::ofstream("report.txt");
::report = report;
::out = &out;

int current_task_count = 0, done_task_count = 0,
     current_list_number = 0;
bool is_filled = 0;

pthread_mutex_t count_mutex, perform_mutex;
pthread_cond_t working_cond;
pthread_mutex_init(&count_mutex, NULL);
pthread_mutex_init(&perform_mutex, NULL);
pthread_cond_init(&working_cond, NULL);

pthread_t threads[5];

ExecutorArgs exec_args;
ReceiverArgs recv_args;
SenderArgs send_args;
ManagerArgs manager_args;
ReporterArgs reporter_args;

exec_args.task_buffer = recv_args.task_buffer =
     send_args.task_buffer = task_buffer;
exec_args.root_rank = recv_args.root_rank = send_args.root_rank =
     root_rank;
exec_args.current_task_count = recv_args.current_task_count =
        send_args.current_task_count = &current_task_count;
```

```
        exec_args.done_task_count = recv_args.done_task_count =
              send_args.done_task_count = &done_task_count;
        exec_args.current_list_number = recv_args.current_list_number =
              send_args.current_list_number = &current_list_number;
        exec_args.count_mutex = recv_args.count_mutex =
              send_args.count_mutex = &count_mutex;

        exec_args.is_filled = recv_args.is_filled = &is_filled;
        exec_args.perform_mutex = recv_args.perform_mutex =
              &perform_mutex;
        exec_args.cond = recv_args.cond = &working_cond;
        exec_args.out_file = &out;

        manager_args.list_count = reporter_args.list_count = list_count;
        manager_args.process_count = reporter_args.process_count =
              process_count;
        manager_args.list_size = list_size;
        reporter_args.main_report = report;

        pthread_create(threads, &attrs, executor_thread, &exec_args);
        pthread_create(threads + 1, &attrs, receiver_thread, &recv_args);
        pthread_create(threads + 2, &attrs, sender_thread, &send_args);

        if(rank == root_rank) {
                pthread_create(threads + 3, &attrs, manager_thread,
                   &manager_args);
                pthread_create(threads + 4, &attrs, reporter_thread,
                   &reporter_args);
        }

        for(int index = 0; index < 3; index++)
                pthread_join(threads[index], NULL);
        for(int index = 3; root_rank == rank && index < 5; index++) {
                pthread_join(threads[index], NULL);
        }

        pthread_attr_destroy(&attrs);
        pthread_mutex_destroy(&count_mutex);
        pthread_mutex_destroy(&perform_mutex);
        pthread_cond_destroy(&working_cond);

        delete[] task_buffer;
        out.close();
        if(report != NULL)
                report->close();
        delete report;

        MPI_Finalize();
}
```

threads.h

```
#ifndef H_THREADS
#define H_THREADS

#include <pthread.h>
#include <fstream>
#include <cstdlib>

#define ASSIGN_TAG 10
#define UNASSIGN_TAG 11
#define TASK_COUNT_TAG 12
#define TIME_REPORT_TAG 13
#define TASK_PERFORMED_TAG 14
#define LIST_NUMBER_TAG 15
#define REMAIN_COUNT_RESPONCE_TAG 16
#define GLOBAL_RESULT_TAG 17

#define SENDER_TERMINATE -1
#define GET_REMAIN -2

typedef int task_t;

struct PthreadArgs {
        task_t *task_buffer;
        int root_rank;
        int *current_task_count;
        int *done_task_count;
        int *current_list_number;
        pthread_mutex_t *count_mutex;
};

struct ExecutorArgs: PthreadArgs {
        std::ostream *out_file;
        pthread_cond_t *cond;
        bool *is_filled;
        pthread_mutex_t *perform_mutex;
};

struct ReceiverArgs: PthreadArgs {
        bool *is_filled;
        pthread_cond_t *cond;
        pthread_mutex_t *perform_mutex;
};

struct SenderArgs: PthreadArgs {};

struct ManagerArgs {
        int list_count;
```

```
        int list_size;
        int process_count;
};

struct ReporterArgs {
        std::ostream *main_report;
        int list_count;
        int process_count;
};

void generate_task_list(task_t *tasks, int count, int list_index);

double perform_task(task_t repeat_number);

void *executor_thread(void *args);

void *receiver_thread(void *args);

void *sender_thread(void *args);

void *manager_thread(void *args);

void *reporter_thread(void *args);

#endif //H_TASK
```

threads.cpp

```
#include "threads.h"
#include <cmath>
#include <mpi.h>
#include <iostream>
#include <unistd.h>

#define multiplier 500

void generate_task_list(task_t *tasks, int task_count, int list_index) {
        for(int index = 0; index < task_count; index++)
                tasks[index] = multiplier + (int)
                (multiplier * pow((index + 1) +
                multiplier, 1. / 3) * sqrt(list_index + 1));
}

double perform_task(task_t repeat_count) {
        double sum = .0;
        for(int index = 0; index < repeat_count; index++)
                sum += sin(sqrt(index));
        return sum;
```

```
}

void *executor_thread(void *args) {
        ExecutorArgs *exec_args = (ExecutorArgs *) args;
        task_t *tasks = exec_args->task_buffer;
        int root_rank = exec_args->root_rank;
        int *current_task_count = exec_args->current_task_count;
        int *done_task_count = exec_args->done_task_count;
        bool *is_filled = exec_args->is_filled;
        int *current_list_number = exec_args->current_list_number;
        pthread_mutex_t *perform_mutex = exec_args->perform_mutex;
        pthread_mutex_t *count_mutex = exec_args->count_mutex;
        pthread_cond_t *cond = exec_args->cond;
        std::ostream *out = exec_args->out_file;

        double total_result = .0;
        double total_exec_time = .0;
        int total_performed_task_count = 0;

        double sub_result = .0;
        double sub_exec_time = .0;
        int performed_task_count = 0;
        int last_saved_list_number = 0;

        *out << "List\ttask\tResult\tTime\n";
        *out << "number\tcount\n";

        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        while(true) {
                pthread_mutex_lock(perform_mutex);
                if(!(*is_filled))
                        pthread_cond_wait(cond, perform_mutex);
                if(last_saved_list_number != *current_list_number) {
                        MPI_Send(&performed_task_count, 1,
                                 MPI_INT, root_rank,
                                 TASK_COUNT_TAG, MPI_COMM_WORLD);
                        MPI_Send(&sub_result, 1, MPI_DOUBLE, root_rank,
                                 GLOBAL_RESULT_TAG, MPI_COMM_WORLD);
                        MPI_Send(&sub_exec_time, 1, MPI_DOUBLE,
                               root_rank,
                                 TIME_REPORT_TAG, MPI_COMM_WORLD);
                        *out << last_saved_list_number << "\t" <<
                                 performed_task_count << "\t" «
                                  sub_result <<
                                 "\t" << sub_exec_time << "\n";
                        last_saved_list_number = *current_list_number;
                        total_result += sub_result;
```

```
                        total_exec_time += sub_exec_time;
                        total_performed_task_count +=
                                performed_task_count;
                        sub_result = .0;
                        performed_task_count = 0;
                        sub_exec_time = .0;
                        MPI_Barrier(MPI_COMM_WORLD);
                }
                if(*current_task_count == 0) {
                        pthread_mutex_unlock(perform_mutex);
                        break;
                }
                for(*done_task_count = 0; ; (*done_task_count)++) {
                        pthread_mutex_lock(count_mutex);
                        bool loop_is_done = *done_task_count >=
                                *current_task_count;
                        pthread_mutex_unlock(count_mutex);
                        if(loop_is_done)
                                break;

                        sub_exec_time -= MPI_Wtime();
                        sub_result += perform_task(
                                tasks[*done_task_count]);
                        sub_exec_time += MPI_Wtime();
                        performed_task_count++;
                }
                MPI_Send(NULL, 0, MPI_INT, root_rank,
                        TASK_PERFORMED_TAG, MPI_COMM_WORLD);
                MPI_Send(NULL, 0, MPI_INT, root_rank,
                        TASK_PERFORMED_TAG, MPI_COMM_WORLD);
                *is_filled = 0;
                pthread_cond_signal(cond);
                pthread_mutex_unlock(perform_mutex);
        }
        *out << "-----------------------------------\n";
        *out << "TOTAL:\t" << total_performed_task_count << "\t" <<
                total_result << "\t" << total_exec_time << "\n";

        pthread_exit(NULL);
}

void *receiver_thread(void *args) {
        ReceiverArgs *recv_args = (ReceiverArgs *) args;
        int root_rank = recv_args->root_rank;
        task_t *task_buffer = recv_args->task_buffer;
        int *current_task_count = recv_args->current_task_count;
        int *list_number = recv_args->current_list_number;
        int *done_task_count = recv_args->done_task_count;
        bool *is_filled = recv_args->is_filled;
```

```
        pthread_cond_t *cond = recv_args->cond;
        pthread_mutex_t *perform_mutex = recv_args->perform_mutex;
        pthread_mutex_t *count_mutex = recv_args->count_mutex;

        while(true) {
                pthread_mutex_lock(perform_mutex);
                if(*is_filled)
                        pthread_cond_wait(cond, perform_mutex);

                MPI_Status recv_status;
                MPI_Probe(MPI_ANY_SOURCE, ASSIGN_TAG, MPI_COMM_WORLD,
                   &recv_status);
                MPI_Get_count(&recv_status, MPI_INT, current_task_count);
                MPI_Request list_request;
                MPI_Irecv(list_number, 1, MPI_INT, root_rank,
                   LIST_NUMBER_TAG,
                        MPI_COMM_WORLD, &list_request);
                *done_task_count = 0;
                MPI_Recv(task_buffer, *current_task_count, MPI_INT,
                         MPI_ANY_SOURCE, ASSIGN_TAG, MPI_COMM_WORLD,
                         MPI_STATUS_IGNORE);
                if(*current_task_count == 0) {
                        *list_number = 0;
                        *is_filled = 1;
                        MPI_Request_free(&list_request);
                        pthread_cond_signal(cond);
                        pthread_mutex_unlock(perform_mutex);
                        break;
                }
                *is_filled = 1;
                int is_got;
                MPI_Test(&list_request, &is_got, MPI_STATUS_IGNORE);
                if(is_got) {
                        MPI_Wait(&list_request, MPI_STATUS_IGNORE);
                }
                else MPI_Request_free(&list_request);

                pthread_cond_signal(cond);
                pthread_mutex_unlock(perform_mutex);
        }

        pthread_exit(NULL);
}


void *sender_thread(void *args) {
        SenderArgs *send_args = (SenderArgs *) args;
        int root_rank = send_args->root_rank;
        task_t *task_buffer = send_args->task_buffer;
```

```
        int *current_task_count = send_args->current_task_count;
        int *done_task_count = send_args->done_task_count;
        pthread_mutex_t *mutex = send_args->count_mutex;

        while(true) {
                int destination;
                MPI_Recv(&destination, 1, MPI_INT, root_rank,
                        UNASSIGN_TAG,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if(destination == SENDER_TERMINATE)
                        break;
                if(destination == GET_REMAIN) {
                        pthread_mutex_lock(mutex);
                        int remain = *current_task_count -
                                *done_task_count;
                        pthread_mutex_unlock(mutex);
                        MPI_Send(&remain, 1, MPI_INT, root_rank,
                                REMAIN_COUNT_RESPONCE_TAG,
                                MPI_COMM_WORLD);
                        continue;
                }
                pthread_mutex_lock(mutex);
                int unassigned_task_count = (*current_task_count -
                         *done_task_count - 1) / 2;
                *current_task_count = *current_task_count -
                         unassigned_task_count;
                pthread_mutex_unlock(mutex);
                if(unassigned_task_count < 1) {
                        continue;
                }
                MPI_Send(task_buffer + *current_task_count,
                         unassigned_task_count, MPI_INT,
                         destination, ASSIGN_TAG, MPI_COMM_WORLD);
        }
        pthread_exit(NULL);
}

void drop_acks() {
        while(true) {
                MPI_Request ack_request;
                MPI_Irecv(NULL, 0, MPI_INT, MPI_ANY_SOURCE,
                        TASK_PERFORMED_TAG, MPI_COMM_WORLD,
                        &ack_request);
                int is_caught;
                MPI_Test(&ack_request, &is_caught, MPI_STATUS_IGNORE);
                if(is_caught) {
                        MPI_Wait(&ack_request, MPI_STATUS_IGNORE);
                }
                else {
```

```
                        MPI_Request_free(&ack_request);
                        break;
                }
        }
}

void *manager_thread(void *args) {
        ManagerArgs *manager_args = (ManagerArgs *) args;
        int list_count = manager_args->list_count;
        int list_size = manager_args->list_size;
        int process_count = manager_args->process_count;

        int *task_list = new int[list_size];
        int *process_statuses = new int[process_count];
        int *counts = new int[process_count];
        int *offsets = new int[process_count];

        offsets[0] = 0; counts[process_count - 1] = list_size -
             list_size * (process_count - 1) / process_count;
        for(int index = 1; index < process_count; index++) {
                offsets[index] = list_size * index / process_count;
                counts[index - 1] = offsets[index] - offsets[index - 1];
        }

        for(int list_num = 0; list_num < list_count; list_num++) {
                drop_acks();
                std::cerr << "List " << list_num << '\n';
                generate_task_list(task_list, list_size, list_num);
                for(int index = 0; index < process_count; index++) {
                        process_statuses[index] = counts[index];
                        MPI_Send(task_list + offsets[index],
                                counts[index], MPI_INT, index,
                                ASSIGN_TAG, MPI_COMM_WORLD);
                        MPI_Send(&list_num, 1, MPI_INT, index,
                                LIST_NUMBER_TAG, MPI_COMM_WORLD);
                }

                int remaining_task_count = list_size;
                while(remaining_task_count != 0) {
                        int destination;
                        MPI_Status ack_status;
                        MPI_Recv(&destination, 0, MPI_INT,
                                MPI_ANY_SOURCE, TASK_PERFORMED_TAG,
                                MPI_COMM_WORLD, &ack_status);
                        destination = ack_status.MPI_SOURCE;
                        int msg = GET_REMAIN;
                        for(int index = 0; index < process_count;
                                index++) {
                                MPI_Sendrecv(&msg, 1, MPI_INT, index,
```

```
                                        UNASSIGN_TAG, process_statuses +
                                        index, 1, MPI_INT, index,
                                        REMAIN_COUNT_RESPONCE_TAG,
                                        MPI_COMM_WORLD,
                                        MPI_STATUS_IGNORE);
                                }

                                int source = 0; int max_rem =
                                        process_statuses[0];
                                for(int rank = 1; rank < process_count; rank++) {
                                        if(max_rem < process_statuses[rank]) {
                                                max_rem = process_statuses[rank];
                                                source = rank;
                                        }
                                }
                                MPI_Send(&destination, 1, MPI_INT, source,
                                        UNASSIGN_TAG, MPI_COMM_WORLD);

                                remaining_task_count = 0;
                                for(int index = 0; index < process_count;
                                   index++) {
                                        remaining_task_count +=
                                        process_statuses[index];
                                }
                                std::cerr << destination << ' ' «
                                        remaining_task_count << '\n';
                        }

                }
                drop_acks();

                int terminate_senders = SENDER_TERMINATE;
                for(int index = 0; index < process_count; index++) {
                        MPI_Send(&terminate_senders, 1, MPI_INT, index,
                           UNASSIGN_TAG, MPI_COMM_WORLD);
                        MPI_Send(NULL, 0, MPI_INT, index, ASSIGN_TAG,
                           MPI_COMM_WORLD);
                }

                delete[] task_list;
                delete[] process_statuses;
                delete[] counts;
                delete[] offsets;
                pthread_exit(NULL);
}

void *reporter_thread(void *args) {
        ReporterArgs *report_args = (ReporterArgs*) args;
        std::ostream *report = report_args->main_report;
```

```
        int process_count = report_args->process_count;
        int list_count = report_args->list_count;

        double *time_reports = new double[process_count];
        double *sub_results = new double[process_count];
        int *task_counts = new int[process_count];


        *report << "List\tTotal\tTotal\tMin\tMax\tDisbalance,\n";
        *report << "number\ttasks\tresult\ttime\ttime\t%\n";

        for(int list_num = 0; list_num < list_count; list_num++) {
                for(int index = 0; index < process_count; index++) {
                        MPI_Recv(task_counts + index, 1, MPI_INT, index,
                                 TASK_COUNT_TAG, MPI_COMM_WORLD,
                                 MPI_STATUS_IGNORE);
                        MPI_Recv(sub_results + index, 1, MPI_DOUBLE,
                                index, GLOBAL_RESULT_TAG, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);
                        MPI_Recv(time_reports + index, 1, MPI_DOUBLE,
                                index, TIME_REPORT_TAG, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);
                }
                double result = sub_results[0];
                int total_task_count = task_counts[0];
                double min_time = time_reports[0], max_time = min_time;
                for(int index = 1; index < process_count; index++) {
                        min_time = std::min(min_time,
                                time_reports[index]);
                        max_time = std::max(max_time,
                                time_reports[index]);
                        result += sub_results[index];
                        total_task_count += task_counts[index];
                }
                *report << list_num << "\t" << total_task_count << '\t'
                         << result << '\t' << min_time << "\t" <<
                         max_time << "\t" << 100. * (1 - min_time /
                         max_time) << "\n";
        }

        delete[] time_reports;
        delete[] sub_results;
        delete[] task_counts;
        pthread_exit(NULL);
}
```

Приложение 4. *Makefile*

Файл сборки проекта

```
CXX=icpc
CXXP=mpiicpc
CFLAGS=-c -Wall -Wextra -std=c++14 -O3 -mt_mpi

MAIN_SOURCE=main
TASK_SOURCE=threads

all: $(MAIN_SOURCE)

$(MAIN_SOURCE): $(MAIN_SOURCE).o $(TASK_SOURCE).o
	$(CXXP) $(MAIN_SOURCE).o $(TASK_SOURCE).o -o $(MAIN_SOURCE)

$(MAIN_SOURCE).o: $(MAIN_SOURCE).cpp $(TASK_SOURCE).h
	$(CXXP) $(CFLAGS) $(MPIFLAGS) $(MAIN_SOURCE).cpp -o \
	     $(MAIN_SOURCE).o

$(TASK_SOURCE).o: $(TASK_SOURCE).cpp $(TASK_SOURCE).h
	$(CXXP) $(CFLAGS) $(TASK_SOURCE).cpp -o $(TASK_SOURCE).o

clean:
	rm -rf *.o
```