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

«Программирование многопоточных приложений. POSIX Threads»

студента 2 курса, группы 21206

Мельникова Никиты Сергеевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
к.т.н., доцент
А.Ю. Власенко

Новосибирск 2023

СОДЕРЖАНИЕ

ЦЕЛЬ

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

ЗАДАНИЕ

Необходимо организовать параллельную обработку списков неделимых заданий на нескольких компьютерах с динамическим распределением работы. Программа не должна зависеть от числа компьютеров.

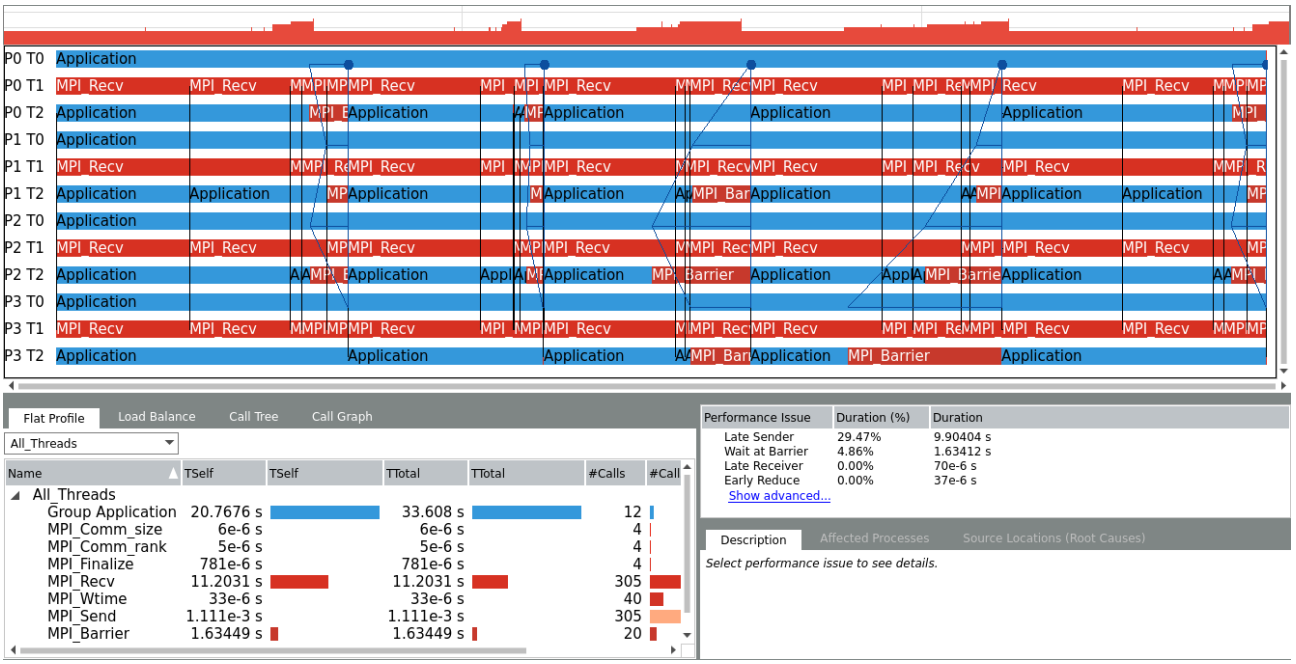
Для ее решения на каждом процессоре требуется завести несколько потоков (как минимум 2) и организовать правильную политику взаимодействия процессов, а также обеспечить работу нескольких потоков с общими структурами данных.

ОПИСАНИЕ РАБОТЫ

1. Была написана (листинг 1) параллельная программа на языке C с использованием MPI и средств POSIX Threads, моделирующая обработку задач с динамическим распределением между процессами.
2. Результаты с балансировкой нагрузки при задействовании 4-х процессов с размером списка задач в 1000 элементов (без нагрузки: доля дисбаланса ≈99,9%, среднее время работы ≈130 сек.):

Номер итерации	Мин. время, с.	Макс. время, с.	Время дисбаланса, с.	Доля дисбаланса, %
0	54,4	58	3,6	6,2
1	38	40,2	2,2	5,5
2	29,4	33,2	3,8	11,5
3	25,4	34,3	8,8	25,8
4	54,3	58	3,7	6,4
			Среднее:	11

3. Было выполнено профилирование программы при использовании 4-х процессов средствами Intel Trace Analyzer and Collector (ITAC).



Flat Profile | Load Balance | Call Tree | Call Graph

All_Threads

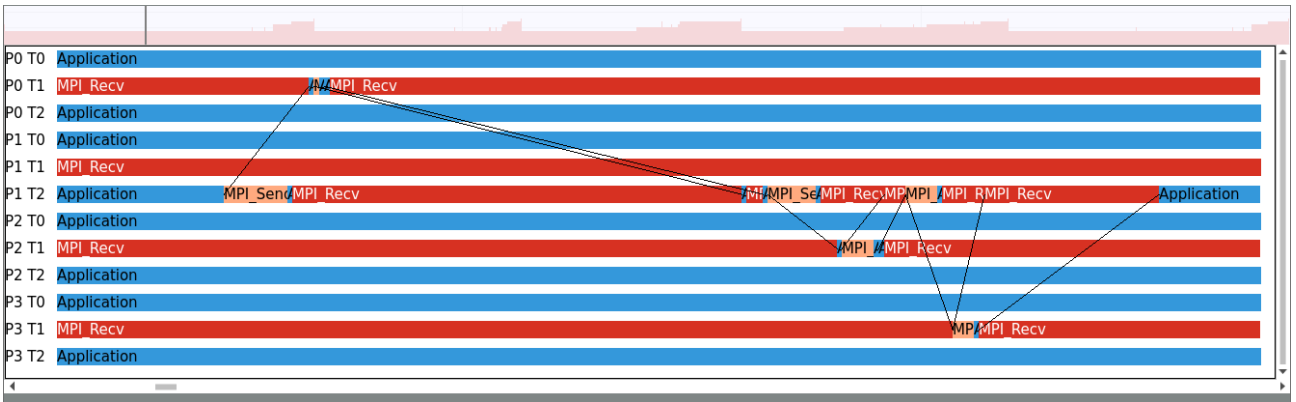
Name	TSelf	TSelf	TTotal	TTotal	#Calls	#Call
All_Threads						
Group Application	20.7676 s		33.608 s		12	
MPI_Comm_size	6e-6 s		6e-6 s		4	
MPI_Comm_rank	5e-6 s		5e-6 s		4	
MPI_Finalize	781e-6 s		781e-6 s		4	
MPI_Recv	11.2031 s		11.2031 s		305	
MPI_Wtime	33e-6 s		33e-6 s		40	
MPI_Send	1.111e-3 s		1.111e-3 s		305	
MPI_Barrier	1.63449 s		1.63449 s		20	

Performance Issue	Duration (%)	Duration
Late Sender	29.47%	9.90404 s
Wait at Barrier	4.86%	1.63412 s
Late Receiver	0.00%	70e-6 s
Early Reduce	0.00%	37e-6 s

Show advanced...

Description | Affected Processes | Source Locations (Root Causes)

Select performance issue to see details.



Flat Profile | Load Balance | Call Tree | Call Graph

All_Threads

Name	TSelf	TSelf	TTotal	TTotal	#Calls	#
All_Threads						
Group Application	1.65803e-3 s		2.73455e-3 s		0	
MPI_Recv	1.03852e-3 s		1.03852e-3 s		9	
MPI_Send	38e-6 s		38e-6 s		9	

Performance Issue	Duration (%)	Duration
Late Sender	12.33%	337.163e-6 s

Show advanced...

Description | Affected Processes | Source Locations (Root Causes)

Select performance issue to see details.

ЗАКЛЮЧЕНИЕ

В ходе практической работы удалось ознакомиться с задачей динамического распределения работы между процессами. Были освоены методы реализации алгоритмов параллелизма для решения этой проблемы средствами POSIX Threads API. Использование POSIX Threads позволяет существенно распараллелить обработку задач.

Приложение 1. *Листинг параллельной программы на языке Си*

Код компиляции: `mpiicc -mt_mpi main.c -o main`

Листинг 1 — Исходный код программы

```
#include <math.h>
#include <mpi.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define TASK_LIST_SIZE     1000
#define NUM_OF_TASK_LISTS  5
#define L                  100000
#define MIN_TASKS_TO_SHARE 2

#define ASK_TAG               100
#define TASK_NUMBER_REPLY_TAG 200
#define TASKS_SENDING_TAG     300

int rank, size;
pthread_mutex_t mutex;
pthread_t threads[2];
int *tasks;
int remainig_tasks, completed_tasks;
double global_res, res;
double total_disbalance_factor;

void init_task_list(int iter_counter) {
    for (int i = 0; i < TASK_LIST_SIZE; ++i)
        tasks[i] = abs(50 - i % 100) *
                   abs(rank - (iter_counter % size)) * L;
}
```

```
void execute_task_list() {
    int i = 0;
    while (remainig_tasks > 0) {
        pthread_mutex_lock(&mutex);
        int repeat_num = tasks[i];
        pthread_mutex_unlock(&mutex);

        for (int j = 0; j < repeat_num; ++j)
            global_res += sin(j) / 2;

        pthread_mutex_lock(&mutex);
        ++completed_tasks;
        --remainig_tasks;
        pthread_mutex_unlock(&mutex);
    }
}

bool ask_more_tasks() {
    int counter = 1;
    for (int i = 0; i < size; ++i) {
        int additional_tasks;
        if (i != rank) {
            MPI_Send(&rank, 1, MPI_INT, i, ASK_TAG, MPI_COMM_WORLD);
            MPI_Recv(&additional_tasks, 1, MPI_INT,
                     i, TASK_NUMBER_REPLY_TAG,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            if (additional_tasks != 0) {
                pthread_mutex_lock(&mutex);
                MPI_Recv(tasks, additional_tasks, MPI_INT,
                         i, TASKS_SENDING_TAG,
                         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                remainig_tasks += additional_tasks;
                pthread_mutex_unlock(&mutex);
            } else { ++counter; }
```

```
        }
    }
    return (counter == size);
}

void *executor_start_routine(void *args) {
    tasks = (int *) malloc(TASK_LIST_SIZE * sizeof(int));
    double time_start, time_end, total_time, min_time, max_time;

    for (int iter_counter = 0; iter_counter < NUM_OF_TASK_LISTS;
++iter_counter) {
        init_task_list(iter_counter);
        remainig_tasks  = TASK_LIST_SIZE;
        completed_tasks = 0;
        time_start = MPI_Wtime();
        execute_task_list();
        for (int i = 0; i < size; ++i) {
            if (ask_more_tasks()) break;
            execute_task_list();
        }
        time_end = MPI_Wtime();
        MPI_Barrier(MPI_COMM_WORLD);
        total_time = time_end - time_start;
        MPI_Reduce(&total_time, &min_time, 1, MPI_DOUBLE,
                   MPI_MIN, 0, MPI_COMM_WORLD);
        MPI_Reduce(&total_time, &max_time, 1, MPI_DOUBLE,
                   MPI_MAX, 0, MPI_COMM_WORLD);
        MPI_Reduce(&global_res, &res, 1, MPI_DOUBLE,
                   MPI_SUM, 0, MPI_COMM_WORLD);
        printf("Rank: %d, tasks done: %d, time: %lf\n",
               rank, completed_tasks, total_time);
        if (rank == 0) {
            double disbalance        = max_time - min_time;
            double disbalance_factor = (disbalance / max_time) * 100;
            total_disbalance_factor += disbalance_factor;
```

```
            fprintf(stderr, "Max time: %lf, min time: %lf, result: %lf,
                    disbalance: %lf, disbalance factor: %lf\n",
                   max_time, min_time, global_res, disbalance,
                   disbalance_factor);
        }
    }
    if (rank == 0) {
fprintf(stderr, "Average disbalance factor: %lf\n",
        total_disbalance_factor / NUM_OF_TASK_LISTS);
    }
    int exit_flag = -1;
    MPI_Send(&exit_flag, 1, MPI_INT, rank, ASK_TAG, MPI_COMM_WORLD);
    free(tasks);
    pthread_exit(EXIT_SUCCESS);
}

void *receiver_start_routine(void *args) {
    while (true) {
        int transmitter_rank;
        MPI_Recv(&transmitter_rank, 1, MPI_INT, MPI_ANY_SOURCE, ASK_TAG,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        if (transmitter_rank == -1) {
            pthread_exit(EXIT_SUCCESS);
        }
        int num_of_sharing_tasks;
        pthread_mutex_lock(&mutex);
        if (remainig_tasks >= MIN_TASKS_TO_SHARE) {
            num_of_sharing_tasks = remainig_tasks / size;
            MPI_Send(&num_of_sharing_tasks, 1, MPI_INT,
                     transmitter_rank, TASK_NUMBER_REPLY_TAG,
                     MPI_COMM_WORLD);
            MPI_Send(&(tasks[TASK_LIST_SIZE - num_of_sharing_tasks]),
                     num_of_sharing_tasks, MPI_INT,
                     transmitter_rank, TASKS_SENDING_TAG, MPI_COMM_WORLD);
```

```
            remainig_tasks -= num_of_sharing_tasks;
            pthread_mutex_unlock(&mutex);
        } else {
            pthread_mutex_unlock(&mutex);
            num_of_sharing_tasks = 0;
            MPI_Send(&num_of_sharing_tasks, 1, MPI_INT,
                     transmitter_rank, TASK_NUMBER_REPLY_TAG,
                     MPI_COMM_WORLD);
        }
    }
}

int main(int argc, char *argv[]) {
    int provided;
    pthread_attr_t thread_attr;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided != MPI_THREAD_MULTIPLE) {
        MPI_Finalize();
        fprintf(stderr, "Required level cannot be provided\n");
        return EXIT_FAILURE;
    }
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    pthread_mutex_init(&mutex, NULL);
    pthread_attr_init(&thread_attr);
    pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);
    pthread_create(&threads[0], &thread_attr, receiver_start_routine, NULL);
    pthread_create(&threads[1], &thread_attr, executor_start_routine, NULL);

    pthread_join(threads[0], NULL);
    pthread_join(threads[1], NULL);
```

```
    pthread_attr_destroy(&thread_attr);
    pthread_mutex_destroy(&mutex);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```