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

«Динамическое распределение работы между процессорами»

студента Бородина Артёма Максимовича 2 курса, 19205 группы

Направление 09.03.01 – «Информатика и вычислительная техника»

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

Новосибирск 2021

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 Задания могут иметь различный вычислительный вес, т.е. требовать при одних и тех же вычислительных ресурсах различного времени для выполнения. Считается, что этот вес нельзя узнать, пока задание не выполнено. После того, как *все* задания из списка выполнены, появляется новый список заданий. Необходимо организовать параллельную обработку заданий на нескольких компьютерах. Количество заданий существенно превосходит количество процессоров. Программа не должна зависеть от числа компьютеров.

Понятно, что для распараллеливания задачи задания из списка нужно распределять между компьютерами. Так как задания имеют различный вычислительный вес, а список обрабатывается итеративно, и требуется синхронизация перед каждой итерацией, то могут возникать ситуации, когда некоторые процессоры выполнили свою работу, а другие -- еще нет. Если ничего не предпринять, первые будут простаивать в ожидании последних. Так возникает задача динамического распределения работы. Для ее решения на каждом процессоре заведем несколько потоков. Как минимум, потоков должно быть 2:

1. поток, который обрабатывает задания и, когда задания закончились, обращается к другим компьютерам за добавкой к работе,
2. поток, ожидающий запросов о работе от других компьютеров

ОПИСАНИЕ РАБОТЫ

1. Была написана программа (Приложение 1), с использованием MPI и POSIX Threads, реализующая алгоритм разделения нагрузки.

2. После завершения списка задач велся подсчет времени и доли дисбаланса. В среднем доля дисбаланса была равна от 2-х до 4-х процентов.

3. Сделано профилирование программы на 4-х ядрах (Приложение 2). Было подтверждено что работа процессов и потоков соответствует алгоритму. У каждого процесса по два потока (не считая основной): один считает задачи и делает запросы дополнительных заданий, а второй постоянно ждет запроса от процессов и отправляет задания своего процесса, если такие имеются.

ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были получены знания о балансировке нагрузки на множество процессов и работе с потоками в MPI программах.

Приложение 1. *Код программы*

```
#include <iostream>
#include <pthread.h>
#include <cmath>
#include "mpi.h"

#define ASK_TAG 1
#define ACK_TAG 2
#define TASK_TAG 3
#define IMBALANCE_TAG 4
#define TASK 10
#define NO_TASK 11
#define ASK_FOR_TASK 12
#define TURN_OFF 13
#define TURN_ON 14

typedef struct Task {
    int weight;
} Task;

typedef struct ACK {
    int count;
} ACK;

Task *list = nullptr;
MPI_Datatype MPI_TASK, MPI_ACK;
pthread_mutex_t mutex;

int size, rank;

int startWeight;
int startSize;
int iterCount;
int curIter = 0;

int currentTask = 0;
int listSize;

int tasksDone = 0;
long long weightDone = 0;
bool gotTask = false;

double totalImbalanceShare = 0;
double totalMinDuration = 0;
double totalMaxDuration = 0;

void createTypes() {
    int blockLengths[1] = {1};
    MPI_Aint displacements[1];
    displacements[0] = 0;
    MPI_Datatype types[] = {MPI_INT};

    MPI_Type_create_struct(1, blockLengths, displacements, types, &MPI_TASK);
    MPI_Type_commit(&MPI_TASK);

    MPI_Type_create_struct(1, blockLengths, displacements, types, &MPI_ACK);
    MPI_Type_commit(&MPI_ACK);
}

void createList() {
    pthread_mutex_lock(&mutex);
    if (list != nullptr) {
        delete (list);
    }
    list = new Task[startSize];
    listSize = startSize;
    currentTask = 0;

    for (int i = 0; i < startSize; ++i) {
        list[i].weight = startWeight + abs(50 - i % 100) * abs(rank - (curIter % size)) * startWeight;
    }
    pthread_mutex_unlock(&mutex);
}

int getTasks(int proc) {
    int message = ASK_FOR_TASK;
    MPI_Send(&message, 1, MPI_INT, proc, ASK_TAG, MPI_COMM_WORLD);

    ACK ack;
    MPI_Recv(&ack, 1, MPI_ACK, proc, ACK_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
    int taskCount = ack.count;
    if (!taskCount) {
        return NO_TASK;
    } else {
        pthread_mutex_lock(&mutex);
        delete (list);
        list = new Task[taskCount];
        std::cout << "Proc " << rank << " is getting " << taskCount << " tasks from " << proc << std::endl;
        MPI_Recv(list, taskCount, MPI_ACK, proc, TASK_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        currentTask = 0;
        listSize = taskCount;
        gotTask = true;
        pthread_mutex_unlock(&mutex);
        return TASK;
    }
}

double countListRes() {
    double globalRes = 0;
    pthread_mutex_lock(&mutex);
    while (currentTask < listSize) {
        int weight = list[currentTask].weight;
        pthread_mutex_unlock(&mutex);

        weightDone += weight;
        for (int i = 0; i < weight; i++) {
            globalRes += sin(i);
        }

        pthread_mutex_lock(&mutex);
        currentTask++;
        tasksDone++;
    }
    pthread_mutex_unlock(&mutex);
    return globalRes;
}

void calculateImbalance(double duration) {
    double imbalanceTime;
    double imbalanceShare;
    double maxDuration = 0;
    double minDuration = 1e100;
    if (rank) {
        MPI_Send(&duration, 1, MPI_DOUBLE, 0, IMBALANCE_TAG, MPI_COMM_WORLD);
    } else {
        double durations[size];
        durations[0] = duration;
        for (int i = 1; i < size; ++i) {
            MPI_Recv(durations + i, 1, MPI_DOUBLE, i, IMBALANCE_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        for (int i = 0; i < size; ++i) {
            if (durations[i] < minDuration) {
                minDuration = durations[i];
            }
            if (durations[i] > maxDuration) {
                maxDuration = durations[i];
            }
        }
        imbalanceTime = maxDuration - minDuration;
        imbalanceShare = imbalanceTime / maxDuration * 100;
        std::cout << "Imbalance time is " << imbalanceTime << "s" << std::endl;
        std::cout << "Share of imbalance is " << imbalanceShare << "%" << std::endl << std::endl;

        totalImbalanceShare += imbalanceShare;
        totalMinDuration += minDuration;
        totalMaxDuration += maxDuration;
    }
}

void *processList(void *args) {
    double globalRes = 0;
    double timeStart, timeEnd, duration;
    int lastReceivedTask = 0;
    while (curIter < iterCount) {
        if (!gotTask) {
            timeStart = MPI_Wtime();
            createList();
        }
        globalRes += countListRes();

        gotTask = false;
        for (int i = lastReceivedTask; i < size; i++) {
            if (i != rank) {
                if (getTasks(i) == TASK) {
```

```
                    lastReceivedTask = i;
                    break;
                }
            }
        }
        if (gotTask) {
            continue;
        }

        timeEnd = MPI_Wtime();
        duration = timeEnd - timeStart;
        MPI_Barrier(MPI_COMM_WORLD);

        std::cout << "Proc " << rank << " finished " << curIter + 1
                  << " list with " << tasksDone << " tasks and " << weightDone << " weight done" << std::endl;
        std::cout << "Proc " << rank << " global res is " << globalRes << " time spent on iteration "
                  << duration << std::endl;
        calculateImbalance(duration);

        tasksDone = 0;
        weightDone = 0;
        globalRes = 0;
        lastReceivedTask = 0;
        ++curIter;
    }
    int message = TURN_OFF;
    MPI_Send(&message, 1, MPI_INT, rank, ASK_TAG, MPI_COMM_WORLD);
    return nullptr;
}

void *loadBalancing(void *args) {
    int message = TURN_ON;
    while (message != TURN_OFF) {
        MPI_Status status;
        MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE, ASK_TAG, MPI_COMM_WORLD, &status);

        if (message == ASK_FOR_TASK) {
            ACK ack;
            pthread_mutex_lock(&mutex);
            if (currentTask >= listSize - 1 || gotTask) {
                pthread_mutex_unlock(&mutex);
                ack.count = 0;
                MPI_Send(&ack, 1, MPI_ACK, status.MPI_SOURCE, ACK_TAG, MPI_COMM_WORLD);
            } else {
                double finishedFraction = currentTask / double(listSize);
                int taskCount = (listSize - currentTask) * finishedFraction / (size - 1) + 1;
                ack.count = taskCount;
                MPI_Send(&ack, 1, MPI_ACK, status.MPI_SOURCE, ACK_TAG, MPI_COMM_WORLD);

                auto *newList = new Task[taskCount];
                for (int i = 0; i < taskCount; ++i) {
                    newList[i].weight = list[listSize - taskCount + i].weight;
                }
                listSize -= taskCount;
                pthread_mutex_unlock(&mutex);

                MPI_Send(newList, taskCount, MPI_TASK, status.MPI_SOURCE, TASK_TAG, MPI_COMM_WORLD);
            }
        }
    }
    return nullptr;
}

int main(int argc, char *argv[]) {
    int provided;
    int error = MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (provided != MPI_THREAD_MULTIPLE) {
        if (!rank) {
            char errorString[MPI_MAX_ERROR_STRING];
            int len;
            MPI_Error_string(error, errorString, &len);
            std::cout << "ERROR: provided is not MPI_THREAD_MULTIPLE, provided: error code :"
                      << error << " - " << errorString << " provided: " << provided << std::endl;
        }
        MPI_Finalize();
        return 0;
    }

    if (argc < 4) {
        if (!rank) {
            std::cout << "Usage: loadBalancing.exe iter_count list_size start_weight" << std::endl;
```

```
        }
        MPI_Finalize();
        return 0;
    }
    iterCount = atoi(argv[1]);
    startSize = atoi(argv[2]);
    startWeight = atoi(argv[3]);

    createTypes();

    pthread_mutex_init(&mutex, nullptr);
    pthread_attr_t attributes;
    if (pthread_attr_init(&attributes) != 0) {
        std::cout << "ERROR: Cannot init attributes: " << errno << std::endl;
        MPI_Finalize();
        return 0;
    }
    pthread_t threads[2];
    double start = MPI_Wtime();
    pthread_create(&threads[0], &attributes, loadBalancing, nullptr);
    pthread_create(&threads[1], &attributes, processList, nullptr);
    for (pthread_t thread : threads) {
        if (pthread_join(thread, nullptr) != 0) {
            std::cout << "ERROR: Cannot join a thread: " << errno << std::endl;
            MPI_Finalize();
            return 0;
        }
    }
    double end = MPI_Wtime();

    if (!rank) {
        std::cout << "Average min duration: " << totalMinDuration / iterCount << "s" << std::endl;
        std::cout << "Average max duration: " << totalMaxDuration / iterCount << "s" << std::endl;
        std::cout << "Average imbalance share: " << totalImbalanceShare / iterCount << "%" << std::endl;
        std::cout << "Time: " << end - start << std::endl;
    }

    pthread_mutex_destroy(&mutex);
    MPI_Finalize();
    return 0;
}
```

Приложение 2. *Профилирование на 4 ядрах*