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, 18209 группы

Большим Максима Антонович

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Матвеев А.С.

Новосибирск 2020

СОДЕРЖАНИЕ

ЦЕЛЬ

В данной лабораторной работе требуется освоить многопоточное программирование, используя инструменты создания POSIX Threads и MPI. Целью данной лабораторной работы является создание сбалансированной системы, распределяющей задания между всеми доступными нодами, т.е. процессами, способных исполнять поставленные общей системой задачи.

ЗАДАНИЕ

Реализовать некоторый аналог вычислительного кластера, принимающего при старте некоторый список задач для каждой “ноды кластера” - процесса MPI программы. Реализовать систему автобалансировки задач между всеми нодами в случае, когда у какой-то из доступных закончились собственные, т.е. в тот момент, когда вычислительная эффективность кластера начинает уменьшаться. Общение нод, т.е. процессов, реализовать посредством стандартного MPI.

Работа ноды кластера должна быть реализована следующим образом: в процессе должно быть запущено 2 POSIX потока:

- поток “*исполнитель*”, исполняющий поступившие к нему задачи;
- поток “*коммуникатор*”, коммуницирующий со всеми остальными нодами.

В момент, когда у исполнителя закончились задачи, он должен “разбудить” коммуникатор, чтобы тот, в свою очередь, послал сообщение “корневой” ноде, что произведёт ребалансировку задач между всеми нодами. Исполнитель после пробуждения коммуникатора должен ожидать от него ответное сообщение, в котором уже говорится, продолжать работу или нет.

Работа коммуникатора корневой ноды заключается в том, чтобы принимать сообщения не только от своего исполнителя, но также запросы ребалансировки ото всех остальных нод. Получив запрос ребалансировки, корневая нода запускает алгоритм получения актуального списка задач ото всех нод, чтобы на их основе создать новый сбалансированный по вычислительной сложности список задач для каждой ноды. После исполнения алгоритма корневая нода возвращает получившиеся списки всем остальным нодам. В случае, если у всех нод закончились задачи, корневая нода посылает всем сообщение о прекращении работы.

Балансировка должна уметь производиться в 2х режимах: когда нам известна вычислительная мощность задач и когда не известна.

Поскольку инструментарий POSIX потоков не является кроссплатформенным полностью, предлагается реализовать потоки из стандарта ANSI C++, которые, при возможности, будут использовать и дополнять функционал POSIX потоков.

КОД ПРОГРАММЫ

main.cpp

```
#include "threads.h"


using namespace std;


static int random(int begin, int end)

{

    return rand() % (end - begin + 1) + begin;

}


int main(int argc, char* argv[])

{

    int provided = 0;

    int rank = 0;


    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    if(argc != 2)

    {

        if(!rank)

            printf("Invalid arg count!\n");


        MPI_Finalize();
```

```
        return EXIT_FAILURE;
    }

    int mode = -1000;

    if(sscanf(argv[1], "%d", &mode) != 1 || (mode !=
I_KNOW_COMPUTATIONAL_COMPLEXITY && mode !=
I_DONT_KNOW_COMPUTATIONAL_COMPLEXITY))
    {
        if(!rank)
            printf("Invalid arg value!\n");

        MPI_Finalize();

        return EXIT_FAILURE;
    }

    if(provided != MPI_THREAD_MULTIPLE)
    {
        fprintf(stderr, "Some error with MPI: failed to get the required
requirements!\n");
        exit(EXIT_FAILURE);
    }

    int totalGlobal = 0;

    int* list = initJobList(rank);

    for (int l = 0; l < listSize(list); ++l)
        totalGlobal += list[l];

    if(!list)
    {
```

```
        threadSavePrint("Can't alloc memory in process %d! Shutting down.",
stdout, rank);
        MPI_Finalize();

        return EXIT_SUCCESS;
    }

    string message = "%*d process begin with: ";

    for (int i = 0; i < TOTAL_LIST_SIZE; ++i)
    {
        if(list[i] == -1)
            break;

        message += to_string(list[i]);
        message += " ";
    }
    message.pop_back();
    message += '\n';

    threadSavePrint(message, stdout, 2, rank);
    std::this_thread::sleep_for(std::chrono::milliseconds(500));

    MPI_Barrier(MPI_COMM_WORLD);

    double start = MPI_Wtime();

    thread worker_thread(worker, list);
    thread communicator_thread(communicator, list, mode);

    int newTask;
    bool brFlag = false;
```

```
    for (int j = 1; j <= 4; ++j)//Add task times
    {
        for (int k = 0; k < 25 * j; ++k) // after 25 sec add new task, but every second check thread is alive
        {
            std::this_thread::sleep_for(std::chrono::seconds (1));
            if(isDead())
            {
                brFlag = true;
                break;
            }
        }
        if(brFlag)
            break;

        lockList();

        threadSavePrint("======Add new tasks to node %*d\n", stdout, 2, rank);
        if(TOTAL_LIST_SIZE - listSize(list) > 1)
            for (int i = 0; i < random((TOTAL_LIST_SIZE - listSize(list)) / 2, TOTAL_LIST_SIZE - listSize(list)); ++i)
            {
                newTask = random(1, MAX_COMPUTATIONAL_COMPLEXITY);

                totalGlobal += newTask;
                list[listSize(list)] = newTask;
            }

        message = "======New list of %*d:\n";

        for (int i = 0; i < TOTAL_LIST_SIZE; ++i)
        {
            if(list[i] == -1)
                break;
```

```
            message += to_string(list[i]);
            message += " ";
        }
        message.pop_back();
        message += '\n';

        threadSavePrint(message, stdout, 2, rank);

        unlockList();
    }

    communicator_thread.join();
    worker_thread.join();

    double finish = MPI_Wtime();

    threadSavePrint("-- %*d process: shutting down\n", stdout, 2, rank);

    MPI_Barrier(MPI_COMM_WORLD);

    int commSize, sendArray[1] = {totalGlobal};

    MPI_Comm_size(MPI_COMM_WORLD, &commSize);
    int *totalArr = rank ? nullptr : (int *)malloc(commSize * sizeof(int));

    MPI_Gather(sendArray, 1, MPI_INT, totalArr, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if(!rank)
    {
        int max = INT_MIN;

        for (int i = 0; i < commSize; ++i)
```

```
            if(max < totalArr[i])
                max = totalArr[i];


        threadSavePrint("Time elapsed: %d, max time: %d\n", stdout,
(int)(finish - start), max);
        free(totalArr);
    }


    free(list);
    MPI_Finalize();


    return EXIT_SUCCESS;
}
```

list.cpp

```
#include "list.h"


int popLast(int* list)
{
    if(!nonEmpty(list))
        return -1;


    for (int i = 0; i < TOTAL_LIST_SIZE; ++i)
        if(list[i] == -1)
        {
            int buf = list[i - 1];
            list[i - 1] = -1;
            return buf;
        }


    return -1;
}


static std::vector<int> sortedVec(int** lists, int size, int& total)
```

```
{
    std::vector<int> res;
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < TOTAL_LIST_SIZE; ++j)
        {
            if(lists[i][j] == -1)
                break;

            res.emplace_back(lists[i][j]);
            total += lists[i][j];
        }

    std::sort(res.begin(), res.end());
    return res;
}

int rebalance(int** lists, int size, int mode)
{
    int total = 0;
    std::vector<int> vec = sortedVec(lists, size, total);
    if(vec.empty())
        return EMPTY_LISTS;

    for (int k = 0; k < size; ++k)
        for (int l = 0; l < TOTAL_LIST_SIZE; ++l)
            lists[k][l] = -1;

    if(mode == I_KNOW_COMPUTATIONAL_COMPLEXITY)
    {
        int balance = total / size;
        int localTotal = 0;
        int j = 0;
        int minimalDifference;
```

```
int minDivIndex;

for (int i = 0; i < size; ++i, localTotal = 0, j = 0)
    while (localTotal < balance && !vec.empty())
    {
        minimalDifference = INT_MAX;
        minDivIndex = 0;
        for (int k = 0; k < vec.size(); ++k)
            if(minimalDifference > abs(balance - localTotal - vec[k]))
            {
                minimalDifference = abs(balance - localTotal - vec[k]);
                minDivIndex = k;
            }

        lists[i][j++] = vec[minDivIndex];
        localTotal += vec[minDivIndex];

        auto it = vec.begin();
        std::advance(it, minDivIndex);
        vec.erase(it);
    }

while (!vec.empty())
{
    int minTotal = INT_MAX;
    total = 0;
    for (int i = 0; i < size; ++i)
    {
        for (int k = 0; k < TOTAL_LIST_SIZE; ++k)
            if(lists[i][k] != -1)
                total += lists[i][k];
            else
                break;
```

```
                if(minTotal > total && listSize(lists[i]) != TOTAL_LIST_SIZE)
                {
                    j = i;
                    minTotal = total;
                }

                total = 0;
            }

            auto max = std::max_element(vec.begin(), vec.end());
            lists[j][listSize(lists[j])] = *max;

            vec.erase(max);
        }


    } else
    {
        int index = 0;

        while (!vec.empty())
        {
            lists[index][listSize(lists[index])] = vec.back();
            vec.pop_back();

            index = (index + 1) % size;
        }
    }
    return SUCCESS_BALANCE;
}

bool nonEmpty(const int* list)
```

```
{
    return list[0] != -1;
}

int listSize(const int* list)
{
    for (int i = 0; i < TOTAL_LIST_SIZE; ++i)
        if(list[i] == -1)
            return i;

    return TOTAL_LIST_SIZE;
}

static int random(int begin, int end)
{
    return rand() % (end - begin + 1) + begin;
}

int* initJobList(int rank)
{
    int* list = (int*) malloc(sizeof(int) * TOTAL_LIST_SIZE);

    if(!list)
        return nullptr;

    for (int j = 0; j < TOTAL_LIST_SIZE; ++j)
        list[j] = -1;

    srand(time(nullptr) * rank);

    int limit;
    if(rank)
        limit = random(1, TOTAL_LIST_SIZE / 2);
```

```
    else
        limit = random(TOTAL_LIST_SIZE / 2, TOTAL_LIST_SIZE);



    for (int i = 0; i < limit; ++i)
        list[i] = random(1, MAX_COMPUTATIONAL_COMPLEXITY);


    return list;
}
```

thread.cpp

```
#include "threads.h"


std::mutex              work_mutex;
std::condition_variable work_wait;
bool                    work_notified;


int                     work_flag = EMPTY;
std::mutex              list_mutex;
std::mutex              print_mutex;


bool                    is_dead = false;


bool isDead()
{
    return is_dead;
}


static void unlockWorker(int message);


void lockList()
{
    list_mutex.lock();
```

```
}

void unlockList()
{
    list_mutex.unlock();
}

void worker(int* list_ref)
{
    int work;
    int rank;
    int flag = NEED_TASKS;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    while(true)
    {
        if(!nonEmpty(list_ref) || work_flag == STOP_WORK)
        {
            if(work_flag == STOP_WORK)
                break;

            {
                sendMsgFlag(flag, rank);

                std::unique_lock<std::mutex> locker(work_mutex);
                while(!work_notified)// may be spurious wakeup
                    work_wait.wait(locker);
            }

            if(work_flag == STOP_WORK)
                break;

            work_flag = EMPTY;
```

```
        }

        lockList();
        work = popLast(list_ref);
        unlockList();

        if(work != -1)
        {
            threadSavePrint("-- %*d process: begin do new task %*d: %*d tasks
left\n", stdout, 2, rank, 2, work, 2, listSize(list_ref));
            std::this_thread::sleep_for(std::chrono::seconds(work));
            work_notified = false;
        } else if(work_notified)
            break;
    }
    is_dead = true;
}

void communicator(int* list_ref, int iKnowCompFlexibility)
{
    int rank;
    int size;
    int flag = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    while (true)
    {
        recvMsgFlag(flag, MPI_ANY_SOURCE);

        if(flag == CONTINUE_WORK)
            continue;
        else if(flag == STOP_WORK)
```

```
        {
            unlockWorker(STOP_WORK);
            return;
        }
        else if(flag == TASK_REQUEST_STEP_2)
        {
            lockList();

            sendMsgArray(list_ref, TOTAL_LIST_SIZE, 0);
            recvMsgFlag(flag, 0);

            if(flag == STOP_WORK)
            {
                for (int i = 0; i < TOTAL_LIST_SIZE; ++i) //Clear list, because
of its empty on rebalance
                    list_ref[i] = -1;

                unlockWorker(STOP_WORK);
                unlockList();
                return;
            }
            else
            {
                recvMsgArray(list_ref, TOTAL_LIST_SIZE, 0);
                unlockWorker(CONTINUE_WORK);
            }

            unlockList();
        }

        if(rank == 0)
        {
            switch (flag)
```

```
            {
                case NEED_TASKS:
                case TASK_REQUEST_STEP_1:
                {
                    threadSavePrint("begin rebalance\n", stdout);
                    flag = TASK_REQUEST_STEP_2;
                    for (int i = 1; i < size; ++i)
                        sendMsgFlag(flag, i);

                    bool mallocErrorFlag = false;
                    int** lists = (int**) malloc(sizeof(int*) * size);

                    if(!lists)
                        mallocErrorFlag = true;
                    else
                        for (int i = 1; i < size; ++i)
                            if(!(lists[i] = (int*) malloc(sizeof(int) *
TOTAL_LIST_SIZE)))
                            {
                                for (int j = 1; j < size; ++j)
                                    if(lists[i])
                                        free(lists[i]);

                                free(lists);

                                mallocErrorFlag = true;
                                break;
                            }

                    if(mallocErrorFlag)
                    {
                        flag = STOP_WORK;
                        for (int i = 1; i < size; ++i)
```

```
                        {
                            recvMsgArray(list_ref, TOTAL_LIST_SIZE, i);
                            sendMsgFlag(flag, i);
                        }

                        threadSavePrint("Can't alloc memory in process 0!
\nShutting down.\n", stderr);
                        unlockWorker(STOP_WORK);
                        return;
                    } else
                        for (int k = 1; k < size; ++k)
                            recvMsgArray(lists[k], TOTAL_LIST_SIZE, k);

                    lockList();
                    lists[0] = list_ref;

                    int res = rebalance(lists, size, iKnowCompFlexibility);
                    unlockList();

                    if(res == EMPTY_LISTS)
                    {
                        threadSavePrint("end rebalance: empty all\nclosing
cluster...\n", stdout);

                        flag = STOP_WORK;
                        for (int i = 1; i < size; ++i)
                        {
                            sendMsgFlag(flag, i);
                            free(lists[i]);
                        }

                        unlockWorker(STOP_WORK);
                        free(lists);
```

```
                return;
            }else
            {
                print_mutex.lock();

                int balanced = 0;

                for (int j = 0; j < size; ++j)
                    for (int k = 0; k < TOTAL_LIST_SIZE; ++k)
                        if(lists[j][k] != -1)
                            balanced += lists[j][k];
                        else
                            break;

                balanced /= size;
                int weight = 0;

                printf("rebalanced task lists:");

                for (int l = 0; l < size; ++l)
                {
                    printf("\n%*d - ", 3, l);
                    if(listSize(lists[l]) == 0)
                        printf("empty");
                    else
                        for (int i = 0; i < TOTAL_LIST_SIZE; ++i)
                            if(lists[l][i] == -1)
                            {
                                printf("\n---- lw vs bl: %*d %*d", 2,
weight, 2, balanced);
                                weight = 0;
                                break;
                            }
```

```
                else
                {
                    weight += lists[l][i];
                    printf("%d ", lists[l][i]);
                }
            }
            printf("\n");

            print_mutex.unlock();

            int stoppedCount = 0;
            for (int i = 1; i < size; ++i)
            {
                if(listSize(lists[i]) != 0)
                {
                    flag = CONTINUE_WORK;
                    sendMsgFlag(flag, i);
                    sendMsgArray(lists[i], TOTAL_LIST_SIZE, i);
                } else
                {
                    stoppedCount++;
                    flag = STOP_WORK;
                    sendMsgFlag(flag, i);
                }

                free(lists[i]);
            }
            size -= stoppedCount;

            unlockWorker(CONTINUE_WORK);
            free(lists);
        }
        threadSavePrint("end rebalance\n", stdout);
```

```
                    break;
                }
                default:
                    threadSavePrint("Invalid receive flag value!", stderr);
                    unlockWorker(STOP_WORK);
                    return;
            }
        } else if(flag == NEED_TASKS)
            sendMsgFlag((flag = TASK_REQUEST_STEP_1), 0);
    }
}

void threadSavePrint(const std::string& message, FILE *file, ...)
{
    va_list list;

    std::lock_guard<std::mutex> locker(print_mutex);

    va_start(list, file);
    vfprintf(file, message.c_str(), list);
    fflush(file);
    va_end(list);
}

void unlockWorker(int message)
{
    std::unique_lock<std::mutex> locker(work_mutex);
    work_flag = message;
    work_notified = true;
    work_wait.notify_all();
}
```

messages.cpp

```
#include "messages.h"


void recvMsgFlag(int &flag, int src)

{

    MPI_Recv(&flag, 1, MPI_INT, src, MPI_LOL_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

}


void recvMsgArray(int* array, int size, int src)

{

    MPI_Recv(array, size, MPI_INT, src, MPI_KEK_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

}


void sendMsgFlag(int &flag, int dest)

{

    MPI_Send(&flag, 1, MPI_INT, dest, MPI_LOL_TAG, MPI_COMM_WORLD);

}


void sendMsgArray(int* array, int size, int dest)

{

    MPI_Send(array, size, MPI_INT, dest, MPI_KEK_TAG, MPI_COMM_WORLD);

}
```

list.h

```
#ifndef MPI_THREADS_LIST_H

#define MPI_THREADS_LIST_H


#include <ctime>

#include <cstring>

#include <climits>

#include <algorithm>

#include <vector>

#include <random>
```

```
#define MAX_COMPUTATIONAL_COMPLEXITY 10

#define I_KNOW_COMPUTATIONAL_COMPLEXITY 1
#define I_DONT_KNOW_COMPUTATIONAL_COMPLEXITY 0

#define TOTAL_LIST_SIZE 20
#define EMPTY_LISTS     0
#define SUCCESS_BALANCE 1

int* initJobList(int rank);
bool nonEmpty(const int* list);
int  popLast(int* list);
int  listSize(const int* list);
int  rebalance(int** lists, int size, int mode);

#endif
```

messages.h

```
#ifndef MPI_THREADS_MESSAGES_H
#define MPI_THREADS_MESSAGES_H

#include <mpi.h>

#define MPI_LOL_TAG 42
#define MPI_KEK_TAG 41

void recvMsgFlag(int &flag, int src);
void recvMsgArray(int* array, int size, int src);

void sendMsgFlag(int &flag, int dest);
void sendMsgArray(int* array, int size, int dest);
```

```
#endif
```

threads.h

```
#ifndef MPI_THREADS_THREADS_H

#define MPI_THREADS_THREADS_H

#include <thread>

#include <mutex>

#include <condition_variable>

#include <cstdarg>

#include "messages.h"

#include "list.h"


#define TASK_REQUEST_STEP_2  4

#define TASK_REQUEST_STEP_1  3

#define NEED_TASKS           2

#define CONTINUE_WORK        1

#define STOP_WORK            0

#define EMPTY  -1


void lockList();

void unlockList();

bool isDead();


void worker      (int* list_ref);

void communicator(int* list_ref, int iKnowCompFlexibility);

void threadSavePrint(const std::string& message, FILE *file, ...);


#endif
```

Компиляция исходного кода проводилась при помощи ключей компиляции -std=c++11 -O3 -pthread.

ЛОГИ РАБОТЫ КЛАСТЕРА

Приведем пример работы кластера с 4 нодами в режиме известной мощности задач. Ниже дан лог кластера:

```
mpirun -n 4 ./MPI_Threads 1
 0 process begin with: 7 18 16 14 16 7 13 10 2 3 8 11 20 4 7 1
 1 process begin with: 17 2 2 8 8 1
 2 process begin with: 6 2 4
 3 process begin with: 13 7 14 16 12 7 10 18 8
--  1 process: begin do new task  1:  5 tasks left
--  3 process: begin do new task  8:  8 tasks left
--  0 process: begin do new task  1: 15 tasks left
--  2 process: begin do new task  4:  2 tasks left
--  1 process: begin do new task  8:  4 tasks left
--  0 process: begin do new task  7: 14 tasks left
--  2 process: begin do new task  2:  1 tasks left
--  2 process: begin do new task  6:  0 tasks left
--  3 process: begin do new task 18:  7 tasks left
--  0 process: begin do new task  4: 13 tasks left
--  1 process: begin do new task  8:  3 tasks left
--  0 process: begin do new task 20: 12 tasks left
begin rebalance
rebalanced task lists:
  0 - 18 17 16 3 2
---- lw vs bl: 56 56
  1 - 16 16 14 10
---- lw vs bl: 56 56
  2 - 14 13 13 12 2 2
---- lw vs bl: 56 56
  3 - 11 10 8 7 7 7 7
---- lw vs bl: 57 56
end rebalance
--  2 process: begin do new task  2:  5 tasks left
--  2 process: begin do new task  2:  4 tasks left
--  2 process: begin do new task 12:  3 tasks left
--  1 process: begin do new task 10:  3 tasks left
--  3 process: begin do new task  7:  6 tasks left
--  1 process: begin do new task 14:  2 tasks left
--  2 process: begin do new task 13:  2 tasks left
--  0 process: begin do new task  2:  4 tasks left
--  3 process: begin do new task  7:  5 tasks left
--  0 process: begin do new task  3:  3 tasks left
--  0 process: begin do new task 16:  2 tasks left
--  3 process: begin do new task  7:  4 tasks left
--  1 process: begin do new task 16:  1 tasks left
--  2 process: begin do new task 13:  1 tasks left
--  3 process: begin do new task  7:  3 tasks left
======Add new tasks to node  0
======New list of  0:
```

```
18 17 13 12 8 3 3 8 10
======Add new tasks to node  1
======New list of  1:
16 19 15 10 10 6 20 17 19
======Add new tasks to node  2
======New list of  2:
14 4 6 1 18 6 7 17 11
======Add new tasks to node  3
======New list of  3:
11 10 8 6 10 14 11 11 2 17
--  0 process: begin do new task 10:  8 tasks left
--  3 process: begin do new task 17:  9 tasks left
--  2 process: begin do new task 11:  8 tasks left
--  1 process: begin do new task 19:  8 tasks left
--  0 process: begin do new task  8:  7 tasks left
--  2 process: begin do new task 17:  7 tasks left
--  3 process: begin do new task  2:  8 tasks left
--  0 process: begin do new task  3:  6 tasks left
--  3 process: begin do new task 11:  7 tasks left
--  0 process: begin do new task  3:  5 tasks left
--  1 process: begin do new task 17:  7 tasks left
--  0 process: begin do new task  8:  4 tasks left
--  2 process: begin do new task  7:  6 tasks left
--  3 process: begin do new task 11:  6 tasks left
--  0 process: begin do new task 12:  3 tasks left
--  2 process: begin do new task  6:  5 tasks left
--  1 process: begin do new task 20:  6 tasks left
--  3 process: begin do new task 14:  5 tasks left
--  2 process: begin do new task 18:  4 tasks left
--  0 process: begin do new task 13:  2 tasks left
--  3 process: begin do new task 10:  4 tasks left
--  0 process: begin do new task 17:  1 tasks left
--  1 process: begin do new task  6:  5 tasks left
--  2 process: begin do new task  1:  3 tasks left
--  2 process: begin do new task  6:  2 tasks left
--  1 process: begin do new task 10:  4 tasks left
--  3 process: begin do new task  6:  3 tasks left
--  2 process: begin do new task  4:  1 tasks left
--  2 process: begin do new task 14:  0 tasks left
--  3 process: begin do new task  8:  2 tasks left
--  0 process: begin do new task 18:  0 tasks left
--  1 process: begin do new task 10:  3 tasks left
--  3 process: begin do new task 10:  1 tasks left
begin rebalance
rebalanced task lists:
  0 - 15
---- lw vs bl: 15 15
  1 - 16
---- lw vs bl: 16 15
  2 - 11 19
---- lw vs bl: 30 15
```

```
  3 - empty
end rebalance
--  2 process: begin do new task 19:  1 tasks left
--  1 process: begin do new task 16:  0 tasks left
--  3 process: shutting down
--  0 process: begin do new task 15:  0 tasks left
======Add new tasks to node  0
======New list of  0:
19 8 17 3 14 20 18 5
======Add new tasks to node  1
======New list of  1:
11 7 8 3 10 15 5 20 6
======Add new tasks to node  2
======New list of  2:
11 3 18 8 5 13 11 15 9 10
--  1 process: begin do new task  6:  8 tasks left
--  2 process: begin do new task 10:  9 tasks left
--  0 process: begin do new task  5:  7 tasks left
--  1 process: begin do new task 20:  7 tasks left
--  0 process: begin do new task 18:  6 tasks left
--  2 process: begin do new task  9:  8 tasks left
--  2 process: begin do new task 15:  7 tasks left
--  1 process: begin do new task  5:  6 tasks left
--  0 process: begin do new task 20:  5 tasks left
--  1 process: begin do new task 15:  5 tasks left
--  2 process: begin do new task 11:  6 tasks left
--  1 process: begin do new task 10:  4 tasks left
--  2 process: begin do new task 13:  5 tasks left
--  0 process: begin do new task 14:  4 tasks left
--  1 process: begin do new task  3:  3 tasks left
--  1 process: begin do new task  8:  2 tasks left
--  2 process: begin do new task  5:  4 tasks left
--  0 process: begin do new task  3:  3 tasks left
--  0 process: begin do new task 17:  2 tasks left
--  2 process: begin do new task  8:  3 tasks left
--  1 process: begin do new task  7:  1 tasks left
--  2 process: begin do new task 18:  2 tasks left
--  1 process: begin do new task 11:  0 tasks left
--  0 process: begin do new task  8:  1 tasks left
begin rebalance
rebalanced task lists:
  0 - 11
---- lw vs bl: 11 11
  1 - 3 19
---- lw vs bl: 22 11
  2 - empty
end rebalance
--  1 process: begin do new task 19:  1 tasks left
--  0 process: begin do new task 11:  0 tasks left
--  2 process: shutting down
begin rebalance
```

```
rebalanced task lists:
  0 - 3
---- lw vs bl:  3  1
  1 - empty
end rebalance
--  0 process: begin do new task  3:  0 tasks left
begin rebalance
end rebalance: empty all
closing cluster...
--  1 process: shutting down
--  0 process: shutting down
Time elapsed: 259, max time: 318
```

Теперь же запустим кластер с режимом неизвестной мощности задач:

```
mpirun -n 4 ./MPI_Threads 0
 1 process begin with: 5 6 14
 2 process begin with: 13 16 5 19 9 18 14 9 15 18
 0 process begin with: 7 18 16 14 16 7 13 10 2 3 8 11 20 4 7 1
 3 process begin with: 13 18
--  0 process: begin do new task  1: 15 tasks left
--  2 process: begin do new task 18:  9 tasks left
--  1 process: begin do new task 14:  2 tasks left
--  3 process: begin do new task 18:  1 tasks left
--  0 process: begin do new task  7: 14 tasks left
--  0 process: begin do new task  4: 13 tasks left
--  0 process: begin do new task 20: 12 tasks left
--  1 process: begin do new task  6:  1 tasks left
--  2 process: begin do new task 15:  8 tasks left
--  3 process: begin do new task 13:  0 tasks left
--  1 process: begin do new task  5:  0 tasks left
begin rebalance
rebalanced task lists:
  0 - 19 16 13 9 7
---- lw vs bl: 64 57
  1 - 18 16 13 9 5
---- lw vs bl: 61 57
  2 - 18 14 11 8 3
---- lw vs bl: 54 57
  3 - 16 14 10 7 2
---- lw vs bl: 49 57
end rebalance
--  1 process: begin do new task  5:  4 tasks left
--  1 process: begin do new task  9:  3 tasks left
--  3 process: begin do new task  2:  4 tasks left
--  0 process: begin do new task  7:  4 tasks left
--  2 process: begin do new task  3:  4 tasks left
--  3 process: begin do new task  7:  3 tasks left
--  2 process: begin do new task  8:  3 tasks left
--  0 process: begin do new task  9:  3 tasks left
```

```
--  1 process: begin do new task 13:  2 tasks left
--  3 process: begin do new task 10:  2 tasks left
--  2 process: begin do new task 11:  2 tasks left
--  0 process: begin do new task 13:  2 tasks left
--  3 process: begin do new task 14:  1 tasks left
--  1 process: begin do new task 16:  1 tasks left
--  2 process: begin do new task 14:  1 tasks left
======Add new tasks to node  0
======New list of  0:
19 16 13 12 8 3 3
======Add new tasks to node  1
======New list of  1:
18 19 19 16 14
======Add new tasks to node  2
======New list of  2:
18 19 16 7 6
======Add new tasks to node  3
======New list of  3:
16 2 16 9 13 2
--  0 process: begin do new task  3:  6 tasks left
--  3 process: begin do new task  2:  5 tasks left
--  0 process: begin do new task  3:  5 tasks left
--  3 process: begin do new task 13:  4 tasks left
--  0 process: begin do new task  8:  4 tasks left
--  1 process: begin do new task 14:  4 tasks left
--  2 process: begin do new task  6:  4 tasks left
--  2 process: begin do new task  7:  3 tasks left
--  0 process: begin do new task 12:  3 tasks left
--  3 process: begin do new task  9:  3 tasks left
--  2 process: begin do new task 16:  2 tasks left
--  1 process: begin do new task 16:  3 tasks left
--  0 process: begin do new task 13:  2 tasks left
--  3 process: begin do new task 16:  2 tasks left
--  2 process: begin do new task 19:  1 tasks left
--  1 process: begin do new task 19:  2 tasks left
--  0 process: begin do new task 16:  1 tasks left
--  3 process: begin do new task  2:  1 tasks left
--  3 process: begin do new task 16:  0 tasks left
--  0 process: begin do new task 19:  0 tasks left
--  2 process: begin do new task 18:  0 tasks left
--  1 process: begin do new task 19:  1 tasks left
======Add new tasks to node  1
======New list of  1:
18 4 19 13 1
begin rebalance
rebalanced task lists:
  0 - 19 1
---- lw vs bl: 20 13
  1 - 18
---- lw vs bl: 18 13
  2 - 13
```

```
---- lw vs bl: 13 13
  3 - 4
---- lw vs bl:  4 13
end rebalance
--  3 process: begin do new task  4:  0 tasks left
begin rebalance
rebalanced task lists:
  0 - 19
---- lw vs bl: 19 12
  1 - 18
---- lw vs bl: 18 12
  2 - 13
---- lw vs bl: 13 12
  3 - 1
---- lw vs bl:  1 12
end rebalance
--  3 process: begin do new task  1:  0 tasks left
begin rebalance
rebalanced task lists:
  0 - 19
---- lw vs bl: 19 12
  1 - 18
---- lw vs bl: 18 12
  2 - 13
---- lw vs bl: 13 12
  3 - empty
end rebalance
--  3 process: shutting down
--  2 process: begin do new task 13:  0 tasks left
--  0 process: begin do new task 19:  0 tasks left
--  1 process: begin do new task 18:  0 tasks left
begin rebalance
end rebalance: empty all
closing cluster...
--  2 process: shutting down
--  0 process: shutting down
--  1 process: shutting down
Time elapsed: 112, max time: 196
```

Здесь за max time обозначено то, сколько времени было бы потрачено без балансировки. За time elapsed - реально потраченное время с балансировкой. Проанализировав логи, можно понять, что любая балансировка в данной задаче хорошо уменьшает общее время работы, но балансировка с известной мощностью в большинстве случаев даёт более лучший результат. Однако в данных примера она оказалась медленней.

ВЫВОД

В ходе лабораторной работы я освоил многопоточное программирование, используя инструменты создания POSIX Threads и MPI. Кластер работает исправно и прекрасно балансирует поступившие задачи в любом доступном режиме.