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. C++ Threads

студента 2 курса, группы 23201

Смирнова Гордея Андреевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.С. Матвеев

Новосибирск 2025

СОДЕРЖАНИЕ

ЦЕЛЬ

Освоить разработку многопоточных программ с использованием C++ Threads.
Познакомиться с задачей динамического распределения работы между процессорами.

ЗАДАНИЕ

Есть список неделимых заданий, каждое из которых может быть выполнено независимо от другого. Как формируется задание, см. ниже. 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. Всего потоков должно быть 3:

- поток, который обрабатывает задания;
- поток, ожидающий запросов о работе от других компьютеров;
- поток, выполняющий подкачку работ на компьютер.

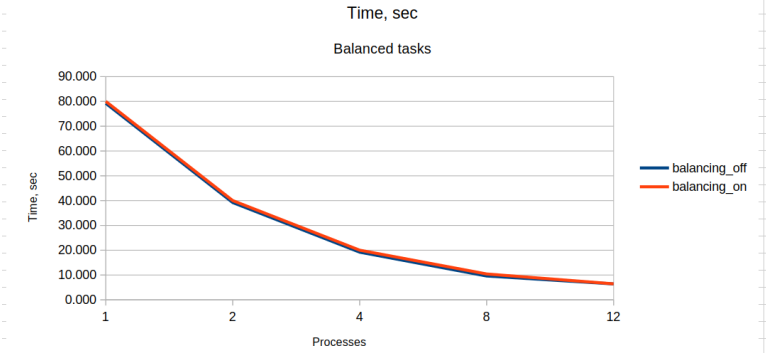
ОПИСАНИЕ РАБОТЫ

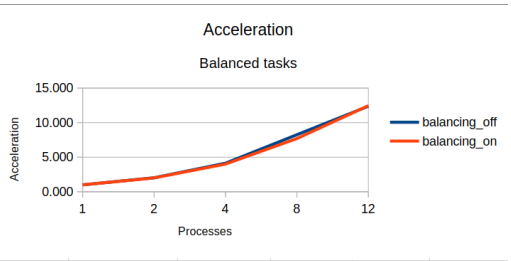
На языке C++ была написана MPI-программа для распределения пула задач, имитирующих работу (в данном случае – спать N миллисекунд). Вариант без балансировки включает в себя только поток worker, последовательно выполняющий задачи из списка, процессы между собой задачами не обмениваются. Вариант с балансировкой дополнительно включает потоки receiver и sender, обеспечивающие равномерное распределение задач между процессами.

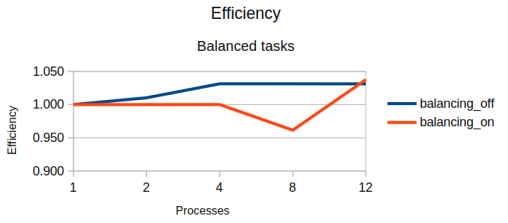
Были произведены замеры производительности с несколькими вариантами заполнения пулла задач: UNBALANCED (все задачи изначально находятся на одном процессе), BALANCED (на каждом процессе одинаковый объем задач) и MODERATE (на каждом процессе изначально свое кол-во процессов, квадратично зависящее от `abs(rank-(iter%commSize))`, при этом суммарное кол-во заданий тем больше, чем больше число процессов).

Balancing OFF			
Processes	Time, sec	Acceleration	Efficiency
1	79.223	1.000	1.000
2	39.209	2.021	1.010
4	19.205	4.125	1.031
8	9.603	8.250	1.031
12	6.402	12.374	1.031

Balancing ON			
Processes	Time, sec	Acceleration	Efficiency
1	80.014	1.000	1.000
2	40.009	2.000	1.000
4	20.003	4.000	1.000
8	10.403	7.692	0.961
12	6.427	12.451	1.038

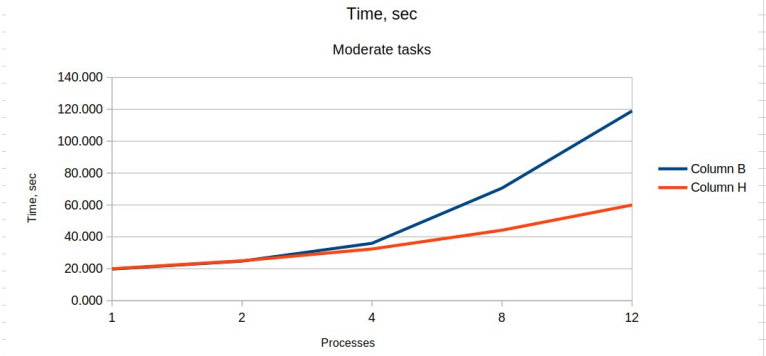


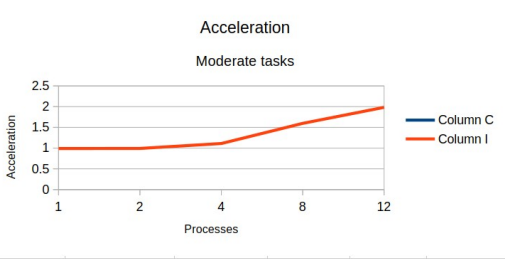


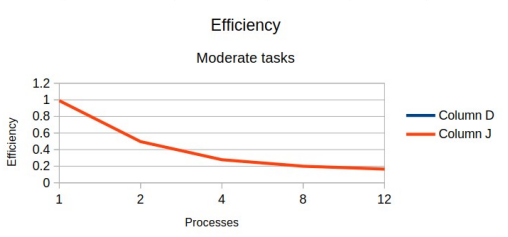


Balancing OFF			
Processes	Time, sec	Acceleration	Efficiency
1	19.817	Here, acceleration and	
2	24.812	efficiency indicators are	
4	36.005	measured relative to Balancing	
8	70.602	OFF mode. The total weight of	
12	119.001	tasks for each number of	
		processes is different.	

Balancing ON			
Processes	Time, sec	Acceleration	Efficiency
1	20.013	0.990	0.990
2	25.008	0.992	0.496
4	32.407	1.111	0.278
8	44.205	1.597	0.200
12	60.005	1.983	0.165

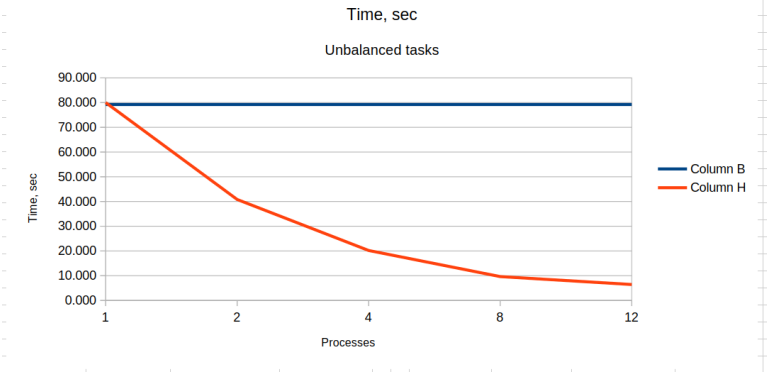


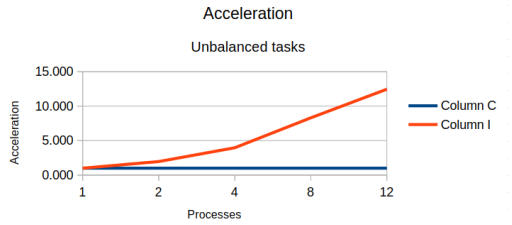


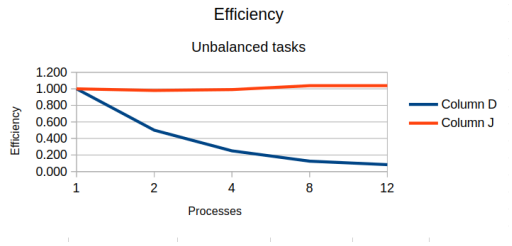


Balancing OFF			
Processes	Time, sec	Acceleration	Efficiency
1	79.220	1.000	1.000
2	79.220	1.000	0.500
4	79.220	1.000	0.250
8	79.220	1.000	0.125
12	79.220	1.000	0.083

Balancing ON			
Processes	Time, sec	Acceleration	Efficiency
1	80.014	1.000	1.000
2	40.807	1.961	0.980
4	20.187	3.964	0.991
8	9.632	8.307	1.038
12	6.422	12.460	1.038







По результатам замеров установлено, что в случае BALANCED балансировка практически не вызывает дополнительных расходов, в случае MODERATE балансировка заметно помогает при большом количестве процессов (а следовательно и задач), а в случае UNBALANCED балансировка позволяет многократно увеличить производительность, задействуя остальные процессы, на которых изначально нет задач.

ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы были изучены функции C++ Threads, позволяющие организовывать многопоточную работу приложений.

ПРИЛОЖЕНИЯ

Приложение 1: *TaskManager.h*

```
#pragma once

#include <mpi.h>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <thread>

class TaskManager {
    enum QueueFill { BALANCED, MODERATE, UNBALANCED };

    static constexpr QueueFill QUEUE_FILL = MODERATE;
    static constexpr bool BALANCING_ENABLED = true;
    static constexpr int TOTAL_TASK_COUNT = 100;
    static constexpr int BASE_WEIGHT = 200;  // ms
    static constexpr int BALANCED_FACTOR = 4;
    static constexpr int UNBALANCED_FACTOR = 4;

    static constexpr int REQUEST_TAG = 0;
    static constexpr int RESPONSE_TAG = 1;
    static constexpr int EMPTY_QUEUE_RESPONSE = -1;
    static constexpr int FINISH_TAG = -1;
    MPI_Comm COMM;

    class ThreadSafeQueue {
     public:
        void push(int value);
        int pop();
        bool empty();
        size_t getSize();
        void setFinish();

     private:
        std::queue<int> data;
        std::mutex qMutex;
        std::condition_variable qCondVar;
        bool qFinish = false;
    };

    int procID, commSize;
    std::mutex rMutex;
    std::condition_variable rCondVar;
    bool rFinish = false;

    ThreadSafeQueue taskQueue;
    std::thread workerThread;
    std::thread receiverThread;
    std::thread senderThread;
```

```
    void workerRun();
    void receiverRun();
    void senderRun();

    void fillBalanced();
    void fillModerate();
    void fillUnbalanced();

 public:
    explicit TaskManager(MPI_Comm comm);
    void run();
};
```

Приложение 2: *TaskManager.cpp*

```
#include "TaskManager.h"
#include <iostream>
#include <chrono>

void TaskManager::ThreadSafeQueue::push(int value) {
    std::lock_guard lock(qMutex);
    data.push(value);
    qCondVar.notify_one();
}

int TaskManager::ThreadSafeQueue::pop() {
    if constexpr (BALANCING_ENABLED) {
        std::unique_lock lock(qMutex);
        qCondVar.wait(lock, [this]{ return !data.empty() || qFinish; });

        if (qFinish) return FINISH_TAG;

        const int value = data.front();
        data.pop();
        return value;
    } else {
        const int value = data.front();
        data.pop();
        return value;
    }
}

bool TaskManager::ThreadSafeQueue::empty() {
    std::lock_guard lock(qMutex);
    return data.empty();
}

size_t TaskManager::ThreadSafeQueue::getSize() {
    std::lock_guard lock(qMutex);
    return data.size();
}
```

```
void TaskManager::ThreadSafeQueue::setFinish() {
    std::lock_guard lock(qMutex);
    qFinish = true;
    qCondVar.notify_all();
}

void TaskManager::workerRun() {
    while (true) {
        if (taskQueue.empty()) {
            if constexpr (BALANCING_ENABLED) {
                std::lock_guard lock(rMutex);
                rCondVar.notify_one();
            } else {
                break;
            }
        }

        int weight = taskQueue.pop();
        if (weight == FINISH_TAG) break;

        if (taskQueue.empty()) {
            if constexpr (BALANCING_ENABLED) {
                std::lock_guard lock(rMutex);
                rCondVar.notify_one();
            } else {
                break;
            }
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(weight));
        std::cout << "Worker " << procID << " slept for " << weight <<
            "ms, tasks left: " << taskQueue.getSize() << std::endl;
    }
    std::cout << "Worker " << procID << " finished" << std::endl;
}

void TaskManager::receiverRun() {
    while (true) {
        if (!taskQueue.empty()) {
            std::unique_lock lock(rMutex);
            rCondVar.wait(lock, [this]
                { return taskQueue.empty() || rFinish; });
        }
        if (rFinish) break;

        int recvCount = 0;
        for (int sendrecvID = 0; sendrecvID < commSize; ++sendrecvID) {
            if (sendrecvID == procID) continue;
            MPI_Send(&procID, 1, MPI_INT, sendrecvID, REQUEST_TAG, COMM);
            // std::cout << "Receiver " << procID << " sent request " <<
            //         "to sender " << sendrecvID << std::endl;

            MPI_Request req;
            int weight, isReceived;
```

```
            MPI_Irecv(&weight, 1, MPI_INT, sendrecvID, RESPONSE_TAG, COMM,
                &req);

            do {
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
                MPI_Test(&req, &isReceived, MPI_STATUS_IGNORE);
            } while (!isReceived && !rFinish);

            if (rFinish) break;

            if (weight != EMPTY_QUEUE_RESPONSE) {
                recvCount++;
                taskQueue.push(weight);
                std::cout << "Receiver " << procID << " received " << weight
<<
                    "ms task from sender " << sendrecvID << std::endl;
            } else {
                // std::cout << "Receiver " << procID << " received empty
queue "
                //     "response from sender " << sendrecvID << std::endl;
            }
        }
        if (recvCount == 0) break;
    }
    taskQueue.setFinish();  // makes own worker finish
    for (int finID = 0; finID < commSize; ++finID) {
        MPI_Send(&FINISH_TAG, 1, MPI_INT, finID, REQUEST_TAG, COMM);
        // makes all senders finish
    }
    std::cout << "Receiver " << procID << " finished" << std::endl;
}

void TaskManager::senderRun() {
    int sendrecvID;
    while (true) {
        MPI_Recv(&sendrecvID, 1, MPI_INT, MPI_ANY_SOURCE,
            REQUEST_TAG, COMM, MPI_STATUS_IGNORE);
        if (sendrecvID == FINISH_TAG) break;
        // std::cout << "Sender " << procID << " received "
        //     "request from receiver " << sendrecvID << std::endl;

        if (!taskQueue.empty()) {
            int weight = taskQueue.pop();
            MPI_Send(&weight, 1, MPI_INT, sendrecvID,
                RESPONSE_TAG, COMM);
            // std::cout << "Sender " << procID << " sent " << weight <<
            //     "ms task to receiver " << sendrecvID << std::endl;
        } else {
            MPI_Send(&EMPTY_QUEUE_RESPONSE, 1, MPI_INT, sendrecvID,
                RESPONSE_TAG, COMM);
            // std::cout << "Sender " << procID << " sent empty queue "
            //     "response to receiver " << sendrecvID << std::endl;
        }
    }
```

```
    {
        std::unique_lock lock(rMutex);
        rFinish = true;  // makes own receiver finish
        rCondVar.notify_all();
    }
    std::cout << "Sender " << procID << " finished" << std::endl;
}

void TaskManager::fillBalanced() {
    const int taskCount = TOTAL_TASK_COUNT / commSize +
        (procID < TOTAL_TASK_COUNT % commSize);
    for (int i = 0; i < taskCount; ++i) {
        taskQueue.push(BASE_WEIGHT * BALANCED_FACTOR);
    }
    if (procID == 0) std::cout << "Generated balanced tasks" << std::endl;
}

void TaskManager::fillModerate() {
    const int taskCount = TOTAL_TASK_COUNT / commSize +
        (procID < TOTAL_TASK_COUNT % commSize);
    for (int i = 0; i < taskCount; ++i) {
        const int factor = std::abs(i % commSize - procID) + 1;
        taskQueue.push(BASE_WEIGHT * factor * factor);
    }
    if (procID == 0) std::cout << "Generated moderate tasks" << std::endl;
}

void TaskManager::fillUnbalanced() {
    if (procID == 0) {
        for (int i = 0; i < TOTAL_TASK_COUNT; ++i) {
            taskQueue.push(BASE_WEIGHT * UNBALANCED_FACTOR);
        }
        std::cout << "Generated unbalanced tasks" << std::endl;
    }
}

TaskManager::TaskManager(MPI_Comm comm): COMM(comm) {
    MPI_Comm_rank(COMM, &procID);
    MPI_Comm_size(COMM, &commSize);

    switch (QUEUE_FILL) {
        case BALANCED: fillBalanced(); break;
        case MODERATE: fillModerate(); break;
        case UNBALANCED: fillUnbalanced(); break;
    }

    if (procID == 0) {
        std::cout << "Total tasks: " << TOTAL_TASK_COUNT << std::endl;
        std::cout << "Base task weight: " << BASE_WEIGHT << std::endl;
    }
}

void TaskManager::run() {
    MPI_Barrier(COMM);
```

```
    const double start = MPI_Wtime();

    workerThread = std::thread(&TaskManager::workerRun, this);
    if constexpr (BALANCING_ENABLED) {
        receiverThread = std::thread(&TaskManager::receiverRun, this);
        senderThread = std::thread(&TaskManager::senderRun, this);
        if (procID == 0) std::cout << "Running w/ balancing..." <<
std::endl;
    } else {
        if (procID == 0) std::cout << "Running w/o balancing..." <<
std::endl;
    }

    workerThread.join();
    if constexpr (BALANCING_ENABLED) {
        receiverThread.join();
        senderThread.join();
    }

    std::cout << "Process " << procID << " finished" << std::endl;

    MPI_Barrier(COMM);
    const double end = MPI_Wtime();
    if (procID == 0) std::cout << "Time taken: " << end - start <<
std::endl;
}
```

Приложение 3: *main.cpp*

```
.#include <mpi.h>
#include "TaskManager.h"

int main(int argc, char** argv) {
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    TaskManager manager(MPI_COMM_WORLD);
    manager.run();

    MPI_Finalize();
    return 0;
}
```

Приложение 4: *CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.28)
project(lab5)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
set(CMAKE_CXX_FLAGS "-Wall -Wextra -Werror -pedantic")
find_package(MPI REQUIRED)

set(SRC
        src/TaskManager.cpp
)

add_executable(${PROJECT_NAME} ${SRC} src/main.cpp)
target_link_libraries(${PROJECT_NAME} PRIVATE MPI::MPI_CXX)
```