МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"

Кафедра 806 "Вычислительная математика и программирование"

Лабораторная работа №2

По курсу «Операционные системы»

Студент: Снетков Н.С.

Группа: М8О-203Б-23

Вариант: 2

Преподаватель: Миронов Е. С.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2025

**Содержание**

1. Репозиторий

2. Постановка задачи

3. Общие сведения о программе

4. Общий метод и алгоритм решения

5. Исходный код

6. Сборка программы

7. Демонстрация работы программы

8. Выводы

**Репозиторий**

https://github.com/Sapfir7/labs\_os\_snet/tree/main/os\_labs/lab2

**Постановка задачи**

**Цель работы**

Целью является приобретение практических навыков в:

Управление потоками в ОС

Обеспечение синхронизации между потоками

**Задание**

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При  
обработки использовать стандартные средства создания потоков операционной системы  
(Windows/Unix). Ограничение максимального количества потоков, работающих в один момент  
времени, должно быть задано ключом запуска вашей программы.  
Так же необходимо уметь продемонстрировать количество потоков, используемое вашей  
программой с помощью стандартных средств операционной системы.  
В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных  
данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант задания:

2.Отсортировать массив целых чисел при помощи параллельного алгоритма быстрой сортировки

**Общие сведения о программе**

Программа представляет собой набор инструментов для параллельных вычислений, включающий в себя:

Сложение матриц — функция SumMatrices, которая позволяет складывать две матрицы с использованием многопоточности.

Быстрая сортировка (QuickSort) — класс QuickSort, реализующий параллельную версию алгоритма быстрой сортировки с использованием пула потоков.

Пул потоков (ThreadPool) — класс ThreadPool, который управляет созданием и выполнением задач в многопоточной среде.

Программа использует стандартную библиотеку C++ и POSIX-потоки (pthread) для реализации многопоточности.

**Общий метод и алгоритм решения**

**Быстрая сортировка (QuickSort)**

Класс QuickSort реализует параллельную версию алгоритма быстрой сортировки. Основной метод заключается в рекурсивном разделении массива на подмассивы с использованием опорного элемента (pivot). Если размер подмассива превышает 1000 элементов, рекурсивные вызовы сортировки выполняются в отдельных потоках через пул потоков. Для небольших подмассивов (размером меньше или равным 1000 элементов) сортировка выполняется в текущем потоке. После завершения всех задач в пуле потоков массив считается отсортированным. Этот подход позволяет эффективно использовать многопоточность для ускорения сортировки больших массивов.

**Пул потоков (ThreadPool)**

Класс ThreadPool управляет созданием и выполнением задач в многопоточной среде. При создании пула инициализируется фиксированное количество потоков, которые ожидают появления задач в очереди. Когда задача добавляется в очередь, один из потоков забирает ее и выполняет. После выполнения задачи поток уведомляет пул о завершении и продолжает ожидать новые задачи. Синхронизация потоков обеспечивается с использованием мьютексов и условных переменных. Пул потоков позволяет эффективно распределять задачи между потоками и управлять их выполнением.

**Исходный код**

**quicksort.h:**

#ifndef QUICKSORT\_H

#define QUICKSORT\_H

#include <vector>

#include "thread\_pool.h"

class QuickSort {

public:

explicit QuickSort(size\_t maxThreads);

void sort(std::vector<int>& arr);

private:

ThreadPool threadPool;

void parallelQuickSort(std::vector<int>& arr, int left, int right);

int partition(std::vector<int>& arr, int left, int right);

};

#endif

**quicksort.cpp:**  
  
#include "quicksort.h"

QuickSort::QuickSort(size\_t maxThreads) : threadPool(maxThreads) {}

void QuickSort::sort(std::vector<int>& arr) {

parallelQuickSort(arr, 0, arr.size() - 1);

threadPool.waitForCompletion();

}

void QuickSort::parallelQuickSort(std::vector<int>& arr, int left, int right) {

if (left < right) {

int pivotIndex = partition(arr, left, right);

int pivotIndexLeft = pivotIndex - 1;

int pivotIndexRight = pivotIndex + 1;

if ((right - left) > 1000) {

threadPool.enqueueTask([this, &arr, left, pivotIndexLeft]() {

parallelQuickSort(arr, left, pivotIndexLeft);

});

threadPool.enqueueTask([this, &arr, pivotIndexRight, right]() {

parallelQuickSort(arr, pivotIndexRight, right);

});

} else {

parallelQuickSort(arr, left, pivotIndexLeft);

parallelQuickSort(arr, pivotIndexRight, right);

}

}

}

int QuickSort::partition(std::vector<int>& arr, int left, int right) {

int pivot = arr[right];

int i = left - 1;

for (int j = left; j <= right - 1; ++j) {

if (arr[j] < pivot) {

++i;

std::swap(arr[i], arr[j]);

}

}

std::swap(arr[i + 1], arr[right]);

return i + 1;

}

**thread\_pool.h**

#ifndef THREAD\_POOL\_H

#define THREAD\_POOL\_H

#include <pthread.h>

#include <queue>

#include <vector>

#include <functional>

class ThreadPool {

public:

explicit ThreadPool(size\_t maxThreads);

~ThreadPool();

void enqueueTask(std::function<void()> task);

void waitForCompletion();

private:

std::vector<pthread\_t> workers;

std::queue<std::function<void()>> tasks;

pthread\_mutex\_t queueMutex;

pthread\_cond\_t condition;

size\_t activeTasks;

bool stop;

static void\* workerThread(void\* arg);

void processTasks();

void taskCompleted();

};

#endif

**thread\_pool.cpp**

#include "thread\_pool.h"

#include <stdexcept>

ThreadPool::ThreadPool(size\_t maxThreads) : activeTasks(0), stop(false) {

pthread\_mutex\_init(&queueMutex, nullptr);

pthread\_cond\_init(&condition, nullptr);

for (size\_t i = 0; i < maxThreads; ++i) {

pthread\_t thread;

if (pthread\_create(&thread, nullptr, workerThread, this) != 0) {

throw std::runtime\_error("Failed to create thread");

}

workers.push\_back(thread);

}

}

ThreadPool::~ThreadPool() {

{

pthread\_mutex\_lock(&queueMutex);

stop = true;

pthread\_mutex\_unlock(&queueMutex);

pthread\_cond\_broadcast(&condition);

}

for (pthread\_t &worker : workers) {

pthread\_join(worker, nullptr);

}

pthread\_mutex\_destroy(&queueMutex);

pthread\_cond\_destroy(&condition);

}

void ThreadPool::enqueueTask(std::function<void()> task) {

pthread\_mutex\_lock(&queueMutex);

tasks.push(std::move(task));

++activeTasks;

pthread\_mutex\_unlock(&queueMutex);

pthread\_cond\_signal(&condition);

}

void ThreadPool::taskCompleted() {

pthread\_mutex\_lock(&queueMutex);

--activeTasks;

if (activeTasks == 0 && tasks.empty()) {

pthread\_cond\_broadcast(&condition);

}

pthread\_mutex\_unlock(&queueMutex);

}

void ThreadPool::waitForCompletion() {

pthread\_mutex\_lock(&queueMutex);

while (activeTasks > 0 || !tasks.empty()) {

pthread\_cond\_wait(&condition, &queueMutex);

}

pthread\_mutex\_unlock(&queueMutex);

}

void\* ThreadPool::workerThread(void\* arg) {

auto\* pool = static\_cast<ThreadPool\*>(arg);

pool->processTasks();

return nullptr;

}

void ThreadPool::processTasks() {

while (true) {

std::function<void()> task;

{

pthread\_mutex\_lock(&queueMutex);

while (!stop && tasks.empty()) {

pthread\_cond\_wait(&condition, &queueMutex);

}

if (stop && tasks.empty()) {

pthread\_mutex\_unlock(&queueMutex);

return;

}

task = std::move(tasks.front());

tasks.pop();

pthread\_mutex\_unlock(&queueMutex);

}

task();

taskCompleted();

}

}

**lab3.cpp**

#include "lab3.h"

#include "utils.h"

#include <thread>

namespace {

void SumGivenRows(const TMatrix& lhs, const TMatrix& rhs, TMatrix& result, int firstRow, int lastRow) {

int m = isize(lhs);

for(int i = firstRow; i < lastRow; ++i) {

for(int j = 0; j < m; ++j) {

result[i][j] = lhs[i][j] + rhs[i][j];

}

}

}

}

TMatrix SumMatrices(const TMatrix& lhs, const TMatrix& rhs, int threadCount) {

TMatrix result(lhs.size(), std::vector<int>(lhs[0].size()));

if(threadCount > 1) {

int actualThreads = std::min(threadCount, isize(result));

std::vector<std::thread> threads;

threads.reserve(actualThreads);

int rowsPerThread = isize(result) / actualThreads;

for(int i = 0; i < isize(result); i += rowsPerThread) {

if(i + rowsPerThread >= isize(result)) {

threads.emplace\_back(SumGivenRows, std::ref(lhs), std::ref(rhs), std::ref(result), i, isize(result));

} else {

threads.emplace\_back(SumGivenRows, std::ref(lhs), std::ref(rhs), std::ref(result),

i, i + rowsPerThread);

}

}

for(auto& thread : threads) {

thread.join();

}

} else {

SumGivenRows(lhs, rhs, result, 0, lhs.size());

}

return result;

}

**lab3.h**

#ifndef OS\_LABS\_LAB3\_H

#define OS\_LABS\_LAB3\_H

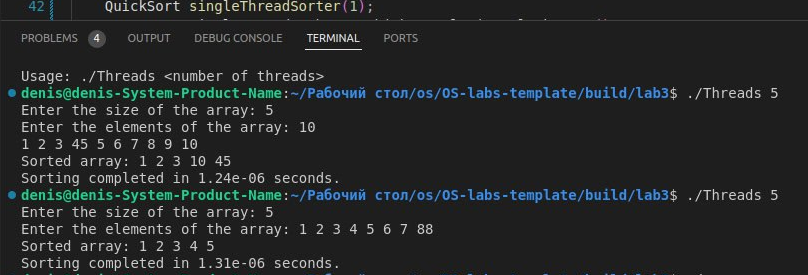
#include <vector>

using TMatrix = std::vector<std::vector<int>>;

TMatrix SumMatrices(const TMatrix& lhs, const TMatrix& rhs, int threadCount);

#endif //OS\_LABS\_LAB3\_H

**Демонстрация работы программы**



**Вывод**

В целом, программа демонстрирует эффективное использование многопоточности для решения вычислительно сложных задач. Она может быть полезна в приложениях, где требуется высокая производительность и параллельная обработка данных.