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: Кириллова Е. К.
Группа: М8О-203Б-23
Вариант: 19
Преподаватель: Миронов Е. С.

Дата: _____

Оценка: _____

Подпись: _____

Москва, 2024

Содержание

Репозиторий

https://github.com/ElenaKirillova05/osLabs/tree/main

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: Вариант 19

Дан массив координат (x, y). Пользователь вводит число кластеров. Проведите кластеризацию методом k-средних

Общие сведения о программе

Программа реализует алгоритм кластеризации **K-means** с поддержкой многопоточности для ускорения вычислений. Она разбивает набор точек на кластеры, где каждая точка относится к ближайшему центроиду. Пользователь задает количество кластеров, точек и потоков, а программа выводит результаты кластеризации. Включены тесты для проверки корректности и сравнения производительности однопоточного и многопоточного режимов. Программа демонстрирует эффективность многопоточности для ускорения вычислений.

Общий метод и алгоритм решения

Программа реализует алгоритм **K-means** для кластеризации точек. Основные шаги:

1. Инициализация центроидов: Случайно выбираются начальные центры кластеров.

2. Назначение точек кластерам: Каждая точка назначается ближайшему центроиду с использованием евклидова расстояния. Этот шаг выполняется параллельно в нескольких потоках.
3. Пересчет центроидов: Центроиды обновляются как среднее значение точек в кластере.
4. Проверка сходимости: Если центроиды не изменились или достигнуто максимальное число итераций, алгоритм завершается. Иначе шаги 2 и 3 повторяются.

Программа поддерживает многопоточность для ускорения вычислений и выводит результаты кластеризации.

Исходный код

KMeans.h:

```
#ifndef KMEANS_H
#define KMEANS_H

#include <vector>
#include "Point.h"

class KMeans {
public:
    KMeans(int k, int maxThreads);
    void run(std::vector<Point>& points);
    void printResults(const std::vector<Point>& points);
private:
    int k;
    int maxThreads;
    std::vector<Point> centroids;
```

```
    struct ThreadData {
        std::vector<Point>* points;
        std::vector<Point>* centroids;
        int start;
        int end;
        int k;
    };

    static void* assignClusters(void* arg);
    void initializeCentroids(std::vector<Point>& points);
};

#endif
```

Point.h:
```
#ifndef POINT_H
#define POINT_H

struct Point {
    double x, y;
    int cluster = -1;

    Point(double x_val = 0.0, double y_val = 0.0) : x(x_val), y(y_val), cluster(-1) {}
};

#endif
```

Timer.h:

```
#ifndef TIMER_H
#define TIMER_H

#include <chrono>
#include <iostream>

class Timer {
public:
    Timer();
    ~Timer();
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> start;
};

#endif
```

Utils.h:

```
#ifndef UTILS_H
#define UTILS_H

#include "Point.h"

double distance(const Point &a, const Point &b);
```

```
#endif
```

KMeans.cpp:

```
#include "KMeans.h"
#include "Utils.h"
#include <pthread.h>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <vector>

KMeans::KMeans(int k, int maxThreads) : k(k), maxThreads(maxThreads) {
    centroids.resize(k);
    std::srand(static_cast<unsigned int>(std::time(0)));
}

void KMeans::initializeCentroids(std::vector<Point>& points) {
    for(int i = 0; i < k; ++i){
        centroids[i] = points[std::rand() % points.size()];
    }
}

void* KMeans::assignClusters(void* arg) {
    ThreadData* data = static_cast<ThreadData*>(arg);
    for (int i = data->start; i < data->end; ++i) {
        double minDist = distance((*data->points)[i], (*data->centroids)[0]);
        int bestCluster = 0;
```

```
        for (int j = 1; j < data->k; ++j) {
            double dist = distance((*data->points)[i], (*data->centroids)[j]);
            if (dist < minDist) {
                minDist = dist;
                bestCluster = j;
            }
        }
        (*data->points)[i].cluster = bestCluster;
    }
    return nullptr;
}

void KMeans::run(std::vector<Point>& points) {
    initializeCentroids(points);

    bool changed;
    int iterations = 0;
    const int maxIterations = 100;

    do {
        changed = false;

        int pointsPerThread = points.size() / maxThreads;
        std::vector<pthread_t> threads(maxThreads);
        std::vector<ThreadData> threadData(maxThreads);

        for (int i = 0; i < maxThreads; ++i) {
            int start = i * pointsPerThread;
            int end;
```

```
        if (i == maxThreads - 1) {
            end = points.size();
        } else {
            end = (i + 1) * pointsPerThread;
        }

        threadData[i] = { &points, &centroids, start, end, k };

        if (pthread_create(&threads[i], nullptr, assignClusters, &threadData[i]) != 0) {
            std::cerr << "Ошибка при создании потока\n";
            exit(1);
        }
    }

    for (int i = 0; i < maxThreads; ++i) {
        pthread_join(threads[i], nullptr);
    }

    std::vector<int> count(k, 0);
    std::vector<double> sumX(k, 0.0), sumY(k, 0.0);

    for (const auto &point : points) {
        sumX[point.cluster] += point.x;
        sumY[point.cluster] += point.y;
        count[point.cluster]++;
    }

    for (int i = 0; i < k; ++i) {
        if (count[i] > 0) {
```

```
            Point newCentroid(sumX[i] / count[i], sumY[i] / count[i]);

            if (centroids[i].x != newCentroid.x || centroids[i].y != newCentroid.y) {
                changed = true;
            }
            centroids[i] = newCentroid;
        }
    }

    iterations++;

  } while (changed && iterations < maxIterations);
}

void KMeans::printResults(const std::vector<Point>& points) {
  for (int i = 0; i < k; ++i) {
    std::cout << "Кластер " << i + 1 << " центр: (" << centroids[i].x << ", " << centroids[i].y << ")\n";
  }

  for (size_t i = 0; i < points.size(); ++i) {
    std::cout << "Точка " << i + 1 << " (" << points[i].x << ", " << points[i].y << ") принадлежит кластеру " << points[i].cluster + 1 << "\n";
  }

  //std::cout << iterations << '\n';
}
```

Timer.cpp:

```
#include "Timer.h"

Timer::Timer(){
    start = std::chrono::high_resolution_clock::now();
}

Timer::~Timer(){
    std::chrono::duration<float> time = std::chrono::high_resolution_clock::now() - start;
    std::cout << "Время выполнения: " << time.count() << " секунд\n";
}
```

Utils.cpp:
```
#include "Utils.h"
#include <cmath>

double distance(const Point &a, const Point &b) {
    return std::sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
```

main.cpp:
```
#include <iostream>
#include <vector>
#include "Point.h"
#include "KMeans.h"
```

```
#include "Timer.h"

int main() {
    int maxThreads;

    std::cout << "Введите максимальное количество потоков:\n";
    std::cin >> maxThreads;

    int n, k;
    std::cout << "Введите количество точек:\n";
    std::cin >> n;

    std::vector<Point> points(n);
    std::cout << "Введите координаты точек (x y):\n";
    for (int i = 0; i < n; ++i) {
        std::cin >> points[i].x >> points[i].y;
    }

    std::cout << "Введите количество кластеров:\n";
    std::cin >> k;

    Timer t;
    KMeans kmeans(k, maxThreads);
    kmeans.run(points);
    //kmeans.printResults(points);

    std::cout << "Используемое количество потоков: " << maxThreads << std::endl;

    return 0;
```

}
```

ТЕСТЫ

test.cpp:
```
#include <gtest/gtest.h>
#include <vector>
#include <fstream>
#include <sstream>
#include <chrono>
#include <filesystem>
#include "KMeans.h"
#include "Point.h"

std::vector<Point> loadPoints(const std::string& relativePath) {
    std::string fullPath = std::filesystem::current_path().string() + "/" + relativePath;
    std::ifstream file(fullPath);
    if (!file.is_open()) {
        throw std::runtime_error("Не удалось открыть файл: " + fullPath);
    }

    std::vector<Point> points;
    int n;
    file >> n;
    for (int i = 0; i < n; ++i) {
        double x, y;
        file >> x >> y;
```
```

```
        points.emplace_back(x, y);
    }

    return points;
}

TEST(Lab2Test, SingleThreadedRun) {
    std::vector<Point> points = loadPoints("files_for_lab2/input_single.txt");

    int k = 2;
    int maxThreads = 1;
    KMeans kmeans(k, maxThreads);

    auto start = std::chrono::high_resolution_clock::now();
    kmeans.run(points);
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> duration = end - start;
    std::cout << "Время выполнения (однопоточный режим): " << duration.count() << " секунд\n";

    for (const auto& point : points) {
        EXPECT_GE(point.cluster, 0) << "Точка не имеет назначенного кластера";
    }
}

TEST(Lab2Test, MultiThreadedRun) {
    std::vector<Point> points = loadPoints("files_for_lab2/speed.txt");
```

```
    int k = 3;
    int maxThreadsSingle = 1;
    KMeans kmeansSingle(k, maxThreadsSingle);

    auto startSingle = std::chrono::high_resolution_clock::now();
    kmeansSingle.run(points);
    auto endSingle = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> durationSingle = endSingle - startSingle;
    std::cout << "Время выполнения (однопоточный режим): " << durationSingle.count() << " секунд\n";

    int maxThreadsMulti = 4;
    KMeans kmeansMulti(k, maxThreadsMulti);

    points = loadPoints("files_for_lab2/speed.txt");

    auto startMulti = std::chrono::high_resolution_clock::now();
    kmeansMulti.run(points);
    auto endMulti = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> durationMulti = endMulti - startMulti;
    std::cout << "Время выполнения (четырехпоточный режим): " << durationMulti.count() << " секунд\n";

    EXPECT_LT(durationMulti.count(), durationSingle.count()) << "Многопоточный режим работает медленнее однопоточного";

    for (const auto& point : points) {
        EXPECT_GE(point.cluster, 0) << "Точка не имеет назначенного кластера";
```

```
    }
}


int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

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.