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»

Группа: М8О-212БВ-24

Студент: Федосов Д.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 18.10.25

Москва, 2025

Постановка задачи

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке 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.

Вариант 19.

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

Общий метод и алгоритм решения

Использованные функции из библиотеки “pthread.h”:

- pthread_create() - запускает новый поток с указанной функцией.
- pthread_join() - приостанавливает основной поток до завершения указанного рабочего потока.

Использованные формулы для расчётов:

Ускорение: S = Ts / Tp, где
Ts – время последовательной реализации,
Tp – время параллельной реализации,
1 <= S <= p,
p – количество ядер
Показывает, во сколько раз параллельная версия быстрее последовательной.

Эффективность: X = S / p, где X < 1
Показывает, насколько эффективно каждое ядро используется.

Алгоритм решения:

1. Создаем некоторое количество потоков.
2. Каждый поток получает свою часть массива точек.
3. Поток для каждой своей точки находит ближайший центроид.
4. Поток локально накапливает суммы координат и счетчик точек для каждого кластера.
5. Основной поток ждет, пока все потоки закончат работу.
6. Основной поток собирает (суммирует) локальные суммы из всех структур в одну общую структуру.
7. Основной поток пересчитывает новые центроиды как среднее арифметическое.
8. Цикл начинается сначала.

Код программы

main.c

```c
#include <stdlib.h>
#include <stdio.h>
```

```c
#include <unistd.h>
#include <pthread.h>

#include <string.h>
#include <math.h>
#include <pthread.h>
#include <linux/time.h>

// Структура точки
typedef struct
{
   double x;
   double y;
   int cluster;
} Point;

// Структура кластера точек
typedef struct
{
   double x;
   double y;
   int pointCount;
} Cluster;

// Структура аргументов потока
typedef struct
{
   int id;
   int clusterCount;
   int pointsCount;
   Point *points;
   Cluster *currCentroids;
   Cluster *clusters;
   int startIndex;
   int endIndex;
```

```
} ThreadArgs;


// Получить разницу во времени
double getTime(struct timespec *start, struct timespec *end)
{
   return (double)(end->tv_sec - start->tv_sec) * 1000 +
(double)(end->tv_nsec - start->tv_nsec) / 1000000.0;
}


// Дистанция между двумя Points
double distance(double x1, double y1, double x2, double y2) {
   return (x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1);
}


// Ближайший центроид
int closestCentroid(Point *p, Cluster *centroids, int clusterCount) {
   double minDist = __DBL_MAX__;
   int closest = -1;


   for (int i = 0; i < clusterCount; i++) {
       double dist = distance(p->x, p->y, centroids[i].x, centroids[i].y);
       if (dist < minDist) {
           minDist = dist;
           closest = i;
       }
   }


   return closest;
}


// Функция для выполнения задания потоками
void *work(void *_args) {
   ThreadArgs *args = (ThreadArgs *)_args;


   for (int k = 0; k < args->clusterCount; k++) {
       args->clusters[k].x = 0.0;
```

```
        args->clusters[k].y = 0.0;
        args->clusters[k].pointCount = 0;
    }

    for (int i = args->startIndex; i < args->endIndex; i++) {
        int closest = closestCentroid(&args->points[i], args->currCentroids,
args->clusterCount);

        args->points[i].cluster = closest;

        args->clusters[closest].x += args->points[i].x;
        args->clusters[closest].y += args->points[i].y;
        args->clusters[closest].pointCount++;
    }

    return NULL;
}

// Функция для выполнения задания последовательно
void classic(Point *points, Cluster *centroids, int clusterCount, int
amountOfPoints, int maxIterations) {
    Cluster *clusters = (Cluster *)calloc(clusterCount, sizeof(Cluster));

    for (int iteration = 0; iteration < maxIterations; iteration++) {
        memset(clusters, 0, clusterCount * sizeof(Cluster));

        for (int i = 0; i < amountOfPoints; i++) {
            int closest = closestCentroid(&points[i], centroids,
clusterCount);
            points[i].cluster = closest;

            clusters[closest].x += points[i].x;
            clusters[closest].y += points[i].y;
            clusters[closest].pointCount++;
        }
```

```
        for (int k = 0; k < clusterCount; k++) {
            if (clusters[k].pointCount > 0) {
                centroids[k].x = clusters[k].x / clusters[k].pointCount;
                centroids[k].y = clusters[k].y / clusters[k].pointCount;
                centroids[k].pointCount = clusters[k].pointCount;
            }
        }
    }

    free(clusters);
}

int main(int argc, char **argv) {
    if (argc != 4) {
        const char msg[] = "Ошибка: ожидается ввод в формате \"./task2
<Количество точек> <Количество кластеров> <Количество потоков>\"\n";
        write(STDERR_FILENO, msg, sizeof(msg));
        exit(EXIT_FAILURE);
    }

    int amountOfPoints = atoi(argv[1]);
    int amountOfAllClusters = atoi(argv[2]);
    int threadsCount = atoi(argv[3]);
    int maxIterations = 10;

    Point *points = (Point *)malloc(amountOfPoints * sizeof(Point));
    Cluster *centroids = (Cluster *)malloc(amountOfAllClusters *
sizeof(Cluster));

    srand(time(NULL));
    for (int i = 0; i < amountOfPoints; i++) {
        points[i].x = (double)(rand() % 1000);
        points[i].y = (double)(rand() % 1000);
    }

    for (int k = 0; k < amountOfAllClusters; k++) {
```

```
        centroids[k].x = points[k].x;

        centroids[k].y = points[k].y;

    }


    Cluster *sameCentroids = (Cluster *)malloc(amountOfAllClusters *
sizeof(Cluster));

    memcpy(sameCentroids, centroids, amountOfAllClusters * sizeof(Cluster));


    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC, &start);


    // Замер последовательного алгоритма


    classic(points, centroids, amountOfAllClusters, amountOfPoints,
maxIterations);


    clock_gettime(CLOCK_MONOTONIC, &end);


    double timeResult1 = getTime(&start, &end);


    clock_gettime(CLOCK_MONOTONIC, &start);


    // Замер параллельного алгоритма


    for (int iteration = 0; iteration < maxIterations; iteration++) {


        pthread_t *threads = (pthread_t *)malloc(threadsCount *
sizeof(pthread_t));

        ThreadArgs *thread_args = (ThreadArgs *)malloc(threadsCount *
sizeof(ThreadArgs));


        for (int i = 0; i < threadsCount; i++) {


            int range = amountOfPoints / threadsCount;


            thread_args[i].startIndex = i * range;

            if (i == threadsCount - 1) {
```

```
                thread_args[i].endIndex = amountOfPoints;
            } else {
                thread_args[i].endIndex = (i + 1) * range;
            }
            thread_args[i].id = i;
            thread_args[i].clusterCount = amountOfAllClusters;
            thread_args[i].pointsCount = amountOfPoints;
            thread_args[i].points = points;
            thread_args[i].currCentroids = sameCentroids;
            thread_args[i].clusters = (Cluster *)malloc(amountOfAllClusters *
sizeof(Cluster));

            pthread_create(&threads[i], NULL, work, &thread_args[i]);
        }

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

        Cluster *sameClusters = (Cluster *)calloc(amountOfAllClusters,
sizeof(Cluster));

        for (int i = 0; i < threadsCount; i++) {
            for (int k = 0; k < amountOfAllClusters; k++) {
                sameClusters[k].x += thread_args[i].clusters[k].x;
                sameClusters[k].y += thread_args[i].clusters[k].y;
                sameClusters[k].pointCount +=
thread_args[i].clusters[k].pointCount;
            }
           free(thread_args[i].clusters);
        }

        for (int k = 0; k < amountOfAllClusters; k++) {
            if (sameClusters[k].pointCount > 0) {
                sameCentroids[k].x = sameClusters[k].x /
sameClusters[k].pointCount;
```

```
                sameCentroids[k].y = sameClusters[k].y /
sameClusters[k].pointCount;
                sameCentroids[k].pointCount = sameClusters[k].pointCount;
            }
        }

        free(sameClusters);
        free(thread_args);
        free(threads);
    }

    clock_gettime(CLOCK_MONOTONIC, &end);
    double timeResult2 = getTime(&start, &end);

    printf("Для последовательного алгоритма время - %.5f \n", timeResult1);

    printf("Для %d потоков время - %.5f \n", threadsCount, timeResult2);

    free(points);
    free(centroids);
    free(sameCentroids);

    return 0;
}
```

Протокол работы программы

```
snadon@fedora:~/OS-Labs/Lab2$ cc main.c -o task2 -pthread -lm
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 2
Для последовательного алгоритма время - 65.50284
Для 2 потоков время - 45.72498
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 4
Для последовательного алгоритма время - 64.89991
Для 4 потоков время - 28.20541
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 8
Для последовательного алгоритма время - 68.45527
Для 8 потоков время - 14.83707
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 16
Для последовательного алгоритма время - 67.69615
Для 16 потоков время - 11.42960
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 64
Для последовательного алгоритма время - 62.77557
Для 64 потоков время - 19.47323
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 256
Для последовательного алгоритма время - 66.16409
Для 256 потоков время - 78.28486
snadon@fedora:~/OS-Labs/Lab2$ ./task2 100000 5 1024
Для последовательного алгоритма время - 65.55833
Для 1024 потоков время - 324.67445
```

Таблица результатов:

Число потоков	Время выполнения (мс)	Ускорение	Эффективность
2	45.72	1.43	0.72
4	28.21	2.30	0.58
8	14.84	4.61	0.58
12	13.14	4.94	0.41
16	11.43	5.92	0.37
64	19.47	3.22	0.05
256	78.28	0.85	0.003
1024	324.67	0.20	0.0002

Вывод

В процессе выполнения лабораторной работы я составил программу на языке Си, обрабатывающую данные в многопоточном режиме. В процессе изучения результатов я на практике убедился, что увеличение числа потоков уменьшает время выполнения программы, поскольку мы используем сразу несколько ядер процессора для вычислений. Однако, при увеличении числа потока выше количества физических ядер выполнение замедляется, так как операционная система вынуждена выполнять множество операций управления ресурсами, что негативно отражается на эффективности решения основной задачи. Ниже представлен график, полученный в результате выполнения задачи, демонстрирующий зависимость эффективности от количества потоков.