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-211БВ-24

Студент: Ергизов Алексей Радикович

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата:25.10.25

Москва, 2025

Постановка задачи

Вариант 19.

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

Общий метод и алгоритм решения

- Задать число кластеров k.
- Случайным образом выбрать k начальных центров кластеров (центроидов).
- Для каждой точки вычислить расстояние до всех центроидов и отнести её к ближайшему кластеру.
- Для каждого кластера пересчитать новый центроид как среднее всех точек, вошедших в него.
- Проверить условие сходимости:

  если центроиды не изменились (или изменения меньше заданного порога) — завершить;

  иначе — перейти к шагу 3.
- Вывести полученные кластеры и их центры.

Использованные системные вызовы:

- `pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)` - создает поток, возвращает 0 при успехе.
- pthread_join(pthread_t thread, void **retval) — блокирует вызывающий поток до завершения указанного потока, возвращает 0 при успехе.

Метод K-средних основан на идее разделения множества точек на k непересекающихся групп (кластеров) таким образом, чтобы точки внутри кластера были как можно ближе друг к другу, а расстояние между кластерами — как можно больше.

Для ускорения вычислений была реализована **параллельная версия** с использованием библиотеки **pthread**.
Каждый поток обрабатывает свою часть массива точек при назначении их к ближайшему центроиду.
Пользователь вводит:

- количество точек,
- количество кластеров.

Далее программа:

1. Генерирует заданное количество точек с произвольными координатами (x,y);

2. Инициализирует центроиды;

3. Итерационно выполняет назначение точек к кластерам и пересчёт центроидов;

4. После завершения итераций выводит координаты финальных центроидов и принадлежность каждой точки к кластеру.

Для оценки эффективности параллельной реализации программы были использованы две основные метрики: **ускорение** и **эффективность**.

Ускорение: $S = T1 / Tp$

S - ускорение

T1 — время выполнения программы в последовательном режиме

Tp — время выполнения программы при p потоках

Если S=2, значит программа при двух потоках работает в два раза быстрее, чем последовательная версия.
Теоретически максимальное ускорение ограничено числом потоков p, но на практике оно обычно меньше из-за накладных расходов на синхронизацию и распределение задач.

Эффективность: $E = S / p$

E - эффективность

S - ускорение

p — количество потоков

На практике эффективность уменьшается с ростом числа потоков из-за:

- затрат на создание и завершение потоков;
- синхронизации данных;
- неравномерного распределения нагрузки между потоками.

Код программ

Параллельный метод:

Centroids.h

```
#ifndef KMEANS_H

#define KMEANS_H

#include <stddef.h>

typedef struct
{
double x;
double y;
int cluster;
} Point;

typedef struct
{
double x;
double y;
} Centroid;

// Функции k-средних
void initialize_points(Point *points, size_t n);
Centroid *initialize_centroids(Point *points, size_t n, size_t k);
void free_centroids(Centroid *centroids);
void assign_points_to_clusters(Point *points, size_t n, Centroid *centroids, size_t k, int
max_threads);
void update_centroids(Point *points, size_t n, Centroid *centroids, size_t k);

#endif
```

Centroids.c

```
#include <stdio.h>

#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include "../include/Centroids.h"

typedef struct
{
Point *points;
size_t start, end;
Centroid *centroids;
size_t k;
} ThreadData;

// Генерация случайных точек
void initialize_points(Point *points, size_t n)
```

```
{
for (size_t i = 0; i < n; i++)
{
points[i].x = (double)rand() / RAND_MAX * 100.0;
points[i].y = (double)rand() / RAND_MAX * 100.0;
points[i].cluster = -1;
}
}

// Инициализация центроидов случайно
Centroid *initialize_centroids(Point *points, size_t n, size_t k)
{
Centroid *centroids = malloc(k * sizeof(Centroid));
for (size_t i = 0; i < k; i++)
{
centroids[i].x = points[i % n].x;
centroids[i].y = points[i % n].y;
}
return centroids;
}

void free_centroids(Centroid *centroids)
{
free(centroids);
}

// Потоковая функция назначения точек к ближайшему центроиду
void *assign_block(void *arg)
{
ThreadData *data = (ThreadData *)arg;
for (size_t i = data->start; i < data->end; i++)
{
double min_dist = INFINITY;
int cluster = 0;
for (size_t j = 0; j < data->k; j++)
{
double dx = data->points[i].x - data->centroids[j].x;
double dy = data->points[i].y - data->centroids[j].y;
double dist = dx * dx + dy * dy;
if (dist < min_dist)
{
min_dist = dist;
cluster = j;
}
}
data->points[i].cluster = cluster;
}
return NULL;
}
```

```
void assign_points_to_clusters(Point *points, size_t n, Centroid *centroids, size_t k, int
max_threads)
{
pthread_t threads[max_threads];
ThreadData thread_data[max_threads];

size_t block = n / max_threads;
for (int t = 0; t < max_threads; t++)
{
thread_data[t].points = points;
thread_data[t].centroids = centroids;
thread_data[t].k = k;
thread_data[t].start = t * block;
thread_data[t].end = (t == max_threads - 1) ? n : (t + 1) * block;
pthread_create(&threads[t], NULL, assign_block, &thread_data[t]);
}

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

// Обновление центроидов
void update_centroids(Point *points, size_t n, Centroid *centroids, size_t k)
{
double *sum_x = calloc(k, sizeof(double));
double *sum_y = calloc(k, sizeof(double));
size_t *count = calloc(k, sizeof(size_t));

for (size_t i = 0; i < n; i++)
{
int c = points[i].cluster;
sum_x[c] += points[i].x;
sum_y[c] += points[i].y;
count[c]++;
}

for (size_t j = 0; j < k; j++)
{
if (count[j] != 0)
{
centroids[j].x = sum_x[j] / count[j];
centroids[j].y = sum_y[j] / count[j];
}
}

free(sum_x);
free(sum_y);
free(count);
```

```
}
```

main.c

```
#define _POSIX_C_SOURCE 199309L

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "../include/Centroids.h"
#include <omp.h>

#define MAX_ITER 50

int input_number(const char *prompt, int min, int max)
{
int value;
printf("%s", prompt);
if (scanf("%d", &value) != 1)
{
printf("Ошибка: введено не число!\n");
exit(1);
}
if (value < min || value > max)
{
printf("Ошибка: значение должно быть от %d до %d\n", min, max);
exit(1);
}
return value;
}

int main()
{
int n_points = input_number("Введите число точек (1000-100000): ", 1000, 100000);
int n_clusters = input_number("Введите число кластеров (1-100): ", 1, 100);

Point *points = malloc(n_points * sizeof(Point));
if (!points)
{
perror("malloc");
exit(1);
}
initialize_points(points, n_points);
Centroid *centroids = initialize_centroids(points, n_points, n_clusters);

int threads_array[] = {1, 2, 4, 8, 16, 128, 1024};
size_t num_experiments = sizeof(threads_array) / sizeof(int);
double times[num_experiments];

printf("Проверка производительности для разных количеств потоков:\n");
```

```
for (size_t t = 0; t < num_experiments; t++)
{
int max_threads = threads_array[t];

struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

for (int iter = 0; iter < MAX_ITER; iter++)
{
omp_set_num_threads(max_threads);
assign_points_to_clusters(points, n_points, centroids, n_clusters, max_threads);
update_centroids(points, n_points, centroids, n_clusters);
}

clock_gettime(CLOCK_MONOTONIC, &end);
double elapsed = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec) / 1e9;
times[t] = elapsed;

printf("Потоки=%d, время=%.6f секунд\n", max_threads, elapsed);
}

double T1 = times[0];
printf("\nЧисло потоков | Время (с) | Ускорение | Эффективность\n");
for (size_t t = 0; t < num_experiments; t++)
{
int p = threads_array[t];
double Sp = T1 / times[t];
double Ep = Sp / p;
printf("%12d | %8.6f | %9.3f | %13.3f\n", p, times[t], Sp, Ep);
}

free(centroids);
free(points);
return 0;
}
```

Последовательная версия:

Centroids.h

```
#ifndef CENTROIDS_H
#define CENTROIDS_H

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>

#define MAX_ITER 100
```

```c
typedef struct Point
{
int x, y;
int cluster;
} Point;

typedef struct Centroid
{
int x, y;
Point *points;
int count_points;
} Centroid;

void Create(Point point[], int count_points);

int distance(Point a, Point b);

void add_point(Centroid *centroid, Point point);

Centroid *Initial_Centroids(Point point[], int count_points, int count_clusters);

void set_centroid(Point point[], int count_points, Centroid centroids[], int count_centroids);

void print_centroids(Centroid *centroids, int count_clusters);

int find_min_index(int distance[], int count_point);

void free_centroids(Centroid *centroids, int count_clusters);

void print_point(Centroid *centroids, int count_centroids);

void Change_Centroids(Centroid *centroids, int count_centroids);

#endif // CENTROIDS_H
```

Centroids.c

```c
#include "../include/Centroids.h"

void Create(Point point[], int count_points)
{
srand(time(NULL));

for (int i = 0; i < count_points; i++)
{
point[i].x = rand() % 100;
point[i].y = rand() % 100;
point[i].cluster = -1;
}
}
```

```
Centroid *Initial_Centroids(Point point[], int count_points, int count_clusters)
{
Centroid *centroids = malloc(count_clusters * sizeof(Centroid));
if (centroids == NULL)
{
printf("Ошибка выделения памяти для центроидов!\n");
return NULL;
}

for (int i = 0; i < count_clusters; i++)
{
int random_index = rand() % count_points;

centroids[i].x = point[random_index].x;
centroids[i].y = point[random_index].y;

centroids[i].points = NULL;
centroids[i].count_points = 0;
}

return centroids;
}

void print_centroids(Centroid *centroids, int count_clusters)
{
printf("\nЦентроиды:\n");
for (int i = 0; i < count_clusters; i++)
{
printf("Центроид №%d: (%d, %d), точек: %d\n",
i, centroids[i].x, centroids[i].y, centroids[i].count_points);
}
}

void add_point_to_centroid(Centroid *centroid, Point point)
{
centroid->count_points++;
Point *new_points = realloc(centroid->points,
centroid->count_points * sizeof(Point));
if (new_points == NULL)
{
printf("Ошибка памяти!\n");
centroid->count_points--;
return;
}
centroid->points = new_points;
centroid->points[centroid->count_points - 1] = point;
}

void set_centroid(Point points[], int count_points, Centroid centroids[], int count_centroids)
{
```

```
for (int j = 0; j < count_centroids; j++)
{
free(centroids[j].points);
centroids[j].points = NULL;
centroids[j].count_points = 0;
}

for (int i = 0; i < count_points; i++)
{
int min_distance = INT_MAX;
int closest_index = 0;

for (int j = 0; j < count_centroids; j++)
{
int dist = distance(points[i],
(Point){centroids[j].x, centroids[j].y, -1});
if (dist < min_distance)
{
min_distance = dist;
closest_index = j;
}
}

points[i].cluster = closest_index;

add_point_to_centroid(&centroids[closest_index], points[i]);
}
}

void Change_Centroids(Centroid *centroids, int count_centroids)
{
for (int i = 0; i < count_centroids; i++)
{
if (centroids[i].count_points > 0)
{
int sum_x = 0;
int sum_y = 0;

for (int j = 0; j < centroids[i].count_points; j++)
{
sum_x += centroids[i].points[j].x;
sum_y += centroids[i].points[j].y;
}

centroids[i].x = sum_x / centroids[i].count_points;
centroids[i].y = sum_y / centroids[i].count_points;
}
}
}
```

```
int distance(Point a, Point b)
{
return ((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

int find_min_index(int distance[], int count_point)
{
if (count_point <= 0)
{
return -1;
}

int min_index = 0;

for (int i = 1; i < count_point; i++)
{
if (distance[i] < distance[min_index])
{
min_index = i;
}
}

return min_index;
}

void free_centroids(Centroid *centroids, int count_clusters)
{
for (int i = 0; i < count_clusters; i++)
{
free(centroids[i].points);
}
free(centroids);
}

void print_point(Centroid *centroids, int count_centroids)
{
printf("\nТочки центроидов: \n");
for (int i = 0; i < count_centroids; i++)
{
printf("\nЦентроид №%d: \n\n", i);
for (int j = 0; j < centroids[i].count_points; j++)
printf("Точка %d: (%d, %d)\n", j,
centroids[i].points[j].x, centroids[i].points[j].y);
}
}
```

main.c

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

```c
#include "../include/Centroids.h"

int input_number(const char *prompt, int min, int max, const char *error_msg)
{
int value;
printf("%s", prompt);

if (scanf("%d", &value) != 1)
{
printf("Ошибка: введено не число!\n");
return -1;
}

if (value < min || value > max)
{
printf("Ошибка: %s\n", error_msg);
printf("Допустимый диапазон: от %d до %d\n", min, max);
return -1;
}
return value;
}

int main(void)
{
int count_points = input_number(
"Введите число точек (2 - 10000): ",
2, 10000,
"число точек должно быть от 2 до 10000");

if (count_points == -1)
return 1;

int count_clusters = input_number(
"Введите число кластеров (1 - 1000): ",
1, 1000,
"число кластеров должно быть от 1 до 1000");

if (count_clusters == -1)
return 1;

if (count_clusters > count_points)
{
printf("Ошибка: кластеров не может быть больше чем точек!\n");
return 1;
}

clock_t start = clock();

Point matrix[count_points];
Create(matrix, count_points);
```

```
Centroid *centroids = Initial_Centroids(matrix, count_points, count_clusters);

set_centroid(matrix, count_points, centroids, count_clusters);

print_centroids(centroids, count_clusters);
print_point(centroids, count_clusters);

for (int i = 0; i < MAX_ITER; i++)
{
Change_Centroids(centroids, count_clusters);
set_centroid(matrix, count_points, centroids, count_clusters);
}

print_centroids(centroids, count_clusters);
print_point(centroids, count_clusters);

free_centroids(centroids, count_clusters);

clock_t end = clock();

double elapsed = (double)(end - start) / CLOCKS_PER_SEC;
printf("\nВремя выполнения программы: %.6f секунд\n", elapsed);

return 0;
}
```

Протокол работы программы

```
@vboxuser bin/main
Введите число точек (1000-100000): 10000
Введите число кластеров (1-100): 100
Проверка производительности для разных количеств потоков:
Потоки=1, время=0.403834 секунд
Потоки=2, время=0.259478 секунд
Потоки=4, время=0.281592 секунд
Потоки=8, время=0.290803 секунд
Потоки=16, время=0.350044 секунд
Потоки=128, время=1.103961 секунд
Потоки=1024, время=9.363638 секунд

Число потоков | Время (с) | Ускорение | Эффективность
            1 | 0.403834 |     1.000 |         1.000
            2 | 0.259478 |     1.556 |         0.778
            4 | 0.281592 |     1.434 |         0.359
            8 | 0.290803 |     1.389 |         0.174
           16 | 0.350044 |     1.154 |         0.072
          128 | 1.103961 |     0.366 |         0.003
         1024 | 9.363638 |     0.043 |         0.000
```

```
Центроид №997:

Точка 0: (27, 86)
Точка 1: (27, 86)
Точка 2: (27, 85)
Точка 3: (26, 85)
Точка 4: (26, 87)
Точка 5: (24, 86)
Точка 6: (24, 86)
Точка 7: (27, 87)
Точка 8: (24, 86)
Точка 9: (24, 85)
Точка 10: (27, 88)
Точка 11: (27, 87)
Точка 12: (26, 87)
Точка 13: (27, 88)
Точка 14: (27, 86)
Точка 15: (27, 88)

Центроид №998:

Точка 0: (0, 60)
Точка 1: (0, 59)
Точка 2: (0, 60)

Центроид №999:

Точка 0: (64, 0)
Точка 1: (64, 0)

Время выполнения программы: 15.721255 секунд
```

1- Поведение при увеличении числа потоков:

При увеличении числа потоков до 4–8 наблюдается

заметное ускорение.
Это объясняется тем, что работа распределяется между несколькими потоками, которые параллельно обрабатывают разные части массива точек.
Потоки эффективно загружают все логические ядра процессора.

- **При дальнейшем росте количества потоков (16 и более) ускорение перестаёт расти и начинает снижаться.**
  Это происходит из-за накладных расходов на управление потоками:

- **создание и завершение большого числа потоков;**
- **переключение контекста между ними;**
- **увеличение количества обращений к общей памяти (shared data);**
- **снижение кэш-локальности (данные не помещаются в кэш-памяти).**

2- При очень большом числе потоков (128–1024)

Время выполнения **значительно увеличивается**, а ускорение падает ниже 1. Это связано с тем, что:

- количество потоков **многократно превышает** число физических и логических ядер процессора;
- возникает **конкуренция за процессорное время** — ОС вынуждена постоянно переключаться между потоками;
- накладные расходы на планирование потоков становятся выше, чем полезная работа.

Таким образом, при слишком большом числе потоков программа работает **медленнее, чем последовательная версия**.

Вывод

Наиболее высокая производительность достигается при числе потоков, равном количеству логических ядер процессора.

Количество потоков, значительно превышающее число логических ядер, снижается производительность из-за накладных расходов на управление потоками и синхронизацию.