Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Операционные системы" №2

Студент: Бурунов М.А.

Преподаватель: Миронов Е.С.

Группа: М8О-206Б-22

Дата: 01.03.2024

Москва 2024 г.

Оглавление

Цель работы

Приобретение практических навыков в:

- Управлении потоками в ОС
- Обеспечении синхронизации между потоками

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. 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 19:

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

Общие сведения о программе

Программа представлена файлом – main.c

В программе используются следующие системные вызовы:

`pthread_create()`– создаёт новый поток для выполнения функции

`pthread_join()` – ожидает завершения работы потока

Общий алгоритм решения

С помощью потоков будем обновлять центры кластеров. Каждый поток получает один или несколько потоков в зависимости от количества кластеров, а также от количества потоков. Центры обновляются методом k-means.

Реализация

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <stdbool.h>
#include <time.h>

const int Dimension = 2;
const int X = 0;
const int Y = 1;
const double Inf = 2e9;

double** cluster_centers;
int** cluster_points;

int** coordinats_of_points;

int* number_of_points_in_claster;

int clasters_number;
int points_number;

typedef struct arguments
{
   int first_cluster;
   int last_cluster;
} arguments;
```

```
int** matrix_initializer(int first_size, int second_size)
{
   int** matrix = (int**) malloc(first_size * sizeof(int*));

   for (int index = 0; index < first_size; ++index)
   {
      matrix[index] = (int*) calloc(second_size, sizeof(int));
   }

   return matrix;
}

double** double_matrix_initializer(int first_size, int second_size)
{
   double** matrix = (double**) malloc(first_size * sizeof(double*));

   for (int index = 0; index < first_size; ++index)
   {
      matrix[index] = (double*) calloc(second_size, sizeof(double));
   }

   return matrix;
}

double** matrix_copying(double** input_matrix, int first_size, int second_size)
{
   double** matrix_copy;
   matrix_copy = double_matrix_initializer(first_size, second_size);
```

```
    for (int line = 0; line < first_size; ++line)
    {
        for (int column = 0; column < second_size; ++column)
        {
            matrix_copy[line][column] = input_matrix[line][column];
        }
    }

    return matrix_copy;
}

bool matrix_equality(double** first_matrix, double** second_matrix, int first_size, int
second_size)
{
    for (int line = 0; line < first_size; ++line)
    {
        for (int column = 0; column < second_size; ++column)
        {
            if (first_matrix[line][column] != second_matrix[line][column])
            {
                return false;
            }
        }
    }

    return true;
}

void delete_int_matrix(int** matrix, int size)
```

```
{
    for (int index = 0; index < size; ++index)
    {
        free(matrix[index]);
    }

    free(matrix);
    matrix = NULL;
}

void delete_double_matrix(double** matrix, int size)
{
    for (int index = 0; index < size; ++index)
    {
        free(matrix[index]);
    }

    free(matrix);
    matrix = NULL;
}


void* thread_update_cluster_centers(void* argument)
{
    arguments* thread_argument = (arguments*) argument;
    int start_cluster = thread_argument -> first_cluster;
    int last_cluster = thread_argument -> last_cluster;

    for (int claster = start_cluster; claster <= last_cluster; ++claster)
```

```
    {
        for (int coordinat = 0; coordinat < Dimension; ++coordinat)
        {
            double new_center = 0;
            int number_of_points = number_of_points_in_claster[claster];

            for (int point = 0; point < number_of_points; ++point)
            {
                int current_point = cluster_points[claster][point];
                new_center += coordinats_of_points[current_point][coordinat];
            }

            if (number_of_points != 0)
            {
                new_center = new_center / number_of_points;
            }

            cluster_centers[claster][coordinat] = new_center;
        }
    }

    return NULL;
}

void points_distribution()
{
    for (int cluster = 0; cluster < clasters_number; ++cluster)
    {
        number_of_points_in_claster[cluster] = 0;
    }
```

```
    for (int point = 0; point < points_number; ++point)
    {
        double minimal_distance = inf;
        int temporary_claster = -1;

        for (int claster = 0; claster < clasters_number; ++claster)
        {
            double distance = 0; //Расстояние от точки до центра кластера

            distance += pow((coordinats_of_points[point][X] - cluster_centers[claster]
[X]), 2);
            distance += pow((coordinats_of_points[point][Y] - cluster_centers[claster]
[Y]), 2);

            distance = pow(distance, 0.5);

            if (distance < minimal_distance)
            {
                minimal_distance = distance;
                temporary_claster = claster;
            }
        }

        int number_of_points = number_of_points_in_claster[temporary_claster];
        cluster_points[temporary_claster][number_of_points] = point;
        ++number_of_points_in_claster[temporary_claster];
    }
}
```

```
void update_claster_centers(pthread_t* threads, int clusters_for_thread, int clusters_for_last_thread, int threads_number)
{
    arguments thread_argument;

    for (int thread = 0; thread < threads_number - 1; ++thread)
    {
        int first_cluster = thread * clusters_for_thread;
        int last_cluster = first_cluster + clusters_for_thread - 1;

        thread_argument.first_cluster = first_cluster;
        thread_argument.last_cluster = last_cluster;

        if (pthread_create(&threads[thread], NULL, thread_update_cluster_centers, &thread_argument) != 0)
        {
            printf("Can't create thread\n");
            exit(-1);
        }
    }

    thread_argument.first_cluster = clasters_number - clusters_for_last_thread;
    thread_argument.last_cluster = clasters_number - 1;
    pthread_create(&threads[threads_number - 1], NULL, thread_update_cluster_centers, &thread_argument);

    for (int thread = 0; thread < threads_number; ++thread)
    {
        if (pthread_join(threads[thread], NULL) != 0)
        {
            printf("Join error\n");
```

```
            exit(-1);
        }
    }

}


void clasterzation(int threads_number)
{

    for (int claster = 0; claster < clasters_number; ++claster)
    {
        cluster_centers[claster][X] = coordinats_of_points[claster][X];
        cluster_centers[claster][Y] = coordinats_of_points[claster][Y];
    }

    pthread_t* threads = (pthread_t*) calloc(threads_number, sizeof(pthread_t));

    if (threads == NULL)
    {
        printf("Can't allocate memory\n");
        exit(-1);
    }

    if (threads_number > clasters_number)
    {
        threads_number = clasters_number;
    }

    int clusters_for_thread = clasters_number / threads_number;
```

```
    int clusters_for_last_thread = 0;
    clusters_for_last_thread = clasters_number - clusters_for_thread * (threads_number - 1);

    points_distribution();

    double** previous_centers = matrix_copying(cluster_centers, clasters_number, Dimension);

    while(1)
    {
        update_claster_centers(threads, clusters_for_thread, clusters_for_last_thread, threads_number);
        points_distribution();

        if (matrix_equality(cluster_centers, previous_centers, clasters_number, Dimension))
        {
            break;
        }

        previous_centers = matrix_copying(cluster_centers, clasters_number, Dimension);
    }
}

void print_result()
{
    for (int cluster = 0; cluster < clasters_number; ++cluster)
    {
        int points_number = number_of_points_in_claster[cluster];
```

```
        printf("%d cluster:\npoints:\n", cluster + 1);

        for (int point = 0; point < points_number; ++point)
        {
            int current_point = cluster_points[cluster][point];

            printf("(%d, %d);\n", coordinats_of_points[current_point][X],
coordinats_of_points[current_point][Y]);
        }

        printf("---------------------------------\n");
    }
}

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("You have to enter Number of threads\n");
        printf("Example: ./a.out 2\n");
        exit(1);
    }

    int threads_number = atoi(argv[1]);

    printf("Enter the number of clasters\n");
    scanf("%d", &clasters_number);
    printf("Enter the number of points\n");
    scanf("%d", &points_number);
```

```
coordinats_of_points = matrix_initializer(points_number, Dimension);

for (int point_index = 0; point_index < points_number; ++point_index)
{

   int coordinat;

   scanf("%d", &coordinat);
   coordinats_of_points[point_index][X] = coordinat;

   scanf("%d", &coordinat);
   coordinats_of_points[point_index][Y] = coordinat;
}

number_of_points_in_claster = (int*) calloc(clasters_number, sizeof(int));

cluster_centers = double_matrix_initializer(clasters_number, Dimension);
cluster_points = matrix_initializer(clasters_number, points_number);

long double start, end;
start = clock();

clasterzation(threads_number);

end = clock();

//printf("Execution time %Lf ms\n", (end - start) / 1000.0);

print_result();
```

```
    free(number_of_points_in_claster);
    delete_double_matrix(cluster_centers, clasters_number);
    delete_int_matrix(cluster_points,clasters_number);
    delete_int_matrix(coordinats_of_points, points_number);

    return 0;
}
```

Пример работы

Test 1

Input	Output
2 3 1 2 7 9 3 5	1 cluster: (1, 2) (3, 5) 2 cluster: (7, 9)

Test 2

Input	Output
3 3 1 2 7 9 3 5	1 cluster: (1, 2) 2 cluster: (3, 5) 3 cluster: (7, 9)

Определение ускорения и эффективности

В тестовом наборе 8 кластеров.

Количество потоков K:

K	Время исполнения	Ускорение	Эффективность
1	8.021 ms	1.0	1.0
2	6.72 ms	1.193	0.59
3	6.879 ms	1.166	0.422
4	5.12 ms	1.568	0.391
5	7.01 ms	1.14	0.228
6	7.88 ms	1.017	0.1695

После 4 потоков ускорение начинает уменьшаться. Это связано с тем, что я выполнял лабораторную работу на виртуальной машине с 4 потоками. Поэтому максимальное ускорение достигается при таком количестве потоков. Также при 3 потоках ускорение несколько уменьшается по сравнению с 2 потоками. Это связано с особенностями распределения кластеров по потокам в моей программе. При 3 потоках, 2 из них получают по 2 кластера, а последний оставшиеся 4.

Вывод

Проделав лабораторную работу, я приобрёл практические навыки в управлении потоками в операционной системе. Также мне удалось синхронизировать их работу путём разбиения на подзадачи для каждого потока, данные в которых не пересекаются.