**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

Оглавление

[Цель работы 3](#__RefHeading__738_2004454639)

[Постановка задачи 3](#__RefHeading__740_2004454639)

[Общие сведения о программе](#__RefHeading__742_2004454639) 3

[Общий алгоритм решения](#__RefHeading__744_2004454639) 3

[Реализация](#__RefHeading__746_2004454639) 4

[Пример работы](#__RefHeading__748_2004454639) 15

[Вывод](#__RefHeading__750_2004454639) 16

# **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 К:

|  |  |  |  |
| --- | --- | --- | --- |
| **К** | **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.