Applying optimization techniques to Lloyd's K-means algorithm in C

João Silva
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50495@alunos.uminho.pt

João Torres
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50499@alunos.uminho.pt

***Abstract*—The present assignment experimentally applied the different optimization techniques taught in the Parallel Computing class at University of Minho. Several compiler flags were used, aswell as in-code techniques such as inlining and avoiding complex mathematical operations in order to achieve better performance.**

***Index Terms*—C, K-means, Lloyd, Optimization**

I. Introduction

We were proposed to develop, using the C programming language, an optimized version of the Lloyd's K-means algorithm.

Therefore, the main objective of this work is to develop a function that randomly creates samples for the algorithm testing. We had the freedom to make changes to the original algorithm in order to optimize and improve the total run time. Nonetheless, we needed to maintain the original algorithm.

Throughout this report we will deeply explain everything we created: the structures, the initialization, the k-means implementation and all the optimizations that we made.

Last but not least, in the final section of this report we will make a critical analysis of the entire projection and development of the project.

II. How the algorithm works

The first step is initializing the data, which consists of N randomized points stored in an array (N is defined by the programmer - in this case, $N = 10.000.000$).
The data can then be divided into K clusters (K also being defined by the programmer - in this case, $K = 4$). After the data is initialized, the algorithm can be done in 3 steps:

1) Calculating the centroid of each cluster;
2) Assign each point to the closest cluster (using Euclidian distance);
3) Repeat steps 1) and 2) until no points are being moved from one cluster to another.

III. Implementation

A. Structures

For the implementation in the C programming language, structures (structs) were used to represent each point and cluster.

The point is a struct made of 2 floats (x and y) and an int - $cluster$ - that indicates the cluster that the point is associated to. The cluster struct contains an integer that keeps track of its size, a point that represents the cluster's centroid, and another point that acts as a temporary centroid, which will be useful later on.

B. Initialization

There are two global variables - points, clusters - that represent an array of points and an array of clusters, respectively. These are just pointers that will point to both of the arrays once the initialization is completed.
The initialization can be divided into three steps:

1) Allocate space for the points and clusters arrays using *malloc()*;
2) Generate N random points and place them in the points array;
3) Define a centroid for each one of the clusters.

In the initialization stage, the centroid of each cluster will simply be one of the first K points that were previously created.

C. K-means

The implementation of the Lloyd's K-means algorithm is the last stage of the implementation. It's also where most of the optimization techniques were applied, which will be discussed later.

The entire code is contained inside a while loop that keeps track of a *ready* variable. When $ready = K$, the loop will end, meaning that the algorithm has ended (the *ready* variable represents the number of clusters that have already reached an optimal solution).

Much like the initialization, the K-means can also be divided into 3 major steps:

1) Resetting the clusters: In each iteration, the cluster's centroid will change, which means that the points associated to that cluster will also change. For this reason, the cluster's $size$ and $newCentroid$ variables will be reset to 0, since we'll be associating new points to the cluster, taking into account the new centroid that was defined in the last iteration.

2) Assigning points to each cluster: This is the step where there's more room for optimization. For each point, the distance between the clusters and the point will need to be calculated and compared to find out what's the closest cluster to said point. After this comparison is done, the point is assigned to the cluster that's closest to it, while updating the cluster's $size$ variable and summing the value of the point to the *newCentroid* variable. This variable will later be divided by the total size of the cluster, so that we can find it's new centroid (this is why the value of *newCentroid* is always set to (0,0) in the first step, a sum always starts by 0).

3) Comparing centroids: In this last step, the value of the current centroid is compared to the value of the newly calculated centroid ($newCentroid$). In case both values coincide, the $ready$ variable is incremented to register that one of the clusters is ready. Once all clusters are ready, the while loop will end, and the algorithm is finished.

IV. Optimization

The optimization stage helped to significantly reduce the execution time of the algorithm. The techniques that were applied were:

- **In-lining**: in-lining small functions such as the distance function helped to remove function calls, making the algorithm faster. To improve code readability, the $static\ inline$ declaration of the $distance()$ function was used instead of writing its code directly inside the $k\_means()$ function.
- **Efficient iterations through arrays**: Iterating through arrays can be very time consuming, which is why it was avoided. For instance, when calculating the sum of the points of each cluster, instead of iterating through the array, a temporary variable is used. Whenever a point is added to the cluster, it's value is added to the temporary variable and the total sum is stored. This way, there's only a need to iterate through the points array twice (during initialization and during the 2^{nd} phase of the algorithm).
- **Avoiding complex mathematical operations**: The euclidian distance formula is $d(x, y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$. Square root is slow to calculate, and since we have no interest in knowing the actual value of the distance between points, assuming that $\sqrt{a} > \sqrt{b} \Rightarrow a > b$, we can discard it, avoiding the $sqrt()$ function from the $math.h$ library. We can also avoid the $math.h$ library altogether by not using the $pow()$ function, and simply in-lining the multiplication ($a^2 \equiv a \times a$).
- **Compiler flags**: Compiler flags were very important to improve the execution time. There's a very drastic difference in performance simply by using flags when compiling the program. The $-O1, -O2, -O3, -Ofast$ flags were tested to measure the performance difference between them. The flag that provided the best performance was the $-O2$ flag.

V. Results

There's a noticeable improvement in performance by not using the $math.h$ library. This is mainly because $sqrt()$ (which is a slow operation) gets discarded. Also, by replacing the $pow()$ function for the inline equivalent, several function calls are avoided, which positively contributes to the performance of the algorithm.

TABLE I
Comparing execution times between using vs not using "math.h" functions

	With "math.h"	**Without "math.h"**
time (s)	11.54	4.30

$Measured\ using\ the\ -O2\ flag$

Simply by adding any one of the listed flags the execution time is dramatically reduced. The flags that offered better performance were the $-O1$ and $-O2$ flags, where the $-O2$ flag offers the best performance by a small margin.

TABLE II
Comparing execution times between different compiler flags

	Flags				
	None	***-O1***	***-O2***	***-O3***	***-Ofast***
time (s)	26.35	4.75	4.30	7.33	7.34

$Not\ using\ the\ math.h\ library$

VI. Conclusion

At this stage of the project we got to the conclusion that we were able to fulfill everything we were asked.

In terms of optimizations, we did a lot of them to reduce our run time, as explained previously in this report. With this being said, we used in-lining and efficient iterations through arrays. We also avoided complex mathematical operations and used suitable compiler flags.

Besides, we consider that this work was very interesting and challenging. It made us practise a lot about the things we have been taught during the first weeks of this semester.

As a group, we managed to work together and mutually support each other. In general, we had a positive performance.

In conclusion, this project helped us develop new skills and consolidate what we've been learning about optimization.