Parallel Computing
K-means Algorithm

Rui Monteiro
Informatics Department
University of Minho
Braga, Portugal
pg50739@alunos.uminho.pt

Diogo Barbosa
Informatics Department
University of Minho
Braga, Portugal
pg50326@alunos.uminho.pt

***Abstract*—Initially, having 10 million that we intend to organize into 4 clusters that are represented by their centroids, we use a simple k-means algorithm based on Lloyd's algorithm, in which we want to iterate progressively in order to calculate new centroids and reorganize the *clusters* in their function until it is verified that they have converged. In addition, we analyze runtime performance based on different metrics and using different optimizations.**

***Index Terms*—K-means, optimization, metrics, sequential, algorithm, performance, testes, complexity**

I. Introduction

In this project we aim to implement a sequential version of k-means, which organizes an array with **N** points as a function of the Euclidean distance to **K** centroids, dividing them into clusters. Firstly, we implemented simple and functional version of the algorithm, and then we tested several metrics, analyzing and comparing the expected results.

II. Sequential K-means

A. Implementation

The implementation of the k-means algorithm is divided into two phases: initialization and convergence.

In the first phase, two vectors are initialized, one with **N** elements, which will be the samples, and another with **K** positions, which will represent the calculated centroids. The initial values assigned to them are generated randomly through a pre-defined seed. The samples are then assigned the closest centroid based on the Euclidean distance.

In the second phase, the algorithm starts by calculating a new centroid (previously initialized randomly) corresponding to the geometric centre of the points belonging to this *centroid*. Then, with the new centroids, the samples are assigned to the closest cluster. This process is done until the convergence is complete, that is until the centroids are in their final position.

This pseudo-code can also translate the implementation described above:

```
v = allocate(N)
clusters = allocate(K)

v = getRandomNubers()
clusters = getRandomNubers()

while(notConverged):
    for i in v:
        i = assignPointToClosestCentroid()
        calculateNewCentroids()
    checkIfConverged()
```

B. Analysis of the Algorithm's complexity

This algorithm mostly depends on the time (number of iterations) that the initial centroids take to converge, as this number is impossible to predict since the samples are generated randomly, we will designate it as **R**.

Bearing in mind that the initialization is only done once, we arrive at the value of $\theta(2N + K)$. In the convergence part of the algorithm, the complexity is $\theta((2N + K) \times R)$, finally, the total complexity of the algorithm is counted, which will be $\theta(R \times (2N + K))$. This means that the algorithm grows linearly with **N**, **K** and **R**.

III. Analysis of possible optimizations

A. Flag based optimizations

There are some optimizations that can be applied to this algorithm, in order to reduce the number of instructions and execution time, such as loop unrolling and vectorization. Applying flags, through the gcc compiler, in this case, version 7.2.0, as well as code optimizations that produce faster execution times, in order to achieve the intended goal.

The optimizations performed were aimed at minimizing the execution time, as well as the number of instructions, in addition to the number of clock cycles and cycles per instruction, in order to obtain a simple but optimized and easily scalable algorithm. For this, we analyzed the impact of using flags from gcc as "-O0", "-01", "-O2" and "-O3", in addition

to combining their use with "-funroll-loops" so that we apply loop unrolling to the code.

B. *Relevant design choices*

At first, we implemented 2 different optimizations in the original code of the algorithm itself, while initially the "sqrt" function from the math.h header was in our program, it was removed since it is an unnecessary cost in the program's execution time, including the number of cycles and instructions performed and because it was only used to compare distances, not using it still produces a correct result. After that initial code optimization, the "pow" function from the same library was substituted by multiplying the given math expressions by themselves once and therefore achieving the same result while reducing execution time and optimizing the program even further.

After this, the cluster of each sample was stored alongside the x and y coordinates in a struct that was used several times throughout the algorithm to get those values and to change the cluster to which a given sample belonged. However, this was more costly than simply not storing that value since it was not necessary after making some minor design changes to the way new centroids were calculated for the clusters, therefore the program only stores two float arrays, one with each coordinate.

A trio of new variables was introduced in order to calculate the new centroids in a more efficient way, saving instructions and clock cycles by adding the x and y values, as well as the number of samples in a cluster, while calculating the new cluster that each sample belongs to and after just using a small loop with K iterations to calculate the centroids themselves.

C. *Performance tests*

In order to obtain the results presented in the table below, the code was compiled on the s7edu2 machine of the SeARCH computational cluster and the execution was performed on a computation node of the "-cpar" partition.

TABLE I
OPTIMIZATION WITH FLAGS

Optimization	# of Instructions	Clock cycles	Execution time (s)
-O0	132,631,501,850	59,680,809,528	19,2349636987
-O1	28,714,236,405	12,434,187,471	4,813095568
-O2	28,313,421,974	12,300,460,755	3,757124834
-O3	20,616,161,221	28,238,969,108	8,865313508

The most significant performance gain in the tests performed was in the use of "-O1" flag in relation to "-O0", with this flag the assembly code is optimized and stands out for its more efficient use of stack. As for the "-O2" flag, the optimizations to the assembly produced are deeper, instructions vectorization and loop unrolling are performed. Finally, we tried the "-O3" flag which implements more agressive optimizations to the code produced, however in our tests it performed worse than the "-O2" flag. This was most likely caused by the structure of our implementation of k_means which did not allow for the auto vectorization provided by the "-O3" flag.

It is then concluded that the "-O2" flag produces the best result in terms of performance, this is due to the optimizations carried out by the compiler, which include better use of stack, reduction of memory accesses and reduction of the instruction number.

In order to test the code optimizations that were implemented, the "-O2" flag was used as it was the one that produced the best possible results, given this, both the decision to remove the "sqrt" function in the calculation of the distance between the sample points and the centroids and the removal of the "pow" function in the same calculation were beneficial to the performance of the implemented algorithm. Other beneficial changes were the removal of the struct that stored each sample and the changes in the way new cenntroids were calculated which produced lower cache misses values and better overall performance statistics,

IV. CONCLUSION

After completing this project, we consider that the tests applied to the sequential algorithm of k-means are sufficient to correctly analyze its performance, although there is possibly the possibility of optimizing it to a higher degree.

Furthermore, in this report we presented a detailed explanation of the implementation of the k-means algorithm based on Lloyd's algorithm and the performance tests performed, taking into account the different optimizations, both in the code itself as well as in the compilation flags used, producing a very well refined result.