Parallel Computation K-Means Algorithm

José Diogo Vieira
Informatics Department
University of Minho
Braga,Portugal
pg50518

António Luís Fernandes
Informatics Department
University of Minho
Braga, Portugal
pg50229

***Abstract*—The K-means algorithm is known as a clustering method used in data mining. Given an initial vector with random values on it (coordinates x and y), it will firstly initiate the K centroids using the first K values of the sample. After that, it will associate the remaining values to the most near cluster using the Euclidean distance. Thirdly, we calculate the centroid of all clusters and then repeat the whole process, which is, assigning all the values to the nearest cluster until there are no values changing in between clusters. This algorithm is a useful method when dealing with loads of data samples and is the most know method to do so. In this report, we will explain the work we have done and also explore some of the optimizations methods we applied into our code in order to lower its execution time.**

***Index Terms*—k-means, algorithm, optimization, performance**

I. Introduction

In this paper, we will firstly describe the data structures needed to implement the K-means algorithm and what were the strategies we used to implement it. Secondly, we will explain some of the optimizations we used to make our code more effective and optimized.

II. K-Means Algorithm

A. Data Structures

The data structures necessary for the algorithm were two arrays of floats:

- **vector**, with size N*2, to store all point's values
- **centroid**, with size K*2, to store the values of each cluster's centroid

And two arrays of integers:

- **cluster**, with size N, to store information about what cluster each point was assigned
- **size** with size K, to store the size of each cluster.

B. Implementation

The first step is to generate the N random samples, and initialize all K clusters with the first K samples. For this, we generated two random values (x,y) between 0 and 1 for each point. Then we iterated for the first K samples, and assigned each one of them to a centroid, we need the value of each centroid to be completed so we can proceed to the next step.

This next step is the main one and the core of the whole algorithm. For each iteration of the algorithm, we will iterate over the N samples, and for each sample calculate the euclidean distance between the point and each centroid and assigned to the respectively cluster.

After finding which cluster has the lowest distance with the point, we will update the cluster array with the new cluster index if necessary, is at this point that we keep track if any point changed cluster. For each iteration, we will have a array of floats with size K*2, with the goal of sum all values that are being added to each cluster, this prevents a new cycle to sum all cluster values after the assignment is done. We will also have an array that keeps track of the size of each cluster.

Below is the pseudocode of our version of the k-means algorithm:

```
while(has_changed)
  has_changed = false
  for i = 0 to N
    lowest = distance(vector[i],centroid[0])
    index_low = 0;
    for k = 1 to K
      dist = distance(vector[i],centroid[k])
      if (dist < lowest)
        lowest = dist
        index_low = k
    if (index_low != cluster[i])
        cluster[i]=index_low;
        has_changed=true
    sum[index_low] += vector[i]
    tempSize[index_low]++
    # centroid calculation
  for i = 0 to K
    size[i] = tempSize[i]
    centroid[i] = (sum[i]/size)
```

C. Optimizations

For the optimizations, we begin to try to improve our code. Firstly, we change the way we were doing the sum of the cluster values, instead of doing it after all point were assigned to the cluster, we do it at the same time the points are being assigned. This decreases by a significant number (almost half)

ours L1 cache misses. We realized that one thing that was bad for our program was the function **distance** and the amount of calculations using pow and sqrt of math.h. Since we don't actually need the values of the distance, there's no need to use sqrt, we just need to check which one is the lowest. So we take off the sqrt and replace the pow(x,2) with x*x. This had a huge decrease in execution time, as well as number of instructions. Then we try different optimization flags and analyze how much impact they had in the execution time, number of instructions and cycles. The results can be found on the table below.

Flag	Execution time(s)	#I (10^9)	Cycles (10^9)
-O0	25	100	75
-O1	6.2	48	20
-O2	4.5	30	14
-O3	8	25	26

These tests were performed at the Search cluster in the partition "cpar". The results are an average of 5 iterations performed with the command "perf stat -e instructions,cycles,L1-dcache-load-misses -r 5 " for each flag. We can see an increase in performance as we change the flag. From -O0 to -O2 we see an expected decrease in every parameter, but with the -O3 flag the expected performance wasn't met. We can see a decrease in the number of instructions but both the execution time and the cycles had a significant increase. These can be explained by the more complex instructions that were generated by this flag. Given these results, the flag we chose to use was -O2 since was the one who performed the best across all parameters.

D. Vectorisation

In order to improve our program, we tried to implement vectorisation. For that we mad a few changes in our algorithm. We revert the decision on keeping the sum of the cluster values inside the main for loop, and instead we have a dedicated for loop to sum of the clusters and to increase its size. These changes are made to try to lower the work of the main loop, and take advantage of vectorisation. We add the flags -ftree-vectorize -mavx with the previously added -O2 and we aligned the vector array at 32-byte memory addresses. We test this new implementation and the results were:

Execution time(s)	#I (10^9)	Cycles (10^9)
3.98	28.7	12.7

As we can see, it was an all around boost on the performance of our algorithm. Even though, there was more vectorization that could be done, we couldn´t get it to work properly, so we decided that this was our final version.

III. Conclusion

Given the main goal of this first project ,the optimization done into our algorithm, we present a critical and thoughtful view of the work developed.

We considered that the the most difficult part was the vectorisation, we still have some doubts about if we took the right approach. We know that we could improve our code to handle vectorisation in a better way. But our tries, did not succeed. Despite of that, we could still do vectorisation on a smaller part, that still improved our past work.

In summary, we consider that all the goals were achieved and the work done was complete. In addition, with this project we were able to amplify the knowledge obtained during the classes.