Parallel Computing
Parallel K-Means Algorithm

1st Rodrigo Rodrigues
Informatics Department
University of Minho
Braga, Portugal
pg50726@alunos.uminho.pt

2nd Daniel Azevedo
Informatics Department
University of Minho
Braga, Portugal
pg50311@alunos.uminho.pt

***Abstract*—*K-means* clustering is a method of vector quantization that aims to partition N observations into K non-overlapping subgroups (clusters) in which each observation belongs to the cluster with the nearest mean (cluster centroid), serving as a prototype of the cluster. In an effort to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible, the *K-Means* algorithm computes centroids and repeats until the optimal centroid is found. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster. In this paper we will use the code developed in the previous developed sequential version of the algorithm and, through the primitive OpenMP, implement a version of the program that in one or more phases of the process divides processing by multiple running threads.**

***Index Terms*—K-means, Algorithm, OpenMP, Parallel, Threads**

I. Introduction

This phase of practical work aims to assess the development of programs that explore parallelism with the main objective of reducing program execution time. Using C, a parallel version of the *K-Means* algorithm was developed. We started off by looking at the optimized sequential version developed in the first phase of this project and identifying which code blocks have the highest computational load.

Then through OpenMP primitives, we implemented a version of the program that divides the processing by multiple running threads in different phases of the algorithm.

Furthermore, we performed tests using different versions of the algorithm each time with better efficiency due to use of different parallelism strategies. Lastly, the obtained results were analysed and compared with the expect results.

II. Implementation with OpenMP

As we mentioned earlier, to create the parallel version of the *K-Means* algorithm, we added *OpenMP* directives to the sequential version of algorithm. We analysed the sequential implementation to verify which blocks of code could be parallelized, in order to obtain better performance in terms of execution time and thus distributing efficiently the computational load for the various threads.

For best results, we try to use the OpenMP directives in the loop cycles where there is the highest computational load, these are the places where the most significant improvements are expected.

The K-Means algorithm can be broken down into different steps, the first step is initialize the points with random values, this task was some computational load, so this part is a candidate to be paralleled. For this step we use the random function, this function is not thread-safe, since it uses hidden state that is modified on on each call. Thus we not have advantage become this code paralleled.

The second step of the algorithm, and also the one that requires more processing power, consists of calculating the centroids and assigning the point to the respective cluster. For this task we use one for cycle that in each interaction calculates the distance to all centroid and attribute assigns cluster. the following omp primitive:

```
#pragma omp parallel for num_threads(N_THREADS)
private(min_dist, min_indice, dist)
reduction(+:sum_x, sum_y, count)
```

With this primitive, namely with the use of clause private, we defined the variables min_dist, min_indice and dist as private, it make each threads have one instance of each of the variables, not sharing with other running threads.

In turn, when we use the clause reduction we define a set of variables that will have a reduction operation, in this case a sum, at the end of the parallel code zone.

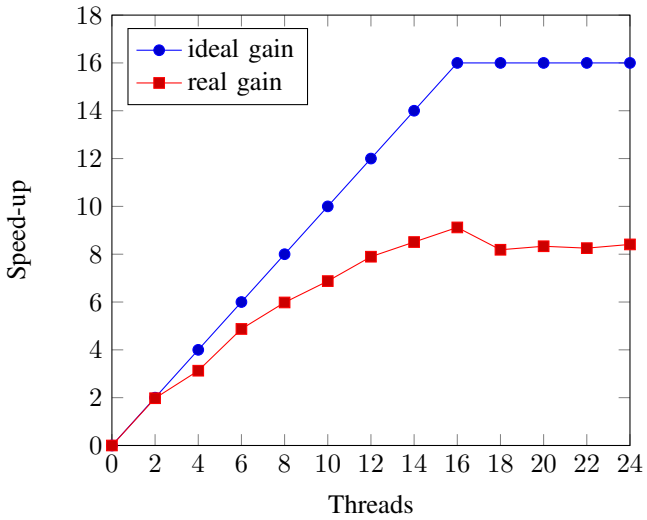
Thus, using this primitive and the two clauses, we managed to turn the part of code that requires more computational resources into a parallel code, there is a clear decrease in execution time.

In some for cycles, that traverse the structure of clusters, we also use the pragma omp parallel for primitive, since these cycles have few iterations, that is, they do not have a large computational load, there are no significant improvements in execution time.

III. Scalability

It was created a graphic that allows us to understand the scalability of the parallel k-means algorithm implemented.

To measure the scalability of the algorithm, the measure speed-up was used. This measure is calculated through the quotient between the execution time of the sequential version and the parallel version.



All in all, it was possible to verify that the parallel version of K-Means has an overall better performance and that it is also a scalable solution. We can verify that the execution time of the algorithm tends to decrease when the number of threads increases, although there are no improvements with more than 16 threads. That said, the speedup scales well.

IV. Results

In this chapter we will make a comparison between the sequential algorithm and its parallel version. In the parallel version, 16 threads will be used, since, as seen previously, it turned out to be the best in terms of execution time.

Number of clusters	Sequential version (s)	Parallel version (s)
4	4,388344318	0,480309411
8	67,610146558	0,738706971
24	173,786834030	1,751194582
32	222,760041100	2,217846568

By analyzing the previous table, we can verify that the greater the number of clusters, the greater the execution time of the algorithm, regardless of the version used, it is also possible to verify that with the increase in the number of clusters there is a greather gap in terms of execution time between the two versions and an even greater benefit of using the parallel version instead of the sequential one.

V. Conclusion and future work

With this practical assignment the team was able to better consolidate the knowledge acquired during theoretical and practical classes regarding the impact of using parallel programming in a shared memory environment using C and *OpenMP*. It is important to emphasize that some improvements or updates could be applied in the future, changing the K-Means algorithm to make it even more parallelizable or even executing more tests using different metrics. Furthermore, using *OpenMPi* and comparing the performance gains with the ones obtained with *OpenMP* would also be an interesting approach.