Parallel Computing
Parallel 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*—Given a K-means sequential algorithm previously developed, which distributes N samples iteratively, using the lowest distance as the deciding factor, between a certain number of clusters until said clusters are verified to have converged, we will improve the already quick execution time by parallelizing it. To convert this algorithm into a parallel version, the OpenMP API will be used, utilizing the most viable alternative keeping scalability in mind. On the other hand, to choose the most optimal number of threads used there will be several performance tests considering different metrics.**

***Index Terms*—K-means, OpenMP, metrics, parallel, algorithm, performance, threads, dependencies, critical zone**

I. Introduction

The objective of this project is to improve a sequential K-means clustering algorithm and evaluate the benefits of parallel programming by using OpenMP, dealing with data dependencies, race conditions and critical zones.

In addition, several tests were performed to analyze which number of threads would have the best performance considering our use case and to get a greater understanding of the overall benefit of parallelizing the initial sequential algorithm.

II. Parallel K-Means

A. Implementation with OpenMP

To create the parallel version of our sequential K-means algorithm, we added OpenMP directives to the code we already had. After analyzing the original version of our K-means algorithm, we determined what sections of the code could be parallelized and decided where it would benefit the program's overall performance in terms of execution time.

The main focus to parallelize this algorithm was in the most extensive loops that run **N** times (10 000 000 in our tests) since they were the ones that require the most computational power to run. There are only 2 such loops: in the initialization section and the point assignment section. Since the rand() function is not thread-safe we had to leave it running sequentially, however, the point assignment section did not have any dependencies that prevented it from being parallelized, although it had to write information to shared memory. The latter section, which happens to be the most demanding section of our code is composed of 2 parts, the point assignment to its closest centroid, and the corresponding centroid update.

The *point_assignment()* has a loop cycle with read-after-write dependencies which we were unable to remove, so it was not parallelized itself, the rest of the loop cycle had no such dependencies so it was parallelized using a *reduction(+:)* to avoid complications with shared memory.

The pseudo-code below shows how it was implemented:

```
#pragma omp parallel
#pragma omp for reduction(+:xCen,yCen,samples)

for(int i = 0; i < N; i++):

    //contains loop with RaW dependencies
    minIndex = point_assigment();

    samples[minIndex] += 1;
    xCen[minIndex] += x[i];
    yCen[minIndex] += y[i];
```

Another aspect of our implementation that needs to be mentioned is that we had several more opportunities to create parallelism in our code, which we chose not to. This decision was taken, despite unresolved memory-sharing conflicts, the overhead created by the use of threads on those code sections outdid any benefits that the parallel execution could provide. This was evident in the loops that were executed **K** times, which is 4 or 32 in our testing, which was too little for the use of threads to be viable.

III. Execution profile measurement

The results presented in table I were obtained by compiling our code on the **s7edu2** machine of the SeARCH computational cluster and the execution was performed in the "cpar" node, equaling the number of CPUs per task to the number of threads, except in the sequential version.

Additionally, the results regarding the second and third columns represent the best execution time obtained, are rounded to 4 decimal places and are presented in seconds.

TABLE I
PERFORMANCE VARYING THE NUMBER OF THREADS AND CLUSTERS.

# of Threads	4 Clusters	32 Clusters
Sequential	2.4543	13.9678
2	2.1227	9.3733
4	1.2101	7.4138
8	0.8295	4.7510
12	0.6459	2.9183
16	0.4708	2.1250
20	0.4364	1.8772
24	0.4989	1.8752
28	0.5072	1,4254
32	0.5215	1.3705
36	0,4101	1,3677
40	0,4614	1,1841

IV. SCALABILITY ANALYSIS

In order to choose the number of threads that our program should use in its parallel version, we made an analysis based on speed up, which is obtained by dividing the time of the best sequential implementation by the time of the parallel version. Those results are presented in figure 1. These results could have been more representative of the true SpeedUp if an average value was used for the execution time of the parallel version instead of just using the best available.

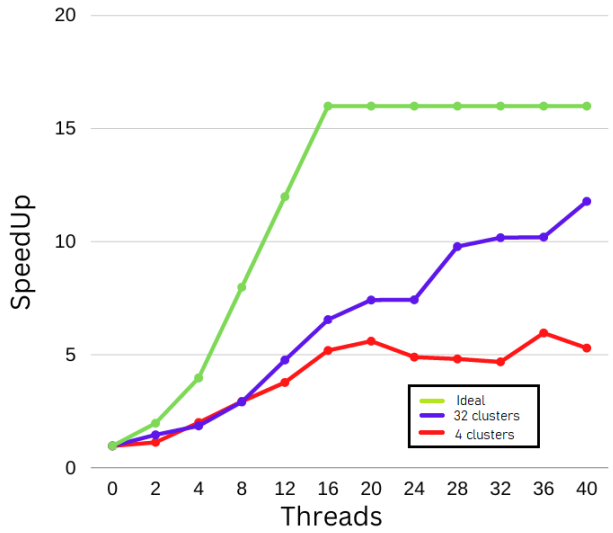

Fig. 1. Graphic representation of the SpeedUp based on the number of threads used.

In the case of 4 clusters, the SpeedUp isn't as evident because the parallelized section of the algorithm contains a loop in **K** (number of clusters), which is relatively insignificant when testing with such a small number of clusters.

When using 32 clusters we can see that the gain in performance is greater overall, which indicates good scalability when increasing the number of clusters and combining that same increase with an increase in the number of threads, although it would be excepted for the gain in performance to drop slightly after it reached a plateau, which we couldn't replicate in this situation, for possibly needing to use more threads to see this phenomenon.

V. LOAD BALANCING

When the number of threads surpasses 16, it is visible that there is a balance between resource distribution and parallelism in the case of testing with 4 clusters. Meanwhile, when testing with 32 clusters, the gain in performance is gradual and continues as the number of threads and CPUs per task increases.

Considering this, the group settled with 16 as the number of threads in our *Makefile* in order to get acceptable performance, while minimizing the overall computational resources used, keeping in mind that the number of clusters can be altered and can possibly benefit from this decision.

VI. CONCLUSION

After completing this project, we concluded that although several sections of code can be parallelized using OpenMP directives, not always translates into a performance gain, due to the overhead introduced by said directives. For this reason, the choices of where and which OpenMP directives to use in the code were critical to the final results, since a poorly implemented parallel code could be worse than its sequential counterpart.

Moreover, the parallelization implemented in our K-means algorithm allows for better performance in terms of execution time, which was optimized by conducting several tests considering a variety of metrics that correctly assess the algorithm produced. This project could potentially profit from the use of other OpenMP directives and code vectorization of the main loop cycle.

To conclude, this report provides a detailed explanation of the decisions made to convert the original sequential algorithm to a parallel one, as well as the reasoning behind the choices made and the performance tests executed, and we consider that the main goal of the project was achieved.