

K-Means Algorithm Parallelization

Analysis and solutions for dividing an algorithm's workload with the objective of reducing its execution time.

1st Vicente Moreira
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50799@alunos.uminho.pt

2nd Joana Alves
Escola de Engenharia
Universidade do Minho
Braga, Portugal
pg50457@alunos.uminho.pt

Abstract—This document will present the evolution of the previous developed clustering algorithm, where many parallelization techniques will be analyzed, considered, implemented and evaluated with the aim to further reduce the algorithm execution time, by splitting and sharing the workload between computer resources.

Index Terms—K-Means, Parallelization, Algorithm, Threads

I. INTRODUCTION

A. Context

With the evolution of computer processors (CPU's), single core performance has improved significantly with better pipelining, branch prediction and higher clock frequencies.

Another improvement was increasing the number of cores on CPU's and the development of features like "hyper-threading". These latter improvements mean that computers are able to handle many tasks at once or divide a heavy task between many threads and cores.

B. The Problem and Strategy Adopted

In this project, our team will improve the clustering algorithm developed in the last project, by applying parallelization techniques on its code and evaluating its execution time.

We will start by measuring our single thread execution time and evaluating the most computationally heavy functions, along with analyzing the possible parallelization of these functions and how much performance gain is expected. We will also perform tests to see how the execution time is affected by the number of threads available.

II. TESTS AND METRICS TO BE EVALUATED

In this project, the only metric used to evaluate the algorithm performance will be its execution time, since it's the most indicative of how effective and well managed the CPU resources are being handled.

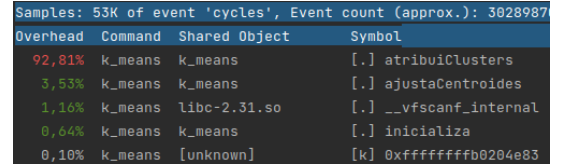
To test the parallelized algorithm, our team will execute a total of 60 test for each version developed. These include testing for a different number of clusters (4 and 32), and for each, we will test the algorithm with an increasing number of threads (from 2 up to 20¹, with increments of 2), and for each of these, we decided to execute the algorithm 3 times and

calculate its mean so as to remove any inconsistencies found on the results.

III. FIRST VERSION DEVELOPMENT

A. Computationally Heavy Functions

Our team started by analyzing the previous sequential version of this algorithm, in order to find what functions need to be parallelized. For this we used the command *perf record*:



The image shows a terminal window with the output of the 'perf record' command. At the top, it says 'Samples: 53K of event 'cycles', Event count (approx.): 3028987'. Below this is a table with four columns: 'Overhead', 'Command', 'Shared Object', and 'Symbol'. The data rows are as follows:

Overhead	Command	Shared Object	Symbol
92,81%	k_means	k_means	[.] atribuiClusters
3,53%	k_means	k_means	[.] ajustaCentroides
1,16%	k_means	libc-2.31.so	[.] __vfprintf_internal
0,64%	k_means	k_means	[.] inicializa
0,10%	k_means	[unknown]	[k] 0xffffffffb0204e83

We can quickly see that the function **atribuiClusters** is by far the heaviest function on this algorithm, as such, we will start by focusing on it.

B. Parallelization analysis

On this particular function, we iterate over all N points, and on each, we iterate over all K Clusters to calculate the distance of that point to all Clusters, and assign the closest Cluster to that particular point, by saving the Cluster value in an N sized array, where the index identifies the point.

With the intention to parallelize the main N for loop, we determined that there is almost no data shared between threads, since every thread will access/write the point data array at different indexes, being the only exception the centroids coordinates, which are only read to calculate the distances, so there won't be any collision problems.

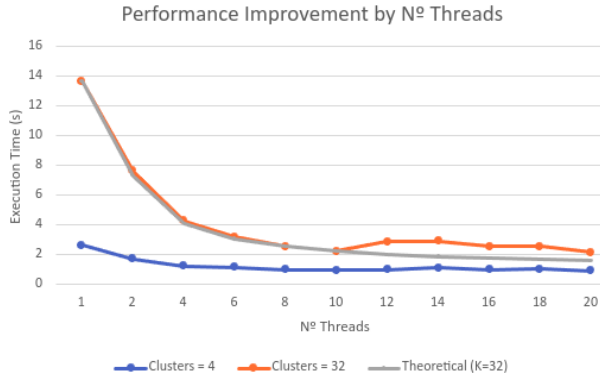
We will also use a static workload division strategy '**schedule(static)**', due to its simplicity, reduced overheads and the fact that, along this particular loop, the amount of computing is equally distributed.

C. Expected Performance Improvements

With the data from the figure above, we can extract the time spent in the execution of each function ($\text{Texec} * \text{FunctionPercentage} = \text{TimeInFunction}$). Therefore, we can see that the *atribuiClusters* function had an overall execution time of: $13.6s * 0.928 = 13.35s$. Thus, we expect a **time gain** of 13.35s divided by the number of threads used.

¹Maximum number of physical cores in the testing machine

D. Results Graph



E. Result Analysis

- **Cluster = 4:** We can see there is only a small performance improvement due to parallelization, maxing out at T=10. This happens since there's a reduced number of clusters, which doesn't allow for a efficient workload division.
- **Cluster = 32:** Our expected time gain matches with the results obtained², until it reaches its optimal point at around T=10, where there are no more improvements.

IV. SECOND VERSION DEVELOPMENT

A. Parallelization analysis

For the next version, our team started by looking for more computationally heavy loops that haven't been parallelized. In the function **ajustaCentroides** there are a total of 3 for loops, one that iterates over K to initialize the 3 arrays (2 for sums, 1 for counting), the next one iterates over N and it's the main loop, where for each point, we identify its cluster and then sum its x and y values to the correct index on the 2 sums array, while incrementing the number of points in that cluster. At last, we iterate over K again, where we now calculate the means of all coordinates and assign the new centroids.

Being that the for loop that iterates over the N points is the major workload of this function, we decided to apply parallelization techniques. Due to its logic of summing the x,y values of a point to a shared K array, we needed to include some control logic, to avoid thread collisions and data errors.

Since the collisions only happen when summing values to a shared array, we added the data control directive **omp reduction(+) , where each thread will sum up its own values and, at the end of the loop, add up to the final results.**

B. Expected Performance Gain

We analyzed the **perf record** of our V1 algorithm:

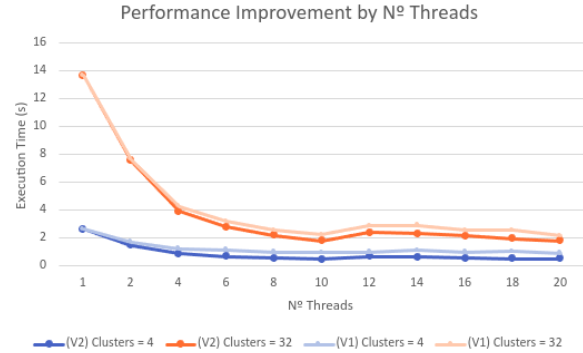
```
Samples: 62K of event 'cycles', Event count (approx.): 52458939169
```

Overhead	Command	Shared Object	Symbol
96,22%	k_means	k_means	[.] atribuiClusters_omp_fn.0
2,25%	k_means	k_means	[.] ajustaCentroides
0,60%	k_means	libc-2.31.so	[.] __vfscanf_internal
0,32%	k_means	k_means	[.] inicializa

²Check Attachments for Expected Gains Graph

As we can see in the image above, the *ajustaCentroides* function doesn't have a heavy impact on the algorithm. Thus, with the next tests, we aren't expecting a significant gain from applying parallelism techniques.

C. Results Graph



D. Result Analysis

The first thing to note is our output result:

```
N = 10000000, K = 4
Center: (0.250, 0.750) : Size: 2498768
Center: (0.250, 0.250) : Size: 2501628
Center: (0.750, 0.250) : Size: 2499387
Center: (0.750, 0.750) : Size: 2500217
Iterations: 20
```

This output differs slightly from the outputs in the V1 version, but only on the number of points in each cluster, being this value consistent across the same test conditions. With a different number of threads, the outputs changes slightly again.

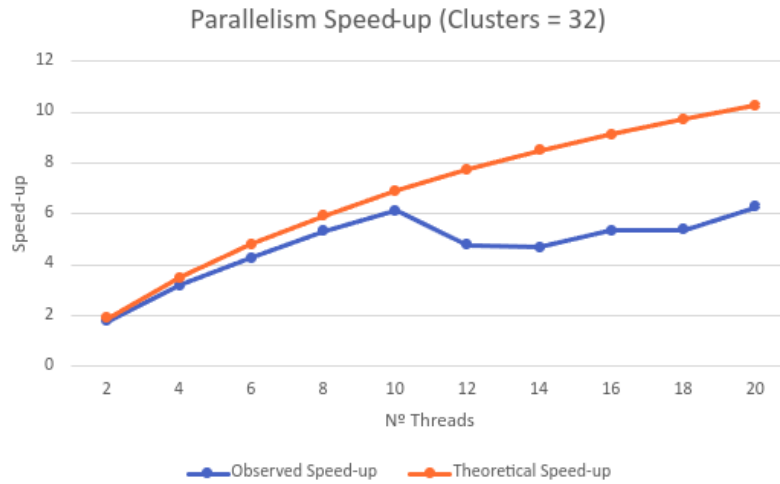
This happens due to the floating point arithmetic, where the order of the added values differs for each number of threads. Since floating values aren't infinitely precise, especially when dealing with larger values, the latter sums on the sequential version of this function are heavily rounded up, whereas in the parallelized version, each thread works with smaller values, reducing the rounding errors.

Regarding this version's performance, we observed a small reduction in execution time, as expected from our analysis.

V. CONCLUSION

Our team is very satisfied with the results achieved on this project, having improved the sequential clustering algorithm execution time by dividing its workload between computer resources, obtaining better execution times such as a reduction from 13,7s to 1,8s (for 32 Clusters), representing a maximum 7.6x improvement.

VI. ATTACHMENTS



Version	K	N°Threads	TExec (s)
V1	4	1	2,638
		2	1,724
		4	1,238
		6	1,150
		8	0,999
		10	0,963
		12	1,01
		14	1,106
		16	1,003
		18	1,028
		20	0,939
	32	1	13,697
		2	7,669
		4	4,285
		6	3,215
		8	2,574
		10	2,239
		12	2,876
		14	2,915
		16	2,556
		18	2,549
		20	2,178

Version	K	N°Threads	TExec (s)
V2	4	1	2,638
		2	1,495
		4	0,866
		6	0,702
		8	0,580
		10	0,499
		12	0,677
		14	0,626
		16	0,554
		18	0,534
		20	0,507
	32	1	13,697
		2	7,554
		4	3,927
		6	2,793
		8	2,192
		10	1,795
		12	2,418
		14	2,324
		16	2,167
		18	1,981
		20	1,797