Parallel Computing
K-Means Algorithm

Diogo Rebelo
Informatics Department
University of Minho
Braga, Portugal
pg50327@alunos.uminho.pt

Henrique Alvelos
Informatics Department
University of Minho
Braga, Portugal
pg50414@alunos.uminho.pt

***Abstract*—Parallelization of the k-means algorithm based on Lloyd's algorithm using *OpenMP* directives and exploring the different clauses and alternatives for our code. This project is focused on a design methodology development of parallel programs.**

***Index Terms*—K-Means, ML, OpenMP, Threads, Performance, Parallel, Optimization, Threads, Lloyd's, Clustering.**

I. Introduction

The main purpose of this assignment is to understand the advantages of algorithms' parallelization in C, through the development of a second version of K-Means, an optimized and parallelized algorithm .

II. Parallel K-Means

A. Approach

Firstly, we start by analyzing the set of functions with the highest cost in our sequential program, that is, those in which the PU spends the majority processing time. Thus, to do this, we use the *Perf* performance tool. The idea is to analyse this profile, observing the functions with higher computational overhead, in order to understand which parts of our code need/are able to be optimized. The main goal on this stage is to reduce the execution time, distributing efficiently the computational load for the various parallel threads.

At first glance, understanding the algorithm, it is possible to state that the computational load remains higher in the 2 most relevant steps of the algorithm, namely, the calculation of the distance of the points to each cluster and their respective (re)assignment in main (using N = 10000000, K = 4 and T = 2):

Overhead	Command	Shared Object	Symbol
78,53%	k_means	k_means	[.] euclidiana
10,24%	k_means	libc .so.6	[.] __vfwscanf ...
8,06%	k_means	k_means	[.] main
0,65%	k_means	k_means	[.] inicializa

We can now proof our expectations that our procedures `euclidiana` and `main`, would have the highest computational cost. With this in mind, let us effectively parallelize this algorithm with the OpenMP directives in the loop cycles, because it is where the largest computational load occurs.

Developed in University of Minho

B. Analysis

*1) **Initialization**:* We also investigated parallelization in random initialization of points, although this is not one of the code blocks with a high computational load. For this, we experimented parallelism in each one of the cycles of the function `initializa()`, not performing parallelism in any other function. Here, we were careful about the "fake" random behaviour of the `rand()` function: in our code, we initialized a random numbers generator in each parallel thread with various values through the combination of current time (seed as `int(time(NULL) ^ omp_thread_number)` and the current thread number. The execution time experienced an improvement, albeit very small (overhead of 0.65% to 0.50%. However, when combined with the parallelization of the `euclidiana()` function actually verified a considerable increase of the execution time (the extra overhead came from starting the threads on each loop). Thus, we chose not to parallelize it.

*2) **Euclidean Distance**:* For this function, we started by understanding whether it made sense to perform parallelization: yes, since it was naturally noticeable that there was similar work that could be shared between different threads. We considered several aspects while we were performing the parallelization, namely, writing operations on the same variable by different threads, reasonable distribution of work by each thread and changing the code to make the parallelization possible. Naturally, we used the right directives to solve each problem, as we will discuss below.

[Scenario 1] We first thought about covering the code block of the both for loops (first and second with the nested ones) of `euclidiana()` with a directive `#pragma omp parallel num_threads(threads)`, which would spawn a group of threads specified by us. Internally, in each main cycle we would place `#pragma omp for` that would distribute the work for each thread, however, in this case, it was necessary to guarantee that the first cycle was carried out and only then would the next one would be executed, with `#pragma omp barrier`, to guarantee it.

[Scenario 2] Secondly, the idea was to generate the group of threads before each loop, using `#pragma omp parallel for num_threads(threads)`, and imme-

diately distribute the work. We were aware about the extra overhead of spawns extra threads, but we thought it might compensate. In this stage we were not thinking much about scheduling.

C. Final Implementation with OpenMP Analysis

1) ***Reduction clause****:* The final implementation considers important aspects such as parallelization, synchronization and correction of our program. Our program has to remain correct after parallelization, since the existence of dependencies between variables can lead to incorrect results. Thus, we consider the use of the reduction clause instead of the combination of parallel and critical, since the first clause is optimized for specific situations of rewriting variables that are used after the cycle (one thread can read a value that is not the most updated one - variable min with WAR dependency). We must also consider the case with arrays that store the sum of the coordinates of the points and the respective points (these arrays with RAW dependency).

2) ***Scheduling****:* It is now well known that different types of scheduling have different overheads. For this section, we tested the different types and verified that with the use of static (with a smaller overhead), better results were verified. To determine the chunk size, we were also testing and verifying the value from which no gain (in terms of execution time) was evaluated. It makes sense using static because each iteration will have almost the same amount of computational load to be performed, so each thread spends almost the same amount of time performing its chunk.

D. Scalability

Each Search node has 20 cores, then let's calculate the maximum speedup in each of the versions. First, we identify the total time and the parallelizable code portion time using `omp_get_wtime`. With 4 Clusters: total time is `4.178` and parallelized one is `3.021438`. With 32 Clusters: total time is `22,035383652` and parallelized one is `22.024466`. Using the Amdahl's law (1), we can calculate the speed-up (S) for 20 PUs, being f the sequential portion:

$$S_{20} = \frac{1}{f + (1-f)/20} \tag{1}$$

III. Performance Tests

	Execution time		
Number of Points	**Sequential**	**Parallel**	**Speed-Up**
1 000	0.0020128	0.0021968	0.916241806
10 000	0.0059261	0.0060306	0.982671708
100 000	0.0517333	0.0423711	1.220957209
1 000 000	0.5154598	0.4033792	1.277854188
10 000 000	4.5125483	4.003171	1.127243453
100 000 000	45.1873995	39.3874058	1.147255032

TABLE I
PERFORMANCE VARYING THE NUMBER OF POINTS

	Execution time		
Number of Cluster Points	**Sequential**	**Parallel**	**Speed-Up**
2	3.2634373	4.06823957	0,802174317
4	4.0313969	3.96495757	1.016756631
8	7.3347390	6.79532066	1.079380851
16	13.701922	10.6249463	1.289599177
32	27.104697	17.5640108	1.543195191
64	55.605142	32.9831656	1.685864319

TABLE II
PERFORMANCE VARYING THE NUMBER OF CLUSTER POINTS

The next table was generated with different values. Instead of using the CLUSTER, we used our machine due to high usage of CLUSTER, making impossible to take statistics. The machine used was an Intel I5 Core 8th Generation with 12 GB of RAM and 256 GB of SSD storage.

	Execution time	
Number of Threads	**Parallel**	**Speed-Up**
1 (Sequential)	3.0601430	1
2	2.7693082	0,9049604
4	3.1729849	1,0368747
8	2.6403263	0,862811411
16	2.7375353	0,894578
32	2.6196662	0,8560601

TABLE III
PERFORMANCE VARYING THE NUMBER OF THREADS

IV. Conclusion

In this second phase, we explored the existence of parallelism in the code blocks with the highest computational load, using the various clauses to achieve a better load distribution among the threads, maintaining their synchronization and the correct functioning of the program. We consider that we carried out a good parallelism analysis, although the estimated improvement in terms of time is not very high.