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*—The ability to organize elements or entities with more similar characteristics has become increasingly important and has marked its impact in terms of various applications in the real world, such as in industry, health, technology or even art. This idea is the base of the K-Means algorithm: given a sample of N points and a number of K clusters, this algorithm calculates the distances from each point to the centroid of each cluster, assigning the point to the cluster whose distance to the centroid is minimum. So, the output is a set of clusters with associated points with the follow property: all points within a cluster are closer in distance to their centroid than they are to any other centroid. K-Means algorithm usually does not take too long, but it is expected to reach his parallelized version that operates with an improved execution time.**

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

I. Introduction

The main purpose of this assignement is to understand the advantages of algorithms' parallelization in C, through the development of a first version of K-Means, an optimized sequential algorithm that will then have to be transformed in order to enjoy the properties of parallelization. So, we will analyze each performance considering appropriate metrics and comparing results properly. This well-known data partitioning algorithm will be explored further on.

Analysis of results is crucial, using appropriate metrics and methods, such as observing assembly instructions, execution time, number of instructions, CPI, number of cache misses, etc. these metrics are detailed below.

II. Sequencial K-Means

A. Implementation

Our implementation is based on Lloyd's algorithm, which allows us to quickly and easily perform the partitioning through 2 main phases. Firstly, the creation of a sample of random points, with the assignment of a random centroid to the clusters. Then, each point is assigned to the cluster with the closest centroid. Secondly, the recalculation of the centroids of each cluster and new assignment of points, until the algorithm converges.

We start by initializing two static vectors: one with the N abscissa of each randomly generated point and the other with the respective ordinate. Then, we initialize two other static vectors: one stores the abscissa of each unique centroid of each clusters and the other stores the respective ordinate.

Then, it's time to calculate the distance between each point and each centroid, in order to find the closest cluster. For that, we use other array that stores all these distances to a specific centroid from a specific point. For each point, we store the closest centroid, with an array where each index references a point and the content is the closest centroid, finally, we compute the sum of each coordinate storing the result also in a new vector.

In order to verify convergence, we copy the array with the current cluster-points assignment to a new one. While the algorithm doesn't converge, we will recalculating the centroids using the previous two arrays which stored the sum of each points coordinate.

B. Analysis of possible optimizations

During the implementation, it was observed that certain changes in the source code obtained better results in terms of execution time, such as the use of static arrays instead of dynamic ones (without resorting to the heap) or the simplification of the distance calculation formula, since it would not be necessary to use functions such as *pow* or *sqrt*. Despite this, some compilation optimizations were also studied, such as the use of *vectorization* and *loop unrolling*. We did't use the last one because N and K can be even or odd, meaning more operations.

Starting with the flags that were used, we combine gcc compilation flags such as `-O1`, `-O2`, `-O3` with `-funroll-loops` and `-ofast`, to understand the impact they have on performance and which changes were performed. After digging about it, we've concluded that it's not recommended to use `-ofast`, because it actually degrades performance with floating-point operations since this flag enables `-ffast-math`. Besides, `-mfpmath=sse` enables the use of XMM registers in floating point instructions (instead of stack in x87 mode), which we expect to be valuable. There's no need to use the `-ftree-vectorize`, because gcc enables auto-vectorization with `-O3`.

The `-O` flags have different purposes: the flag `-O1` forces the compiler to reduce code size and execution time, without

Developed in University of Minho

performing any optimizations that take a great deal of compilation time, `-O2` increases both compilation and performance and `-O3` optimizes even more. Having this in mind, we start by testing each `-O` flag. Between `-O1` and `-O2`, we expected to have a decrease of the execution time: with the increase of optimization, there's a decrease of the number of instructions and the gain was not that much. We try to understand why has this happened: looking at results, `-O2` has less `#I, #CC, #Misses, CPI`, but a higher time, because `-O1` of the higher complexity of assembly instructions to perform, even though they're less.

Considering the existence of 4 main loops, excluding the assignment ones, we expected the unrolling and vectorization to reduce the execution time - and it was proved by the results obtained - because inevitably we do less iterations and cycle test conditions. In the assembly code, we detect a loop unrolling of 4 and that all loops were vectorized, what resulted in a cleaner code. Finally, we can conclude that the best option for optimization is with `-O2 -funroll-loops -ftree-vectorize`, what produces an execution time of 4.178 seconds.

As explicit above, the `-msse2 -mfpmath=sse` flags were used in all test cases, being implicit in the cases below, although they are not shown in the table below.

As we were observing the changes at the assembly level, we were also getting different results for each performance metric, mainly the execution time per optimization. These mean results are shown in the table below, for a sample of N = 10000000 points and K = 4 clusters.

Optimization (flags)	Execution Time (s)	#I (inst_retired.any)	#Misses (L1_DMiss)	#CC	CPI
-O1	6.633	32369374850	131188984	20627425234	0.7
-O2	6,819	31965188865	130860055	17446725469	0.5
-O3	4,273	26358928059	130490322	13365584920	0.5
-O2 -funroll-loops	4,743	26336926813	130626841	14259650090	0.5
-O2 -ftree-vectorize	4,524	17959160198	130589689	14623039720	0.8
-O2 -funroll-loops -ftree-vectorize	4,178	17334670812	130481064	13348257865	0.8
-O3 -funroll-loops	6,663	26343154911	131189718	20996005877	0.8

TABLE I
OBTAINED RESULTS OF OPTIMIZATIONS.

	Execution time		
Number of Points	**Sequential**	**Parallel**	**Speed-Up**
1 000		0.0033667	
10 000		0.0068068	
100 000		0.1397496	
1 000 000		0.5879930	
10 000 000		6.5286183	
100 000 000		39.3874058	

TABLE II
PERFORMANCE VARYING THE NUMBER OF POINTS

III. CONCLUSION

In this primary phase of the assignment, we were able to develop a optimized sequential version of K-Means algorithm, evaluating the impact of each type of optimization on each metric used to compare the performance of the program. We also can predict that the parallelized version of this algorithm can decrease even more the execution time and so increase the performance of the algorithm.