SIMPLE K-MEANS ALGORITHM

Ana Filipa Ribeiro Murta
Informatics Department
Master in Informatics Engineering
University of Minho
Guimarães, Braga, Portugal
pg50184

Ana Paula Oliveira Henriques
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Braga, Portugal
pg50196

***Abstract* — This report aims to present and analyze the performance of an optimized sequential version of a simple k-means algorithm, inspired on the Lloyd algorithm.**

Keywords — algorithm, sequential, k-means, cluster, optimization, complexity, performance, analysis, solution.

I. INTRODUCTION

The purpose of this project is to evaluate the code optimization techniques learned in the scope of the course *Parallel Computing*, by firstly implementing a simple k-means algorithm through the usage of C programming language. Based on the Lloyd algorithm, this program assigns a set of points to its closest cluster. Consequently, the points within a cluster are closer to their centroid than to any other centroid in the program. As long as the ideology is preserved, some changes to the original algorithm are authorized since the main goal is to minimize the total execution time of the program. After the code development comes the code optimization, where the techniques applied are loop unrolling and vectorization. At last, we proceeded to evaluate the program's performance, taking advantage of some analysis' metrics obtained when running commands with the tool *perf*.

II. K-MEANS ALGORITHM

K-means clustering is a method of vector quantization, where N two-dimensional points (x,y) are partitioned into K clusters. The algorithm starts by initializing the N points with random values (from 0 to 1) and the K clusters' centroids with the first K points. Subsequently, the other N-K points are assigned to the closest cluster to them according to the Euclidean distance. These clusters will serve as prototypes of the final clusters. After this initialization phase, the clusters' centers will be recalculated, and the points will be once more assigned to the closest cluster to them now that their centers are different. This last process will repeat until the points no longer switch clusters.

In our solution, we use a global variable, *points*, to save the N points. This variable is an array with N*3 elements because, besides the coordinates of the point (x,y), we also save the cluster to which the point is assigned. Aside from this, our program uses another tree arrays to implement this algorithm: *num_elems* (size K) where it's kept the number of points of each cluster; *sum* (size K*2), where it's saved the sum of the points (x,y) of each cluster; and *centroids* (size K*2), where it's kept the centroids (x,y) of each cluster.

```
if(N <= K) notOver = 0;
do{
    allEquals = addToClosestCluster(count,
num_elems, centroids);
    if(allEquals == N) notOver = 0;
    else count++;
} while (notOver);
```

The strategy used to verify if we already found our solution is through the variable allEquals. Basically, we increment this variable whenever the cluster to which the point is assigned to didn't change. Therefore, if allEquals equals N, it means that the points didn't exchanged clusters, being in the same cluster they were before.

```
if((beforClu == minClu) allEquals++;
```

In our previous model, the array was iterated N*2 times. To optimize the code, we go through the array points N times, and, for this to happen, it's executed two things. First, it's calculated the closest cluster to the current point iterated. The other thing is, depending on the cluster to which the point is assigned to, the array sum is accessed. As previously mentioned, this array contains the sum of the points of each cluster, so the coordinates of the current point iterated are sumed and stored in the array using the number of cluster as an index.

A. Complexity

TABLE I. ALGORITHM'S COMPLEXITY

Complexity		
Best Case	Worst Case	Average Case
$O(\mathrm{N}+\mathrm{K})$	$O(N^2)$	$O(\frac{\mathrm{N}+N^2}{K+K})$

III. LOOP UNROLLING AND VECTORIZATION

After implementing the k-means algorithm based on the Lloyd algorithm, the team attempted to optimize the performance of the solution by using loop unrolling and vectorization.

- *Loop Unrolling*

Loop unrolling is a loop transformation technique that optimizes a program´s execution time by reducing the number of iterations. Loop unrolling increases the program's speed by removing and reducing iterations. Basically, in the loop body, it adds the necessary code so that the loop can occur multiple times and then it updates the conditions and counters accordingly. An advantage of loop unrolling is the improvement of the program's efficiency and the reduction of loop overhead.

- *Vectorization*

Vectorization is a parallel computing method that maximizes computer speed. It allows the execution of repetitive instructions simultaneously by transforming them into a single vector. This means that vectorization transforms

an algorithm from operating on a single value at a time to operating on a set of values at one time to produce more and better results.

Both techniques were automatically executed by *gcc* compiler. To enable the loop unrolling optimization, we added the flag *-funroll-loops* into the Makefile's set of flags. We also added the flag *-ftree-vectorize -msse4* to test the vectorization optimization.

IV. Solution Evaluation

The results of our solution weren't quite desirable. After abandoning a model of dynamic memory allocation to a model where the array variables used were static, we are aware that our algorithm is heavier than expected due to some loops inside loops. In the table presented down below, there are the results obtained after executing the program with the command *perf record.*

TABLE II. Results to different N values

N Points	Flags		Time (sec)	CPI	#I	branch-misses/branches
1000	-O2		0.003	0.96	2586355	7.90%
	-O3		0.003	0.94	3055198	3.93%
	-O2	-funroll-loops	0.002	1.20	2227268	7.64%
		-ftree-vectorize -msse4	0.003	1.25	2798137	2.41%
	-O3	-funroll-loops	0.002	0.82	2986771	4.15%
		-ftree-vectorize-msse4	0.003	1.26	3085862	3.39%
1000000	-O2		0.875	0.75	3041780009	12.86%
	-O3		0.658	0.51	3750848775	0.13%
	-O2	-funroll-loops	0.793	0.83	2442198563	23.26%
		-ftree-vectorize-msse4	0.556	0.41	3354463995	0.05%
	-O3	-funroll-loops	0.725	0.50	3676763662	2.18%
		-ftree-vectorize-msse4	0.753	0.51	3751288041	0.13%
10000000	-O2		8.565	0.75	33639287023	13.05%
	-O3		7.306	0.51	41516817300	0.11%
	-O2	-funroll-loops	7.802	0.83	26977887792	23.70%
		-ftree-vectorize-msse4	5.358	0.41	37109814077	0.04%
	-O3	-funroll-loops	7.193	0.50	40693739520	1.78%
		-ftree-vectorize-msse4	7.374	0.51	41518499639	0.11%

V. Conclusions

Even though the final result could be more optimized, the group put in practice the content learned in the practical classes involving code optimizations. As previously emphasized, we are aware that the code could support more improvements and that's why, in the next phase, we expect to present a more sufficient and optimized version of this algorithm while fulfilling the new phase's requirements. For example, taking advantage of spatial locality and applying loop unrolling and vectorization in the same loop