Parallel Computing
K-means Algorithm

Laura Rodrigues
Masters in Software Engineering
University of Minho
Braga, Portugal
pg50542@alunos.uminho.pt

Mariana Rodrigues
Masters in Software Engineering
University of Minho
Braga, Portugal
pg50628@alunos.uminho.pt

***Abstract*—The K-means Algorithm takes a given number of points and groups them by clusters. As a result we get a set of cluster centroids that allow us to organize the points. Each point is closer to it's centroid than to any other centroid. In this paper we implemented and analysed an optimised sequencial version of this algorithm.**

***Index Terms*—k-means, cluster, centroid, point**

I. Introduction

The goal of this project was to test several code optimization techniques while trying to minimize the execution time. Due to the vast variety of optimizations, we had to analyse the performance impact of each that was implemented.

II. Implementations

A. Original version

We started our implementation by thinking about the structures we needed to define and how we should define then.

We decided we would define a struct Point, that would have two floats, which would be equivalent to coordinates.

Secondly, we needed to think of a way to define the clusters. As we know, the clusters should have the points that are assigned to it and a center point (centroid). By looking at the algorithm, we know that we would have to move the points of this universe from one cluster to another and also determine the distance between them and the center of the cluster they are in.

To do so, we would have to move a lot of data around. In order to simplify it, we chose to add to a point a variable that would indicate to which cluster the point is assigned to. This would allows us to easily change a point from one cluster to another.

We also defined the centers of all the clusters as an array and grouped all the points that exist inside an array. We did so to take advantage of the spatial locality.

```
typedef struct point{
    float x;
    float y;
    int nCluster;
} * Point;
/.../
Point *array_points = createArrayPoints();
/.../
Point *centroids = createArrayCentroids();
```

Initially, the data was stored dynamically using the malloc function.

Our algorithm started by initializing every point and centroid with a random number at the *init* function.

In a loop, we then determined the centroids (*determine_new_centroid*) and updated the clusters (*update_cluster_points*) multiple times until no more points would change cluster. Therefore, we can considered that this previous function mentioned execute most of the work of our implementation.

determine_new_centroid : Determines the central position of each cluster by calculating the sum of all the coordinates of points in it and dividing it by the cluster's size.

update_cluster_points : Determines the distance between every point and every centroid and verifies if any point should be moved around. Returns how many points were changed from one cluster to another.

In order to better understand the complexity of our problem and implement a better version of it we analysed the complexity of our most important functions.

Function	Complexity
init	O(N x K)
update_cluster_points	O(N x K)
determine_new_centroid	O(2 x K + N)

TABLE 1: Complexity Analysis

B. Math Library

We decided to use the command *perf* to analyze which functions would have the more significant overhead. We obtained the following results:

Function	Overhead
determineDistance	41.76%
update_cluster_points	33.95%
determine_new_centroid	17.22%
init	0.79%

TABLE 2: Results of the command perf.

The function *determineDistance* is a function used by *update_cluster_points* and that would be called in all the iterations. This function calculates the euclidean distance between two points, and used the math library.

This library is quite heavy to use, not only because of its functions weight but also because it add extra calls for functions.

As mentioned, we used the Euclidean formula to get the distance between different points; however, we would only be using the values resulted to compared them with other results. So we concluded that we could remove the function *sqrt*, as the results from the comparison without it would be the same. We also chose to replace the function *pow* by the multiplication of the values.

C. Loop Unroll

Without any optimization option, the compiler tried to reduce the cost of compilation and to make debugging produce normal results. In order to make the compiler attempt to improve the performance and code size we tried to use a loop unrolling flag.

When we used the loop unroll flag, the program tried to get the number of iterations that can be determined at compile time or upon entry to the loop. The flag *-funroll-loops* made code larger and made it run faster.

We also opted by adding the *–funroll-loops* flag to the makefile in some of the versions tested.

D. Vectorization

By definition, vectorization is the process of converting an algorithm from operating on a single value at a time to operating on a set of values at one time.

Just like with loop unrolling we added optimization flags to our makefile so that we could benefit from the vectorization process. The *-ftree-vectorize -msse4* option turns on auto-vectorization on.

E. Memory Allocation

After the previous version presented, we decided to try and change the way the data was allocated. Previously, we were using the function malloc() to dynamically store the data. When using this function, there would be some extra time that would be spend on allocating the data, as this is allocated at runtime, and on calling the function. On the other hand, when memory is statically allocated, it is done at compiling time. Therefore, we did some tests with static allocated memory.

```
struct point
{
    float x;
    float y;
    int nCluster;
};
struct point array_points[N];
struct point array_centroids[K];
```

III. Results Comparison

To take a closer look at the possible optimization options we decided to try different ones together. In order to establish some comparison, we will look into the following different metrics: execution time in seconds, number of cycles, number of instructions and cycles per instructions.

#V	TExec(s)	#Cycles	#I	CPI	L1-Misses
1	21.16	6.58x10^{10}	5.64x10^{10}	1.2	5.24x10^{8}
2	13.66	4.21x10^{10}	3.63x10^{10}	1.2	5.24x10^{8}
3	10.80	3.37x10^{10}	2.75x10^{10}	1.2	5.22x10^{8}
4	5.62	1.74x10^{10}	2.66x10^{10}	0.7	5.15x10^{8}
5	9.79	3.12x10^{10}	3.31x10^{10}	0.9	1.54x10^{8}
6	7.75	2.41x10^{10}	2.41x10^{10}	1.0	1.54x10^{8}
7	8.94	2.79x10^{10}	3.46x10^{10}	0.8	1.53x10^{8}
8	4.03	1.25x10^{10}	2.34x10^{10}	0.5	1.54x10^{8}

TABLE 3: Results of different solutions

The **first version** corresponds to the original code already described above. As we can see it is the one with the worst results in the various parameters shown. Clearly showing it's a version that should be improved.

On the **second version** we removed the Math Library functions already mentioned (sqrt and pow). This was the optimization that most affected our execution time.

For the **third** test, we compiled the previous code obtained with the flag *-funroll-loops*. For the **forth** one, we combined the previous one with the flag *-ftree-vectorize -msse4*. The use of the first flag, created a slight improvement; however, it proved to be more beneficial when combined with vectorization. This turned out to be the best implementation, for when the data was dynamically stored.

Version five was when we removed memory allocation. This code version's performance can be compared to version two. Without any optimization flags, we still see a huge performance enhancement.

For the last three versions, we compiled the code from the version five with the flags mentioned previously.

Version six we only used the flag *-funroll-loops*; for **version seven**, we used the flag *-ftree-vectorize -msse4*. Lastly, for **version eight**, we combined both flags. When comparing the results obtained when the flags are used separately, we can observe that the flag *-funroll-loops* proved to be more beneficial, as it reduced the execution time, the number of cycles and instructions; even thought, the **version seven** presents a smaller CPI in comparison, we can't deem it positive, as it is due the higher number of instructions. The version that showed the best performance results in all the metrics was the last one, where both flags are used combined.

Lastly, we can observe that when data is statically allocated, the number of L1-cache misses decreases significantly. Then, with this changes, we were able to take advantage of spatial and temporal locality, which was the proposed of our initial implementation of structures.

IV. Conclusions

Given all the study that was taken in this algorithm's performance, our final code was of version eight, compiled with the flags *-funroll-loops* and *-ftree-vectorize -msse4*.

This report shows a detailed study of the k-means algorithm, possible improvements and tests performed. It also allowed us to fully understand the importance of each optimization and how it affected our program.