CP - TP1

1st Gonçalo dos Santos
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg50396

2nd Luís Faria
Departamento de Informática
Universidade do Minho
Braga, Portugal
pg50560

I. Introduction

This project's objective was to develop, in C, a sequential and optimized version of a simple k-means algorithm, based on Loyd's algorithm.

The algorithm can be resumed in the following steps:

1) Create samples and initiate *clusters*;
   a) Initiate a vector with random values of N size in a (x,y) space;
   b) Initiate the the K clusters with the coordinates of the first sample;
   c) Attribute each sample to the nearest cluster using the euclidean distance;
2) Calculate each *cluster*'s centroid (also known as the geometric center);
3) Attribute each sample to the nearest *cluster* using the euclidean distance;
4) Repeat the second and third steps until no point changes *cluster*.

II. Implementation and Optimization

While planning the implementation of the project, the group decided that the first version of the program must already be as optimized as our knowledge allows it.

Which made our first version be able to vectorize all of the iterations in the algorithm and make the most use of space and temporal locality by using the same array to algorithm related variables and guarantee the continuity by not altering the same variable in consecutive operations.

A. Algorithm

After the first implementation, we tried to to optimize the algorithm and decided to calculate the distances with this formula:

$$x^2 + y^2 \tag{1}$$

That is based on the euclidean distance formula:

$$\sqrt{x^2 + y^2} \tag{2}$$

The group did that because the algorithm only compares the distances and doesn't need the specific values and supporting our decision is the following equivalence:

$$x < y \equiv \sqrt{x} < \sqrt{y} \qquad x, y \in \mathbb{R} \geq 0 \tag{3}$$

This decision was also based on the fact that the mathematical function **sqrt()** is computationally heavy.

The stats after this optimization can be seen in figure 1.

B. Code

After optimizing the algorithm, the next step was to analyse the code, where we noticed that we were iterating the same list twice.

We tried to integrate those two iterations into one and noticed that the execution time had gone down and the iteration was no longer vectorized but unrolled, which was could be seen in the increase of instructions, as we expected.

In this optimization the group obtained the stats shown in figure 2.

After the previous optimization, the group didn't find any part of the code that could be optimized with our experience at that moment, so we experimented changing some parts and found a few cases of optimization.

One of the cases we found was changing the expression

flag = (old_points[i] != ind)?1:flag

to

if(flag) flag = (old_points[i] != ind)?1:flag

and noticed that even though we used **?**, that is less computationally heavy than **if**, because we had used a memory access as the sole condition, it created a lot of cache misses and by introducing a **if(flag)**, cache misses decreased massively. We end up simplifying the expression to

flag = (flag or old_points[i] != ind)

and obtained the stats shown in figure 3.

The group's last optimization was to delete one of the the two lists we had dedicated to store the old and the attribution of the points to the clusters, to do the comparisons. We did it because we noticed that one list was enough to serve it's purpose and that by doing so there were marginally less cache misses and branch misses. The figure 4 shows the program final stats.

C. Compilation

To compile the program, we used the following flags:

-std=c99 -O2 -ftree-vectorize -msse4 -funroll-loops

We used the flag **-std=99** because we were required to compile the code using the c99 version.

We picked the **O2** compilation flag to apply the most generic optimizations that the compiler has available, without it increasing the size of the generated binary file, and because it gave us the best results, comparing to the **O1** and **03** flags, as we can see from comparing the stats in the figures **??**, **??** and **??**.

To support the **O2** flag, the group used **ftree-vectorize** to allow the vectorization of instructions together with **msse4** that indicates to the compiler that it has available records of 256 bits size and with **funroll-loops** to unroll the loops that the compiler considers beneficial to performance.

III. Conclusion

We finished this first project satisfied with what we accomplished and pleased about reinforcing the previously learned knowledge about the subjects of optimization of code and it's compilation, as well as further deepening our understanding of the subjects.

We would like to further improve this project with the the knowledge that will be deepened with the next objectives of this class.

IV. Anexos

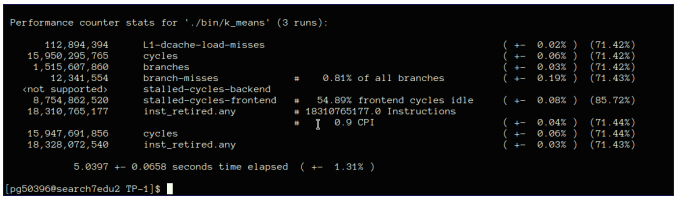

Fig. 1. Algorithm Optimization

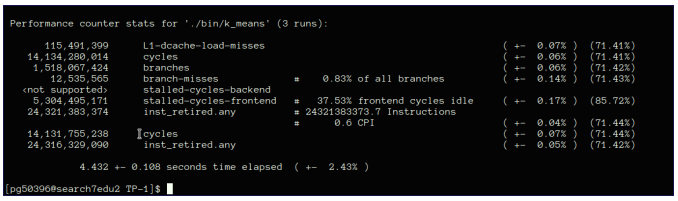

Fig. 2. Iterations joined

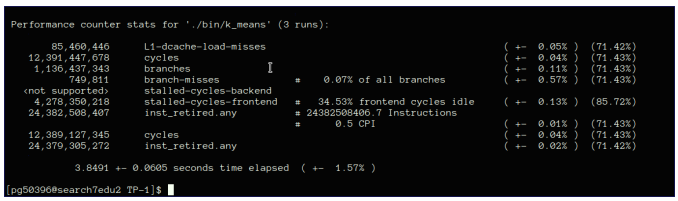

Fig. 3. *If statements* simplified

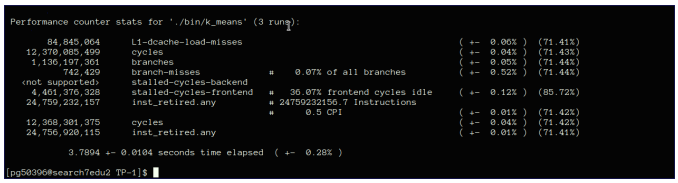

Fig. 4. One point-cluster attribution list

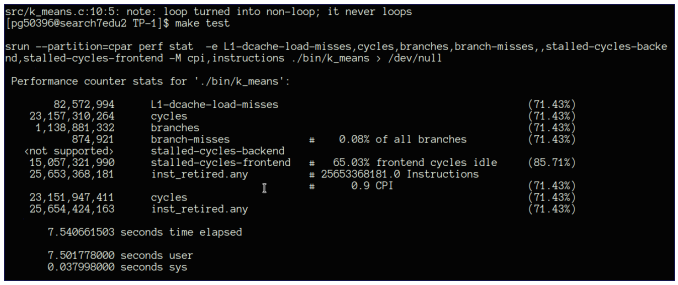

Fig. 5. flag -O1 compilation

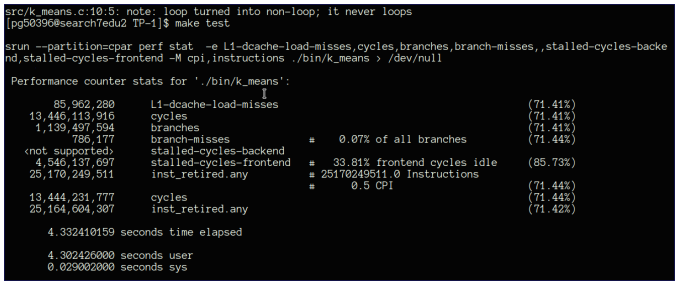

Fig. 6. Oflag -O2 compilation

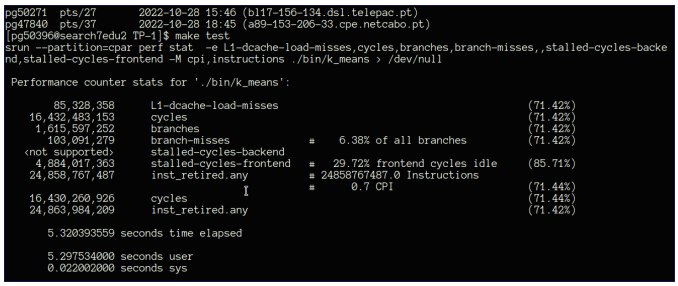

Fig. 7. flag -O3 compilation