K-Means Algorithm Optimization

1st José Gonçalves
IT department
University of Minho
Braga, Portugal
pg50519@alunos.uminho.pt

2nd Miguel Fernandes
IT department
University of Minho
Braga, Portugal
pg50654@alunos.uminho.pt

***Abstract*—The present document refers to a practical assignment from a Masters Degree in Software Engineering course unit named Parallel Computing. The project goal is to develop a sequential version of a K-Means algorithm, as-well as optimize it in order to obtain a efficient and fast result.**

***Index Terms*—k-means, parallel computing, performance, optimization, analysis**

I. Introduction

The present project aims to develop a sequential version of the K-Means, by evaluating and exploring in order to reach the maximum performance, without compromising parameters like code legibility and scalability. Succinctly, this algorithm consists in calculating clusters and placing them in a position between all the points assigned to it, and will be explained in detail later in this document.

This report is divided in four sections, starting from an introduction where the algorithm is presented. Right after, the second chapter describes the initial approach taken by the group, which had the main concern was of achieving the algorithm correctly and obtain the right results, but it had a bad memory management, which led the group to analyse and start optimizing, as described in the following chapter. The third chapter describes the optimized version, where the goal was to analyse the code with the desired tools and start optimizing common hotspots areas while interpreting the resulting impacts. It describes and explains all the changes made, the new functions created and the results that it led to. Finally, the fourth and final chapter resumes and concludes about the project, as well as describes some of the improvements that could be made to make the algorithm more effective.

II. Initial Implementation

A. How K-Means algorithm work

K-means is a algorithm that consists in a segregation method around clusters called *centroids*, that results in partitioning N elements in K groups where each observation is part of the closest group.

B. Data Structures

In a first approach, the structures created consisted in a struct called *point*, which had the x and y coordinates and a struct called cluster, that had the size, the center and the set of points (as an array of pointers to point struct) that are part of it. To store all the points and cluster information, it was used a point array and two arrays of the cluster struct - one for the actual cluster and the other to save the information of the last cluster before it got updated.

```
struct point{
    float x;
    float y;
};

struct cluster{
    int size;
    struct point *center;
    struct point **elements;
};
```

C. Functions

Initially, the solution was designed to be divided in different functions with very specific jobs. With that in mind, we created 7 functions to achieve the main objective of getting the desired results:

1) **Initialize**
   - Allocate space for 3 data structures (struct point with N elements, struct point with K elements for centroids and struct cluster with K elements).
   - Initialize all N elements with random numbers.
   - Initialize all K centroids with the first K elements.
   - $\mathcal{O}(N + K + K)$
2) **Reevaluate Centers**
   - Calculate the centroids of each cluster based on the mean of their points
   - Iterates by clusters
   - $\mathcal{O}(N)$
3) **Cluster Points**
   - For each point, calculates the distance to all centroids.
   - Insert the point dynamically in the cluster(*realloc*)
   - $\mathcal{O}(N * K)$
4) **Clear Cluster**
   - In the end of each iteration, resize all clusters to 0.
   - $\mathcal{O}(K)$
5) **Clone Cluster**
   - To verify if all points remain in the same cluster after an iteration, the cluster is cloned and stored before every iteration
   - $\mathcal{O}(N)$
6) **Has converged**
   - Compare the new cluster with the cloned one
   - $\mathcal{O}(N)$
7) **Free Clusters**
   - Free all data structures allocated initially
   - $\mathcal{O}(K + K)$

D. Algotithm analysis

After the implementation it's important to run *perf* to analyse the performance of the program. The commands used were *perf stat* and *perf report*.

Perf stat results (see fig.1) show that the program was making almost 335 millions cache-references with a miss rate of $39,67\%$. The number instructions of **inst. per cycle** was $1,29$ and that shows the instructions were executed almost by a sequential order, which led to think that the algorithm could be improved and take more advantage of that.

The result of *Perf report* (see fig.1) indicates that *cluster_points* must be the main goal of optimization, since its taking almost 60% of samples obtained.

III. Optimized Implementation

In order to tackle the previous version downsides and its associated overhead, some changes were made, that led to a much better results both in terms of performance and readability.

A. Data structures

Considering that miss-rate and the huge number of cache accesses were a problem in first implementation, the way that data was stored needed to be changed. So all the structures used before were eliminated and replaced by arrays in order to reduce this problem.

Aiming to minimize that issue, 6 arrays were created:

- Array with K integers that store the size of each cluster.
- Two arrays with K floats that store the centers of each cluster, one for the x coordinates and another to the y coordinates.
- Two arrays with N floats that store all points, one for the x coordinates and another to the y coordinates.
- Array of N integers that store the cluster each point belongs to.

The new way of storing all the data replaces the structures defined before that used a lot more space by storing the same data multiple times and not allow to take advantage of spacial locality when going through all points. Also the old structures obligate the use of *realloc* when assign a new point to a cluster, which lead to the fragmentation of the memory and worst performance, because calling *realloc* too many times is very expensive.

To understand how the arrays take more advantage of spacial locality let's take a look in the differences to calculate the centroid of each cluster, where it's necessary to add all their points. In the initial implementation, the code was:

```
for(int i = 0; i < K; i++) {
    ...
    for(int j = 0; j < clusters[i].size; j++) {
    sum_x += clusters[i].elements[j]->x;
    sum_y += clusters[i].elements[j]->y;
```

And in the optimized one is:

```
for(int i = 0; i < N; i++) {
    sum_x[cluster[i]] += pointX[i];
    sum_y[cluster[i]] += pointY[i];
```

Considering that the cache is cold, in the first implementation the cache loads 1 block of 64 bytes and each cluster has 20 bytes, so it loads 3 completed clusters(1 miss in every 3 external iteration), knowing that 8 bytes are from center and that is never used in this calculation. Beside that the cache needs to load one more block of points to get the coordinates that contains 8 points but the elements of a cluster couldn't be consecutive and results in more misses.

In the second implementation the cache loads 1 block to get the cluster, that is an array of integers, so the block could contain 16 integers, resulting in one miss on every 16 loops. The same happens for pointX and pointY, a block contains 16 floats so the miss rate is 1 in 16.

There were numerous situations like that in first implementation, like the calculation of the distance between a point and a center that happens N*K times. So the new solution solve the problem of the huge number of accesses to cache and also the number of misses as will be seen in the subsection **Optimization analysis**(III-C).

B. Functions

After a close look to the implementation, it was clear that some functions can be done at the same time, reducing the number of iterations. So the 7 functions in the first implementation, now are 3 and the main differences are:

1) **Initialize**
   - Allocate N data structures for all N points.
   - $\mathcal{O}(N + K)$
2) **Reevaluate Centers**
   - Iterates by points (more spacial locality) and calculates the cluster to each one of them.
   - $\mathcal{O}(N + K)$
3) **Cluster Points**
   - There is a flag that after assigning a point to a cluster, verifies if that point changed from one cluster to another changing the flag to true, replacing the functions *Has converged*, *Clone clusters* and *Clear cluster* that were present in the initial version.
   - $\mathcal{O}(N * K)$

The complexity off the algorithm went from
$\mathcal{O}((N + K + K) + N + (N * K) + K + N + N + (K + K)) = \mathcal{O}(4N + 5K + (N * K))$
to
$\mathcal{O}((N + K) + (N + K) + (N * K) = \mathcal{O}(2N + 2K + (N * K))$

C. Optimization analysis

Due to the use of less structures and the store of less data, the results of the optimized implementation are much better from the older version. Doing a simple comparison:

- The number of cache-references went from 335 millions to 20 millions and consequently the number of misses went from 133 millions to 5 millions.
- The number of Last Level Cache (LLC) misses that results in a biggest penalty and the access to the memory, were reduced 52 million in loads and 78 millions in stores.

By reducing the complexity of the functions, the number of instructions are significantly lower, more specifically the number was reduced by a third.

D. Results

The optimized implementation was tested with *perf stat* several times in the **Search Cluster**, the execution time was always between 3.6 seconds and 3.9 seconds, being the best result 3.66 seconds (see fig.4).

IV. Conclusion and final considerations

Given the completion of the first practical assignment, we conclude that the result is an optimized and scalable algorithm that ensures the objectives. Overall, after the initial approach, many steps were taken back and forward in order to test many optimization techniques and the presented improvements and their justification makes all sense to the final result.

However, we also think it could be even better by implementing the vectorization with *pragma omp simd* which would led to a better performance. The optimized implementation have if conditions in the loops that disable this optimization, problem that was tried to solve but led to worst results.

V. Attachments

```
[pg50519@search7edu2 ~]$ gcc cod_antigo.c -O2 -std=c99 -lm
[pg50519@search7edu2 ~]$ srun --partition=cpar perf stat -B -e cache-references,cache-misses,cycles,instructions,branches,faults,migrations,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,L1-icache-load-misses,LLC-loads,LLC-load-misses,LLC-stores,LLC-store-misses,LLC-prefetches ./a.out
Center: (0.250, 0.750) Size : 2499108
Center: (0.250, 0.250) Size : 2501256
Center: (0.750, 0.250) Size : 2499824
Center: (0.750, 0.750) Size : 2499812
Iterations: 39

 Performance counter stats for './a.out':

       334796038      cache-references                                        (21,43%)
       132828094      cache-misses              #   39,674 % of all cache refs      (28,57%)
     72279542121      cycles                                                  (35,72%)
     93131352126      instructions              #    1,29  insn per cycle                                        (42,86%)
     13615117181      branches                                                (42,86%)
         906456       faults
              0       migrations
       559329241      L1-dcache-load-misses     #    2,97% of all L1-dcache hits    (42,86%)
     18850737696      L1-dcache-loads                                         (42,83%)
      5805181369      L1-dcache-stores                                        (14,28%)
        82759854      L1-icache-load-misses                                   (14,28%)
       412127899      LLC-loads                                               (14,28%)
       249899604      LLC-load-misses           #   60,64% of all LL-cache hits     (21,43%)
       108396280      LLC-stores                                              (14,28%)
        80207562      LLC-store-misses                                        (14,28%)
       200899161      LLC-prefetches                                          (14,28%)

    24,700776918 seconds time elapsed

    22,485009000 seconds user
     2,215099000 seconds sys
```

Fig. 1. *Perf stat* results for the first implementation

```
Samples: 53K of event 'cycles', Event count (approx.): 52345471065
Overhead  Command  Shared Object      Symbol
  58,47%  k_means  k_means            [.] cluster_points
   8,57%  k_means  libc-2.31.so       [.] __btowc
   6,44%  k_means  k_means            [.] reevaluate_centers
   6,25%  k_means  libc-2.31.so       [.] __strncpy_sse2_unaligned
   3,04%  k_means  libc-2.31.so       [.] __wcpncpy
   2,64%  k_means  k_means            [.] clone_cluster
   1,02%  k_means  [kernel.kallsyms]  [k] syscall_exit_to_user_mode
   0,84%  k_means  [kernel.kallsyms]  [k] irqentry_exit_to_user_mode
   0,77%  k_means  k_means            [.] 0x00000000000010e4
   0,74%  k_means  [kernel.kallsyms]  [k] clear_page_erms
   0,65%  k_means  libc-2.31.so       [.] __vfscanf_internal
   0,61%  k_means  [kernel.kallsyms]  [k] error_entry
   0,55%  k_means  [kernel.kallsyms]  [k] native_irq_return_iret
   0,55%  k_means  [kernel.kallsyms]  [k] syscall_return_via_sysret
   0,51%  k_means  [kernel.kallsyms]  [k] __irqentry_text_end
   0,46%  k_means  [kernel.kallsyms]  [k] entry_SYSCALL_64_after_hwframe
   0,44%  k_means  [kernel.kallsyms]  [k] sync_regs
   0,35%  k_means  [kernel.kallsyms]  [k] __handle_mm_fault
   0,32%  k_means  k_means            [.] inicializa
```

Fig. 2. *Perf report* result for the first implementation

```
[pg50654@search7edu2 ~]$ srun --partition=cpar perf stat -B -e cache-references,cache-misses,cycles,instructions,branche
s,faults,migrations,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,L1-icache-load-misses,LLC-loads,LLC-load-miss
es,LLC-stores,LLC-store-misses,LLC-prefetches ./a.out
Center: (0.250, 0.750)Size : 2499108
Center: (0.250, 0.250)Size : 2501256
Center: (0.750, 0.250)Size : 2499824
Center: (0.750, 0.750)Size : 2499812
Iterations: 39

 Performance counter stats for './a.out':

         20013197      cache-references                                              (21.40%)
          5774322      cache-misses              #   28.853 % of all cache refs      (28.55%)
      12044232054      cycles                                                        (35.69%)
      24284230127      instructions              #    2.02  insn per cycle
 (42.84%)
       2714749286      branches                                                      (42.87%)
             5837      faults
                0      migrations
        170043072      L1-dcache-load-misses     #    2.67% of all L1-dcache hits    (42.89%)
       6373656748      L1-dcache-loads                                               (42.70%)
       1444886699      L1-dcache-stores                                              (14.29%)
           376345      L1-icache-load-misses                                         (14.30%)
         32449252      LLC-loads                                                     (14.30%)
         18383588      LLC-load-misses           #   56.65% of all LL-cache hits     (21.44%)
          2098320      LLC-stores                                                    (14.26%)
          1821560      LLC-store-misses                                              (14.25%)
        134190482      LLC-prefetches                                                (14.25%)

      4.099308821 seconds time elapsed

      4.069078000 seconds user
      0.030000000 seconds sys
```

Fig. 3. *Perf stat* results for the optimized implementation

```
[pg50654@search7edu2 ~]$ srun --partition=cpar perf stat -e instructions,cycles ./a.out
Center: (0.250, 0.750)Size : 2499108
Center: (0.250, 0.250)Size : 2501256
Center: (0.750, 0.250)Size : 2499824
Center: (0.750, 0.750)Size : 2499812
Iterations: 39

 Performance counter stats for './a.out':

      24339796243      instructions              #    2.02  insn per cycle
      12070333451      cycles

      3.663597366 seconds time elapsed

      3.639304000 seconds user
      0.024002000 seconds sys
```

Fig. 4. Results of the optimized implementation