Parallel Computing Project 1

1st Lucas Carvalho
Minho University
pg50555
Braga, portugal
pg50555@alunos.uminho.pt

2nd Duarte Moreira
Minho's University
pg50349
Braga, Portugal
pg50349@alunos.uminho.pt

***Abstract*—This work aims to evaluate the optimization/code analysis techniques acquired in the PC - Parallel Computing - subject.**

***Index Terms*—optimization, analyze, performance, code, algorithm, k-means, cluster**

I. Introduction

As a first assignment for the PC subject, we were proposed to develop a sequential k-means algorithm in C language, based on Lloyd's algorithm, and then, proceed to apply optimization techniques preceded by an analysis of their impact.

II. Data Structs and Variables

To develop the algorithm we created two structs, the first one (**Fig.1**) to represent a point that only contains the coordinates x and y.

The second one (**Fig.2**) represents a cluster, it contains the **size** (number of points that "belong" to the cluster), a Point **pos** and **old_pos** (that represents the current coordinates of the cluster centroid and the last ones, respectively) and an array **points** with the points that are associated with that cluster.

In terms of variables we decided to use two global variables to avoid passing them as arguments in all functions. As we can see in **Fig.3** the variable **points** its an array of Point's and represents the coordinates of all the **N** samples. The variable **clusters** its an array of Cluster's and gathers the information of all **K** clusters.

III. Optimizations

A. Right Algorithm

The first major decision encountered in carrying out this project was the choice of how to "attack" the algorithm. For that, we made twop completely different versions and proceeded to compare them.

The first consisted on traversing the array of N points once, and at the same time, perform the distance calculations and necessary calculations to discover the position of each centroid. Only the struct Point was used.

On the other hand the second version, traversed the N array and just associated each point to the respective cluster, storing it within the cluster's array of points. Subsequently, the array of points of each cluster was accessed and the position of the new centroid was calculated.

Analyzing **Fig.4** and **Fig.5** we can see the differences between these two algorithms. That said, the selection criterion was the execution time, despite version 1 having a relatively better number of instructions.

B. Euclidean Distance

As we see in **Fig.6** we realized that the main problem of our implementation was in the calculation of the closest cluster to a point. So the first function to be analyzed was the calculation of distances, which presented costly operations such as: **sqrt** and **pow**.

Regarding the **pow** operation the following susbstition was performed: pow(a,2) by a*a.

For **sqrt** it was decided to remove it from the equation since it makes no difference to the problem in question.

These small changes significantly reduced the execution time of the program, as well as the number of instructions and clock cycles, **Fig.7**.

C. Vectorization

In terms of vectorization, we were only able to make the function that calculated the centroids vectorizable, **Fig.8**. The most complicated task here was trying to vectorize the function **cluster_points**, **Fig.9**, which wasn't possible, due to the if statment that was inside the inner loop.

D. Register Variables

Use of register variables, since registers are faster than memory to access. This little change reduced the number of cache load misses in one million and improved the execution time, **Fig.10**.

Acknowledgment

After some attempts, we got to the best execution time in the **Shearch** of 4.0 seconds, **Fig.11**, and 2.3 seconds on our local machine, **Fig.12**.

Working on this area made us understand the difficulty behind code optimization and how important it is to understand the code we write and how we do it because it can have great influences on the final result.

Lastly, making this first project familiarized us with code analysis and optimization techniques, which allowed us to reach these results. However, we hope to improve them in the future.

ATTACHMENTS

```c
typedef struct Point {
  float x;              // x coord
  float y;              // y coord
} Point;
```

Fig. 1. Struct Point

```c
typedef struct Cluster {
  int size;       // cluster size
  Point pos;      // centroid position
  Point old_pos;  // last centroid position
  Point * points; // points that belong to the cluster
} Cluster;
```

Fig. 2. Struct Cluster

```c
Point * points;      // N points coordinates
Cluster * clusters;  // K clusters
```

Fig. 3. Global Variables

```
Performance counter stats for 'bin/k_means':

     20816862907      instructions              #    0.62  insn per cycle
     33825524443      cycles
     20816862907      inst_retired.any          #      1.6 CPI
     33825524434      cycles

    11.115228225 seconds time elapsed

    11.106056000 seconds user
     0.009000000 seconds sys
```

Fig. 4. Version 1

```
Performance counter stats for 'bin/k_means':

     31142516984      instructions              #    1.88  insn per cycle
     16553119478      cycles
     31142516984      inst_retired.any          #      0.5 CPI
     16553119469      cycles

    5.342006807 seconds time elapsed

    5.290043000 seconds user
    0.051000000 seconds sys
```

Fig. 5. Version 2

```
# Samples: 32K of event 'cycles:ppp'
# Event count (approx.): 23286877137
#
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ......................
#
    87.55%  k_means  k_means            [.] cluster_points
     7.69%  k_means  k_means            [.] reevaluate_centers
     1.70%  k_means  libc-2.17.so       [.] __random_r
     1.23%  k_means  libc-2.17.so       [.] __random
     0.59%  k_means  k_means            [.] initialize
     0.51%  k_means  [unknown]          [k] 0xffffffffa1195377
     0.36%  k_means  libc-2.17.so       [.] rand
     0.08%  k_means  k_means            [.] rand@plt
     0.02%  k_means  [unknown]          [k] 0xffffffffa158c48a
     0.01%  k_means  [unknown]          [k] 0xffffffffa158ca5b
     0.01%  k_means  [unknown]          [k] 0xffffffffa158c4ef
     0.01%  k_means  [unknown]          [k] 0xffffffffa158cbc0
```

Fig. 6. Perf report

```
Performance counter stats for './bin/k_means':

     72769379092      instructions              #    2.16  insn per cycle
     33694323860      cycles
     72769379092      inst_retired.any          #      0.5 CPI
     33694323851      cycles

    10.532215606 seconds time elapsed

    10.483884000 seconds user
     0.048004000 seconds sys
```

```
Performance counter stats for './bin/k_means':

     35147246630      instructions              #    2.24  insn per cycle
     15666445170      cycles
     35147246630      inst_retired.any          #      0.4 CPI
     15666445161      cycles

    5.402881672 seconds time elapsed

    5.362752000 seconds user
    0.040005000 seconds sys
```

Fig. 7. Analysis with sqrt and pow(up), and without(down)

```c
void reevaluate_centers() {
    for (int i = 0; i < K; i++) {
        float x = 0;
        float y = 0;

        // vectorizable
        for (int j = 0; j < clusters[i].size; j++) {
            x += clusters[i].points[j].x;
            y += clusters[i].points[j].y;
        }

        clusters[i].pos.x = x / (float) clusters[i].size;
        clusters[i].pos.y = y / (float) clusters[i].size;
    }
}
```

Fig. 8. Reevaluate_centers code

```c
void cluster_points() {
    for (register int i = 0; i < N; i++) {

        register float lowest_dist = dist(points[i],clusters[0].pos);
        register int cluster_id = 0;

        for (register int j = 1; j < K; j++) {
            register float distance = dist(points[i],clusters[j].pos);

            if (distance < lowest_dist) {
                lowest_dist = distance;
                cluster_id = j;
            }
        }

        clusters[cluster_id].points[clusters[cluster_id].size] = points[i];
        clusters[cluster_id].size++;
    }
}
```

Fig. 9. Reevaluate_centers code

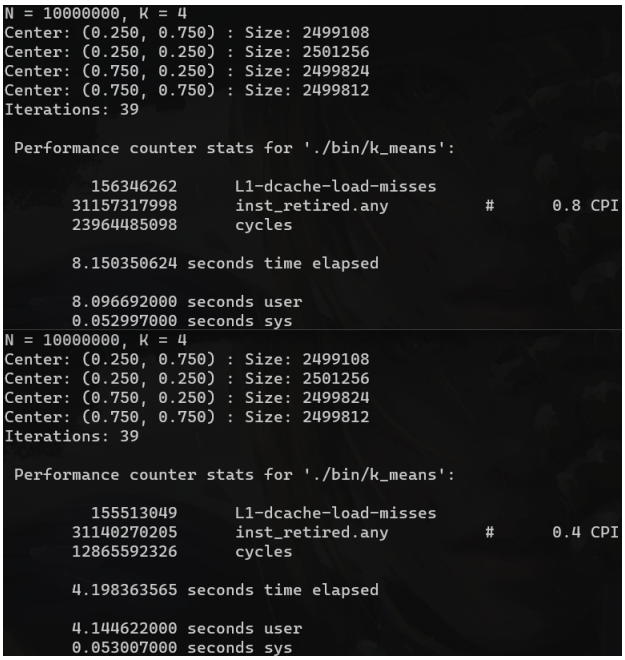

Fig. 10. Analysis without register variables(up), and with(down)

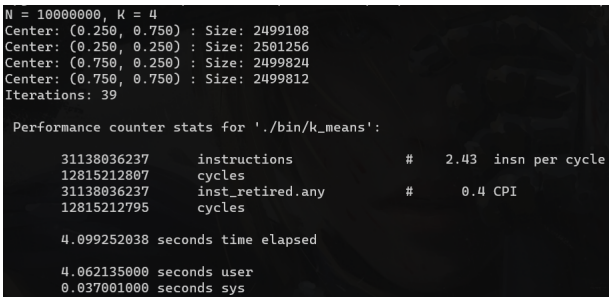

Fig. 11. Best result Shearch

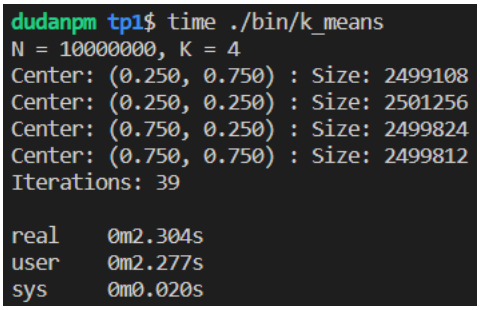

Fig. 12. Best result local machine