

K-Means Clustering

1st Cristiano Pereira
Software Engineering Student
Universidade do Minho
Braga, Portugal
pg50304@alunos.uminho.pt

2nd Rúben Santos
Software Engineering Student
Universidade do Minho
Braga, Portugal
pg50733@alunos.uminho.pt

Abstract—This document serves as a report to the first practical assignment of the Parallel Computing Course given at Universidade do Minho. Here we can find an analysis of the k-means algorithm, based on the Lloyd’s algorithm. We start by creating a sequential version of it (written in the C programming language) and then proceed to optimize the code.

Index Terms—parallel, computing, k-means, lloyd’s, algorithm

I. WORK ENVIRONMENT

In this project we used GCC version 4.8.5 to compile the code and perf version 3.10.0-1160.41.1.el7 for benchmarking. The flags -O2 and -std=c99 were used throughout the tests.

All the results presented in this document were obtained on the computational nodes of the SeARCH cluster of the Informatics Department at Universidade do Minho. To be more precise, the code was executed on the *cpar* partition, which has an Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz with 20 cores and 64GB of RAM.

II. K-MEANS ALGORITHM

In this section we present a pseudocode version of the k-means algorithm. The number of points and clusters is given by the variables N and K , respectively. The algorithm can be represented as follows:

Algorithm 1 Pseudocode for the K-Means algorithm.

- 1: Generate N random points.
 - 2: Set each Cluster Center as the first K points.
 - 3: Assign each point to the nearest Cluster.
 - 4: **repeat**
 - 5: Calculate the Centroid of each Cluster.
 - 6: Assign each point to the nearest Cluster.
 - 7: **until** No points are assigned to a different Cluster.
-

In order to ensure reproducibility, the random numbers are generated following the professors’ recommendations.

III. SEQUENTIAL VERSION

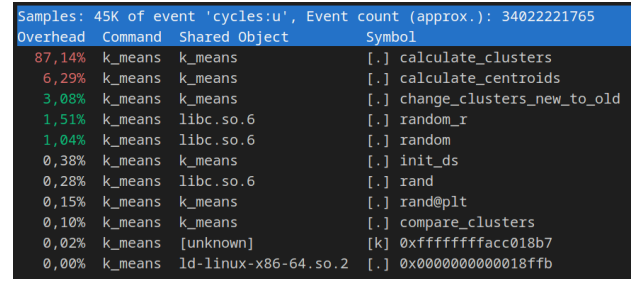
The code developed in the first part of this assignment is meant to be executed sequentially. We develop a first iteration of the algorithm and then we analyse the code with (*Perf*). This tool allows us to determine which section of the code is consuming more resources. We can also get information about the execution time, number of instructions, number of cycles and cache misses.

A. Initial Code

For the first version of the algorithm, we didn’t specifically focus on performance, meaning that we didn’t use any profiling tools to guide the development.

B. Perf Report

In the image below we can see the result of the *perf report*.



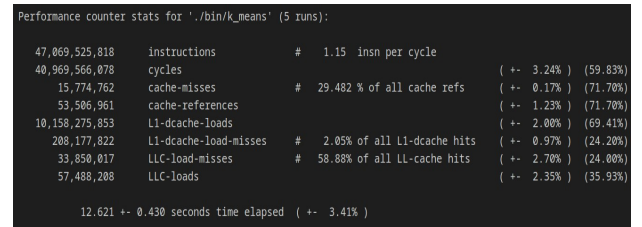
| Overhead | Command | Shared Object | Symbol |
|----------|---------|----------------------|--------------------------------|
| 87.14% | k_means | k_means | [.] calculate_clusters |
| 6.29% | k_means | k_means | [.] calculate_centroids |
| 3.08% | k_means | k_means | [.] change_clusters_new_to_old |
| 1.51% | k_means | libc.so.6 | [.] random_r |
| 1.04% | k_means | libc.so.6 | [.] random |
| 0.38% | k_means | k_means | [.] init_ds |
| 0.28% | k_means | libc.so.6 | [.] rand |
| 0.15% | k_means | k_means | [.] rand@plt |
| 0.10% | k_means | k_means | [.] compare_clusters |
| 0.02% | k_means | [unknown] | [k] 0xfffffffffacc018b7 |
| 0.00% | k_means | ld-linux-x86-64.so.2 | [.] 0x0000000000018ffb |

Fig. 1. Perf report of the initial code.

We can see that the first two functions (*calculate_clusters* and *calculate_centroids*) are the ones that can easily impact performance the most. In section IV we will focus on improving these and other functions that do not appear in figure 1.

C. Perf Stat

In order to get more statistics from our first algorithm, we used the *perf stat* command. The code was run five times with the flag -r 5. The execution time was, on average, 12.6 seconds.



| Performance counter stats for './bin/k_means' (5 runs): | | | | |
|---|-----------------------|---|-----------------------------|-----------------------|
| 47,069,525,818 | instructions | # | 1.15 insns per cycle | |
| 40,969,566,078 | cycles | | | (+- 3.24%) (59.83%) |
| 15,774,762 | cache-misses | # | 29.482 % of all cache refs | (+- 0.17%) (71.70%) |
| 53,506,961 | cache-references | | | (+- 1.23%) (71.70%) |
| 10,158,275,853 | L1-dcache-loads | | | (+- 2.00%) (69.41%) |
| 208,177,822 | L1-dcache-load-misses | # | 2.05% of all L1-dcache hits | (+- 0.97%) (24.20%) |
| 33,850,017 | LLC-load-misses | # | 58.88% of all L1-cache hits | (+- 2.70%) (24.00%) |
| 57,488,208 | LLC-loads | | | (+- 2.35%) (35.93%) |
| 12.621 +- 0.430 seconds time elapsed (+- 3.41%) | | | | |

Fig. 2. Perf statistics of the initial code.

We can see in figure 2 that we have, on average, more than one instruction being calculated per cycle. On average, $\approx 30\%$ of all cache references miss. There are not many L1

cache misses ($\approx 2\%$) compared to the last level cache misses ($\approx 59\%$). This is obviously a problem, since every last level cache miss will imply accessing the disk, which will most certainly degrade the performance of the algorithm.

We decided to also run the *perf record* command to monitor LLC misses. The results are shown in figure 3.

Samples: 20K of event 'LLC-loads-misses:u', Event count (approx.): 2148419

| Overhead | Command | Shared Object | Symbol |
|----------|---------|---------------|--------------------------------|
| 57.46% | k_means | k_means | [.] change_clusters_new_to_old |
| 25.52% | k_means | k_means | [.] calculate_clusters |
| 15.50% | k_means | k_means | [.] calculate_centroids |
| 0.99% | k_means | k_means | [.] compare_clusters |
| 0.17% | k_means | libc.so.6 | [.] random |

Fig. 3. Perf report for LLC misses.

Most of the misses are happening in the *change_clusters_new_to_old* function. This function is only used for around 3% of the time but is responsible for more than 50% LLC misses. In the next section we discuss the options to solve this problem.

IV. IMPROVED SEQUENTIAL VERSION

In this section we will report the results of the described optimizations. Each of these next steps is done on top of the previous one. The last optimization also represents the most optimized version of the code.

A. Square Root Removal

The function calculating the euclidian distance was using the square root operator. This operator is not required since, if $A \geq B$ then $A^2 \geq B^2$, assuming that $A, B \geq 0$.

Performance counter stats for './bin/k_means_no_square_root' (5 runs):

| | | | | | | |
|---|-----------------------|---|------------------------------|----------------|----------|--|
| 37,050,517,194 | instructions | # | 1.14 | insn per cycle | | |
| 32,635,490,319 | cycles | | | (+/- 8.27%) | (62.50%) | |
| 13,584,850 | cache-misses | # | 30.022 % of all cache refs | (+/- 5.63%) | (75.00%) | |
| 45,249,433 | cache-references | | (+/- 9.41%) | (75.00%) | | |
| 7,158,322,773 | L1-dcache-loads | | (+/- 0.11%) | (74.95%) | | |
| 205,806,955 | L1-dcache-load-misses | # | 2.88% of all L1-dcache hits | (+/- 0.40%) | (25.00%) | |
| 27,779,631 | LLC-load-misses | # | 58.51% of all LLC-cache hits | (+/- 12.88%) | (25.00%) | |
| 47,478,971 | LLC-loads | | (+/- 12.68%) | (37.49%) | | |
| 10.553 +/- 0.723 seconds time elapsed (+/- 6.85%) | | | | | | |

Fig. 4. Perf stat without the square root.

As we can see in figure 4, with this optimization we gained around 2.1 seconds of execution time. Meanwhile, the percentage of cache misses remains the same.

B. Data Structures

Another major improvement was to rethink the data structures we were using. We started with an array of size $2N$ (being N the number of points i.e., 10,000,000) that stored the x coordinate of the first point at index 0 and the respective y coordinate at index 1. This is obviously not optimal in comparison to having two separate arrays, of size N , each one representing the x and y coordinates, as shown in the listing below.

```
1 struct{
2     float *x;
3     float *y;
4 } typedef points;
```

We would expect a reduction in cache misses, especially the LLC misses, but unfortunately this didn't happen. The L1 cache misses did actually halved and we gained around 2.4 seconds in execution time, as shown in figure 5.

Performance counter stats for './bin/k_means_merge' (5 runs):

| | | | | | | |
|--|-----------------------|---|------------------------------|----------------|----------|--|
| 28,085,410,744 | instructions | # | 1.09 | insn per cycle | | |
| 25,791,014,823 | cycles | | | (+/- 0.81%) | (62.50%) | |
| 7,372,684 | cache-misses | # | 61.200 % of all cache refs | (+/- 0.64%) | (75.00%) | |
| 12,046,789 | cache-references | | (+/- 0.46%) | (75.00%) | | |
| 6,414,724,007 | L1-dcache-loads | | (+/- 0.83%) | (74.93%) | | |
| 115,051,977 | L1-dcache-load-misses | # | 1.79% of all L1-dcache hits | (+/- 0.20%) | (25.00%) | |
| 4,840,468 | LLC-load-misses | # | 64.50% of all LLC-cache hits | (+/- 2.25%) | (24.99%) | |
| 7,584,756 | LLC-loads | | (+/- 1.71%) | (37.49%) | | |
| 8.1687 +/- 0.0789 seconds time elapsed (+/- 0.97%) | | | | | | |

Fig. 5. Perf stat with improved data structures.

C. Loop Unrolling

We also thought of loop unrolling in order to reduce the number of instructions that the program will have, because there will be fewer loop related instructions. So we used the GCC flag *-funroll-loops* to help us apply this optimization.

Performance counter stats for './bin/k_means_new_ds' (5 runs):

| | | | | | | |
|--|-----------------------|---|------------------------------|----------------|----------|--|
| 28,086,134,539 | instructions | # | 1.09 | insn per cycle | | |
| 25,700,525,973 | cycles | | | (+/- 0.03%) | (62.49%) | |
| 7,371,583 | cache-misses | # | 62.268 % of all cache refs | (+/- 0.68%) | (75.00%) | |
| 11,838,522 | cache-references | | (+/- 0.63%) | (75.01%) | | |
| 6,419,386,784 | L1-dcache-loads | | (+/- 0.01%) | (74.93%) | | |
| 115,651,465 | L1-dcache-load-misses | # | 1.80% of all L1-dcache hits | (+/- 0.20%) | (24.99%) | |
| 4,934,978 | LLC-load-misses | # | 64.07% of all LLC-cache hits | (+/- 2.23%) | (24.99%) | |
| 7,702,478 | LLC-loads | | (+/- 1.71%) | (37.48%) | | |
| 7.9467 +/- 0.0921 seconds time elapsed (+/- 1.16%) | | | | | | |

Fig. 6. Perf stat with loop unroll flag (-funroll-loops).

As we can see in figure 6, the execution time improved around 0.2s. In previous runs we were able to gain around 1s with this flag, but we were unable to get the same results.

V. CONCLUSION

To conclude this report, we would like to express our frustration about not being able to reduce significantly the execution time of the code. We tried several other optimizations, such as using the -O3 and -free-vectorize flags. None of these flags produced a better execution time, as a matter of fact, the performance degraded.

On the other hand, there are lots of opportunities to optimize the code using parallel programming techniques. We ensured that all the loops can be executed in parallel and the code remains generic enough for modifications.

REFERENCES

- [1] "Cluster Nodes", UMinho Informatics Department, http://search6.di.uminho.pt/wordpress/?page_id=55
- [2] "K-Means Clustering", Wikipedia, https://en.wikipedia.org/wiki/K-means_clustering
- [3] John L. Hennessy and David A. Patterson. 2017. Computer Architecture, Sixth Edition: A Quantitative Approach (6th. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.