

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 third and final 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. On the first assignment we created a sequential version of the algorithm, then we explored parallelism optimizations using OpenMP and now we will do more in depth analysis of the parallelism optimization.

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

I. WORK ENVIRONMENT

In this project, we used GCC version 7.2.0 to compile the code, perf version 3.10.0-1160.41.1.el7 for benchmarking, and OpenMP 4.5 for the parallelism constructs. The flags -O2, -std=c99, -g, -fno-omit-frame-pointer and -funroll-loops were used throughout the tests.

The results presented in this document were obtained on a computer with an AMD Ryzen™ 7 3700U @ 2.3GHz CPU with 4 cores and 12GB of RAM, as well as on the computational nodes of the SeARCH cluster of the Informatics Department at Universidade do Minho. To be more specific, 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. BRIEF INTRODUCTION TO OPENMP

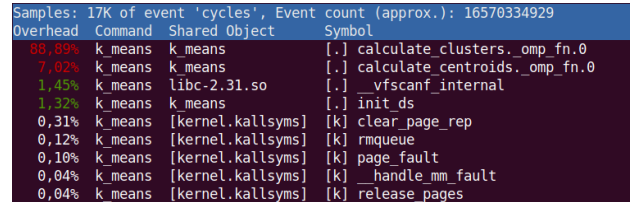
In order to implement parallelization in the code, we used OpenMP, which is an API that supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer [4].

To apply it in C, we used pragmas such as `#pragma omp parallel` and added the flag `-fopenmp` when compiling the code to allow the compiler to optimize it for OpenMP implementation.

IV. PARALLELISM OPPORTUNITIES

A. Identify the most expensive blocks of code

As seen in figure 1, most of the computing power is being spent at `calculate_clusters`. This function is responsible for calculating the distance from each point to the cluster center. The `calculate_centroids` function also takes up a bit of computation.



Samples: 17K of event 'cycles', Event count (approx.): 16570334929			
Overhead	Command	Shared Object	Symbol
88.89%	k_means	k_means	[.] calculate_clusters_omp_fn.0
7.02%	k_means	k_means	[.] calculate_centroids_omp_fn.0
1.45%	k_means	libc-2.31.so	[.] __vfprintf_internal
1.32%	k_means	k_means	[.] init_ds
0.31%	k_means	[kernel.kallsyms]	[k] clear_page_rep
0.12%	k_means	[kernel.kallsyms]	[k] rmqueue
0.10%	k_means	[kernel.kallsyms]	[k] page_fault
0.04%	k_means	[kernel.kallsyms]	[k] __handle_mm_fault
0.04%	k_means	[kernel.kallsyms]	[k] release_pages

Fig. 1. Perf report for the sequential version with 4 clusters.

B. Analyze parallelism alternatives

1) `calculate_clusters`: In this function, the OpenMP implementation is more straightforward since no different iterations can change the same value. We only need to apply the `pragma omp parallel for`.

2) `calculate_centroids`: Here, we have two loops. The first one is a bit more complex to implement, while in the second one, we can use the same approach as in the `calculate_clusters` function.

In the first loop, when we try to sum all the (x,y) values, we encounter a problem where each iteration can change values that other iterations are also changing. To solve this, and since we have different sums happening in each iteration, we create copies of the three arrays for each thread (which

is not memory-intensive, as the size depends on the number of clusters). Each thread holds the values of the iterations that it performs, and later, after all the threads have finished, we update the main arrays with the sum of these copies. To implement this with OpenMP, we can use the reduction directive for each array we want to create copies of.

3) *General*: In general, we used static scheduling for all the situations where parallelism was used. This is justified as the load of the computations does not increase proportionally with the iterator variable value, and therefore, there is no need for the extra overhead to dynamically schedule the threads.

V. MEASUREMENT METHOD

In order to conduct all types of measurements, we used the Perf tool to gather all of the data. Since it is not possible to run *perf record* on the SeARCH cluster, we used the computer mentioned in I to obtain the execution profile of the implementation. As for the various metrics such as clock cycles, execution time, and cache misses, we used the SeARCH cluster to gather all the required values.

For the different metrics, the perf stat command was used. We hadded different flags,

- **-r**, to choose the number of times Perf runs the program, for example, using *-r 10* runs the program 10 times and gives the mean values.
- **-e**, To select which metrics Perf will display in the results, for example, using *-e cache-misses* will show the cache misses when running the program.

VI. PERFORMANCE ANALYSIS

A. Execution Profile

Here, we can check the computational workload of the program after parallelization. It is possible to see that the *calculate_clusters* function is still taking up most resources, but at a lower value, while the *calculate_centroids* function has increased its usage. It is also possible to see the appearance of OpenMP workload.

Overhead	Command	Shared Object	Symbol
83.91%	k_means	k_means	[.] calculate_clusters_omp_fn.0
10.10%	k_means	k_means	[.] calculate_centroids_omp_fn.0
1.41%	k_means	libc-2.31.so	[.] __vfprintf_internal
1.28%	k_means	k_means	[.] init_ds
0.72%	k_means	libgomp.so.1.0.0	[.] omp_get_num_procs
0.32%	k_means	[kernel.kallsyms]	[k] clear_page_rep
0.17%	k_means	[kernel.kallsyms]	[k] page_fault
0.17%	k_means	[kernel.kallsyms]	[k] __default_send_IPI_dest_field
0.17%	k_means	[kernel.kallsyms]	[k] smp_call_function_many
0.15%	k_means	[kernel.kallsyms]	[k] call_function_interrupt
0.09%	k_means	[kernel.kallsyms]	[k] rmqueue
0.06%	k_means	[kernel.kallsyms]	[k] free_pcppages_bulk
0.05%	k_means	[kernel.kallsyms]	[k] native_flush_tlb_one_user
0.05%	k_means	[kernel.kallsyms]	[k] zap_pte_range.isra.0

Fig. 2. Perf report for 4 threads and 4 clusters.

In Figure 3, we have the computational workload for the same number of clusters but with a larger number of threads. By examining this, we can see the OpenMP workload increase and the load reduction on the *calculate_centroids* function.

This indicates that at some point, it will not be worth increasing the number of threads, as the work to use more threads will be greater than the gain from using more of them.

Overhead	Command	Shared Object	Symbol
79.92%	k_means	k_means	[.] calculate_clusters_omp_fn.0
11.44%	k_means	k_means	[.] calculate_centroids_omp_fn.0
3.38%	k_means	libgomp.so.1.0.0	[.] omp_get_num_procs
1.25%	k_means	libc-2.31.so	[.] __vfprintf_internal
1.07%	k_means	k_means	[.] init_ds
0.40%	k_means	[kernel.kallsyms]	[k] clear_page_rep
0.24%	k_means	[kernel.kallsyms]	[k] call_function_interrupt
0.17%	k_means	[kernel.kallsyms]	[k] smp_call_function_many
0.13%	k_means	[kernel.kallsyms]	[k] __default_send_IPI_dest_field
0.11%	k_means	[kernel.kallsyms]	[k] page_fault
0.06%	k_means	[kernel.kallsyms]	[k] flush_smp_call_function_queue
0.05%	k_means	[kernel.kallsyms]	[k] native_flush_tlb_one_user
0.04%	k_means	[kernel.kallsyms]	[k] rmqueue

Fig. 3. Perf report for 8 threads and 4 clusters.

As a final note in the profile analysis, we were unable to create samples for a larger number of threads since the CPU we used has a maximum of 8 threads. Therefore, next, we will analyze this scalability using the SeARCH cluster.

B. Scalability Analysis for 10 Million Points

In order to get a first idea of the scalability of the implementation, we can see in Figure 4 how the execution time changes with the increase of threads in use. We also used different numbers of clusters to check if the scalability changes with an increase in the number of clusters.

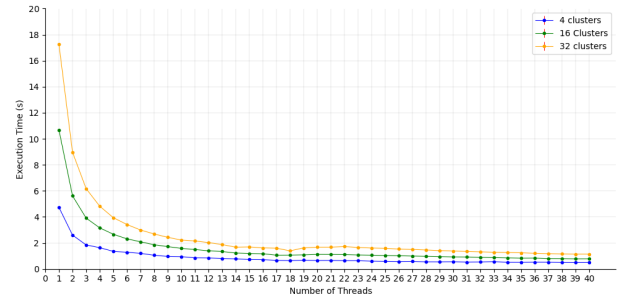


Fig. 4. Execution time variation with the number of threads and clusters: 32 clusters (yellow), 16 clusters (green), and 4 clusters (blue).

It is possible to see an immediate reduction in execution time as soon as we start to use additional threads, but it is also possible to verify that the more threads we use, the less we will gain with the addition of another thread. We cannot get much more information from this figure, but with these values, we can gain another perspective through the calculation of speedup, as Figure 5 illustrates.

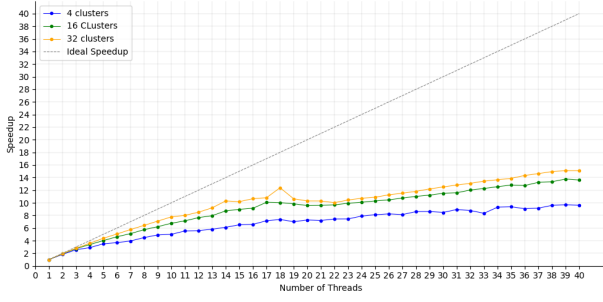


Fig. 5. Speedup for introducing respective thread: 4 clusters (blue), 16 clusters (green), and 32 clusters (yellow).

Ideally, we would like to decrease the execution time in proportion to the number of threads we increase, but as the previous figure shows, that is not the case. The more threads we use, the less we gain from it, moving further and further away from the ideal speedup (something that was already visible in Figure 4). It is also possible to verify that there is a greater speedup the more clusters we are using.

The loss in gain from adding threads increases as we increase the number of threads, which may be related to the overhead from OpenMP to manage the parallelization. To verify this assertion, we will examine the number of clock cycles as we add more threads.

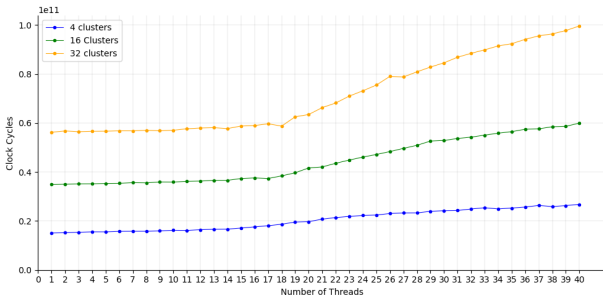


Fig. 6. Clock cycle variation per thread: 4 clusters (blue), 16 clusters (green), and 32 clusters (yellow).

Ideally, we should have about the same number of clock cycles regardless of how many threads we use, but as Figure 6 shows, that is not the case in practice. This confirms that the overhead of parallelization is what makes it not worthwhile to add more threads at a certain point, as we previously discussed in Section VI-A where we observed an increase in OpenMP related workload as we increased the number of threads.

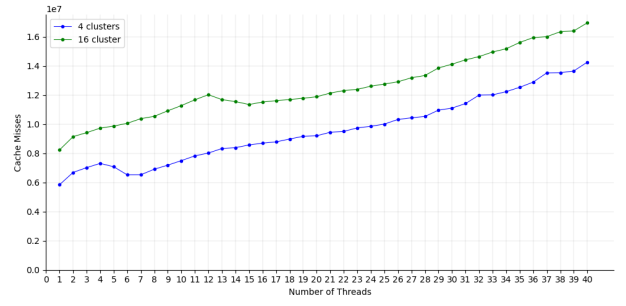


Fig. 7. Cache misses variation per thread: 4 clusters (blue), and 16 clusters (green).

As shown in Figure 7, it is possible to see that cache misses increase as the number of threads in use increases. This is due to the OpenMP overhead that leaves less space in the cache for the values of the program.

To conclude this section, we can see that until 40 threads, the execution time will decrease, but after around 18 threads, the gain is so small that it may not be worth the additional resources required to add more threads.

C. Scalability Analysis for Different Numbers of Points

In order to see the impact of the L1, L2, and L3 caches on the scalability of the implementation, we will change the number of points used in the algorithm.

Regarding the L1 cache, the CPU in use has a size of 64 KB and, since each point is represented by two floats (4 bytes each), we would need approximately 8192 points to fill the entire cache.

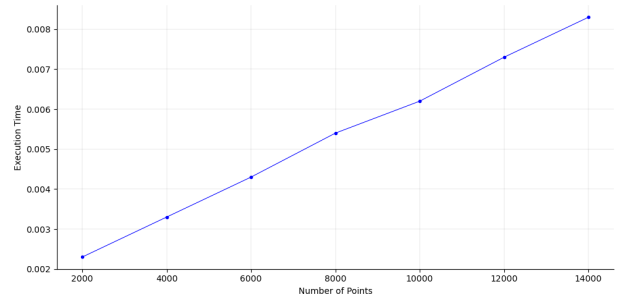


Fig. 8. Execution time per number of points.

In Figure 3, we can see that as the number of points approaches and exceeds the maximum size of the L1 cache, the execution time increases in a linear way with the increase in the number of points.

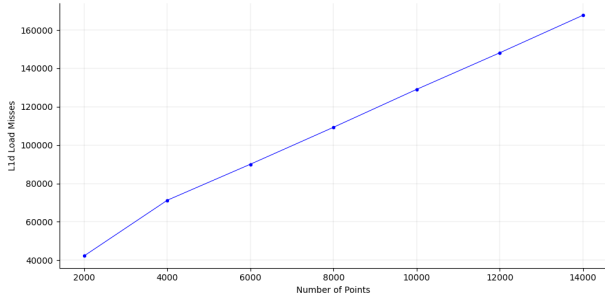


Fig. 9. L1d cache misses variation per number of points.

In order to measure the amount of L1 cache misses, we used Perf to measure L1d Load Cache misses. This metric captures all misses that occur at the L1 cache, including multiple attempts to fetch before the instructions have been returned, so it does not provide a precise number of data reads. In Figure 9, we can see that even though we used a number of points below and above the L1 cache maximum size, the number of L1d Load Misses that we obtained increased in a linear way with the number of points.

In Figure 10, we can see how the execution time varies as the number of points increases from below to above the maximum size of the L2 cache. The CPU in use has an L2 cache size of 542KB, so in order to fill the entire cache, we would need approximately 65536 points, with each point taking up 8 bytes.

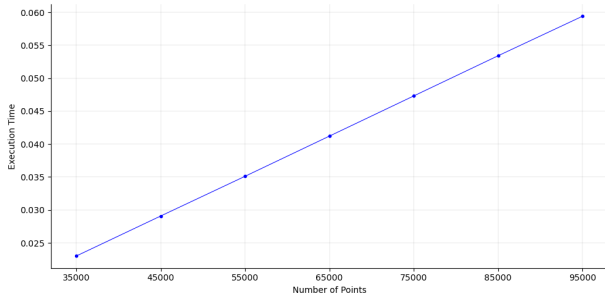


Fig. 10. Execution time per number of points.

The execution time increases linearly as the number of points increases, as shown in the preceding figure. To gain more insight, we should check if the L2 cache misses provide additional information. Although Perf is unable to directly measure L2 cache misses, it is possible to observe how general cache misses vary as the number of points increases around the point of reference for the maximum size of the L2 cache.

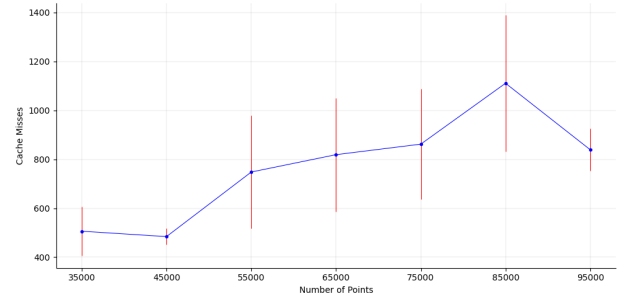


Fig. 11. Cache misses variation per number of points.

From Figure 11, it is immediately apparent that this measure has a high error bar, making it difficult to accurately count the number of L2 cache misses that occurred during the program's execution and to observe how cache misses vary with the increase in points around the point of reference for the maximum size of the L2 cache. However, it is possible to see that in general, the number of cache misses increases as the number of points increases, which is to be expected.

Regarding the L3 cache, the CPU in use has a size of 10236 KB, and as we previously discussed, each point is represented by two floats, taking up 8 bytes. Therefore, we would need approximately 1310208 points to fill the entire cache. In Figure 12, we can observe how the execution time changes as we increase the number of points around the amount needed to fill the cache.

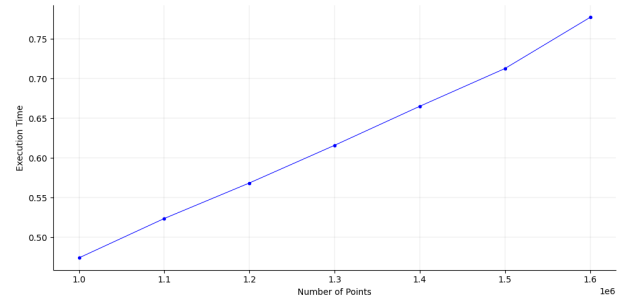


Fig. 12. Execution time per number of points.

In the previous figure, it is possible to see that the execution time increases in a linear way, even though we are providing points around the maximum size of the L3 cache. To gain a deeper understanding of this, we should examine how the number of cache misses in this cache varies.

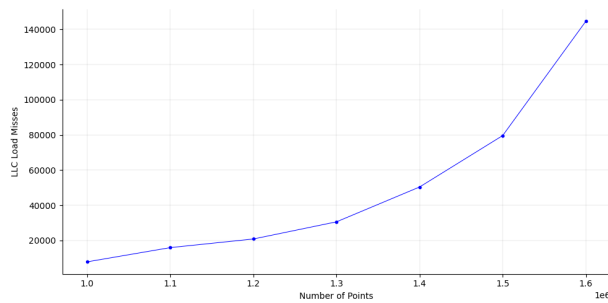


Fig. 13. LLC cache misses variation per number of points.

In Figure 13, it is important to note that LLC (Last Level Cache) refers to the L3 cache as it is the last level of cache of the CPU we are using. It is also apparent that around 1300000 points, which is approximately the size of the L3 cache, the number of cache misses begins to significantly increase. This is understandable as after this point, it is not possible to cache all the points and we have to constantly go to the RAM to retrieve point values.

In general, it was observed that for the different points of reference for cache size, the execution time varied linearly with the increase in the number of points used by the program. As for the different types of cache misses, the behavior of L3 cache misses was the most distinct. Instead of increasing linearly, it increased in a logarithmic way.

VII. CONCLUSION

In this third phase, our initial focus was on developing a solution that solved this problem using the GPU with CUDA. Our effort helped us understand some of the principles of CUDA and its importance in the parallel computing realm. Unfortunately we were not able to solve some of the bugs that surfaced from the use of CUDA and also the results of the clustering calculation were wrong. We had to go back to the original OpenMP implementation developed in the previous phase which already was exploiting the parallel computing power of the CPU in a fairly reasonable way. Anyway, the CUDA code is also provided with the rest of the solution.

With respects to the study and analysis of the working solution, we focused on the continuous increase of the number of points and the impact that it has on cache misses (across all levels) and the overall performance of the system. We also offer a view on the influence that the excessive use of threads may have in the cache misses.

In conclusion, we end the practical component of *Computação Paralela* (Parallel Computing) with a vast array of concepts that will definitely help us think, from the ground up, on how to build parallel applications and also, how to measure their performance.

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.
- [4] "OpenMP", <https://www.openmp.org/>
- [5] "Notes on the mystery of hardware cache performance counters", <https://sites.utexas.edu/jdm4372/2013/07/14/notes-on-the-mystery-of-hardware-cache-performance-counters/>