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 second 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, now we will explore parallelism optimizations using OpenMP.**

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

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 and -std=c99 were used throughout the tests. The number of points is constant (10,000,000).

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.

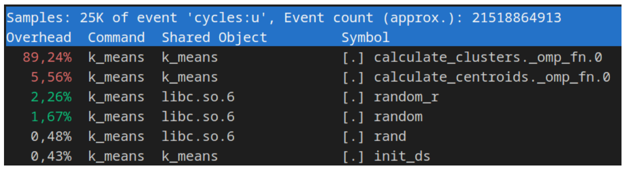
```
1: Generate N random points.
2: Set each Cluster Center as the first K points.
3: Assign each point to the nearest Cluster.
4: repeat
5:     Calculate the Centroid of each Cluster.
6:     Assign each point to the nearest Cluster.
7: until No points are assigned to a different Cluster.
```

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

III. Parallelism Opportunities

A. Indentify 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, especially when the number of clusters increases (Section VI. Appendix).



```
Samples: 25K of event 'cycles:u', Event count (approx.): 21518864913
Overhead  Command  Shared Object         Symbol
  89,24%  k_means  k_means               [.] calculate_clusters._omp_fn.0
   5,56%  k_means  k_means               [.] calculate_centroids._omp_fn.0
   2,26%  k_means  libc.so.6             [.] random_r
   1,67%  k_means  libc.so.6             [.] random
   0,48%  k_means  libc.so.6             [.] rand
   0,43%  k_means  k_means               [.] init_ds
```

Fig. 1. Perf report for 1 thread and 8 clusters.

B. Analyze parallelism alternatives

1) calculate_clusters: In this function, there was a dependency that made the parallel code slow and unreadable. We were updating the size of a given cluster at each iteration. This meant that, each thread, would have to update the same array, possibly the same value, leading to synchronization problems. The solution was to postpone this cluster size update to the *calculate_centroids* function.

2) calculate_centroids: Here the parallelism is a bit more complex. We have two loops: the first updates each cluster size and gets the sum of the (x,y) coordinates of all points; the second has no dependencies. In order to sum all the (x,y) values of each point we come to the same problem as before (we also update the cluster size here). To solve this, we do a copy of the three arrays per each thread (which is not memory expensive since the size depends on the number of cluster). This copy will hold the values of the iterations it performs. Later there is a critical section where each thread, at a time, updates the values of the shared variables.

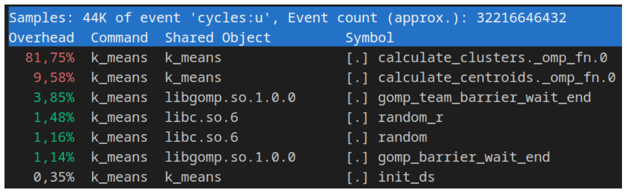
In the second loop, there is no obvious gain in using parallelism. There is too much overhead of managing the threads to run only for *n_cluster* iterations. If this value tends to increase in the future, this is a place where parallelism might come in handy. Since there are no dependencies, one could just insert the *pragma* command.

3) General: In general, we used the *pragma static scheduling* command. This is justified in both the situations that parallelism was used. Since the load of the computations does not increase proportionally with the iterator variable value, there is no need for the extra overhead to dynamically schedule the threads.

IV. Performance Analysis

A. Execution Profile

Here we can check the computational load of the program post-parallelization. The *calculate_clusters* function is still taking up most resources but the *calculate_centroids* doubled it's usage. This is due to the modifications we did (and explained previously at section III-B) in order to parallelize the algorithm. See section VI (appendix) for more.



```
Samples: 44K of event 'cycles:u', Event count (approx.): 32216646432
Overhead  Command  Shared Object       Symbol
  81,75%  k_means  k_means             [.] calculate_clusters._omp_fn.0
   9,58%  k_means  k_means             [.] calculate_centroids._omp_fn.0
   3,85%  k_means  libgomp.so.1.0.0    [.] gomp_team_barrier_wait_end
   1,48%  k_means  libc.so.6           [.] random_r
   1,16%  k_means  libc.so.6           [.] random
   1,14%  k_means  libgomp.so.1.0.0    [.] gomp_barrier_wait_end
   0,35%  k_means  k_means             [.] init_ds
```

Fig. 2. Perf report for 8 threads and 8 clusters.

B. Scalability Analysis

In the graph bellow we can see the variation of the execution time when using different number of threads. There is an immediate reduction in execution time when the second thread is introduced. The gain in introducing more threads fades away quickly though.

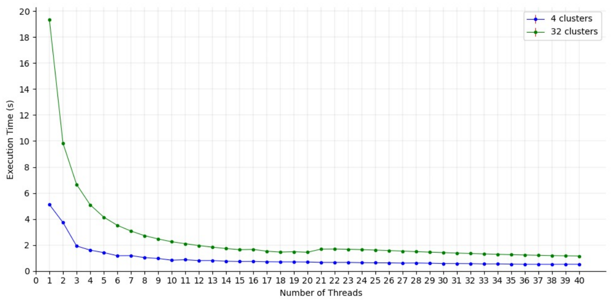


Fig. 3. Execution time variation with number of threads and clusters (32 clusters (green) and 4 clusters (blue)).

Another interesting prespective is the gain introduced by adding more threads. In the graph bellow we can see this metric. The ideal gain is presented in gray. Ideally, we would like to decrease the execution time in the same proportion we increase the number of threads.

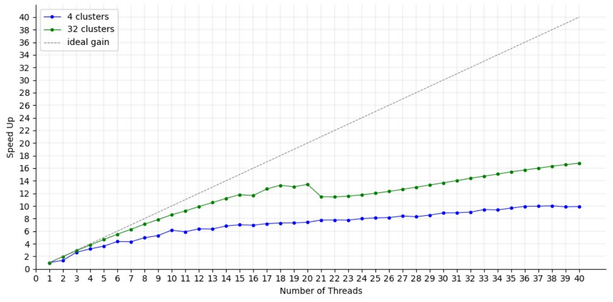


Fig. 4. Gain for introducing respective thread (4 clusters (blue), 32 clusters (green)).

Introducing parallelism brings overhead. We can see in the following graph the variation of the number of clock cycles in relation to the number of threads. We can see here that there is an obvious overhead in having a higher number of threads since that, ideally, we should have about the same number of clock cycles.

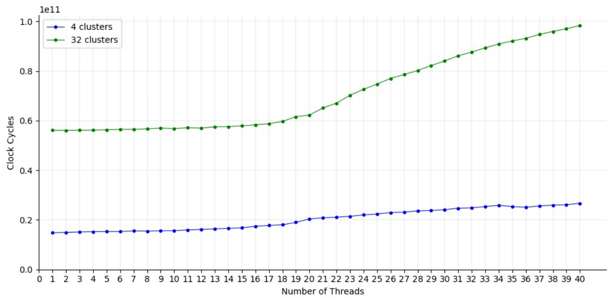


Fig. 5. Clock cycle variation per thread (4 clusters (blue), 32 clusters (green)).

One of the most important metrics is cache misses. Here we can see a small increase in cache misses when the number of threads increases.

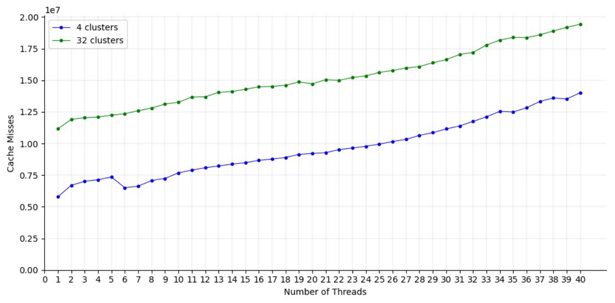


Fig. 6. Cache misses evolution per thread (4 clusters (blue), 32 clusters (green)).

The ammount of work that is being done sequentially is insignificant, that is because the most time consuming functions are being computed in parallel.

To conclude this section, the optimal number of threads to use in search, according with our tests is around 20. This number guarantees a good balance between the ammount of computation being done and the execution time.

V. Conclusion

In conclusion, we felt that the first implementation of the code was not hard to adapt to this parallel version. We were able to understand and implement several of the most important OpenMP primitives.

References

[1] "Cluster Nodes", UMinho Informatics Department, http://search6.di.uminho.pt/wordpress/?page_id=55
[2] "K-Means Custering", 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.

VI. Appendix

The number of clusters used also has an impact in the execution time. Here we show the variation of the execution time in relation to the number of clusters.

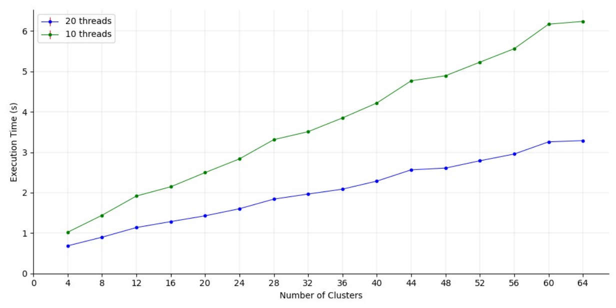


Fig. 7. Execution time variation with number of clusters and threads (10 threads (green) and 20 threads (blue)).