Parallel Computing K-means Algorithm

Rui Monteiro
Informatics Department
University of Minho
Braga, Portugal
pg50739@alunos.uminho.pt

Diogo Barbosa
Informatics Department
University of Minho
Braga, Portugal
pg50326@alunos.uminho.pt

***Abstract*—Given a K-means parallel algorithm previously developed, which distributes N samples iteratively, using the lowest distance as the deciding factor, between a certain number of clusters until said clusters are verified to have converged, we will implement a new version utilizing MPI keeping scalability in mind by making several performance tests, using the number of samples, the number of clusters and the number of processes as metrics and interpreting these results. On the other hand, we will analyse cache misses and appraise the results obtained, considering the program's memory hierarchy.**

***Index Terms*—K-means, MPI, metrics, parallel, algorithm, performance, processes, dependencies, critical zone, memory, cache, cores, message, rank**

I. Introduction

The objective of this project is to implement a Message Passing Interface approach in a previously developed K-means clustering algorithm and evaluating the benefits and the downsides, dealing with data dependencies, race conditions and critical zones. In addition, several tests were performed to analyze which number of processes would have the best performance considering our use case and to get a greater understanding of the overall benefit of implementing MPI to the initial parallel algorithm that used OpenMP.

II. Implementation with OpenMP

After completing the second assignment, the result was very pleasing in terms of performance, which is why the decision was made to not further improve our parallel implementation of the K-means algorithm using OpenMP and to instead explore a new implementation using a Message Passing Interface as our new approach.

III. Implementation with MPI

A. Pipeline approach

This was our first approach to producing an implementation using MPI to achieve a parallel version of the K-means algorithm. In our sequential implementation, the primary way of gaining performance was to reduce the number of cycles in N and the dispensable work within its body. To take advantage of a pipeline we tried to divide the work among several cycles in N, with the objective of connecting them together to form a pipeline. This pipeline would communicate with each other with *MPI_SEND* and *MPI_RECV*.

We soon noticed that this version performed very poorly against its sequential counterpart for two main reasons.

Firstly, we concluded that no matter how we try to divide the work between the various sections of the pipeline (of which each one was a process) since there was a lot of time wasted on the message passing between processes.

Secondly, after several attempts to solve the problem, we noticed that this way of solving the problem will never produce good results since the K-means algorithm requires data from the end of the previous iteration to start the next one. After all, we had just created an over-complicated and bloated version of the sequential one, which had a terrible performance and was therefore abandoned.

B. Reduce approach

Since pipelining our code was not a viable solution, we turned to what worked for us previously with OpenMP, using a reduce, in which every process was responsible for a section of the input N and then merges on the root process for the critical section.

Given an S size (number of processes), the single cycle in N was divided into *N/S* elements for each process and then the root process makes a reduction of the result (Summation in this case). After this, the new centroids are calculated and then broadcasted to all the other processes, so they can perform the next iteration.

With this approach, we practically just made an MPI version of the previous OpenMP code, in which the cycle in N was divided into S sections and processed at the same time, converging all to a single place and then moving to the next iteration.

To this version, one last optimization was made, which was the use of *MPI_Allreduce* instead of *MPI_Reduce* plus *MPI_Bcast*. This change made it so that instead of all processes reducing on the root and waiting for a broadcast to continue the execution, all of them reduce on themselves and eliminated the need for a broadcast since now they would have all the same up-to-date information.

IV. Performance Tests

A. *Machine used*

The machine that will be used to record all of these performance tests will be one of ours. It is relevant to mention that it has 8 cores (4 physical + 4 threads) for later on.

B. *Number of Samples*

In order to test the performance difference by changing the number of samples, the number of clusters used was 4, the number of processes used was 4 and, finally, the number of threads for the parallel version was 16. For the sake of getting results, that more closely represent the real performance of the programs, the results below were obtained by averaging 5 executions times.

TABLE I
Performance varying the number of Samples.

# of Samples	**OpenMP (s)**	**MPI (s)**
10 million	0.59573	0.8565
50 million	2.91587	3.5288
100 million	5.84400	6.8503

C. *Number of Clusters*

In order to test the performance difference by changing the number of clusters, the number of samples used was 10 million, the number of processes was 4 and the number of threads for the parallel version was 16. For the sake of getting results, that more closely represent the real performance of the programs, the results below were obtained by averaging 5 executions times.

TABLE II
Performance varying the number of Clusters.

# of Clusters	**OpenMP**	**MPI**
4	0.5911	0.8405
8	0.8372	1.1174
16	1.3602	1.6478
32	2.3760	2.7270

D. *Number of Processes*

Using the ”-np” option, the number of processes was varied to analyse the benefit of having more processes in our code and therefore its’ scalability. As mentioned in section A, the machine used in our tests was one of ours which has 4 physical cores and 4 virtual cores thanks to hyperthreading. For the sake of getting results for our performance tests, that more closely represent the real performance of the programs, the data below was obtained by averaging 5 executions times.

TABLE III
Performance varying the number of Processes.

# of Processes	**Execution time (s)**
2	1.06496
4	0.8565
8	1.0572

E. *Memory Hierarchy*

In order to compare the cache misses of using a Message Passing Interface against an OpenMP approach, we used 10 million samples and 4 clusters, alongside a varying number of clusters for the MPI tests and 16 threads for the OMP version. Following the same methodology as in the tests before, the average of 5 executions was considered for the results presented in the ensuing table.

TABLE IV
Analysis of Cache Misses.

	Execution time (s)	L1 Cache Misses	# of Instructions
OpenMP	0.5869	28656861	16087830901
MPI (2 processes)	1.06035	32472987	18427107830
MPI (4 processes)	0.8609	36627809	21897499825
MPI (8 processes)	1.0381	45650919	26307334463

V. Results Analysis

A. *Number of Samples*

Analysing the performance tests while varying the number of samples used, it is visible (as expected) that the bigger the number of samples used, the more time it will take for the program to converge or in this case to reach the limit of 20 iterations that the group set. It is also possible to verify that the version which used OpenMP directives outperformed the MPI version of the program. This can be attributed to the fact that there was the use of 16 threads in OMP but only 4 processes in MPI, because of the limitations of our test machine, which also means that it is easier to obtain better performances overall using OMP because of its inherent cheaper way to perform the parallelism of code (Threads vs Processes), that is the inexpensive creation and context switch of threads in comparison with the expensiveness of processes, as well as the other reasons that lead to a bigger overhead.

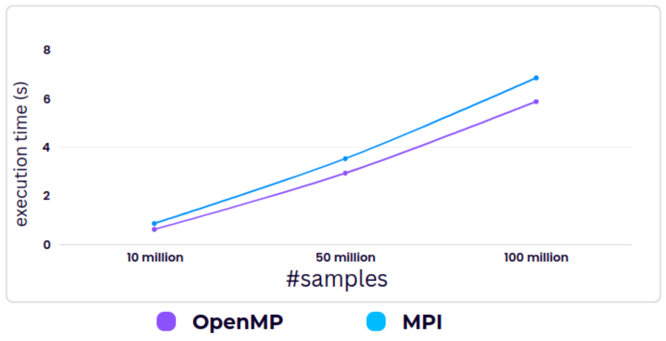

Fig. 1. Execution time, while varying the number of samples for both the MPI and the OpenMP K-Means implementations.

B. Number of Clusters

As expected, the number of clusters greatly influences the performance of our program, if the computational resources aren't scaled properly and accordingly to the increase in clusters used to divide the samples, because of the big increase in instructions and the number of cycles necessary for the programs to converge or to reach the iteration limit artificially imposed by the group for testing purposes.

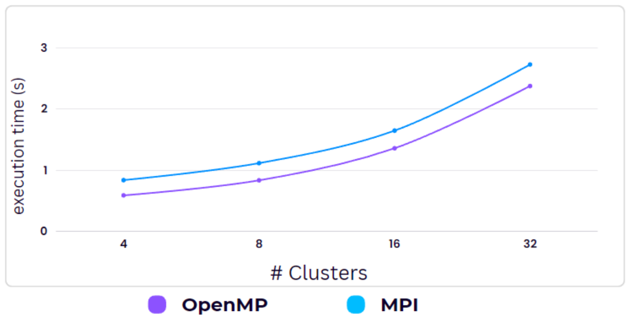

Fig. 2. Execution time, while varying the number of clusters for both the MPI and the OpenMP K-Means implementations.

C. Number of Processes

Regarding the performance tests made varying the number of processes, it is possible to see that the best result was achieved using only 4 processes as opposed to 2 or 8. Dividing the workload from 2 to 4 processes is an intuitive upgrade which shows its results, however, there is an increase in execution time when the number of processes is increased to 8. This is due to the fact that when 4 processes were being used, they were all physical cores on our machine while with 8 processes 4 of those cores were CPU threads (Intel's Hyper-Threading).

Using CPU threads instead of physical cores creates a huge bottleneck that worsens the performance of our algorithm even though the workload was being divided into smaller pieces and all executed at once. Turns out that the physical cores on this machine were too fast and the CPU threads could not keep up, leaving the physical cores waiting for the remaining tasks.

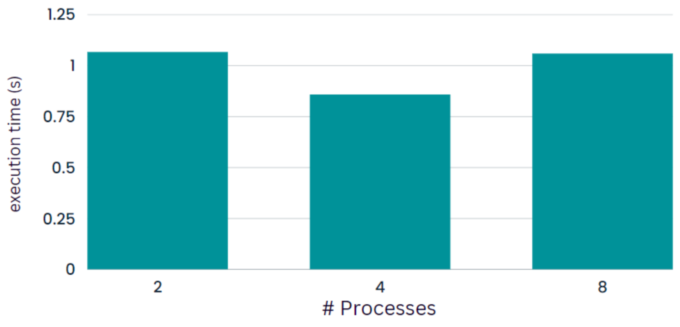

Fig. 3. Execution time, while varying the number of processes.

D. Memory Hierarchy

The memory display on the OpenMP version and MPI version differ significantly due to the nature of the tools of parallelization. The OpenMP version of our code uses threads to achieve a greater result compared with its sequential counterpart, threads are lower in resource consumption and all share the same central memory (unless told not to). Considering the code free of Data Races, this mechanic ensures that are able to share the same memory without compromising the final result, which will end up consuming fewer resources, namely, cache decreasing the number of cache misses and therefore the number of memory accesses.

The MPI version of this algorithm relies on several processes to achieve a parallel execution in order to better its performance. There are however some downsides, due to the way these processes work, they end up using more resources, and occupying more memory and cache because every single process has its own memory, independent of others. This means that the cache and memory that were previously shared are now occupied with possibly unnecessary data, which is isolated and owned by each of the processes.

E. Scalability Analysis

It is probable that the program will scale well if it is executed on a machine that possesses enough cores to reach the maximum speedup point, including changes in the number of clusters and samples, but it is likely that our OpenMP version scales better because of the low overhead of thread creation and context switch.

VI. Conclusion

In summary, code parallelization is a powerful technique that can significantly improve the performance of computationally intensive tasks, either by using OpenMP or a message-passing interface. By splitting a problem into smaller, independent pieces and executing them simultaneously using threads or processes, the execution time can be significantly accelerated. It should be noted, however, that parallelization is not always the best solution, and factors such as data dependencies, the overhead incurred by parallelization, and scalability must be carefully considered. However, with the right approach, code parallelization can provide significant benefits and enable the efficient execution of complex computations.

In addition, this report details the decisions made to convert the OpenMP algorithm to an MPI algorithm, the rationale behind the decisions made, the performance tests performed, and we believe that the primary goal of the project has been achieved, while we could have achieved greater results in terms of performance by using a different machine to test our program on, like the SeARCH computational cluster, and therefore take more advantage of the parallelism implemented with MPI, which greatly benefits of more cores to scale equally the number of processes.

Lastly, we would have liked to have done further scalability analysis, taking into account load balancing and more greatly analysing the effect of varying the number of samples and clusters, while scaling the computation resources used (number of cores) in order to obtain a speedup graphic that could further help us understand the benefits and disadvantages of our MPI approach to the K-means algorithm.