K-Means Algorithm Optimization with MPI primitives

1st José Gonçalves
IT department
University of Minho
Braga, Portugal
pg50519@alunos.uminho.pt

2nd Miguel Fernandes
IT department
University of Minho
Braga, Portugal
pg50654@alunos.uminho.pt

***Abstract*—The present document refers to a practical assignment from a Masters Degree in Software Engineering course unit named Parallel Computing. This phase's goal is to explore an alternative way of implementing the K-Means algorithm by implementing new ways of parallelism, aiming to optimize it in the best way possible, in order to obtain an efficient and fast result.**

***Index Terms*—k-means, parallel computing, performance, optimization, analysis, processes, threads, scalability, MPI, Profilling**

I. Introduction

In the last phase of the project, it was used OpenMP to optimize the performance of the sequential implementation, which led to (approximately) 10,4x performance improvement, considering 4 clusters and the best results in both algorithms. With those results, the only missing improvement could be with the usage of tasks, which was tested and got a worse result than before.

As there wasn't that much more to explore, it was decided to try new ways of parallelism, not only to try to improve the past results, but also to expand the skills to a new parallel programming strategy. That way, it was used MPI to work with in this phase of the project.

While OpenMp is a way to program with shared memory that shares the global addressing space, the MPI works on a memory space for each process in isolation from the others. Here, every bit of code written is executed independently by every process. The parallelism occurs because it's told to each process exactly which part of the global problem they should be working on, based entirely on their process ID (rank).

II. Implementation

The implementation changed from Multi-threading to communicating processes, more specifically, the implementation changed from multi-threading with **OpenMP** to communicating processes with **MPI** (Message Passing Interface). That new paradigm, which uses distributed memory, requires some changes to be made to the previous code.

The new strategy is to divide equitably the N points to all processes and each one is responsible to allocate that points to the clusters. The root processes or process 0 is the one in charge for in each iteration gather the results and broadcast them, which it will be detailed further ahead.

Overall, the implementation in MPI didn't change the core of the last implementation, the main changes from the last implementation on OpenMP are:

- The allocation of variables, which were responsible to store the N points and also the information of the cluster of each point, went from N to N / number of processes. The initialization of that variables also changed, each process initializes randomly N / number of processes points and the root calculates the centroids and broadcast them. It's important to mention that attention was paid to the fact that since the division of N might not be able to be perfectly divided, the last process is programmed to work with the rest of the division.
- All the cycles that iterates N times now were changed to N / number of processes.
- Each process calculates the distances from its points to cluster, allocate them in the nearest cluster and calculates the sum of x and y coordinates, as well as the number of points of each cluster.
- In the end of each iteration, that information is sent to to the root, where the centroids are recalculated and then transmitted to all processes.

In each iteration all the information computed passes through the root which is responsible for the calculation of centroids. So, the primitives of communication between the processes that were used are *MPI_Bcast* and *MPI_Reduce*.

In a first implementation, it was used MPI_Send and MPI_Recv to send from root to all the processes and vice versa. As the root does the main job every time and need to inform the remaining processes, it wasn't necessary to spend 3 Send's and 3 Recv's, so it was replaced with the Bcast. Now, the *MPI_Bcast* is used when root needs to flood some information for all processes. Since it is also needed to share the work of each process, it was used the *MPI_Reduce* for the opposite case, i.e., when root needs to gather information from all processes and combine the results using a operator, for example, to calculate the sum of x and y coordinates of all clusters to recalculate the centroids.

III. Scalability analysis

A. *Strong Scalability*

With the aim to have a reliable comparison with the previous tests done in the past phases, all the tests were done in the Search cluster.

Based on a strong scalability analysis, multiple tests were performed including many different values of clusters and for different task values, which will be presented and discussed later on.

Considering the maximum amount of processes is 40, corresponding to the 20 physical + 20 logical cores in the Search Cluster, the following diagram shows the results obtained in the strong scalability analysis obtained, which measures the speed-up (quocient between times of the best sequential implementation and the parallel version) increase with PU for a fixed problem size. In the analysis were tested 4, 32, 132 and 264 clusters.

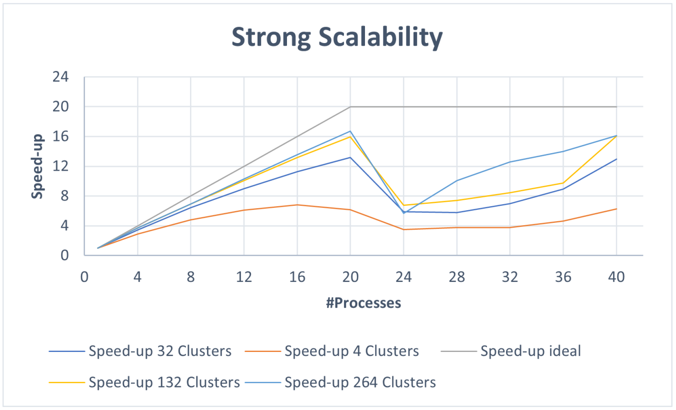

Fig. 1. *Strong scalability* analysis graph

From the Figure 1, it is instantly possible to verify that there is a break in the 20 cores mark, being explained because that is the point where all the physical cores get used and it is needed to invoke the logical cores.

Although, there is an exception at the 4 clusters, which happens to slowly start decreasing the speed-up at the 16 processes mark, just before the others. The most time consuming task in each interaction is the distance calculation (as in the previous phases), which explains the exception presented before. The amount of calculations increases proportionally with the amount of distances calculated, so the bigger the number of clusters, the bigger the computational charge in each process.

The communication overhead is present in between the processes with the reduces and broadcasts, as the lower the number of clusters results in lower effort per process and ends up not being worthy because of the communication overhead. This is especially noticeable by the increase of the speed-up with the increase of the number of clusters chosen.

Knowing the number of clusters doesn't increase much the problem size (more centroids), but the program complexity - i.e., in each iteration more distance calculations are made (as this is the program hotspot) - it was seen as an important part to explore until which number of clusters it was possible to scale up, having an reasonable speed up.

With the graph present in the Figure 1, it is possible to conclude that the ideal amount of processes is 20, being that the general inflection point in the majority of the cases, so, to make it a common measure element, were used bigger cluster sizes to explore and test the scalability of the designed solution, so all the speedups were calculated with 20 processes.

The obtained results are displayed in the following table:

#Clusters	Speed-up
454	17,709
640	17,754
960	16,973
1600	17,83

All the measurements showed before were made with 10 millions points with the propose to only observe the behavior of program against the variation in the number of clusters. That table show that the program scale well with the increase of clusters because the Speed-up remained close to 20, which is the ideal speed-up.

B. *Weak Scalability*

The weak scalability consists in increase the problem data size as the number of processing units. Ideally the execution time should remain constant.

To do this test, only the number of points were increased and the number of cluster were always the same, knowing that the number of points represent almost the totality of the problem size.

The table below show that the execution time stayed constant until 20 processes which was the ideal. Above 20 processes the problem is the same that was described in strong scalability, the Search cluster has 20 physical cores and 20 logical cores, so that performance drop could be explained by the use of the logical cores.

Overall, the weak scalability analysis shows that the program is scalable.

Number of Points	Clusters	Processes	Time
10 000 000	4	1	2,87 sec
20 000 000	4	2	3,06 sec
40 000 000	4	4	3,07 sec
80 000 000	4	8	3,26 sec
120 000 000	4	12	3,16 sec
160 000 000	4	16	3,15 sec
200 000 000	4	20	3,69 sec
280 000 000	4	28	5,63 sec
320 000 000	4	32	8,15 sec

IV. Best input

One of the objectives of that assignment is to find a input suited to each of the levels of the machine's memory hierarchy.

In the analysis of the strong scalability, the measures were made with a fixed number of points. In this section, the goal is to change all the input variables to find the best input. Knowing that the size of the cache of the compute nodes of Search used are:

- **L1d cache** - 64K;
- **L1i cache** - 64K;
- **L2 cache** - 512K;
- **L3 cache/LL cache** - 10236k;

The tables above, show the results of different inputs, in that case, the number of points, the number of clusters was always 132. The number of processes is 20, given that this was the number of processes that provided a better speed-up. The metrics were obtained from the perf tool.

Points	Time	Instructions
10.000	0,18	1,2 * 10^9
100.000	0,22	5,2 * 10^9
1.000.000	0,77	42,8 * 10^9
10.000.000	5,04	408,4 * 10^9

Points	Cache misses	L1-load-misses	LLC-load-misses
10.000	13.133%	11,34%	10,70%
100.000	11,066%	3,59%	12,50%
1.000.000	8,192%	0,38%	11, 48%
10.000.000	20,771%	0,13%	36,94%

Analysing the both tables, considering the L1-load-misses and the LLC-load-misses, that are the most penalizing misses because it means an access to the memory, the best number of points is 1.000.000.

And to find the best number of clusters was used the same technique. The number of clusters does not change significantly the problem size, but the group tested some different numbers to find the best input.

Clusters	Cache misses	L1-load-misses	LLC-load-misses
32	8,207%	1,75%	13,99%
64	5,165%	0,27%	8,45%
132	8,192%	0,38%	11, 48%
264	7,373%	0,30%	11,43%

With that results and using the same metrics that were used before, i.e., the input that minimize the number of misses in all caches, it is possible to conclude that the best input is 1 million of points, 64 clusters and 20 processes.

V. Performance of parallel implementation

Measuring the performance of an MPI implementation is different from OpenMP implementation(last homework). The metrics used to analysis the performance of the parallel implementation were:

A. Inter-process communication

One of the most important aspects to consider is the overhead of the communication between processes. To measure the time wasted between communications, the group used the primitive *MPI_Wtime* before and after each communication.

In the table bellow it is possible to observe that the time of communication don't increase proportionally with the number of points. All the tests were made with 4 processes.

The number of points only changes the size of the arrays transmitted between processes.

Number of points	Clusters	Time of communication
100.000	4	0.004
1.000.000	4	0.02
10.000.000	4	0.13
100.000.000	4	0.84

The same test was made but for a constant number of points and different values of clusters. The number of clusters changes the number of arrays changed between processes

Number of points	Clusters	Time of communication
1.000.000	4	0.02
1.000.000	16	0.26
1.000.000	64	0.78
1.000.000	132	2,13

Looking at both tables, it is possible to conclude that the number of arrays changed affects more the time of communication that the size of the arrays.

B. Load balance

The approach used in the implementation, as described earlier, was to divide the number of points equally by the number of processes so that they all had the same workload. The process 0 have the responsibility to recalculate the centers, which give that process a amount of extra work, while the others processes wait for that centers. That is possible a bottleneck of our implementation.

VI. Conclusion and future work

Given the completion of the project, it is clear that it not only helped to consolidate the knowledge acquired during the semester, regarding the impact of minimal code optimizations that can improve the program performance to another level. With this third and last phase, it can be concluded that the inter-process communication can be an alternative way of parallel programming, instead of the, e.g., multi-threaded OpenMP that was used in the previous phase, being possible to achieve very close results in both.

Indeed, the group would like to deliver a better metric analysis, by using propped MPI profiling tools like PAPI, Scalasca or Tau, as a way of helping to develop a better implementation, since with that it's possible to get reports of communication overhead and other important metrics. Although the *profiler* used wasn't the most adequate for the problem type, it served as a tool for code and metric analysis, revealing very helpful.