A Parallel K-Means clustering algorithm with OpenMP and MPI

Elia Zonta
Department of Information Engineering
University of Trento
Trento, Italy
elia.zonta@studenti.unitn.it
227092

Alex Pegoraro
Department of Information Engineering
University of Trento
Trento, Italy
alex.pegoraro@studenti.unitn.it
227642

***Abstract*—Clustering stands out as a widely adopted data analysis method, finding applications across diverse fields such as image segmentation, bioinformatics, pattern recognition, and statistics. The K-means algorithm, renowned for its simplicity, ease of implementation, efficiency, and empirical success, is a prevalent choice for unsupervised clustering tasks. However, real-world scenarios often involve massive datasets, posing a considerable challenge in efficiently managing and mining such extensive data. The parallel programming models come in as valuable assets in such scenarios. This algorithm allows taking advantage of parallel architectures for effective clustering applications. Moreover, our experimental studies compare the shared memory model OpenMP and distributed memory model MPI and will reveal that the K-Means algorithm implementation using the Message Passing Interface (MPI) exhibits notable stability and demonstrates low time overhead even when handling extensive datasets.**

***Index Terms*—k-means, clustering, unsupervised learning, OMP, MPI**

I. Introduction

Clustering serves as a fundamental approach in unsupervised learning, and serves as a prevalent technique for data analysis with widespread applications in various domains. Its utility extends across disciplines such as image segmentation, bioinformatics, pattern recognition, and statistics, embodying a versatile tool for uncovering recurrent patterns and structures within datasets. As the definition of unsupervised learning, this method, devoid of predefined labels or categories, enables the identification of inherent relationships among data points. Its adaptability and effectiveness contribute to its prominence as a go-to strategy in diverse fields, providing valuable insights into the underlying organization of complex datasets [1]. K-Means is well-known for its simplicity and easy implementation. It was ranked second of top 10 algorithms in data mining by the ICDM Conference in October 2006 [3]. Compared with other clustering algorithms, K-Means has a simple implementation, is really efficient when handling large datasets and has a solid theoretical foundation based on the greedy optimization of Voronoi partition [2]. Our work aims to further improve the runtime duration of K-Means, exploiting parallelization techniques. The two main tools we will dive into are OpenMP and MPI, of which we will evaluate performances through metrics such as Strong Scaling, Weak Scaling, Speedup and Efficiency gain.

II. Background

This section is developed to give a comprehensive overview of K-Means algorithm, together with a brief presentation of the Application Programming Interfaces (APIs) used to craft a parallel implementation of K-Means algorithm. More precisely Section A will focus on an introduction to K-Means clustering algorithm followed by Section B that will present the OpenMP API concluding with Section C that will delve into the Message Passing Interface (MPI) API.

A. K-Means

K-Means is a popular and widely used clustering algorithm in the field of machine learning and data analysis. The algorithm is designed to partition a dataset into K distinct, non-overlapping subsets or clusters, where each data point belongs to the cluster with the nearest mean. The central idea behind K-Means is to iteratively assign N data points to K clusters ($N \geq K$) based on their proximity to the cluster centroids and update the centroids to minimize the total intra-cluster variance. This process is repeated until convergence, resulting in a set of clusters that capture the inherent structure of the data.

1) Random centroids initialization
2) Points assignment based on the distance from the centroid (e.g. Euclidean distance)
3) Computation of new centroids
4) Iterate 2. and 3. until convergence is reached

B. OpenMP

The Open Multi-Processing (OpenMP) application programming interface (API) facilitates parallel programming in shared-memory systems. It provides a set of compiler directives, library routines, and environment variables to enable the creation of parallel programs. It aims to simplify parallel programming by allowing developers to add parallelism to their existing code easily. Key features of OpenMP include:

- **Directive-based Approach:** utilizes compiler directives (pragmas) added in the source code, to specify parallel

regions and guide the compiler in generating parallel code.

- **Shared-memory Model:** OpenMP is well-suited for shared-memory architectures, where multiple processors share a common address space. It enables the creation of parallel threads that can communicate and synchronize through shared memory.
- **Parallel Constructs:** the API provides a set of parallel constructs that include parallel regions, loops, and sections. These constructs allow developers to parallelize specific parts of their code.
- **Portability:** OpenMP is designed to be portable across different architectures and compilers.

In summary, OpenMP is a widely adopted and flexible parallel programming model that simplifies the development of parallel applications for shared-memory architectures.

C. Message Passing Interface

The Message Passing Interface (MPI) stands as a standard crafted by the Message Passing Interface Forum (MPIF). This standard library specification is designed to facilitate parallel computing within a distributed memory environment. One of the notable strengths of MPI is its versatility, accommodating both point-to-point and collective communication mechanisms. This approach empowers developers to harness the potential of parallel processing by seamlessly exchanging information among distributed computing nodes. It is able to provide an exhaustive exploitation of the parallel architecture to get the best absolute performance, but it requires an in-depth review of your code.

III. Methodology

A. Computing System

Every experiment has been tested on the UniTrento HPC Cluster. The complete infrastructure is composed of:

- 142 CPU nodes for a total of 7.674 cores
- 10 GPU nodes for a total of 48.128 CUDA cores
- 2 frontend nodes
- 65 TB of Ram
- Total theoretical peak performance: 478,1 TFLOPs
- Theoretical peak performance CPU: 422,7 TFLOPs
- Theoretical peak performance GPU: 55,4 TFLOPs

All the nodes are connected with a 10Gb/s network, some of them with Infiniband and others with Omni-Path connectivity. The operating system is Linux CentOS 7, while the cluster workload manager software is PBS. During each of our experiment, an output log has been created. Among other information, inside it you will also find the exact structure of the computing node to which the algorithm was launched. The selected parameters have been 65536 datapoints, 64 centroids, both with 8 features each, and 128 maximum epochs.

B. Algorithmic approach and data structures

The sequential algorithm features a rather straightforward implementation where, at each iteration, the distance from each data point to every centroid is calculated, and the point is then assigned to the nearest centroid. Following the assignment, the centroids undergo recomputation. In our implementation, we leverage what we refer to as a "cumulative matrix," an $N \times M$ matrix (`std::vector⟨std::vector⟨double⟩ ⟩`) with N rows corresponding to the number of clusters and M columns representing the number of features in the dataset. Whenever a new datapoint is assigned to the ith cluster, the ith row of the cumulative matrix accumulates the features of the datapoint, aiming to reduce overhead during centroid computation at the assignment's conclusion. A counter (`std::vector⟨int⟩`) tracks the number of points assigned to each cluster, ensuring that the centroid computation cost remains *O($K \times N$).*

IV. Parallel Approaches

A. OpenMP algorithm

The suitability of algorithms for the shared memory model has been assessed using both static and dynamic scheduling techniques. Static scheduling is well-suited for tasks where the thread workload is known at compile time, enabling the data to be divided into chunks to prevent uneven distribution of work among threads. In contrast, dynamic scheduling in OpenMP involves dynamically assigning loop iterations to threads at runtime. Unlike static scheduling, which determines workload distribution at compile time, dynamic scheduling allows the OpenMP runtime system to adjust the distribution of iterations among available threads during the parallel loop's execution. This aligns more effectively with our task, as we are unaware, during each iteration, of which point will be assigned to which cluster. In both implementations a particular care has been put on the pragma directives, especially with the "reduction(+:delta)" : the update of the delta variable allows the correct track of the tolerance, that if lower than an arbitrary boundary allows for an early stopping of the algorithm. A data race problem on that variable could end up in more iterations and more time wasted when the algorithm has already converged.

B. Message Passing Interface (MPI)

The algorithm for the distributed memory model required a comprehensive overhaul compared to its OpenMP counterpart. The first problem is that MPI messages does not support C++ classes, so we had to first convert our `point` data structure into a dynamic matrix. Regarding the implementation, each MPI core is allocated a subset of the dataset and all centroids, creating its own cumulative matrix. The core independently manages the assignment of points to their respective clusters. This phase operates autonomously, with each MPI core functioning independently and having awareness of the points it assigned and the updates to its cumulative_matrix.

In the subsequent phase, leveraging the `MPI_Allreduce` directive, the cumulative matrix is merged with others using a reduce operation together with a broadcast operation, enabling every MPI core to compute the global cumulative matrix during the message itself. The same approach is followed

to reduce the counters, and so centroids can now be computed by every node independently. This approach minimizes data transfer, ensuring efficient communication between cores while moving the least amount of data.

V. Results and Discussion

The algorithms have been compared throw two main aspects. First of all, the correctness of the parallel approaches have been ensured, thanks to the `compare_results` script. It showed that all the instances of K-Means produce the exact same result, with the only exceptions be the weak scalings, as of course they operate on different data sizes.

On the aspect of performance, four main metrics have been taken into account:

- Strong Scaling: Increasing the number of processors while the dataset is still, we expect to see the time decreasing in an almost linear manner.
- Speedup: Is the ratio between the time took to complete the algorithm with only one core and the one needed with multiple cores.
- Efficiency: The next step is to divide the speedup by the number of cores at a certain step. This tell us how much overhead we are introducing.
- Weak Scaling: This time, we are increasing the dataset dimension together with the number of cores. We expect the execution time to be almost the same.

Having a look at our plots, that you can find in the plot folder of our repository, we unfortunately discover that Speedup and efficiency of the algorithms are quite bad. This is often expected and explainable with the increasing communication overhead. Also the strong scaling is not so ideal. We can witness a clear dominance of the MPI approach with asynchronous communication, whose plot is reported in Fig. 1. On the other hand, pure MPI quickly become worst than the serial and OMP stays quite stable.

Surprisingly, the weak scaling are on the other hand quite nice. OMP presents an almost liner increasing of time (Plots are in logarithmic scale so this look like an exponential) while both MPI codes remains quite stable until the last cases. We report a plot of a weak scaling in Fig. 2.

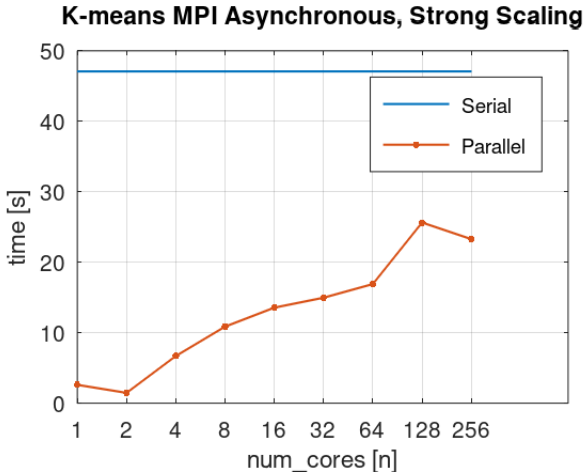

Fig. 1. Strong scaling of MPI asynchronous approach.

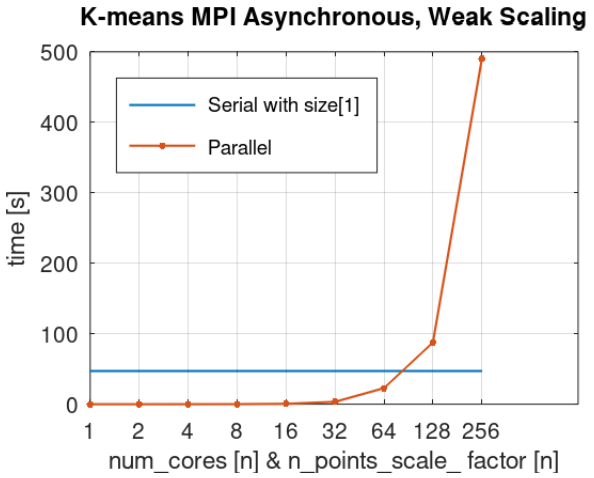

Fig. 2. Weak scaling of MPI asynchronous approach.

VI. Conclusion

In this report we provided a comparison between a naively implemented K-Means algorithm and two parallel programming models. Moreover we delved into the shared memory model OpenMP, taking advantage of static and dynamic scheduling and comparing its performances, then the whole algorithm and the toolchain has been modified and adapted for the distributed memory programming model, Message Passing Interface (MPI) exploring asynchronous communication. OpenMP offers a set of ready-to-use directives but the performance gain obtained isn't huge. Whereas MPI requires a non-trivial amount of effort, fine tuning and debugging the code but the performance gains are much better and becomes a must-use for massively parallel tasks if a distributed memory architecture is available. The experiment revealed that K-Means is not so trivially parallelizable, neither in the implementation nor in the performance gain. Even thought the results are not the best, we have anyway formally explored a possible approach, highlighting the difficulties and challenges that we had to tackle. The algorithm implementation with MPI has the best performing especially over extensive datasets, however tests on the stability, efficiency and portability of such algorithms will be addressed by further work and studies.

VII. Authors and Contributions

- **Elia Zonta:** project structure, dataset creation algorithm, serial algorithm, OMP algorithms, report.
- **Alex Pegoraro:** output comparison algorithm, MPI algoriths, benchmarks, plots, report.

VIII. Git and Instructions for Reproducibility

https://github.com/AlphaNightLight/parallel_k_means.git
Please refer to the GitHub code repository and particularly refer to README.md for specific instructions for reproducibility.

REFERENCES

[1] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010. Award winning papers from the 19th International Conference on Pattern Recognition (ICPR).

[2] J. MacQueen. Some methods for classification and analysis of multivariate observations. 1967.

[3] Xindong Wu, Vipin Kumar, Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, G. Mclachlan, Shu Kay Angus Ng, Bing Liu, Philip Yu, Zhi-Hua Zhou, Michael Steinbach, David Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14, 12 2007.