Data Mining Algorithms
Parallelizing K-Means Clustering with MPI

Eleonora Renz, 239020
eleonora.renz@studenti.unitn.it

1 INTRODUCTION

While the amount of data available to companies keeps increasing, a way of processing this data in a timely manner needs to be utilized. Parallelizing popular data mining algorithms helps data scientists work with large datasets. In this report parallelization of the k-means algorithm using MPI is examined. The reason why the k-means algorithm was chosen for this project is that it is the most popular clustering algorithm for most real world applications [1]. K-means is a clustering algorithm that clusters a dataset into k clusters, where each cluster is defined by the mean of its members [3]. However, as the input dataset size increases, the time it takes to run the algorithm also increases, making it more computationally intensive. The idea is to reduce computational time by parallelizing this classic algorithm using MPI. While there already are a couple of papers discussing parallelization of K-means using MPI or OpenMP [5], it would not be possible to directly compare the results of the papers to the results we would achieve with our solution, because no code nor data was provided with the papers. Instead, the developed code was written from scratch and applied to a real life dataset concerning credit data.

2 PROBLEM ANALYSIS

In this chapter, the k-means clustering algorithm is explained in its serial form and possible limitations of it are discussed. K-means is a data mining algorithm that clusters a dataset into k clusters, where each cluster is defined by the mean of its members [4]. The algorithm starts with an initial set of k centroids and iteratively assigns each data point to a cluster based on the shortest distance to a centroid it has. The centroids of each cluster are then recalculated as the mean of all associated points and the process is repeated until the centroids stabilize. The pseudocode for this as follows:

Algorithm 1 Serial k-means

1: Set initial centroids k = $<k_1, k_2, \ldots, k_k>$
2: Calculate distance of each point to the k centroids
3: Assign each point to a cluster based on the shortest distance
4: Calculate new centroids based on cluster members
5: If k and k' are different
6: Set k to be k' and repeat from step 3
7: Else stop

While this algorithm is simple and intuitive, it has some shortcomings. One of the key difficulties is the increase in computational time with the increase of data points, clusters, or feature dimensions. The time complexity for k-means is O(nkI*d) [2]. There also are additional issues such as sensitivity to outliers and a poor choice of number of clusters, which can lead to bad results.

3 MAIN STEPS TOWARDS PARALLELIZATION

3.1 Solution Design

The k-means clustering algorithm can be easily parallelized because the distance computations between one point with the centroids is irrelevant to the distance computations between other points with the centroids. Distance computations between different points with centroids can be executed in parallel. The pseudocode for this looks like so:

Algorithm 2 Parallel k-means

1: Set initial centroids k = $<k_1, k_2, \ldots, k_k>$
2: Split data amongst s processes into s subgroups of equal size
3: **for** every subgroup **do**
4: Calculate distance of each point to the k centroids
5: Assign each point to a cluster based on the shortest distance
6: **end for**
7: Gather all points with their cluster associations
8: Calculate new centroids based on cluster members
9: If k and k' are different
10: Set k to be k' and repeat from step 3
11: Else stop and return all points with their cluster association

In this implementation the root process is responsible for splitting the data through the number of processes used and sending the chunks out to their corresponding processes using the collective communication technique MPI_Scatter. Additionally, the k initial centroids are broadcasted from the root process to all others using MPI_Broadcast. This lets every process do the distance calculations for its respective data subset. The re-calculation of the centroids only needs to be done by one process, however. For this all data subsets with their new cluster associations are gathered together in the root process using MP_Gather. From here the root process re-calculates the centroids and checks if they have changed. If they have changed the new centroids need to be broadcasted to all processes and another iteration starts. If they did not change though, the algorithm can stop. To indicate to all processes that the iterative loop can stop, an indicator needs to be broadcasted to all as well.

3.2 Implementation

As this was not only my first MPI, but also the first C application, a step-wise approach to the implementation was taken. For a first step a serial implementation in C was developed on a small self-generated 2D dataset. This was done to ensure that the single operations work as expected and the results were sanity checked by plotting them. As a next step the parallel MPI solution was built on the self-generated 2D data set. After again ensuring that the application runs as expected, the code was adjusted to process a much bigger, real world dataset. Most difficulties encountered were memory and C related. One of the MPI related tricky parts stumbled

upon while implementing this project is the need to define a custom MPI_Datatype for the data struct so that it can be sent and received by the different processes. While not implemented in the end due to inoperativeness, it was also tried to use MPI_Reduce with a custom operator MPI_Op to combine newly calculated centroids from each process.

4 BENCHMARKING

4.1 Setup

To benchmark the developed solution, several experiments were defined. These experiments aim at understanding how well k-means is parallelized and how best to set nodes, CPUs, and processes used in relation to each other. All experiments are benchmarking wall time of k-means clustering without I/O, as the I/O part is not parallelized anyway. Additionally, to have more representative time measurements, each run was submitted ten times and later the time was averaged out. The data set used for these experiments is based on a credit card data set found on Kaggle. However, the data set size was very small, which is why it was artificially increased by adding rows of random data within the min-max bounds of each column. The experiments run for the benchmarking are listed in Table 4.1 and described as follows:

(1) Light data set size (8950 rows, 6 columns) with a fixed size of nodes and CPUs while increasing the number of processes used.
(2) Heavy data set size (17950 rows, 6 columns) with a fixed size of nodes and CPUs while increasing the number of processes used.
(3) Heavy data set size with increasing nodes along number of processes while keeping CPUs fixed.
(4) Heavy data set size with increasing CPUs along number of processes while keeping nodes fixed.

To ease the execution of the benchmarking separate bash scripts were written for each experiment to automatically submit the customized jobs. The results are processed through a python script by taking the generated output files, averaging the runs and plotting the graphs.

Experiment	Dataset size	Nodes	CPUs	Processes
Light	501200B	3	3	1–30
Heavy	1005200B	3	3	1–30
Heavy Nodes	1005200B	1-30	1	1–30
Heavy CPUs	1005200B	1	1-30	1–30

4.2 Results

In this section the results of the different experiments done for the benchmarking are evaluated. In Figure 1, the run of the first experiment on a small dataset can be seen. While keeping the number of nodes and number of CPUs constant at three each, the number of processes increases along the X-Axis. The general idea that with increasing processes the wall time it takes to solely run the k-means algorithm decreases is reflected by this graph. This means that the parallelization implemented works. However, given that it is a very small dataset some unexpected behavior regarding the efficiency while increasing processes is seen. This might stem from the fact that the time even for the serial run is only three seconds, leading to any unexpected behavior from the cluster to be disproportionately high.

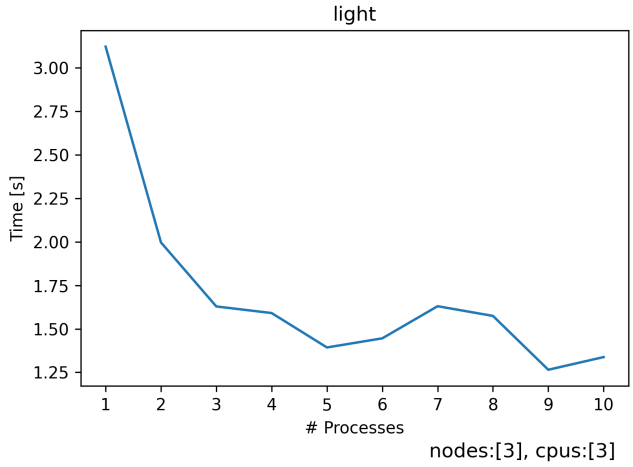

Figure 1: K-Means Benchmark Experiment #1

In Figure 2, the second experiment with an increased dataset size compared to experiment one is seen. Here we take a closer look at the behavior of the graph. As expected, the time it takes to execute the algorithm decreases between one and three processes. Once a fourth process is added the time no longer decreases monotone. This is because of the CPU configuration set per node being three. As soon as there are more processes than CPUs for one node a new node needs to be used for most reasonable resource utilization. This however introduces parallel overhead in terms of data communications hence increasing the time a bit again. From here one we can see that whenever the maximum number of CPUs per node is used by the processes, overhead leads to the explained phenomena. One more interesting point is the last run with ten processes. Considering that there are a total of nine CPUs (3nodesx3CPUs/node) available in this configuration, it makes sense that as soon as there are more processes than total CPUs the time will increase stark as one CPU now has to run two processes.

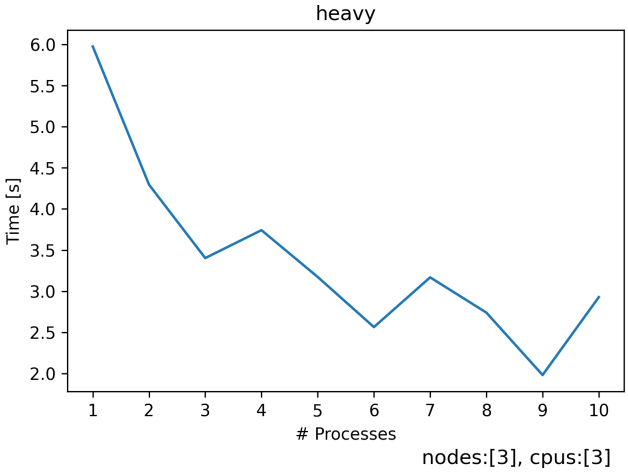

Figure 2: K-Means Benchmark Experiment #2

Figure 3 depicts the third experiment in which the number of nodes increases alongside the number of processes while being set to one CPU/node. As each process is being executed by a different node the overall decrease in time is logical. The increase in overhead between the steps should be about the same. However, it is interesting to see that there is a time increase between 8 and 24 nodes. While it is run on the declared heavy dataset, it is unlike to be due to network issues, as the dataset in fact is not very big actually. A different explanation could just be unlucky runs in terms of variance of times. While running the experiments it was sometimes observed that one out of ten runs with the same configuration would take up to 27x times longer than the others. At this point we are not sure why these outliers appear on this cluster sometimes.

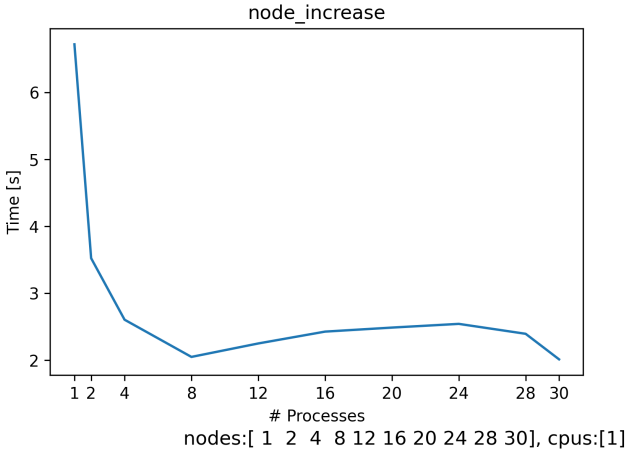

Figure 3: K-Means Benchmark Experiment

The last experiment as defined in the setup is shown in Figure 4. For this experiment the number of CPUs was increased alongside the number of processes on one node. In this scenario no communication overhead over the network is needed as it all is done within the same node. Here we see that the time it takes is reduced with every new process introduced. This is maximizing the resources available as each CPU is running one process.

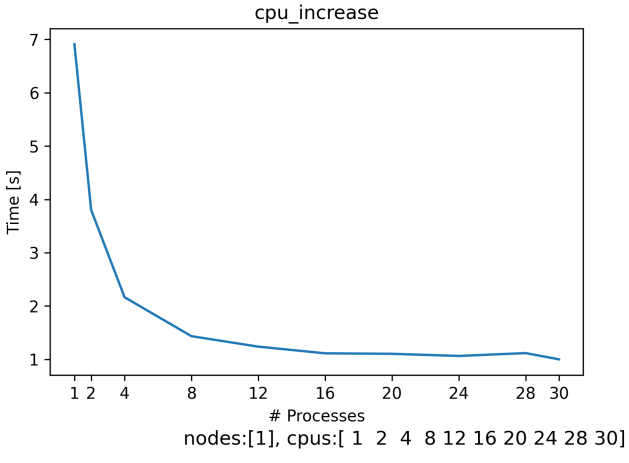

Figure 4: K-Means Benchmark Experiment #4

As two last graphs, Figures 5 & 6 display the speedup for the k-means algorithm. As with the last experiment, Figure 5 shows that the number of CPUs was increased alongside the number of processes within one node. While a linear speedup would be ideal, it usually is not achievable due to overhead such as task start-up time and synchronizations. The parallel k-means did however achieve a typical success for parallelized algorithms with an increasing speedup that flattens out over increased processes. Figure 6 shows the speedup of increasing nodes along processes as done in experiment three. The in the corresponding experiment discussed strange behavior can be seen here as well. While there is an overall speedup, the decrease between processes 8 and 24 are abnormal.

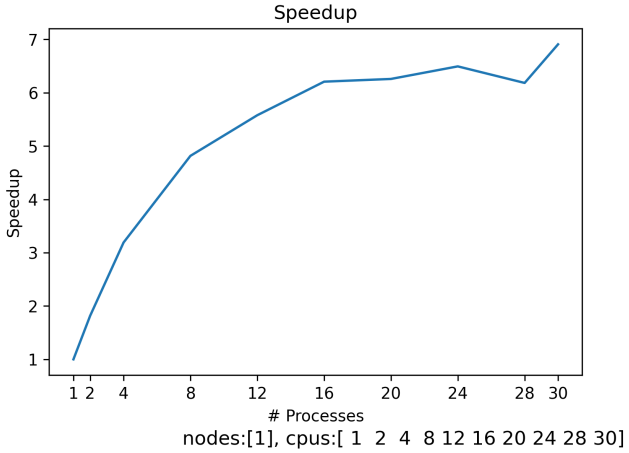

Figure 5: K-Means Speedup with CPUs

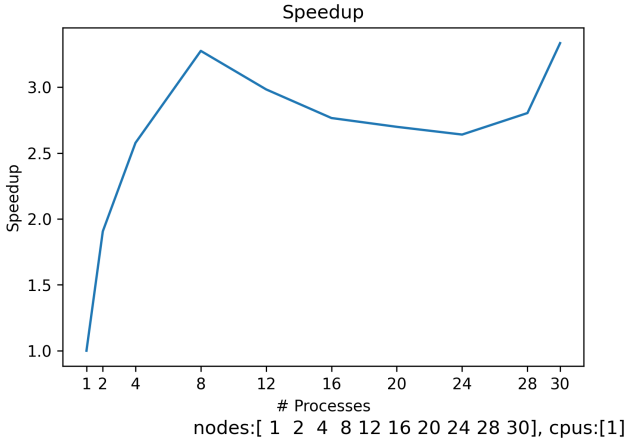

Figure 6: K-Means Speedup with nodes

5 FINAL DISCUSSION

Developing an MPI-based k-means clustering algorithm helped me learn about High Performance Computing, MPI, and C. Being new to all three of these topics, it was challenging to develop this application from scratch. However, understanding both the benefits and complexity, I see how it is important for data scientists to learn this in order to develop more efficient solutions to big data problems. Besides of C-specific improvements to make the code more robust

and flexible, there also are possibilities to enhance the solution in other directions. Some improvements that can still be made on top of this project are benchmarking on a bigger data set, as the computation time even for the heavy data set only took a few seconds in its serial fashion. Increasing the data set size would smoothen out the benchmarking results as other factors such as e.g. cashing would make a proportionally less heavy impact on the results. A concrete improvement of the MPI code and communication patterns would be to have each process in the current iteration calculate their centroids themselves and have the root process gather these current centroids and average them out. This would also remove the need to send all data points with their cluster associations back and forth between the root and all other processes all the time. As an extension to the MPI solution an OpenMP solution on the same codebase could be developed as well. This is possible, which is why it was started, but unfortunately not finished due to time constraints. Comparing the MPI solution to the OpenMP solution would have given insights into performance differences between these two solutions in this scenario. Another aspect to expand on is increasing the number of dimensions in the dataset and comparing benchmarks on this could be interesting as this is another factor increasing the computational complexity of k-means. For this a different approach to the parallelization could be chosen, splitting the data along the columns and comparing results between the different parallelization approaches.

REFERENCES

[1] P. Berkhin. 2006. *A Survey of Clustering Data Mining Techniques.* Springer Berlin Heidelberg, Berlin, Heidelberg, 25–71. https://doi.org/10.1007/3-540-28349-8_2

[2] Jiawei Han, Micheline Kamber, and Jian Pei. 2012. 10 - Cluster Analysis: Basic Concepts and Methods. In *Data Mining (Third Edition)* (third edition ed.), Jiawei Han, Micheline Kamber, and Jian Pei (Eds.). Morgan Kaufmann, Boston, 443–495. https://doi.org/10.1016/B978-0-12-381479-1.00010-1

[3] J. A. Hartigan and M. A. Wong. 1979. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108. http://www.jstor.org/stable/2346830

[4] Stuart P. Lloyd. 1957. K-means Clustering. *J. Soc. Indust. Appl. Math.* 2, 1 (1957), 128–137.

[5] Sighakolli Tirumal Rao. 2009. Parallelization Of Data Mining Algorithms Along With Memory Mapped Files On Dual-Core Processors.