OMP KMeans

Implementation of KMeans Algorithm With OMP

Kiarash Vosough
Amin Erfanian
Mahsa Akhavan Fard
Dr. Savadi
June 10, 2022

Abstract

The world of computers and algorithms has evolved a lot during the last decades. As computations grow, computer hardware must increase to provide more computation power for complicated problems. However, there is a limit to improving the hardware, and it is not solved by now. So scientists decided to find another way to take advantage of current hardware to solve those complicated problems. They managed to set up several pieces of CPU next to each other and use them as one Entity. This method was named Multiprocessing. So far, Multiprocessing has helped scientists, programmers, engineers, etc., solve problems faster. In this project, we have used Multiprocessing to reimplement the KMeans algorithm to parallelize it with the help of the OMP API on C++. In the end, we measure the benefits of parallelization on KMeans by performing several tests.

1 - Introduction

Multicore programming helps create concurrent systems for deployment on a multicore processor and multiprocessor systems. A multi-core processor system is a single processor with multiple execution cores in one chip. It has various processors on the motherboard or chip. A Field-Programmable Gate Array (FPGA) might be included in a multiprocessor system. An FPGA is an integrated circuit containing an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects. Input data is processed to produce outputs. It can be a processor in a multicore or multiprocessor system or an FPGA.

The multicore programming approach has the following advantages and minus:

1. Multicore and FPGA processing helps to increase the performance of an embedded system.
2. It also helps to achieve scalability so that the system can take advantage of increasing numbers of cores and FPGA processing power over time.

Concurrent systems that we create using multicore programming have multiple tasks executing in parallel. This is known as concurrent execution. When a processor executes numerous parallel tasks, it is known as multitasking. A CPU scheduler handles the functions that perform in parallel. The CPU implements studies using operating system threads. So that tasks can run independently but have some data transfer between them, such as data transfer between a data acquisition module and controller for the system. Data transfer occurs when there is a data dependency.

2 - Multicore Systems Challenges.

Since a multicore system consists of more than one processor, you need to keep all of them busy to use multiple computing cores better. Scheduling algorithms must be designed to use various computing cores to allow parallel computation. The challenge is also to modify the existing and new multithreaded programs to take advantage of the multi-core system.

In general, five areas present challenges in programming for multicore systems:

Dividing Activities

The challenge is to examine the task correctly to find places that can be divided into separate, concurrent subtasks that can execute parallelly on individual processors to make exclusive use of multiple computing cores.

Balance

While dividing the task into sub-tasks, equality must be ensured that every sub-task should perform almost equal work. It should not be the case that one sub-task has much work to perform and other sub-tasks have significantly less to do because, in that case, multicore system programming may not enhance performance compared to a single-core system.

Data splitting

Just as the task is divided into smaller sub-tasks, data accessed and manipulated by that task must also be divided to run on different cores so that data can be easily accessible by each sub-tasks.

Data dependency

Since various smaller sub-tasks run on different cores, it may be possible that one sub-task depends on the data from another sub-tasks. So the data

needs to be adequately examined so that the execution of the whole task is synchronized.

Testing and Debugging

When different smaller sub-tasks are executing parallelly, testing and debugging such concurrent tasks is more complex than testing and debugging single-threaded applications.

3 - K-Means

K-Means clustering is a vector quantization method, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. K-Means clustering minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances, which would be the more difficult Weber problem: the mean optimizes squared errors, whereas only the geometric median minimizes Euclidean distances. For instance, better Euclidean solutions can be found using K-Medians and K-Medoids.

3.1 - How does It Operate?

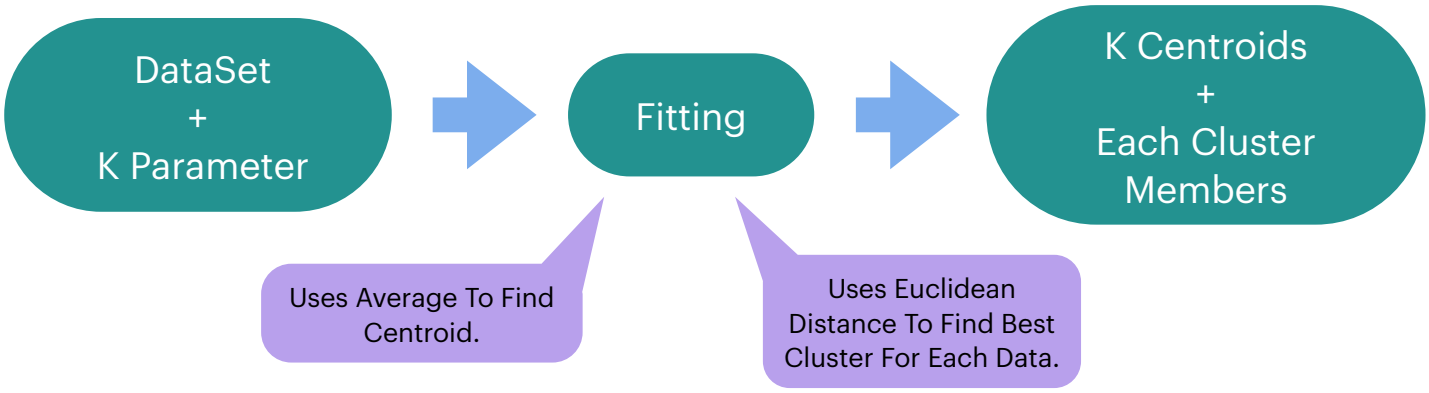
The algorithm takes the unlabeled dataset as input, divides the dataset into k-number clusters, and repeats the process until it does not find the best clusters. The value of k should be predetermined in this algorithm.

The k-means clustering algorithm mainly performs two tasks:

1. An iterative process determines the best value for K center points or centroids by using the average of cluster members.

2. Assigns each data point to its closest k-center by using Euclidean Distance. Those data points which are near to the particular k-center create a cluster.

Hence each cluster has data points with some commonalities, and it is away from other clusters.



4 - Project

We started our project by defining what we expected at the end and came up with some features. Some of these features require infrastructures to be implemented at first. So we first implemented those infrastructures. Also, messy codes without any structure were avoided. Hence OOP paradigm was used to shape the project's design.

4.1 - Features

- Structured
- Documented In Code
- Accepts Multi Dimension DataSet Inputs
- Open-Sourced
- Accept Generic DataSet
- Used Multiple Parallel Approaches
- Plotting Final Clusters

4.2 - Skeleton

The infrastructure will be discussed in this section. Some classes and models were implemented to be used for each data input or each approach.

4.2.1 - Data<T>

This generic class was created to model input data. It can hold several attributes related to data:

1. Id
2. Cluster Id
3. Values of type T
4. Dimensions

It is also an Equatable class, which can be compared with other instances of type *Data<T>*. Input data can have several dimensions. Therefore the value associated with each dimension is stored in a vector where the index indicates the dimension number.

4.2.2 - Cluster<T>

To model the created K clusters, this class was implemented. Each cluster contains children and a centroid. To Distinguish clusters, each of them has a unique Id.

4.2.3 - KMeans<T>

This is an abstract class that has some capability to be used in the KMeans algorithm. It more or less follows the Template-Method pattern. All the approaches inherit from this class.

4.2.3.1 - properties

1. K Parameter
2. Thread Count To Be Used
3. Max Iteration
4. Dimension count
5. DataSet Count
6. Clusters

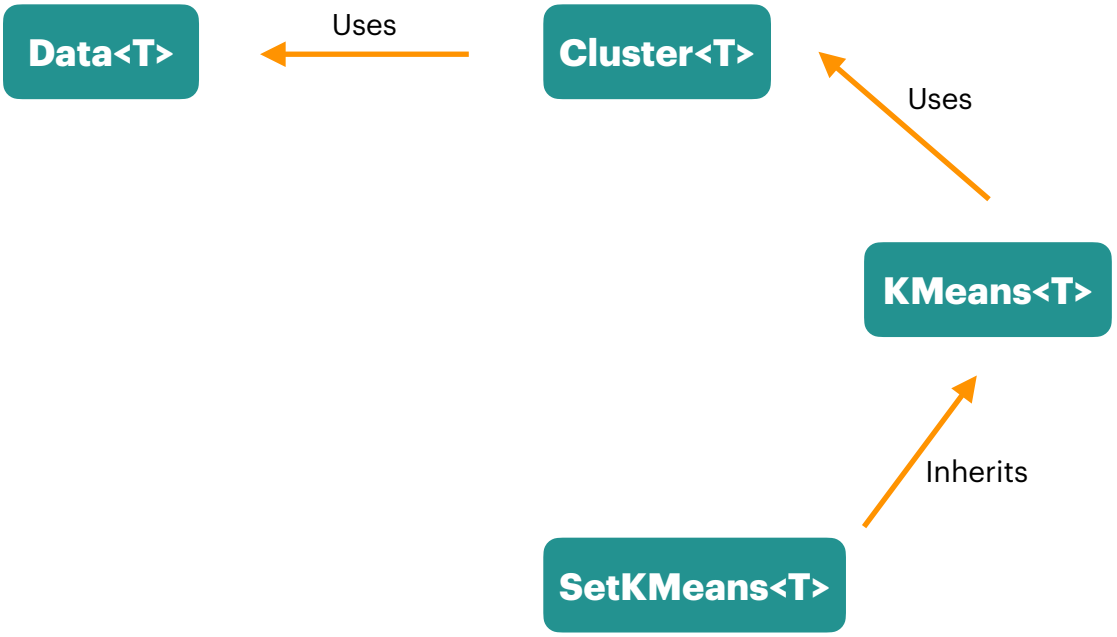
4.2.3.2 - Methods

1. fit
2. findClosestClusterId
3. initializeClusters
4. detectDataDimensions
5. storeCentroidWithUsedIteration
6. reviseCentroidsOfClusters
7. checkForCompletion
8. saveStringToFileAndPrintOnConsole
9. printResults
10. saveCSV

4.2.4 - SetKMeans<T>

Padding approaches are needed to use a 2D Array. Therefore they required two methods for *intersection* and *union* to detect the final cluster's members. Those methods were implemented inside ***SetKMeans***.

4.2.5 - Observation Until Now



4.3 - Parallelization Approaches

A separate class is defined For each approach. Each will be discussed in this section.

4.3.1 PadKMeans<T>

This approach can address the **false-sharing** problem which can occur in caching. By using padding, the mentioned problem can be solved. As mentioned, clusters' results in the 2D array should be merged to detect real clusters. Therefore this approach inherits from *SetKMeans<T>* to access the *union* and *intersection* methods. The class also accept padding number.

4.3.1.1 - properties

1. paddingCountToBeUsed

4.3.2 - SerialKMeans<T>

SerialKMeans is the simple serial implementation of KMeans and uses no parallelism. It was implemented at first to get an insight into how to implement the rest of the approaches.

4.3.3 - CriticalKMeans<T>

In this approach, *omp parallel* construction was used. Hence to avoid data race, *omp critical* was used to modify the cluster's members. In the end, this is the best approach, at least for provided test cases.

4.3.4 - SPMDKMeans<T>

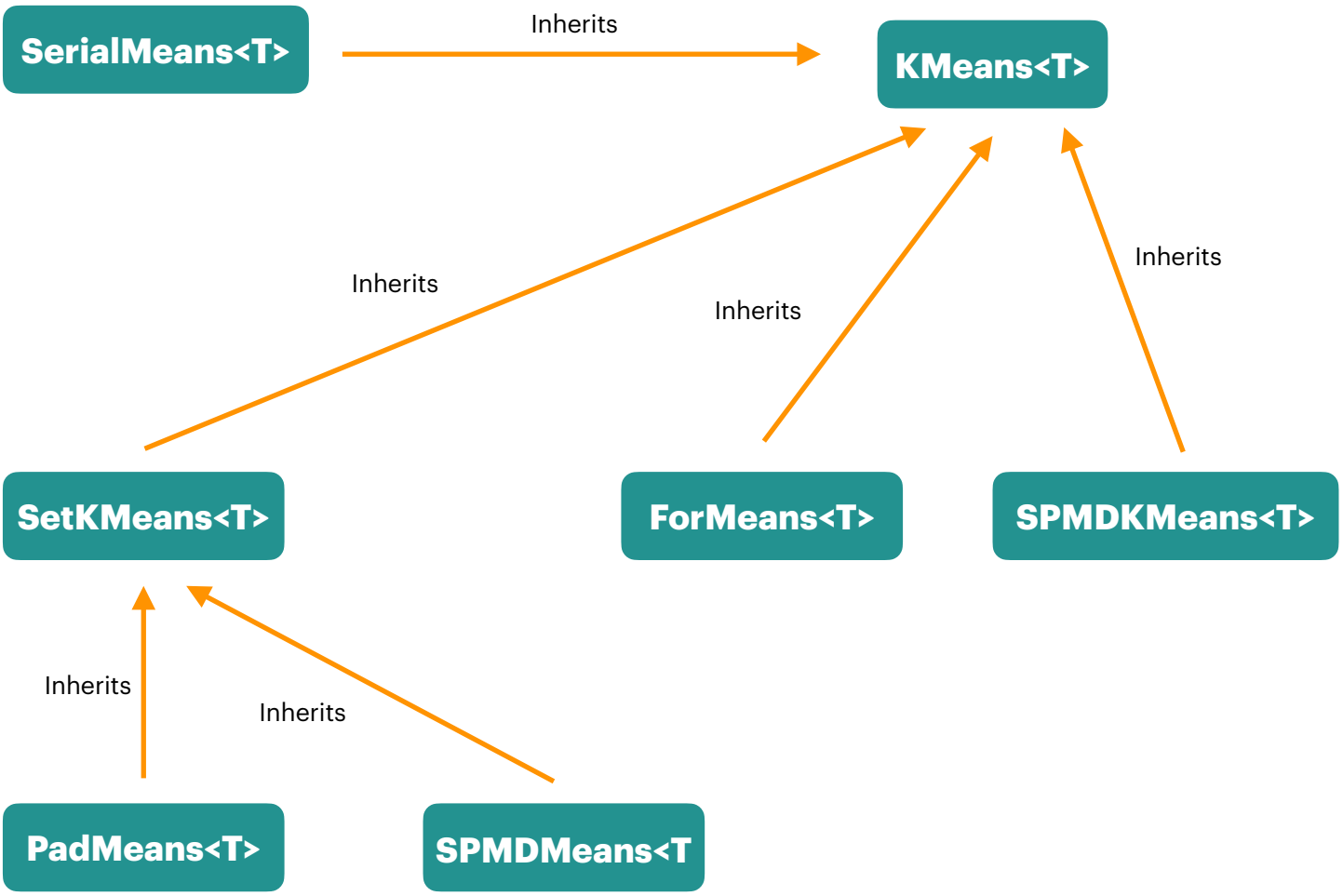
In computing, SPMD (single program, multiple data) is a technique employed to achieve parallelism; it is a subcategory of MIMD. Tasks are split up and run simultaneously on multiple processors with different inputs to obtain results faster. SPMD is the most common style of parallel programming.

So to achieve this approach, We used a 2D Array to store Clusters for each thread and modify each on its running line. This way, we split up our tasks and avoid data races, but in the end, we should merge the results of each thread's cluster. Hence this class inherits from *SetKMeans<T>*.

4.3.5 - ForKMeans<T>

omp for construct was used for several loops inside the algorithm within this approach.

4.3.6 - Observation Until Now



5 - Results

All the approaches were tested on a computer with four cores i7 CPU, and 50000 randomly generated datasets. After running the project and when it is finished, there will be an outputResult.txt file that logs every approach, the final centroid of clusters, and more info. Also, a CSV file for each algorithm will be generated for plotting with a python script.

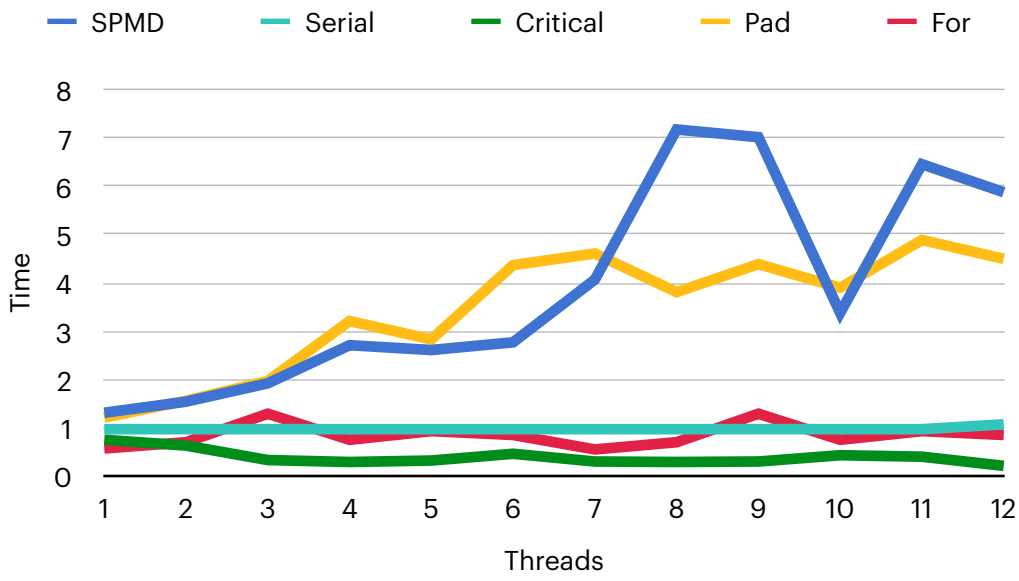
5.1 - Consumed Time

The table below gathers the consumed time (second) per number of threads.

Type \ Thread	1	2	3	4	5	6	7	8	9	10	11	12
SPMD	1.311	1.54	1.92	2.71	2.61	2.77	4.07	7.18	7.02	3.38	6.46	5.88
Serial	1.07	-	-	-	-	-	-	-	-	-	-	-
Critical	0.75	0.63	0.33	0.29	0.32	0.46	0.3	0.29	0.3	0.43	0.4	0.21
Pad	1.21	1.56	1.97	3.21	2.83	4.37	4.61	3.81	4.39	3.90	4.89	4.50
For	0.57	0.70	1.29	0.75	0.93	0.85	0.55	0.70	1.29	0.75	0.93	0.85

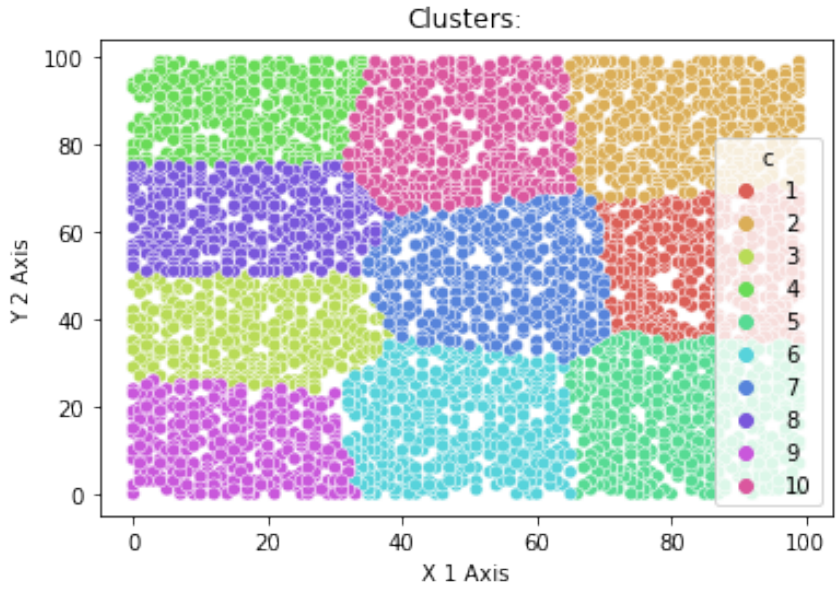
The Result shows that *CriticalKMeans* and *ForKMeans* had the best performance among other approaches.

The final result of consumed time per thread can also be observed from the below chart.



5.2 - Clusters Visualization

The final cluster can be visualized using the *plot_clusters.py* script located at the project's root. There is a CSV file for each approach that can be passed to this script.



6 - References

1. https://www.openmp.org
2. https://en.wikipedia.org/wiki/OpenMP
3. https://www.tutorialspoint.com/what-is-openmp
4. https://nesi.github.io/perf-training/python-scatter/openmp
5. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
6. https://en.wikipedia.org/wiki/K-means_clustering
7. https://www.javatpoint.com/k-means-clustering-algorithm-in-machine-learning
8. https://stanford.edu/~cpiech/cs221/handouts/kmeans.html
9. https://jakevdp.github.io/PythonDataScienceHandbook/05.11-k-means.html
10. https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf
11. https://people.math.umass.edu/~johnston/PHI_WG_2014/OpenMPSlides_tamu_sc.pdf
12. https://docs.microsoft.com/en-us/cpp/parallel/openmp/a-examples?view=msvc-170
13. https://github.com/muatik/openmp-examples/blob/master/data-sharing/main.cpp
14. https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html