**Journal of Computer and Communications, 2014, 2, 15-23**
Published Online September 2014 in SciRes. http://www.scirp.org/journal/jcc
http://dx.doi.org/10.4236/jcc.2014.211002

Scientific Research

# Parallel *K*-Means Algorithm for Shared Memory Multiprocessors

**Tayfun Kucukyilmaz**

Computer Engineering Department, University of Turkish Aeronautical Association, TR06800, Ankara, Turkey
Email: tayfunkucukyilmaz@gmail.com

Received May 2014

## Abstract

**Clustering is the task of assigning a set of instances into groups in such a way that is dissimilarity of instances within each group is minimized. Clustering is widely used in several areas such as data mining, pattern recognition, machine learning, image processing, computer vision and etc. *K*-means is a popular clustering algorithm which partitions instances into a fixed number clusters in an iterative fashion. Although *k*-means is considered to be a poor clustering algorithm in terms of result quality, due to its simplicity, speed on practical applications, and iterative nature it is selected as one of the top 10 algorithms in data mining [1]. Parallelization of *k*-means is also studied during the last 2 decades. Most of these work concentrate on shared-nothing architectures. With the advent of current technological advances on GPU technology, implementation of the *k*-means algorithm on shared memory architectures recently start to attract some attention. However, to the best of our knowledge, no in-depth analysis on the performance of *k*-means on shared memory multiprocessors is done in the literature. In this work, our aim is to fill this gap by providing theoretical analysis on the performance of *k*-means algorithm and presenting extensive tests on a shared memory architecture.**

## Keywords

***K*-Means, Clustering, Data Mining, Shared Memory Systems, High Performance**

## 1. Introduction

Clustering is grouping similar objects according to their resemblances. It is widely used in several areas of computer science such as data mining, pattern recognition, image processing, computer vision, and etc. The *k*-means algorithm is one of the most popular techniques for clustering. Simplicity, speed of convergence, and its easily parallelizable nature makes *k*-means an attractive clustering technique.

Parallelization of clustering techniques is receiving an increasing attention due to ever-increasing data sizes today. Many parallel implementations of various clustering techniques [2] is studied in the literature and *k*-means algorithm is one of them. Being a very simple iterative clustering algorithm that relies on *Lloyd's* iteration, the *k*-means algorithm has an almost "embarrassingly parallel" nature.

During the last two decades, due to the advances in computer networking, heterogeneous and geographically distributed computational resources become available for both scientists and developers. Thus, shared-nothing

**How to cite this paper:** Kucukyilmaz, T. (2014) Parallel *K*-Means Algorithm for Shared Memory Multiprocessors. *Journal of Computer and Communications*, **2**, 15-23. http://dx.doi.org/10.4236/jcc.2014.211002

architectures become the more popular parallel paradigm of this era. The computer science literature followed this trend and most of the efforts on parallelizing the *k*-means algorithm were also based on shared-nothing architectures. However today, with the new advances in GPU technologies huge computational resources are also become available in the form of shared memory multiprocessors. In order to get the most out of the new technologies, careful analysis must be done before developing new parallel solutions to already existing algorithms.

In this paper, we will focus on the *k*-means algorithm; the implementation and analysis of a parallel *k*-means algorithm for shared memory multiprocessors. In Section 2, the previous work on the efforts of parallelizing the *k*-means algorithm is presented. In Section 3, sequential algorithm is mentioned in detail. The parallel algorithm will be presented in Section 4, while extensive analytical analysis will be provided in Section 5. The results of the conducted experiments will be presented in Section 6 and we will conclude in Section 7.

## 2. Related Work

In the scope of data mining, clustering is the task of partitioning a set of instances in such a way that the dissimilarity between instances that belong to the same partition is minimized. *K*-means algorithm is a clustering technique that specifically aims to find disjoint partitions where instances are defined within a numeric domain. The similarity of instances belonging to a partition, namely a cluster, is ensured by defining the cluster with a centroid (center of the cluster), and assigning instances to partitions with respect to their relative euclidean distances to cluster centroids.

One of the first works on the *k*-means algorithm is presented in [3], where the optimality of grouping instances into clusters with respect to their similarities is examined. [4] extends this work, and give a detailed analysis of the algorithm. In the author's own words, "the *k*-means algorithm is an easily programmable and computationally economical" method. By then, numerous studies concerning the *k*-means algorithm are published in the literature. [5] and [2] present extensive surveys about such works.

Due to its simplicity, parallel implementations of the *k*-means algorithm is also studied extensively in the literature. [6] presented a parallel implementation of *k*-means on a shared-nothing architecture using message passing interface (MPI). Their work mainly concentrates on the speed up and scalability aspects of the algorithm. [7] propose another implementation for the same setting using a master-slave paradigm. [8] and [9] presents two variations of the *k*-means algorithm for shared-nothing architectures where, in both studies *MapReduce* framework is used. In [10], authors propose a generic methodology for creating parallel data mining algorithms for shared-nothing multiprocessors including the *k*-means algorithm.

Most studies in the literature consider the iterative nature of *k*-means algorithm as the computational bottleneck. There are also numerous efforts for improving the performance of parallel *k*-means algorithm in the literature. [11] present an optimal adaptive *k*-means algorithm where the learning rate of the algorithm can be dynamically change. In their work, the authors show that their work can achieve near-optimal clustering solutions. In [12], authors propose a parallelization of the *k*-means algorithm, where two different formulations for centroid recalculation step is given. [13] present a work on a new initialization method for *k*-means algorithm. The proposed method also selects the initial centroids in a parallel fashion. [14] present a pruning method for shared memory multiprocessor systems. In their work, the authors use a method that avoids computation of distances to redundant cluster centroids. In a recent article, [15] present a new method for avoiding the unnecessary distance calculations using triangle inequality technique. They also address the load imbalance problems for their framework when such computations are eliminated.

## 3. Sequential *K*-Means Algorithm

For sake of clarity, this section is divided into two subsections. In the first subsection, the sequential *k*-means algorithm is presented in detail as a preliminary for further discussions. In the second subsection, in order to present a more accurate analysis, the details of the sequential implementation is discussed. Assumptions and restrictions that are put on the experimental setup are clarified for more consistent test results.

The purpose of *k*-means clustering algorithm is to partition instances defined in numeric domains into disjoint clusters with the objective of minimizing the dissimilarities of instances within partitions. The similarity or dissimilarity of two instances is defined as the euclidean or manhattan distance between pairs. More formally, given a set of n instances $I = \{m_i, 0 < i < n\}$, a set of k initial cluster centroids $C = \{c_j, 0 < j < k\}$, and a distance function $D(m_i, c_j)$, *k*-means algorithm minimizes the function:

$$\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} Min\left(Distance\left(m_i, c_j\right)\right) \quad (1)$$

In other words, *k*-means algorithm finds k data points on the instance space such that the *mean square error* (that is, the total distance of all instances to the nearest cluster center) is minimized. The sequential version of the *k*-means algorithm is depicted in **Algorithm 1**.

In order to calculate the minimized solution, {\em k}-means algorithm follows a 4-step process:

- **Initialization**: This step consists of selecting k starting points C = {$c_j$, 0 < j < k} from the instance space. as it will become obvious in the forthcoming steps of the algorithm, the quality of the results highly depend on the quality of this selection. The reason that initialization plays a critical role on the quality of the results of the *k*-means algorithm is twofold. First, finding n clusters with minimal mean square error is NP-complete. The *k*-means algorithm only finds local minima's based on the initial centroid selection. Second, the number of iterations required to reach to a stable solution usually depends on the initial selection of the centroids. Since computational cost of *k*-means algorithm is dominated by the cost of the iterations, selecting more accurate initial centroids also plays an important role in the computational complexity of the overall algorithm.
- **Distance Calculation**: This step is the computational bottleneck of *k*-means algorithm. It involves calculation of finding the nearest cluster center for each instance and computing the distance to it. The naive solution to this step involves computing distances from all instances to all centroids.
- **Centroid Recalculation**: After executing the first two steps, for the given cluster centers, a set of clusters with minimum variance is found. However, at this point centroids, the data points that denote the center of each cluster, has been selected in an arbitrary fashion. Thus, they cannot be seen as the descriptors of the clusters that they belong to. As a consequence, in this step, the centers of clusters are re-calculated. This is done by taking the mean of each attribute for all instances within the cluster. In the literature, two distinct methods are proposed for re-calculating the cluster centers. In the first method, upon finishing distance calculation for an instance, the centroids are recalculated. It is claimed that this method prevents *k*-means algorithm from getting stuck at a local minima, resulting on better global optimization. In the second method, centroid recalculation is followed by the distance calculation step, computing distance calculation for all instances in the dataset. This method is also called *h*-means, providing faster convergence. In this work, the second method will be adopted in order to exploit parallelism better.
- **Convergence**: The clusters found after the above three steps are actually not optimized since the cluster centroids are re-calculated and shifted on the instance space. That is, refinement for individual instances is still possible. In order to find a minimal mean square error steps 2 and 3 must be iterated until the results become stable. The stability condition can be defined threefold. Steps 2 and 3 are iterated until: no instance changes clusters, the total shift of cluster centroids is smaller than some threshold, or a pre-determined number of iterations has passed.

When the above algorithm is examined, *k*-means algorithm has two undesirable properties. First, *k*-means is a local optimization algorithm which cannot guarantee a global optima. The result quality usually depends on the selection of initial centroids. In the literature, there are several studies that concentrate on selecting better centroids for *k*-means algorithm. Second, *k*-means algorithm is iterative which is undesirable when dealing with

```
1:  MSE ← ∞
2:  OldMSE ← 0
3:  Let Instance set be INST = {i_x}_{0<x<n}
4:  Select initial centroids CENT = {c_y}_{0<y<k} as the first k instances in I
5:  Let DIST = {d_z}_{0<z<P} be the distances between i_x and c_z
6:
7:  procedure SEQUENTIAL_K_MEANS
8:      while MSE < OldMSE do
9:          OldMSE ← MSE
10:         for i ← 1, n do
11:             for j ← 1, k do
12:                 Dist_j ← distance(INST_i, CENT_j);
13:             MSE ← minimum(DIST_{j,0<j<k});
14:         for all instances I_i ⊂ INST assigned to cluster i, 0 < i < k do
15:             c_i = ∑ I_i
```

**Algorithm 1.** The pseudocode for the sequential *k*-means algorithm.

large amounts of data. Although the complete stabilization of cluster centroids usually requires a large number of iterations, the convergence rate of *k*-means is what makes it a still-popular data mining algorithm for the problems where the quality of results within some proximity to a global optima is sufficient.

During implementation of the adopted sequential algorithm, several assumptions are made. These are:

- *K*-means is local optimization algorithm whose results heavily depend on the initial centroid selection. In order to test parallel implementation without any bias, the first *k* instances from the dataset are selected as the candidate cluster centroids in all experiments.
- Whether the processed dataset can fit into memory or not would affect the performance of both parallel and sequential algorithm. In this work, our experiments will be based on a disk-based assumption. That is, it is assumed that the dataset is assumed to be larger than the total memory and reside on the physical disk.
- Disk access latency could be a detrimental factor on the performance for both parallel and sequential implementations. In our experiments, each time the disk is accessed 1000 instances are read from the disk. In the parallel implementation, the data retrieved after each disk access is assigned as a task for the shared memory multiprocessors.

## 4. The Parallel Implementation

In this section, implementation of a *k*-means algorithm based on shared memory multiprocessors on a disk-based setting is discussed. For this purpose, first a brief discussion about the shared memory architecture is given, then, the parallel implementation is presented in detail.

### 4.1. The Shared Memory Architecture

A shared memory architecture is a multi-processor computer system, where multiple processors can access a single block of memory simultaneously. In this architecture, processing units are connected to a single physical memory block via a shared bus or a switching network.

The basic advantage of a shared memory architecture is twofold: First, there is negligible cost associated with inter-process communication. That is, processors can use the physically-shared memory to pass information to other processors. Second, shared memory systems prevent the need for creating redundant replications of data between processors. Since processors can physically share information, there is no need to store redundant/ unused copies of data partitions. In fact, any data assigned to a processor is also accessible by all other processors within the system.

However, shared memory architectures also suffer from some serious drawbacks. First in many state-of-the-art systems, a processor has an associated cache. In a shared memory architecture, coherence between the physically shared memory and CPU caches must be maintained. Ensuring this coherence is a candidate for becoming a bottleneck of the performance of the system. Also, shared memory systems are not scalable in terms of the number of processors. Increasing the number of processors would either create a bottleneck in the interconnection network that is connecting the processors, or create performance degradation due to contention when multiple processors try to access to the same memory location. Last, shared memory systems are also not scalable in terms of problem size. Since such systems contain a single memory block, increasing the total memory is usually not possible for these systems when the problem size grows larger.

On shared memory systems, in order to maintain constant coherency of data the widely accepted method is to use a locking mechanism on the memory locations that are accessible by multiple processors. By this way, processors read and write globally available data in a mutually exclusive way. However, locking the contents within the shared memory will degrade the parallel system performance since multiple processors have to sequentially complete tasks if access to such memory blocks is required.

### 4.2. *K*-Means Algorithm Based on Shared Memory Multiprocessors

The parallel version of the k-means algorithm is depicted in **Algorithm 2** below. Similar to the sequential algorithm discussed in Section 3, the parallel k-means algorithm can be divided into 4 steps. These steps are initialization, distance calculation, centroid recalculation, and convergence. The first step of *k*-means is initialization of centroids. Potentially, this first step is not parallelizable; each centroid must be assigned globally. However, by exploiting the shared memory architecture, each processor could be assigned to the task of selecting *C/P* centroids. At the end of this operation a set of globally accessible centroids will be decided among processors.

```
1: MSE ← ∞
2: OldMSE ← 0
3: Let MSE_p be the partial MSE computed by processor p
4: Let Instance set be INST = {i_x}_{0<x<n}
5: Select initial centroids CENT = {c_y}_{0<y<k} as the first k instances in I
6: Let DIST_p ={ d_z}_{0<z<k} be the distances between i_x and c_z on processor p
7:
8: procedure PARALLEL_K_MEANS
9:     while MSE < OldMSE do
10:        OldMSE ← MSE
11:        for i ← 1, P do
12:            Assign n/P instances to processor i
13:            Assign k/P centroids to processor i
14:        for each Processor p ∈ P do
15:            for i ← (i − 1) ∗ (n/P), i ∗ (n/P) do
16:                for j ← 1, k do
17:                    Dist_{p,j} ← distance(INST_i, CENT_j);
18:                MSE_p ← minimum(DIST_{j,0<j<k});
19:        for each Processor p ∈ P do
20:            for all instances I_i ⊂ INST assigned to cluster i, 0 < i < k do
21:                c_i = ∑ I_i/ number of instances assigned to cluster i
22:        WAIT ALL PROCESSORS and LOCK MEMORY MSE
23:        for each Processor p ∈ P do
24:            MSE += MSE_p;
25:        WAIT ALL PROCESSORS and UNLOCK MEMORY MSE
```

**Algorithm 2.** The pseudocode for the parallel *k*-means algorithm.

The computational bottleneck of *k*-means algorithm is step 2, where each processor computes a distance for all instances assigned to itself and all cluster centroids. If Euclidean distance function is used, each distance calculation will cost *2d+c* floating point operations where, the first term constitutes to calculating the distance, and the second term constitutes to finding the closest centroid. This step of the algorithm can be easily parallelized by assigning *n/P* instances to each processor, where processors compute the closest centroid each instance in a parallel fashion.

After concluding step 2, each instance is assigned to a cluster according to their proximity to each cluster centroid, and thus, membership information of all instances to each cluster is established. In step 3, *k*-means algorithm computes new cluster centroids by calculating the total mean square error, in other words the relative distance, of all cluster instances to the initial cluster centroids. Although calculation of individual mean square error for each individual instance, or each individual batch can be done in parallel, the task of calculating the new cluster centroids must be done in parallel. Thus after each iteration of centroid recalculation, the processors should be synchronized. In this work, in order to maximize the parallelization, the data is divided into computational batches called threads, where, for each batch a mean square error is calculated in parallel. After all threads finish computation, parallel tasks got synchronized in order to compute new cluster centroids. For this purpose again, cluster centroids are assigned to one of the existing threads, and computations are carried out in parallel. However, convergence condition must be executed sequentially.

Although shared memory systems do not induce any communication costs to a process there are two additional costs for executing a parallel task on these systems. First, allocating system resources to a new thread require additional time. Also if at any time, there is more threads than the number of processors, the computational resources must be shared between threads. Thus, a successful parallelization should either: take into account computational power sharing, or make sure that the maximum number of threads active at any is not larger than the total number of processing components. The main cost component of a shared memory system is due to the need of maintaining concurrency over the globally accessible data. Since it is not acceptable for a shared data to be manipulated by multiple processors at the same time, processes must maintain a lock on the data during any manipulation. Consequently, operations that need to be carried out on shared data would inevitably be serialized. In this work, for maximizing the performance of the parallel *k*-means algorithm, each instance and each cluster centroid is assigned to one of the processors. In order to reduce the processor-wait time during synchronization at step 3, computational load balancing is also ensured. The number of instances and centroids assigned to each processor are kept at roughly equal sizes.

## 5. Numeric Analysis

In this section, in order to compare the theoretical results with experimental results an in-depth analysis of the sequential algorithm is provided. The notation used throughout this section is as follows:

- $n$: The number of instances in the dataset.
- $d$: The number of attributes for each instance.
- $k$: The number of clusters.
- $T$: The total number of iterations required for $k$-means algorithm to finalize.
- $P$: The total number of processors.
- $T_{flop}$: The time required to execute a floating point operation.
- $T_{thread}$: The time required to allocate and manage a thread.
- $z$: The total number of instances assigned to a thread during parallel execution.

The sequential algorithm consists of 4 steps. The first step consists of selecting $k$ cluster centroids. In the second step, the algorithm first computes the distance for each instance with each centroid. This operation involves $nk$ pair wise distance calculation, where each distance calculation requires $2d$ floating point operations. At step 3, the algorithm compares the computed distances for each instance-centroid pair and find the closest centroid to each instance. This comparison will require $nk$ floating point operations to be performed. Before, checking whether the algorithm terminates or not, the algorithm computes the means square error for the whole dataset and calculate the new cluster centroids. The former operation would take $kd$ floating point operations and the latter would require $nd$ floating point operations, given that all the previous steps are already carried out.

Assuming that it would take $T$ iterations for the $k$-means algorithm to terminate, the total computational complexity of the sequential algorithm can be depicted as in Equation (2):

$$\left(2nkd + nk + nd + kd\right) \times T \times T_{flop} \tag{2}$$

If the dataset is sufficiently large, the fourth term of the above equation would incur negligible overhead. thus the time complexity of the sequential $k$-means algorithm would converge to:

$$\left(2nkd + nk + nd\right) \times T \times T_{flop} \tag{3}$$

Equation (3) summarizes the execution time of the sequential $k$-means algorithm. For a shared memory parallel implementation, the same operations will be carried out on $P$ processors. However, all such operations will be carried out as batches using threads, and every thread must write partial mean square error contributions to the disk which takes $kd$ floating point operations per thread per iteration. Thus, the execution time for a shared memory implementation of the $k$-means algorithm can be formulized as:

$$\left[\left(2nkd + nk + nd\right)/P + nkd/z\right] \times T \times T_{flop} \tag{4}$$

Since $z$ is a constant and should be selected fairly large, one may assume that $z >> P$. Thus the dominating factor of this equation would still be the first partial term.

Another cost of the shared memory parallel implementation is the thread allocation and scheduling in the event queue. Taking this into account, the computational complexity of a shared memory parallel system can be summarized as $T_{par} = T_{comp} + T_{thread}$, where $T_{comp}$ is derived from Equation (4) and $T_{thread}$ is proportional to the number of active threads per iteration since at the end of each iteration the algorithm should syncronize for all processing nodes for updating cluster centroids. That is,

$$\left[\left(2nkd + nk + nd\right)/P + nkd/z\right] \times T \times T_{flop} + n/z \times T \times T_{thread} \tag{5}$$

## 6. Experiments

During our evaluations we conducted extensive experiments on both sequential and the parallel implementations of the k-means algorithm. in these experiments, both versions are tested on an Intel Nehalem-EX Xeon 7550 with 8 cores running at 2.0 GHz.

The dataset and the instances are synthetically created for our experiments. All attributes for each instance in the dataset is a floating point with 3 digits of precision. If not mentioned during this section, the default setting is as follows: The dataset consists of 10,000 instances, where each instance has 30 attributes. The default $k$-means algorithm is run for finding 3 clusters.

During our experiments, we put limits to both number of active threads and the total number of instances a thread can process. For the sequential implementation, the whole algorithm is run as a single thread on a single processor. For the parallel implementation the maximum number of active threads is limited by the number of processors, and each such thread has a capacity to process maximum 5.000 instances. Every time the capacity of a thread is completed, the thread is killed and a new thread is created.

**Figure 1** shows the performance of sequential *k*-means algorithm for varying data sizes. The experiment is conducted for datasets of size 10, 20, 40, 100, 200 thousand and 1 million instances. Our experiments show that when the size of the dataset increases the execution time also increases proportionally, conforming the theoretical results.

In order to evaluate the performance of the parallel *k*-means algorithm we did 3 sets of experiments. These are: varying thread capacity, varying number of clusters, and varying number of attributes.

**Varying thread capacity**: In order to evaluate whether the performance of parallel k-means algorithm conforms our theoretical results, we used two different thread capacities. In the first set of experiments the thread capacity is set to 5.000 and in the second set, the thread capacity is set to 10.000 instances.

**Figure 2** presents the results of our experiments. When the total number of instances is less than 40.000, the parallel algorithm perform almost constant. However when the total number of instances is greater than 40.000 the execution time start to increase linearly for both cases. The reason for this behavior is that, for small number of instances since all threads are not active adding new instances result in creation of new threads. Since all threads run in parallel, we cannot observe any performance loss. However, for instances above 40.000, all threads become saturated and instances need to wait threads to terminate before getting processed.

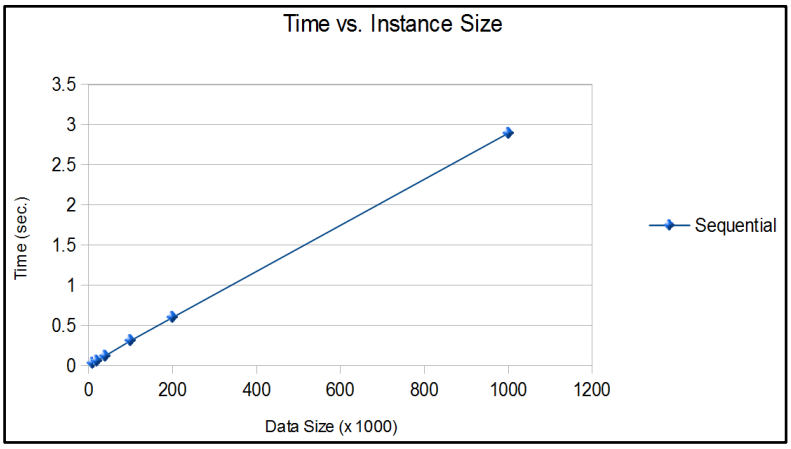


**Figure 1.** Performance of sequential k-means with respect to data size.

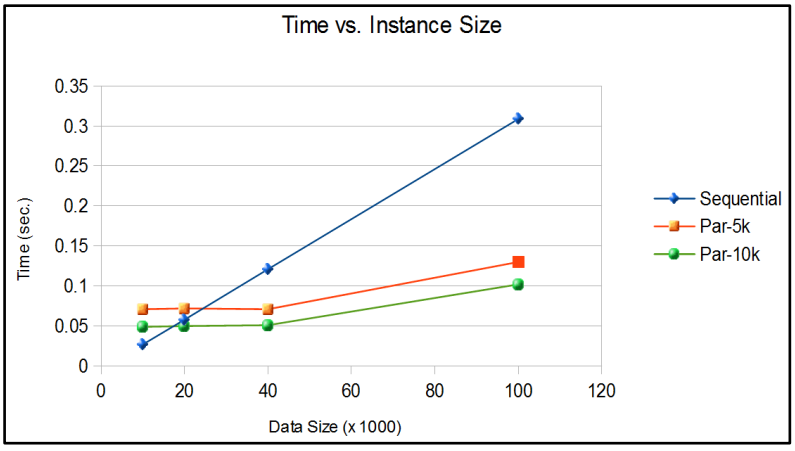


**Figure 2.** Performance of sequential and parallel k-means algorithm with varying thread capacities.

Also, the performance of experiments with 10.000 thread capacity is slightly better than experiments with 5.000 thread capacity, since less threads are created, and thus less time is consumed for allocating new threads, and locking global data. Finally, as expected the execution the increase rate for parallel algorithm is less than the sequential version. In fact, the parallel algorithm's execution time scales with 1/8 of the sequential version.

**Varying number of clusters**: Our theoretical results show that the execution time of k-means algorithm also depend on the number of clusters. In order to evaluate the effects of varying number of clusters we conducted experiments using 2, 3, 4, 5, and 10 clusters.

**Figure 3** presents the results of these experiments. Our theoretical analysis results in 2 conclusions. First, although the performance of the k-means algorithm depends on the number of clusters, the execution time increase rate should be lesser than that of varying number of instances. Second, the theoretical analysis on the parallel implementation shows that in the parallel version, the speed up proportional to the number of processors should be expected.

The experimental results conform to our theoretical analysis. However, due to overheads the performance of the parallel implementation cannot achieve the performance of the sequential implementation if the number of clusters is less than 10.

**Varying number of attributes**: Our theoretical results show that the effect of varying number of attributes should be similar to that of varying number of clusters. In order to test this claim we conducted experiments with varying number of attributes. We conducted 5 experiments, where the number of attributes is set to 20, 30, 40, 50, and 100 respectively.

As stated in **Figure 4**, our experiments show that, the performance of the parallel algorithm increases with a shallower slope than that of the sequential algorithm. And when the total number of attributes is around 90 the parallel implementation start to perform better than the sequential version.

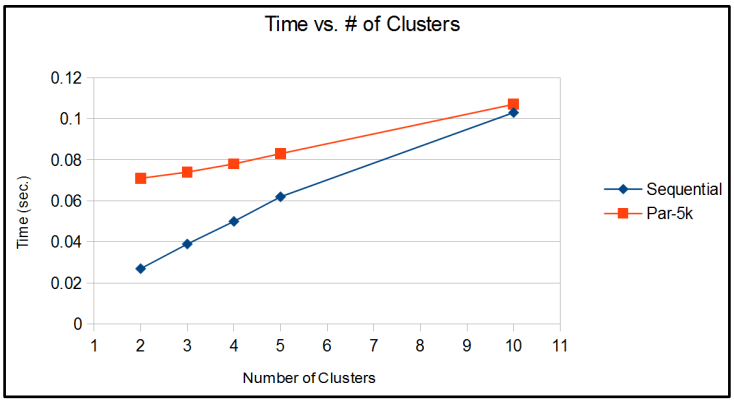


**Figure 3.** Performance of parallel and sequential k-means algorithms with varying number of clusters.

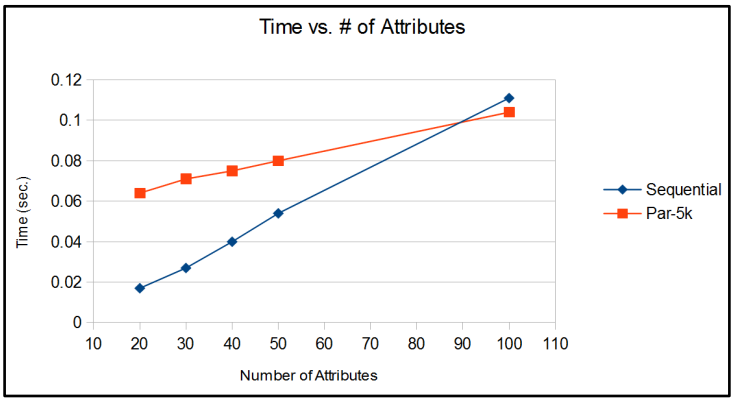


**Figure 4.** Performance of parallel and sequential k-means algorithms with varying number of attributes.

Comparing these results with the results presented in **Figure 3**, it is possible to see that the results also conform to the theoretical results; the performance of the parallel algorithm scales similarly with both varying number of cluster and attributes.

## 7. Conclusions

In this work, a parallel k-means algorithm on shared memory multiprocessors is presented and detailed analysis has been performed. In this work, our contributions are as follows:

A detailed implementation of the parallel k-means algorithm is presented. This algorithm is optimized in a way that the total number of memory locks performed by the processors is minimized. In our algorithm, the centroid recalculation step is divided into two parts. In the first part, each processor finds partial mean square errors for all centroids in parallel, eliminating the need for global locks on the memory, and in the second step the partial results are accumulated where processors synchronize. To the best of our knowledge this technique has not been used in the literature before.

We give a detailed analysis of both sequential and shared memory based parallel implementations. This analysis show that in terms of computational complexity both number of instances, number of clusters, and the number of dimensions on the dataset holds almost equal importance.

We evaluate our theoretical results by performing detailed experiments. The results show that our theoretical results hold if the dataset is sufficiently large.

## References


[1] Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Yu, P.S., Zhou, Z.-H., Steinbach, M., Hand, D.J. and Steinberg, D. (2007) Top 10 Algorithms in Data Mining. *Knowl. Inf. Syst*., **14**, 37.

[2] Witten, I.H., Frank, E. and Hall, M. (2011) Data Mining: Practical Machine Learning Tools and Techniques. 3rd Edition, Morgan Kaufmann, ISBN: 9780123748560.

[3] Cox, D.R. (1957) Note on Grouping. *Journal of Amer. Statist. Assoc*., **52**, 543-547.

[4] McQueen, J. (1997) Some Methods for Classification and Analysis of Multivariate Observations. In: *Proc. of* 5*th Berkeley Symp. on Mathematical Statistics and Probability*, 173-188.

[5] Hartigan, J.A. (1975) Clustering Algorithms. Wiley Publishing Ltd., Chichester.

[6] Dhillon, S. and Modha, D.S. (2000) A Data Clustering Algorithm on Distributed Memory Multiprocessors. In: *Know. Data Discovery* (*KDD*), *Lecture Notes in Computer Science*, **1759**, 245-260.

[7] Kantrabutra, S. and Couch, A.L. (2000) Parallel k-Means Algorithm on NOWs. *Medtec Technical Journal*, **1**, 243-249.

[8] Das, A., Datar, M., Garg, A. and Rajaram, S. (2007) Google News Personalization: Scalable Online Collaborative Filtering. In *WWW*, 271-280.

[9] Ene, A., Im, S. and Moseley, B. (2011) Fast Clustering using MapReduce. In: *Know. Data Discovery* (*KDD*), 681-689.

[10] Jin, R. and Agrawal, G. (2003) Combining Distributed Memory and Shared Memory Parallelization for Data Mining Algorithms. In: *HPDM*: 6*th International High Performance, Pervasive, and Data Stream Mining.*

[11] Chinrungrueng, C. and Sequin, C.H. (1995) Optimal Adaptive k-Means Algorithm with Dynamic Adjustment of Learning Rate. *IEEE Transactions on Neural Networks*, **6**, 157-169. http://dx.doi.org/10.1109/72.363440

[12] Stoffel, K. and Belkoniene, A. (1999) Parallel *k/h*-Means Clustering for Large Data Sets. In *Euro-Par* '99 *Proceedings of the* 5*th International Euro-Par Conference on Parallel Processing*, 1451-1454.

[13] Bahmani, B., Moseley, B., Vattani, A., Kumar, R. and Vasilvitskii, S. (2012) Scalable K-Means++. In: *Proceedings of the VLDB Endowment*, **5**, 622-633.

[14] Gursoy, A. and Cengiz, I. (2001) Parallel Pruning for K-Means Clustering on Shared Memory Architectures. *Lecture Notes in Computer Science*, **2150**, 321-325. http://dx.doi.org/10.1007/3-540-44681-8_45

[15] Kumar, J., Mills, R.T., Hoffman, F.M. and Hargrove, W.W. (2011) Parallel *k*-Means Clustering for Quantitative Ecoregion Delineation Using Large Data Sets. *Proceedings of the International Conference on Computational Science, ICCS*, **4**, 1602-1611.