International Journal of New Computer Architectures and their Applications (IJNCAA) 8(4): 168-178
The Society of Digital Information and Wireless Communications, 2018 ISSN 2220-9085 (Online); ISSN 2412-3587 (Print)

Parallel *k*-means Clustering Algorithm on SMP

Athari M. Alrajhi and Soha S. Zaghloul, PhD
College of Computer & Information Sciences, Department of Computer Science, King Saud University, Riyadh
437203971@student.ksu.edu.sa
smekki@ksu.edu.sa

ABSTRACT

The *k*-means clustering algorithm is one of the popular and simplest clustering algorithms. Due to its simplicity, it is widely used in many applications. Although *k*-means has low computational time and space complexity, increasing the dataset size results in increasing the computational time proportionally. One of the most prominent solutions to deal with this problem is the parallel processing. In this paper, we aim to design and implement a parallel *k*-means clustering algorithm on shared memory multiprocessors using parallel java library. The performance of the parallel algorithm is evaluated in terms of speedup, efficiency and scalability. Accuracy and quality of clustering results are also measured. Furthermore, this paper presents analytical results for the parallel program performance metrics.

KEYWORDS

k-means, Clustering, SMP, Parallel Java, Parallel Programming, pj2, Shared Memory Multiprocessors.

1 INTRODUCTION

Data mining clustering techniques are unsupervised learning because they don’t use predefined class labels. The clustering goal is to obtain meaningful groupings of objects based on a measure of similarity such that all objects in one group are similar to each other and different from the objects in other groups. Cluster analysis has been widely used in data recovery, web and text mining, image segmentation and pattern recognition. Therefore, several clustering algorithms have been developed. *k*-means is one of the popular partial clustering algorithms [1]. The idea of *k*-means is based on dividing datasets into k number of groups (clusters) such that the squared error between the mean of a cluster and the data points in the cluster is minimized. The mean of a cluster is called *centroid*. The initial centroids are chosen randomly one for each cluster. Then, each point or object belongs to the cluster which has the nearest centroid by computing the Euclidian distance between the point and each centroid. These centroids are updated based on means of each cluster which assign as a new centroid. The assignments and updates are repeated until each centroid remains the same (convergence criterion) [2].

Although *k*-means is capable of dividing the problem domain into smaller parts, it suffers an increase in computation time as the size of the dataset becomes very large. Therefore, an additional technique like parallel processing, to accelerate the computation process is required. Parallel programming can divide the program tasks into smaller independent parts with the aim of running them on multiple processors simultaneously [3]. So, finding those independent parts to reduce the computational time is a challenging issue.

In this paper, we aim to study the parallel *k*-means algorithm and examine its performance on one of the parallel computer architectures called shared memory multiprocessor (SMP).

The rest of this paper is structured as follows; Section 2 discusses the most related work to the problem in-hand. In section 3, a detailed design of sequential and parallel *k*-means clustering algorithm is described. Section 4 presents the results of implementing both sequential and parallel versions. The analytical results are introduced in section 5. Finally, section 6 contains the conclusion and potential future work.

2 RELATED WORK

Parallel processing of k-means clustering algorithm has been able to attract the attention of many researchers around the world. They used different parallel programming models and various techniques in order to achieve a high performance and less computational time. One of the recent studies on the k-means algorithm is presented by Kucukyilmaz [4]. In this study, a parallel k-means algorithm is implemented on shared memory multiprocessors with 8 cores. Extensive experiments are conducted with varying number of instances, clusters and attributes to illustrate the impact of them on the performance. The results show that the previous parameters hold almost equal importance. These results are obtained by comparing the theoretical results with experimental results. Although this work shows a detailed implementation of the algorithm and a good analysis of the results, no evaluation metrics for the parallel program are used.

In another study, message passing interface is used for parallelizing k-means on distributed memory paradigm in [5]. In this work, Kantabutra and Couch proposed a technique to improve the performance in terms of time complexity. Using the evaluation measures, the experimental results show that their technique achieves 50% efficiency of time complexity. In the context of message passing, Ramesh et al. [6] implemented parallel k-means for cluster large agricultural dataset. Using a varying number of data size and clusters, the results prove that the parallel algorithm achieves more efficiency and time complexity than the sequential algorithm.

In [7], Farivar et al. proposed an algorithm to implement k-means clustering on an NVIDIA GPU using CUDA. The dataset consists of 1 million instances, and the number of clusters is 4000. For an objective comparison, different platforms are used, and consequently, different speed improvement is achieved. The results suggest that the speed performance is increased up to 13x and 68x for each platform compared to the PC implementation. CUDA architecture is also used in [8]. In this work, Wu and Hong presented an efficient CUDA-based k-means with load balancing using the triangle inequality. Through extensive experiments, the algorithm achieves better efficiency as compared to CPU-based k-means algorithms. As a result, improved performance in terms of speed and scalability is achieved. In the same way, Kumar et al. [8] used the triangle inequality to decrease the unnecessary distance calculations. In addition, they solve the problem of load imbalance which is related to their framework when these computations are avoided.

3 DESIGN AND IMPLEMENTATION

This section represents the heart of this paper where the aspects of k-means algorithm design are discussed. Furthermore, the inputs and outputs of the algorithm are illustrated in section 3.1. Detailed design steps are presented in section 3.2. Both sequential and parallel versions of the k-means algorithm are designed and discussed in sections 3.3 and 3.4 respectively.

3.1 Inputs and Outputs

The inputs of the k-means algorithm are:

- Dataset of n 2-dimensional data points.
- k value which indicates the number of clusters.

The output of the k-means algorithm is:

- k clusters, each one includes a set of points.

3.2 k-Means Design

As mentioned in section 1, k-means is one of the popular and simplest clustering algorithms that partitions a dataset into k groups by minimizing the sum of squared error (SSE) between the mean of a cluster and the data points in the cluster. The algorithm starts with k initial centroids and works iteratively to assign each point to one of the k clusters based on feature similarity until a convergence criterion is met. More formally, given a set of n d-dimensional data points $X = \{x_i, 0 < i < n\}$, a set of k initial centroids $C = \{c_j, 0 < j < k\}$, and a mean of each cluster μ_j, our goal is to minimize SSE as follows [9]:

$$SSE = \sum_{j=1}^{k} \sum_{x_i \in C_j} \left\| x_i - \mu_j \right\|^2 \qquad (3.1)$$

The algorithm of k-means is described in the following four steps:

1. Initialization:

This step involves selecting k initial centroids $C = \{c_j, 0 < j < k\}$ from the instance space, where k is the number of clusters [10]. There are many methods proposed for selecting initial centroids. One common way is to randomly either choose k actual data points from the dataset or generate k virtual data points. The actual data point is a point that comes directly from the dataset. In contrast to the actual data point, a virtual data point is a point that not related to any point in the actual dataset [11].

2. Distance Calculation:

This step includes calculation of finding the closest centroid for each data point and computing the distance to it. There are many distance metrics to measure the distance between centroids and data points such as Manhattan, Euclidean distance, cosine similarity, correlation, etc. Euclidean distance is often used as a measure of distance for k-means clustering [12]. The distance between x_i and c_j is given by:

$$d(c_j, x_i) = \sqrt{\sum_{t=1}^{d} \left(c_{j,t} - x_{i,t}\right)^2} \qquad (3.2)$$

3. Centroid Recalculation:

After assigning each point x_i to the closest cluster c_j , the centroids are re-calculated by compute the average of all points within the cluster as follows [1]:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i \qquad (3.3)$$

4. Convergence:

The clusters obtained after the previous steps are actually not optimized. In order to find a minimal SSE, steps 2 and 3 must be repeated until the results become stable. The stability condition is called convergence criterion and can be specified in multiple ways such as the convergence criterion is met after a fixed number of iterations or when centroids remain the same [10].

It is worth noting that the time complexity of the k-means algorithm is $O(n^* k^* i^* d)$, where:
n: number of data points (instances) in the dataset.
k: number of clusters.
i: number of iterations.
d: number of dimensions.

3.3 Sequential k-Means

In this study, sequential model of the k-means clustering algorithm is designed with the aim of calculating the speedup gains of parallel implementation which express the impact of parallelization. As mentioned in section 3.2, the first step of k-means algorithm is selecting k initial centroids randomly. Because the quality of the clustering results highly depends on the quality of this selection, choosing good initial centroids can play an important role in obtaining better results as well as reducing the computational complexity of the algorithm [10]. The results in [11] show that the virtual points perform better than the actual points. Therefore, in this research, we will generate k virtual points randomly as initial centroids. To guarantee that the generated virtual points values don't exceed the values range of the points in the dataset and consequently get good results, the random virtual points should be scaled using any normalization technique. The sequential algorithm is explained in figure 3.1.

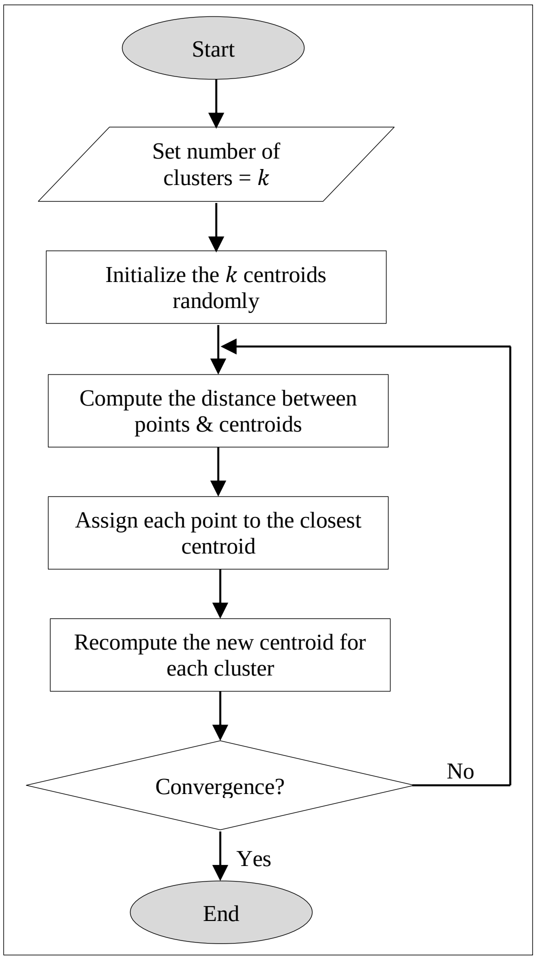
Start
Set number of clusters = k
Initialize the k centroids randomly
Compute the distance between points & centroids
Assign each point to the closest centroid
Recompute the new centroid for each cluster
Convergence?
No
Yes
End

Figure 3.1 Sequential k-means Algorithm

3.4 Parallel k-Means

Designing a parallel model of the k-means on SMP is a big challenge because k-means is inherently sequential. Furthermore, the challenge lies not only in the design of a parallel algorithm, but the parallel algorithm must be superior to the serial algorithm in terms of execution time reduction which is the main objective of parallelism. Thus, we have to look for the independent parts of the algorithm that takes a long time of execution, and then execute them in parallel. These parts are often related to computations. When we look at the four steps of k-means, we find that the first step (initialization) cannot be parallelized, because it is too simple, and each centroid must be initialized globally. The computational bottleneck of the algorithm is the second step, where the distance between each point and centroid is computed, especially if there is a large number of points. This step can be parallelized by dividing the data points among processors and then, making each thread represents a point. Thus, each point is assigned to one thread to compute the nearest centroid for each point in a parallel manner. After that, each thread stores its result (closest centroid) in its own per-thread variable. At the end of this step, a reduction parallel pattern is used to collect the all threads results according to the closest centroid. The per-thread variables have to be reduced together into one overall variable to be ready to the next step. Figure 3.2 explains this idea. In the third step, new centroids for each cluster are re-computed which can be implemented in parallel, since each thread will represent a centroid. For the last step, the loop cannot be parallelized since an iteration relies on the results of the previous one. Figure 3.3 illustrates the synchronous parallel k-means algorithm on SMP. Figure 3.4 shows a flowchart of the parallel k-means algorithm.

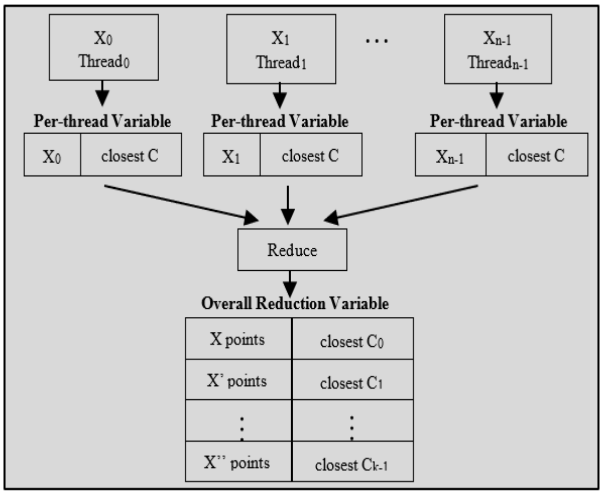

Figure 3.2 Parallel Reduction

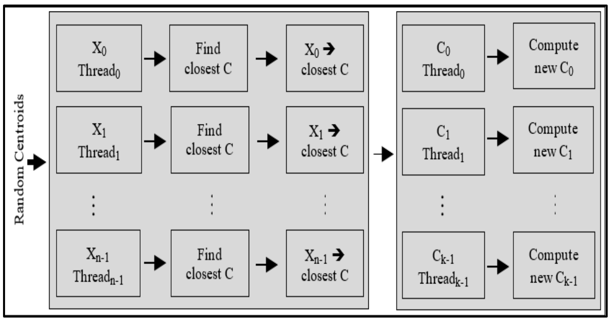

Figure 3.3 Synchronous Parallel *k*-means on SMP

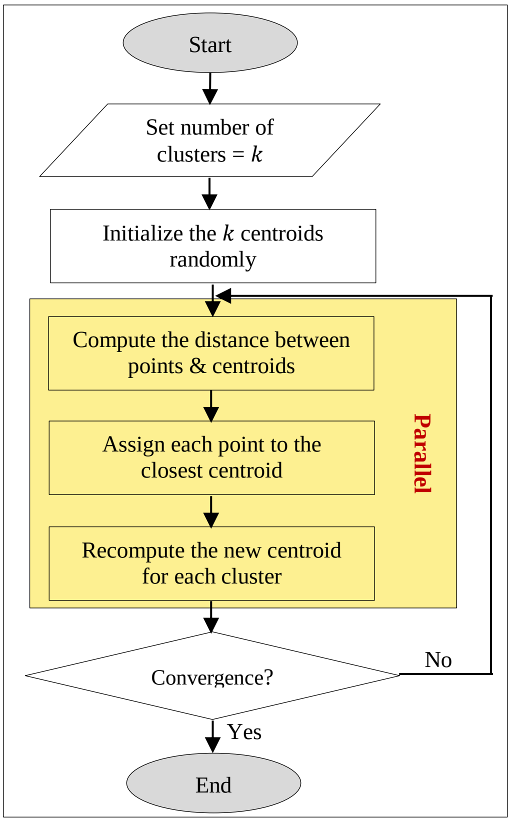
Start
Set number of clusters = k
Initialize the k centroids randomly
Compute the distance between points & centroids
Assign each point to the closest centroid
Recompute the new centroid for each cluster
Parallel
Convergence?
No
Yes
End

Figure 3.4 Parallel *k*-means Algorithm

4 EXPERIMENTS AND RESULTS

To study the performance of the parallel *k*-means on SMP, some experiments are conducted on both parallel and sequential versions of the algorithm.

In section 4.1, the software and hardware specifications are illustrated. The main performance measurements to evaluate our work are described in section 4.2. Further, the detailed experiments and results are explained and presented numerically and graphically in section 4.3.

4.1 Hardware and Software Environment

Sequential and parallel versions of the *k*-means algorithm are implemented on Intel core i7 with quad-cores running at 2.30 GHz and supports hyperthreading. So, the system operates like it has 8 cores because each core can handle 2 threads. The operating system is Linux Ubuntu 16, and the programming language is Java using NetBeans IDE. Parallel Java 2 (pj2) library is used for threads management. Further, it supports shared memory parallel programming on multicore computers [13].

4.2 Evaluation Measurements

In this section, evaluation metrics are presented and explained to evaluate our work. These measures are divided into two categories. The first category includes the measures that evaluate the quality and the accuracy of *k*-means. The second category involves the measures that related to the performance of parallel programs. The following sections describe these measures.

4.2.1 Quality and Accuracy of Clustering Results

To evaluate the quality of *k*-means clustering results, Sum of Squared Error measure is used to handle this issue. Furthermore, the accuracy of clustering is computed as follows:

1. **Sum of Squared Error:**

When centroids are initialized randomly, different runs of *k*-means lead to different results in total SSEs because *k*-means algorithm only converges to the local minimum. As mentioned earlier, the quality of the clustering results relies on chosen of the centroids. So, choosing poor initial centroids may cause poor clustering results with higher SSE. One way to address this problem is to perform multiple runs with multiple different initial centroids and choose the one that gives the smallest squared error [9], [1]. Sum of squared error is a common measure to evaluate *k*-means. This evaluation defines a good measure for the homogeneity of the clustering results. SSE is given by:

$$SSE = \sum_{j=1}^{k} \sum_{x_i \in C_j} \left\| x_i - \mu_j \right\|^2 \qquad (4.1)$$

2. Accuracy of Clustering:

In order to compute the accuracy of points clustering, we have to check that all points used are included in the clusters. Simply, we will sum the number of points in all clusters and compare it with the number of points in the dataset. If the sum is equal to the number of points, then all the points have been clustered. Otherwise, some points are un-clustered (k-means is a crisp (hard) clustering algorithm, it assigns each data point to one cluster exclusively. So, there is no choice for some points to be duplicated). Figure 4.1 shows a flowchart of checking steps. The accuracy of clustering is computed by dividing the number of clustered points by the total number of points and multiplying the answer by 100. The accuracy of clustering is given by the following formula:

$$Acc_{clst} = \frac{cp}{tp} \times 100 \qquad (4.2)$$

Where:
cp: the number of clustered points.
tp: the total number of points.

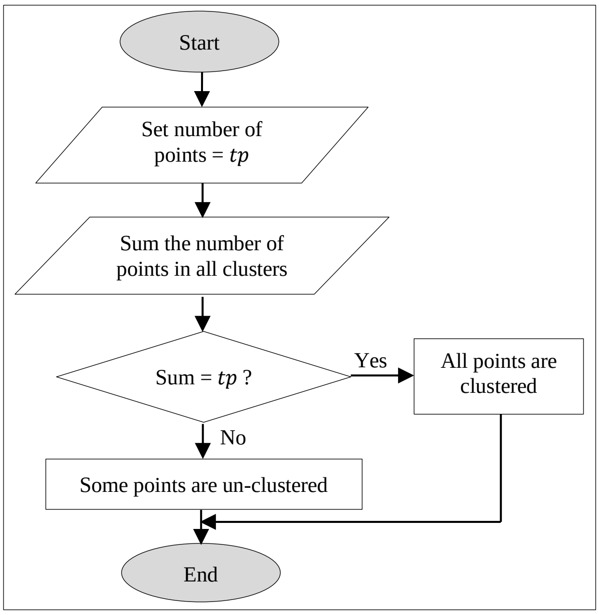
Start
Set number of points = tp
Sum the number of points in all clusters
Sum = tp ?
Yes
All points are clustered
No
Some points are un-clustered
End

Figure 4.1 Checking the Clustering of Points

4.2.2 Performance Metrics of Parallel Program

It is important to study the parallel programs performance with the aim of determining the best algorithm, evaluating the efficiency of a parallel algorithm, examining the benefits from parallelism, and evaluating parallel hardware platforms. Some main metrics are used to analyze the parallel programs performance as below:

1. Speedup:

To see the benefit of parallelism, it is important to know how much speed gain is achieved by parallelizing a program over a sequential implementation. Therefore, a comparison with the running time of a sequential version of a given application is very important to analyze the parallel version. Speedup can be defined as the ratio of the execution time of the sequential version of a given program running on one processor to the execution time of the parallel version running on P processors with problem size n [14]. Speedup is given by the following formula [15]:

$$Speedup = \frac{S_{seq}(n, 1)}{S_{par}(n, p)} \qquad (4.3)$$

Where:
S_{seq}: speed (running time) of the sequential version.
S_{par}: speed (running time) of the parallel version.
p: number of processors.
n: problem size which denotes the number of computations that the program has to perform. As the problem size increases, the speedup increases.

In fact, as the number of processors increases, the speedup increases. However, there is upper bound on speedup due to that there is no program can be completely parallel. Thus, there is a time allocated for the sequential fraction of the program. If f is the sequential fraction of the program, then its speedup is bounded by $\frac{1}{f}$ regardless of the number of processors. This is known as Amdahl's Law.

In this research, serial and parallel models of the k-means clustering algorithm are designed and implemented in order to calculate the speedup gains of parallel implementation which express the impact of parallelization.

2. Efficiency:

An alternative performance measure of a parallel application is the efficiency. It captures how a program's speedup is close to ideal. In other words, efficiency measures the effectiveness of processors utilization of the parallel program [15]. It can be defined as the ratio of actual speedup to the number of processors [14] and expressed as [15]:

$$Efficiency = \frac{Speedup(n, p)}{p} \qquad (4.4)$$

Where:
n: problem size.
p: number of processors.

In an ideal parallel program, efficiency is equal to one and speedup is equal to p. However, in practice, it cannot achieve the ideal behavior because while running a parallel program, the processors cannot spend 100% of their time on program's computations. Thus, a real parallel program has an efficiency value between zero and one and speedup less than p [16].

3. Scalability:

The scalability of a parallel program is a measure to describe how the program performance changes as the number of processors is increased. As mentioned earlier, a speedup saturation can be observed when the problem size n is fixed, and the number of processors p is increased. However, the attained speedup increases when the problem size n increases for a fixed number of processors p [14]. In this sense, a parallel program is scalable if its performance improve continues as both problem size n and number of processors p are increased. Scalability is given by the following formula [17]:

$$Scalability = \frac{n_{par}}{n_{5eq}} \qquad (4.5)$$

Where:
n_{5eq}: the largest problem size that the sequential program can be handled.
n_{par}: the largest problem size that the parallel program can be handled for a specific number of processors.

4.3 Experimental Results

Extensive experiments on both sequential and parallel models of k-means clustering algorithm are conducted. For a fair comparison between the two versions, the convergence criterion is met after a fixed number of iterations. We used a different number of points and a different number of cores to illustrate the impact of both on running time. Generally, the dataset consists of more than 100,000 points with 2 dimensions. Clustering benchmark dataset called Birch is used [18]. The additional parameters are illustrated in table 4.1.

Table 4.1 Parameters of Experiments

Parameter	Value
No. of Points	40,000 / 80,000 / 120,000 / 160,000
No. of Clusters	20
No. of Iterations	500
No. of Cores / Threads	2 / 4 / 8

1. Accuracy of Clustering and SSE:

For all experiments that aim to evaluate the performance of parallel k-means, the accuracy of clustering and SSE values have been measured. As shown in table 4.2, all experiments result in an accuracy of 100%, implying the efficiency of k-means clustering algorithm despite its simplicity. In order to measure the homogeneity of the clustering results, the program is run multiple times with multiple different initial centroids and choose the results that give the smallest squared error as well as give the smallest running time as illustrated in the next section. The detailed results of accuracy and SSE are recorded in table 4.2. The large numbers of SSE do not mean large error values, but the reason is that there is a large number of points and each attribute of a point consists of six digits.

Table 4.2 Accuracy of Clustering and SSE

Points	Measure	Sequential	Parallel		
			P=2	P=4	P=8
40,000	Accuracy	100%	100%	100%	100%
	SSE	14382.9	25322.4	71432.6	12756.8
80,000	Accuracy	100%	100%	100%	100%
	SSE	50697.8	45876.6	31665.2	53777.5
120,000	Accuracy	100%	100%	100%	100%
	SSE	22473.2	29548.3	78532.9	43672.1
160,000	Accuracy	100%	100%	100%	100%
	SSE	55821.7	81325.4	28775.1	92341.5

2. Measuring Running Time:

The running time of a sequential program is the time that the program takes from start to end its execution on a computer. The running time of a parallel program is the time that starts with the beginning of parallel computation and ends when the last processor finishes execution [16]. When a parallel program runs multiple times, it hardly yields the same running time, even with identical inputs. One of the potential reasons is that the background processes and other user programs running on the same computer stealing some CPU time away from the parallel program, causing the running time of the parallel program to increase [15], [17]. To address this issue, the parallel program should be run multiple times (e.g., 10 times) and measure the running time for each run, then the smallest running time value would be the best estimate of the true running time of the parallel program [17].

The first set of experiments are conducted to illustrate the impact of both the problem size and the number of cores on running time. The program is run multiple times with a various number of points and cores. The detailed results are recorded in table 4.3 based on the smallest running time. The results are also represented graphically in figure 4.2. As seen in the graph, as the number of points increases, the running time also increases for all cores. Further, the sequential *k*-means has a higher increasing rate in terms of running time as compared to the parallel *k*-means version for all cores. Also, the increasing number of cores results in decreasing the running time as shown in the graph.

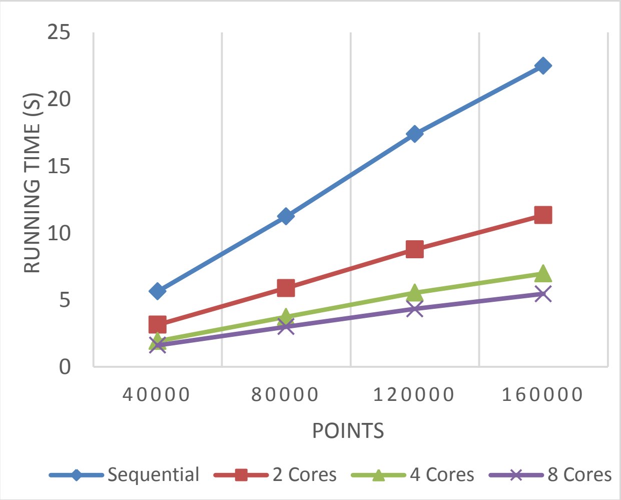

Figure 4.2 The Running Time Against the Number of Cores

3. Speedup Measurement:

In terms of speedup, multiple experiments are conducted to measure the gained speedup of parallelizing the *k*-means algorithm. The speedup is calculated according to formula 4.3 and based on the running time results for a different number of points and a different number of cores. The detailed results are recorded in table 4.3. Furthermore, the speedup results are represented in figure 4.3. As seen in the graph, the increasing number of cores results in increasing the speedup for all number of points. However, the rate of increase begins to slow as the number of points and cores increases. These results are expected since the scheduling of a large number of threads becomes more complicated. Therefore, the imposed overhead is higher.

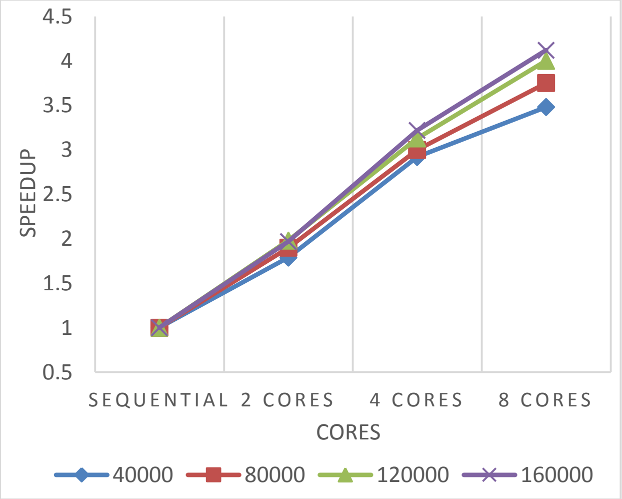

Figure 4.3 The Speedup Against the Number of Cores

4. Efficiency Measurement:

The third set of experiments are conducted with the aim of evaluating the efficiency of using the hardware resources. It is computed according to formula 4.4. The efficiency results shown in table 4.3 are also represented graphically in figure 4.4 for a different number of points and cores. It is noted that as the number of cores increases, the efficiency decreases. For 2 and 4 cores/threads the efficiency is close to 1 because the experiments are conducted on a quad-core, while the efficiency decreases with 8 threads (4 cores with hyperthreading) and may reach to zero with increasing number of cores. These results are expected because the hyperthreading is very expensive in terms of threads scheduling despite its advantages. Therefore, this overhead yield less efficiency.

Although the efficiency of the sequential version is equal to 1, it does not mean that there is a full using of hardware. However, the number of cores and the speedup of the sequential version are both equal to 1.

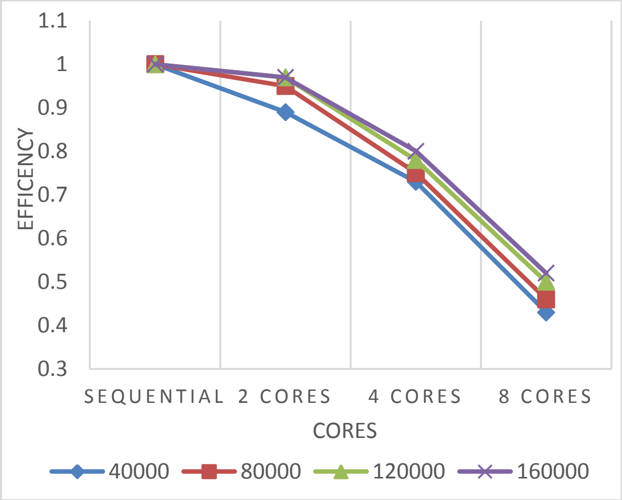

Figure 4.4 The Efficiency Against the Number of Cores

Table 4.3 Running Time, Speedup and Efficiency

Points	Measure	Sequential	Parallel		
			P=2	P=4	P=8
40,000	Time(s)	5.64	3.15	1.93	1.62
	Speedup	1	1.79	2.92	3.48
	Efficiency	1	0.89	0.73	0.43
80,000	Time(s)	11.24	5.89	3.74	3.00
	Speedup	1	1.90	3.00	3.75
	Efficiency	1	0.95	0.75	0.46
120,000	Time(s)	17.39	8.78	5.54	4.34
	Speedup	1	1.98	3.13	4.00
	Efficiency	1	0.97	0.78	0.50
160,000	Time(s)	22.51	11.34	6.98	5.46
	Speedup	1	1.98	3.22	4.12
	Efficiency	1	0.97	0.80	0.52

5. Scalability Measurement:

The scalability is measured according to formula 4.5 by dividing the largest number of points that each core/thread can handle by the largest number of points that the sequential program can handle. The results are recorded in table 4.4. Furthermore, Figure 4.5 shows the relationship between the maximum number of points for each core against the running time graphically. According to the results, the scalability is increased as the number of cores increased also.

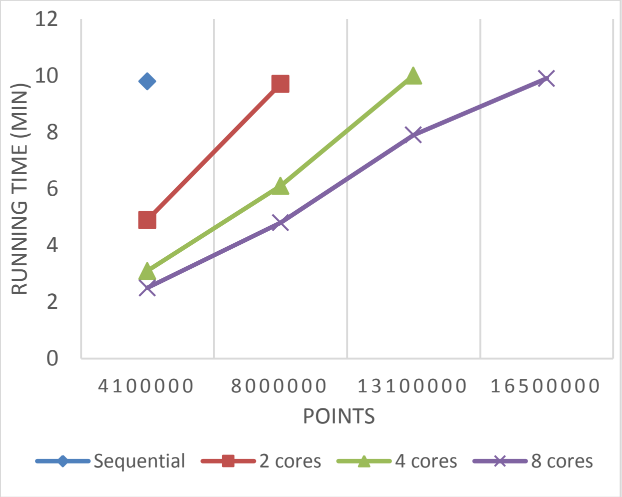

Figure 4.5 The Scalability Against the Running Time

Table 4.4 Scalability of Parallel k-means

Version	Cores	Size up	Scalability
Sequential	1	4100000	-
Parallel	2	8000000	1.9
Parallel	4	13100000	3.2
Parallel	8	16500000	4.0

5 ANALYTICAL RESULTS

For any parallel program, there is a fraction that should be implemented sequentially. In other words, it cannot parallelize the whole program. This is known as Amdahl's Law. The sequential fraction f of the program may consist of:

- Initialization statements.
- Thread creation and synchronization.
- Input and Output.
- Memory deallocation.

Regardless of the number of processors that are available in the parallel architecture, the sequential fraction f uses only one processor. Therefore, the speedup under Amdahl's Law is given by the following formula [15], [17]:

$$Speedup(n,p) \leq \frac{1}{f + \frac{(1-f)}{p}} \qquad (5.1)$$

Where:
$Speedup(n,p)$: the analytical speedup.
f: the sequential fraction.
$1-f$: the parallel fraction.
p: the number of processors.

To get the value of the sequential fraction f, the following formula is used:

$$f = \frac{p * T(n,p) - T(n,1)}{p * T(n,1) - T(n,1)} \qquad (5.2)$$

This formula calculates the sequential fraction f from running time measurements $T(n,p)$ and $T(n,1)$. Thus, Amdahl's Law can calculate the upper bound of the parallel program speedup. The speedup results are recorded in table 5.1. Relatively, the actual and the analytical speedup are almost the same.

Table 5.1 Actual and Analytical Speedup

Points	Measure	Parallel		
		P=2	P=4	P=8
40,000	**Actual Speedup**	1.79	2.92	3.48
	Analytical Speedup	1.79	2.92	3.48
80,000	**Actual Speedup**	1.90	3.00	3.75
	Analytical Speedup	1.91	3.01	3.75
120,000	**Actual Speedup**	1.98	3.13	4.00
	Analytical Speedup	1.98	3.14	4.01
160,000	**Actual Speedup**	1.98	3.22	4.12
	Analytical Speedup	1.98	3.23	4.12

6 CONCLUSION AND FUTURE WORK

This paper targets to accelerate the computation process of k-means clustering algorithm since the time complexity is increased with increasing the problem size. A parallel processing is used as a mechanism to achieve a high performance in terms of running time reduction. Parallel k-means is developed on SMP using java programming language under Linux operating system. We utilize 4 cores with hyperthreading (8 threads) managed by PJ2 library. Accuracy and SSE values are measured to evaluate the quality of clustering results. All results prove the efficiency of k-means with 100% clustering accuracy. In addition, speedup, efficiency and scalability are the metrics used to measure the performance of the parallel program. The results show that the gained speedup and the scalability increase with increasing the problem size and the number of cores, where the maximum speedup achieved is 4.12 and the maximum scalability reached is 4.0. On the other hand, the efficiency decreases with increasing the number of cores. The analytical results prove that the actual speedup is almost the same as the analytical speedup.

As a future work, we will develop another parallel version of k-means to run on cluster parallel programming model. This leads to a valuable comparison between the two models; SMP and clusters.

REFERENCES

[1] E. Karoussi, “Data Mining K-Clustering Problem,” 2012.

[2] J. A. Hartigan and M. A. Wong, “A K-Means Clustering Algorithm,” *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[3] T. Rauber and G. Rünger, “Parallel Computer Architecture,” in *Parallel Programming for Multicore and Cluster Systems*, 2nd ed., Springer, 2013, pp. 9–104.

[4] T. Kucukyilmaz, “Parallel K-Means Algorithm for Shared Memory Multiprocessors,” *Journal of Computer and Communications*, vol. 2, pp. 15–23, 2014.

[5] S. Kantabutra and A. L. Couch, “Parallel K-means Clustering Algorithm on NOWs,” *NECTEC Technical journal*, vol. 1, no. 6, pp. 243–248, 2000.

[6] V. Ramesh, K. Ramar, and S. Babu, “Parallel K-Means Algorithm on Agricultural Databases,” *International Journal of Computer Sciences*, vol. 10, no. 1, pp. 710–713, 2013.

[7] R. Farivar, D. Rebolledo, and E. Chan, “A parallel implementation of k-means clustering on GPUs,” *International Conference on Parallel and Distributed Processing Techniques and Applications*, vol. 13, no. 2, pp. 212–312, 2008.

[8] J. Wu and B. Hong, “An efficient k-means algorithm on CUDA,” in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1740–1749.

[9] A. K. Jain, “Data clustering: 50 years beyond K-means,” *Pattern Recognition Letters*, vol. 31, pp. 651–666, 2010.

[10] P. MacKey and R. R. Lewis, “Parallel k-Means++ for Multiple Shared-Memory Architectures,” *Proceedings of the International Conference on Parallel Processing*, vol. 2016–Septe, pp. 93–102, 2016.

[11] A. Apon, F. Robinson, D. Brewer, L. Dowdy, and D. Hoffman, *Initial starting point analysis for K-means clustering: a case study*. 2006.

[12] D. J. Bora and A. K. Gupta, “Effect of Different Distance Measures on the Performance of K-Means Algorithm : An Experimental Study in Matlab,” *International Journal of Computer Science and Information Technologies*, vol. 5, no. 2, pp. 2501–2506, 2014.

[13] A. Kaminsky, “The Parallel Java 2 Library,” in *The International Conference for high performance computing, networking, storage and analysis.*, 2014.

[14] S. Sahni and V. Thanvantri, “Parallel computing: Performance metrics and models,” *IEEE Parallel and Distributed Technology*, vol. 4, no. 1, pp. 43–56, 1996.

[15] A. Kaminsky, “Measuring Speedup,” in *Building Parallel Programs: SMPs, Clusters & Java*, 2009, pp. 99–110.

[16] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Second Edi. Addison Wesley, 2003.

[17] A. Kaminsky, *BIG CPU , BIG DATA: Solving the World’s Toughest Computational Problems with Parallel Computing*. 2015.

[18] T. Zhang, R. Ramakrishnan, and M. Livny, “BIRCH: A New Data Clustering Algorithm and Its Applications,” *Data Mining and Knowledge Discovery*, vol. 1, no. 2, pp. 141–182, 1997.