Table of Contents

1 Introduction

Clustering is a widely-used unsupervised learning technique with the goal of partitioning a dataset into distinct, non-overlapping sets. Each set, called a cluster, contains observations that are more similar to each other than to those in different clusters. Clustering is particularly useful when the aim is to discover structure on the basis of a particular dataset [1]. This technique has broad applications such as image compression, market segmentation and pattern recognition. There are many different clustering techniques, but *K*-means clustering is arguably the most popular among them.

The *K*-means clustering algorithm, otherwise known as Lloyd's algorithm, is known primarily for its simplicity. Starting with random centroids, the algorithm works to assign data points to the nearest cluster by calculating squared Euclidean distance. The centroids are then updated and the process repeats iteratively until convergence. The *K*-means clustering algorithm has a linear computational complexity of O(*nKt*), where *n* is the number of data points, *K* refers to the desired number of clusters, and *t* represents the number of iterations it takes for the algorithm to converge [2]. However, this algorithm suffers from a lack of scalability because a substantial amount of time is spent assigning points to the nearest cluster.

Fortunately, parallelization is an effective way to overcome this problem. Parallel computing describes a type of computation in which multiple CPUs work in tandem to complete smaller pieces of a larger task. The obvious benefit of parallel computing, as compared to serial computing, is that significantly less time is required to achieve the same end result. The basic *K*-means clustering algorithm is described as "embarrassingly parallel" [3]. That is, the task of assigning points to the nearest cluster is one that can be easily split and run simultaneously. This is due to the fact that finding the minimum squared Euclidean distance for each data point is a largely independent task.

In this report, we propose a parallel version of the *K*-means clustering algorithm and implement it using Python's *multiprocessing* library. We then run a series of simulations to compare the runtime and speedup between the basic *K*-means clustering algorithm and our parallel version. We also explore the impact of incorporating more CPUs into our parallel algorithm. Our findings confirm what we intuitively thought would happen based on our

understanding of parallel computing and the “embarrassingly parallel” nature of the *K*-means clustering algorithm. For datasets with fewer than 5,000 observations, the basic *K*-means clustering algorithm outperforms the parallel version. However, as the number of data points increases, our parallel *K*-means clustering algorithm significantly reduces overall time complexity. Added measures for speedup, scaleup and sizeup provide further evidence that our parallel *K*-means clustering algorithm is both robust and scalable.

2 Related Work

The *K*-means clustering algorithm was first formally proposed by James MacQueen in 1967 [4]. The same algorithm, however, was originally presented by Stuart Lloyd two years earlier for application in the field of pulse-code modulation, although the idea wasn’t formally published until 1982 [5]. It is for this reason that the *K*-means clustering algorithm and Lloyd’s algorithm are used almost interchangeably. Today, the algorithm is still used in its original form for a vast array of applications. Despite its ongoing popularity, this simple algorithm has severe limitations when handling large, complicated datasets.

There has been a significant amount of research over the years proposing new ways to improve the *K*-means clustering algorithm. One obvious area for improvement relates to the random initialization of centroids. Researchers have devised new initialization methods that aim to improve the overall effectiveness and efficiency of the *K*-means clustering algorithm. Arguably the best among these new methods is *k-means*++, an algorithm proposed in 2006 which considers spreading the initial centroids evenly. The use of *k-means*++ in generating initial centroids has been shown to improve both the speed and accuracy of the basic *K*-means clustering algorithm [6]. However, it still presents similar limitations to the original *K*-means clustering algorithm because of its sequential nature.

Consequently, many researchers have turned to parallel computing to efficiently perform *K*-means clustering on large datasets. Li and Fang (1989) were among the first to explore the application of parallel computing to *K*-means clustering. The authors proposed a parallel *K*-means clustering algorithm using a SIMD computer with multiple processors. They found a significant reduction in the overall time complexity compared with that achieved by a single processor computer [7]. Dhillon and Modha (2000) proposed a parallel *K*-means clustering

algorithm using distributed memory multiprocessors [8]. Rodrigues *et al.* (2012) proposed a hybrid parallelization of the *K*-means clustering algorithm using MPI and OpenMP [9]. The authors employ a reduction algorithm with a linear computational complexity that is dependent on the number of processors used.

The parallel *K*-means clustering algorithm in our paper closely follows the version proposed in 2015 by Kerdprasop and Kerdprasop. The authors implement a "message passing" approach which allows master-to-slave communication and any created processes to be run at different times. In this way, the algorithm becomes "highly parallel and fault tolerant." The authors additionally propose an approximate parallel *K*-means clustering algorithm for especially large datasets [2]. Both the parallel and approximate parallel *K*-means clustering algorithms demonstrated significantly reduced runtimes compared to traditional *K*-means clustering. Due to limited computational resources, our report focuses on a much smaller-scale implementation of this parallel *K*-means clustering algorithm using Python's *multiprocessing* library.

3 Intuition Behind *K*-Means Clustering

The first step in performing *K*-means clustering is to specify the desired number of clusters *K*. The *K*-means clustering algorithm then assigns each data point to one of the *K* clusters. The following figure demonstrates *K*-means clustering results for different values of *K*.

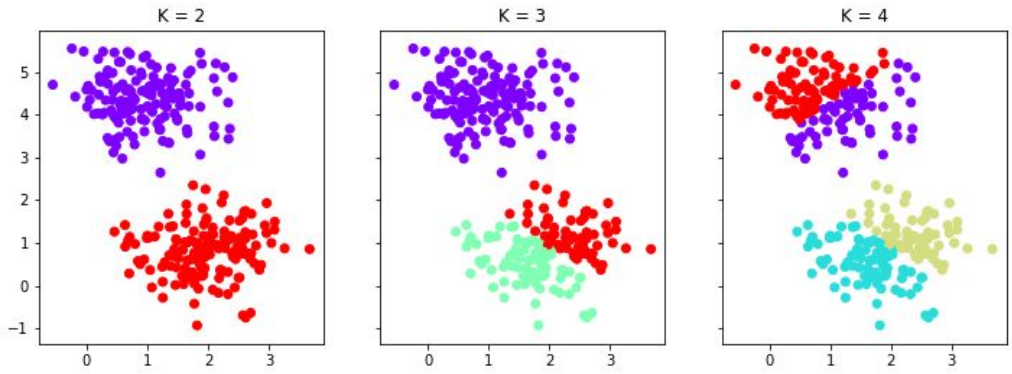

Figure 1. The *K*-Means clustering algorithm applied to a simulated dataset with 300 observations in a two-dimensional space for *K* = 2, 3, and 4. The different colors represent separate clusters.

The *K*-means clustering algorithm seeks to solve the problem of minimizing within-cluster variation, as measured by squared Euclidean distance. In this problem we have a set of points $P \subset R^d$ in a Euclidean space. The goal is to find a set $C \subset R^d$ of *K* points such that the sum of the squared Euclidean distances of the points in *P* to their nearest centroid in *C* is minimized. Thus, the objective function is:

$$\varphi(C) = \sum_{p \in P} min_{c \in C} \|p - c\|^2 ,$$

where $\|\cdot\|^2$ represents squared Euclidean distance. Note that the data points are partitioned into *K* clusters by assigning each point to its nearest centroid. Additionally, the *K*-means clustering algorithm requires that the number of clusters *K* be specified in advance [10].

4 Proposed Algorithms

In this analysis, we are interested in examining the overall performance differences between the basic *K*-means clustering algorithm and our parallel alternative. We implemented both the basic and parallel *K*-means clustering algorithms from scratch. In this way, we are able to accurately capture the performance differences between the two. The algorithms are implemented in Python 3.7.3 using the *Numpy, Pandas,* and *Multiprocessing* libraries.

4.1 Basic *K*-Means Clustering Algorithm

The basic *K*-means clustering technique is very simple. We start by randomly identifying *K* initial centroids, where *K* denotes the number of desired clusters and is typically specified by the user as an input. Next, we assign each data point to the nearest centroid (“assignment stage”). Each resulting collection of points assigned to a centroid is referred to as a cluster. We then update the centroids based on the new information from each cluster (“update stage”). We repeat the assignment and update stages iteratively until either our algorithm converges or the maximum number of iterations is reached. The steps involved in performing basic *K*-means clustering are more formally described in Algorithm 1 below.

Algorithm 1. *Basic K-Means Clustering*

(1) Randomly generate initial centroids
(2) Form *K* clusters by assigning each data point to its nearest centroid
(3) Update the centroids by calculating the cluster means
(4) Repeat steps 2 and 3 iteratively until convergence or maximum iterations is reached

Our basic *K*-means clustering algorithm is encompassed within a class called *K_Means. K_Means* contains a bundle of functions; however, most of them are referenced in the primary function *fit(),* which executes the basic *K*-means clustering algorithm. *K_Means* takes inputs *n_clusters* and *max_iter* which specify the desired number of clusters, *K,* and the maximum number of iterations the algorithm will perform in the event that it does not converge quickly. The function *fit()* takes an input *X,* which represents the complete set of data points in a sample.

The first line of *fit()* calls to a function *initial_centroid(),* which randomly generates the initial centroids. The disadvantage of this technique is that a poor initial guess may result in the algorithm converging to a local minimum, rather than a global one. It should be noted that there are several other ways of performing this step allowing for the selection of better initial centroids. The *k-means*++ algorithm, which was introduced in Section 2 above, is one such example. For the purposes of this analysis, however, we choose to use the original technique of generating initial centroids at random because we are examining the performance of serial vs. parallel algorithms. As a result, we can examine if we have an improvement even for bad initial choices of centroids.

Once we have our initial centroids, we begin our iterative steps. First, we form clusters by calling the function *assign_points_to_cluster().* In the first line of *assign_points_to_cluster()*, we call a function *_nearest()* which does the following: (1) calls a function *_distance()* to calculate the squared Euclidean distance between each data point and the initial centroids; and (2) assigns each data point to the cluster for which this distance is minimized. The output of *_nearest()* is a vector of labels corresponding to each of the *K* clusters. The second part of *assign_points_to_cluster()* uses nested for-loops in order to map these cluster labels to the corresponding data points. The function then returns *X_by_cluster,* a list of *K* arrays corresponding to the data points within each cluster.

Next, we update our centroids by using the information in *X_by_cluster* to calculate the cluster means. We then check to see if our algorithm has converged. That is, we are done if there is no difference between the new centroids and the old centroids. Otherwise, if the new centroids and the old centroids are not equal, then we repeat the assignment and update stages and check for convergence again. This continues until either the algorithm has converged or until the maximum number of iterations is reached.

4.2 Parallel *K*-Means Clustering Algorithm

While our basic *K*-means clustering algorithm is quite simple, it is also very computationally expensive for large datasets. The algorithm requires the calculation of squared Euclidean distances between each data point and the *K* centroids at each iteration until convergence. In an effort to increase the computational speed of this part of the algorithm, we parallelize the process of assigning points to clusters by dividing the task across *p* processors. As a result, each processor only has to complete this task for *n/p* data points where n is the number of observations in the dataset. Next, we aggregate the results from all *p* processors and use the information to update our centroids. The rest of the algorithm proceeds similarly to the basic *K*-means clustering algorithm. The details of our parallel *K*-means clustering algorithm are more formally defined in Algorithm 2 below.

Algorithm 2. *Parallel K-Means Clustering*

---

(1) Randomly generate initial centroids
(2) Randomly shuffle the data points and partition them into *p* subgroups
(3) For each subgroup *p,* assign each data point to its nearest centroid
(4) Aggregate the results from each subgroup *p* to form *K* clusters
(5) Update the centroids by calculating the cluster means
(6) Repeat steps 2-5 iteratively until convergence or maximum iterations is reached

---

Like our basic *K*-means clustering algorithm, the parallel version is encompassed in a class called *K_Means_parallel* and the bulk of the algorithm is run through a primary function *fit(). K_Means_parallel* takes inputs *n_clusters, max_iter,* and *num_cores,* the latter of which

specifies the number of processors to be used for parallelizing the *K*-means clustering algorithm. The function *fit()* takes an input *X* representing the complete set of data points.

The first line of *fit(),* like in the basic *K*-means clustering algorithm, serves to randomly generate the initial centroids. The next part, however, is where our parallel algorithm diverges. We begin our iterative steps by calling a function *_partition()* that randomly shuffles the data points in *X* and divides them into *p* subgroups according to the user specification for the parameter *num_cores*. Next, we make use of Python's *multiprocessing* library to create a pool [11]. We use this pool to map a task (in this case the *assign_points_to_cluster()* function) to each of the *p* available processors. By doing this, we are able to perform the assignment stage for each subgroup of data points simultaneously. This results in *p* versions of *X_by_cluster,* a list of *K* arrays corresponding to the data points within each cluster. We use nested for loops to aggregate each of the *p* lists to get a complete account of all the data points assigned to each of the *K* clusters. We proceed from here in a similar manner to the basic *K*-means clustering algorithm by updating the centroids and checking for convergence.

5 Experimental Procedure

We evaluate the accuracy and performance of our *K*-means clustering algorithms by generating artificial datasets using the *make_blobs()* function from Python's *sklearn.datasets* module [12]. This function generates isotropic Gaussian blobs that are ideal for clustering analyses because we have complete control over the centroids and standard deviations of the data clusters. We specify the following parameters: (1) *n_samples,* the number of observations to be divided equally among the clusters; (2) *centers,* the desired number of clusters; (3) *cluster_std,* the standard deviation of the clusters; and (4) *random_state,* which enables your output to be reproducible when an integer value is passed. Our code is executed on a machine with the following characteristics detailed in Table 1 below:

Table 1. Detailed description of the computer used to perform experiments

Feature	**Description**
CPU Model Name	Intel(R) Xeon(R) CPU E5-2530 v4 (2.20 GHz)
CPU Architecture	x86_64
CPU(s)	4
RAM	8GB

Due to both limited time and resources, we are unable to evaluate our algorithms using datasets with more than 500,000 observations.

5.1 Evaluating Algorithm Accuracy

To assess the accuracy of our basic and parallel *K*-means clustering algorithms, we cross-reference our results with the results from the *KMeans()* function housed within Python's *scikit-learn* library [13]. We start by generating artificial datasets using Python's *make_blobs()* function. We then pass the datasets through the *fit()* functions in our *K_Means* and *K_Means_parallel* classes. Additionally, we pass the same datasets through Python's *KMeans()* function. To run *KMeans()* in parallel, we specify a parameter *n_jobs,* representing the number of jobs to be used in the computation. We then examine our results to see if all functions converge to the same final centroids, as well as to check if the final cluster labels match. We repeat this process for several different artificial datasets to spot check the accuracy of our algorithms, and find that our results match those from Python's *KMeans()* function. As a result, we are confident that our algorithms are implemented correctly.

5.2 Evaluating Algorithm Performance

We evaluate the performance of our proposed algorithms by running simulations to explore the difference in runtime as the size of our dataset increases. We implement our simulations by creating artificial datasets using *make_blobs()* with the parameter *n_samples* ranging from 500 all the way to 500,000. For each value of *n_samples,* we conduct four simulations:

(1) Basic *K*-means clustering algorithm;

(2) Parallel *K*-means clustering algorithm (2 cores);
(3) Parallel *K*-means clustering algorithm (3 cores); and
(4) Parallel *K*-means clustering algorithm (4 cores).

For each simulation, the dataset is held constant at 2 dimensions and the number of clusters *K* is held constant at $K = 3$.

To measure the runtime of each experiment, we utilize the *timeit()* function within Python's *timeit* library [14]. The *timeit()* function allows us to record more accurate runtimes than do other similar functions, such as *time(),* for several reasons. First, *timeit()* by default turns off garbage collection during the timing. This eliminates the possibility that results will be skewed by an untimely collection run. Second, *timeit()* automatically picks the most accurate timer for our system. Last, *timeit()* allows us to repeat tests multiple times to eliminate the influence of other tasks that may be occurring on our machine. This is also especially useful in the context of our algorithms. As discussed earlier, we choose to generate the initial centroids randomly. The closer the initial centroids are to the final centroids, the faster the algorithm converges. Therefore, the runtime of a single experiment may be unduly influenced by the accuracy of the randomly-generated initial centroids. By using *timeit()* to repeat the experiment multiple times, we are able to stabilize any subsequent runtime fluctuations.

Next, we determine the optimal number of iterations for each experiment. We begin by trying 10 iterations and find that the results do not make much intuitive sense. This is primarily due to the fact that 10 is not a large enough number and as a result some randomness is affecting the results. Next, we try 30 iterations and find that the results are much more stable. In order to be conservative, however, we increase the number of iterations to 50 to ensure that our results are accurate. We decided that the marginal benefit of exceeding 50 iterations is insignificant, especially when compared to the added computational expense of doing so. For each experiment, then, we repeat our test 50 times and divide the final computational time by 50 to record an average runtime.

6 Experimental Results

As discussed in the previous section, we evaluate the performance of our basic and parallel *K*-means clustering algorithms by running various simulations on artificially-generated datasets.

In these simulations, we hold the dataset constant at 2 dimensions and the desired number of clusters constant at $K = 3$ but vary the number of data points. For our parallel *K*-means clustering algorithm, specifically, we also vary the number of processors used. The runtimes for each individual simulation are calculated by repeating our experiment 50 times and then dividing by 50 to arrive at an average runtime. The computational speeds for each of our simulations are presented in detail in Table 2 below:

Table 2. Runtime (seconds) of basic and parallel *K*-means clustering algorithms

Data Points	**Basic *K*-Means**	**Parallel *K*-Means**		
		2 CPUs	**3 CPUs**	**4 CPUs**
500	0.14	0.62	0.62	0.63
1,000	0.49	0.88	0.87	0.87
5,000	2.09	1.58	1.17	0.98
10,000	4.62	2.97	2.24	2.04
50,000	27.64	13.19	9.94	8.16
100,000	46.38	27.82	20.45	17.15
200,000	107.63	63.96	46.03	37.59
300,000	184.66	79.78	58.46	47.72
400,000	205.77	106.77	81.07	66.05
500,000	285.40	122.99	88.99	73.75

To make the trends observable in Table 2 more obvious, we display the runtimes from our simulations on a regular and log scale:

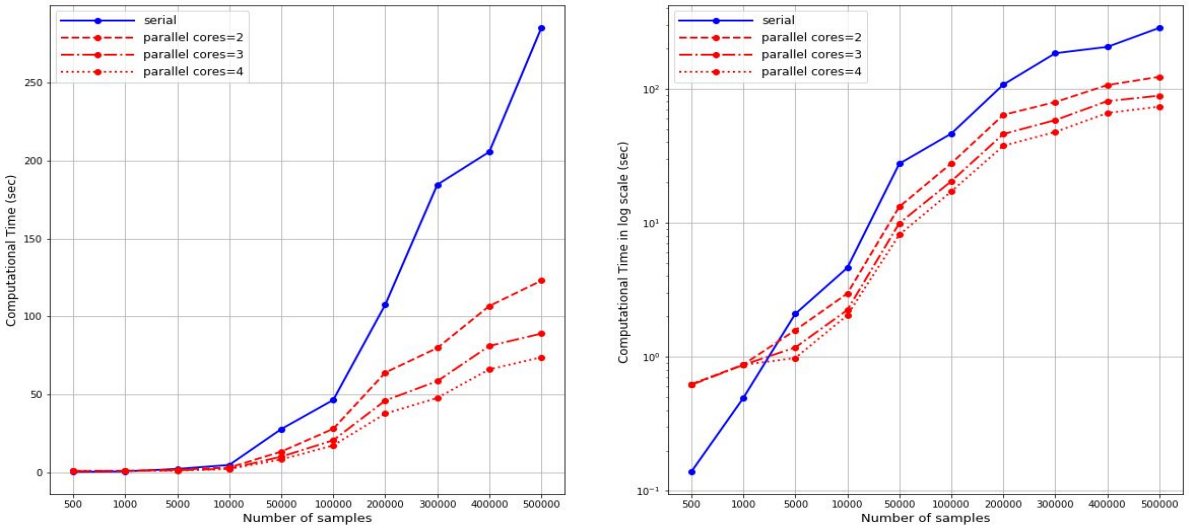

Figure 2. Simulation runtimes (seconds) of basic and parallel K-means clustering algorithms on both a regular scale (left) and log scale (right).

The results in both Table 2 and Figure 2 suggest that for datasets with fewer than 5,000 observations, the basic *K*-means clustering algorithm outperforms the parallel *K*-means clustering algorithm. This is likely due to the communication overhead between processors. However, when the number of observations reaches 5,000, the parallel *K*-means clustering algorithm performs better. For each subsequent increase in dataset size, the difference in runtimes between the basic and parallel *K*-means clustering algorithms becomes even more significant. Looking at the parallel *K*-means clustering algorithm specifically, our results suggest that for datasets with fewer than 50,000 observations, the runtimes are very similar for different numbers of processors. However, as the dataset grows even larger, the differences in runtimes become much more pronounced.

While the results in Table 2 and Figure 2 provide some information about the differences between our basic and parallel *K*-Means clustering algorithms, runtime alone is not a sufficient measure of scalability. We therefore incorporate measures of speedup, scaleup, and sizeup to gain more insight into the advantage offered by our parallel *K*-Means clustering algorithm.

Speedup is measured by keeping the number of observations in a dataset constant but varying the number of processors being used. To calculate speedup you must capture the time it takes for the algorithm to run using 1 processor (T_1) and the time it takes for the algorithm to

run using *p* processors (T_p) [15]. An algorithm is described as perfectly parallel if it displays a linear speedup [16]. We measure speedup using our simulation results for datasets with observations ranging from 100,000 to 500,000 and processors ranging from 1 to 4.

Scaleup can be defined as "the ability of a *p*-times larger system to perform a *p*-times larger job in the same execution time" [17]. It is measured by increasing both the number of observations in the dataset and the number of processors being used. To measure scaleup you must calculate the time it takes for the algorithm to run using 1 processor on a dataset of size *s* (T_{1s}) and the time it takes for the algorithm to run using *p* processors on a dataset of size *p x s* (T_{ps}). An algorithm is considered to have good scaleup if the ratio is close to 1 [15]. We measure scaleup using our simulation results where the number of observations in a dataset increases in proportion to the number of processors being used. Specifically, a dataset with 100,000 observations is run using 1 processor, a dataset with 200,000 observations is run using 2 processors, and so on until we reach 4 processors.

Sizeup is measured by keeping the number of processors used constant but increasing the number of observations in the dataset by a factor *p.* It represents the capability of an algorithm to handle a *p*-times larger dataset. Measuring sizeup requires calculating the time it takes for the algorithm to run on a dataset of size *s* (T_s) and the time it takes for the algorithm to run on a dataset of size *p x s* (T_{ps}). An algorithm has good sizeup performance if the ratio is close to *p* [15]. We measure sizeup by using our simulation results and fixing the number of processors used to 1, 2, 3 and 4, respectively. We begin by using a dataset with 100,000 observations and increase it by a factor *p* ranging from 1 to 5.

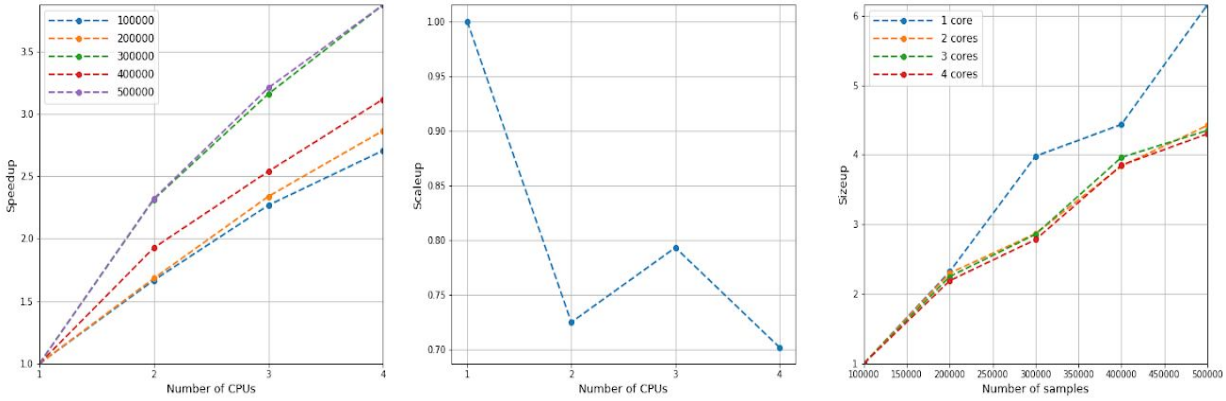

Figure 3. Speedup, Scaleup, and Sizeup results on different numbers of processors.

The plot on the left-hand side of Figure 3 displays speedup for different processors and dataset sizes. It is immediately clear that our parallel *K*-means clustering algorithm does not have linear speedup. This is not an issue, however, because speedup is typically sublinear in practice. For the most part, the speedup increases as the number of observations in the dataset increases. The only exception is that the dataset with 300,000 observations exhibits an unusually large speedup. This may be due to the fact that we were only able to repeat our experiments 50 times, so the results still contain some influence from the random generation of initial centroids. Alternatively, this could be due to the fact that our step size of 100,000 data points is relatively small. Overall, our parallel *K*-means clustering algorithm has good speedup performance.

The middle plot in Figure 3 displays the scaleup results for varying processors. The overall trend is decreasing. It is clear, however, that our scaleup value for each number of processors is still relatively close to 1. There is a small spike in the scaleup for 3 processors, which again is due to our simulation results for the dataset with 300,000 observations. Overall, our parallel *K*-means has good scalability.

The plot on the right-hand side of Figure 3 displays the sizeup results for varying dataset sizes and number of processors. Note that we flip the ratio in order to better visualize whether or not our parallel *K*-means clustering algorithm has good sizeup performance. The sizeup values do hover relatively close to the factor of increase p, which in this case ranges from 1 to 5. As a result, we are able to conclude that our parallel *K*-means clustering algorithm has the capacity to efficiently support large datasets.

7 Conclusion

K-means clustering is a simple yet powerful technique with broad applications spanning a vast array of industries. As such, it is essential that the *K*-means clustering algorithm can efficiently process datasets of varying size and complexity. The basic *K*-means clustering algorithm performs well with datasets with less than around 5,000 observations but becomes computationally burdensome for anything larger. In this paper, we propose a parallel *K*-means clustering algorithm that seeks to alleviate this problem. By exploiting the “embarrassingly parallel” nature of the *K*-means clustering algorithm, we show that the computational time

speeds up considerably for datasets with more than 5,000 observations. Additionally, our measures of speedup, scaleup, and sizeup provide further evidence for the scalability and robustness of our parallel *K*-means clustering algorithm.

Our future work will focus on evaluating algorithm performance by varying additional factors, such as number of clusters *K* and dataset dimensionality. We will also seek out more powerful resources, allowing us to run simulations for datasets with more than 500,000 observations. Without such computational limitations, we will also be able to further stabilize our results by repeating experiments more than 50 times.

References

[1] James, G., Witten, D., Hastie, T., & Tibshirani, R. *An Introduction to Statistical Learning: with Applications in R.* New York: Springer, 2013, pp. 385-386.

[2] Kerdprasop, K. & Kerdprasop, N. (2010) Parallelization of K-Means Clustering on Multi-Core Processors. *Selected Topics in Applied Computer Science,* pp. 472-477. ISSN: 1792-4863.

[3] 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

[4] MacQueen, J. (1967) Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* 1(14): 281-297. https://projecteuclid.org/euclid.bsmsp/1200512992

[5] Lloyd, S. Least squares quantization in PCM. (1982) *IEEE Transactions on Information Theory* 28(2): 129-137. https://doi.org/10.1109/TIT.1982.1056489

[6] Arthur, D. & Vassilvitskii, S. (2006) *k-means++: The Advantages of Careful Seeding.* Technical Report. Stanford. http://ilpubs.stanford.edu:8090/778/

[7] Li, X. & Fang, Z. (1989) Parallel clustering algorithms. *Parallel Computing* 11(3): 275-290. https://doi.org/10.1016/0167-8191(89)90036-7

[8] Dhillon, I. & Modha, D. (2000) A Data-Clustering Algorithm on Distributed Memory Multiprocessors. *Large-Scale Parallel Data Mining*. Lecture Notes in Computer Science, vol 1759. https://doi.org/10.1007/3-540-46502-2_13

[9] Rodrigues, L, Zárate, L., Nobre, C. & Freitas, H. (2012) Parallel and distributed kmeans to identify the translation initiation site of proteins. *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. https://doi.org/10.1109/ICSMC.2012.6377972

[10] Blömer, J., Lammersen, C., Schmidt, M., & Sohler, C. (2016) Theoretical Analysis of the *k*-Means Algorithm – A Survey. *Algorithm Engineering*. Lecture Notes in Computer Science, vol 9220. https://doi.org/10.1007/978-3-319-49487-6_3

[11] Documentation for Python *multiprocessing* Library. https://docs.python.org/3.7/library/multiprocessing.html

[12] Documentation for Python *sklearn.datasets.make_blobs* Function. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

[13] Documentation for Python *sklearn.cluster.KMeans* Function. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

[14] Documentation for Python *timeit* Library. https://docs.python.org/3.7/library/timeit.html

[15] Nasraoui, O. & Ben N'Cir, C. (editors). *Clustering Methods for Big Data Analytics: Techniques, Toolboxes and Applications.* Springer International Publishing, 2019, pp. 107-108.

[16] Zhao, W., Ma, H., He, Q. (2009) Parallel *K*-Means Clustering Based on MapReduce. *Cloud Computing.* CloudCom 2009. Lecture Notes in Computer Science, vol 5931. https://doi.org/10.1007/978-3-642-10665-1_71

[17] Wang, Y., Li, T. (editors). *Practical Applications of Intelligent Systems: Proceedings of the Sixth International Conference on Intelligent Systems and Knowledge Engineering, Shanghai, China, Dec 2011 (ISKE 2011).* Springer-Verlag Berlin Heidelberg, 2012, pp. 65-66.

Appendices

A.1 Python Code - Basic *K*-Means Clustering Algorithm

```
# Import Python modules
from __future__ import division
import numpy as np
import timeit

class K_Means(object):
    # Initialize input values n_clusters and max_iter
    def __init__(self, n_clusters, max_iter):
        self.n_clusters = n_clusters
        self.max_iter = max_iter

    # Function that assigns points to a cluster
    def assign_points_to_cluster(self, X):
        # Label points according to the minimum euclidean distance
        self.labels_ = [self._nearest(self.cluster_centers_, x) for x in X]
        # Map labels to data points
        indices=[]
        for j in range(self.n_clusters):
            cluster=[]
            for i, l in enumerate(self.labels_):
                if l==j: cluster.append(i)
            indices.append(cluster)
        X_by_cluster = [X[i] for i in indices]
        return X_by_cluster

    # Function that randomly selects initial centroids
    def initial_centroid(self, X):
        initial = np.random.permutation(X.shape[0])[:self.n_clusters]
        return X[initial]

    # Function that updates centroids and repeats
    # assign_points_to_cluster until convergence or max_iter is reached
    def fit(self, X):
        self.cluster_centers_ = self.initial_centroid(X)
        for i in range(self.max_iter):
            X_by_cluster = self.assign_points_to_cluster(X)
            # calculate the new centroids
```

```
            new_centers=[c.sum(axis=0)/len(c) for c in X_by_cluster]
            new_centers = [arr.tolist() for arr in new_centers]
            old_centers=self.cluster_centers_
            # compare the new centroid to the old one
            # if they are same our algorithms has converged
            # else continue
            if np.all(new_centers == old_centers):
                self.number_of_iter=i
                break;
            else:
                self.cluster_centers_ = new_centers
        self.number_of_iter=i
        return self

    # Function that calculates the minimum euclidean distance
    def _nearest(self, clusters, x):
        return np.argmin([self._distance(x, c) for c in clusters])

    # Function that calculates the euclidean distance between two points
    def _distance(self, a, b):
        return np.sqrt(((a - b)**2).sum())

    # Function that returns predicted clusters for each point
    def predict(self, X):
        return self.labels_

##########################################################

# SIMULATIONS
sim1 = []

# TEST_CODE is used to initialize Kmeans Class having 3 clusters
# and 500 as maximum number of iterations
TEST_CODE = """
kmeans = K_Means(n_clusters = 3, max_iter = 500)
kmeans.fit(X)
"""
## 1 ####################################################
# SETUP_CODE is used to generate the dataset
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=500, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
```

```
"""
# run simulation and append the average computational time to sim1 list
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 2 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=1000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 3 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=5000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 4 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=10000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 5 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=50000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 6 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=100000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)
```

```
## 7 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=200000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 8 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=300000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 9 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=400000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

## 10 ################################################
SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=500000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means
"""
sim1.append(timeit.timeit(stmt=TEST_CODE,setup=SETUP_CODE,number=50)/50)

import pandas as pd
# Creating a dataframe with the results
results = pd.DataFrame(sim1)
# save data to a csv file
results.to_csv('./sim_serial_k3.csv', sep='\t')
```

A.2 Python Code - Parallel *K*-Means Clustering Algorithm

```
# Import Python modules
from __future__ import division
import numpy as np
from multiprocessing import Pool
import timeit

class K_Means_parallel(object):
    # Initialize input values n_clusters and max_iter
    def __init__(self, n_clusters, max_iter, num_cores):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.num_cores = num_cores

    # Function that assigns points to a cluster
    def assign_points_to_cluster(self, X):
        # Label points according to the minimum euclidean distance
        self.labels_ = [self._nearest(self.cluster_centers_, x) for x in X]
        # Map labels to data points
        indices=[]
        for j in range(self.n_clusters):
            cluster=[]
            for i, l in enumerate(self.labels_):
                if l==j: cluster.append(i)
            indices.append(cluster)
        X_by_cluster = [X[i] for i in indices]
        return X_by_cluster

    # Function that randomly selects initial centroids
    def initial_centroid(self, X):
        initial = np.random.permutation(X.shape[0])[:self.n_clusters]
        return  X[initial]

    # Function that updates centroids and repeats
    # assign_points_to_cluster until convergence or max_iter is reached
    def fit(self, X):
        self.cluster_centers_ = self.initial_centroid(X)
        for i in range(self.max_iter):
            # split data to self.num_cores chunks
            splitted_X=self._partition(X,self.num_cores)
            # Parallel Process for assigning points to clusters
            p=Pool()
            result=p.map(self.assign_points_to_cluster, splitted_X )
            p.close()
```

```
            # Merge results
            p.join()
            X_by_cluster=[]
            for c in range(0,self.n_clusters):
                r=[]
                for p in range(0,self.num_cores):
                    tmp=result[p][c].tolist()
                    r=sum([r, tmp ], [])
                X_by_cluster.append(np.array(r))
            # calculate the new centers
            new_centers=[c.sum(axis=0)/len(c) for c in X_by_cluster]
            new_centers = [np.array(arr) for arr in new_centers]
            old_centers=self.cluster_centers_
            old_centers = [np.array(arr) for arr in old_centers]
            # if the new centroids are the same as the old centroids then the
            # algorithm has converged
            if all([np.allclose(x, y) for x, y in zip(old_centers, new_centers)]) :
                self.number_of_iter=i
                break;
            else :
                self.cluster_centers_ = new_centers
        self.number_of_iter=i
        return self

    # Function that randomly shuffles and partitions the dataset
    def _partition ( self,list_in, n):
        temp = np.random.permutation(list_in)
        result = [temp[i::n] for i in range(n)]
        return result

    # Function that calculates the minimum euclidean distance
    def _nearest(self, clusters, x):
        return np.argmin([self._distance(x, c) for c in clusters])

    # Function to calculate euclidean distance between two points
    def _distance(self, a, b):
        return np.sqrt(((a - b)**2).sum())

    # Function that returns predicted clusters for each point
    def predict(self, X):
        return self.labels_

sim2 = []
```

```
sim3 = []
sim4 = []

TEST_CODE2 = """
parakmeans = K_Means_parallel(n_clusters = 3, max_iter = 500, num_cores = 2)
parakmeans.fit(X)
"""

TEST_CODE3 = """
parakmeans = K_Means_parallel(n_clusters = 3, max_iter = 500, num_cores = 3)
parakmeans.fit(X)
"""

TEST_CODE4 = """
parakmeans = K_Means_parallel(n_clusters = 3, max_iter = 500, num_cores = 4)
parakmeans.fit(X)
"""

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=500, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""
sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("1")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=1000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("2")

SETUP_CODE = """
import sklearn.datasets as skl
```

```
X, y = skl.make_blobs(n_samples=5000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("3")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=10000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)
print ("4")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=50000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("5")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=100000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)
```

```
print ("6")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=200000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)
print ("7")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=300000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("8")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=400000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""

sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("9")

SETUP_CODE = """
import sklearn.datasets as skl
X, y = skl.make_blobs(n_samples=500000, centers=3, cluster_std=0.60, random_state=0)
from __main__ import K_Means_parallel
"""
```

```
sim2.append(timeit.timeit(stmt=TEST_CODE2,setup=SETUP_CODE,number=50)/50)
sim3.append(timeit.timeit(stmt=TEST_CODE3,setup=SETUP_CODE,number=50)/50)
sim4.append(timeit.timeit(stmt=TEST_CODE4,setup=SETUP_CODE,number=50)/50)

print ("10")
```

A.3 Python Code - Assessing Algorithm Accuracy

```
import sklearn.datasets as skl

# generate dataset
n= ‘number of samples’
k=3 # number of clusters
X, y = skl.make_blobs(n_samples=n, centers=k, cluster_std=0.30, random_state=0)

# parallel version of K-means
p= ‘number of cores used [2,4]’
kmeans_ = K_Means_parallel(n_clusters = k, max_iter = 500, num_cores = p)
kmeans_.fit(X) # fit the model
print (kmeans_.cluster_centers_) # print final centroids

# sklearn version of k-means
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=k, random_state=0).fit(X) # fit the model
print (kmeans.cluster_centers_)  # print final centroids

# Visualize clustered data
plt.scatter(X[:,0],X[:,1],c=kmeans.predict(X),cmap='rainbow')
Centroids = np.array(kmeans_.cluster_centers_)
plt.scatter(Centroids[:,0],Centroids[:,1],s=500,c='yellow')
```

A.4 Python Code - *K*-Means Clustering Algorithm Intuition

```
# Import Python modules
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
import sklearn.datasets.samples_generator as skl
%matplotlib inline

#########################################################################
# Part 1 - Original K-Means Clustering
```

##

```
class K_Means(object):
    # Initialize input values n_clusters and max_iter
    def __init__(self, n_clusters, max_iter):
        self.n_clusters = n_clusters
        self.max_iter = max_iter

    # Function that assigns points to a cluster
    def assign_points_to_cluster(self, X):
        # Label points according to the minimum euclidean distance
        self.labels_ = [self._nearest(self.cluster_centers_, x) for x in X]
        # Map labels to data points
        indices=[]
        for j in range(self.n_clusters):
            cluster=[]
            for i, l in enumerate(self.labels_):
                if l==j: cluster.append(i)
            indices.append(cluster)
        X_by_cluster = [X[i] for i in indices]
        return X_by_cluster

    # Function that randomly selects initial centroids
    def initial_centroid(self, X):
        # Set the seed for illustrative purposes ONLY
        np.random.seed(1)
        initial = np.random.permutation(X.shape[0])[:self.n_clusters]
        return X[initial]

    # Function that updates centroids and repeats
    # assign_points_to_cluster until convergence or max_iter is reached
    def fit(self, X):
        self.cluster_centers_ = self.initial_centroid(X)
        for i in range(self.max_iter):
            X_by_cluster = self.assign_points_to_cluster(X)
            new_centers=[c.sum(axis=0)/len(c) for c in X_by_cluster]
            new_centers = [arr.tolist() for arr in new_centers]
            old_centers=self.cluster_centers_
            if np.all(new_centers == old_centers):
                self.number_of_iter=i
                break;
            else:
                self.cluster_centers_ = new_centers
```

```
        self.number_of_iter=i
        return self

    # Function that calculates the minimum euclidean distance
    def _nearest(self, clusters, x):
        return np.argmin([self._distance(x, c) for c in clusters])

    # Function that calculates euclidean distance between two points
    def _distance(self, a, b):
        return np.sqrt(((a - b)**2).sum())

    # Function that returns predicted clusters for each point
    def predict(self, X):
        return self.labels_

################################################################################
# Test
################################################################################

X, y = skl.make_blobs(n_samples=300, centers=2, cluster_std=0.60, random_state=0)

fig, ax = plt.subplots(1, 3, sharex='col', sharey='row')

# Get current size
fig_size = plt.rcParams["figure.figsize"]
print("Current size:", fig_size)

# Set figure width to 12 and height to 9
fig_size[0] = 12
fig_size[1] = 4
plt.rcParams["figure.figsize"] = fig_size

# Clusters = 2
kmeans = K_Means(n_clusters=2, max_iter=500).fit(X)
ax[0].scatter(X[:,0],X[:,1],c=kmeans.predict(X),cmap='rainbow')
ax[0].set_title('K = 2')

# Clusters = 3
kmeans = K_Means(n_clusters=3, max_iter=500).fit(X)
ax[1].scatter(X[:,0],X[:,1],c=kmeans.predict(X),cmap='rainbow')
ax[1].set_title('K = 3')

# Clusters = 4
```

```
kmeans = K_Means(n_clusters=4, max_iter=500).fit(X)
ax[2].scatter(X[:,0],X[:,1],c=kmeans.predict(X),cmap='rainbow')
ax[2].set_title('K = 4')
```

A.5 Python Code - *Diagrams for Experimental Results*

```
df=pd.read_csv('./results_final.csv')
df=df.T.reset_index()
df = df.iloc[1:]
df.columns = ['simulations', 'serial', 'parallel cores=2', 'parallel cores=3', 'parallel cores=4']
df['sample_size']=["500","1000","5000","10000","50000","100000","200000","300000","400000","
500000"]

df['speedup cores=2']=df['serial']/df['parallel cores=2']
df['speedup cores=3']=df['serial']/df['parallel cores=3']
df['speedup cores=4']=df['serial']/df['parallel cores=4']

speed_up=df[['speedup cores=2','speedup cores=3','speedup cores=4']]
speed_up['serial']=[1,1,1,1,1,1,1,1,1,1]
speed_up=speed_up.T
speed_up["cores"]=["2","3","4","1"]
speed_up.columns =["500", "1000","5000","10000",
"50000","100000","200000","300000","400000","500000","cores"]
speed_up = speed_up.sort_values(by ='cores' )

scale_up=df[['sample_size',"parallel cores=2","parallel cores=3","parallel cores=4", "serial"]]
scale_up=pd.DataFrame([1.0 ,
                       scale_up.iloc[5,4]/scale_up.iloc[6,1] ,
                       scale_up.iloc[5,4]/scale_up.iloc[7,2], scale_up.iloc[5,4]/scale_up.iloc[8,3]])
scale_up['cores']=["1","2","3","4"]
scale_up.columns=['results','cores']

p_1_=[1.0,df.iloc[6,1]/df.iloc[5,1], df.iloc[7,1]/df.iloc[5,1], df2.iloc[8,1]/df2.iloc[5,1],
     df2.iloc[9,1]/df2.iloc[5,1] ]
p_2_=[1.0,df.iloc[6,2]/df.iloc[5,2], df.iloc[7,2]/df.iloc[5,2], df.iloc[8,2]/df.iloc[5,2],
       df.iloc[9,2] /df.iloc[5,2]]
p_3_=[1.0,df.iloc[6,3]/df.iloc[5,3], df.iloc[7,3]/df.iloc[5,3], df.iloc[8,3]/df.iloc[5,3],
       df.iloc[9,3] /df.iloc[5,3]]
p_4_=[1.0,df.iloc[6,4]/df.iloc[5,4], df.iloc[7,4]/df.iloc[5,4], df.iloc[8,4]/df.iloc[5,4],
     df.iloc[9,4]/df.iloc[5,4] ]

size_up=pd.DataFrame()
```

```
size_up["1"]=p_1_
size_up["2"]=p_2_
size_up["3"]=p_3_
size_up["4"]=p_4_
size_up['data_points']=[100000,200000,300000,400000,500000]

fig, ax =plt.subplots(1,3, figsize=(33,11))

# speed up
line, = ax[0].plot('cores',"100000", data=speed_up, lw=2, marker='o', linestyle='dashed')
line, = ax[0].plot('cores',"200000", data=speed_up, lw=2, marker='o', linestyle='dashed')
line, = ax[0].plot('cores',"300000", data=speed_up, lw=2, marker='o', linestyle='dashed')
line, = ax[0].plot('cores',"400000", data=speed_up, lw=2, marker='o', linestyle='dashed')
line, = ax[0].plot('cores',"500000", data=speed_up, lw=2, marker='o', linestyle='dashed')
plt.margins(0)
ax[0].margins(0)
ax[0].legend(loc="upper left")
ax[0].set_ylabel('Speedup',  fontsize=13)
ax[0].set_xlabel('Number of CPUs',  fontsize=13)
ax[0].grid()

# scale up
line, = ax[1].plot('cores',"results", data=scale_up, lw=2, marker='o', linestyle='dashed')
ax[1].set_ylabel('Scaleup',  fontsize=13)
ax[1].set_xlabel('Number of CPUs',  fontsize=13)
ax[1].grid()

# size up
line, = ax[2].plot('data_points',"1", data=size_up, lw=2, marker='o', linestyle='dashed', label='1
core')
line, = ax[2].plot('data_points',"2", data=size_up, lw=2, marker='o', linestyle='dashed', label='2
cores')
line, = ax[2].plot('data_points',"3", data=size_up, lw=2, marker='o', linestyle='dashed', label='3
cores')
line, = ax[2].plot('data_points',"4", data=size_up, lw=2, marker='o', linestyle='dashed', label='4
cores')

ax[2].legend(loc="upper left")
ax[2].set_ylabel('Sizeup',  fontsize=13)
ax[2].set_xlabel('Number of samples',  fontsize=13)
ax[2].grid()

plt.savefig('Metrics_28012020.png')
```