Analyzing Digital Evidence Using Parallel k-means with Triangle Inequality on Spark

Ambika Shrestha Chitrakar
Norwegian University of Science and Technology (NTNU)
Gjøvik, Norway
ambika.chitrakar2@ntnu.no

Slobodan Petrović
Norwegian University of Science and Technology (NTNU)
Gjøvik, Norway
slobodan.petrovic@ntnu.no

***Abstract*—Analyzing digital evidence has become a big data problem, which requires faster methods to handle them on a scalable framework. Standard k-means clustering algorithm is widely used in analyzing digital evidence. However, it is a hill-climbing method and it becomes slower with the increase of data, its dimension, and the number of cluster centers. This paper presents a framework to implement parallel k-means with triangle inequality (k-meansTI) algorithm on Spark, which is supposed to improve the speed of the standard k-means algorithm by skipping many point-center distance computations, giving the same clustering results. Our experimental results show that the parallel implementation of k-meansTI on Spark can be faster than the Spark ML k-means when a data set is large, does not contain many sparse data, and is high dimensional. These results are based on the experiments performed on six different data sets that have variations on the number of features and the number of data instances.**

***Index Terms*—Digital Evidence, Analysis, Clustering, Triangle Inequality, Spark**

I. Introduction

Any digital data or information (audio, video, picture, or text) that can be presented in the court of law is digital evidence and it is increasing exponentially because of the advancement in digital technologies such as mobiles, hard disks, solid state memories, tablets, cloud services, and network traffic. Analyzing such a big data requires faster methods and highly scalable frameworks. There are many promising methods for data analysis. However, they may not be suitable to handle big data on a scalable framework. There is a need of re-formulating promising algorithms to handle big data problems and improve their performance on the scalable big data frameworks such as Hadoop and Apache Spark.

Clustering algorithms have been widely used in analyzing digital evidence. Text clustering [1], document clustering [2], clustering digital forensic search [3], and log analysis [4] are few of the examples. Standard k-means algorithm, also known as Lloyd's algorithm [5], [6], is one of the simplest and popular partitional clustering algorithms used in data analysis. It helps to group similar instances in one cluster and separate the dissimilar instances in other clusters. The similarity of the instances is usually measured by Euclidean distance metric. The instances closer to each other are considered similar and the instances that are far away from each other are considered dissimilar.

Besides the simplicity of the k-means algorithm, it is a hill-climbing method. Its time complexity is $O(kne)$ [7] and is highly affected by the number of clusters k, the number of instances n, and the number of iterations e. k-means algorithm becomes slower with the increase of k, n, e. Elkan's k-means with triangle inequality (k-meansTI) is one of the improved versions of the standard k-means algorithm that improves its performance by skipping many point-center distance computations. This algorithm has possibility to reduce the time complexity from $O(kne)$ to $O(n)$, giving the same clustering results as standard k-means when the initial cluster centers for both algorithms are the same [7].

Implementing parallel k-meansTI on a big data framework to analyze digital evidence can speedup the process of analysis. Apache Spark is one of such open source big data platforms, which is in-memory persistent, fault tolerant, and provides methods to process data in parallel. Spark can keep the distributed data in memory instead of writing to disk. This helps in improving its performance. It has unique method for fault tolerance. Instead of logging updates or duplicating data across machines, it recalculates the partition based on the dependency information stored in each partition whenever a partition is lost [8]. Spark has Spark MLlib and Spark ML machine learning libraries, which contain parallel implementation of popular machine learning algorithms. They have a library for parallel k-means but not for the parallel k-meansTI.

This paper presents a framework to implement parallel k-meansTI on Spark and compare its performance with the Spark ML parallel k-means algorithm by varying the number of clusters and the number of iterations. It also shows how many point-center distance computations the k-meansTI algorithm can skip. Our experimental results show that the k-meansTI algorithm can skip many distance computations and it can be faster than the Spark ML k-means algorithm when the data set is large, does not contain a lot of sparse data, and is high-dimensional.

The structure of this paper is the following: Section 2 provides the background and related work required for this paper. Section 3 presents a framework to implement parallel k-meansTI on Spark and explains how to to implement it. Section 4 shows the experimental work and results. Section 5 discuss about the experimental results and the paper is concluded in Section 6.

II. Background and Related Work

Parallel implementation of an algorithm is one of the ways to improve the performance of the algorithm and there already exist several parallel implementations of k-means and k-meansTI algorithms by using various technologies. For example, parallel k-means algorithm has been implemented using shared-memory architectures such as multi-core CPU, GPU, and multi-threaded Cray XMT platform [9], Message Passing Interface (MPI) [10], MapReduce platform Hadoop [11], and Apache Spark [12]. Only few of the academic papers show the implementation of parallel k-meansTI algorithm. It has been implemented by using multi-threaded architecture [6], OpenMPI [13], and Hadoop MapReduce framework [14]. All these implementations reach a speedup by the parallel k-meansTI algorithm over the parallel k-means algorithm.

This section provides background about k-meansTI algorithm in detail, Apache Spark, and a framework to implement parallel k-means algorithm on Spark, which is a base for designing a framework to implement parallel k-meansTI on Spark.

A. k-means with Triangle Inequality (k-meansTI)

k-meansTI algorithm takes advantage of the fact that only few data instances change their closest cluster center (centroid) in the standard k-means algorithm when it reaches closer to its convergence point. Therefore, instead of computing all the point-center distances, it skips the distance computations based on the upper bound, and lower bounds of the instances. Lower bounds can be obtained by using lemmas as below, where p is a data point, c is a cluster center to which p is currently assigned, and c' is any other cluster center [7].

Lemma 1: If $d(c, c') \geq 2d(p, c)$ then $d(p, c') \geq d(p, c)$.
Lemma 2: $d(p, c) \geq max\{0, d(p, c') - d(c, c')\}$.

According to Lemma 1, $d(p, c')$ is computed only when $d(p, c) > \frac{1}{2}d(c, c')$. If $d(p, c)$ is unknown (happens after updating the new cluster centers) but the upper bound (u) is known then $d(p, c')$ and $d(p, c)$ are computed when $u > \frac{1}{2}d(c, c')$. If $u \leq \frac{1}{2}\text{min}(d(c, c'))$, there is no need to compute the distance of p with other cluster centers.

Lemma 2 is used when the cluster centers are updated. Let c'' be a cluster center of c cluster center from the previous iteration. The lower bound of an instance p for c can be inferred as $l = max\{0, d(p, c'') - d(c'', c)\}$. Here, $d(c'', c)$ gives the distance moved by a cluster center from the previous iteration to the current iteration.

B. Apache Spark

Spark currently supports three kinds of cluster managers: Standalone Cluster Manager, Apache Mesos, and Hadoop YARN and it provides APIs in Scala, Java, Python, and R high-level programming languages. It includes components such as Spark SQL, Spark MLlib, Spark ML, Spark Streaming, and GraphX. Spark SQL improves the performance of Spark and it handles data as $DataFrames$. Spark MLlib and Spark ML are Spark's two machine learning packages. Spark ML provides a higher-level API than Spark MLlib that allows users to easily create the machine learning pipelines. Spark streaming can be used for streaming analytics on minibatches of data. GraphX is used for graph computations [8].

When Spark is installed in a cluster, it allows users to write a driver program in a master node that can perform parallel operations. Data on Spark are represented as RDDs (Resilient Distributed Datasets), which is a collection of data that are stored in the executors or slave nodes. The Spark cluster manager (in a master node) handles starting and distributing the Spark executors across a distributed system.

Spark executes RDDs in a lazy way. It computes the partitions only when an action such as write and collect is called. These actions triggers the scheduler to build Directed Acyclic Graph (DAG) and Spark evaluates these actions by working backward following the DAG [8].

C. Parallel k-means on Spark

Fig. 1 depicts a framework to parallelize k-means based clustering algorithms on Spark. In this framework, data from Hadoop Distributed File System (HDFS) is loaded into RDDs. This results in distributing the data across multiple machines.

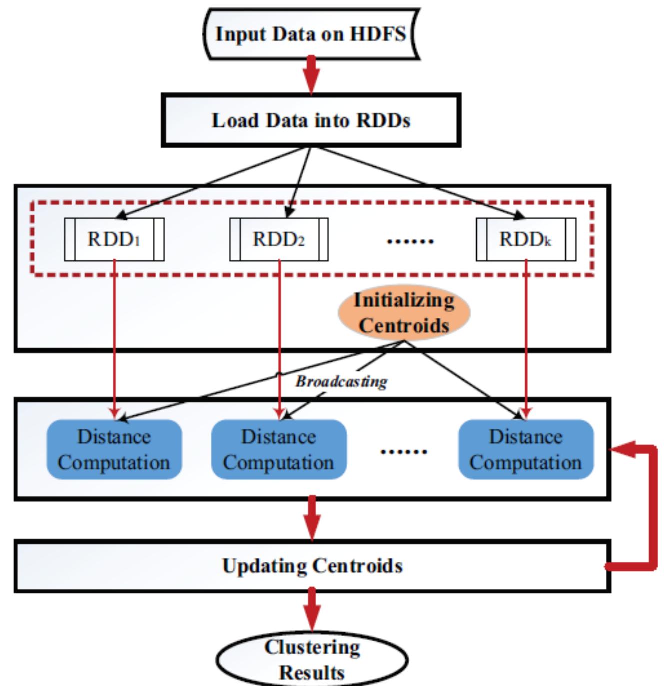


Fig. 1. Framework Overview of Parallel k-means on Spark [15]

Selecting initial cluster centers (initializing centroids) is done on a single machine. Wang et. al. [15] offered three strategies for seeding: i) randomly selecting k data points, ii) sequentially selecting k instances based on probability (k-means++ [16]), and iii) selecting seeds in parallel (k-means|| [17]). The initial cluster seeds are broadcast to all the nodes for point-center distance computations and to assign the instances to their closest cluster center, which are then aggregated in the

master node to update the new cluster centers. Broadcasting wraps the value of a variable and makes sure that it is distributed to the slave nodes only once [18]. This helps in improving the speed of the algorithm. The process of assigning the instances to their closest cluster center and updating the new cluster centers is repeated until some stopping conditions are met. During these iterations, the new cluster centers are broadcast to all the nodes instead of the initial cluster centers.

Squared Euclidean distance, Cosine distance, and KL-divergence distance are the distance functions that can be used in the k-means algorithm, where squared Euclidean distance is the most popular one. Computing the point-center distance is the most time consuming step in the k-means algorithm. However, its implementation on Spark is faster, if one of the vectors in distance computation is sparse.

In Spark, all the instances of the data are converted to `VectorsWithNorm` format before applying the algorithm. This means, all the instances are in vector format and their norms (p-norm, where p=2) are computed in advance. The vector norms are then used to compute the faster squared Euclidean distance. For example, Spark MLlib (for example, version 2.3) provides an API for computing faster squared Euclidean distance by using following formula:

$\|a-b\|_2^2 = \|a\|_2^2 + \|b\|_2^2 - 2a^T b$

This formula is computed until it does not introduce too many numerical errors. Here, $\|a\|_2^2$ and $\|b\|_2^2$ are the squared norm 2 of the vectors a and b, respectively. This is faster than computing the squared Euclidean distance between the vectors directly.

While updating cluster centers, Spark uses a $key - value$ pair representation in each node to store unique clusters as key and the sum of all the instances belonging to a specific cluster center as $value$. A reduce function is then used to compute the vector sum of all the instances from all the nodes for each cluster center. New cluster centers are collected by taking the average of their summed vectors.

III. Implementing Parallel k-meansTI on Spark

Fig. 2 shows a framework to implement parallel k-meansTI on a standalone Spark cluster. In this framework, input data can be provided from any file format such as CSV (Comma Separated Values) and text. These data are loaded into RDDs and computations such as selecting initial cluster centers (see Subsection III-B), inter-cluster distances, and minimum cluster distance for each cluster (see Subsection III-C) are done in a single node. Elkan's k-meansTI algorithm skips distance computations of an instance based on its lower bounds and upper bound values. Therefore, the data inside each RDD are extended to include their lower bounds, upper bound, and nearest cluster center. This can be done in parallel (see Subsection III-D).

The next step is to broadcast the initial cluster centers, inter-cluster distances, and minimum cluster distance for each cluster to all the nodes in order to compute point-center distances in parallel. k-meansTI follows the theory mentioned in the Subsection II-A to skip distance computations between

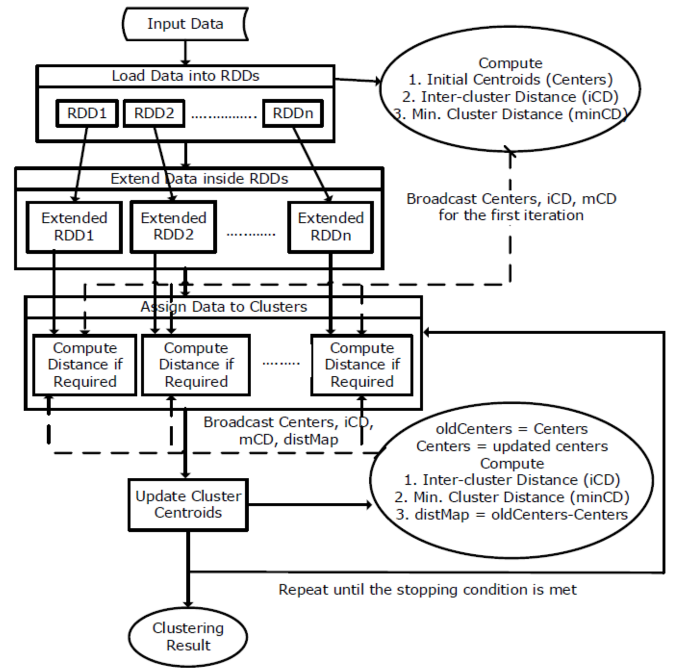


Fig. 2. Framework Overview of Parallel k-meansTI on Spark

an instance and the cluster centers. After computing distances and assigning data to their nearest cluster center, the cluster centers are updated by taking average of the instances inside each cluster center. This is also done in parallel. In the next iteration, the inter-cluster distances, minimum cluster distance for each cluster, and the movement of a cluster center from the previous iteration to the current iteration are computed. These variables are then broadcast with the new cluster centers to all the nodes for distance computations. This process is repeated until some stopping conditions are met. After reaching the stopping condition, the clustering result is obtained, which contains the final cluster centers.

A. Load Data into RDDs

The first step while applying parallel k-meansTI algorithm on Spark is to load data into RDDs. This is necessary to distribute data into multiple slave nodes and to perform parallel processing on them. This can be done by using `SparkContext` or `SparkSession`. These are the API's gateway to Spark for any application. An application begins when one of these is started [8]. After loading data into RDDs, all the instances are converted to `VectorWithNorm` format. This is done for faster point-center distance computations.

B. Selecting Initial Centers

Our parallel implementation of k-meansTI algorithm on Spark includes the existing functionality from Spark MLlib k-means to select initial cluster centers. There are mainly two types of methods to select initial cluster centers. The first one is by random selection. This method randomly selects k instances from the list of all the instances. The other is k-means$\|$ [17], which is a parallel version of k-means++ method

[16]. This method selects cluster centers that are far away from each other based on the probability proportional to the squared distance to the current cluster set. It is also possible to set a seed in order to get the same cluster centers every time.

C. Computing Inter-cluster and Minimum Cluster Distances

Inter-cluster distance is basically a center-center distance computation for all the cluster centers ($iCD_{i,j}$, where $i, j = 0 \ldots k-1$ and $i \neq j$). Our implementation uses squared Euclidean distance to measure the inter-cluster distances. Minimum cluster distance is also calculated for all the cluster centers. Minimum cluster distance for a cluster c is half the minimum inter-cluster distance between c and other cluster centers. It is computed as $minCD_i = 0.5 * min(iCD_{i,j})$, where $i, j = 0 \ldots k-1$ and $i \neq j$. These distances are broadcast to all the slave nodes and they are used in conditions to skip distances.

D. Extend Data inside RDDs

Our algorithm extends the structure of all the data instances to include their lower bounds, upper bound, and the closest cluster center values. Fig. 3 shows the initialization of this extended structure for all the data instances.

An Instance <VectorWithNorm>	Lower Bounds $<lb_{0 \ldots k-1} = 0>$	Upper Bound <ub = 0>	Closest Cluster <cid = -1>

Fig. 3. Initializing an extended instance

The extended data structure includes a data point in `VectorWithNorm` format, its lower bounds with all the cluster centers (lb_i, where $i = 0 \ldots k-1$), an upper bound (ub), and its closest cluster center (cid). A lower bound of an instance with a cluster center is a squared Euclidean distance between them. Upper bound stores the distance of the instance with its closest cluster center. While initializing this structure, all the lower bounds and the upper bound are assigned 0 value, and the closest cluster center is set to -1. These values are updated in every iteration.

E. Assign Data to Clusters

Elkan's k-meansTI algorithm applies some conditions to skip point-center distance computations, before assigning instances to their closest cluster centers. Similar to the parallel k-means on Spark, this process is done sequentially within the RDDs but in parallel between the RDDs. k-meansTI algorithm applies Lemma 1 rule in the first iteration and both Lemma 1 and Lemma 2 rules are applied in further iterations to skip unnecessary distance computations and to compute lower bounds (see Subsection II-A). This algorithm also follows the same distance computation method as in parallel k-means on Spark.

In the first iteration, for all the instances p within each slave node, squared Euclidean distance is computed with a cluster center C_i, $d(p, C_i)$. The distance of p is computed to other cluster centers C_j only when $d(p, C_i) > 0.5 * iCD_{i,j}$, where $i, j = 0 \ldots k-1$ and $i \neq j$. Whenever this condition is satisfied, the distance $d(p, C_j)$ is compared with $d(p, C_i)$. If $d(p, C_j)$ is smaller than $d(p, C_i)$, the closest cluster center of p becomes C_j, instead of C_i. Lower bound values of p for a cluster center are updated whenever a distance is computed between them and the upper bound is updated with the distance between p and its closest cluster center.

In other iterations, for all the instances p within each slave node, first the lower bounds are updated as $d(p, C_i) = max(0, lb_{C'_i} - distMap_{C_i, C'_i})$, where $i = 1 \ldots k-1$, C'_i is the cluster center of C_i from the previous iteration, and $distMap_{C_i, C'_i}$ is the distance moved by this cluster center from the previous iteration. Then the upper bound (ub) is updated with an addition to the distance moved by the closest cluster center from the previous iteration. After this, three different nested conditions to skip distance computations are applied to each instance.

The first condition skips the distance computations of an instance with all other cluster centers, if its $ub \leq minCD_{cid}$, where cid is the closest cluster center of the instance from the previous iteration. When the first condition is not satisfied, the algorithm attempts to skip distance computations with the cluster centers C_i, if $ub \leq iCD_{cid, C_i}$ or $ub \leq lb_{C_i}$, where $i = 0 \ldots k-1$ and $i \neq cid$. The upper bound value is updated with the distance of the instance and the first cluster center for which the second skip condition is not satisfied. The third condition is just a repetition of the second condition but with the updated upper bound value. The distance of the instance is computed with all the cluster centers for which the third condition is not satisfied. If the distance is smaller than the upper bound value, this distance becomes the upper bound value and the cluster center at that point becomes the closest one. Lower bounds are updated whenever a distance is computed.

F. Updating Cluster Centers

The process of updating cluster centers in each iteration is exactly the same as in parallel k-means algorithm on Spark. It uses the $key - value$ pair to store unique cluster centers and their summed vectors in each node and then they are reduced to combine all the $key - value$ pairs from all the nodes. The new cluster centers are updated by taking the average of the summed vectors for each cluster center.

G. Stopping Conditions

Two different conditions are defined as the stopping conditions in the implementation of the parallel k-meansTI clustering algorithm on Spark. The first one is the maximum number of allowed iterations. By default, it is 20. Another condition is the difference between cluster centers from the previous iteration and the current iteration. The algorithm is stopped, if the squared distance of any one of the cluster centers from the previous iteration is less than or equal to 0.0001.

IV. Experimental Work

We first implemented parallel k-meansTI algorithm on Spark on top of RDDs by using Spark MLlib APIs, following

the framework shown in Section III. Since Spark ML provides high-level API for data pipeline from data preparation to model training, we wrapped this algorithm by using Spark ML APIs.

All the pipeline stages on Spark ML are grouped into estimators (algorithms that require training) and transformers (algorithms that don't require training) [8]. We implemented an estimator for parallel k-meansTI algorithm, used various Spark ML transformers from Spark ML for data preparation, and used our estimator for clustering.

A. Experimental Setup

A Standalone Spark Cluster was created on a cloud platform with one master node and 7 slave nodes, each node with Spark 2.3 version to perform experiments. The master node had 60GB disk space, 8 GB RAM, and 4 virtual CPUs. All the slave nodes had 40GB disk space, 4 GB RAM, and 2 virtual CPUs. The algorithms in our standalone cluster use 2 cores and 2GB memory from each slave node, whereas they use 4GB memory from the master node.

B. Data sets

This subsection includes six different data sets that are used in the experiment (see Table I). These data sets are selected based on the variety of the number of feature dimensions and the number of instances. Since the type of digital evidence depends on the context, any digital data can be used for the purpose of testing our methods. Therefore, we could use a variety of data sets, not necessarily related to malware or intrusion (for example, Secom [19] or MNIST [20]).

TABLE I
DATASETS FOR THE EXPERIMENT

Data sets	Number of Attributes	Number of Instances
BaIoT [19]	115	1009145
Secom [19]	590	1372
MNIST [20]	785	70000
KDDCup99 [21]	41	1048575
KDDCup98 [22]	481	95412
KDDCup98-Big	962	95412

1) BaIoT Data Set: This data set contains Internet of Things (IoT) traffic data, collected from 9 commercial IoT devices authentically infected by Mirai and BASHLITE. This data set was created to detect network-based IoT botnet attacks by using deep autoencoders [23]. In this experiment, data set from Danmini_Doorbell folder is used. It has 1009145 instances and 115 attributes.

2) Secom Data Set: This data set contains data from a semi-conductor manufacturing process. A file *secom.data* [19] was downloaded for the experiment and it has total 1372 instances and 590 attributes.

3) MNIST Data Set: This data set is a subset of a larger set avaialble from NIST. It contains handwritten digits with 60000 training instances and 10000 test instances [20]. We combined both training and test data for the experiment. This data set includes 785 attributes.

4) KDDCup99 Data Set: This data set was used for *The Third International Knowledge Discovery and Data Mining Tools Competition*, which was held in conjunction with KDD-99. It contains a variety of intrusions simulated in a military network environment [21]. This is an old data set but since the foucs of this paper is on examining the speed of the parallel k-meansTI on Spark, this data set is suitable for our experiment. A compressed file *kddcup.data.gz* was downloaded for the experiment and it includes 1048575 instances and 41 attributes.

5) KDDCup98 Data Set: This data set was used for the *The Second International Knowledge Discovery and Data Mining Tools Competition*, which was held in conjunction with KDD-98 [22]. A zipped file *cup98LRN.zip* was downloaded. This data set has 95412 instances and 481 attributes.

6) KDDCup98-Big Data: This data set is created by copying the features from KDDCup98 data set twice, such that it has double features than KDDCup98 and has the same number of instances. This is done to check the performance of the algorithms on a high-dimensional data set. This data set contains total 95412 instances and 962 attributes.

C. Data Preparation

All the data sets we collected already had attributes and their values were in the CSV format. Some of these data sets had categorical attributes, which is not supported by the clustering algorithms and some of the attributes were empty. Following are some of the pre-processing steps performed on the downloaded data sets in order to make them suitable for the clustering algorithms on Spark:

- *Converting categorical attributes into numerical*: Spark ML `StringIndexer` transformer was used to encode all the string attributes to numeric attributes. The numbers range from 0 to the number of labels, ordered by the frequencies of the labels. The most frequent label gets 0 number. After converting categorical attributes to numerical, categorical attributes were dropped from the DataFrames.
- *Converting null values to 0*: When the data is loaded in Spark, Spark interprets empty values as null. Since null is not a numerical value, we converted them to 0.
- *Converting all numerical attributes to double*: This is done to create dense vectors for all the instances of the data sets.

D. Applying parallel k-meansTI on Spark

We first created a `SparkSession`, which is similar to the `SparkContext` in Spark MLlib. The `SparkSession` is an entry point for Spark SQL and it is created by using the `builder` pattern and `getOrCreate()` as follows:

```
val spark = SparkSession
            .builder()
            .appName(name="k-meansTI")
            .master(master="local")
            .getOrCreate()
```

The `getOrCreate()` returns an existing session if one is already running. We can specify the path of a master in

the builder `master(name=<value>)` option. Spark runs locally, if its value is `local`, it runs locally with 5 cores when the value is `local[5]`, and it runs on a Spark standalone cluster, if a Spark master URL is provided as its value.

The next step was to load the pre-processed data from CSV files into DataFrames. Below script shows how to do it by using the `spark SparkSession` variable. The `spark` variable reads CSV including headers only when the header option is `true`.

```
val dfTraining = spark
            .read
            .format(source="csv")
            .option("header","true")
            .option("inferSchema",true)
            .load(filename)
```

After loading the data into DataFrames, a Spark ML transformer `VectorAssembler()` was used to convert data into vectors. We set all the columns of the DataFrames as input columns to the `VectorAssemer()` transformer and `Features` column as its output column. At the end of this script, all the data are in vector format under the `Features` column of the DataFrames. Below script shows the use of `VectorAssembler()`:

```
val assembler = new VectorAssembler()
            .setInputCols(dfTraining.columns)
            .setOutputCol("Features")
```

The next step was to create an object of the parallel k-meansTI model. We set the number of clusters by using `setK()` option, maximum iterations by using `setMaxIter()`, feature columns by using `setFeatureCol()`, and prediction output column by using `setPredictionCol()` as below:

```
val kmeansTI = new KMeansTI()
            .setK(10)
            .setMaxIter(5)
            .setFeatureCol("Features")
            .setPredictionCol("Prediction")
```

Before training the data set, pipeline stages have to be created and they are used to train the model. The order of setting the pipeline stages is important, since these stages are run in order. Below scripts show how to use pipeline and fit functionalities. The `fit()` function fits the pipeline to the training DataFrame, where the DataFrame is transformed through each stage.

```
val pipeline = new Pipeline()
        .setStages(Array(assembler, kmeansTI))
val model = pipeline.fit(dfTraining)
```

Below script shows how to collect final cluster centers and predict the testing DataFrame (dfTesting) by using `transform()` option of the training model.

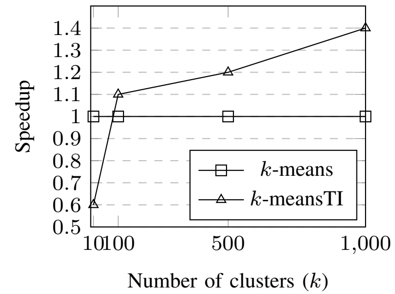
```
val centers = model
            .stages(2)
            .asInstanceOf[KMeansTIModel]
            .clusterCenters
val result = model.transform(dfTesting)
```

E. Experiments and Results

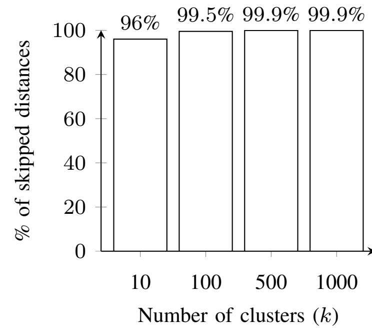
This subsection shows the comparison of the performance of our implementation of parallel k-meansTI algorithm on Spark with the Spark ML parallel k-means algorithm. These two algorithms are compared in three different ways for each data set listed in the Table I.

In the first case, the speedup of parallel k-meansTI on Spark is compared with the Spark ML k-means by varying the number of clusters from 10 to 1000, keeping the maximum number of iterations to 5. The second case includes the computation of the number of skipped distances in percentage by parallel k-meansTI on Spark for the same setting as in the first case. The third case includes the speedup comparison of these algorithms when the maximum number of iterations was varied. In this case, the number of clusters was 100, and the number of maximum iterations were varied from 5 to 50.

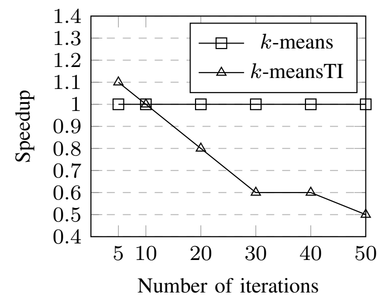
We executed both algorithms 31 times for the first and the third cases of all the data sets. The final speed is computed as an average of these 31 outputs. While executing the algorithms, a seed was initialized and k-means$\|$ algorithm was used to get the same initial cluster centers for both algorithms.



(a) Varying the number of clusters



(b) % of skipped distances from Fig. 4(a)

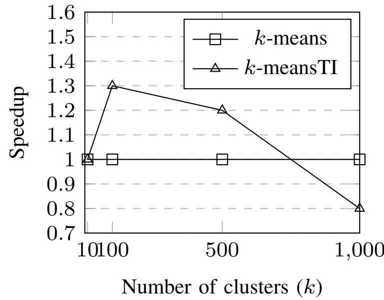


(c) Varying the number of iterations

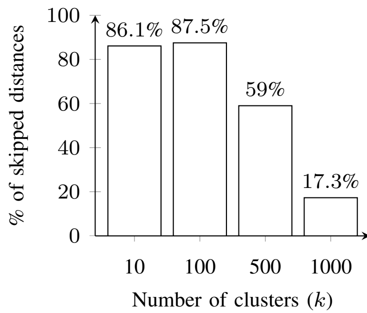
Fig. 4. Applying k-means and k-meansTI on BaIoT data set.

Fig. 4 shows the experimental results obtained on BaIoT

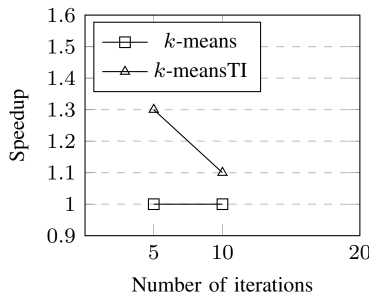
data set by applying Spark ML k-means algorithm and our implementation of parallel k-meansTI algorithm on Spark. These results show that the speed of parallel k-meansTI algorithm on Spark was slower than the speed of Spark ML k-means algorithm on BaIoT data set for a small number of cluster centers (k=10). However, it outperformed Spark ML k-means algorithm with the increase of the number of clusters (see Fig. 4(a)). The number of skipped distances is also very high in this case and it increased with the number of cluster centers (see Fig. 4(b)). The speed of parallel k-meansTI algorithm on Spark decreased with the increasing number of iterations (see Fig. 4(c)).



(a) Varying the number of clusters



(b) % of skipped distances from Fig. 5(a)



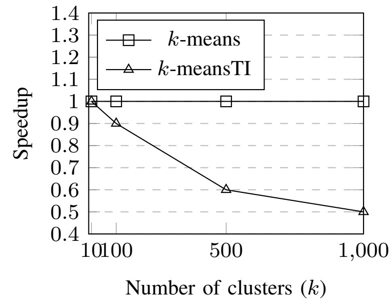
(c) Varying the number of iterations

Fig. 5. Applying k-means and k-meansTI on Secom data set.

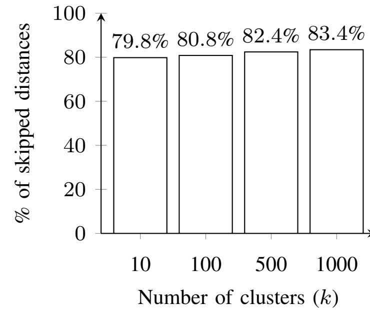
Fig. 5 presents the experimental results of applying these clustering algorithms on Spark for Secom data set. Both algorithms had the same speed when the number of clusters used in the algorithms was 10, but parallel k-meansTI on Spark outperformed Spark ML k-means with the increase of the number of clusters. Fig. 5(a) shows a drop in speed when the number of clusters was 500 and 1000. In these cases, the maximum number of iterations was 5, but the algorithms converged in 3 iterations when the number of clusters was 500 and in 1 iteration when it was 1000.

Fig. 5(b) shows the number of skipped distances in percentage by these algorithms on Secom data set. Due to the fast convergence of the algorithms for $k = 500$ and $k = 1000$, the percentage of skipped distances were 59% and 17.3% for these cases, respectively. The percentage of the number of skipped distances increased up to 87.5% when the algorithms did not converge before 5 iterations.

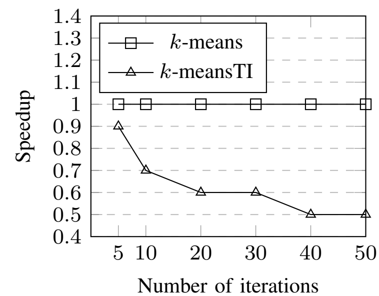
When the number of iterations were varied, both algorithms converged very fast. Therefore, we compared their performance up to maximum 10 iterations only. Fig. 5(c) shows that the speed of parallel k-meansTI is better than Spark ML k-means for both iterations. However, there is a drop in speed when the iterations was increased.



(a) Varying the number of clusters



(b) % of skipped distances from Fig. 6(a)

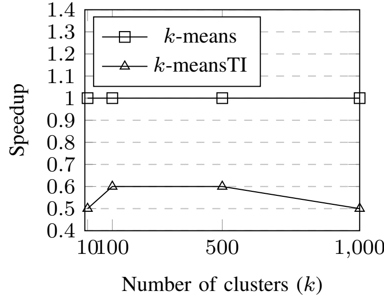


(c) Varying the number of iterations

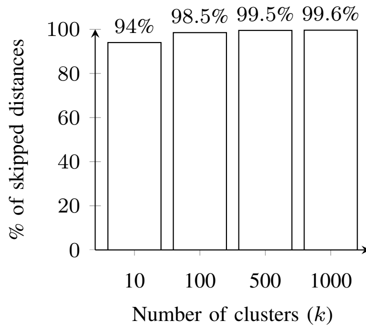
Fig. 6. Applying k-means and k-meansTI on MNIST data set.

Fig. 6 shows the experimental results obtained by the

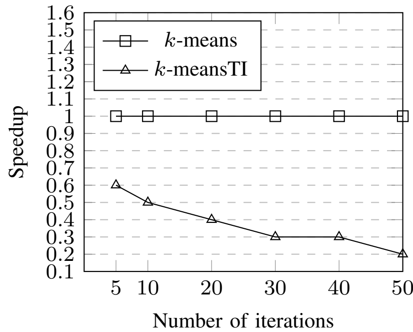
algorithms on the MNIST data set. The speed of parallel k-meansTI on Spark is similar to the speed of Spark ML k-means for the small number of clusters ($k = 10$). Otherwise, Spark ML k-means outperformed our parallel k-meansTI on Spark for the larger number of cluster centers for the MNIST data set. Fig. 6(b) shows that the parallel k-meansTI has skipped distances up to 83%. When the number of iterations was increased, the speed of parallel k-meansTI on Spark also decreased (see Fig. 6(c)).



(a) Varying the number of clusters



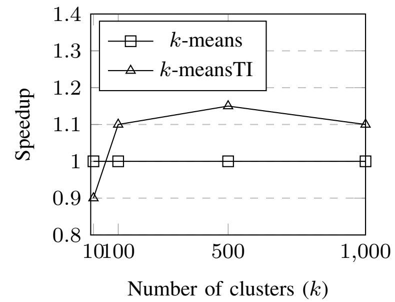
(b) % of skipped distances from Fig. 7(a)



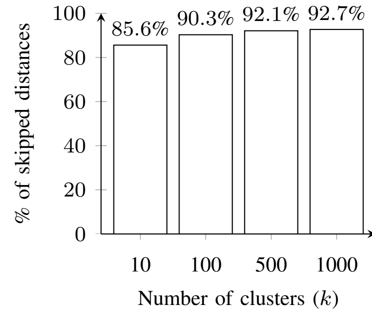
(c) Varying the number of iterations

Fig. 7. Applying k-means and k-meansTI on KDDCup99 data set.

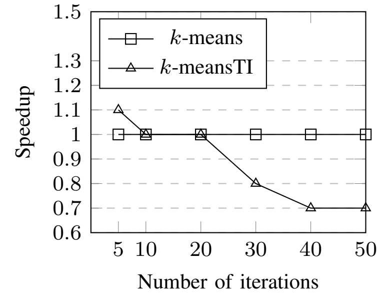
Fig. 7 shows that the parallel k-meansTI algorithm on KDDCup99 skipped many distance computations, but it performed worse than the Spark ML k-means algorithm for all the number of clusters. The difference in their speed increased with the increasing number of clusters. Its speed became even worse when the number of clusters was constant (k=100) and the number of iterations was varied (see Fig. 7(c)).



(a) Varying the number of clusters



(b) % of skipped distances from Fig. 8(a)

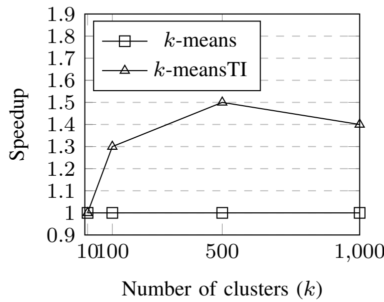


(c) Varying the number of iterations

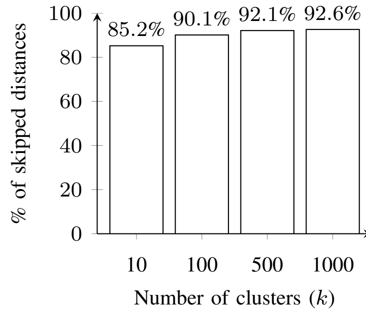
Fig. 8. Applying k-means and k-meansTI on KDDCup98 data set.

Fig. 8 shows that the parallel implementation of k-meansTI on Spark performed better compared to the Spark ML k-means clustering algorithm for the larger number of clusters on the KDDCup98 data set. It also skipped many distance computations and it increased with the increasing number of clusters. The percentage of skipped distances was more than 85% when the number of clusters was 10 and it increased up to approximately 93% with the increased number of cluster centers. When the iterations were varied and the cluster size was kept constant (see Fig. 8(c)), the speed of parallel k-means decreased with the increase in the number of iterations.

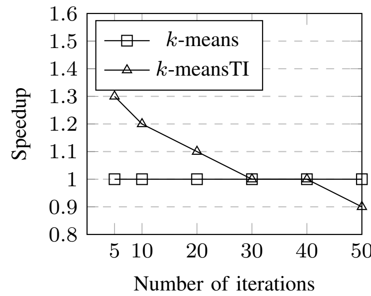
Fig. 9 shows the experimental results on the KDDCup98-Big data set. In this case, the parallel k-meansTI on Spark outperformed Spark ML k-means for all the variations of cluster centers, where it was increased with the larger number of cluster centers (see Fig. 9(b)). The number of skipped distances is similar as on KDDCup98 data set. The speed of our implementation of parallel k-meansTI on Spark decreased



(a) Varying the number of clusters



(b) % of skipped distances from Fig. 9(a)



(c) Varying the number of iterations

Fig. 9. Applying k-means and k-meansTI on KDDCup98-Big data set.

with the increasing number of iterations like in all other data sets (see Fig. 9(c)).

V. Discussion

The experimental results in Subsection IV-E shows that the parallel k-meansTI on Spark can skip many point-center distance computations. However, its speed depends on the type of data set.

Since Spark ML/MLlib uses vector norms for computing point-center distance computations instead of vectors until it does not introduce too many numerical errors, the distance computation is faster when at least one of the vectors is sparse. We can take an example of MNIST data set. The MNIST data set contains 785 attributes and 70000 instances and its data is sparse. When we executed Spark ML parallel k-means and our implementation of parallel k-meansTI on Spark on this data set, the Spark ML k-means outperformed in speed. This shows that, if the data set contains a lot of sparse data, Spark ML k-means outperforms k-meansTI on Spark, regardless of the high dimensionality of the data set, number of instances, and the possibility to skip a large number of point-center distances. The speed of parallel k-meansTI on Spark becomes even worse for such data sets when the number of clusters is increased.

On the contrary, when the data set contains more non-sparse data, is high dimensional, and contains a large number of instances, our implementation of k-meansTI on Spark outperforms the Spark ML k-means. For example, parallel k-meansTI on Spark outperformed Spark ML k-means for the high dimensional data sets (BaIoT had 115 attributes, Secom had 590 attributes, KDDCup98 had 481 attributes, and KDDCup-Big had 962 attributes). Most of the data inside these data sets were not sparse. The speed of parallel k-meansTI on Spark increased with the increasing number of clusters.

The experimental results showed an uniform trend on the speedup of the parallel k-meansTI on Spark when the number of iterations was varied and when the number of clusters was constant. Its speed decreased with the increasing number of iterations for all the selected data sets.

These experimental results suggest that it is better to use Spark ML k-means algorithm when the big data like digital evidence contains mostly sparse data. However, it is better to use parallel k-meansTI algorithm on Spark if the data set does not contain a lot of sparse data.

VI. Conclusion

This paper presented a framework to implement parallel k-meansTI algorithm on Apache Spark. It also compared its performance with the existing Spark ML k-means algorithm on different types of data sets. The experimental results show that the parallel k-meansTI algorithm on Spark can skip many point-center distance computations and is faster than the Spark ML k-means algorithm, if the data set is high dimensional, big in size, and does not contain a lot of sparse data. Spark ML k-means algorithm outperforms our implementation of parallel k-meansTI algorithm on Spark, if the data set is highly sparse, regardless of its size and dimensionality. These results suggest to use parallel k-meansTI algorithm on Spark when the digital evidence is high dimensional and does not contain a lot of sparse data.

References

[1] S. Decherchi, S. Tacconi, J. Redi, A. Leoncini, F. Sangiacomo, and R. Zunino, *Text Clustering for Digital Forensics Analysis*. Springer Berlin Heidelberg, 2009.

[2] L. F. C. Nassif and E. R. Hruschka, "Document Clustering for Forensic Analysis: An Approach for Improving Computer Inspection," *IEEE Transactions on Information Forensics and Security*, vol. 8, pp. 46–54, 2013.

[3] N. L. Beebe and L. Liu, "Clustering digital forensic string search output," *Digital Investigation*, vol. 11, no. 4, pp. 314 – 322, 2014.

[4] I. Riadi, J. E. Istiyanto, A. Ashari, and S. S. Seno, "Log Analysis Techniques using Clustering in Network Forensics," *International Journal of Computer Science and Information Security*, vol. 10, no. 7, 06 2013.

[5] S. Lloyd, "Least squares quantization in PCM," in *IEEE Transactions on Information Theory*, vol. 28, 1982, pp. 129–137.

[6] G. Hamerly and J. Drake, "Accelerating Lloyd's Algorithm for k-Means Clustering," *Springer International Publishing*, pp. 41–78, 2015.

[7] C. Elkan, "Using the Triangle Inequality to Accelerate K-means," in *Proceedings of the Twentieth International Conference on Machine Learning*, ser. ICML'03, 2003, pp. 147–153.
[8] H. Karau and R. Warren, *High Performance Spark*. O'Reilly, 2017.
[9] P. Mackey and R. R. Lewis, "Parallel k-Means++ for Multiple Shared-Memory Architectures," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 93–102.
[10] J. Zhang, G. Wu, X. Hu, S. Li, and S. Hao, "A Parallel K-Means Clustering Algorithm with MPI," in *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, Dec 2011, pp. 60–64.
[11] W. Zhao, H. Ma, and Q. He", "Parallel K-Means Clustering Based on MapReduce," in *Cloud Computing*. Springer Berlin Heidelberg, 2009, pp. 674–679.
[12] B. Wang, J. Yin, Q. Hua, Z. Wu, and J. Cao, "Parallelizing K-Means-Based Clustering on Spark," in *2016 International Conference on Advanced Cloud and Big Data (CBD)*, Aug 2016, pp. 31–36.
[13] R. Krohn and C. Karlsson, "Parallel K-Means Clustering with Triangle Inequality," *ISCA, CAINE*, 2016.
[14] S. A. Ghamdi and G. D. Fatta, "Efficient Parallel K-Means on MapReduce Using Triangle Inequality," in *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Nov 2017, pp. 985–992.
[15] B. Wang, J. Yin, Q. Hua, Z. Wu, and J. Cao, "Parallelizing K-Means-Based Clustering on Spark," in *2016 International Conference on Advanced Cloud and Big Data (CBD)*, Aug 2016, pp. 31–36.
[16] D. Arthur and S. Vassilvitskii, "K-means++: The Advantages of Careful Seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07, 2007, pp. 1027–1035.
[17] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable K-means++," *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 622–633, mar 2012.
[18] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, pp. 10–10.
[19] D. Dua and K. T. Efi, "UCI Machine Learning Repository," https://archive.ics.uci.edu/ml/datasets.html, 2017.
[20] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST Database," http://yann.lecun.com/exdb/mnist/, 2017, [Accessed: July 2018].
[21] KDDCup99, "KDD Cup 1999 Data," http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 2018, [Accessed: July 2018].
[22] KDDCup98, "KDD Cup 1998 Data," https://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html, 2018, [Accessed: July 2018].
[23] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, D. Breitenbacher, A. Shabtai, and Y. Elovici, "N-BaIoT: Network-based Detection of IoT Botnet Attacks Using Deep Autoencoders," *CoRR*, vol. abs/1805.03409, 2018.