Fast and Exact Out-of-Core and Distributed K-Means Clustering

Ruoming Jin Anjan Goswami Gagan Agrawal

Department of Computer Science and Engineering
Ohio State University
{jinr,goswamia,agrawal}@cse.ohio-state.edu

Abstract

Clustering has been one of the most widely studied topics in data mining and k-means clustering has been one of the popular clustering algorithms. K-means requires several passes on the entire dataset, which can make it very expensive for large disk-resident datasets. In view of this, a lot of work has been done on various approximate versions of k-means, which require only one or a small number of passes on the entire dataset.

*In this paper, we present a new algorithm, called Fast and Exact K-means Clustering (*FEKM*), which typically requires only one or a small number of passes on the entire dataset, and provably produces the same cluster centers as reported by the original k-means algorithm. The algorithm uses sampling to create initial cluster centers, and then takes one or more passes over the entire dataset to adjust these cluster centers. We provide theoretical analysis to show that the cluster centers thus reported are the same as the ones computed by the original k-means algorithm. Experimental results from a number of real and synthetic datasets show speedup between a factor of 2 and 4.5, as compared to k-means.*

This paper also describes and evaluates a distributed version of FEKM*, which we refer to as* DFEKM*. This algorithm is suitable for analyzing data that is distributed across loosely coupled machines. Unlike the previous work in this area,* DFEKM *provably produces the same results as the original k-means algorithm. Our experimental results show that* DFEKM *is clearly better than two other possible options for exact clustering on distributed data, which are down-loading all data and running sequential k-means, or running parallel k-means on a loosely coupled configuration. Moreover, even in a tightly coupled environment,* DFEKM *can outperform parallel k-means if there is a significant load imbalance.*

1. Introduction

Clustering has been one of the most widely studied topics in data mining. Clustering refers to techniques for grouping similar objects in clusters. Formally, given a set of d dimensional points and a function $f : \Re^d \times \Re^d \mapsto \Re$ that gives the *distance* between two points in $\Re^d$, we are required to compute k cluster centers, such that the points falling in the same cluster are similar and points that are in different cluster are dissimilar.

Most of the initial clustering techniques were developed by statistics or pattern recognition communities, where the goal was to cluster a modest number of data instances. However, within the data mining community, the focus has been on clustering large datasets. Developing clustering algorithms to effectively and efficiently cluster rapidly growing datasets has been identified as an important challenge. For example, Ghosh states *"The holy grail of scalable clustering can be to find near linear algorithm that involve only a small number of passes through the database"* [13].

In this paper, we address the problem of *fast data clustering* on a very large and out-of-core datasets, using one or a small number of passes over the data, *without compromising on its result*. Our work is in the context of k-means clustering. K-means clustering algorithm was developed by MacQueen [22] in 1967 and later improved by Hartigan [16]. Bottou and Bengio [3] proved the convergence properties of the k-means algorithm. It has been shown to be very useful for a corpus of practical applications. The original k-means algorithm works with memory resident data, but can be easily extended for disk-resident datasets.

The main problem with the k-means algorithm is that it makes one scan over the entire dataset for every iteration, and it needs many such iterations before converging to a quality solution. This makes it potentially very expensive to use, particularly for large disk-resident datasets. A number of algorithms or approaches focus on reducing the number of passes required for k-means [4, 5, 10, 14]. However, these approaches only provide approximate solutions, possibly with deterministic or probabilistic bounds on the quality of the solutions. A key advantage of k-means has been that it converges to a local minimum [15], which does not hold true for the approximate versions.

Therefore, an interesting question is, *"Can we have an algorithm which requires fewer passes on the entire dataset, and can produce the same results as the original k-means algorithm?"*. In this paper, we present an algorithm that makes one or a few passes over the data and produces the exact cluster centers as would be generated by the original k-means algorithm. We refer this algorithm as Fast and Exact K-Means algorithm, denoted by FEKM.

The main idea in the algorithm is as follows. We initially sample the data and run the original k-means algorithm. We store the centers computed after each iteration of the run of the k-means on the sampled data. We now use this information and take one pass over the entire dataset. We identify and store the points which are more likely to shift from one cluster to another, as the cluster centers could move. These points are now used to try and adjust the cluster centers.

We provide theoretical analysis to show that the algorithm produces the same cluster centers. In the worst case, the algorithm can require the same number of passes as the original k-means. However, our detailed experimental analysis on several synthetic and real datasets shows that it requires at most 3 passes, whereas, the average number of passes required is less than 1.5. This results in speedups between 2 and 4.5 as compared to the original k-means.

This paper also describes and evaluates a distributed version of FEKM, which we refer to as DFEKM. This algorithm is suitable for analyzing data that is distributed across loosely coupled machines. Unlike the previous work in this area, DFEKM provably produces the same results as the original k-means algorithm. Our experimental results show that DFEKM is clearly better than two other possible options for exact clustering on distributed data, which are downloading all data and running sequential k-means, or running parallel k-means on a loosely coupled configuration. Moreover, even in a tightly coupled environment, DFEKM can outperform parallel k-means if there is a significant load imbalance.

The outline of the rest of the paper is as follows. In Section 2, we discuss the related work. In Section 3, we present the main ideas of the FEKM algorithm and explain the details of the pseudo code of the algorithm. In Section 4, we provide the theoretical framework for the algorithm. The experimental results from FEKM are provided in Section 5. The distributed version of the FEKM and its evaluation is presented in Section 6. We conclude in Section 7.

2. Related Work

There has been an extensive study on clustering algorithms in the literature. Comprehensive survey on this subject can be obtained from the book [17] and papers [2, 13]. In this discussion, we limit ourselves to the improvements over k-means.

Moore and Pelleg [26] proposed a variant of k-means using a k-d tree based data structure to store distance information, which can make each iteration of k-means significantly faster. This algorithm focuses on the in-core datasets.

Bradley and Fayyad [4] have developed a single pass approximation of multi-pass k-means. This algorithm summarizes the input points based on their likely-hood to belong to different centers. Farnstorm and his colleagues [11] have further refined this idea.

Domingos and Hulten [10] proposed a faster version (sublinear) of k-means using sampling based on Hoeffding or similar statistical bound. The algorithm consists of a number of runs of k-means with sample where in every iteration sample size is increased to maintain the loss bound from the multi-pass k-means. The goal here is to converge to a solution which is close to that of a multi-pass k-means by a predefined bound with good probability. Motwani, Charikar, and their colleagues [6, 1, 24, 14, 5] proposed a series of constant factor approximation algorithms for one pass k-center and k-median problems.

More recently, Nittel *et. al.* [23] propose to apply k-means algorithm to cluster massive datasets, scanning the dataset only once. Their algorithm splits the entire dataset into chunks, and each chunk can fit into the main memory. Then, it applies k-means on each chunk of data, and merge the clustering results by another k-means type algorithm. Good results are shown for a real dataset, however, no theoretical bounds on the results are established.

All of the above efforts on reducing the number of passes on the data involve algorithms that cannot maintain the exact result which will be obtained using a multi-pass k-means algorithm. Developing an algorithm with this goal is the focus of our work.

3. Algorithm Description

This section describes new algorithm, Fast and Exact K-means (FEKM) that we have developed. Initially, we describe the main ideas behind the algorithm. Then, we give some formal definitions, present and explain the pseudo-code, and explain some of the choices we have made in our current implementation.

3.1. Main Ideas

The basic idea behind our algorithm is as follows. We believe that approximate cluster centers computed using sampling can be corrected and moved to *exact* cluster centers using only one or a small number of passes on the entire data. By exact cluster centers, we mean the cluster centers that are computed by the original k-means algorithm. Thus, we can use sampling to speedup the computation of exact clusters.

There are three key questions to be addressed. First, when approximate cluster centers are computed using sampling, what information need to be stored. Second, how can this information be used to avoid a large number of passes on the entire dataset. Third, how do we know that we have computed the same cluster centers as in the original k-means algorithm.

We initially run the k-means algorithm on a sample, using the same convergence criteria and same initial points as we would use for the k-means. The following information is stored for future use. After every iteration of k-means on the sampled data, we store the k centers that have been computed. In addition, we compute and store another value, referred to as the *Confidence Radius* of each cluster, whose computation will be described later. This information can be stored in a table with k columns, and the number of rows equaling the number of iterations for which k-means was run on the sampled data. Each entry of the table contains a tuple (center, radius) for each cluster.

Next, we take one pass through the entire dataset. For every point and each row of the table, we compute the cluster to which this point will be assigned at this iteration, assuming that executing the algorithm on the entire dataset produces the same cluster centers as the initial run on sampled data. Next, we try to estimate how likely it is that this point will be assigned to a different cluster when the algorithm is executed on the entire dataset.

Our goal is to identify and store the points which could be assigned to a different cluster during any of the iterations. These points are refereed to as *boundary points*, because intuitively, they fall at the boundary of the clusters. If these points could be identified and stored in memory, we can eliminate any need for any further passes on the entire dataset. However, we can only estimate these points, which means that we could require additional passes if our estimate is not correct.

Thus, for a given point and row of the table, we determine if this point is a boundary point. If it is, it is stored in a buffer. Otherwise, we update the *sufficient statistics tuple*, which has the number and sum of the data points for the cluster.

After the pass through the dataset and storing the boundary point, we do the following processing. Starting from the first row of the table, we recompute centers using the boundary points and sufficient statistics tuple. If any of the new computed centers fall outside the pre-estimated confidence radius which means that our computation of boundary points is not valid, we need to take another pass through the data. We use the new centers as new initialization points and again repeat all the steps. However, if the new computed centers are within the confidence radius, we use these centers for the next iteration and continue. The key observation is that using cluster centers from sampling, boundary points, and sufficient statistics, we are able to compute the same cluster centers that we would have gotten through one pass on the entire dataset. Finally, the algorithm terminates by checking for the same termination condition that one would use in the original algorithm.

3.2 Formal Definitions

This subsection formalizes some of the ideas on which the algorithm is based.

Suppose we execute the original k-means algorithm on the complete dataset At i^{th} iteration, the k centers are denoted by $c_1^i, c_2^i, \ldots, c_k^i$, respectively. In the new algorithm, FEKM, initially the k-means algorithm is executed on the sampled dataset with the same initialization. At i^{th} iteration, let the k centers be denoted as $s_1^i, s_2^i, \ldots, s_k^i$, respectively. For convenience, the first k centers are called as the *k-means centers*, and the later k centers are called as the *sampling centers*.

Further, for each sampling center s_j^i, FEKM associates a *confidence radius*, δ_j^i with it. The confidence radius δ_j^i is based upon an estimate of the upper-bound of the distance between the sampling center s_j^i and the corresponding k-means center c_j^i. Ideally, the confidence radius δ_j^i should be small, but should still satisfy the condition $d(c_j^i, s_j^i) \leq \delta_j^i$, where d is the distance function.

Now, consider the scan of the complete dataset taken by FEKM. As we discussed before, the sampling centers are stored in a table with k columns, where the ith row represents the ith iteration. To facilitate our discussion, we call the closest center among a set of k centers for a point as the *owner* of this point.

Definition 1 *For any point p in the complete dataset, assuming s_j^i to be the owner of point p with respect to the sampling centers at the i-th iteration, if there exists $l, l \neq j$, such that*

$$0 \leq d(s_l^i, p) - d(s_j^i, p) \leq \delta_j^i + \delta_l^i$$

then, p is a **boundary point** *for the i-th iteration.*

The complete set of boundary points is the union of boundary points for all iterations. Thus, the complete set of boundary points includes the points in the entire dataset whose owners with respect to the k-means centers are quite likely to be different from the owners with respect to the sampling centers, for one or more of the iterations.

For a given iteration i, the **stable points** are the points in the complete dataset that are not boundary points for the ith iteration. Usually, for any stable point, the difference between its distance to its owner with respect to the sampling centers and its distance to other sampling centers is quite large. Mathematically, assuming s_j^i to be owner of the point p with respect to the sampling centers at the i-th iteration, for any $l, l \neq j$, we have

$$d(s_l^i, p) - d(s_j^i, p) > \delta_j^i + \delta_l^i$$

3.3 Detailed Description

The detailed algorithm is shown in Figure 3.2. We now explain the algorithm.

The main data structure in FEKM is the table containing the summary of k-means run on sample data. We call this table as the *cluster abstract table* or the *CAtable*. Our algorithm starts with building a CAtable from a sample of the original dataset. Initially, each entry of the CAtable contains the two tuple, the center and the confidence radius of each cluster in that iteration. This is done through the function

BuildCATable. After this, we take one scan over the complete dataset and find out the likely *boundary points* for each iteration or for each row of the table. The function *IsBndrPoint* checks for each data point if it meets the conditions of being a boundary point.

If one point becomes a boundary point for one particular row, it is possible that the same point also be a boundary point for the next rows or next iterations of the CAtable. We define two lists, one to store the points and another to store the indexes of the rows where these points are found as boundary point. The first list is named as *Buffer* and the second list is named as *Index*. The second list is two dimensional where each row signifies one specific point and each column has ν bits, where ν is the number of iterations or rows in CAtable. If the specific point is found as a boundary point in the j-th row of the CAtable, then the j-th bit of the corresponding column of the *Index* list is set to 1. We also store the number and sum of the non-boundary points with each CAtable entry. The function *UpdateSufficientStats* accomplishes this.

Next, we recompute centers for each row of the CAtable from the boundary points corresponding to that row and from the sufficient statistics. In the Figure 3.2, it has been done by the function *RecomputeCtrs*. We then verify if the new centers are located within the pre-estimated confidence radius to maintain the correctness. The function *IsCtrsWithinRadii* is responsible for this verification. If we find that the new centers are located within the confidence radius of corresponding clusters, we update the centers of the CAtable in the next row using the function *UpdateCAtableCtrs*. If any of the new centers is found outside the confidence radius of the corresponding cluster, the initial centers are replaced by those new centers and the algorithm repeats from the creation of CAtable.

Input: D_i (Data Points), S_i (Sample Data Points), $InitCtrs$ (Initial Centers), ϵ (Stopping criteria in kmeans algorithm)
Output: k Cluster Centers
begin
  $flag \longleftarrow 1$;
  while *flag* **do**
    $List\ Buffer \longleftarrow NULL$;
    $Index[] \longleftarrow NULL$;
    $Table\ CAtable \longleftarrow NULL$;
    $NumRow \longleftarrow$ `BuildCATable`($InitCtrs$, S_i, $CAtable$, ϵ) ;
    for *each* D_i **do**
      for *each* $row_j \in NumRow$ **do**
        if ($ClosestCenter \longleftarrow$ `IsBndrPoint`(D_i)) **then**
          `BufferInsert`(D_i) ;
          $Index[BndCnt][j] \longleftarrow 1$;
          $BndCnt++$;
        else
          `UpdateSufficientStats`($ClosestCtr$, $CAtable$, D_i, row_j) ;
        end
      end
    end
    for *each* $row_j \in NumRow$ **do**
      $NewCtrs \longleftarrow$ `RecomputeCtrs`($CAtable$, $Buffer$, $Index$, row_j) ;
      if (`IsCtrsWithinRadii`($NewCtrs$, $CAtable$, row_{j+1}) **then**
        `UpdateCATableCtrs`($NewCtrs$, $CAtable$, row_{j+1}) ;
        $flag \longleftarrow 0$;
      else
        $InitCtrs \longleftarrow NewCtrs$;
        $flag \longleftarrow 1$;
      end
    end
  end
  `OutputCATableCtrs`($CAtable$, row_{NumRow}) ;
end

Algorithm 1: Pseudo Code of Fast and Exact Out of Core KMeans (FEKM).

3.4. Computation of Confidence Radius

The computation of confidence radius for each cluster and each iteration is an important aspect of the algorithm. Large radius values are likely to result in a large number of boundary points, which cannot be stored in memory. At the same time, very small confidence radius values could mean that the difference between corresponding sampling centers and k-means centers could be greater than this value, and therefore, an additional pass on the entire dataset may be required.

In our implementation, we use the following method for computing the confidence radius. Recall that at the iteration i, the confidence radius for the cluster j is denoted by δ_j^i. We use

$$\delta_j^i = f \times \left(\frac{\sum_{p=1}^{N} (X_p - X_c)}{N} \right)^{\frac{1}{2}}$$

where, X_p denotes a d dimensional point assigned to the cluster j at the iteration i, X_c is the center of the cluster j at iteration i, N is the number of points assigned to the cluster j, and f is a factor that is chosen experimentally. For a fixed f, the above expression will choose confidence radius value that

is proportional to the average distance of a point in the cluster to the cluster center. This ensures small confidence radius values for a dense cluster, and larger radius values otherwise.

Clearly, a problem that can arise is that number of boundary points can be huge and in the worst case, can also exceed memory size. Thus, in our implementation, we have two conditions on the number of boundary points. First, they should not exceed 20% of all points in the complete dataset. Second, they should not exceed the available memory. If during an execution, the number of boundary points violate either of the above two conditions, we reduce all the confidence radii by choosing a lower value of f, and repeat the computation of boundary points. For our experiments, the value of f was always fixed at 0.05.

4 Theoretical Analysis

In this section, we initially present a proof of correctness for the FEKM algorithm. Then, we also analyze the execution time of this algorithm.

4.1 Proof of Correctness

We now show how FEKM computes the same cluster centers as k-means. Our description here builds on the definitions of k-means centers, sampling centers, owners, boundary points, and stable points given in the previous section. We further add the definition of changing points.

Definition 2 *For the ith iteration, the* **changing points** *are defined as the points in the dataset that have different owners with respect to the sampling centers and the k-means centers.*

Lemma 1 *Suppose at i^{th} iteration, the following condition holds for each center j, $1 \leq j \leq k$,*

$$d(s_j^i, c_j^i) \leq \delta_j^i$$

Then, the stable points will have the same owners with respect to the sampling centers and the k-means centers, and the set of changing point is a subset of the set of boundary points.

Proof: Consider any point p in the complete dataset, and let s_j^i be the owner of p with respect to the sampling centers at the ith iteration. For any $l, l \neq j$, from the *triangle inequality*, we have

$$d(s_j^i, p) - d(s_j^i, c_j^i) \leq d(c_j^i, p) \leq d(s_j^i, p) + d(s_j^i, c_j^i)$$

$$d(s_l^i, p) - d(s_l^i, c_l^i) \leq d(c_l^i, p) \leq d(s_l^i, p) + d(s_l^i, c_l^i)$$

Further, applying the condition that is assumed, we can have the following inequalities

$$d(s_j^i, p) - \delta_j^i \leq d(c_j^i, p) \leq d(s_j^i, p) + \delta_j^i$$

$$d(s_l^i, p) - \delta_l^i \leq d(c_l^i, p) \leq d(s_l^i, p) + \delta_l^i$$

Therefore,

$$d(s_l^i, p) - d(s_j^i, p) - \delta_l^i - \delta_j^i \leq d(c_l^i, p) - d(c_j^i, p)$$

Case 1: If p is a stable point,

$$d(s_l^i, p) - d(s_j^i, p) > \delta_l^i + \delta_j^i$$

Therefore, we have the inequality

$$0 < d(c_l^i, p) - d(c_j^i, p)$$

This suggests that the center j is still the owner of the point p.

Case 2: If the point p changes its owner in the complete dataset, there exists a center l, such that

$$d(c_l^i, p) - d(c_j^i, p) < 0$$

Therefore, we have

$$d(s_l^i, p) - d(s_j^i, p) \leq \delta_l^i + \delta_j^i$$

This suggests that the point p is a boundary point. Combining both cases, we prove the lemma.

Lemma 2 *If* FEKM *has computed the k-means correctly at the ith iteration, and at the ith iteration, the condition*

$$d(s_j^i, c_j^i) \ \leq \ \delta_j^i \ \ \forall j, 1 \leq j \leq k$$

holds, then FEKM *will compute the k-means centers for the iteration $i+1$ correctly.*

Proof: This follows the result of Lemma 1. The stable points will have the same owners with respect to sampling centers and the k-means centers at the ith iteration. Therefore, in the ith row in the $CAtable$, we maintain the correct and sufficient statistics to summarize the stable points. Further, after the ith iteration, each boundary point can be assigned to the correct owner since we have the correct k-means centers for the ith iteration. Therefore, each center in the ith iteration owns the correct partition of the complete dataset, and the k-means centers of the iteration $i+1$ can be computed correctly.

Theorem 1 *Suppose that for each iteration $i, 0 \leq i \leq m$, the condition*

$$d(s_j^i, c_j^i) \ \leq \ \delta_j^i \ \ \forall j, 1 \leq j \leq k$$

holds, and at the iteration $m+1$, this condition does not hold. Then, for each iteration $i, 0 \leq i \leq m+1$, the k-means centers of the ith iteration can be computed correctly by FEKM.

Proof: This can be proved inductively. For the base case, we use the fact that at iteration 0, the same initialization centers are used by k-means and FEKM. For the induction step, we use the Lemma 2.

Theorem 2 *Assuming the same termination condition,* FEKM *will iterate the same number of times for the centers as the k-means algorithm, and at each iteration, will generate the same centers as the k-means algorithm.*

Proof: Recall that once the FEKM algorithm finds that the distance between sampling centers and k-means centers is greater than the confidence radius, it will sample again and take the k-means centers at that iteration as the initialization centers. Using this, and the Theorem 1, we have the above result.

4.2 Analysis of Performance

We now analyze the execution time for our algorithm, and compare it with that of original k-means.

Let the number of iterations that k-means takes on entire dataset be n. Let the I/O cost for reading the dataset once be C_I, and let the computing cost (besides the I/O cost) associated with each pass on the complete dataset be C_c. Therefore, the total running time of k-means algorithm $T_{k-means}$ can be expressed as

$$T_{KM} = n \times (C_I + C_c)$$

Now, let us consider our algorithm. Let P denote the number of times we need to sample the dataset. Also, let the size of each sample be a fraction SS of the entire dataset. Further, let the execution of k-means on the sampled dataset require an average of m iterations. This suggests that the total number of rows that are maintained in FEKM is $m \times P$. Therefore, the total running time of FEKM algorithm T_{KEKM} can be expressed as

$$\begin{aligned} T_{FEKM} &= P \times (SS \times (C_I + C_c) \ + \ C_I + m \times C_c) \\ &= P \times (SS + 1) \times C_I + P \times (SS + m) \times C_c \end{aligned}$$

From the expression above, we can see that in most cases, FEKM has higher computing cost than the k-means algorithm, since usually, FEKM has to compute more rows ($P \times m$) than the number for k-means (n). For execution on disk-resident datasets, the computing cost of k-means is typically much smaller than the I/O cost. Also, if we have the ability to overlap computation and I/O, the overall execution time reduces to the maximum of the I/O and computational costs, which is likely to be the I/O cost. In either case, we can see that if P is small, FEKM will be much faster than the k-means algorithm.

Our experiments have shown that P is 1 in most cases, and at most 2 or 3. Furthermore, a sampling fraction of 5% or 10% is usually sufficient. For such cases, the above expressions suggest a clear advantage for the FEKM algorithm. The next section further demonstrates this through experimental results.

5. Experimental Results from FEKM

This section reports on a number of experiments we conducted to evaluate the FEKM algorithm. Our experiments were conducted using a number of synthetic and real datasets. Our main goal was to compare the execution time of our algorithm with that of the k-means algorithm. Additionally, we were interested in seeing how many passes over the entire dataset were required by the FEKM algorithm. All our experiments were conducted on 700 MHz Pentium machines with 1 GB memory.

cxdy	dataset with x clusters and y dimensions
I_{KM}	No. of iterations in k-means.
Init "g"	good initialization
Init "b"	bad initialization
T_{KM}	Running Time of k-means (Sec.)
T_{FEKM}	Running Time of FEKM (Sec.)
SS	Sample Size (%)
P	Number of Passes by FEKM
se	Squared Error between final centers and the centers after sampling

Table 1. Explanation of the notations used in the result tables.

In all the experiments, the initial center points and the stopping criteria for this algorithm are kept same as those of the k-means algorithm. As the performance of k-means is very sensitive to the initialization, we considered different initializations. We used two different initialization techniques. In the first technique, which could only be applied for the synthetic datasets, we perturbed each dimension in the original center points of the Gaussians which were used to generate the data sets. Two different initializations, referred to as *good* and *bad*, were obtained by varying the range of perturbation. In the second technique, we randomly selected the initial center points from a sample of the dataset, such that distance between any two points chosen is greater than a *threshold*. In this case, the *good* and the *bad* initializations corresponded to a *large* and *small* value of this threshold, respectively.

Data	I_{KM}	Init	T_{KM}	T_{FEKM}	SS	P
c5d200	3	g	1452.83	644.66	10	1
c5d200	3	g	1452.83	571.22	5	1
c5d100	6	b	2688.65	902.81	10	1
c5d100	6	b	2688.65	762.47	5	1
c5d50	8	b	3602.31	1114.62	10	2
c5d50	8	b	3602.31	987.43	5	2
c5d20	8	b	3313.84	1098.41	10	1
c5d20	8	b	3313.84	940.47	5	1
c5d20	2	g	829.29	507.94	10	1
c5d20	2	g	829.29	412.53	5	1
c5d10	8	g	3833.44	1633.39	10	1
c5d10	8	g	3833.44	1302.52	5	1
c5d5	6	b	3116.89	1387.13	10	1
c5d5	6	b	3116.89	1236.51	5	1

Table 2. Performance of k-means and FEKM **Algorithms on Synthetic Datasets, 5 Clusters**

Data	I_{KM}	Init	T_{KM}	T_{FEKM}	SS	P
c10d200	10	b	4808.48	1559.70	10	2
c10d200	10	b	4808.48	1324.38	5	2
c10d200	2	g	954.33	577.22	10	1
c10d200	2	g	954.33	459.52	5	1
c10d100	3	g	1081.45	435.25	10	1
c10d100	3	g	1081.45	386.49	5	1
c10d50	100	b	49144.98	11267.28	10	2
c10d50	3	g	1462.43	725.22	10	1
c10d50	3	g	1462.43	649.60	5	1
c10d20	10	b	4570.63	1708.57	10	2
c10d20	10	b	4570.63	1408.57	5	2
c10d10	3	g	1623.10	867.50	10	1
c10d10	3	g	1623.10	773.64	5	1
c10d5	100	b	60310.89	26491.76	10	2
c10d5	100	b	60310.89	19349.28	5	2

Table 3. Performance of k-means and FEKM Algorithms on Synthetic Datasets, 10 Clusters

Data	I_{KM}	Init	T_{KM}	T_{FEKM}	SS	P
c20d200	100	b	54862.33	27388.85	10	2
c20d200	3	g	1898.65	746.51	10	1
c20d200	3	g	1898.65	584.88	5	1
c20d100	100	b	41029.15	18106.51	10	2
c20d100	3	g	1233.12	646.75	10	1
c20d100	3	g	1233.12	585.63	5	1
c20d50	3	g	1796.30	938.90	10	1
c20d50	3	g	1796.30	882.36	5	1
c20d20	10	b	5335.15	2528.11	10	2
c20d20	10	b	5335.15	2112.42	5	2
c20d10	6	g	3919.08	1814.73	10	1
c20d10	6	g	3919.08	1643.75	5	1
c20d5	6	b	4619.95	2899.76	10	1
c20d5	6	b	4619.95	2353.41	5	1

Table 4. Performance of k-means and FEKM Algorithms on Synthetic Datasets, 20 Clusters

We used two convergence criteria. The algorithm stops when (1) the new centers are not sufficiently different from those generated in the previous iteration, or (2) it has run for a specified maximum number of iterations. The second criteria is useful with bad initializations, where the algorithm could run for a large number of iterations. The notation used in the tables containing the results of the experiments are explained in the Table 1.

5.1 Evaluation with Synthetic Datasets

The evaluation with synthetic datasets was done using 18 1.1 GB datasets and 2 4.4 GB datasets. The datasets involve different number of clusters and dimensions. For generating each synthetic dataset, points are drawn from a mixture of fixed number of Gaussian distributions. Each Gaussian is assigned a random weight which determines the size of each cluster. For each dimension, we kept the mean and variance of each Gaussian in the interval $[-5, 5]$ and $[0.7, 1.5]$, respectively, to retain the flavor of the datasets used in the experiments by Bradley *et al.* [4] and in the experiments by Farnstorm *et al.* [11]. We did the experiments using 5, 10 and 20 clusters and with 5, 10, 20, 50, 100, and 200 dimensions.

Data	I_{KM}	Init	T_{KM}	T_{FEKM}	SS	P
c20d100	2	g	4393.02	2931.53	10	1
c20d100	2	g	4393.02	2204.42	5	1
c20d100	10	b	21985.62	8194.07	10	1
c20d100	10	b	21985.62	7467.53	5	1
c5d20	10	b	43254.34	10341.42	10	2
c5d20	10	b	43254.34	9632.71	5	2

Table 5. Performance of k-means and FEKM Algorithms with 4.4 GB Synthetic Datasets

We first consider the results from the 1.1 GB datasets. Tables 2, 3, and 4 show the execution times for FEKM and original k-means with 5, 10, and 20 clusters, respectively. As these tables show, FEKM requires one or at most two passes on the entire dataset. FEKM is faster by a factor between 2 and 4 in almost all cases. The relative speedup of FEKM is higher with bad initializations, where a larger number of iterations are required. We have considered sample sizes that are 5% and 10% of the entire dataset. FEKM is always faster with 5% sample size, because it reduces the execution time for the k-means, and did not require any additional passes on the entire dataset. The number of clusters or dimensions do not make a significant difference to the relative performance of the two algorithms.

Next, we consider the results from the 2 4.4 GB that we generated. Table 5 shows these results. The first set has 20 clusters and 100 dimensions. The second dataset has 5 clusters and 20 dimensions. The relative speedup of FEKM is between 2 and 4.5.

5.2. Evaluation with Real Data Set

We evaluated our algorithm with three publicly available real datasets. These datasets are KDDCup99, Corel image database, and the Reuters-21578. All these datasets are available from University of Irvine's KDD archive[1]. We preprocessed each of these datasets and generated feature vectors using standard techniques, briefly described below. We then applied k-means and the FEKM algorithm. To be able to experiment with out-of-core datasets, we increased the size of the datasets by random sampling.

It should be noticed that the running time of both the algo-

[1] http://kdd.ics.uci.edu

Data	I_{KM}	Init	T_{KM}	T_{FEKM}	SS	P	se
kdd99	19	g	7151	2317	10	2	4.0
kdd99	19	g	7151	2529	15	2	3.5
kdd99	19	g	7151	2136	5	2	4.2
Corel	43	g	28442	10503	10	3	2.2
Corel	43	g	28442	12603	15	3	2.15
Corel	43	g	28442	9342	5	3	3.24
Reuter	20	b	41290	10311	10	2	10.1
Reuter	20	b	41290	11204	15	2	8.6
Reuter	20	b	41290	9214	5	2	14.9

Table 6. Performance of k-means and FEKM **Algorithms, Real Datasets**

rithms can vary depending on particular preprocessing of the real dataset. In our experiments, we used simple preprocessing techniques which can be improved upon. We used Euclidean distance function to compute distance between two points. Different distance metrics may help in obtaining better quality clusters and it can also reduce number of iterations, particularly for the datasets with categorical attributes.

The KDD Cup 99 data consists of feature vectors generated from network connection. This dataset is used for evaluating network intrusion detection techniques. The size of this dataset is about 743 MB. We enumerated different symbols of each type of categorical attributes. Each attribute is normalized by dividing with the maximum value of that attribute. After the preprocessing step, we obtained normalized continuous-valued feature vectors of 38 dimensions. The number of clusters specified in our experiments was 5. By supersampling the data, we created 5 million feature vectors for a resulting dataset size of 1.8 GB.

Corel image database has 68,040 images of different categories. We used the 32 dimensions color histogram feature vectors of these images which is available from UCI KDD archive. This dataset is about 20 MB. We thus increased the size of the dataset by randomly selecting vectors from the dataset and created a 1.9 GB dataset containing 6,804,000 continuous-valued feature vectors. We kept the number of clusters at 16. All attribute values are normalized between zero to one.

Reuters text database is extensively used for text categorization. We created integer-valued feature vectors of 258 dimensions by counting the frequency of most frequent words in 135 different categories. Following Fayyad, Bradley and Reina [4], we kept the number of clusters at 25 for our experiments. We then increased the size of the dataset by supersampling and created 4.3 million records of 258 dimensions. The size of the resulting dataset was approximately 2 GB.

Table 6 presents the experimental results from these three real datasets. Similar to what we observed from synthetic datasets, the speedup from FEKM is between 2 and 4.5. The number of passes required by FEKM is either 2 or 3.

Clearly, FEKM produces the same cluster centers as the ones from the original k-means. One question is, how do they compare with the results from clustering the sampled data. We report some data in this regard from the real datasets. The column labeled *se* shows the squared difference between the final centers and the centers obtained from running k-means on sampled data. The values of all attributes were normalized to be between 0 and 1 for the real datasets. In view of this, we consider the reported squared errors to be significant. Further, it should be noted that all datasets were super-sampled, which favors sampling. Thus, we believe that using an accurate algorithm like FEKM is required for getting high-quality cluster centers.

6 Distributed Version of FEKM

This section describes the distributed version of FEKM, which is refered to as DFEKM.

6.1 Motivation

With the emergence of the internet, web, and now grid computing, data is increasingly being shared through data repositories. Often, data of interest to a data analyzer is distributed across multiple data repositories. Analysis of large and geographically distributed scientific datasets has emerged as an important problem [7]. The challenges of developing mining algorithms for distributed datasets are well recognized [20]. The biggest issue is that it is typically not feasible to down-load the entire dataset on a single machine and apply standard algorithms. Because the sites hosting the data are only loosely coupled, the communication latencies are very high and parallel algorithms cannot be directly applied either.

Now, let us consider the problem of clustering distributed datasets. If there are no privacy considerations, one can download the data to a local machine and then use any of the standard algorithms for data clustering. Downloading large volumes of data and then applying a serial clustering algorithm can be very time-consuming. Moreover, this approach also requires significant communication, storage, and computing resources which may simply not be available.

Another potential approach could be to try and execute a parallel clustering algorithm across the distributed data repositories. k-means clustering algorithm has been parallelized by many [9, 12], and has been shown to give high parallel efficiency on tightly coupled parallel machines. However, these implementations require uniform data distribution across the nodes of the parallel machine, and one round of communication after every iteration of the clustering algorithm. If these algorithms are executed on distributed data repositories, uneven data distribution and high communication latencies will likely result in poor performance.

In this section, we describe and evaluate a distributed version of our FEKM algorithm, which is refered to as DFEKM. This algorithm is suitable for analyzing data that is distributed across loosely coupled machines. Unlike the

previous work in this area [18, 19, 25, 27], our algorithm provably produces the same results as the original k-means algorithm.

We are not aware of any existing distributed clustering algorithm which provably produces the same clusters as the original k-means algorithm, or which maintains k-means property of achieving a local minimum. Samatova, Ostrouchov and their group [27] has proposed a technique called "Recursive Agglomeration of Clustering Hierarchies by Encircling Tactic" (RACHET). This technique is based on sufficient statistics. It collects local *dendograms* and then merges them to create a global dendogram. However, this needs to iterate until the sufficient statistics converges to the desired quality. Parthasarathy and Ogihara [25] provided an algorithm where the distance metric is formed applying association rules locally. Kargupta and his group [19] have applied PCA to do high dimensional clustering in a distributed fashion. Januzaj *et al.* have recently proposed a distributed clustering technique that involves creating local clusters, and then deriving global clusters from them [18]. There have been many efforts on parallelizing k-means, or other related clustering algorithms such as k-harmonic and EM. Key efforts in this area include Dillon and Modha [9] (for k-means), Kruengkrai *et al.* [21] and Lopez *et al.* [8] (both for EM), and Forman and Zhang (for k-means, k-harmonic, and EM). All of these approaches require data to be evenly distributed between the nodes, and one round of communication after every pass on the data. Therefore, these approaches are not suitable for clustering data resident on distributed repositories.

6.2 Algorithm Description

Our description in this section assumes that data to be clustered is available at two or more nodes, which are referred to as the *data sources*. In addition, we have a node denoted as the *central site*, where the results of clustering are desired. It is also assumed that additional computation for clustering can be performed at the central site. We only consider horizontal partitioning of the data, i.e., each data source has values along all dimensions of a subset of the points.

Distributed version of FEKM proceeds as follows. We sample data from each data source, and communicate it to the central node. Then, on the central node, we run the k-means algorithm on this sampled data. The main data structure of FEKM, the CATable, is computed and stored. Then, we send the table to all the data sources. Next, at each data source, we take one pass through the portion of the dataset available at that data source. For a given point and row of the table, we determine if this point is a boundary point. If it is, it is stored in a buffer. Otherwise, we update the *sufficient statistics tuple*, which has the number and sum of the data points for the cluster.

After the pass through the dataset and storing the boundary point, all the nodes send their boundary points and sufficient statistics to central node. The central node then does the following processing. Starting from the first row of the table, it recomputes centers using the boundary points and sufficient statistics tuple. If any of the new computed centers fall outside the pre-estimated confidence radius, which means that our computation of boundary points is not valid, we need to send the last corrected centers to all other nodes. Using these centers as the new initialization points, we have to go through another iteration and repeat all the steps. However, if the new computed centers are within the confidence radius, we use these centers for the next iteration and continue. Finally, the algorithm terminates by checking for the same termination condition that one would use in the original algorithm.

6.3 Experimental Results

We now report on a number of experiments we conducted to evaluate the DFEKM algorithm. We initially discuss the experiments we designed. As we stated earlier, two existing approaches for applying k-means like clustering algorithm on distributed datasets are: 1) down-loading data on a single node and applying the centralized k-means algorithm, and 2) executing parallel k-means algorithm across distributed data sources. Our experiments compare our DFEKM algorithm to these two approaches. As the main property of our algorithm is that it produces the same results as the exact k-means algorithm applied centrally, we did not compare DFEKM against any of the existing approximate distributed clustering approaches.

Another challenge in designing our experiments was to simulate execution on distributed data repositories. As compared to a tightly coupled parallel configuration, executing parallel code on distributed data repositories potentially involves large load imbalance and/or high communication latencies. Our experiments were conducted on a parallel machine, and the above two effects were simulated by introducing delays during each round of communication. We also considered cases in which data has not evenly distributed. For such cases, we compare parallel k-means, sequential k-means, and our DFEKM algorithm. Additionally, we were interested in seeing how many passes over the entire dataset and how many rounds of communication were required by the DFEKM algorithm.

All our experiments were conducted on IA-32 cluster at Ohio Super-computing Center (OSC). Each node of IA-32 cluster has two 900 MHz Itanium-2 processors and 4GB main memory. These nodes are connected using Myrinet, which is a switched 2.0GB/s network.

The evaluation with synthetic datasets was done using four 1.1 GB datasets. The method used for generating these was described in the previous section. As in the previous section, a synthetic dataset with n clusters and m dimensions is referred to as cndm.

The four datasets used in our experiments had 5 and 20 clusters, and 10 and 100 dimensions. We considered parallel/distributed configurations with 1, 2, and 4 nodes, and with 0, 50, 100, 150, and 200 seconds *communication delays*. This communication delay is a waiting time before any

Dataset	No. of Points (Millions)	Size (GB)	No. of Iterations (k-means)	No. of Passes (DFEKM)
c5d100	2	1.1	20	2
c5d10	20	1.1	20	2
c20d100	2	1.1	20	2
c20d10	20	1.1	20	2
kdd	1.5	1.8	18	2
Corel	1.5	1.9	20	2

Table 7. Statistics from Parallel k-means and DFEKM Executions on Synthetic and Real data sets.

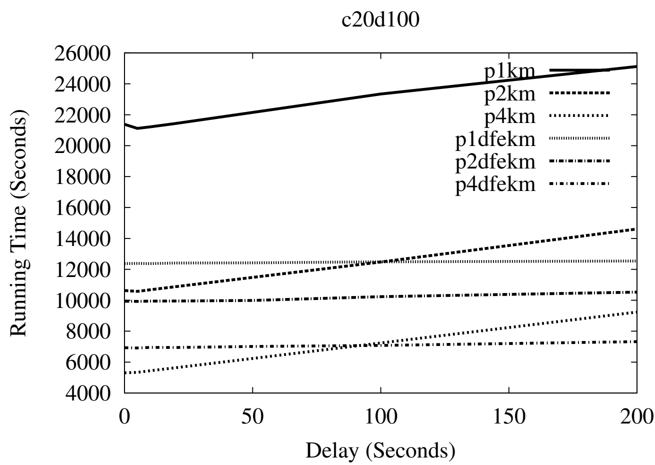

Figure 1. Running Time of DFEKM and Parallel k-means with Increasing Delays: 1,2 and 4 nodes, dataset `c20d100`

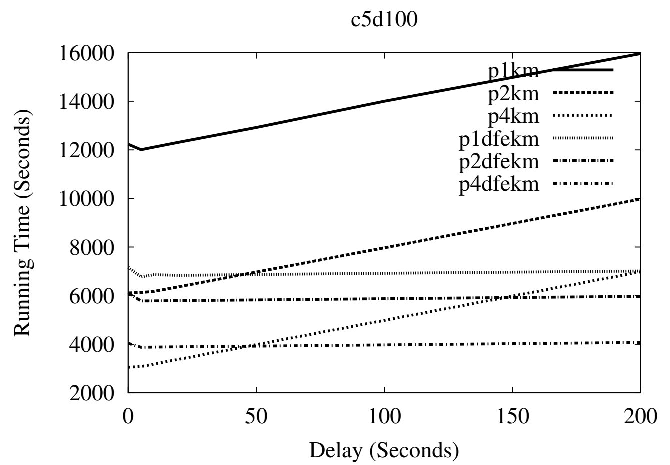

Figure 3. Running Time of DFEKM and Parallel k-means with Increasing Delays: 1,2 and 4 nodes, dataset `c5d100`

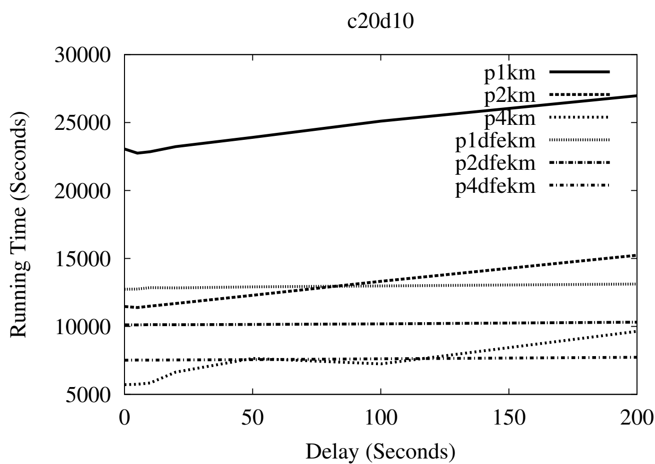

Figure 2. Running Time of DFEKM and Parallel k-means with Increasing Delays: 1,2 and 4 nodes, dataset `c20d10`

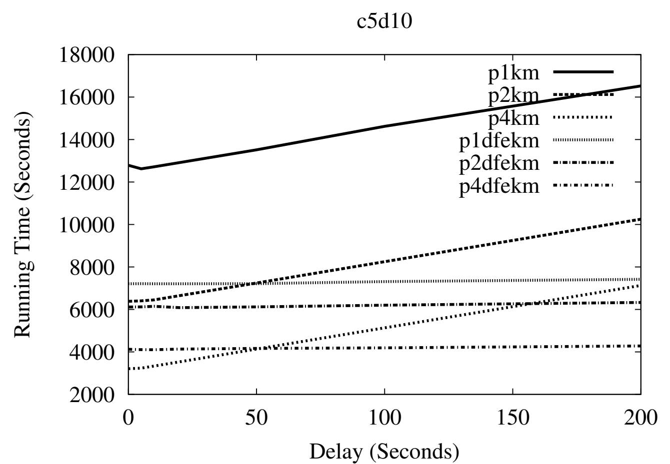

Figure 4. Running Time of DFEKM and Parallel k-means with Increasing Delays: 1,2 and 4 nodes, dataset `c5d10`

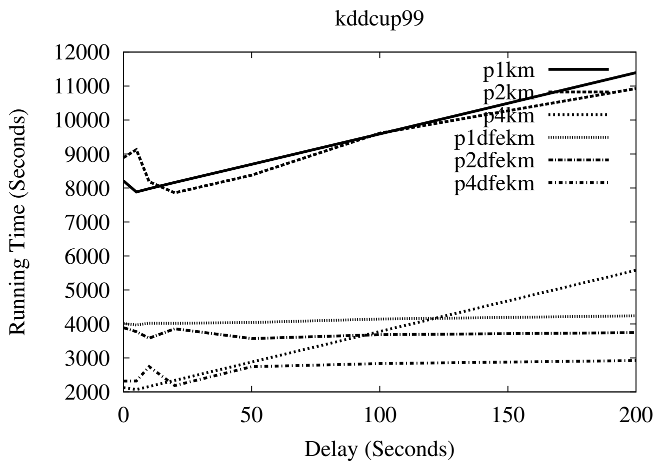

Figure 5. Running Time of DFEKM and Parallel k-means with Increasing Delays: 1,2 and 4 nodes, `kddcup99` dataset

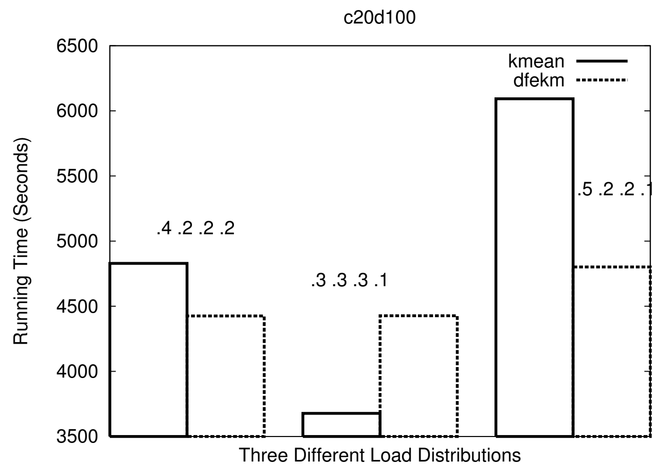

Figure 7. Running Time of DFEKM and Parallel k-means in the Presence of Load Imbalance: `c20d100` dataset

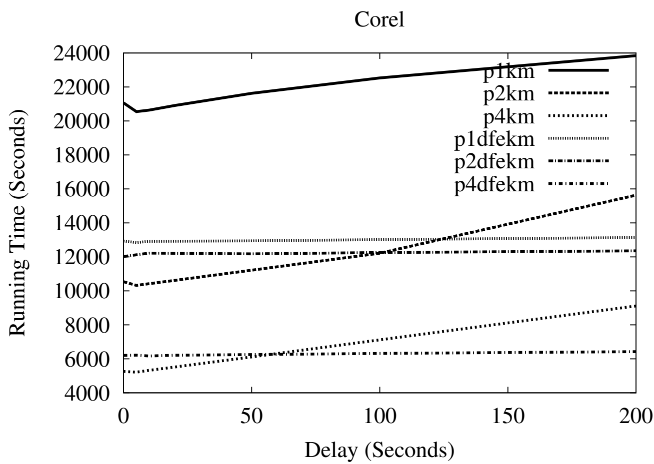

Figure 6. Running Time of DFEKM and Parallel k-means with Increasing Delays: 1,2 and 4 nodes, `Corel` dataset

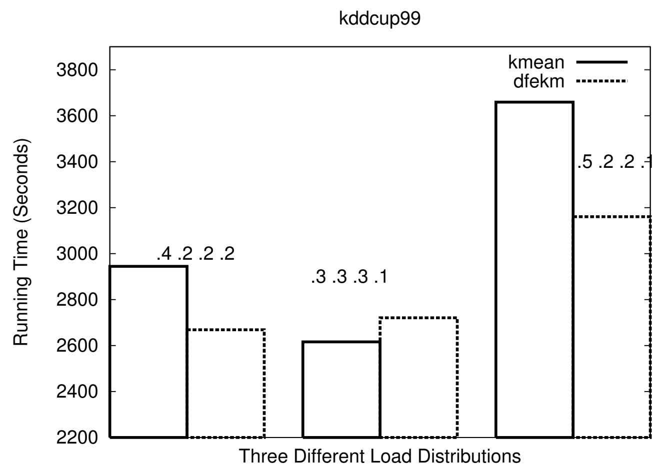

Figure 8. Running Time of DFEKM and Parallel k-means in the Presence of Load Imbalance: `kddcup99` dataset

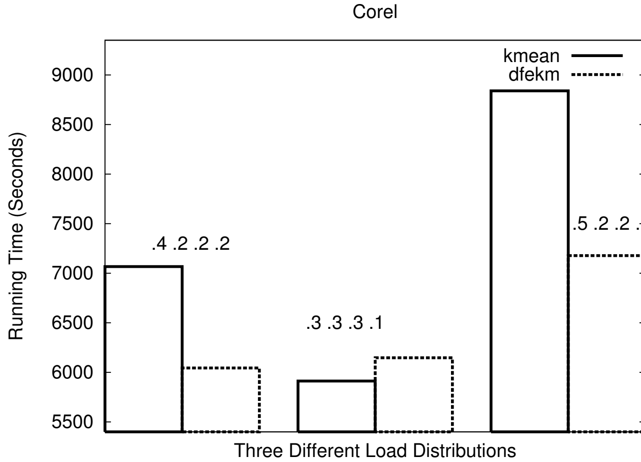

Figure 9. Running Time of DFEKM and Parallel k-means in the Presence of Load Imbalance: `Corel` dataset

round of interprocessor communication. A delay of 0 seconds corresponds to a tightly coupled parallel configuration in which data is evenly distributed between the nodes. Non-zero delays simulate high communication latencies in loosely coupled environments, as well as waiting time for the slowest node, when the data is not evenly distributed.

Table 7 compares the number of passes or iterations required by (parallel) k-means and DFEKM. Parallel k-means on each of the four synthetic datasets required 20 iterations, which also means 20 rounds of interprocessor communication. DFEKM required two passes in each case, which also corresponds to 3 rounds of interprocessor communication. The execution times for these four synthetic datasets are presented in Figures 1, 2, 3, and 4, respectively.

Several things should be noted from these experiments. First, even on a single node, `DFEKM` does better than k-means. This is because this algorithm needs much fewer passes on the data. As expected, the relative speedup of parallel k-means is linear. In comparison, DFEKM sequentializes a part of the computation, and does not scale that well. However, because it requires much fewer rounds of communication, increasing delays only have a modest impact on its performance. As communication delays increase, DFEKM outperforms parallel k-means on 2 and 4 nodes.

As we stated earlier, one way of clustering data from distributed data repositories is to down-load all data at one node, and apply a centralized clustering algorithm. Our results in Figures 1, 2, 3, and 4, show that DFEKM on 2 and 4 nodes outperforms both k-means and DFEKM on 1 node. Thus, even without including the cost of down-loading and storing data, DFEKM has better performance than applying a centralized algorithm.

In Figure 7, we explicitly create load imbalance and compare the relative performance of parallel k-means and DFEKM. The dataset `c20d100` was used for this experiment. On a 4 nodes configuration, we considered three different distributions of data. In the first case, the fraction of data resident on each of the 4 nodes was 40%, 20%, 20%, and 20%, respectively. In the second case, the fractions were 30%, 30%, 30%, and 10%, respectively. In the third case, the fractions were 50%, 20%, 20%, and 10%, respectively. No further delays were introduced.

In both first and third cases, DFEKM performs better. In the second case, parallel k-means did better. This was because the performance of k-means depends upon the slowest of the nodes. The fraction of the data with the slowest node was 40%, 30%, and 50% for the three cases, respectively. Thus, we can see that if the load imbalance is significant, DKEKM can outperform parallel k-means even in tightly coupled configuration.

We also evaluated our algorithm with KDDCup99 and Corel image database, which we had described in the previous section. Initially, we looked at the number of passes required by k-means and DFEKM. As shown in Table 7, the number of iterations for k-means were 18 and 20, whereas the number of passes required by DFEKM was 2 for both the datasets. The execution times with increasing communication delays are shown in Figures 5 and 6. The results are quite similar to those obtained from the synthetic datasets. As communication delays increase, DFEKM performs better on 2 and 4 nodes. Also, the performance of DFEKM on 2 and 4 nodes is better than sequential execution with either k-means or DFEKM. Again, this shows that even without including the cost of down-loading and storing data, DFEKM has better performance than applying a centralized clustering algorithm.

We also repeated the experiment with load imbalance. The results are shown in Figures 8 and 9. When the slowest of the 4 nodes has 30% of the data, parallel k-means performs better. However, in the cases when the slowest node has 40% or 50% of the data, DFEKM is better.

7. Conclusions

We have presented, analyzed, and evaluated an algorithm that provably produces the same cluster centers as the k-means clustering algorithm, and typically requires one or a small number of passes on the entire dataset. This can significantly reduce the execution times for clustering on large or disk-resident datasets, with no compromise on the quality of the results. While a number of approaches existed for approximating k-means or similar algorithms with sampling or using a small number of passes, none of these approaches could provably produce the same cluster centers as the original k-means algorithm. The basic idea in our algorithm is to use sampling to create approximate cluster centers, and use these approximate cluster centers for speeding up the compu-

tation of correct or exact cluster centers. Our experimental evaluation on a number of synthetic and real datasets have shown a speedup between 2 and 4.5.

This paper has also described and evaluated a distributed version of FEKM. This algorithm is suitable for analyzing data that is distributed across loosely coupled machines. Unlike the previous work in this area, DFEKM provably produces the same results as the original k-means algorithm. Our experimental results show that DFEKM is clearly better than two other possible options for exact clustering on distributed data, which are down-loading all data and running sequential k-means, or running parallel k-means on a loosely coupled configuration. Moreover, even in a tightly coupled environment, DFEKM can outperform parallel k-means if there is a significant load imbalance.

References

[1] Mihai Badoiu, Sariel Har-Pelad, and Piotr Indyk. Approximate clustering via core-sets. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, 2002.

[2] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, 2002.

[3] Leon Bottou and Yoshua Bengio. Convergence properties of the K-means algorithms. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 585–592. The MIT Press, 1995.

[4] P. S. Bradley, Usama Fayyad, and Cory Reina. Scaling clustering algorithms to large databases. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, August 1998.

[5] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahi. Better streaming algorithms for clustering problems. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, 2003.

[6] Moses Cherikar, Chandra Chekuri, Tomas Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of Symposium of Theory of Computing*, 1997.

[7] A. Chervenak, I. Foster, C. Kesselman, C. Salisbusy, and S. Tuecke. The Data Grid: Towards An Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2001.

[8] P.E. Lopez de Teruel, , J. M. Garcia, and M. Acacio. A Parallel Algorithm and Its Application to Computer Vision. In *Proceedings of PDPTA*, 1999.

[9] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Lecture Notes In Computer Science, Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260. Springer-Verlag, 1999.

[10] Pedro Domingos and Geoff Hulten. A general method for scaling up machine learning algorithms and its application to clustering. In *Proceedings of the Eighteenth International Conference on Machine Learning*, 2001.

[11] Fredrik Farnstrom, James Lewis, and Charles Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explorations*, 2(1):51–57, 2000.

[12] G. Forman and B. Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explorations*, 2, 2000.

[13] Joydeep Ghosh. Scalable clustering methods for data mining. In Nong Ye, editor, *Handbook of Data Mining*, chapter 10, pages 247–277. Lawrence Ealbaum Assoc, 2003.

[14] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):515–528, May 2003.

[15] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[16] J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *Applied Statistics*, (28):100–108, 1979.

[17] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall International, 1988.

[18] E. Januzaj, H-P Kriegel, and M. Pfeifle. Towards effective and efficient distributed clustering. In *Proceedings of the ICDM 2003 Workshop on Clustering Large Datasets*, 2003.

[19] H. Kargupta, W. Huang, K. Sivakumar, and E. Johnson. Distributed clustering using collective principal component analysis. *Knowledge and Information Systems*, 3(4):422–448, 2001.

[20] Hillol Kargupta and Phillip Chan (Editors). *Advances in Distributed Data Mining*. AAI/MIT Press, 1999.

[21] C. Kruengkrai and C. Jaruskulchai. A parallel learning algorithm for text classification. In *Proceedings of ACM SIGKDD 2002*, pages 201–206. ACM Press, August 2002.

[22] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

[23] Silvia Nittel, Kelvin T. Leung, and Amy Braverman. Scaling clustering algorithms for massive data sets using data stream. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.

[24] Liadan OCallaghan, Nina Mishra, Adam Meyerson, Sudipto Guha, and Rajeev Motwani. Streaming-data algorithms for high-quality clustering. In *Proceedings of International Conference of Data Engineering*, 2002.

[25] S. Parthasarathy and M. Ogihara. Clustering Distributed Homogeneous Datasets. In *Proceedings of the Fourth European Conference on Principles of Data Mining and Knowledge Discovery*, volume 1910, pages 566–574. Springer-Verlag Lecture Notes in Computer Science, 2000.

[26] Dan Pelleg and Andrew Moore. Accelerating exact k-means algorithms with geometric reasoning. In *In Proceedings of Fifth International Conference of Knowledge Discovery and Data Mining*, pages 277–281, 1999.

[27] N. F. Samatova, G. Ostrouchov, A. Geist, and A. Melechko. RACHET: An Efficient Cover-Based Merging of Clustering Hierarchies from Distributed Datasets. *Distributed and Parallel Databases*, 11(2):157–180, 2002.