A Data-Clustering Algorithm On Distributed Memory Multiprocessors

Inderjit S. Dhillon and Dharmendra S. Modha
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099
emails: inderjit@cs.utexas.edu, dmodha@almaden.ibm.com

IBM Research Report RJ 10134 (95009), Nov. 11, 98
Proc. Large-scale Parallel KDD Systems Workshop, ACM SIGKDD, Aug. 15-18, 99

Abstract

To cluster increasingly massive data sets that are common today in data and text mining, we propose a parallel implementation of the k-means clustering algorithm based on the message passing model. The proposed algorithm exploits the inherent data-parallelism in the k-means algorithm. We analytically show that the speedup and the scaleup of our algorithm approach the optimal as the number of data points increases. We implemented our algorithm on an IBM POWERparallel SP2 with a maximum of 16 nodes. On typical test data sets, we observe nearly linear relative speedups, for example, 15.62 on 16 nodes, and essentially linear scaleup in the size of the data set and in the number of clusters desired. For a 2 gigabyte test data set, our implementation drives the 16 node SP2 at more than 1.8 gigaflops.

Keywords: k-means, data mining, massive data sets, message-passing, text mining.

1 Introduction

Data sets measuring in gigabytes and even terabytes are now quite common in data and text mining, where a few million data points are the norm. For example, the patent database (www.ibm.com/patents/), the Lexis-Nexis document collection containing more than 1.5 billion documents (www.lexisnexis.com), and the Internet archive (www.alexa.com) are in multi-terabyte range. When a sequential data mining algorithm cannot be further optimized or when even the fastest available serial machine cannot deliver results in a reasonable time, it is natural to look to parallel computing. Furthermore, given the monstrous sizes of the data sets, it often happens that they cannot be processed *in-core*, that is, in the main memory of a single processor machine. In such a situation, instead of implementing a disk based algorithm which is likely to be considerably slower, it is appealing to employ parallel computing and to exploit the main memory of *all* the processors.

Parallel data mining algorithms have been recently considered for tasks such as association rules and classification, see, for example, Agrawal and Shafer [1], Chattratichat et al. [6], Cheung and Xiao [8], Han, Karypis, and Kumar [22], Joshi, Karypis, and Kumar [24], Kargupta, Hamzaoglu, and Stafford [25], Shafer, Agrawal, and Mehta [32], Srivastava, et al. [38], Zaki, Ho, and Agrawal [41], and Zaki et al. [42]. Also, see Stolorz and Musick [39] and Freitas and Lavington [17] for recent books on scalable and parallel data mining.

In this paper, we consider parallel clustering. Clustering or *grouping of similar objects* [23] is one of the most widely used procedures in data mining [16]. Practical applications of clustering include unsupervised classification and taxonomy generation [23], nearest neighbor searching [18], scientific discovery [7, 35], vector quantization [20], time series analysis [33], and multidimensional visualization [12, 13].

Our interest in clustering stems from the need to mine and analyze heaps of unstructured text documents. Clustering has been used to discover "latent concepts" in sets of unstructured text documents, and to summarize and label such collections. Clustering is inherently useful in organizing and searching large text collections, for example, in automatically building an ontology like Yahoo! (www.yahoo.com). Furthermore, clustering is useful for compactly summarizing, disambiguating, and navigating the results retrieved by a search engine such as AltaVista (www.altavista.com). Conceptual structure generated by clustering is akin to the "Table-of-Contents" in *front* of books. Finally, clustering is useful for personalized information delivery by providing a setup for routing new information such as that arriving from newsfeeds and new scientific publications. For experiments describing a certain syntactic clustering of the whole web and its applications, see [5]. For detailed review of various classical text clustering algorithms such as the k-means algorithm and its variants, hierarchical agglomerative clustering, and graph-theoretic methods, see [29, 40]. Recently, there has been a flurry of activity in this area, see [3, 10, 30, 34, 43]. For our recent work on matrix approximations using a variant of the k-means algorithm applied to text data, see [11]. Our results have been extremely promising; their applicability to extremely large collections of text documents requires a highly scalable implementation, and, hence, the motivation for this work.

In this paper, as our main contribution, we propose a parallel clustering algorithm on distributed memory multiprocessors, that is, on a shared-nothing parallel machine, and analytically and empirically validate our parallelization strategy. Specifically, we propose a parallel version of the popular k-means clustering algorithm [15, 23] based on the message-passing model of parallel computing [21, 37]. To the best of our knowledge, a parallel implementation of the k-means clustering algorithm has not been reported in the literature. In this paper, our focus in on parallelizing the classical direct k-means algorithm. Recently, Alsabti, Ranka and Singh [2] have proposed a serial, tree-based k-means algorithm; parallelizing their implementation remains an open problem.

We now briefly outline the paper, and summarize our results. In Section 2, we present the k-means algorithm. In Section 3, we carefully analyze the computational complexity of the k-means algorithm. Based on this analysis, we observe that the k-means algorithm is inherently data-parallel. By exploiting this parallelism, we design a parallel k-means algorithm. We analytically show that the speedup and the scaleup of our algorithm approach the optimal as the number of data points increases. In other words, we show that as the number of data points increases the communication costs incurred by our parallelization strategy are relatively insignificant compared to the overall computational complexity. Our parallel algorithm is based on the message-passing model of parallel computing; this model is also briefly reviewed in Section 3. In Section 4, we empirically study the performance of our parallel k-means algorithm (that is, speedup and scaleup) on an IBM POWERparallel SP2 with a maximum of 16 nodes. We empirically establish that our parallel k-means algorithm has nearly linear speedup, for example, 15.62 on 16 nodes, and has nearly linear scaleup behavior. To capture the effectiveness of our algorithm in a nutshell, note that we are able to to drive the 16 node SP2 at nearly 1.8 gigaflops (floating point operations) on a 2 gigabyte test data set. In Section 5, we include a brief discussion on future work.

Our parallelization strategy is simple but very effective; in fact, the simplicity of our algorithm makes it ideal for rapid deployment in applications.

2 The k-means Algorithm

Suppose that we are given a set of n data points $X_1, X_2, \cdots, X_n$ such that each data point is in R^d. The problem of finding the *minimum variance* clustering of this data set into k clusters is that of finding k points $\{m_j\}_{j=1}^k$ in R^d such that

$$\frac{1}{n}\sum_{i=1}^{n}\left(\min_j d^2(X_i, m_j)\right), \tag{1}$$

is minimized, where $d(X_i, m_j)$ denotes the Euclidean distance between X_i and m_j. The points $\{m_j\}_{j=1}^k$ are known as *cluster centroids* or as *cluster means*. Informally, the problem in (1) is that of finding k cluster centroids such that the average squared Euclidean distance (also known as the mean squared error or MSE, for short) between a data point and its nearest cluster centroid is minimized. Unfortunately, this problem is known to be NP-complete [19].

The classical k-means algorithm [15, 23] provides an easy-to-implement approximate solution to (1). Reasons for popularity of k-means are ease of interpretation, simplicity of implementation, scalability, speed of convergence, adaptability to sparse data, and ease of out-of-core implementation [11, 31, 44]. We present this algorithm in Figure 1, and intuitively explain it below:

1. (**Initialization**) Select a set of k starting points $\{m_j\}_{j=1}^k$ in R^d (line 5 in Figure 1). The selection may be done in a random manner or according to some heuristic.

2. (**Distance Calculation**) For each data point X_i, $1 \leq i \leq n$, compute its Euclidean distance to each cluster centroid m_j, $1 \leq j \leq k$, and then find the closest cluster centroid (lines 14-21 in Figure 1).

3. (**Centroid Recalculation**) For each $1 \leq j \leq k$, recompute cluster centroid m_j as the average of data points assigned to it (lines 22-26 in Figure 1).

4. (**Convergence Condition**) Repeat steps 2 and 3, until convergence (line 28 in Figure 1).

The above algorithm can be thought of as a gradient-descent procedure which begins at the starting cluster centroids and iteratively updates these centroids to decrease the objective function in (1). Furthermore, it is known that k-means will always converge to a local minimum [4]. The particular local minimum found depends on the starting cluster centroids. As mentioned above, the problem of finding the global minimum is NP-complete.

Before the above algorithm converges, steps 2 and 3 are executed a number of times, say $\mathfrak{I}$. The positive integer $\mathfrak{I}$ is known as the *number of* k*-means iterations*. The precise value of $\mathfrak{I}$ can vary depending on the initial starting cluster centroids even on the same data set.

In Subsection 3.2, we analyze, in detail, the computational complexity of the above algorithm, and propose a parallel implementation.

3 Parallel k-means

Our parallel algorithm design is based on the Single Program Multiple Data (SPMD) model using message-passing which is currently the most prevalent model for computing on distributed memory multiprocessors; we now briefly review this model.

3.1 Message-Passing Model of Parallel Computing

We assume that we have P processors each with a local memory. We also assume that these processors are connected using a communication network. We do not assume a specific interconnection topology for the communication network, but only assume that it is generally cheaper for a processor to access its own local memory than to communicate with another processor. Such machines are commercially available from vendors such as Cray and IBM.

Potential parallelism represented by the distributed-memory multiprocessor architecture described above can be exploited in software using "message-passing." As explained by Gropp, Lusk, and Skjellum [21, p. 5]:

> The message-passing model posits a set of processes that have only local memory but are able to communicate with other processes by sending and receiving messages. It is a defining feature of the message-passing model that data transfers from the local memory of one process to the local memory of another process require operations to be performed by both processes.

MPI, the Message Passing Interface, is a standardized, portable, and widely available message-passing system designed by a group of researchers from academia and industry [21, 37]. MPI is robust, efficient, and simple-to-use from FORTRAN 77 and C/C++.

```
MSE = LargeNumber;
Select k initial cluster centroids {m_j}_{j=1}^k;
do {
    OldMSE = MSE;
    MSE' = 0;
    for j = 1 to k
        m'_j = 0; n'_j = 0;
    endfor;
    for i = 1 to n
        for j = 1 to k
            compute squared Euclidean
                distance d^2(X_i, m_j);
        endfor;
        find the closest centroid m_ℓ to X_i;
        m'_ℓ = m'_ℓ + X_i; n'_ℓ = n'_ℓ + 1;
        MSE' = MSE' + d^2(X_i, m_ℓ);
    endfor;
    for j = 1 to k
        n_j = max(n'_j, 1); m_j = m'_j/n_j;
    endfor;
    MSE = MSE';
} while (MSE < OldMSE)
```

Figure 1: Sequential k-means Algorithm.

```
P = MPI_Comm_size ();
μ = MPI_Comm_rank ();
MSE = LargeNumber;
if (μ = 0)
    Select k initial cluster centroids {m_j}_{j=1}^k;
endif;
MPI_Bcast ({m_j}_{j=1}^k, 0);
do {
    OldMSE = MSE;
    MSE' = 0;
    for j = 1 to k
        m'_j = 0; n'_j = 0;
    endfor;
    for i = μ * (n/P) + 1 to (μ + 1) * (n/P)
        for j = 1 to k
            compute squared Euclidean
                distance d^2(X_i, m_j);
        endfor;
        find the closest centroid m_ℓ to X_i;
        m'_ℓ = m'_ℓ + X_i; n'_ℓ = n'_ℓ + 1;
        MSE' = MSE' + d^2(X_i, m_ℓ);
    endfor;
    for j = 1 to k
        MPI_Allreduce (n'_j, n_j, MPI_SUM);
        MPI_Allreduce (m'_j, m_j, MPI_SUM);
        n_j = max(n_j, 1); m_j = m_j/n_j;
    endfor;
    MPI_Allreduce (MSE', MSE, MPI_SUM);
} while (MSE < OldMSE)
```

Figure 2: Parallel k-means Algorithm. See Table 1 for a glossary of various MPI routines used above.

MPI_Comm_size()	returns the number of processes
MPI_Comm_rank()	returns the process identifier for the calling process
MPI_Bcast(message, root)	broadcasts "message" from a process with identifier "root" to all of the processes
MPI_Allreduce(A, B, MPI_SUM)	sums all the local copies of "A" in all the processes (reduction operation) and places the result in "B" on *all* of the processes (broadcast operation)
MPI_Wtime()	returns the number of seconds since some fixed, arbitrary point of time in the past

Table 1: Conceptual syntax and functionality of MPI routines which are used in Figure 2. For the exact syntax and usage, see [21, 37].

From a programmer's perspective, parallel computing using MPI appears as follows. The programmer writes a single program in C (or C++ or FORTRAN 77), compiles it, and links it using the MPI library. The resulting object code is loaded in the local memory of every processor taking part in the computation; thus creating P "parallel" *processes*. Each process is assigned a unique identifier between 0 and $P-1$. Depending on its processor identifier, each process may follow a distinct execution path through the same code. These processes may communicate with each other by calling appropriate routines in the MPI library which encapsulates the details of communications between various processors.

Table 1 gives a glossary of various MPI routines which we use in our parallel version of k-means in Figure 2. Next, we discuss the design of the proposed parallel algorithm.

3.2 Parallel Algorithm Design

We begin by analyzing, in detail, the computational complexity of the sequential implementation of the k-means algorithm in Figure 1.

We count each addition, multiplication, or comparison as one floating point operation (flop). It follows from Figure 1 that the amount of computation within each k-means iteration is constant, where each iteration consists of "distance calculations" in lines 14-21 and a "centroid recalculations" in lines 22-26. A careful examination reveals that the "distance calculations" require roughly $(3nkd + nk + nd)$ flops per iteration, where $3nkd$, nk, and nd correspond to lines 15-17, line 18, and line 19 in Figure 1, respectively. Also, "centroid recalculations" require approximately kd flops per iteration. Putting these together, we can estimate the computation complexity of the sequential implementation of the k-means algorithm as

$$(3nkd + nk + nd + kd) \cdot \mathfrak{I} \cdot T^{\text{flop}}, \tag{2}$$

where $\mathfrak{I}$ denotes the number of k-means iterations and T^{flop} denotes the time (in seconds) for a floating point operation. In this paper, we are interested in the case when the number of data points n is quite large in an absolute sense, and also large relative to d and k. Under this condition the serial complexity of the k-means algorithm is dominated by

$$T_1 \sim (3nkd) \cdot \mathfrak{I} \cdot T^{\text{flop}}. \tag{3}$$

By implementing a version of k-means on a distributed memory machine with P processors, we hope to reduce the total computation time by nearly a factor of P. Observe that the "distance calculations" in lines 14-21 of Figure 1 are inherently data parallel, that is, in principle, they can be executed asynchronously and in parallel for each data point. Furthermore, observe that these lines dominate the computational complexity in (2) and (3), when the number of data points n is large. In this context, a simple, but effective, parallelization strategy is to divide the n data points into P blocks (each of size roughly n/P) and compute lines 14-21 for each of these blocks in parallel on a different processor. This is the approach adopted in Figure 2.

For simplicity, assume that P divides n. In Figure 2, for $\mu = 0, 1, \cdots, P-1$, we assume that the process identified by "μ" has access to the data subset $\{X_i, i = (\mu) * (n/P) + 1, \cdots, (\mu + 1) * (n/P)\}$. Observe that each of the P processes can carry out the "distance calculations" in parallel or asynchronously, if the centroids $\{m_j\}_{j=1}^k$ are available to each process. To enable this potential parallelism, in Figure 1, a local copy of the centroids $\{m_j\}_{j=1}^k$ is maintained for each process, see, line 7 and lines 22-26 in Figure 2 (see Table 1 for a glossary of the MPI calls used). Under this parallelization strategy, each process needs to handle only n/P data points, and hence we expect the total computation time for the parallel k-means to decrease to

$$T_P^{\mathrm{comp}} = \frac{T_1}{P} \sim \frac{(3nkd) \cdot \mathcal{I} \cdot T^{\mathrm{flop}}}{P}. \tag{4}$$

In other words, as a benefit of parallelization, we expect the computational burden to be shared equally by all the P processors. However, there is also a price attached to this benefit, namely, the associated communication cost, which we now examine.

Before each new iteration of k-means can begin, all the P processes must communicate to recompute the centroids $\{m_j\}_{j=1}^k$. This global communication (and hence synchronization) is represented by lines 22-26 of Figure 2. Since, in each iteration, we must "MPI_Allreduce" roughly $d \cdot k$ floating point numbers, we can estimate the communication time for the parallel k-means to be

$$T_P^{\mathrm{comm}} \sim d \cdot k \cdot \mathcal{I} \cdot T_P^{\mathrm{reduce}}, \tag{5}$$

where T_P^{reduce} denotes the time (in seconds) required to "MPI_Allreduce" a floating point number on P processors. On most architectures, one may assume that $T_P^{\mathrm{reduce}} = O(\log P)$ [9].

Line 27 in Figure 2 ensures that each of the P processes has a local copy of the total mean-squared-error "MSE", hence each process can independently decide on the convergence condition, that is, when to exit the "do{ $\cdots$ }while" loop.

In conclusion, each iteration of our parallel k-means algorithm consists of an asynchronous computation phase followed by a synchronous communication phase. The reader may compare Figures 1 and 2 line-by-line to see the precise correspondence of the proposed parallel algorithm with the serial algorithm. We stress that Figure 2 is optimized for understanding, and not for speed! In particular, in our actual implementation, we do not use $(2k+1)$ different "MPI_Allreduce" operations as suggested by lines 23, 24, and 27, but rather use a single block "MPI_Allreduce" by assigning a single, contiguous block of memory for the variables $\{m_j\}_{j=1}^k$, $\{n_j\}_{j=1}^k$, and MSE and a single, contiguous block of memory for the variables $\{m_j'\}_{j=1}^k$, $\{n_j'\}_{j=1}^k$, and MSE$'$.

We can now combine (4) and (5) to estimate the computational complexity of the parallel k-means algorithm as

$$T_P = T_P^{\mathrm{comp}} + T_P^{\mathrm{comm}} \sim \frac{(3nkd) \cdot \mathcal{I} \cdot T^{\mathrm{flop}}}{P} + d \cdot k \cdot \mathcal{I} \cdot T_P^{\mathrm{reduce}}. \tag{6}$$

It can be seen from (4) and (5) that the relative cost for the communication phase T_P^{comm} is insignificant compared to that for the computation phase T_P^{comp}, if

$$\boxed{\frac{P \cdot T_P^{\text{reduce}}}{3 \cdot T^{\text{flop}}} \ll n}. \tag{7}$$

Since the left-hand side of the above condition is a machine constant, as the number of data points n increases, we expect the relative cost for the communication phase compared to the computation phase to progressively decrease.

In the next section, we empirically study the performance of the proposed parallel k-means algorithm.

4 Performance and Scalability Analysis

Sequential algorithms are tested for correctness by seeing whether they give the right answer. For parallel programs, the right answer is not enough: we would like to decrease the execution time by adding more processors or we would like to handle larger data sets by using more processors. These desirable characteristics of a parallel algorithm are measured using "speedup" and "scaleup," respectively; we now empirically study these characteristics for the proposed parallel k-means algorithm.

4.1 Experimental Setup

We ran all of our experiments on an IBM SP2 with a maximum of 16 nodes. Each node in the multiprocessor is a Thin Node 2 consisting of a IBM POWER2 processor running at 160 MHz with 256 megabytes of main memory. The processors all run AIX level 4.2.1 and communicate with each other through the High-Performance Switch with HPS-2 adapters. The entire system runs PSSP 2.3 (Parallel System Support Program). See [36] for further information about the SP2 architecture.

Our implementation is in C and MPI. All the timing measurements are done using the routine "MPI_Wtime()" described in Table 1. Our timing measurements ignore the I/O times (specifically, we ignore the time required to read in the data set from disk), since, in this paper, we are only interested in studying the efficacy of our parallel k-means algorithm. All the timing measurements were taken on an otherwise idle system. To smooth out any fluctuations, each measurement was repeated five times and each reported data point is to be interpreted as an average over five measurements.

For a given number of data points n and number of dimensions d, we generated a test data set with 8 clusters using the algorithm in [27]. A public domain implementation of this algorithm is available from Dave Dubin [14]. The advantage of such data generation is that we can generate as many data sets as desired with precisely specifiable characteristics.

As mentioned in Section 2, each run of the k-means algorithm depends on the choice of the starting cluster centroids. Specifically, the initial choice determines the specific local minimum of (1) that will be found by the algorithm, and it determines the number of k-means iterations. To eliminate the impact of the initial choice on our timing measurements, for a fixed data set, identical starting cluster centroids are used–irrespective of the number of processors used.

We are now ready to describe our experimental results.

4.2 Speedup

Relative speedup is defined as the ratio of the execution time for clustering a data set into k clusters on 1 processor to the execution time for identically clustering the same data set on P processors. Speedup is a summary of the efficiency of the parallel algorithm.

Using (3) and (6), we may write relative speedup of the parallel k-means roughly as

$$\text{Speedup} = \frac{(3nkd) \cdot \mathcal{I} \cdot T^{\text{flop}}}{(3nkd) \cdot \mathcal{I} \cdot T^{\text{flop}}/P + d \cdot k \cdot \mathcal{I} \cdot T_P^{\text{reduce}}}, \tag{8}$$

which approaches the linear speedup of P when condition (7) is satisfied, that is, the number of data points n is large. We report three sets of experiments, where we vary n, d, and k, respectively.

Varying n: First, we study the speedup behavior when the number of data points n is varied. Specifically, we consider five data sets with $n = 2^{13}, 2^{15}, 2^{17}, 2^{19}$, and 2^{21}. We fixed the number of dimensions $d = 8$ and the number of desired clusters $k = 8$. We clustered each data set on $P = 1, 2, 4, 8$, and 16 processors. The measured execution times are reported in Figure 3, and the corresponding relative speedup results are reported in Figure 4. We can observe the following facts from Figure 4:

- For the largest data set, that is, $n = 2^{21}$, we observe a relative speedup of 15.62 on 16 processors. Thus, for large number of data points n our parallel k-means algorithm has nearly linear relative speedup.

  But, in contrast, for the smallest data set, that is, $n = 2^{11}$, we observe that relative speedup flattens at 6.22 on 16 processors.
- For a fixed number of processors, say, $P = 16$, as the number of data points increase from $n = 2^{11}$ to $n = 2^{21}$ the observed relative speedup generally increases from 6.22 to 15.62, respectively. In other words, our parallel k-means has an excellent *sizeup* behavior in the number of data points.

All these empirical facts are consistent with the theoretical analysis presented in the previous section; in particular, see condition (7).

Varying d: Second, we study the speedup behavior when the number of dimensions d is varied. Specifically, we consider three data sets with $d = 2, 4$, and 8. We fixed the number of data points $n = 2^{21}$ and the number of desired clusters $k = 8$. We clustered each data set on $P = 1, 2, 4, 8$, and 16 processors. For the sake of brevity, we omit the measured execution times, and report the corresponding relative speedup results in Figure 5.

Varying k: Finally, we study the speedup behavior when the number of desired clusters k is varied. Specifically, we clustered a fixed data set into $k = 2, 4, 8$, and 16 clusters. We fixed the number of data points $n = 2^{21}$ and the number of dimensions $d = 8$. We clustered the data set on $P = 1, 2, 4, 8$, and 16 processors. The corresponding relative speedup results are given in Figure 6.

In Figure 5, we observe nearly linear speedups between 15.42 to 15.53 on 16 processors. Similarly, in Figure 6, we observe nearly linear speedups between 15.08 to 15.65 on 16 processors. The excellent speedup numbers can be attributed to the fact that for $n = 2^{21}$

the condition (7) is satisfied. Also, observe that all the relative speedup numbers in Figures 5 and 6 are essentially independent of d and k, respectively. This is consistent with the fact that neither d nor k appears in the condition (7).

4.3 Scaleup

For a fixed data set (or a problem size), speedup captures the decrease in execution speed that can be obtained by increasing the number of processors. Another figure of merit of a parallel algorithm is *scaleup* which captures how well the parallel algorithm handles larger data sets when more processors are available. Our scaleup study measures execution times by keeping the problem size per processor fixed while increasing the number of processors. Since, we can increase the problem size in either the number of data points n, the number of dimensions d, or the number of desired clusters k, we can study scaleup with respect to each of these parameters at a time.

Relative scaleup of the parallel k-means algorithm with respect to n is defined as the ratio of the execution time (per iteration) for clustering a data set with n data points on 1 processor to the the execution time (per iteration) for clustering a data set with $n \cdot P$ data points on P processors–where the number of dimensions d and the number of desired clusters k are held constant. Observe that we measure execution time per iteration, and not raw execution time. This is necessary since the k-means algorithm may require a different number of iterations $\mathfrak{I}$ for a different data set. Using (3) and (6), we can analytically write relative scaleup with respect to n as

$$\text{Scaleup} = \frac{(3nkd) \cdot T^{\text{flop}}}{(3nPkd) \cdot T^{\text{flop}}/P + d \cdot k \cdot T_P^{\text{reduce}}}. \tag{9}$$

It follows from (9) that if

$$\boxed{\frac{T_P^{\text{reduce}}}{3 \cdot T^{\text{flop}}} \ll n}, \tag{10}$$

then we expect relative scaleup to approach the constant 1. Observe that condition (10) is weaker than (7), and will be more easily satisfied for large number of data points n which is the case we are interested in. Relative scaleup with respect to either k or d can be defined analogously; we omit the precise definitions for brevity. The following experimental study shows that our implementation of parallel k-means has linear scaleup in n and k, and surprisingly better than linear scaleup in d.

Scaling n: To empirically study scaleup with respect to n, we clustered data sets with $n = 2^{21} \cdot P$ on $P = 1, 2, 4, 8, 16$ processors, respectively. We fixed the number of dimensions $d = 8$ and the number of desired clusters $k = 8$. The execution times per iteration are reported in Figure 7, from where it can be seen that the parallel k-means delivers virtually constant execution times in number of processors, and hence has excellent scaleup with respect to n. The largest data set with $n = 2^{21} \cdot 16 = 2^{25}$ is roughly 2 gigabytes. For this data set, our algorithm drives the SP2 at nearly 1.2 gigaflops. Observe that the main memory available on each of the 16 nodes is 256 megabytes, and hence this data set will not fit in the main memory of any single node, but easily fits in the combined main memory of 16 nodes. This is yet another benefit of parallelism–the ability to cluster significantly large data sets *in-core*, that is, in main memory.

Scaling k**:** To empirically study scaleup with respect to k, we clustered a data set into $k = 8 \cdot P$ clusters on $P = 1, 2, 4, 8, 16$ processors, respectively. We fixed the number of data points $n = 2^{21}$, and the number of dimensions $d = 8$. The execution times per iteration are reported in Figure 7, from where it can be seen that our parallel k-means delivers virtually constant execution times in number of processors, and hence has excellent scaleup with respect to k.

Scaling d**:** To empirically study scaleup with respect to d, we clustered data sets with the number of dimensions $d = 8 \cdot P$ on $P = 1, 2, 4, 8, 16$ processors, respectively. We fixed the number of data points $n = 2^{21}$, and the number of desired clusters $k = 8$. The execution times per iteration are reported in Figure 7, from where it can be seen that our parallel k-means delivers better than constant execution times in number of processors, and hence has surprisingly nice scaleup with respect to d. We conjecture that this phenomenon occurs due to the reduced loop overhead in the "distance calculations" as d increases (see Figure 2). The largest data set with $d = 8 \cdot 16 = 128$ is roughly 2 gigabytes. For this data set, our algorithm drives the SP2 at nearly 1.8 gigaflops.

5 Future Work

In this paper, we proposed a parallel k-means algorithm for distributed memory multiprocessors. Our algorithm is also easily adapted to shared memory multiprocessors where all processors have access to the same memory space. Many such machines are now currently available from a number of vendors. The basic strategy in adapting our algorithm to shared memory machine with P processors would be the same as that in this paper, namely, divide the set of data points n into P blocks (each of size roughly n/P) and compute distance calculations in lines 14-21 of Figure 1 for each of these blocks in parallel on a different processor while ensuring that each processor has access to a separate copy of the centroids $\{m_j\}_{j=1}^k$. Such an algorithm can be implemented on a shared memory machine using threads [28].

It is well known that the k-means algorithm is a hard thresholded version of the expectation-maximization (EM) algorithm [26]. We believe that the EM algorithm can be effectively parallelized using essentially the same strategy as that used in this paper.

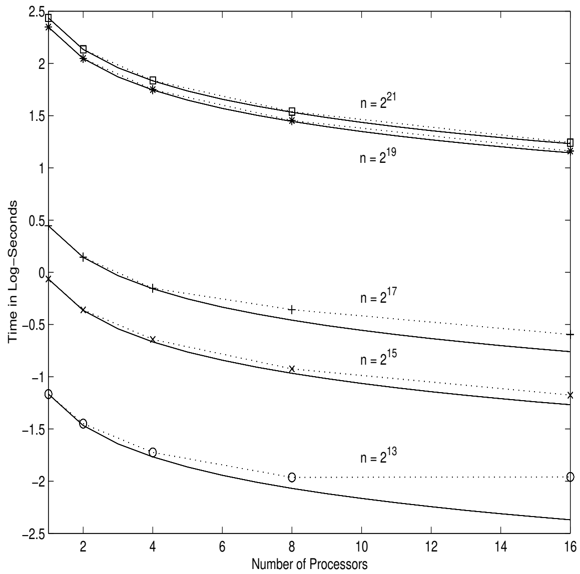

Figure 3: Speedup curves. We plot execution time in $\log_{10}$-seconds versus the number of processors. Five data sets are used with number of data points $n = 2^{13}, 2^{15}, 2^{17}, 2^{19}$, and 2^{21}. The number of dimensions $d = 8$ and the number of clusters $k = 8$ are fixed for all the five data sets. For each data set, the k-means algorithm required $\mathfrak{I} = 3, 10, 8, 164$ and 50 number of iterations, respectively. For each data set, a dotted line connects the observed execution times, while a solid line represents the "ideal" execution times obtained by dividing the observed execution time for 1 processor by the number of processors.

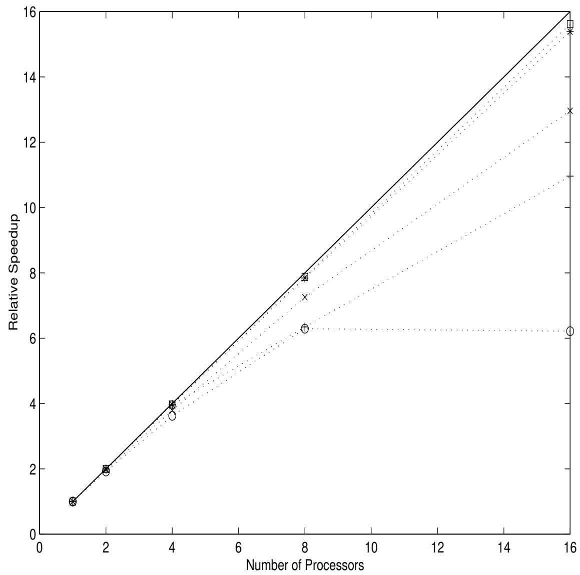

Figure 4: Relative Speedup curves corresponding to Figure 3. The solid line represents "ideal" linear relative speedup. For each data set, a dotted line connects observed relative speedups.

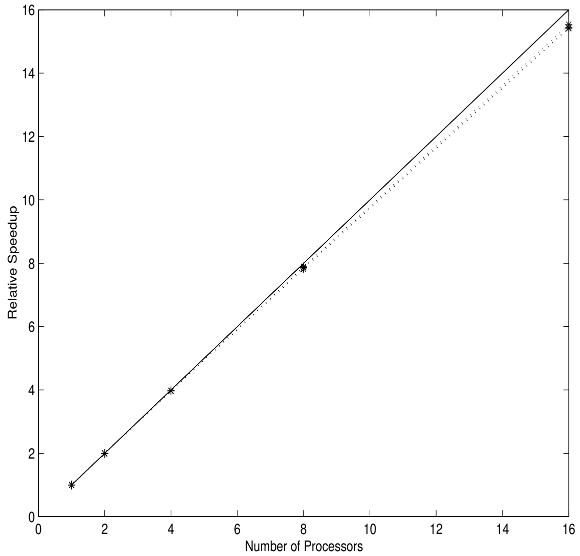

Figure 5: Relative speedup curves for three data sets with $d = 2, 4$, and 8. The number of data points $n = 2^{21}$ and the number of clusters $k = 8$ are fixed for all the three data sets. The solid line represents "ideal" linear relative speedup. For each data set, a dotted line connects observed relative speedups. It can be seen that relative speedups for different data sets are virtually indistinguishable from each other.

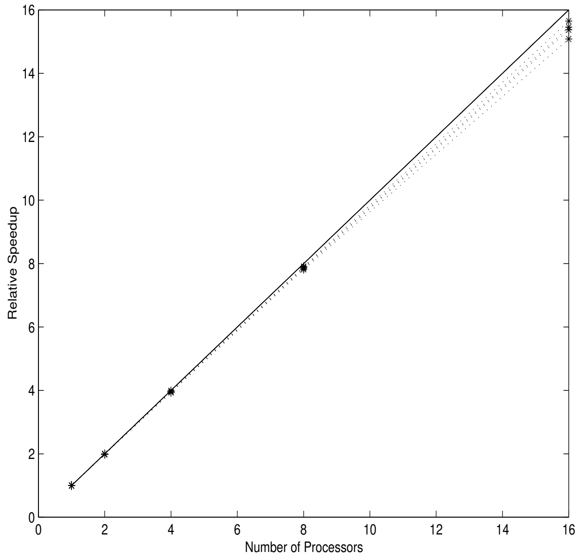

Figure 6: Relative speedup curves for four data sets with $k = 2, 4, 8$, and 16. The number of data points $n = 2^{21}$ and the number of dimensions $d = 8$ are fixed for all the four data sets. The solid line represents "ideal" linear relative speedup. For each data set, a dotted line connects observed relative speedups. It can be seen that relative speedups for different data sets are virtually indistinguishable from each other.

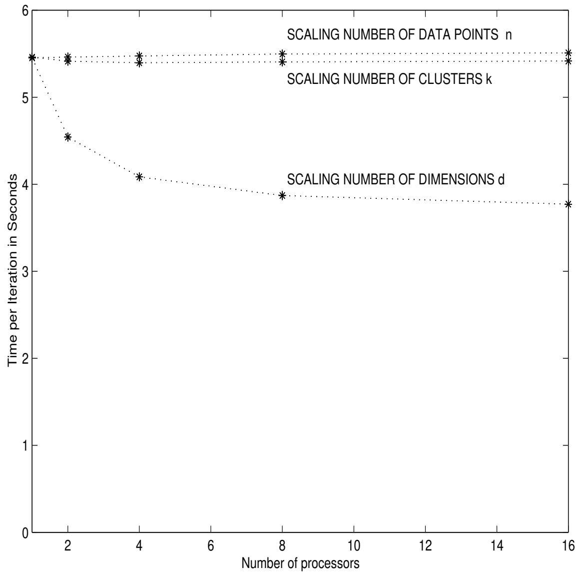

Figure 7: Scaleup curves. We plot execution time per iteration in seconds versus the number of processors. The same data set with $n = 2^{21}$, $d = 8$, and $k = 8$ is used for all the three curves–when the number of processors is equal to 1. For the "n" curve, the number of data points is scaled by the number of processors, while d and k are held constant. For the "k" curve, the number of clusters is scaled by the number of processors, while n and d are held constant. For the "d" curve, the number of dimensions is scaled by the number of processors, while n and k are held constant.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Eng.*, 8(6):962–969, 1996.

[2] K. Alsabti, S. Ranka, and V. Singh. An efficient k-means clustering algorithm. In *Proceedings of IPPS/SPDP Workshop on High Performance Data Mining*, 1998.

[3] D. Boley, M. Gini, R. Gross, E.-H. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore. Document categorization and query generation on the World Wide Web using WebACE. *AI Review*, 1998.

[4] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In G. Tesauro and D. Touretzky, editors, *Advances in Neural Information Processing Systems 7*, pages 585–592. The MIT Press, Cambridge, MA, 1995.

[5] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. Technical Report 1997-015, Digital Systems Research Center, 1997. http://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-015-html/.

[6] J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H. Hüning, M. Köhler, J. Sutiwaraphun, H. W. To, and D. Yang. Large scale data mining: Challenges and responses. In D. Pregibon and R. Uthurusamy, editors, *Proceedings Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, CA*, pages 61–64. AAAI Press, 1997.

[7] P. Cheeseman and J. Stutz. Bayesian classification (autoclass): Theory and results. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 153–180. AAAI/MIT Press, 1996.

[8] D. W. Cheung and Y. Xiao. Effect of data distribution in parallel mining of associations. *Data Mining and Knowledge Discovery*, 1999. to appear.

[9] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[10] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/gather: A cluster-based approach to browsing large document collections. In *ACM SIGIR*, 1992.

[11] I. S. Dhillon and D. S. Modha. Concept decompositions for large sparse text data using clustering. Technical Report Research Report RJ 10147 (95022), IBM Almaden Research Center, July 8, 1999.

[12] I. S. Dhillon, D. S. Modha, and W. S. Spangler. Visualizing class structure of multidimensional data. In S. Weisberg, editor, *Proceedings of the 30th Symposium on the Interface: Computing Science and Statistics, Minneapolis, MN*, May 13–16 1998.

[13] I. S. Dhillon, D. S. Modha, and W. S. Spangler. Visualizing class structure of high-dimensional data with applications. *Submitted for publication*, 1999.

[14] D. Dubin. clusgen.c. http://alexia.lis.uiuc.edu/~dubin/, March 1996.

[15] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* Wiley, 1973.

[16] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[17] A. A. Freitas and S. H. Lavington. *Mining Very Large Databases with Parallel Processing.* Kluwer Academic Publishers, 1998.

[18] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, pages 750–753, 1975.

[19] M. R. Garey, D. S. Johnson, and H. S Witsenhausen. The complexity of the generalized Lloyd-Max problem. *IEEE Trans. Inform. Theory*, 28(2):255–256, 1982.

[20] A. Gersho and R. M. Gray. *Vector quantization and signal compression.* Kluwer Academic Publishers, 1992.

[21] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* The MIT Press, Cambridge, MA, 1996.

[22] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *SIGMOD Record: Proceedings of the 1997 ACM-SIGMOD Conference on Management of Data, Tucson, AZ, USA*, pages 277–288, 1997.

[23] J. A. Hartigan. *Clustering Algorithms.* Wiley, 1975.

[24] M. V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, Orlando, FL, USA*, pages 573–579, 1998.

[25] H. Kargupta, I. Hamzaoglu, B. Stafford, V. Hanagandi, and K. Buescher. PADMA: Parallel data mining agents for scalable text classification. In *Proceedings of the High Performance Computing, Atlanta, GA, USA*, pages 290–295, 1997.

[26] G. J. McLachlan and T. Krishnan. *The EM Algorithm and Extentions.* Wiley, 1996.

[27] G. Milligan. An algorithm for creating artificial test clusters. *Psychometrika*, 50(1):123–127, 1985.

[28] C. J. Northrup. *Programming with UNIX Threads.* John Wiley & Sons, 1996.

[29] E. Rasmussen. Clustering algorithms. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 419–442. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[30] M. Sahami, S. Yusufali, and M. Baldonado. SONIA: A service for organizing networked information autonomously. In *Digital Libraries 98: Proceedings of the Third ACM Conference on Digital Libraries, New York, NY*, pages 200–209. ACM, 1999.

[31] SAS Institute, Cary, NC, USA. *SAS Manual*, 1997.

[32] J.C. Shafer, R. Agrawal, and M. Mehta. A scalable parallel classifier for data mining. In *Proc. 22nd International Conference on VLDB, Mumbai, India*, 1996.

[33] C. T. Shaw and G. P. King. Using cluster analysis to classify time series. *Physica D*, 58:288–298, 1992.

[34] C. Silverstein and J. O. Pedersen. Almost-constant-time clustering of arbitrary corpus subsets. In *ACM SIGIR*, 1997.

[35] P. Smyth, M. Ghil, K. Ide, J. Roden, and A. Fraser. Detecting atmospheric regimes using cross-validated clustering. In D. Pregibon and R. Uthurusamy, editors, *Proceedings Third International Conference on Knowledge Discovery and Data Mining, Newport Beach, CA*, pages 61–64. AAAI Press, 1997.

[36] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[37] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, MA, 1997.

[38] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. In *Proc. 1998 International Conference on Parallel Processing*, 1998.

[39] Paul Stolorz and Ron Musick. *Scalable High Performance Computing for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, 1997.

[40] P. Willet. Recent trends in hierarchic document clustering: a critical review. *Information Processing & Management*, 24(5):577–597, 1988.

[41] M. J. Zaki, C. T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. Technical report, IBM Almaden Research Center, 1998.

[42] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New parallel algorithms for fast discovery of association rule. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.

[43] O. Zamir and O. Etzioni. Web document clustering: A feasibility demonstration. In *ACM SIGIR*, 1998.

[44] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data, Montreal, Canada*, 1996.