Parallel *K* - Means Algorithm on Distributed Memory Multiprocessors

Manasi N. Joshi
Spring 2003
Computer Science Department
University of Minnesota, Twin Cities

Abstract

Clustering large data sets can be time consuming and processor intensive. This project is an implementation of the parallel version of a popular clustering algorithm, the k-means algorithm, to provide faster clustering solutions. This algorithm was tested such that 3,4,5,7 clusters were created on a cluster of Sun workstations. Optimal levels of speedup were not achieved; but the benefits of parallelization were observed. This methodology exploits the inherent data-parallelism in the *k*-means algorithm and makes use of the message-passing model.

1 Introduction

Clustering is the process of partitioning or grouping a given set of patterns into disjoint *clusters*. This is one such that patterns in the same cluster are alike and patterns belonging to two different clusters are different. Clustering is the grouping of similar objects and a clustering of a set is a partition of its elements that is chosen to minimize some measure of dissimilarity. Clustering algorithms are often useful in applications in various fields such as visualization, pattern recognition, learning theory, computer graphics, neural networks, AI, and statistics. Practical applications of clustering include unsupervised classification and taxonomy generation, nearest neighbor searching, time series analysis, text analysis and navigation.

Data clustering is a frequently used and is a useful tool in the data mining. There are a variety of data clustering algorithms, which are generally classified, into two categories: hierarchical algorithms and partitional algorithms. A hierarchical algorithm creates a hierarchical decomposition of the given data set forming a tree, which splits the data set recursively into smaller sub-trees. A partitional clustering algorithm obtains a single partition of the data instead of a clustering structure like a tree produced by the hierarchical technique.

Clustering:

- Have to define some notion of "similarity" between examples
- Goal: maximize intra-cluster similarity and minimize inter-cluster similarity

2 K-Means

The *k*-means method has been shown to be effective in producing good clustering results for many practical applications. K-means method is well known for its relatively simple implementation and decent results. However, a direct algorithm of k-means method requires time proportional to the product of number of documents (vectors) and number of clusters per iteration. This is computationally very expensive especially for large datasets.
The algorithm is an iterative procedure and requires that the number of clusters k be given a priory. Suppose that the k initial cluster centers are given, and then the algorithm follows the steps as below:

1. Compute the Euclidean distance from each of the documents to each cluster center. A document is associated with a cluster such that its distance from that cluster is the smallest among all such distances.
2. After this assignment or association of all the documents with one of k clusters is done, each cluster center is recomputed so as to reflect the true mean of its constituent documents.
3. Steps 1 and 2 are repeated until the convergence is achieved.

The *k*-means Algorithm:
Suppose there are n data points or documents $X_1, X_2, \ldots, X_n$ [Dhillon] are given such that each one of them belongs to $\mathbf{R^d}$. The problem of finding the minimum variance clustering of this data set into k clusters is that of finding the k points $\{ m_j \}_{j=1}^{k}$ in $\mathbf{R^d}$ such that

$$(1/n) * \sum (\min_j d^2 (X_i , m_j)), \text{ for } i = 1 \text{ to } n,$$

is minimized, where d (Xi , m_j) denotes the Euclidean distance between X_i and m_j. The points $\{ _j \}_{j=1}^{k}$ are known as cluster centroids or as cluster means.

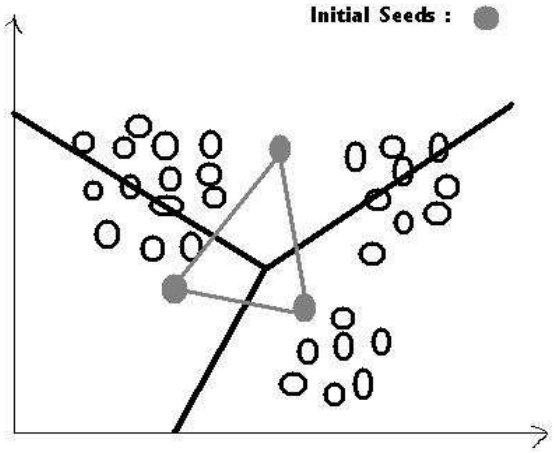


Fig 1: Initial Seeds of the *k*- Means

Reasons for popularity of *k*-means algorithm are ease of interpretation, simplicity of implementation, scalability, speed of convergence, and adaptability to sparse data. The *k*-means algorithm can simply be explained as follows:

1. Phase Initialization:

Select a set of k starting points $\{ m_j \}_{j=1}^{k}$ in $\mathbf{R^d}$. This selection may be done in a random manner or making use of some heuristic. See Fig 1.

2. Phase 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.

3. Phase Centroid Recalculation:

For each $1 \leq j \leq k$, recompute cluster centroid m_j as the average of data points assigned to it. See Fig 2.

4. Convergence Condition:

Repeat steps 2 and 3 until convergence. At convergence see Fig 3.

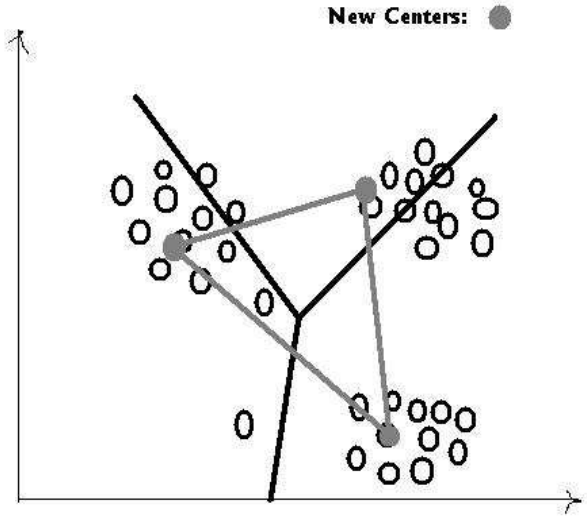


Fig 2: Calculating New Cluster Centers

Before this algorithm converges steps 2 and 3 are executed a number of times, say μ**.** The positive integer **μ** is known as number of *k*-means iterations. The precise value of **μ** can vary depending on the initial staring cluster centroids, even on the same data set.

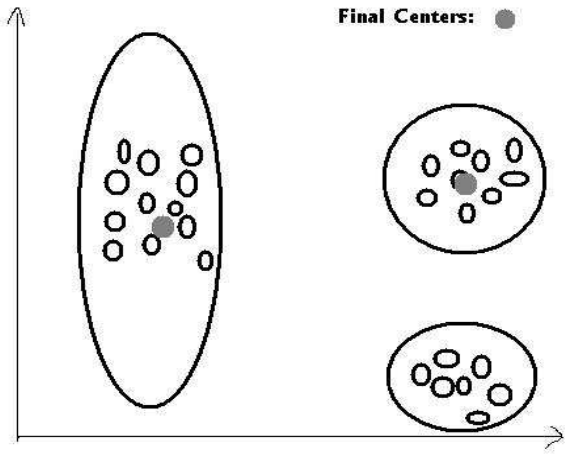


Fig 3: Final Clustering achieved by *k*- Means

3 Parallel *k*-means

Parallel *k*-means has been studied by [Dhillon], [Xu], [Stoffle] previously for very large databases. There has been some study for studying the advantages of the parallelism in the *k*-means procedure. The speedup and scale-up variation with respect to number of documents (vectors), the K of the *k*-means and the dimension of each of the documents has been studied [Dhillon].

Parallelization is basically studied for performance advantages mainly the data partitioning and parallel processing. Inherent to classical k-means algorithm is intrinsic parallelism. The most intensive calculation to occur is the calculation of distances, and it would require a total of dk distance computations where **d** is the number of documents and **k** is the number of clusters being created. On the parallel system, the main idea is that of splitting the dataset among processes for faster computation of vector-cluster membership for each partition generated. However it is important to note that the communication cost between the processors is significant (for small datasets $T_{communication} \geq T_{computation}$). So, for an efficient parallel algorithm, it is necessary to minimize the communication between nodes. The centroid lists consists of k vectors of length d each and in practical applications, it will be much smaller compared to the actual data set, which consists of n vectors each of length d. Upon beginning the clustering, each process will receive the entire centroid list, but only the process with id as the "root" will compute the initial centroids and then broadcast this selected k initial centroids to all other processes. Each process takes care of only the part of the dataset. Thus each process is responsible for (n / number of processes) vectors rather than the entire set. Each process will compute the distances for only part of vectors to each of the centroids, which are all locally stored on each process. A series of assignments are generated mapping vectors (documents) to clusters. Each process then gathers the sum of all vectors allocated to a given cluster and

computes the mean of the vectors assigned to a particular cluster. This is repeated for every cluster and a new set of centroids is available on every process, and the vectors can be re-assigned with respect to the newly calculated centroids.
To compute the quality of the given partition, each process can compute the mean squared error for the portion of dataset over which it is operating. As these values are calculated, the process will sum its local mean squared error values. A simple reduction of these values among all processes will determine the overall quality of the cluster.

- Root calculates the initial means
- Replicate the k centroids
- Each processor computes distance of each local document vector to the centroids
- Assign points to closest centroid and compute local MSE (Mean Squared Error)
- Perform reduction for global centroids and global MSE value

In the implementation of this algorithm, Single Program Multiple Data (SPMD) model was assumed and the Message Passing Interface was used which works the best for computing on distributed memory multiprocessors. The communication between the processors is captured using MPI.

4 Message Passing Interface

MPI: A specification for message passing libraries, designed to be a standard for distributed memory, message passing, parallel computing.
The important features: portability, heterogeneity, topologies and modularity
All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing the resulting algorithm using MPI constructs.
There are a variety of implementations available, both vendor and public domain.
It is possible to be used with C and FORTRAN programs.

MPI Message Passing Routines
Following MPI routines are used:

MPI_Init() :
Initialize the MPI execution environment

MPI_Comm_size():
Returns the number of processes currently running the program

MPI_Comm-rank():
Returns the process identifier for the calling process

MPI_Bcast():

Broadcast “message” form a process with identifier “root” to all the processes

MPI_Allreduce(A, B, MPI_SUM):

Sums up all the local copies of “A” in all the processes (reduction operation) and place 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. Between the two calls to MPI_Wtime() to get the execution time of the parallel program, this past reference is kept the same

MPI_Finalize():

Terminate the MPI execution environment.

Broadcast:

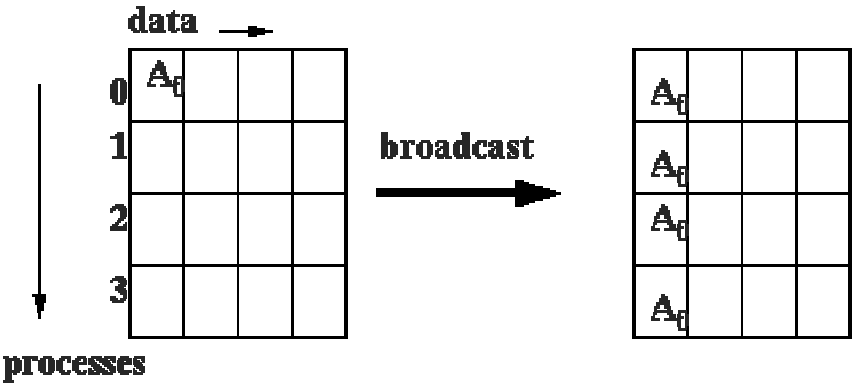


Fig 4: MPI Broadcast Operation

Allreduce:

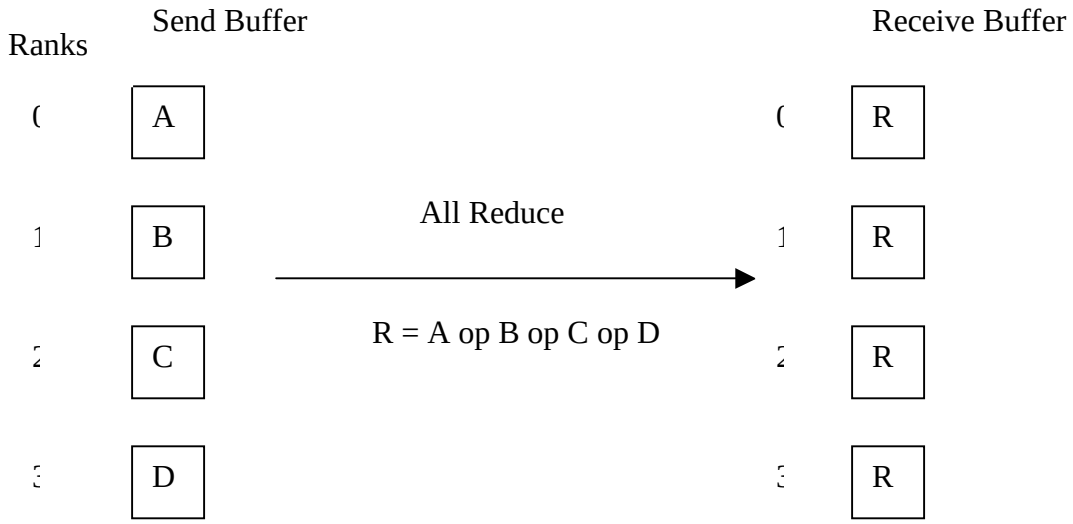


Fig 5: MPI All-reduce Operation

When MPI code is compiled, the copy of the running program is executed on each of the p processes involved. Each process is assigned an identifier (rank using MPI_Comm_rank()) from 0 to p-1. Every process has its own memory.

For the parallel k-means algorithm, there is a communication in the beginning, when the initial guess of the centroid vectors is broadcast to each node. At the stage of distance recomputation and quality determination, there must be communication between processes. If the number of vectors to cluster dominates the length of each vector for the clustering algorithm, then it is safe to assume that there is no difference in cost between any two communications within the centroid re-computation and quality determination process.

In re-computing the centroids, each process checks its local set of vectors to see which vectors has been already assigned to a cluster C_i. The process stores the sum of these vectors into a new vector. The count of how many vectors belong to the same cluster is also updated each time as required. This information, which exists on every process, is now used in interprocess communication. The MPI_Allreduce () command is called to compute the overall sum of arrays stored on individual processes. The summed array is also available on every process, so that each of these can compute the new centroids.

While computing the quality of a resulting partition, each process calculates, for its part of vectors, the mean squared error of each vector to its corresponding centroid. The values computed are summed and are represented as a single value one each process using the MPI_Allreduce () operation. This ensures that all the processes have the same quality measure to evaluate against the stopping condition.

Each iteration of the algorithm uses k communications for re-computation of the centroids. Additional communication is required to determine the quality of the partition.

5 Heuristics to compute the initial centroids

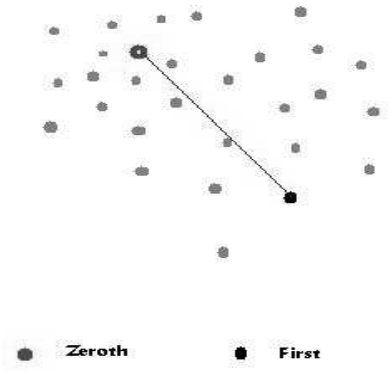
The partitioning of the documents/ vectors highly depends on the initial choice of the centroids, in the *k*-means algorithm. Both in serial as well as the parallel version of *K*-means algorithm, this determination of initial cluster centroids is done in a sequential manner; there is no parallel component to it. In the parallel *k*-means, only the process with id "root" does this.

What is important here is that we need to choose the initial *k* means (centroids) of the given dataset. Let the number of documents / vectors in the dataset be ndoc. Then these initial centroids are generated randomly with some specified seed-value for the random number generator. This choice of random number seed is very crucial. Note that this choice of initial centroids should be uniformly random.

For this purpose an extension of the Bisecting *K*-means algorithm to generate the initial centroids was used incase of k = 3. Note that this computation of initial centroid calculation increases the serial component in the overall otherwise parallel implantation of the algorithm. Hence it is to be kept simple. Following steps are used to choose the initial means:

1. Choose the **zeroth** mean as any random number.
2. Using this **zeroth** value for the mean, **first** mean is computed as the one which is the farthest from the zeroth value among all the other values. Using this value of first mean, similarly the **second** mean is computed
3. For the **third** mean, the vector with maximum sum of squared distances between that vector and first mean and that vector and the second mean is chosen. By this time the initially chosen zeroth value is discarded.

Other than this extension of the Bisecting *K*-means algorithm for choice of initial means, we could also use some method like PDDP for initial centroids.



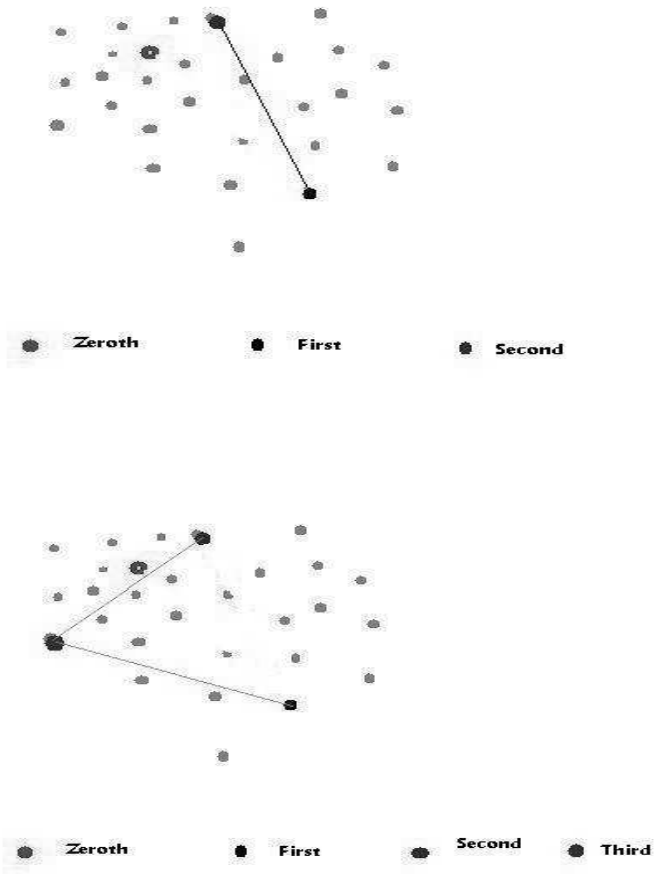


Fig 6: Extension of Bisecting *k*-means for Initial Centers

6 Analysis

The computational complexity for the sequential *k*-means algorithm comes form counting individual addition, multiplication or comparison as 1 flop (using notation given before). The most expensive part is computation of the distances.

By implementing a version of *k*-means on a distributed memory machine with P processors, the total computation time is reduced by P. For simplicity it was assumed that P divides n. The process identified by 'myid' has access to data subset $\{X_i, i = (myid) *(n/P),\ldots, (myid + 1)*(n/P)\}$. Each of the P processes can carry out the "distance calculations" in parallel or asynchronously, if the centroids $\{m_j\}$ $j = 1, \ldots, K$ are available to each process. To enable this potential for parallelism a local copy of the centroids is maintained at each process. Under parallelization each process has to handle only (n/P) data points and hence the cost is given by-

Sequential: ($3nkd + nk + nd + kd$)* μ * T
-- *3nkd* for distance calculation
-- *nk* finding the closest centroid
-- *nd* ($m'_l = m'_l + X_i$; $n'_l = n'_l + 1$;)

Parallel: ($3nkd$ * μ * T)
$$\frac{3nkd * \mu * T}{P} + d*k* \mu * Tp^{reduce}$$

Before each new iteration of K-means can begin, all the P processors must communicate to recomputed the centroids $\{m_j\}$ j = 1, … ,K. In addition to this, 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(…)while" loop.

As the number of data-points n increases, it is expected that the relative cost for the communication phase compared to computation phase to progressively decrease.

7 Experimental Data

To obtain the experimental data sets, UCI Machine learning Data Repository was used and the algorithm was tested with following data sets.

Iris Data Set:
Title: Iris Plants Database
Number of Instances**:** 150 (50 in each of three classes)
Number of Attributes**:** 4 (sepal length in cm, sepal width in cm, petal length in cm, petal width in cm)
Class Distributions**:**
-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica
Missing Attribute Values: None

Wine Data Set:
Title**:** Wine recognition data
Number of Instances**:** 178
Number of Attributes**:** 13
(Alcohol, Malic acid, Ash, Alcalinity of ash, Magnesium, Total phenols, Flavanoids, Nonflavanoid phenols, Proanthocyanins, Color intensity, Hue, OD280/OD315 of diluted wines, Proline)
Class Distribution**:** number of instances per class
Class 1 --59

Class 2 --71
Class 3 --48
Missing Attribute Values: None

Soybean Data set:
Title**:** Small Soybean Database
Number of Instances**:** 47
Number of Attributes: 35 (all have been normalized)
Class Distribution:
D1: 10
D2: 10
D3: 10
D4: 17
Number of Missing Attribute Values: 0

This table shows the variation of means and its effect on the clusters. Here M_i (i = 1,…,4)denote the means selected by randomization and the D_i (i = 1,…,4) denote the classification into individual clusters by program. This table summarizes the observations in 3 datasets (Soybean, Iris and Wine).

	M1	M2	M3	M4		D1	D2	D3	D4
Soybean Data Set	8	12	20	30		10	10	12	15
With k = 4	8	12	20	38		10	10	11	16
[10 10 10 17]	8	19	23	18		10	10	10	17
Iris Data Set	5	58	115			51	54	45	
With k = 3	40	80	120			50	48	52	
[50 50 50]	5	53	120			50	52	48	
	5	65	115			51	51	48	
Wine Data Set	50	117	127			61	27	90	
With k = 3	70	85	167			62	65	51	
[59 71 48]	56	91	167			59	65	54	

Table 1: Summary of Means for Datasets and Corresponding Clustering

Conclusion

In this project, I implemented the parallel version of the popular *K*-means algorithm on a cluster of Sun Workstations. I have considered only hard clustering (a document belongs to only one cluster at a time) on a shared nothing (SPMD) model for the processors. The choice of initial centroid vectors is very crucial and caused a large variation on changing slightly. Hence to control such deviation from the true boundaries of classification, I have used the extension of Bisecting *K*-means algorithm for computing initial centroids. The benefits of parallelization were observed. But the sequential cost due to initial centroids limits the speedup. In some databases the clusters easily pulled off the centers by outliers.

Future Work

In this project parallel *k*-means algorithm for the distributed memory multiprocessors was implemented for SPMD model. This could also be extended for shared memory multiprocessors. The datasets with no missing values were mainly considered and some heuristic could be included in the implementation to take care of this issue. To avoid poor speedups very large datasets could be used for testing the advantages of parallelization. Deviation or outlier detection issues cabn be considered. Probabilistic variants of *K*-means as well as for very large categorical datasets *K*-modes [Huang] variant of k-means could also be parallelized.

References

[Zhang *et.al.*2002] ShuTing Xu and Jun Zhang. An improved parallel algorithm for web document clustering.

[Dhillon *et.al.*1993] I.S. Dhillon and D.S. Modha. A Data-clustering algorithm on distributed memory multiprocessors.

[Siegel 2002] Matthew C. Siegel. A parallelization of the batch k-means clustering algorithm.

[Kumar *et.al.*] V. Kumar and M. Zaky. Tutorial PM-3 – High performance data mining.

[Huang 2000] Zhexue Huang. *K*-means-type Algorithms on Distributed Memory Computer. *International Journal of High Speed Computing*, 11 (2000), 75-91.

[Stoffle *et.al.*1999] K. Stoffle and A. Belkonience. “Parallel *k*-Means clustering for large datasets”. Proceedings of EuroPar -1999.

http://www-unix.mcs.anl.gov/mpi/mpich/