Parallel k/h-Means Clustering for Large Data Sets

Kilian Stoffel and Abdelkader Belkoniene

Université de Neuchâtel
Groupe Informatique
Pierre-à-Mazel 7
CH-2000 Neuchâtel
(Switzerland)
{Kilian.Stoffel;Abdelkader.Belkoniene}@seco.unine.ch
Phone: ++41 32 718 1376, Fax: ++41 32 718 1231

Abstract. This paper describes the realization of a parallel version of the k/h-means clustering algorithm. This is one of the basic algorithms used in a wide range of data mining tasks. We show how a database can be distributed and how the algorithm can be applied to this distributed database. The tests conducted on a network of 32 PCs showed for large data sets a nearly ideal speedup.

1 Introduction

Clustering, the process of grouping similar objects, is a well known and a well studied problem. Some of early work has been done in statistics (e.g. [2][7]). In more recent years clustering was identified as a key technique in data mining tasks [3]. This fundamental operation can be applied to many common tasks such as unsupervised classification, segmentation and dissection. We are focusing here on one specific algorithm for clustering namely k/h-means clustering [1]. The original version of the k/h-means algorithm was designed for numerical data [4][6][5].

Our contribution in this paper is the development of a parallel version of the k/h-means algorithm. We present a realization of this algorithm on top of a distributed object store installed on a simple PC network.

2 The k/h-Means Algorithm

The k/h-means belongs to the class of the partitional or non-hierarchical clustering algorithms. The goal of the algorithm is, starting from a set of objects $\mathcal{X}$ and an integer number K, to find a partitioning of the objects in $\mathcal{X}$ into K clusters. If M denotes the number of objects in the database then $K \leq M$. The partitioning is optimized in the sense that the distance between the objects in one cluster is minimized. This optimization is often called "whiting group sum squared error minimization".

More formally this problem can be defined in the following way:
Given $\mathcal{X}$ a set of M objects. Let $X_i = [x_{i1}, x_{i2}, ..., x_{in}]$ denote a vector that represents the values of i^{th} object over the n attributes, let $\mathcal{P}_0(M,K)$ be a partition matrix, and let $C_l,\ l = 1, K$ be the clusters of the partition. Each of the M objects lies in one of the K clusters. The mean of the K clusters are defined by $\bar{X}_l = \frac{1}{n_l}\sum_{i\in C_l} X_i,\ l = 1, K$ where n_l stands for the number of elements in cluster C_l. The error for a given cluster C_l is given by: $\epsilon_l = \sum_{i\in C_l}\delta(X_i, \bar{X}_l)$ where δ represents any distance function. Then the error for a partition $\mathcal{P}_0(M,K)$ can be defined as: $\epsilon[\mathcal{P}_0(M,K)] = \sum_{l=1}^{K}\epsilon_l$ The overall goal is to minimize $\epsilon[\mathcal{P}_0(M,K)]$. The large number of all possible partitions makes it impracticable to search for an optimal partition $\mathcal{P}^*(M,K)$ such that $\epsilon[\mathcal{P}^*(M,K)]$ is minimal. Instead, it is necessary to use a local optimization technique.

Let $\mathcal{P}_1(M,K)$ be a partition in the neighborhood of $\mathcal{P}_0(M,K)$ obtained by moving the object k from the cluster C_r of the partition $\mathcal{P}_0(M,K)$ to the cluster C_j of the partition $\mathcal{P}_1(M,K)$. If for δ we use the squared Euclidean distance, then the relation between $\epsilon[\mathcal{P}_0(M,K)]$ and $\epsilon[\mathcal{P}_1(M,K)]$ is given by:
$\epsilon[\mathcal{P}_1(M,K)] = \epsilon[\mathcal{P}_0(M,K)] + \frac{n_j}{n_j+1}\delta(\bar{X}_j, X_k) - \frac{n_r}{n_r-1}\delta(\bar{X}_r, X_k)$. $\epsilon[\mathcal{P}_1(M,K)]$ decreases if $\frac{n_j}{n_j+1}\delta(\bar{X}_j, X_k) - \frac{n_r}{n_r-1}\delta(\bar{X}_r, X_k) < 0$ or $\frac{n_j}{n_j+1}\delta(\bar{X}_j, X_k) < \frac{n_r}{n_r-1}\delta(\bar{X}_r, X_k)$

3 The Parallel *k*/*h*-Means Algorithms

In the previous section we describe the original *k*-means algorithm. This algorithm can be translated into the pseudo code version given in Figure 1. This is the version we will use to present the reflection to be made in order to parallelize the algorithm given above. However, another formulation of the *k*-means algorithm exists. This version is sometimes also called *h*-means . Very often people do not distinguish between the two algorithms. However, we think it is important to make this difference, even though the conceptual difference in the algorithms is very small. The only difference is the place in the algorithm where the means are recalculated. Instead of recalculating them after each migration of an object, all objects are migrated if a better cluster assignment can be found, and then the new mean values are calculated (see Figure 1). The theoretical properties of the two algorithms are different. The *k*-means algorithm has better convergence properties and is less likely to get stuck in a local minimum. However in a wide range of tests we conducted, no major differences could be measured.

We will now show how the *k*-means algorithm can be parallelized. We will first start with the `mainLoop` given in Figure 1. At first glance this loop seems to be ideal for the parallelization of an object based data distribution. However, the parallelization of the loop as it stands is very difficult. The problem lies in the link between the two function calls `getNearestMean` and `recalculateMean`. Once an object is selected to be processed all three function calls inside the main loop have to be finished before another object can be considered. Therefore an easy parallelization of the `mainLoop` is not possible. The *h*-means algorithm given in Figure 1 does not have this problems and is therefore much easier to parallelize. Every processor can, independently of all other processors, find the

```
function K-MeanMainLoop {                       function H-MeanMainLoop {
    assign each object randomly to one cluster;     assign each object randomly to one cluster;
    do {                                            do {
        for each object t in the database {             for each object t in the database {
            nC = getNearestMean(t);                         nC = getNearestMean(t);
            insertIntoCluster(t,nC);                        insertIntoCluster(t,nC);
            recalculateMeans(t,nC);                     }
        }                                               recalculateMeans();
    } while at least one t changes its cluster      } while at least one t changes its cluster
}                                               }
```

Fig. 1. Pseudo code of the k-means and h-means algorithm.

nearest clusters for all local objects. Once the cluster membership for every object is defined, the mean value and the membership values have to be globally update. This modified version has exactly the same properties as the sequential algorithm given in Figure 1.

4 Experiments

In order to test the performance we conducted a series of tests. The environment we used consists of a network of 32 PC connected through a 10 MBits Ethernet. This is a very simple environment, but fairly realistic for many of the application environments we are focusing on.

In order to get real timings we used some reference data sets from the *Machine Learning Database Repository at UC Irvine* as well as artificially generated ones. To present the timings here we use a database with the following characteristics: 20 continuous attributes, 100'000 objects in 20 clusters.

If we execute the program "as it is" on one PC the average execution time for the clustering is 528.7 seconds. This time can not be directly compared to timings for the parallel version running in an environment of two and more nodes. Every node could keep its part of the data in memory. In order to get a usable reference time we added memory to one PC to allow it to keep the whole database in memory. The execution time under these conditions goes down to 343.5 seconds. This time can now be used as reference time for the parallel timings.

In order to measure the execution time in a parallel/distributed setting, first we have to distribute the data to all the processors participating in the clustering. In the real world scenario this distribution would be given. Here we installed the data that was processed by each node on its local disk. This distribution is done by the underlying object store [8]. The time for this operation is not included in the timings presented here.

The first measures of interest are of course the overall execution times[1] of the application. For the one node configuration we measured the timings for the standard machine (528.7 secs) and the machine with the augmented memory (343.5). With an increasing number of nodes the speedup slowly degrades, but

[1] The timings are the mean values over 100 runs.

even with 32 nodes we still have 90% efficiency. This is reasonable with respect to the slow interconnection network we are using. The increase in time consumption we are measuring after adding nodes is not only related to the increase in communication overhead, but is also related to the variations in the execution times of the different processors. These variations are relatively high and are very hard to control largely because of OS limitations (Windows 95) that do not allow us to control many of the parameters we would like to.

The previously presented results are representative for all the other tests we conducted. Because of space constraints we did not add other similar results. Overall, the results are very satisfactory for the given environment.

5 Conclusion and Future Work

We have presented a parallel version of the k/h-means clustering algorithm. The algorithm is designed to be used on large distributed data sets. Even on a very simple distributed computing environment, namely a PC cluster on a 10 MBits Ethernet, we are able to achieve about 90% efficiency for a configuration up to 32 processors. These results show that parallel k/h-means is scalable and thus enlarges its field of application to clustering tasks where it would be the preferred algorithm, but the task's computational complexity previously made it impossible

The basic algorithm presented here can be used in conjunction with broad variety of other settings. We are currently using this algorithm in a system that handles a wide range of classification problems. The parallel algorithm presented was extended to handle, not only continuous, but also categorical data. We are currently working on a version that would allow us to combine both types of data.

References

[1] M.R. Anderberg. *Cluster Analysis for Applications.* Academic Press, 1973.

[2] John A. Hatigan. *Clustering Algorithms.* John Wiley and Sons, 1975.

[3] W. Kloesgen and J.M. Zytkow. Knowledge discovery in database terminology. *Advances in Knowledge Discovery and Data Mining*, pages 573–592, 1996.

[4] J.B. MacQueen. Some methods for classification and analysis of multivariate observations. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[5] C.F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21, 1995.

[6] E.M. Rasmussen and P. Willett. Efficiency of hierarchical agglomerative clustering using the icl distributed array oricessor. *Journal of Documentation*, 45(1), 1989.

[7] Helmuth Spaeth. *Cluster Analysis Algorithms.* John Wiley and Sons, 1980.

[8] Kilian Stoffel. Pattern matching in time series. Technical Report University of Neuchâtel, September 1998.