Scalable Parallel Clustering for Data Mining on Multicomputers

D. Foti, D. Lipari, C. Pizzuti and D. Talia

ISI-CNR
c/o DEIS, UNICAL
87036 Rende (CS), Italy
{pizzuti,talia}@si.deis.unical.it

Abstract. This paper describes the design and implementation on MIMD parallel machines of *P-AutoClass*, a parallel version of the *AutoClass* system based upon the Bayesian method for determining optimal classes in large datasets. The *P-AutoClass* implementation divides the clustering task among the processors of a multicomputer so that they work on their own partition and exchange their intermediate results. The system architecture, its implementation and experimental performance results on different processor numbers and dataset sizes are presented and discussed. In particular, efficiency and scalability of *P-AutoClass* versus the sequential *AutoClass* system are evaluated and compared.

1 Introduction

Clustering algorithms arranges data items into several groups so that similar items fall into the same group. This is done without any suggestion from an external supervisor, classes and training examples are not given a priori. Most of the early cluster analysis algorithms come from the area of statistics and have been originally designed for relatively small data sets. In the recent years, clustering algorithms have been extended to efficiently work for knowledge discovery in large databases and some of them are able to deal with high-dimensional feature items. When used to classify large data sets, clustering algorithms are very computing demanding and require high-performance machines to get results in reasonable time. Experiences of clustering algorithms taking from one week to about 20 days of computation time on sequential machines are not rare. Thus, scalable parallel computers can provide the appropriate setting where to execute clustering algorithms for extracting knowledge from large-scale data repositories.

Recently there has been an increasing interest in parallel implementations of data clustering algorithms. Parallel approaches to clustering can be found in [8, 4, 9, 5, 10]. In this paper we consider a parallel clustering algorithm based on Bayesian classification for distributed memory multicomputers. We propose a parallel implementation of the AutoClass algorithm, called *P-AutoClass*, and validate by experimental measurements the scalability of our parallelization strategy. The *P-AutoClass* algorithm divides the clustering task among the processors of a parallel

machine that work on their own partition and exchange their intermediate results. It is based on the message passing model for shared-nothing MIMD computers.

This paper describes the design and implementation of *P-AutoClass*. Furthermore, experimental performance results on different processor numbers and dataset sizes are presented and discussed. The rest of the paper is organized as follows. Section 2 provides a very short overview of Bayesian classification and sequential AutoClass. Section 3 describes the design and implementation of *P-Autoclass* for multicomputers. Section 4 presents the main experimental performance results of the algorithm. Section 5 describes related work and section 6 contains conclusions.

2 Bayesian Classification and AutoClass

The Bayesian approach to unsupervised classification provides a probabilistic approach to induction. Given a set $X = \{X_1, \ldots, X_I\}$ of data instances X_i, with unknown classes, the goal of Bayesian classification is to search for the best class description that predict the data. Instances X_i are represented as ordered vectors of attribute values $\overrightarrow{X_i} = \{X_{i1}, \ldots, Xik\}$.

In this approach class membership is expressed probabilistically, that is an instance is not assigned to a unique class, instead it has a probability of belonging to each of the possible classes. The classes provides probabilities for all attribute values of each instance. Class membership probabilities are then determined by combining all these probabilities. Class membership probabilities of each instance must sum to 1, thus there not precise boundaries for classes: every instance must be a member of some class, even though we do not know which one. When every instance has a probability of about 0.5 in any class, the classification is not well defined because it means that classes are abundantly overlapped. On the contrary, when the probability of each instance is about 0.99 in its most probable class, the classes are well separated.

A Bayesian classification model consists of two sets of parameters: a set of discrete parameters *T* which describes the functional form of the model, such as number of classes and whether attributes are correlated, and a set of continuos parameters $\vec{V}$ that specifies values for the variables appearing in *T*, needed to complete the general form of the model. The probability of observing an instance having particular attribute value vector is referred to as *probability distribution or density function* (p.d.f.). Given a set of data *X*, AutoClass searches for the most probable pair $\vec{V}$, *T* which classifies *X*. This is done in two steps:

- For a given *T*, AutoClass seeks the maximum posterior (MAP) parameter values $\vec{V}$.
- Regardless of any $\vec{V}$, AutoClass searches for the most probable *T*, from a set of possible *Ts* with different attribute dependencies and class structure.

Thus there are two levels of search: *parameter level search* and *model level search*. Fixed the number classes and their class model, the space of allowed parameter values is searched for finding the most probable $\vec{V}$. Given the parameter values, AutoClass

calculates the likelihood of each case belonging to each class *L* and then calculates a set of weights $w_{ij}=(L_i / \Sigma_j L_j)$ for each case. Given these weigths, weighted statistics relevant to each term of the class likelihood are calculated. These statistics are then used to generate new MAP values for the parameters and the cycle is repeated.

Based on this theory, Cheeseman and colleagues at NASA Ames Research Center developed AutoClass [1] originally in Lisp. Then the system has been ported from Lisp to C. The C version of AutoClass improved the performance of the system of about ten times and has provided a version of the system that can be easily accessed and used by researchers at a variety of universities and research laboratories.

3 P-AutoClass

In spite of the significant improvement of the C version performance, because of the computational needs of the algorithm, the execution of AutoClass with large datasets requires times that in many cases are very high. For instance, the sequential AutoClass runs on a dataset of 14K tuples, each one composed of a few hundreds bytes, have taken more the 3 hours on Pentium-based PC. Considering that the execution time increases linearly with the size of dataset, more than 1 day is necessary to analyze a dataset composed of about 140K tuples, that is not a very large dataset. For the clustering of a satellite image AutoClass took more than 130 hours [6] and the analysis of protein sequences the discovery process required from 300 to 400 hours [3].

These considerations and experiences suggest that it is necessary to implement faster versions of AutoClass to handle very large data set in reasonable time. This can be done by exploiting the inherent parallelism present in the AutoClass algorithm implementing it in parallel on MIMD multicomputers. Among the different parallelization strategies, we selected the SPMD approach. In AutoClass, the SPMD approach can be exploited by dividing up the dataset among the processors and by the parallel updating on different processors of the weights and parameters of classifications. This strategy does not require to replicate the entire dataset on each processor. Furthermore, it also does not have load balancing problems because each processor execute the same code on data of equal size. Finally, the amount of data exchanged among the processors is not so large since most operations are performed locally at each processor.

3.1 Design of the parallel algorithm

The main steps of the structure of the AutoClass program are described in figure 1. After program starting and structure initialization, the main part of the algorithm is devoted to classification generation and evaluation (*step 3*). This loop is composed of a set of substeps specified in figure 2. Among those substeps, the *new try of classification* step is the most computationally intensive. It computes the weights of each items for each class and computes the parameters of the classification. These operations are executed by the function **`base_cycle`** which calls the three functions **`update_wts`**, **`update_parameters`** and **`update_approximations`** as shown in figure 3.

1. Program Start and Files Reading ;
2. Data Structures Initialized ;
3. Classification Generation and Evaluation (called also *BIG_LOOP*) ;
4. Check the Stopping Conditions, if they are not verified go to step **3**, else go to step **5**;
5. Store Results on the Output Files.

Fig. 1. Scheme of the sequential AutoClass algorithm.

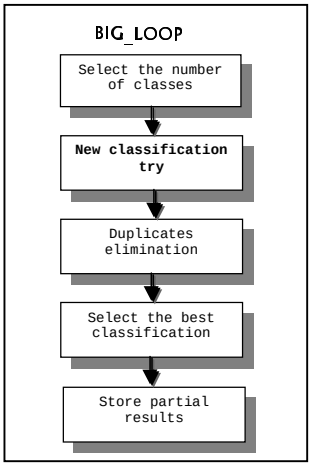

Fig. 2. Main steps of the *BIG_LOOP*.

We analyzed the time spent in the **`base_cycle`** function and it resulted about the 99,5% of the total time, therefore we identified this function as that one where parallelism must be exploited to speed up the AutoClass performance.

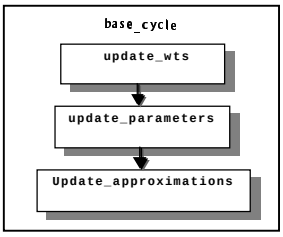

Fig. 3. The structure of the *base_cycle* function.

In particular, analyzing the time spent in each of the three functions called by **`base_cycle`**, it appears, as observed in other experiences [7], that the **`update_wts`** and **`update_parameters`** functions are the time consuming functions whereas the time spent in the **`update_approximation`** is negligible. Therefore, we studied the parallelization of these two functions using the SPMD approach. To maintain the same semantics of the sequential algorithm of AutoClass, we designed the parallel

version by partitioning data and local computation on each of **P** processors of a distributed memory MIMD computer and by exchanging among the processors all the local variables that contribute to form global values of a classification.

3.1.1 Parallel update_wts

In AutoClass the class membership of each data item is expressed probabilistically. Thus every item has a probability that it belongs to each of the possible classes. In the AutoClass algorithm, class membership is expressed by weights. The function **`update_wts`** calculates the weights w_{ij} for each item i of the active classes to be the normalized class probabilities with respect to the current parameterizations.

The parallel version of this function first calculates on each processing element the weights w_{ij} for each item belonging to the local partition of the data set and sum the weights w_j of each class j ($w_j=\Sigma_i\ w_{ij}$) relatively to its own data. Then all the partial w_j values are exchanged among all the processors and summed in each of them to have the same value in every processor. This strategy is described in figure 4.

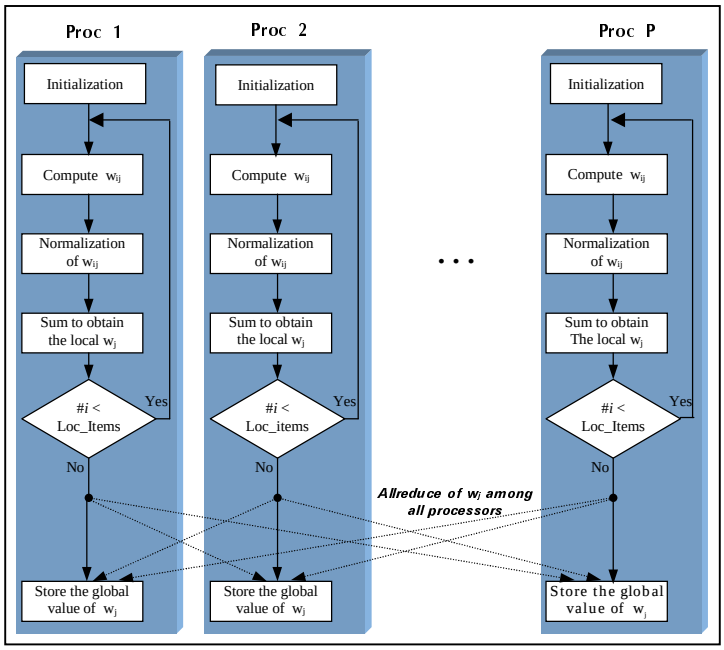

Fig. 4. The parallel version of the *update_wts* function.

To implement the total exchange of the w_j values in the **`update_wts`** function we used a *global reduction* operation (`Allreduce`) that sums all the local copies in the all processes (*reduction* operation) and places the results on all the processors (*broadcast* operation).

3.1.2 Parallel update_parameters

The **`update_parameters`** function computes for each class a set of class posterior parameter values, which specify how the class is distributed along the various

attributes. To do this, the function is composed of three nested loops, the external loop scans all the classes, then for each class all the attributes are analyzed and in the inner loop all the items are read and their values are used to compute the class parameters.

In parallelizing this function, we executed the partial computation of parameters in parallel on all the processors, then all the local values are collected on each processor before to utilize them for computing the global values of the classification parameters. Figure 5 shows the scheme of the parallel version of the function. To implement the total exchange of the parameter values in the **`update_parameters`** function we used a *global reduction* operation that sums all the local copies in all the processes and places the results on every processor.

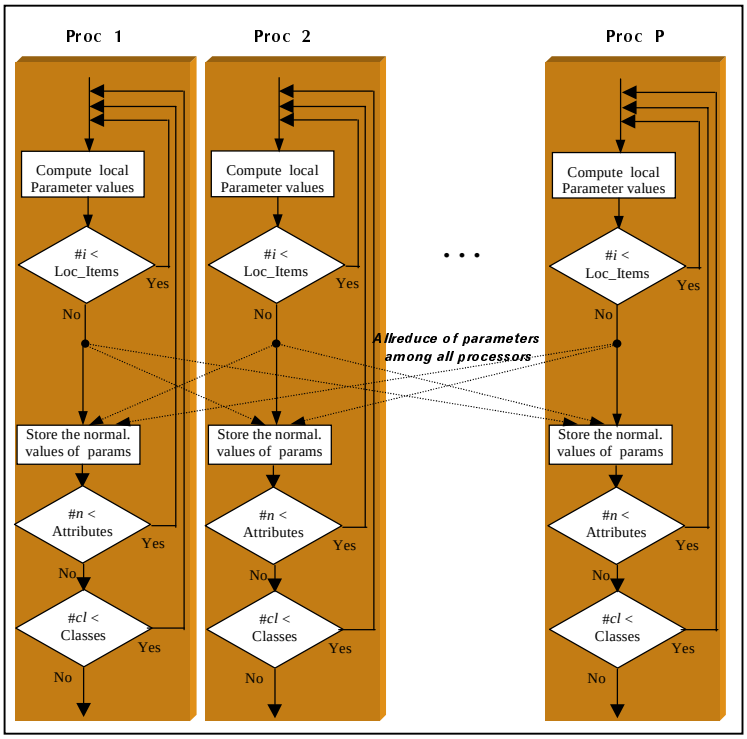

Fig. 5. The parallel version of the *update_parameters* function.

P-AutoClass has been implemented using the Message Passing Interface (MPI) toolkit on a Meiko Computing Surface 2 using the version 3.3 of sequential AutoClass C. Because of the large availability of MPI, *P-AutoClass* is portable practically on every parallel machine from supercomputers to PC clusters.

4 Experimental results

We run our experiments on a Meiko CS 2 with up to 10 SPARC processors connected by a fat tree topology with a communication bandwidth of 50 Mbytes/s in both

directions. We used a synthetic dataset composed of 100000 tuples each one composed of two real attributes. To perform our experiments we used different partitions of data from 5000 tuples to the entire dataset, and asked the system to find the best clustering starting with different number of clusters (*start_j_list=2, 4, 8, 16, 24, 50, 64*). Each classification has been repeated 10 times and results presented here represent the mean values obtained after these classifications. We measured the elapsed time, the speedup and the scaleup of P-AutoClass. *Speedup* gives the efficiency of the parallel algorithm when the number of processors varies. It is defined as the ratio of the execution time for clustering a dataset on *1* processor to the execution time for clustering the same dataset on *P* processors. Another interesting measure is scaleup. *Scaleup* captures how a parallel algorithm handles larger datasets when more processors are available.

Figure 6 shows the elapsed times of P-AutoClass on different numbers of processors. We can see that the total execution time substantially decreases as the number of used processors increases. In particular, for the largest datasets the time decreases in a more significant way. We can observe that as the dataset size increases the time gain increases as well.

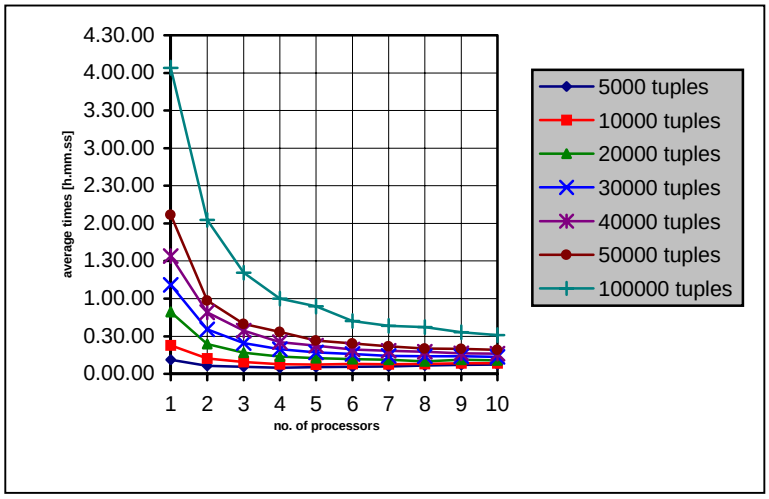

Fig. 6. Average elapsed times of *P-AutoClass* on different numbers of processors.

In figure 7 the speedup results obtained for different datasets are given. We can observe that the *P-AutoClass* algorithm scales well up to 10 processors for the largest datasets, whereas for small datasets the speedup increases until the optimal number of processors are used for the given problem (e.g., 4 procs for 5000 tuples or 8 procs for 20000 tuples). When more processes are used we observe that the algorithm does not scale because the processors are not effectively used and the communication costs increases.

For scaleup measures we evaluated the execution time of a single iteration of the *base_cycle* function by keeping the number of data items per processor fixed while increasing the number of processors. To obtain more stable results we asked *P-AutoClass* to group data into 8 and 16 clusters. Figure 8 shows the scaleup results. For all the experiments we have 10000 tuples per processor. We started with 10000 tuples

on 1 processor up to 100000 tuples on 10 processors. It can be seen that the parallel AutoClass algorithm, for a given classification, shows a nearly stable pattern. Thus it delivers nearly constant execution times in number of processors showing good scaleup.

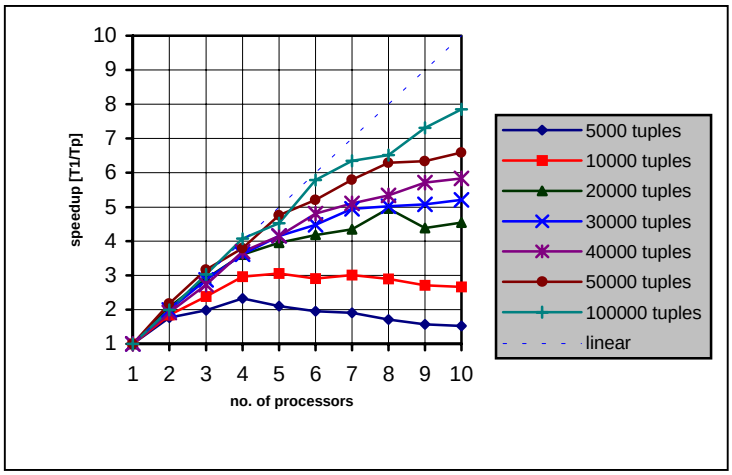

Fig. 7. Speedup of *P-AutoClass* on different numbers of processors.

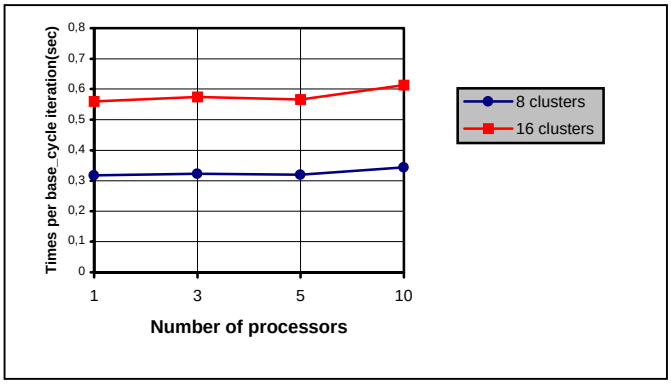

Fig. 8. Scaleup of the *base_cycle* of *P-AutoClass* on different sets of tuples scaled by the number of processors.

5 Related work

In the past few years there has been an increasing interest in parallel implementations of data mining algorithms [2]. The first approaches to the parallelization of AutoClass have been done on SIMD parallel machines by using compilers that automatically generated data-parallel code starting from the sequential program to which only few instructions have been added. The first data-parallel version of AutoClass has been developed in *Lisp to run on a Connection Machine-2 [1], and the second one has been developed adding C* code to the C source to run it on a CM-5 [9]. These

approaches are simple from the programming point-of-view but they do not exploit all the potential parallelism. In fact, it is not known how well compilers extract parallelism from sequential programs.

The only experience we know about the implementation of AutoClass on a MIMD computer is described in [7]. This prototype is based on the exploitation of parallelism in the `update_wts` function. Concerning to the SIMD approach, our implementation is more general and allows to exploit parallelism in a more complete, flexible, and portable way. On the other hand, considering the mentioned MIMD parallel implementation, *P-AutoClass* exploits parallelism also in the parameters computing phase, with a further improvement of performance.

6 Conclusion

In this paper we proposed *P-AutoClass*, a parallel implementation of the AutoClass algorithm based upon the Bayesian method for determining optimal classes in large datasets. We have described and evaluated the *P-AutoClass* algorithm on a MIMD parallel computer. The experimental results show that *P-AutoClass* is scalable both in terms of speedup and scaleup. This means that for a given dataset, the execution times can be reduced as the number of processors increases, and the execution times do not increase if, while increasing the size of datasets, more processors are available. Finally, our algorithm is easily portable to various MIMD distributed-memory parallel computers that are now currently available from a large number of vendors. It allows to perform efficient clustering on very large datasets significantly reducing the computation times on several parallel computing platforms.

References

1. P. Cheeseman and J. Stutz. Bayesian Classification (AutoClass): Theory and Results. In *Advances in Knowledge Discovery and Data Mining*, AAAI Press/MIT Press, pp. 61-83, 1996.
2. A.A. Freitas and S.H. Lavington. *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers, 1998.
3. L. Hunter and D.J. States. Bayesian Classification of Protein Structure. *IEEE Expert*, 7(4):67-75, 1992.
4. D. Judd, P. McKinley, and A. Jain. Large-Scale Parallel Data Clustering. *Proceedings of the Int. Conf. on Pattern Recognition*, 1996.
5. D. Judd, P. McKinley, A. Jain. Performance Evaluation on Large-Scale Parallel Clustering in NOW Environments. *Proceedings of the Eight SIAM Conf. on Parallel Processing for Scientific Computing*, Minneapolis, March 1997.
6. B. Kanefsky, J. Stutz, P. Cheeseman, and W. Taylor. An Improved Automatic Classification of a Landsat/TM Image from Kansas (FIFE). Technical Report FIA-94-01, NASA Ames Research Center, May 1994.
7. R. Miller and Y. Guo. Parallelisation of AutoClass. *Proceedings of the Parallel Computing Workshop (PCW'97)*, Canberra, Australia, 1997.
8. C.F. Olson. Parallel Algorithms for Hierarchical Clustering. *Parallel Computing*, 21:1313-1325, 1995.
9. J.T. Potts. Seeking Parallelism in Discovery Programs. Master Thesis, University of Texas at Arlington, 1996.
10. K. Stoffel and A. Belkoniene. Parallel K-Means Clustering for Large Data Sets. *Proceedings Euro-Par '99*, LNCS 1685, pp. 1451-1454, 1999.