High Performance Data Mining

Scaling Algorithms, Applications and Systems

Edited by

Yike Guo

and

Robert Grossman

Kluwer Academic Publishers

HIGH PERFORMANCE DATA MINING

Scaling Algorithms, Applications and Systems

This page intentionally left blank.

HIGH PERFORMANCE DATA MINING

Scaling Algorithms, Applications and Systems

edited by

Yike Guo
Imperial College, United Kingdom

Robert Grossman
University ofIllinois at Chicago

A Special Issue of
DATA MINING AND KNOWLEDGE DISCOVERY
Volume 3, No. 03 (1999)

KLUWER ACADEMIC PUBLISHERS
New York / Boston / Dordrecht / London / Moscow

eBook ISBN: 0-306-47011-X
Print ISBN: 0-7923-7745-1

Created in the United States of America

Visit Kluwer Online at: http://www.kluweronline.com
and Kluwer's eBookstore at: http://www.ebooks.kluweronline.com

DATA MINING AND KNOWLEDGE DISCOVERY

Volume 3, No. 3, September 1999

Special issue on Scaling Data Mining Algorithms, Applications, and Systems to Massive Data Sets by Applying High Performance Computing Technology
Guest Editors: Yike Guo, Robert Grossman

Full Paper Contributors

Data Mining and Knowledge Discovery, 3, 235–236 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

Editorial

YIKE GUO yg@doc.ic.ac.uk
Department of Computing, Imperial College, University of London, UK

ROBERT GROSSMAN grossman@uic.edu
Magnify, Inc. & National Center for Data Mining, University of Illinois at Chicago, USA

> His promises were, as he then was, mighty;
> But his performance, as he is now, nothing.
> —Shakespeare, King Henry VIII

This special issue of Data Mining and Knowledge Discovery addresses the issue of scaling data mining algorithms, applications and systems to massive data sets by applying high performance computing technology. With the commoditization of high performance computing using clusters of workstations and related technologies, it is becoming more and more common to have the necessary infrastructure for high performance data mining. On the other hand, many of the commonly used data mining algorithms do not scale to large data sets. Two fundamental challenges are: to develop scalable versions of the commonly used data mining algorithms and to develop new algorithms for mining very large data sets. In other words, today it is easy to spin a terabyte of disk, but difficult to analyze and mine a terabyte of data.

Developing algorithms which scale takes time. As an example, consider the successful scale up and parallelization of linear algebra algorithms during the past two decades. This success was due to several factors, including: a) developing versions of some standard algorithms which exploit the specialized structure of some linear systems, such as block-structured systems, symmetric systems, or Toeplitz systems; b) developing new algorithms such as the Wierderman and Lancos algorithms for solving sparse systems; and c) developing software tools providing high performance implementations of linear algebra primitives, such as Linpack, LA Pack, and PVM.

In some sense, the state of the art for scalable and high performance algorithms for data mining is in the same position that linear algebra was in two decades ago. We suspect that strategies a)–c) will work in data mining also.

High performance data mining is still a very new subject with challenges. Roughly speaking, some data mining algorithms can be characterised as a heuristic search process involving many scans of the data. Thus, irregularity in computation, large numbers of data access, and non-deterministic search strategies make efficient parallelization of a data mining algorithms a difficult task. Research in this area will not only contribute to large scale data mining applications but also enrich high performance computing technology itself. This was part of the motivation for this special issue.

This issue contains four papers. They cover important classes of data mining algorithms: classification, clustering, association rule discovery, and learning Bayesian networks. The paper by Srivastava et al. presents a detailed analysis of the parallelization strategy of tree induction algorithms. The paper by Xu et al. presents a parallel clustering algorithm for distributed memory machines. In their paper, Cheung et al. presents a new scalable algorithm for association rule discovery and a survey of other strategies. In the last paper of this issue, Xiang et al. describe an algorithm for parallel learning of Bayesian networks.

All the papers included in this issue were selected through a rigorous refereeing process. We thank all the contributors and referees for their support. We enjoyed editing this issue and hope very much that you enjoy reading it.

Yike Guo is on the faculty of Imperial College, University of London, where he is the Technical Director of Imperial College Parallel Computing Centre. He is also the leader of the data mining group in the centre. He has been working on distributed data mining algorithms and systems development. He is also working on network infrastructure for global wide data mining applications. He has a B.Sc. in Computer Science from Tsinghua University, China and a Ph.D. in Computer Science from University of London.

Robert Grossman is the President of Magnify, Inc. and on the faculty of the University of Illinois at Chicago, where he is the Director of the Laboratory for Advanced Computing and the National Center for Data Mining. He has been active in the development of high performance and wide area data mining systems for over ten years. More recently, he has worked on standards and testbeds for data mining. He has an AB in Mathematics from Harvard University and a Ph.D. in Mathematics from Princeton University.

Data Mining and Knowledge Discovery, 3,237–261 (1999)

Parallel Formulations of Decision-Tree Classification Algorithms

ANURAG SRIVASTAVA anurag@digital-impact.com
Digital Impact

EUI-HONG HAN han@cs.umn.edu
VIPIN KUMAR kumar@cs.umn.edu
Department of Computer Science & Engineering, Army HPC Research Center; University of Minnesota

VINEET SINGH vsingh@hitachi.com
Information Technology Lab, Hitachi America, Ltd.

Editors: Yike Guo and Robert Grossman

Abstract. Classification decision tree algorithms are used extensively for data mining in many domains such as retail target marketing, fraud detection, etc. Highly parallel algorithms for constructing classification decision trees are desirable for dealing with large data sets in reasonable amount of time. Algorithms for building classification decision trees have a natural concurrency, but are difficult to parallelize due to the inherent dynamic nature of the computation. In this paper, we present parallel formulations of classification decision tree learning algorithm based on induction. We describe two basic parallel formulations. One is based on *Synchronous Tree Construction Approach* and the other is based on *Partitioned Tree Construction Approach.* We discuss the advantages and disadvantages of using these methods and propose a hybrid method that employs the good features of these methods. We also provide the analysis of the cost of computation and communication of the proposed hybrid method. Moreover, experimental results on an IBM SP-2 demonstrate excellent speedups and scalability.

Keywords: data mining, parallel processing, classification, scalability, decision trees

1. Introduction

Classification is an important data mining problem. A classification problem has an input dataset called the training set which consists of a number of examples each having a number of attributes. The attributes are either *continuous,* when the attribute values are ordered, or *categorical,* when the attribute values are unordered. One of the categorical attributes is called the *class label* or the *classifying attribute.* The objective is to use the training dataset to build a model of the class label based on the other attributes such that the model can be used to classify new data not from the training dataset. Application domains include retail target marketing, fraud detection, and design of telecommunication service plans. Several classification models like neural networks (Lippman, 1987), genetic algorithms (Goldberg, 1989), and decision trees (Quinlan, 1993) have been proposed. Decision trees are probably the most popular since they obtain reasonable accuracy (Spiegelhalter et al., 1994) and they are relatively inexpensive to compute. Most current classification algorithms such as *C4.5* (Quinlan, 1993), and *SLIQ* (Mehta et al., 1996) are based on the *ID3* classification decision tree algorithm (Quinlan, 1993).

In the data mining domain, the data to be processed tends to be very large. Hence, it is highly desirable to design computationally efficient as well as scalable algorithms. One way to reduce the computational complexity of building a decision tree classifier using large training datasets is to use only a small sample of the training data. Such methods do not yield the same classification accuracy as a decision tree classifier that uses the entire data set [Wirth and Catlett, 1988; Catlett, 1991; Chan and Stolfo, 1993a; Chan and Stolfo, 1993b]. In order to get reasonable accuracy in a reasonable amount of time, parallel algorithms may be required.

Classification decision tree construction algorithms have natural concurrency, as once a node is generated, all of its children in the classification tree can be generated concurrently. Furthermore, the computation for generating successors of a classification tree node can also be decomposed by performing data decomposition on the training data. Nevertheless, parallelization of the algorithms for construction the classification tree is challenging for the following reasons. First, the shape of the tree is highly irregular and is determined only at runtime. Furthermore, the amount of work associated with each node also varies, and is data dependent. Hence any static allocation scheme is likely to suffer from major load imbalance. Second, even though the successors of a node can be processed concurrently, they all use the training data associated with the parent node. If this data is dynamically partitioned and allocated to different processors that perform computation for different nodes, then there is a high cost for data movements. If the data is not partitioned appropriately, then performance can be bad due to the loss of locality.

In this paper, we present parallel formulations of classification decision tree learning algorithm based on induction. We describe two basic parallel formulations. One is based on *Synchronous Tree Construction Approach* and the other is based on *Partitioned Tree Construction Approach.* We discuss the advantages and disadvantages of using these methods and propose a hybrid method that employs the good features of these methods. We also provide the analysis of the cost of computation and communication of the proposed hybrid method, Moreover, experimental results on an IBM SP-2 demonstrate excellent speedups and scalability.

2. Related work

2.1. Sequential decision-tree classification algorithms

Most of the existing induction-based algorithms like *C4.5* (Quinlan, 1993), *CDP* (Agrawal et al., 1993), *SLIQ* (Mehta et al., 1996), and *SPRINT* (Shafer et al., 1996) use Hunt's method (Quinlan, 1993) as the basic algorithm. Here is a recursive description of Hunt's method for constructing a decision tree from a set T of training cases with classes denoted $\{C_1, C_2, \ldots, C_k\}$.

Case I. T contains cases all belonging to a single class C_j. The decision tree for T is a leaf identifying class C_j.

Case 2. T contains cases that belong to a mixture of classes. A test is chosen, based on a single attribute, that has one or more mutually exclusive outcomes $\{O_1, O_2, \ldots, O_n\}$. Note that in many implementations, n is chosen to be 2 and this leads to a binary decision tree. T is partitioned into subsets $T_1, T_2, \ldots, T_n$, where T_i contains all the cases in T that have outcome O_i of the chosen test. The decision tree for T consists of a decision node identifying the test, and one branch for each possible outcome. The same tree building machinery is applied recursively to each subset of training cases.

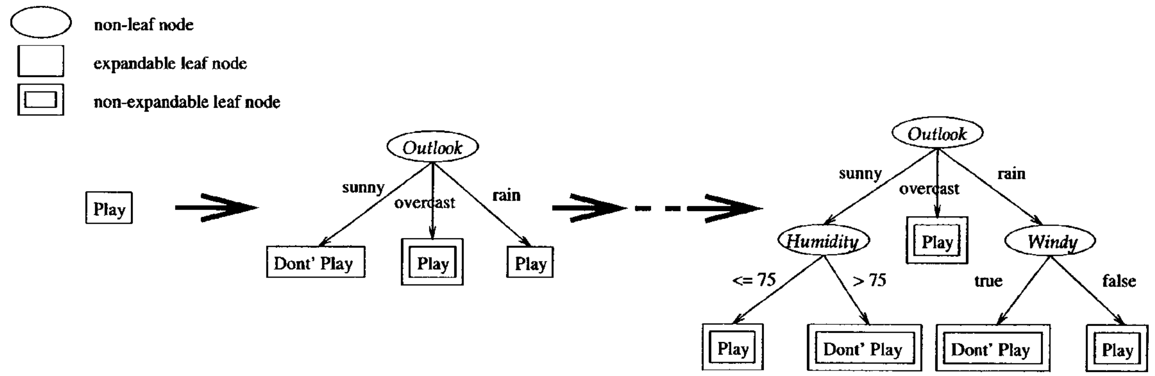
Case 3. T contains no cases. The decision tree for T is a leaf, but the class to be associated with the leaf must be determined from information other than T. For example, *C4.5* chooses this to be the most frequent class at the parent of this node.

Table 1 shows a training data set with four data attributes and two classes. Figure 1 shows how Hunt's method works with the training data set. In case 2 of Hunt's method, a test based on a single attribute is chosen for expanding the current node. The choice of an attribute is normally based on the entropy gains of the attributes. The entropy of an attribute is calculated from class distribution information. For a discrete attribute, class distribution information of each value of the attribute is required. Table 2 shows the class distribution information of data attribute *Outlook* at the root of the decision tree shown in figure 1. For a continuous attribute, binary tests involving all the distinct values of the attribute are considered. Table 3 shows the class distribution information of data attribute *Humidity*. Once the class distribution information of all the attributes are gathered, each attribute is evaluated in terms of either *entropy* (Quinlan, 1993) or *Gini Index* (Breiman et al., 1984). The best attribute is selected as a test for the node expansion.

The *C4.5* algorithm generates a classification—decision tree for the given training data set by recursively partitioning the data. The decision tree is grown using depth—first strategy.

Table1. A small training data set [Qui93].

Outlook	Temp (F)	Humidity (%)	Windy?	Class
Sunny	75	70	True	Play
Sunny	80	90	True	Don't play
Sunny	85	85	False	Don't play
Sunny	72	95	False	Don't play
Sunny	69	70	False	Play
Overcast	72	90	True	Play
Overcast	83	78	False	Play
Overcast	64	65	True	Play
Overcast	81	75	False	Play
Rain	71	80	True	Don't play
Rain	65	70	True	Do'nt play
Rain	75	80	Flase	Play
Rain	68	80	False	Play
Rain	70	96	False	Play



(a) Initial Classification Tree (b) Intermediate Classification Tree (c) Final Classification Tree

Figure 1. Demonstration of Hunt's method.

Table 2. Class distribution information of attribute *Outlook*.

Attribute value	Class	
	Play	Don't play
Sunny	2	3
Overcast	4	0
Rain	3	2

Table 3. Class distribution information of attribute *Humidity*.

Attribute value	Binary test	Class	
		Play	Don't play
65	≤	1	0
	>	8	5
70	≤	3	1
	>	6	4
75	≤	4	1
	>	5	4
78	≤	5	1
	>	4	4
80	≤	7	2
	>	2	3
85	≤	7	3
	>	2	2
90	≤	8	4
	>	1	1
95	≤	8	5
	>	1	0
96	≤	9	5
	>	0	0

The algorithm considers all the possible tests that can split the data set and selects a test that gives the best information gain. For each discrete attribute, one test with outcomes as many as the number of distinct values of the attribute is considered. For each continuous attribute, binary tests involving every distinct value of the attribute are considered. In order to gather the entropy gain of all these binary tests efficiently, the training data set belonging to the node in consideration is sorted for the values of the continuous attribute and the entropy gains of the binary cut based on each distinct values are calculated in one scan of the sorted data. This process is repeated for each continuous attribute.

Recently proposed classification algorithms *SLIQ* (Mehta et al., 1996) and *SPRINT* (Shafer et al., 1996) avoid costly sorting at each node by pre-sorting continuous attributes once in the beginning. In *SPRINT,* each continuous attribute is maintained in a sorted attribute list. In this list, each entry contains a value of the attribute and its corresponding record id. Once the best attribute to split a node in a classification tree is determined, each attribute list has to be split according to the split decision. A hash table, of the same order as the number of training cases, has the mapping between record ids and where each record belongs according to the split decision. Each entry in the attribute list is moved to a classification tree node according to the information retrieved by probing the hash table. The sorted order is maintained as the entries are moved in pre-sorted order.

Decision trees are usually built in two steps. First, an initial tree is built till the leaf nodes belong to a single class only. Second, pruning is done to remove any *overfitting* to the training data. Typically, the time spent on pruning for a large dataset is a small fraction, less than 1% of the initial tree generation. Therefore, in this paper, we focus on the initial tree generation only and not on the pruning part of the computation.

2.2. Parallel decision-tree classification algorithms

Several parallel formulations of classification rule learning have been proposed recently. Pearson presented an approach that combines node-based decomposition and attribute-based decomposition (Pearson, 1994). It is shown that the node-based decomposition (task parallelism) alone has several probelms. One problem is that only a few processors are utilized in the beginning due to the small number of expanded tree nodes. Another problem is that many processors become idle in the later stage due to the load imbalance. The attribute-based decomposition is used to remedy the first problem. When the number of expanded nodes is smaller than the available number of processors, multiple processors are assigned to a node and attributes are distributed among these processors. This approach is related in nature to the partitioned tree construction approach discussed in this paper. In the partitioned tree construction approach, actual data samples are partitioned (horizontal partitioning) whereas in this approach attributes are partitioned (vertical partitioning).

In (Chattratichat et al., 1997), a few general approaches for parallelizing C4.5 are discussed. In the Dynamic Task Distribution (DTD) scheme, a master processor allocatesa subtree of the decision tree to an idle slave processor. This schemedoes not require communication among processors, but suffers from the load imbalance. DTD becomes similar to the partitioned tree construction approach discussed in this paper once the number of available nodes in the decision tree exceeds the number of processors. The DP-rec scheme distributes the data set evenly and builds decision tree one node at a time. This scheme is identical to the synchronous tree construction approach discussed in this paper and suffers from the high communication overhead. The DP-att scheme distributes the attributes. This scheme has the advantages of being both load-balanced and requiring minimal communications. However, this scheme does not scale well with increasing number of processors. The results in (Chattratichat, 1997) show that the effectiveness of different parallelization schemes varies significantly with data sets being used.

Kufrin proposed an approach called Parallel Decision Trees (PDT) in (Kufrin, 1997). This approach is similar to the DP-rec scheme (Chattratichat et al., 1997) and synchronous tree construction approach discussed in this paper, as the data sets are partitioned among processors. The PDT approach designate one processor as the "host" processor and the remaining processors as "worker" processors. The host processor does not have any data sets, but only receives frequency statistics or gain calculations from the worker processors. The host processor determines the split based on the collected statistics and notify the split decision to the worker processors. The worker processors collect the statistics of local data following the instruction from the host processor. The PDT approach suffers from the high communication overhead, just like DP-rec scheme and synchronous tree construction approach. The PDT approach has an additional communication bottleneck, as every worker processor sends the collected statistics to the host processor at the roughly same time and the host processor sends out the split decision to all working processors at the same time.

The parallel implementation of SPRINT (Shafer et al., 1996) and ScalParC (Joshi et al., 1998) use methods for partitioning work that is identical to the one used in the synchronous tree construction approach discussed in this paper. Serial SPRINT (Shafer et al., 1996) sorts the continuous attributes only once in the beginning and keeps a separate attribute list with record identifiers. The splitting phase of a decision tree node maintains this sorted order without requiring to sort the records again. In order to split the attribute lists according to the splitting decision, SPRINT creates a hash table that records a mapping between a record identifier and the node to which it goes to based on the splitting decision. In the parallel implementation of SPRINT, the attribute lists are split evenly among processors and the split point for a node in the decision tree is found in parallel. However, in order to split the attribute lists, the full size hash table is required on all the processors. In order to construct the hash table, all-to-all broadcast (Kumar et al., 1994) is performed, that makes this algorithm unscalable with respect to runtime and memory requirements. The reason is that each processor requires $O(N)$ memory to store the hash table and $O(N)$ communication overhead for all-to-all broadcast, where N is the number of records in the data set. The recently proposed ScalParC (Joshi, 1998) improves upon the SPRINT by employing a distributed hash table to efficiently implement the splitting phase of the SPRINT. In ScalParC, the hash table is split among the processors, and an efficient personalized communication is used to update the hash table, making it scalable with respect to memory and runtime requirements.

Goil et al. (1996) proposed the Concatenated Parallelism strategy for efficient parallel solution of divide and conquer problems. In this strategy, the mix of data parallelism and task parallelism is used as a solution to the parallel divide and conquer algorithm. Data parallelism is used until there are enough subtasks are genearted, and then task parallelism is used, i.e., each processor works on independent subtasks. This strategy is similar in principle to the partitioned tree construction approach discussed in this paper. The Concatenated Parallelism strategy is useful for problems where the workload can be determined based on the size of subtasks when the task parallelism is employed. However, in the problem of classificatoin decision tree, the workload cannot be determined based on the size of data at a particular node of the tree. Hence, one time load balancing used in this strategy is not well suited for this particular divide and conquer problem.

3. Parallel formulations

In this section, we give two basic parallel formulations for the classification decision tree construction and a hybrid scheme that combines good features of both of these approaches. We focus our presentation for discrete attributes only. The handling of continuous attributes is discussed in Section 3.4. In all parallel formulations, we assume that N training cases are randomly distributed to P processors initially such that each processor has N/P cases.

3.1. Synchronous tree construction approach

In this approach, all processors construct a decision tree synchronously by sending and receiving class distribution information of local data. Major steps for the approach are shown below:

1. Select a node to expand according to a decision tree expansion strategy (e.g. Depth-First or Breadth-First), and call that node as the current node. At the beginning, root node is selected as the current node.
2. For each data attribute, collect class distribution information of the local data at the current node.
3. Exchange the local class distribution information using global reduction (Kumar et al., 1994) among processors.
4. Simultaneously compute the entropy gains of each attribute at each processor and select the best attribute for child node expansion.
5. Depending on the branching factor of the tree desired, create child nodes for the same number of partitions of attribute values, and split training cases accordingly.
6. Repeat above steps (1–5) until no more nodes are available for the expansion.

Figure 2 shows the overall picture. The root node has already been expanded and the current node is the leftmost child of the root (as shown in the top part of the figure). All the four processors cooperate to expand this node to have two child nodes. Next, the leftmost node of these child nodes is selected as the current node (in the bottom of the figure) and all four processors again cooperate to expand the node.

The advantage of this approach is that it does not require any movement of the training data items. However, this algorithm suffers from high communication cost and load imbalance. For each node in the decision tree, after collecting the class distribution information, all the processors need to synchronize and exchange the distribution information. At the nodes of shallow depth, the communication overhead is relatively small, because the number of training data items to be processed is relatively large. But as the decision tree grows and deepens, the number of training set items at the nodes decreases and as a consequence, the computation of the class distribution information for each of the nodes decreases. If the average branching factor of the decision tree is k, then the number of data items in a child node is on the average $\frac{1}{k}$th of the number of data items in the parent. However, the size of communication does not decrease as much, as the number of attributes to be considered goes down only by one. Hence, as the tree deepens, the communication overhead dominates the overall processing time.

The other problem is due to load imbalance. Even though each processor started out with the same number of the training data items, the number of items belonging to the same node of the decision tree can vary substantially among processors. For example, processor 1 might have all the data items on leaf node A and none on leaf node B, while processor 2

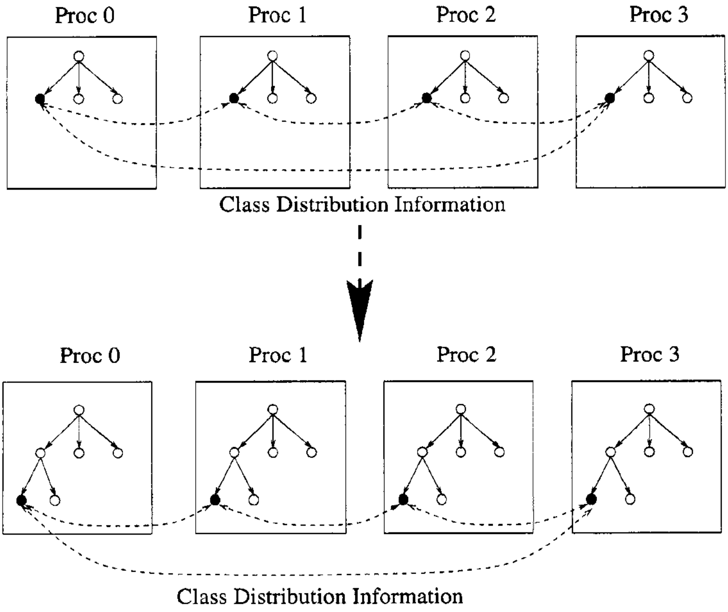

Proc 0
Proc 1
Proc 2
Proc 3
Class Distribution Information
Proc 0
Proc 1
Proc 2
Proc 3
Class Distribution Information

Figure 1. Synchronous tree construction approach with depth—first expansion strategy.

might have all the data items on node B and none on node A. When node A is selected as the current node, processor 2 does not have any work to do and similarly when node B is selected as the current node, processor 1 has no work to do.

This load imbalance can be reduced if all the nodes on the frontier are expanded simultaneously, i.e. one pass of all the data at each processor is used to compute the class distribution information for all nodes on the frontier. Note that this improvement also reduces the number of times communications are done and reduces the message start-up overhead, but it does not reduce the overall volume of communications.

In the rest of the paper, we will assume that in the synchronous tree construction algorithm, the classification tree is expanded breadth-first manner and all the nodes at a level will be processed at the same time.

3.2. Partitioned tree construction approach

In this approach, whenever feasible, different processors work on different parts of the classification tree. In particular, if more than one processors cooperate to expand a node, then these processors are partitioned to expand the successors of this node. Consider the case in which a group of processors P_n, cooperate to expand node *n*. The algorithm consists of following steps:

Step 1. Processors in P_n cooperate to expand node n using the method described in Section 3.1.

Step 2. Once the node n is expanded in to successor nodes, $n_1, n_2, \ldots, n_k$, then the processor group P_n, is also partitioned, and the successor nodes are assigned to processors as follows:

Case 1. If the number of successor nodes is greater than $|P_n|$,

1. Partition the successor nodes into $|P_n|$ groups such that the total number of training cases corresponding to each node group is roughly equal. Assign each processor to one node group.
2. Shuffle the training data such that each processor has data items that belong to the nodes it is responsible for.
3. Now the expansion of the subtrees rooted at a node group proceeds completely independently at each processor as in the serial algorithm.

Case 2. Otherwise (if the number of successor nodes is less than $|P_n|$),

1, Assign a subset of processors to each node such that number of processors assigned to a node is proportional to the number of the training cases corresponding to the node.
2. Shuffle the training cases such that each subset of processors has training cases that belong to the nodes it is responsible for.
3. Processor subsets assigned to different nodes develop subtrees independently. Processor subsets that contain only one processor use the sequential algorithm to expand the part of the classification tree rooted at the node assigned to them. Processor subsets that contain more than one processor proceed by following the above steps recursively.

At the beginning, all processors work together to expand the root node of the classification tree. At the end, the whole classification tree is constructed by combining subtrees of each processor.

Figure 3 shows an example. First (at the top of the figure), all four processors cooperate to expand the root node just like they do in the synchronous tree construction approach. Next (in the middle of the figure), the set of four processors is partitioned in three parts. The leftmost child is assigned to processors 0 and 1, while the other nodes are assigned to processors 2 and 3, respectively. Now these sets of processors proceed independently to expand these assigned nodes. In particular, processors 2 and processor 3 proceed to expand their part of the tree using the serial algorithm. The group containing processors 0 and 1 splits the leftmost child node into three nodes. These three new nodes are partitioned in two parts (shown in the bottom of the figure); the leftmost node is assigned to processor 0, while the other two are assigned to processor 1. From now on, processors 0 and 1 also independently work on their respective subtrees.

The advantage of this approach is that once a processor becomes solely responsible for a node, it can develop a subtree of the classification tree independently without any communication overhead. However, there are a number of disadvantages of this approach. The

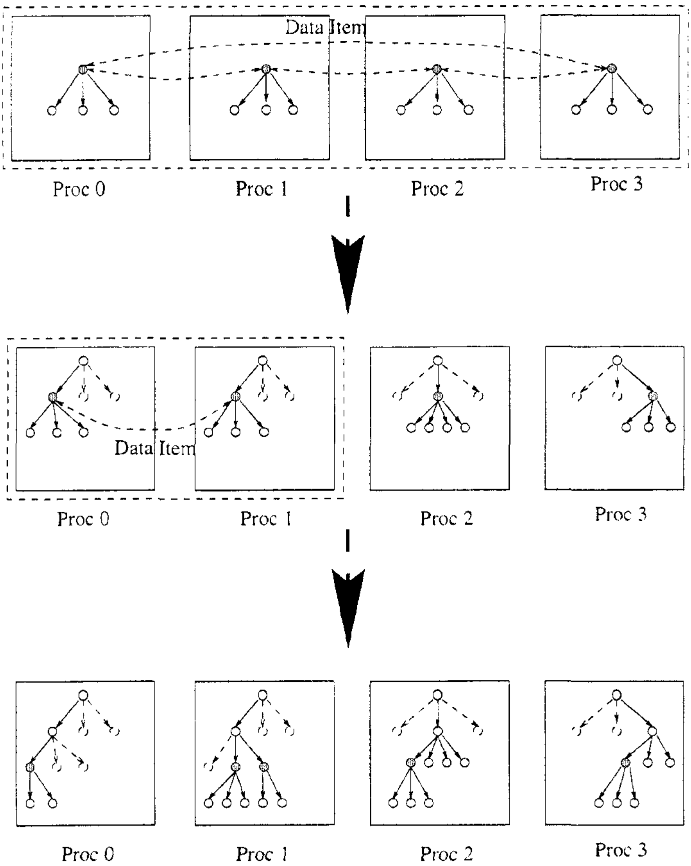


Figure 2. Partitioned tree construction approach

first disadvantage is that it requires data movement after each node expansion until one processor becomes responsible for an entire subtree. The communication cost is particularly expensive in the expansion of the upper part of the classification tree. (Note that once the number of nodes in the frontier exceeds the number of processors, then the communication cost becomes zero.) The second disadvantage is poor load balancing inherent in the algorithm. Assignment of nodes to processors is done based on the number of training cases in the successor nodes. However, the number of training cases associated with a node does not necessarily correspond to the amount of work needed to process the subtree rooted at the node. For example, if all training cases associated with a node happen to have the same class label, then no further expansion is needed.

3.3. Hybrid parallel formulation

Our hybrid parallel formulation has elements of both schemes. The *Synchronous Tree Construction Approach* in Section 3.1 incurs high communication overhead as the frontier gets larger. The *Partitioned Tree Construction Approach* of Section 3.2 incurs cost of load balancing after each step. The hybrid scheme keeps continuing with the first approach as long as the communication cost incurred by the first formulation is not too high. Once this cost becomes high, the processors as well as the current frontier of the classification tree are partitioned into two parts.

Our description assumes that the number of processors is a power of 2, and that these processors are connected in a hypercube configuration. The algorithm can be appropriately modified if P is not a power of 2. Also this algorithm can be mapped on to any parallel architecture by simply embedding a virtual hypercube in the architecture. More precisely the hybrid formulation works as follows.

- The database of training cases is split equally among P processors. Thus, if N is the total number of training cases, each processor has N/P training cases locally. At the beginning, all processors are assigned to one partition. The root node of the classification tree is allocated to the partition.
- All the nodes at the frontier of the tree that belong to one partition are processed together using the synchronous tree construction approach of Section 3.1.
- As the depth of the tree within a partition increases, the volume of statistics gathered at each level also increases as discussed in Section 3.1. At some point, a level is reached when communication cost become prohibitive. At this point, the processors in the partition are divided into two partitions, and the current set of frontier nodes are split and allocated to these partitions in such a way that the number of training cases in each partition is roughly equal. This load balancing is done as described as follows:

> On a hypercube, each of the two partitions naturally correspond to a sub-cube. First, corresponding processors within the two sub-cubes exchange relevant training cases to be transferred to the other sub-cube. After this exchange, processors within each sub-cube collectively have all the training cases for their partition, but the number of training cases at each processor can vary between 0 to $\frac{2*N}{P}$. Now, a load balancing step is done within each sub-cube so that each processor has an equal number of data items.

- Now, further processing within each partition proceeds asynchronously. The above steps are now repeated in each one of these partitions for the particular subtrees. This process is repeated until a complete classification tree is grown.
- If a group of processors in a partition become idle, then this partition joins up with any other partition that has work and has the same number of processors. This can be done by simply giving half of the training cases located at each processor in the donor partition to a processor in the receiving partition.

A key element of the algorithm is the criterion that triggers the partitioning of the current set of processors (and the corresponding frontier of the classification tree). If partitioning is done too frequently, then the hybrid scheme will approximate the partitioned tree construction approach, and thus will incur too much data movement cost. If the partitioning is done too late, then it will suffer from high cost for communicating statistics generated for each node of the frontier, like the synchronized tree construction approach. One possibility is to do splitting when the accumulated cost of communication becomes equal to the cost of moving records around in the splitting phase. More precisely, splitting is done when

$$\sum(\text{Communication Cost}) \geq \text{Moving Cost} + \text{Load Balancing}$$

As an example of the hybrid algorithm, figure 4 shows a classification tree frontier at depth 3. So far, no partitioning has been done and all processors are working cooperatively on each node of the frontier. At the next frontier at depth 4, partitioning is triggered, and the nodes and processors are partitioned into two partitions as shown in figure 5.

A detailed analysis of the hybrid algorithm is presented in Section 4.

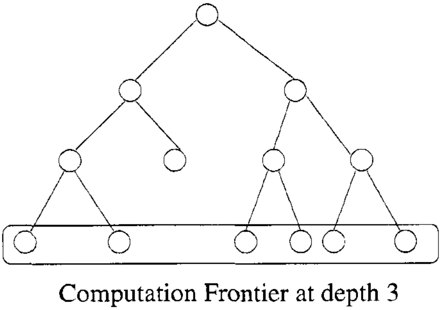


Figure 3. The computation frontier during computation phase.

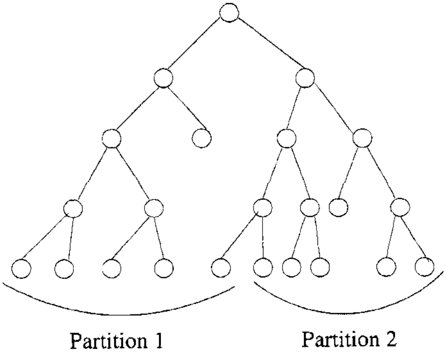


Figure 4. Binary partitioning of the tree to reduce communication costs.

3.4. Handling continuous attributes

Note that handling continuous attributes requires sorting. If each processor contains *N/P* training cases, then one approach for handling continuous attributes is to perform a parallel sorting step for each such attribute at each node of the decision tree being constructed. Once this parallel sorting is completed, each processor can compute the best local value for the split, and then a simple global communication among all processors can determine the globally best splitting value. However, the step of parallel sorting would require substantial data exchange among processors. The exchange of this information is of similar nature as the exchange of class distribution information, except that it is of much higher volume. Hence even in this case, it will be useful to use a scheme similar to the hybrid approach discussed in Section 3.3.

A more efficient way of handling continuous attributes without incurring the high cost of repeated sorting is to use the pre-sorting technique used in algorithms *SLIQ* (Mehta et al., 1996), *SPRINT* (Shafer et al., 1996), and *ScalParC* (Joshi et al., 1998). These algorithms require only one pre-sorting step, but need to construct a hash table at each level of the classification tree. In the parallel formulations of these algorithms, the content of this hash table needs to be available globally, requiring communication among processors. Existing parallel formulations of these schemes [Shafer et al., 1996; Joshi et al., 19981 perform communication that is similar in nature to that of our synchronous tree construction approach discussed in Section 3.1. Once again, communication in these formulations [Shafer et al., 1996; Joshi et al., 1998] can be reduced using the hybrid scheme of Section 3.3.

Another completely different way of handling continuous attributes is to discretize them once as a preprocessing step (Hong, 1997). In this case, the parallel formulations as presented in the previous subsections are directly applicable without any modification.

Another approach towards discretization is to discretize at every node in the tree. There are two examples of this approach. The first example can be found in [Alsabti et al., 19981 where quantiles (Alsabti et al., 1997) are used to discretize continuous attributes. The second example of this approach to discretize at each node is *SPEC* (Srivastava et al., 1997) where a clustering technique is used. *SPEC* has been shown to be very efficient in terms of runtime and has also been shown to perform essentially identical to several other widely used tree classifiers in terms of classification accuracy (Srivastava et al., 1997). Parallelization of the discretization at every node of the tree is similar in nature to the parallelization of the computation of entropy gain for discrete attributes, because both of these methods of discretization require some global communication among all the processors that are responsible for a node. In particular, parallel formulations of the clustering step in *SPEC* is essentially identical to the parallel formulations for the discrete case discussed in the previous subsections [Srivastava et al., 1997].

4. Analysis of the hybrid algorithm

In this section, we provide the analysis of the hybrid algorithm proposed in Section 3.3. Here we give a detailed analysis for the case when only discrete attributes are present. The analysis for the case with continuous attributes can be found in (Srivastava et al., 1997). The

Table 4. Symbols used in the analysis.

Symbol	Definition
N	Total number of training samples
P	Total number of processors
P_i	Number of processors cooperatively working on tree expansion
A_d	Number of categorical attributes
C	Number of classes
M	Average number of distinct values in the discrete attributes
L	Present level of decision tree
t_c	Unit computation time
t_s	Start up time of comminication latency [KGGK94]
t_w	Per-word transfer time of communication latency [KGGK94]

detailed study of the communication patterns used in this analysis can be found in (Kumar et al., 1994). Table 4 describes the symbols used in this section.

4.1. Assumptions

- The processors are connected in a hypercube topology. Complexity measures for other topologies can be easily derived by using the communication complexity expressions for other topologies given in (Kumar et al., 1994).
- The expression for communication and computation are written for a full binary tree with 2^L leaves at depth L. The expressions can be suitably modified when the tree is not a full binary tree without affecting the scalability of the algorithm.
- The size of the classification tree is asymptotically independent of N for a particular data set. We assume that a tree represents all the knowledge that can be extracted from a particular training data set and any increase in the training set size beyond a point does not lead to a larger decision tree.

4.2. Computation and communication cost

For each leaf of a level, there are A_d class histogram tables that need to be communicated. The size of each of these tables is the product of number of classes and the mean number of attribute values. Thus size of class histogram table at each processor for each leaf is:

$$\text{Class histogram size for each leaf} = C * A_d * M$$

The number of leaves at level L is 2^L. Thus the total size of the tables is:

$$\text{Combined class histogram tables for a processor} = C * Ad * M * 2^L$$

At level L, the local computation cost involves I/O scan of the training set, initialization and update of all the class histogram tables for each attribute:

$$\text{Local computation cost} = \theta\left(\frac{A_d * N}{P} + C * A_d * M * 2^L\right) * t_c = \theta\left(\frac{N}{P}\right) \tag{1}$$

where t_c is the unit of computation cost.

At the end of local computation at each processor, a synchronization involves a global reduction of class histogram values. The communication cost[1] is:

$$\text{Per level communication cost} = \left(t_s + t_w * C * A_d * M * 2^L\right) * \log P_i \leq \theta(\log P) \tag{2}$$

When a processor partition is split into two, each leaf is assigned to one of the partitions in such a way that number of training data items in the two partitions is approximately the same. In order for the two partitions to work independently of each other, the training set has to be moved around so that all training cases for a leaf are in the assigned processor partition. For a load balanced system, each processor in a partition must have $\frac{N}{P}$ training data items.

This movement is done in two steps. First, each processor in the first partition sends the relevant training data items to the corresponding processor in the second partition. This is referred to as the "moving" phase. Each processor can send or receive a maximum of $\frac{N}{P}$ data to the corresponding processor in the other partition.

$$\text{Cost for moving phase } \leq 2 * \frac{N}{P} * t_w \tag{3}$$

After this, an internal load balancing phase inside a partition takes place so that every processor has an equal number of training data items. After the moving phase and before the load balancing phase starts, each processor has training data item count varying from 0 to $\frac{2*N}{P}$. Each processor can send or receive a maximum of $\frac{N}{P}$ training data items. Assuming no congestion in the interconnection network, cost for load balancing is:

$$\text{Cost for load balancing phase } \leq 2 * \frac{N}{P} * t_w \tag{4}$$

A detailed derivation of Eq. 4 above is given in (Srivastava et al., 1997). Also, the cost for load balancing assumes that there is no network congestion. This is a reasonable assumption for networks that are bandwidth-rich as is the case with most commercial systems. Without assuming anything about network congestion, load balancing phase can be done using transportation primitive (Shankar, 1995) in time $2 * \frac{N}{P} * t_w$ time provided $\frac{N}{P} \geq O(P^2)$.

Splitting is done when the accumulated cost of communication becomes equal to the cost of moving records around in the splitting phase (Karypis, 1994). So splitting is done when:

$$\sum(\text{Communication Cost}) \geq \text{Moving Cost} + \text{Load Balancing}$$

This criterion for splitting ensures that the communication cost for this scheme will be within twice the communication cost for an optimal scheme (Karypis and Kumar, 1994). The splitting is recursive and is applied as many times as required. Once splitting is done, the above computations are applied to each partition. When a partition of processors starts to idle, then it sends a request to a busy partition about its idle state. This request is sent to a partition of processors of roughly the same size as the idle partition. During the next round of splitting the idle partition is included as a part of the busy partition and the computation proceeds as described above.

4.3. *Scalability analysis*

Isoefficiency metric has been found to be a very useful metric of scalability for a large number of problems on a large class of commercial parallel computers (Kumar et al., 1994). It is defined as follows. Let P be the number of processors and W the problem size (in total time taken for the best sequential algorithm). If W needs to grow as $f_E(P)$ to maintain an efficiency E, then $f_E(P)$ is defined to be the isoefficiency function for efficiency E and the plot of $f_E(P)$ with respect to P is defined to be the isoefficiency curve for efficiency E.

We assume that the data to be classified has a tree of depth L_l. This depth remains constant irrespective of the size of data since the data "fits" this particular classification tree.

Total cost for creating new processor sub-partitions is the product of total number of partition splits and cost for each partition split $(=\theta(\frac{N}{P}))$ using Eqs. (3) and (4). The number of partition splits that a processor participates in is less than or equal to L_l—the depth of the tree.

$$\text{Cost for creating new processors partitions} \leq L_1 * \theta\left(\frac{N}{P}\right) \tag{5}$$

Communication cost at each level is given by Eq. (2) (=θ(log P)). The combined communication cost is the product of the number of levels and the communication cost at each level.

$$\text{Combined communication cost for processing attributes} \leq L_1 * \theta(\log P) = \theta(\log P) \tag{6}$$

The total communication cost is the sum of cost for creating new processor partitions and communication cost for processing class histogram tables, the sum of Eqs. (5) and (6).

$$\text{Total communication cost} = \theta(\log P) + \theta\left(\frac{N}{P}\right) \tag{7}$$

Computation cost given by Eq. (1) is:

$$\text{Total computation time} = \theta\left(\frac{N}{P}\right) \tag{8}$$

Total parallel run time (Sum of Eqs. (7) and (8) = Communication time + Computation time.

$$\text{Parallel run time} = \theta(\log P) + \theta\left(\frac{N}{P}\right) \tag{9}$$

In the serial case, the whole dataset is scanned once for each level. So the serial time is

$$\text{Serial time} = \theta(N) * L_1 = \theta(N)$$

To get the isoefficiency function, we equate P times total parallel run time using Eq. (9) to serial computation time.

$$\theta(N) = P * \left(\theta(\log P) + \theta\left(\frac{N}{P}\right)\right)$$

Therefore, the isoefficiency function is $N = \theta(P \log P)$. Isoefficiency is $8 (P \log P)$ assuming no network congestion during load balancing phase. When the transportation primitive is used for load balancing, the isoefficiency is $O(P^3)$.

5. Experimental results

We have implemented the three parallel formulations using the MPI programming library. We use binary splitting at each decision tree node and grow the tree in breadth first manner. For generating large datasets, we have used the widely used synthetic dataset proposed in the *SLIQ* paper (Mehta et al., 1996) for all our experiments. Ten classification functions were also proposed in (Mehta et al., 1996) for these datasets. We have used the function 2 dataset for our algorithms. In this dataset, there are two class labels and each record consists of 9 attributes having 3 categoric and 6 continuous attributes. The same dataset was also used by the *SPRINT*algorithm (Shafer et al., 1996) for evaluating its performance. Experiments were done on an IBM SP2. The results for comparing speedup of the three parallel formulations are reported for parallel runs on 1, 2, 4, 8, and 16 processors. More experiments for the hybrid approach are reported for up to 128 processors. Each processor has a clock speed of 66.7 MHz with 256 MB real memory. The operating system is AIX version 4 and the processors communicate through a high performance switch (hps). In our implementation, we keep the "attribute lists" on disk and use the memory only for storing program specific data structures, the class histograms and the clustering structures.

First, we present results of our schemes in the context of discrete attributes only. We compare the performance of the three parallel formulations on up to 16 processor IBM SP2. For these results, we discretized 6 continuous attributes uniformly. Specifically, we discretized the continuous attribute *salary* to have 13, *commission* to have 14, *age* to have 6, *hvalue* to have 11, *hyears* to have 10, and *loan* to have 20 equal intervals. For measuring the speedups, we worked with different sized datasets of 0.8 million training cases and 1.6

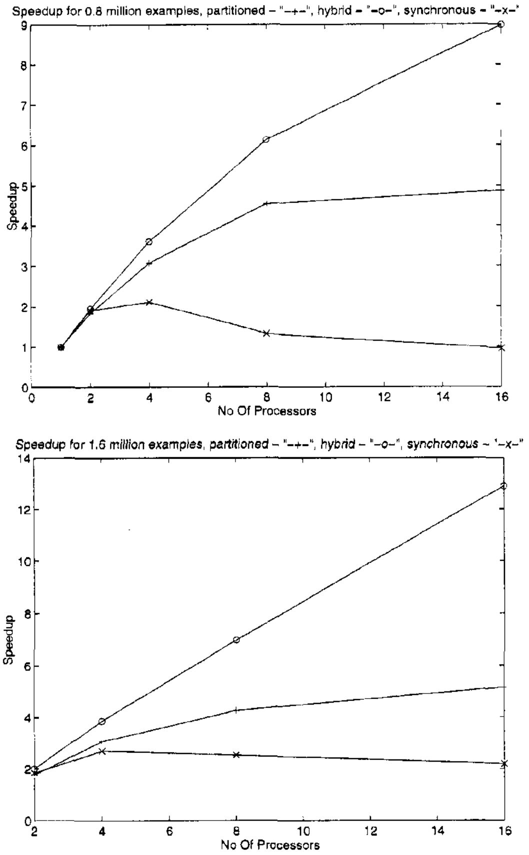


Figure 5. Speedup comparison of the three parallel algorithms.

million training cases. We increased the processors from 1 to 16. The results in figure 6 show the speedup comparison of the three parallel algorithms proposed in this paper. The graph on the left shows the speedup with 0.8 million examples in the training set and the other graph shows the speedup with 1.6 million examples.

The results show that the synchronous tree construction approach has a good speedup for 2 processors, but it has a very poor speedup for 4 or more processors. There are two reasons for this. First, the synchronous tree construction approach incurs high communication cost, while processing lower levels of the tree. Second, a synchronization has to be done among different processors as soon as their communication buffer fills up. The communication buffer has the histograms of all the discrete variables for each node. Thus, the contribution of each node is independent of its tuples count, the tuple count at a node being proportional to the computation to process that node. While processing lower levels of the tree, this synchronization is done many times at each level (after every 100 nodes for our experiments). The distribution of tuples for each decision tree node becomes quite different lower down in the tree. Therefore, the processors wait for each other during synchronization, and thus, contribute to poor speedups.

The partitioned tree construction approach has a better speedup than the synchronous tree construction approach. However, its efficiency decreases as the number of processors increases to 8 and 16. The partitioned tree construction approach suffers from load imbalance. Even though nodes are partitioned so that each processor gets equal number of tuples, there is no simple way of predicting the size of the subtree for that particular node. This load imbalance leads to the runtime being determined by the most heavily loaded processor. The partitioned tree construction approach also suffers from the high data movement during each partitioning phase, the partitioning phase taking place at higher levels of the tree. As more processors are involved, it takes longer to reach the point where all the processors work on their local data only. We have observed in our experiments that load imbalance and higher communication, in that order, are the major cause for the poor performance of the partitioned tree construction approach as the number of processors increase.

The hybrid approach has a superior speedup compared to the partitioned tree approach as its speedup keeps increasing with increasing number of processors. As discussed in Section 3.3 and analyzed in Section 4, the hybrid controls the communication cost and data movement cost by adopting the advantages of the two basic parallel formulations. The hybrid strategy also waits long enough for splitting, until there are large number of decision tree nodes for splitting among processors. Due to the allocation of decision tree nodes to each processor being randomized to a large extent, good load balancing is possible. The results confirmed that the proposed hybrid approach based on these two basic parallel formulations is effective.

We have also performed experiments to verify our splitting criterion of the hybrid algorithm is correct. Figure 7 shows the runtime of the hybrid algorithm with different ratio of communication cost and the sum of moving cost and load balancing cost, i.e.,

$$\text{ratio} = \frac{\sum(\text{Communication Cost})}{\text{Moving Cost} + \text{Load Balancing}}.$$

The graph on the left shows the result with 0.8 million examples on 8 processors and the other graph shows the result with 1.6 million examples on 16 processors. We proposed that splitting when this ratio is 1.0 would be the optimal time. The results verified our hypothesis as the runtime is the lowest when the ratio is around 1.0. The graph on the right with 1.6 million examples shows more clearly why the splitting choice is critical for obtaining a good performance. As the splitting decision is made farther away from the optimal point proposed, the runtime increases significantly.

The experiments on 16 processors clearly demonstrated that the hybrid approach gives a much better performance and the splitting criterion used in the hybrid approach is close to optimal. We then performed experiments of running the hybrid approach on more number of processors with different sized datasets to study the speedup and scalability. For these experiments, we used the original data set with continuous attributes and used a clustering

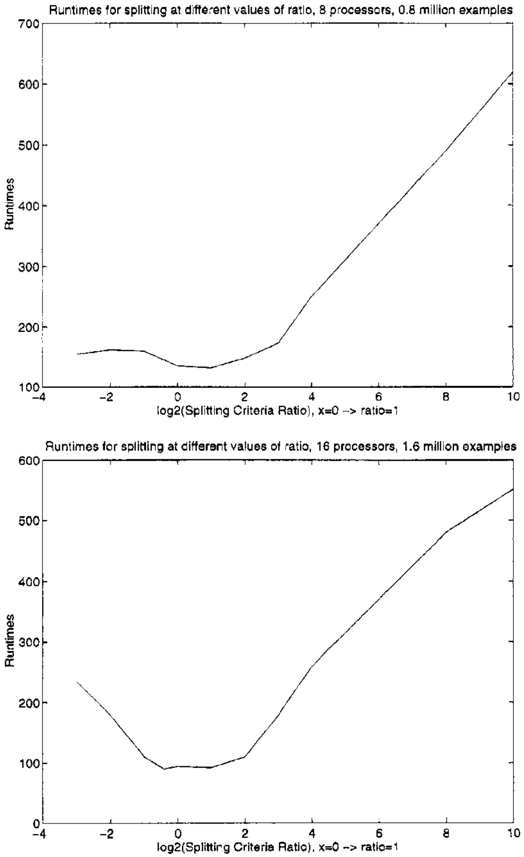


Figure 6. Splitting criterion verification in the hybrid algorithm.

technique to discretize continuous attributes at each decision tree node (Srivastava et al., 1997). Note that the parallel formulation gives *almost identical* performance as the serial algorithm in terms of accuracy and classification tree size (Srivastava et al., 1997). The results in figure 8 show the speedup of the hybrid approach. The results confirm that the hybrid approach is indeed very effective.

To study the scaleup behavior, we kept the dataset size at each processor constant at 50,000 examples and increased the number of processors. Figure 9 shows the runtime on increasing number of processors. This curve is very close to the ideal case of a horizontal line. The deviation from the ideal case is due to the fact that the isoefficiency function is $O(P \log P)$ not $O(P)$. Current experimental data is consistent with the derived isoefficiency function but we intend to conduct additional validation experiments.

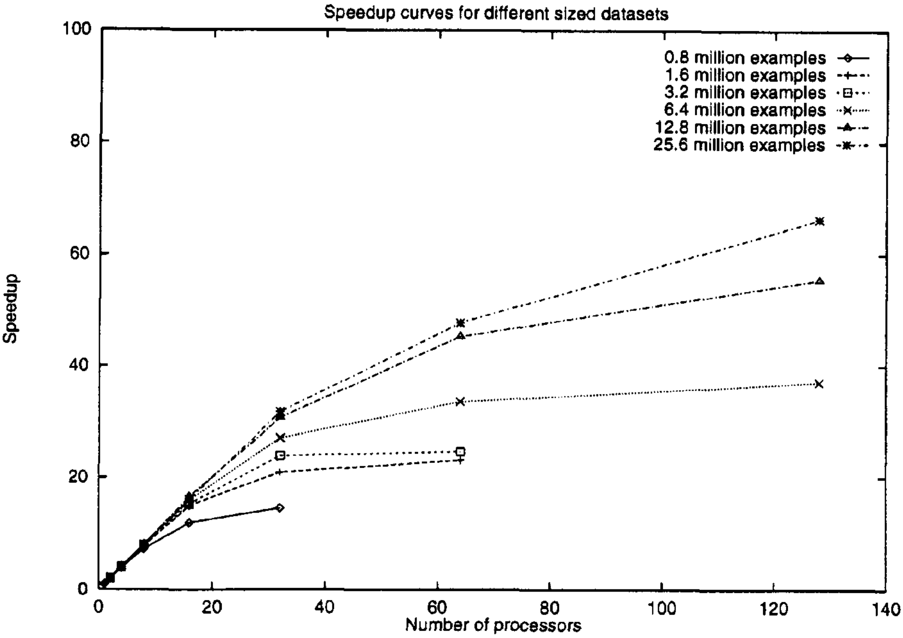


Figure 7. Speedup of the hybrid approach with different size datasets.

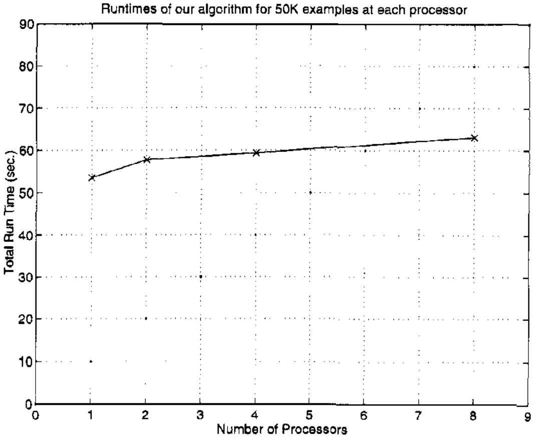


Figure 8. Scaleup of our algorithm.

6. Concluding remarks

In this paper, we proposed three parallel formulations of inductive-classification learning algorithm. The *Synchronous Tree Construction Approach* performs well if the classification tree remains skinny, having few nodes at any level, throughout. For such trees, there are relatively large number of training cases at the nodes at any level; and thus the communication overhead is relatively small. Load imbalance is avoided by processing all nodes at a level, before synchronization among the processors. However, as the tree becomes bushy, having a large number of nodes at a level, the number of training data items at each node decrease. Frequent synchronization is done due to limited communication buffer size, which forces communication after processing a fixed number of nodes. These nodes at lower depths of the tree, which have few tuples assigned to them, may have highly variable distribution of tuples over the processors, leading to load imbalance. Hence, this approach suffers from high communication overhead and load imbalance for bushy trees. The *Partitioned Tree Construction Approach* works better than *Synchronous Tree Construction Approach* if the tree is bushy. But this approach pays a big communication overhead in the higher levels of the tree as it has to shuffle lots of training data items to different processors. Once every node is solely assigned to a single processor, each processor can construct the partial classification tree independently without any communication with other processors. However, the load imbalance problem is still present after the shuffling of the training data items, since the partitioning of the data was done statically.

The hybrid approach combines the good features of these two approaches to reduce communication overhead and load imbalance. This approach uses the *Synchronous Tree Construction Approach* for the upper parts of the classification tree. Since there are few nodes and relatively large number of the training cases associated with the nodes in the upper part of the tree, the communication overhead is small. As soon as the accumulated communication overhead is greater than the cost of partitioning of data and load balancing, this approach shifts to the *Partitioned Tree Construction Approach* incrementally. The partitioning takes place when a reasonable number of nodes are present at a level. This partitioning is gradual and performs randomized allocation of classification tree nodes, resulting in a better load balance. Any load imbalance at the lower levels of the tree, when a processor group has finished processing its assigned subtree, is handled by allowing an idle processor group to join busy processor groups.

The size and shape of the classification tree varies a lot depending on the application domain and training data set. Some classification trees might be shallow and the others might be deep. Some classification trees could be skinny others could be bushy. Some classification trees might be uniform in depth while other trees might be skewed in one part of the tree. The hybrid approach adapts well to all types of classification trees. If the decision tree is skinny, the hybrid approach will just stay with the *Synchronous Tree Construction Approach.* On the other hand, it will shift to the *Partitioned Tree Construction Approach* as soon as the tree becomes bushy. If the tree has a big variance in depth, the hybrid approach will perform dynamic load balancing with processor groups to reduce processor idling.

Acknowledgments

A significant part of this work was done while Anurag Srivastava and Vineet Singh were at IBM TJ Watson Research Center. This work was supported by NSF grant ASC-96347 19, Army Research Office contract DA/DAAH04-95-1-0538, Cray Research Inc. Fellowship, and IBM partnership award, the content of which does not necessarily reflect the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPCRC, Minnesota Supercomputer Institute, Cray Research Inc., and NSF grant CDA-9414015.

Note

1. If the message size is large, by routing message in parts, this communication step can be done in time : $(t_s + t_w * \text{MesgSize}) * k_0$ for a small constant k_0. Refer to (Kumar et al., 1994) section 3.7 for details.

References

Agrawal, R., Imielinski, T., and Swami, A. 1993. Database mining: A performance perspective. IEEE Transactions on Knowledge and Data Eng., 5(6):914–925.

Alsabti, K., Ranka, S., and Singh, V. 1997. A one-pass algorithm for accurately estimating quantiles for disk-resident data. Proc. of the 23rd VLDB Conference.

Alsabti, K., Ranka, S., and Singh, V. 1998. CLOUDS: Classification for large or out-of-core datasets. http://www.cise.uft.edu/~ranka/dm.html.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. 1984. Classification and Regression Trees. Monterrey, CA: Wadsworth.

Catlett, J. 1991. Megainduction: machine learning on very large databases. PhD thesis, University of Sydney.

Chan, Philip K. and Stolfo, Salvatore J. 1993a. Experiments on multistrategy learning by metaleaming. Proc. Second Intl. Conference on Info. and Knowledge Mgmt, pp. 314–323.

Chan, Philip K. and Stolfo, Salvatore J. 1993b. Metalearning for multistrategy learning and parallel learning. Proc. Second Intl. Conference on Multistrategy Learning, pp. 150–165.

Chattratichat, J., Darlington, J., Ghanem, M., Guo, Y., Huning, H., Kohler, M., Sutiwaraphun, J., To, H.W., and Yang, D. Large scale data mining: Challenges and responses. Proc. of the Third Int'l Conference on Knowledge Discovery and Data Mining.

Goil, S., Alum, S., and Ranka, S. 1996. Concatenated parallelism: A technique for efficient parallel divide and conquer. Proc. of the Symposium of Parallel and Distributed Computing (SPDP'96).

Goldberg, D.E. 1989. Genetic Algorithms in Search, Optimizations and Machine Learning. Morgan-Kaufman.

Hong, S.J. 1997. Use of contextual information for feature ranking and discretization. IEEE Transactions on Knowledge and Data Eng., 9(5):718–730.

Joshi, M.V., Karypis, G., and Kumar, V., 1998. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. Proc. of the International Parallel Processing Symposium.

George Karypis and Vipin Kumar. 1994. Unstructured tree search on simd parallel computers. Journal of Parallel and Distributed Computing, 22(3):379–391.

Kufrin, R. 1997. Decision trees on parallel processors. In Parallel Processing for Artificial Intelligence 3. J. Geller, H. Kitano, and C.B. Suttner (Ed.). Elsevier Science.

Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. 1994. Introduction to Parallel Computing: Algorithm Design and Analysis. Redwod City: Benjamin Cummings/Addison Wesley.

Lippmann, R. 1987. An introduction to computing with neural nets. IEEE ASSP Magazine, 4(22).

Mehta, M., Agrawal, R., and Rissaneh, J. 1996. SLIQ: A fast scalable classifier for data mining. Proc. of the Fifth Int'l Conference on Extending Database Technology. Avignon. France.

Pearson, R.A. 1994. A coarse grained parallel induction heuristic. In Parallel Processing for Artificial Intelligence 2, H. Kitano, V. Kumar, and C.B. Suttner (Ed.). Elsevier Science, pp. 207–226.

Ross Quinlan, J. 1993. C4.5: Programs for Machine Learning. San Mateo, CA: Morgan Kaufmann.

Shafer, J., Agrawal, R., and Mehta, M. 1996. SPRINT A scalable parallel classifier for data mining. Proc. of the 22nd VLDB Conference.

Shankar, R., Alsabti, K., and Ranka, S. 1995. Many-to-many communication with bounded traffic. Frontiers '95, the Fifth Symposium on Advances in Massively Parallel Computation. McLean, VA.

Spiegelhalter, D.J., Michie, D., and Taylor, C.C. 1994. Machine Learning, Neural and Statistical Classification. Ellis Horwood.

Anurag Srivastava, Vineet Singh, Eui-Hong Han, and Vipin Kumar. 1997. An efficient, scalable, parallel classifier for data mining. Technical Report TR-97-0 10,http://www.cs,umn.edu/~kumar, Department of Computer Science, University of Minnesota, Minneapolis.

Wirth, J. and Catlett, J. 1988. Experiments on the costs and benefits of windowing in ID3.5th Int'l Conference on Machine learning.

Anurag Srivastava works at Digital Impact, a silicon valley start-up, developing data mining technologies for application to targeted email marketing. Prior to this, he was a researcher at Hitachi's data mining research labs. He did his B.Tech. from Indian Institute of Technology, Delhi 1995 and M.S. from University of Minnesota, Minneapolis in 1996. Most of his work has been in design and implementation of parallel and scalable data mining algorithms.

Eui-Hong (Sam) Han is a Ph.D. candidate in the Department of Computer Science and Engineering at the University of Minnesota. He holds a B.S. in Computer Science from the University of Iowa and an M.S. in Computer Science from the University of Texas at Austin. He worked at CogniSeis Development and IBM for several years before joining the Ph.D. program. His research interests include high performance computing, clustering, and classification in data mining. He is a member of ACM.

Vipin Kumar is a Professor in the Department of Computer Science and Engineering, and the director of graduate studies for the Graduate Program in Scientific Computation. Vipin Kumar's current research interests include High Performance computing, parallel algorithms for scientific computing problems, and data mining. His research has resulted in the development of the concept of isoefficiency metric for evaluating the scalability of parallel algorithms, as well as highly efficient parallel algorithms and software for sparse matrix factorization (PSPACES), graph partitioning (METIS, ParMetis), VLSI circuit partitioning (hMetis), and dense hierarchical solvers. He has authored over 100 research articles, and coedited or coauthored 5 books including the widely used text book "Introduction to Parallel Computing" (Publ. Benjamin Cummings/Addison Wesley, 1994). Kumar has given numerous invited talks at various conferences. workshops, national labs, and has served as chair/co-chair for many conferences/workshops in the area of parallel computing and high performance data mining. Kumar serves on the editorial boards of IEEE Concurrency, Parallel Computing, the Journal of Parallel and Distributed Computing, and served on the editorial board of IEEE Transactions of Data and Knowledge Engineering during 93-97. He is a senior member of IEEE, a member of SIAM, and ACM, and a Fellow of the Minnesota Supercomputer Institute.

Vineet Singh is an a start-up developing new products for ecommerce marketing. Previously, he has been Chief Researcher at Hitachi America's Information Technology Lab and he has held research positions in IBM, HP, MCC, and Schlumberger. He has a Ph.D. from Stanford University and a Master's from MIT.

This page intentionally left blank.

Data Mining and Knowledge Discovery, 3,263–290 (1999)
© 1999 Kluwer Academic Publishers. Manufactured in The Netherlands.

A Fast Parallel Clustering Algorithm for Large Spatial Databases

XIAOWEI XU* Xiaowei.Xu@mchp.siemens.de
Corporate Technology, Siemens AG, Otto-Hahn-Ring 6, D-81730 Munchen, Germany

JOCHEN JÄGER jaeger@informatik.uni-muenchen.de
HANS-PETER KRIEGEL kriegel@informatik.uni-muenchen.de
Institute for Computer Science, University of Munich, Oettingenstr. 67, D-80538 Munchen, Germany

Editors: Yike Guo and Robert Grossman

Abstract. The clustering algorithm DBSCAN relies on a density-based notion of clusters and is designed to discover clusters of arbitrary shape as well as to distinguish noise. In this paper, we present PDBSCAN, a parallel version of this algorithm. We use the 'shared-nothing' architecture with multiple computers interconnected through a network. A fundamental component of a shared-nothing system is its distributed data structure. We introduce the dR*-tree, a distributed spatial index structure in which the data is spread among multiple computers and the indexes of the data are replicated on every computer. We implemented our method using a number of workstations connected via Ethernet (10 Mbit). A performance evaluation shows that PDBSCAN offers nearly linear speedup and has excellent scaleup and sizeup behavior.

Keywords: clustering algorithms, parallel algorithms, distributed algorithms, scalable data mining, distributed index structures, spatial databases

1. Introduction

Spatial Database Systems (SDBS) (Gueting, 1994) are database systems for the management of spatial data, i.e. point objects or spatially extended objects in a 2D or 3D space or in some high-dimensional feature space. Knowledge discovery becomes more and more important in spatial databases since increasingly large amounts of data obtained from satellite images, X-ray crystal-lography or other automatic equipment are stored in spatial databases.

Data mining is a step in the KDD process consisting of the application of data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data (Fayyad et al., 1996). Clustering, i.e. grouping the objects of a database into meaningful subclasses, is one of the major data mining methods (Matheus et al., 1993). There has been a lot of research on clustering algorithms for decades but the application to large spatial databases introduces the following new conditions:

*This work was performed while the author was still working at the Institute for Computer Science, University of Munich.

(1) Minimal requirements of domain knowledge to determine the input parameters, because appropriate values are often not known in advance when dealing with large databases.
(2) Discovery of clusters with arbitrary shape, because the shape of clusters in spatial databases may be non-convex, spherical, drawn-out, linear, elongated, etc.
(3) Good efficiency on very large databases, i.e. on databases of significantly more than just a few thousand objects.

Ester et al. (1996) present the density-based clustering algorithm DBSCAN. For each point of a cluster, its *Eps*-neighborhood (for some given $Eps > 0$) has to contain at least a minimum number of points *(MinPts* > 0). DBSCAN meets the above requirements in the following sense: first, DBSCAN requires only two input parameters *(Eps, MinPts)* and supports the user in determining an appropriate value for it. Second, it discovers clusters of arbitrary shape and can distinguish noise. Third, using spatial access methods, DBSCAN is efficient even for very large spatial databases. In addition, a generalized version of DBSCAN can cluster point objects as well as spatially extended objects (Sander et al., 1998).

In this paper, we present a parallel clustering algorithm PDBSCAN which is based on DBSCAN for knowledge discovery in very large spatial databases. We use the 'shared-nothing' architecture which has the main advantage that it can be scaled up to hundreds and probably thousands of computers. As a data structure, we introduce the dR*-tree, a distributed spatial index structure. The main program of PDBSCAN, the master, starts a clustering slave on each available computer in the network and distributes the whole data set onto the slaves, Every slave clusters only its local data. The replicated index provides an efficient access of data, and the interference between computers is also minimized through the local access of the data. The slave-to-slave and master-to-slaves communication is implemented by message passing. The master manages the task of dynamic load balancing and merges the results produced by the slaves.

We implemented our method on a number of workstations connected via Ethernet (10 Mbit). A performance evaluation shows that PDBSCAN scales up very well and has excellent speedup and sizeup behavior. The results from this study, besides being of interest in themselves, provide a guidance for the design of parallel algorithms for other spatial data mining tasks, e.g. classification and trend detection.

This paper is organized as follows. Section 2 surveys previous efforts to parallelize other clustering algorithms. Section 3 briefly describes the algorithm DBSCAN and Section 4 presents our parallel clustering algorithm PDBSCAN. Section 5 shows experimental results and evaluates our parallel algorithm with respect to speedup, scalability, and sizeup. Section 6 lists the conclusions and highlights directions for future work.

2. Related work on parallel clustering algorithms

Several authors have previously proposed some parallel clustering algorithms. Rasmussen and Willett (1989) discuss parallel implementations of the single link clustering method on an SIMD array processor. Their parallel implementation of the SLINK algorithm does not decrease the $O(n^2)$ time required by the serial implementation, but a significant constant speedup factor is obtained. Li and Fang (1989) describe parallel partitioning clustering (the

k-means clustering algorithm) and parallel hierarchical clustering (single link clustering algorithm) on an n-node hypercube and an n-node butterfly. Their algorithms run in $O(n \log n)$ time on the hypercube and $O(n \log^2 n)$ on the butterfly. Olson (1995) has described several implementations of hierarchical clustering algorithms. His implementation of hierarchical clustering algorithm achieves $O(n)$ time on a n-node CRCW PRAM and $O(n \log n)$ time on $\frac{n}{\log n}$ node butterfly networks or trees. All these parallel clustering algorithms have the following drawbacks:

1. They assume that all objects can reside in main memory at the same time.
2. They need a large number of processors (about the size of the data set) to achieve a reasonable performance.

Both assumptions are prohibitive for very large databases with millions of objects. Therefore, database oriented parallel clustering algorithms should be developed.

Recently, (Pfitzner et al., 1998) present a parallel clustering algorithm for finding halos in N-body cosmology simulations. While overcoming the above two drawbacks, their method relies on some problem-specific knowledge and may be inappropriate for other disciplines.

In the literature, several parallel algorithms for mining association rules have been proposed recently (cf. Agrawal and Shafer, 1996; Cheung et al., 1996; Park et al., 1995). However, for many applications, especially for mining in large spatial databases, scalable parallel clustering algorithms are still in great demand.

In this section, we present a parallel clustering algorithm PDBSCAN which is based on DBSCAN for knowledge discovery in very large spatial databases. We use the shared-nothing architecture, with multiple computers interconnected through a network (Stonebraker, 1986).

3. The algorithm DBSCAN

The key idea of density-based clustering is that for each point of a cluster the neighborhood of a given radius *(Eps)* has to contain at least a minimum number of points *(MinPts),* i.e. the cardinality of the neighborhood has to exceed some threshold.

We will first give a short introduction of DBSCAN including the definitions which are required for parallel clustering. For a detailed presentation of DBSCAN see Ester et al. (1996).

Definition I. Directly density-reachable: A point p is *directly density-reachable* from a point q w.r.t. *Eps* and *MinPts* in the set of points D if

1. p E $N_{Eps}(4)$ $(N_{Eps}(q)$ is the subset of D contained in the Eps-neighborhood of q.)
2. $Card(N_{Eps}(q)) \geq MinPts$.

Definition 2. Density-reachable: A point p is *density-reachable* from a point q w.r.t. Eps and *MinPts* in the set of points D, denoted as $p >_D q$, if there is a chain of points $p_1, \ldots, p_n$,

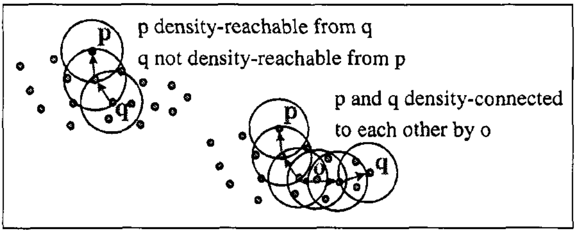

p density-reachable from q
q not density-reachable from p
p
q
p and q density-connected
to each other by o
p
o
q

Figure 1. Density-reachability and density-connectivity.

$p_1 = q$, $p_n = p$ such that $p_i \in D$ and p_{i+1} is directly density-reachable from p_i w.r.t. *Eps* and *MinPts*.

Density-reachability is a canonical extension of direct density-reachability. This relation is transitive, but not symmetric. Although it is not symmetric in general, it is obvious that the density-reachability is symmetric for points o with $Card(N_{Eps}(o)) \geq MinPts$. Two "border points" of a cluster are possibly not density-reachable from each other because there are not enough points in their *Eps*-neighborhoods. However, there must be a third point in the cluster from which both "border points" are density-reachable. Therefore, we introduce the notion of density-connectivity.

Definition 3. Density-connected: A point p is *density-connected* to a point q w.r.t. *Eps* and *MinPts* in the set of points D if there is a point $o \in D$ such that both p and q are density-reachable from o w.r.t. *Eps* and *MinPts* in D.

Density-connectivity is a symmetric relation. Figure 1 illustrates the definitions on a sample database of points from a 2-dimensional *vector* space. Note however, that the above definitions only require a distance measure and will also apply to data from a metric space.

A *cluster* is defined as a set of density-connected points which is maximal w.r.t. the density-reachability and the *noise* is the set of points not contained in any cluster.

Definition 4. Cluster: Let D be a set of points. A *cluster* C w.r.t. *Eps* and *MinPts* in D is a non-empty subset of D satisfying the following conditions:

1. Maximality: $\forall p, q \in D$: if $p \in C$ and $q >_D p$ w.r.t. *Eps* and *MinPts*, then also $q \in C$.
2. Connectivity: $\forall p, q \in C$: p is density-connected to q w.r.t. *Eps* and *MinPts* in D.

Definition 5. Noise: Let $C_1, \ldots, C_k$ be the clusters w.r.t. *Eps* and *MinPts* in D. Then, we define the *noise* as the set of points in the database D not belonging to any cluster C_i, i.e. $noise = \{p \in D \mid \forall i\colon p \notin C_i\}$.

We omit the term "w.r.t. *Eps* and *MinPts*" in the following whenever it is clear from the context. There are two different kinds of points in a clustering: *core points* (satisfying condition 2 of Definition 1) and *non-core points* (otherwise). In the following, we will refer to this characteristic of a point as the *core point property* of the point. The non-core points in turn are either *border points* (not a core point but density-reachable from another core point) or *noise points* (not a core point and not density-reachable from other points).

The algorithm DBSCAN was designed to discover the clusters efficiently and the noise in a database according to the above definitions. The procedure for finding a cluster is based on the fact that a cluster is uniquely determined by any of its core points:

- First, given an arbitrary point p for which the core point condition holds, the set $\{o \mid o >_D p\}$ of all points o density-reachable from p in D forms a complete cluster C and $p \in C$.
- Second, given a cluster C and an arbitrary core point $p \in C$, C in turn equals the set $\{o \mid o >_D p\}$ (cf. Lemmata 1 and 2 in Ester et al., 1996).

To find a cluster, DBSCAN starts with an arbitrary point p in D and retrieves all points of D which are density-reachable from p with respect to *Eps* and *MinPts.* If p is a core point, this procedure yields a cluster with respect to *Eps* and *MinPts.* If p is a border point, no points are density-reachable from p, and p is assigned to the noise. Then, DBSCAN visits the next point of the database D.

The retrieval of density-reachable points is performed by successive *region queries.* A *region query* returns all points intersecting a specified query region. Such queries are supported efficiently by spatial access methods such as the R*-trees (Beckmann et al., 1990).

The algorithm DBSCAN is sketched in figure 2.

```
Algorithm DBSCAN (D, Eps, MinPts)
// Precondition: All objects in D are unclassified.
 FOR ALL objects o in D DO:
  IF o is unclassified
    call function expand_cluster to construct a c;ister wrt/ E[s and MinPts containing o.

FUNCTION expand_cluster (o, D, Eps, MinPts):
 retrieve the EPS- neighborhood $N_{Eps}(o)$ of l;
 IF $|N_{Eps}(o)| < MinPts$  // i.e. o is not a core object
    mark o as noise and RETURN:
 ELSE // i.e. o is a core object
    select a new cluster-id and mark all objects in $N_{Eps}(o)$with this current cluster-id;
    push all objects from NEps(o)\{o} onto the stack seeds;
    WHILE NOT seeds.empty() DO
       currentObject := seeds.top();
       seeds.pop();
       retrievetheEPS-neighborhoodNEps(currentObject)ofcurrentObject;
       IF $|N_{Eps}(currentObject)| > MinPts$
          select all objects in $N_{Eps}(currentObject)$ not yet classified or makred as noise,
          push the unclassified objects onto seeds and mark all of these objects with current cluster-id;
 RETURN
```

Figure 2. Algorithm DBSCAN.

4. PDBSCAN: A fast parallel clustering algorithm

In this section, we present the parallel clustering algorithm PDBSCAN for mining in large spatial databases. We outline the proposed method in Section 4.1. The data placement strategy is crucial for the performance of the parallel algorithm. In Section 4.2, we propose an R*-tree based data placement strategy and the distributed spatial access method dR* -tree. The implementation of PDBSCAN is described in Section 4.3.

4.1. Proposed method

In this section, we outline the basic idea of our parallel clustering algorithm. We focus on the parallelization of DBSCAN for the following reasons:

1. DBSCAN is a clustering algorithm designed for knowledge discovery in spatial databases and it satisfies the requirements of discovering clusters of arbitrary shape from noisy databases as well as good efficiency on large databases.
2. The experience in the implementation of parallel DBSCAN may be directly used in other parallel clustering algorithms, e.g. DBCLASD (Xu et al., 1998), because they have the same algorithmic schema.

An overview of the hardware architecture is presented in figure 3. It consists of a number of computers (e.g. workstations or PCs) connected via a network (e.g. Ethernet). The problem is defined as follows:

Problem. Given a set of d-dimensional points $DB = \{p_1, p_2, \ldots, p_n\}$, a minimal density of clusters defined by *Eps* and *MinPts,* and a set of computers $CP = \{C_1, C_2, \ldots, C_N\}$ connected by a message passing network; find the density-based clusters with respect to the given *Eps* and *MinPts* values.

We use a partitioning strategy (Jaja, 1992) to implement parallel DBSCAN. Our method consists of three main steps. The first step is to divide the input into several partitions, and to distribute these partitions to the available computers. The second step is to cluster partitions concurrently using DBSCAN. The third step is to combine or merge the clusterings of the partitions into a clustering of the whole database. We describe this method more formally in the following:

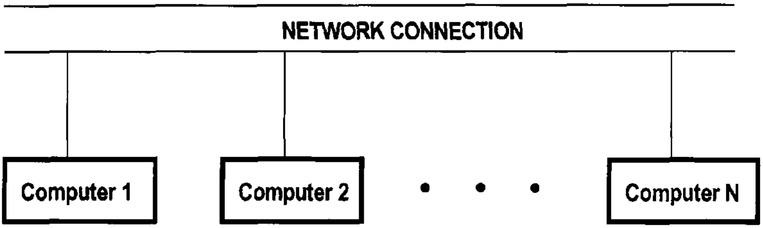


Figure 3. Proposed architecture (shared-nothing).

1. divide the input data set *DB* into *N* partitions $S_1, S_2, \ldots, S_N$ such that $DB = \cup_{i=1}^{N} S_i$ and $S_i \cap S_j = \emptyset$, for $i \neq j$.' The partition *Si* is distributed on C_i where $i = 1, 2, \ldots, N$.
2. process the *N* partitions concurrently using DBSCAN on the available computers C_1, $C_2, \ldots, C_N$., i.e. call algorithm DBSCAN(Si, *Eps, MinPts)* concurrently on C_i for $i = 1, 2, \ldots, N$.
3. merge the clustering results obtained from the partitions S_i, $i = 1, 2, \ldots, N$, into a clustering result for *DB*.

The first step is called *data placement* in the literature (Mehta and DeWitt, 1997). In a shared-nothing environment, a proper data placement is not only crucial for the performance and scalability of the parallel algorithm, but also for its load balancing. An ideal data placement strategy for our parallel clustering algorithm should satisfy the following requirements:

1. *Load balancing:* The data should be placed such that in the second step all concurrent parallel *DBSCAN(*S_i*, Eps, MinPts)*, $i = 1, 2, \ldots, N$, will be finished at the same time. Since the run-time of DBSCAN only depends on the size of the input data, the partitions should be almost of equal size if we assume that all computers have the same processing (computing and *I/O)* performance. If the computers have different processing performance, then we can distribute the input data on computers according to their processing performance. To simplify the description, we assume that all computers have the same processing performance in the following.
2. *Minimized communication cost:* The data should be placed such that the communication cost is minimized. To achieve this goal, each concurrent process of *DBSCAN(*S_i*, Eps, MinPts)*, $i = 1, 2, \ldots, N$, should avoid accessing those data located on any of the other computers, because the access of the remote data requires some form of communication. Nearby objects should be organized on the same computer.
3. *Distributed data access:* The data should be placed such that both local and remote data can be efficiently accessed. Without any spatial access method to support efficient access to local data, the run-time of the concurrent DBSCAN in step 2 is $O(|S_i|^2)$, where $|S_i|$ is the number of objects contained in the input data set S_i, $i = 1, 2, \ldots, N$. This is also the run-time complexity of parallel DBSCAN which does not scale well for large databases. Therefore, we need a spatial access method such as the R*-tree to support efficient access to the local data in order to reduce the run-time to $O(|S_i| \log |S_i|)$. Figure 4

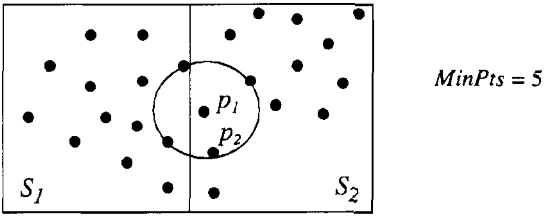


Figure 4. Illustration of the necessity of the access to remote data.

illustrates the necessity of the access to remote data: For a given *Eps,* and *MinPts* = 5, if there is no support of accessing remote data, then the neighborhood of object p_1 would contain only 3 points which is less than *MinPts,* and therefore p_1 would not be a core point. In this case, p_2 would not be density-reachable from any point in the partition S_2. According to the cluster definition (cf. Definition 4), p_2 would not be assigned to the cluster. Therefore, to obtain correct clustering results, a "view" over the border of partitions is necessary, i.e. the access to remote data should be supported. Of course, we have to pay communication cost for every access to remote data. This communication cost, however, can be minimized by the replication of indices which we will introduce in the following Section 4.2. On the other hand, access to remote data takes place only for the objects located on the border of two neighboring partitions. Another pay-off of remote data access is that we can efficiently merge the clustering results. We will discuss the merging issues in Section 4.3.

In the following section, we will present a new data placement method which satisfies the three requirements above. Our method, based on the R*-tree, provides not only a *spatial data placement* strategy for clustering, but also efficient access to spatial data in a shared-nothing architecture through the replication of indices. The new data placement strategy is not only useful for the parallelization of clustering, but may also be directly applied to other spatial data mining algorithms such as trend detection and classification.

4.2. Data placement and a distributed spatial access method

Data placement is an important resource management issue in the shared-nothing parallel and distributed database system. Much excellent research has been conducted on both relational databases and spatial databases. All previous work used the *declustering strategy* to place data among available computers.

Declustering exploits *I/O* parallelism but it also leads to higher communication cost. Declustering minimizes the query time for a single query. DBSCAN needs one range query for every object in the database, and thus we have to maximize the throughput of range queries, If we use a declustering strategy, the network may became the bottleneck. Therefore, declustering is not an optimal data placement strategy for efficient parallel clustering according to the requirements stated in Section 4.1.

According to the requirement of minimized communication cost in section 4.1, the objects that are close together in space and therefore likely to belong to the same cluster should be stored on the same computer. The goal is to reduce the communication cost and the interference between concurrent clustering operations. We call this strategy a *clustering data placement strategy.*

Given a spatial database, our first reaction would be to divide the data space into equi-sized grid cells and distribute buckets over available computers such that adjacent buckets are placed on the same computer. While this method satisfies the minimized communication cost requirement, it does not provide an efficient method for distributed data access (requirement 3 in Section 4.1).

Due to its good performance and its robustness, we use the R*-tree as our database interface to spatial data mining, as mentioned in (Ester et al., 1995). Let *M* be the number of directory entries that fit into a node and let m be a parameter specifying the minimum number of entries in a non-leaf node, $2 \leq m \leq [M/2]$. An R*-tree satisfies the following properties:

- The root has at least two children unless it is a leaf.
- Every non-leaf node contains between m and M entries unless it is the root.
- The tree is balanced, i.e. every leaf node has the same distance from the root.
- Non-leaf nodes contain entries of the form *(ptr, R),* whereptr is a pointer to a child node in the R*-tree; R is the MBR (minimal bounding rectangle) that covers all rectangles in the child node.
- Leaf nodes contain entries of the form *(obj_id, R)* where *obj_id* is a pointer to the object description, and R is the MBR of the object.

These facts lead to the idea of grouping the MBRs of leaf nodes of the R*-tree into N partitions such that the nearby MBRs should be assigned to the same partition and the partitions should be almost of equal size with respect to the number of MBRs. We assign the partitions to the computers. To achieve efficient access to distributed data, the index will be replicated on all computers. We have the following three design decisions:

1. How to partition the MBRs of the leaf nodes such that nearby rectangles are in the same partition, and the size of each partition is almost the same?
2. How to distribute the partitions of rectangles onto the computers?
3. How to replicate the index among N computers?

For the first question, we propose to use space filling Hilbert curves to achieve good clustering. In a k-dimensional space, a space-filling curve starts with a path on a k-dimensional grid of side 2. The path visits every point in the grid exactly once without crossing itself. This basic curve is said to be of order 1. To derive a curve of order n, each vertex of the basic curve is replaced by the curve of order $n-1$ which may be appropriately rotated and/or reflected. Figure 5 shows the Hilbert curves of order 1, 2 and 3 in the 2-dimensional space. The space filling curve can be generalized for higher dimensionality. An algorithm for higher dimensionality is presented by Bially (1969). The path of a space filling curve imposes a linear ordering which may be calculated by starting at one end of the curve and following the path to the other end. This ordering assigns a unique value, the Hilbert *value,* to each grid point. Figure 5 shows such an ordering. It was shown experimentally that the Hilbert curve achieves better clustering than other comparable methods (Faloutsos and Roseman, 1989). For a given R*-tree, this method works as follows. Every data page of the R*-tree is assigned to a Hilbert value according to its center of gravity. Thanks to the good clustering properties of the Hilbert curve, successive data pages will be close in space. We sort the list of pairs (Hilbert value/data page) by ascending Hilbert values. If the R*-tree has d data pages and we have n slaves, every slave obtains d/n data pages of the sorted list.

The solution to the second question is obvious: we distribute the partitions of the data pages on all available computers.

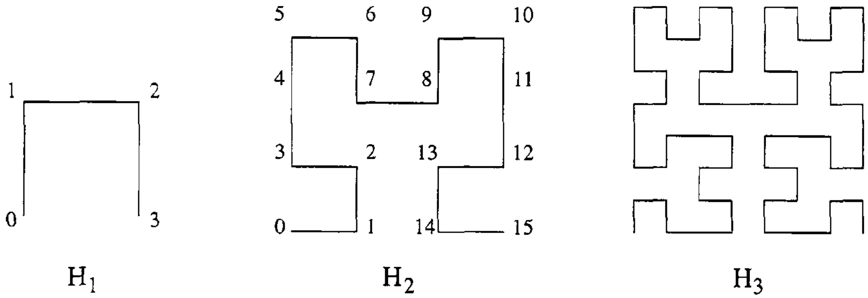


Figure 5. Hilbert curves of order 1, 2 and 3.

We propose to replicate the directory of the R*-tree on all available computers to enable efficient access to the distributed data. This replicated R*-tree is called dR*-tree which stands for distributed R*-tree. The goal is to increase the concurrency of the access. The dR*-tree has only the following structural differences from a traditional centralized R*-tree:

- the data pages are distributed on different computers
- the indices are replicated on all computers
- the pointer to the child node consists of a computer identifier *cptr* and a page identifier *ptr,* i.e. *(cptr, ptr, R)*

An example of a dR*-tree is given in figure 6. The original R*-tree has 7 data pages. These data pages are distributed onto two computers with 3 data pages on computer 1 and 4 data pages on computer 2.

The query can be started on every available computer. The query processing on a dR*-tree is very similar to the query processing on an R*-tree: a query is performed by starting at the root and computing all entries whose rectangle qualifies. For these entries, the corresponding child nodes are read into the main memory and the query process is repeated, unless the

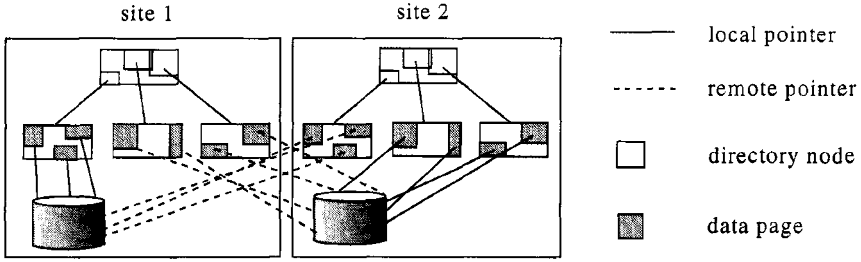


Figure 6. Distributed spatial index.

node in question is a leaf node. If the qualifying node is a local node, then this node can be read into the main memory. Otherwise, a “point to point” message must be sent to the computer where the node is stored. The remote node will be sent back by a “point to point” message.

In a central system, the run-time of a query is often measured by the number of pages accessed. In a distributed system, the run-time of a query is usually measured by the number of pages accessed and the number of messages. The following lemmata describe the performance of the dR*-tree with respect to the query processing.

Lemma 1. *A dR*-tree is a balanced tree with a height of* $O(\log n)$.

Proof: Since a dR*-tree has the same structure as the corresponding R*-tree except that the leaf nodes are distributed on different computers, a dR*-tree is balanced and the height of a dR*-tree is equal to the height of the corresponding R*-tree, i.e. $O(\log n)$. □

According to Lemma 1, a dR*-tree has the same performance as an R*-tree with respect to the number of accessed pages. In addition to the *I/O* cost, a dR*-tree has also communication cost but the following lemma shows:

Lemma 2. *The run-time of a query on a dR*-tree is of the same order of complexity as the corresponding R*-tree with respect to the number of accessed pages and messages.*

Proof: See Appendix. □

Although it makes no significant difference whether a query is processed by using a dR*-tree or an R*-tree according to Lemma 2, the dR*-tree enables a batch of queries to be concurrently processed on a distributed environment without interference. This advantage of the dR*-tree makes it very attractive for data mining algorithms if the goal is maximizing the throughput.

To summarize, the proposed dR*-tree meets the requirements for parallel clustering for the following reasons:

1. *Load balancing:* The number of objects (workload) on every computer is almost the same, because the number of data pages on every computer is almost the same. If the space utilization of the R*-tree is high (near 100%), the number of objects on every data page will be almost the same. 100% space utilization can be achieved by using index packing techniques (cf. Kamel and Faloutsos, 1993).
2. *Minimized communication cost:* Nearby objects are assigned to the same computer by partitioning data pages using Hilbert curves.
3. *Distributed data access:* Local and remote data can be efficiently accessed (cf. Lemma 2).

We can also use the same idea to extend other spatial access methods of the R-tree family, such as the X-tree (Berchtoldet al., 1996), to distributed spatial index structures onto several computers.

4.3. Algorithm PDBSCAN

After introducing the data placement strategy and the distributed spatial access method dR*-tree, we will present the algorithm PDBSCAN in this section.

We implemented PDBSCAN using the *master-slave model* (Geist et al., 1996) in which a separate "control" program termed *master* is responsible for process spawning, initialization, collection, displaying of results, and the timing of functions. The *slave* programs perform the actual computations involved.

In our implementation, the master program of PDBSCAN spawns one slave on each available computer (site). Every slave is responsible for clustering the data stored locally on its computer and reports the result to the master. The workload is a partition of the database which is obtained by using the data placement strategy proposed in Section 4.2. Later it is also called *S.*

Figure 7 illustrates the master-slave model. Obviously, the run-time of PDBSCAN is determined by the slowest slave. Therefore, in order to maximize the throughput, load balancing between all slaves is required. We achieve good load balancing with our data placement strategy which gives each slave nearly equal workload.

SLAVE is implemented to cluster points which are stored locally by using the modified DBSCAN algorithm PartDBSCAN which we will discuss later in this section. Once the initial workload is processed, SLAVE sends the clustering result as one packet to the master.

The goal of PartDBSCAN is, to find clusters in a partition *S* (workload) of the database *DB.* PartDBSCAN uses the same density-based notion of clusters as DBSCAN. Unlike the algorithm DBSCAN, PartDBSCAN handles only the partition *S* instead of the whole database *DB.* Therefore, we have to adapt DBSCAN to the *space constraint.* This leads to the adaptation of the related definitions adding a space constraint.

First, we introduce the adaptation of direct reachability. In the adapted definition, point *q* (core point) is restricted to the partition *S,* because PartDBSCAN is only responsible for finding clusters in the partition *S.* On the other hand, to achieve a correct clustering result, PartDB-SCAN has to know for every point *q* in *S* the number of objects contained in the *Eps*-neighborhood $N_{Eps}(q)$. If *q* is near the border of *S,* the Eps-neighborhood $N_{Eps}(q)$ may contain objects located in adjacent partitions of *S.* This case is illustrated in figure 8

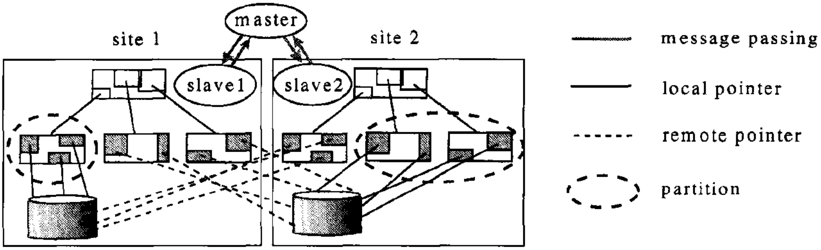
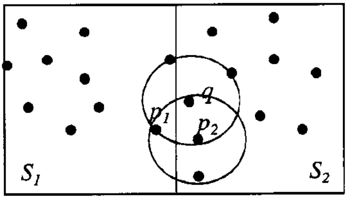


Figure 7. Master-slave model for PDBSCAN.



both p_1 and p_2 are directly density-reachable from q wrt. the space constraint S_2

q is directly density-reachable wrt. the space constraint S_2 neither from p_1 nor from p_2.

$MinPts = 5$

Figure 8. Illustration of the direct density-reachability with respect to a space constraint.

where our core point q is located in partition S_2. However, p_1 contained in $N_{Eps}(q)$ is located outside of S_2. Therefore, we adapt the definition of the direct density-reachability as follows:

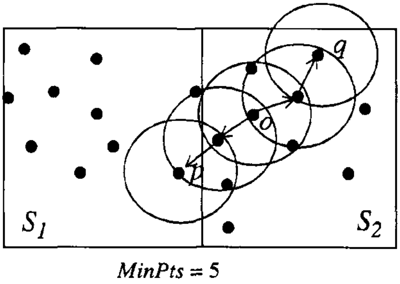
Definition 6. Directly density-reachable with respect to the space constraint *S:*
A point p is directly density-reachable from a point q w.r.t. the space constraint *S, Eps* and *MinPts* if

1. $q \in S$,
2. $p \in N_{Eps}(q)$ and
3. $Card(N_{Eps}(q)) \geq MinPts$ (core point condition).

From Definition 6, core point q must be located in the partition S, however, point p does not necessarily reside in S; and if S is equal to the whole data space *DB,* then being directly density-reachable w.r.t. the space constraint S is equivalent to being directly density-reachable. In general, it is obvious that if a point p is directly density-reachable from a point q w.r.t. the space constraint *S, Eps* and *MinPts* where $S \subseteq T$, then p is also directly density-reachable from q w.r.t. the space constraint *T, Eps* and *MinPts.* Obviously, this direct density-reachability is symmetric for pairs of core points. In general, however, it is not symmetric if one core point and one border point are involved. Figure 8 illustrates the definition and also shows the asymmetric case.

Definition 7. Density-reachable with respect to the space constraint *S:* A point p is density-reachable from a point q w.r.t. the space constraint *S, Eps* and *MinPts,* denoted by p >5 q, if there is a chain of points $p_1, \ldots, p_n$, $p_1 = q$, $p_n = p$ such that p_{i+1} is directly density-reachable from p_i w.r.t. the space constraint *S, Eps* and *MinPts.*

In Definition 7, points $p_1 = q, \ldots, p_{n-1}$ (core points) must be located in the partition S. However, point p does not necessarily reside in S. If S is equal to the whole data space *DB,* then being density-reachable w.r.t. the space constraint S is equivalent to being density-reachable. In general, it is obvious that if a point p is density-reachable from a point q w.r.t. the space constraint *S, Eps* and *MinPts* where $S \subseteq T$, then p is also density-reachable from q w.r.t. the space constraint *T, Eps* and *MinPts.* As density-reachability is a canonical extension of direct density-reachability, the density-reachability defined here is also a canonical extension of the direct density-reachability. This relation is transitive, but it is



both p and q are density-reachable wrt. the space constraint S_2 from o.

o is density-reachable wrt. the space constraint S_2 from neither p nor q.

p and q are density-connected wrt. the space constraint S_2 to each other by o.

Figure 9. Density-reachability and density-connectivity w.r.t. a space constraint.

not symmetric. Figure 9 depicts the relations of some sample points and, in particular, the asymmetric case. Although it is not symmetric in general, it is obvious that the density-reachability w.r.t. the space constraints is symmetric for core points because a chain from q to p can be reversed if p is also a core point.

Similar to the density-connectivity for a cluster, we introduce the notion of density-connectivity with respect to a space constraint.

Definition 8. Density-connected with respect to the space constraint S: A point p is density-connected to a point q w.r.t. the space constraint *S, Eps* and *MinPts* if there is a point o such that both, p and q are density-reachable from o w.r.t. the space constraint *S, Eps* and *MinPts*.

In Definition 8, point o (core point) must be located in the partition *S*. However, the points p and q do not necessarily reside in *S*. If *S* is equal to the whole data space *DB*, then being density-connected w.r.t. the space constraint *S* is equivalent to being density-connected. In general, it is obvious that if a point p is density-connected to a point q w.r.t. the space constraint S_1, *Eps* and *MinPts,* where $S_1 \subseteq S_2$, then p is also density-connected to q w.r.t. the space constraint S_2, *Eps* and *MinPts.* This density-connectivity is symmetric and reflexive (cf. figure 9).

Now, we can define a density-based cluster found w.r.t. a space constraint. Similar to our previous density-based cluster definition, a density-based cluster found w.r.t. the space constraint *S* is defined to be the maximum set of density-connected points which are density-reachable w.r.t. the space constraint S.

Definition 9. Cluster found with respect to the space constraint *S:* Let *DB* be a database of points and $S \subseteq DB$. A cluster *C* found w.r.t. the space constraint *S, Eps* and *MinPts* in *DB* is a non-empty subset of *DB* satisfying the following conditions:

1. Maximality: $\forall p, q$: if $p \in C$ and $p >_S q$, then also $q \in C$.
2. Connectivity: $\forall p, q \in C$: p is density-connected to q w.r.t. the space constraint *S, Eps* and *MinPts.*

Note that a cluster C found w.r.t. the space constraint S, *Eps* and *MinPts* contains at least one core point for the following reasons. Since C contains at least one point p, p must be density-connected w.r.t. the space constraint S to itself via some point o (which may be equal to p). Thus, at least o has to satisfy the core point condition. Consequently, the *Eps*- neighborhood of o has at least *MinPts*. This leads to the statement that a cluster found w.r.t. the space constraint S contains at least *MinPts* points. According to the definition of a cluster (cf. Definition 4), a cluster is obviously a cluster found with respect to the space constraint *DB* (cf. Definition 9). But a cluster found with respect to the space constraint S, where $S \subset DB$, is not necessarily a cluster. (Note: Lemma 6 shows when a cluster found w.r.t. the space constraint S, where $S \subset DB$, is also a cluster in the whole data space, i.e. a cluster found wrt. the space constraint *DB*.).

The following lemmata are important for validating the correctness of our clustering algorithm PartDBSCAN, executed by the slaves. Intuitively, they state the following. Given the parameters *Eps* and *MinPts*, we can discover a cluster w.r.t. the space constraint S in a two-step approach. First, choose an arbitrary point from S satisfying the core point condition as a seed. Second, retrieve all points that are density-reachable w.r.t. the space constraint S from the seed to obtain the 'constrained' cluster containing the seed. We omit the proof, because it is an analogue to the proof of Lemma 1 in (Ester et al., 1996).

Lemma 3. *Let p be a point in S and $Card(N_{Eps}(p)) \geq MinPts$. Then the set O $\{o \in DB \mid o >_S p\}$ is a clusterfound w.r.t. the space constraint S, Eps and MinPts.*

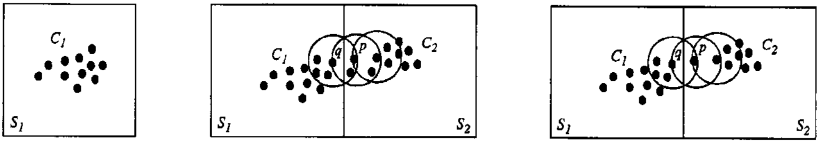
Let C be a cluster found w.r.t. the space constraint S, *Eps* and *MinPts*. C is uniquely determined by any of the core points in $C \cap S$ and, therefore, Ccontains exactly the points which are density-reachable w.r.t. the space constraint S from an arbitrary core point in $C \cap S$. The following lemma states the fact. We omit the proof again, because it is analogous to the proof of Lemma 2 in (Ester et al., 1996).

Lemma 4. *Let C be a clusterfound w.r.t. the space constraint S, Eps and MinPts, where $S \subseteq DB$. Let p be any point in $C \cap S$ with a $Card(N_{Eps}(p)) \geq MinPts$. Then C is equal to the set $O = \{o \in DB \mid o >_S p\}$.*

We want to show that clusters (found w.r.t. the space constraint *DB*), i.e. clusters found by DBSCAN, can be obtained by merging clusters with respect to adjacent space constraints found by PartDBSCAN. Obviously, if all members of a cluster C_1 found w.r.t. the space constraint S_1 are contained in its partition S_1, i.e. $C_1 \subseteq S_1$, then C_1 is also a cluster w.r.t. the space constraint *DB*. However, according to Definition 9, C_1 may contain a point p outside of S_1, i.e. $p \in C_1 \backslash S_1$. This could take place in two cases:

1. p is a core point, i.e. $Card(N_{Eps}(p)) \geq MinPts$.
2. p is a border point, i.e. $Card(N_{Eps}(p)) < MinPts$.

If p is a core point, a cluster C_2 w.r.t. the space constraint S_2 will be generated from p according to Lemma 3, where S_2 is adjacent to S_1. In this case, C_1 and C_2 should be merged in order to achieve the clustering result for DBSCAN. If p is a border point, there will be no cluster w.r.t. the space constraint S_2 generated from p, and in this case p is kept as a member of C_1. Figure 10 illustrates our discussion using *MinPts* = 4. The following lemmata are important to validate the algorithm of merging two clusters found w.r.t. their space constraints.



(a) $C_1 \subseteq S_1$, C_1 is a cluster found wrt. the space constraint *DB*

(b) $p \in C_1 \backslash S_1$ is a core point; C_1 should be merged with C_2

(c) $p \in C_1 \backslash S_1$ is a border point; $p \in C_1$, C_1 and C_2 are NOT merged!

MinPts = 4

Figure 10. Illustration of the relationship between clusters found w.r.t. adjacent space constraints.

Lemma 5. *Let C_1 be a clusterfound w.r.t. the space constraint S_1, and C_2 be a clusterfound w.r.t. thespaceconstraint S_2. If $\exists p \in C_1 \cap C_2 \cap (S_1 \cup S_2)$. such that $Card(N_{Eps}(p)) \geq MinPts$, then $C_1 \cup C_2$ is a cluster w.r.t. the space constraint $S_1 \cup S_2$, Eps and MinPts.*

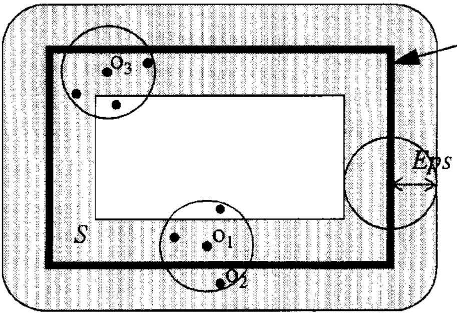
Proof: (1) Maximality: let $q_1 \in C_1 \cup C_2$ and $q_2 >_{S_1 \cup S_2} q_1$. Since $q_1 >_{S_1 \cup S_2} p$ and the density-reachability w.r.t. a space constraint is transitive, it follows that $q_2 >_{S_1 \cup S_2} p$. Hence, $q_2 \in C_1 \cup C_2$. *(2)* Connectivity: All points in $C_1 \cup C_2$ are density-connected w.r.t. the space constraint $S_1 \cup S_2$ via point *p*. □

The following lemma tells us when the fusion of clusters found w.r.t. adjacent space constraints should be terminated. So for a cluster *C* (found w.r.t. the space constraint S) if there is no core point *p* belonging to *C,* located outside of *S,* then *C* is also a cluster w.r.t. the space constraint *DB.* This lemma is also important to validate the algorithm of PDBSCAN.

Lemma 6. *Let C be a cluster found w.r.t. the space constraint S.* If $\forall p \in C \backslash S$: *$Card(N_{Eps}(p)) < MinPts$, then C is a cluster w.r.t. the space constraint DB.*

Proof: See Appendix. □

PartDBSCAN is a modified DBSCAN algorithm for finding clusters w.r.t. the given space constraint *S.* To find such a cluster, PartDBSCAN starts with an arbitrary point *p* within *S* and retrieves all points which are density-reachable from *p* w.r.t. the space constraint *S, Eps* and *MinPts.* If *p* is a core point, this procedure yields a cluster w.r.t. the space constraint *S, Eps* and *MinPts* (see Lemmata *3* and 4). If *p* is not a core point, no points are density-reachable from *p* w.r.t. the space constraint *S* and PartDBSCAN visits the next point in partition *S.* If all members of *C* are contained in *S, C* is also a cluster (w.r.t. the space constraint *DB)* according to Lemma 6. If there are members of *C* outside of *S, C* may need to be merged with another cluster found w.r.t. an adjacent space constraint according to Lemma 5. In this case, *C* should be sent to the master. We call *C* a *merging candidate.* To achieve an efficient implementation, we apply two techniques: (1) merging candidates will be gathered during the clustering procedure and sent to the master as one packet at the end



border of the constraint S.

o_1 is a core point wrt. *MinPts*=4 and o_2 is directly density-reachable from o_1. Both, o_1 and o_2 belongs to $MC(C,S)$ because o_2 may be also a core point.

o_3 does not belong to $MC(C,S)$ Although it is a core point, o_3 can not belong to another cluster, because it has no direct neighbor outside of S

Figure11. Illustration of the intersecting area of two clusters.

of the clustering procedure. (2) the size of each merging candidate is reduced to a subset which consists of only points near the border of the partition *S.*

The gathering of merging candidates is very simple: we use a list *L* to store all merging candidates. If a cluster extends to a point outside of its space constraint, this cluster will be appended to the list *L.* At the end of the clustering procedure, list *L* will be sent to the master.

The reduction in size of a merging candidate *C* is a little more complex: according to Lemma 5, the master PDBSCAN needs to merge *C* with another cluster found w.r.t. an adjacent space constraint if these two clusters have a core point located in their intersection. According to the definition of a cluster found w.r.t. a space constraint (see Definition 9), each point must be density-reachable from a core point in the space constraint. Therefore, outside of *S* (the space constraint of *C),* there are no points of *C* having a distance greater than *Eps* from the border of *S.* For the same reason, inside of *S,* there are no points of another cluster found w.r.t. an adjacent space constraint having a distance larger than *Eps* from the border of *S.* Hence, the area where two clusters may intersect *(intersecting area)* is an area around the border of the constraint S with a distance to either side of the border of no more than *Eps.* Figure 11 depicts the intersecting area for *C,* shown as a hollow shaded rounded rectangle, and the constraint *S,* represented as the thick rectangle. Therefore, the merging candidate can be reduced from *C* to a subset of *C, MC(C, S).*

Obviously, *MC(C,* S) should be contained in the intersecting area and consists of all points *o* of *C* which satisfy the following condition: If o is inside of *S,* then *O* must be a core point and its *Eps*-neighborhood must contain at least one point *p* outside of *S.* If this is the case, *O* may also be directly density-reachable from *p* w.r.t. an adjacent space constraint, and may be a member of another cluster found w.r.t. an adjacent space constraint according to Definition 9. In this case, C may intersect with another cluster found w.r.t. an adjacent space constraint in *O* or *p.* Therefore, both, *O* and *p* should be contained in *MC(C, S).* If *o* is outside of *S,* then *o* must be directly density-reachable from a core point *q* inside of *S* according to the cluster definition (cf. Definition 9). In this case, *C* may intersect with another cluster found w.r.t. an adjacent space constraint in *o* or *q*. Therefore, both, *q* and *o* should be included in *MC(C, S)*. In figure 11, points o_1 and o_2 meet the condition for *MC(C, S)*, and point o_3 does not meet the condition. To summarize, if a core point *q* of *C* which is located inside of *S* and its *Eps*- neighborhood contains at least one point outside of *S*, then both point *q* and all points which are located outside of *S* but are still directly density-reachable from *q* w.r.t. the space constraint *S* are contained in *MC(C, S)*:

$$MC(C, S) = \{o \in \{q\} \cup (N_{Eps}(q) \backslash S) \mid q \in C \cap S \wedge Card(N_{Eps}(q)) \geq MinPts \wedge N_{Eps}(q) \backslash S \neq \emptyset\}$$

The following lemma validates *MC* as reduced merging candidate:

Lemma 7. *Let C_1 and C_2 be clustersfound w.r.t. the space constraint S_1 and S_2, then*

$$MC(C_1, S_1) \supseteq \{o \in C_1 \cap C_2 \cap (S_1 \cup S_2) \mid Card(N_{Eps}(o)) \geq MinPts\}$$

Proof: See Appendix. □

Lemma 7 means that $MC(C_1, S_1)$ contains all core points belonging to $C_1 \cap C_2 \cap (S_1 \cup S_2)$ for any C_2 and S_2. Therefore, according to Lemmas 5 and 7, we can use $MC(C_1, S_1)$ and $MC(C_2, S_2)$ instead of C_1 and C_2 to test whether C_1 and C_2 should be merged or not. The advantage is that the size of the merging candidates is reduced, and this makes the size of the packet to send to the master smaller and the merging algorithm more efficient.

The test when two clusters found with respect to adjacent space constraints should be merged can be further simplified: we just have to calculate the intersection $MC(C_1, S_1) \cap MC(C_2, S_2)$ and do not have to check whether there is any core point in this intersection or not. The advantage is that we can avoid the core point test and this makes the fusion of clusters even more efficient. The following lemma validates this idea.

Lemma 8. *Let C_1 be a clusterfound w.r.t. the space constraint S_1 and C_2 be a cluster found w.r.t. the space constraint S_2. IfMC(C_1, S_1) $\cap$ MC(C_2, S_2) $\neq \emptyset$, then $C_1 \cup C_2$ is a cluster w.r.t. the space constraint $S_1 \cup S_2$, Eps andMinPts.*

Proof: See Appendix. □

The algorithm PartDBSCAN is sketched in figure 12. *S* is the partition (workload). The dR*-tree is a distributed R*-tree which provides efficient access to local and remote data. L is a list of merging candidates. The merging candidates are the subset of clusters (found w.r.t. some space constraint) that may need to be merged in order to achieve the same clustering result as DBSCAN. The merging candidates are collected and appended to *L*. *L* will be sent to PDBSCAN at the end of the clustering procedure. ExpandCluster, which is the most important function used by PartDBSCAN, is presented in figure 13.

During the expansion of the current cluster, whenever a point outside of the partition *S* is reached, the point and its core point are inserted into a set *MC* which defines the reduced

```
Algorithm: PartDBSCAN (S, dR*-tree, Eps, MinPts)
  // S is the workload and UNCLASSIFIED
  // L is the list of merging candidates
  initialize L to be empty;
  FORALL objects o in S DO
      IF o is UNCLASSIFIED
          //construct a cluster wrt. S, Eps and MinPts containing o.
          IF ExpandCluster(S, dR*-tree, o, Clusterld Eps, MinPts, L)
              increase ClusterId:
  IF L NOT empty
      send L to the master;
```

Figure 12. Algorithm PartDBSCAN.

```
FUNCTION: ExpandCluster(S, dR*-tree, o, Clusterld, Eps, MinPts, L): Boolean;
  MC.init(); // initialize the merging candidate set;
  retrieve Eps- neighborhood $N_{Eps}(o)$
  IF $|N_{Eps}(o)| < MinPts$        // i.e. o is not a core point;
      mark o as noise and RETURN false;
  ELSE   // i.e. o is a core point;
      select a new Clusterld and mark all objects in $N_{Eps}(o)$ with Clusterld;
      push all objects from $N_{Eps}(o) \setminus \{o\}$ onto the stack seeds;
      WHILE NOT seeds.empty() DO
          currentobject := seeds.top();
          seeds.pop();
          IF $currentobject \in S$
              retrieve Eps- neighborhood $N_{Eps}(currentObject)$;
              IF $|N_{Eps}(currentObject)| \geq MinPts$
                  select all objects in $N_{Eps}(currentObject)$ not yet classified or marked as noise;
                  push the unclassified objects onto seeds and mark them with Clusterld;
                  IF $N_{Eps}(currentObject) \setminus S$ NOT Empty
                      insert $(currentobject) \cup N_{Eps}(currentObject) \setminus S$ into the set MC
          ELSE   // currentobject is not element of S;
              insert o and currentobject into set MC
      IF MC # Empty THEN
          L.append(MC);
      RETURNTrue;
```

Figure 13. ExpandCluster function.

merging candidate of the current cluster (see Lemma 7). If *MC* is empty, this means that the current cluster (found w.r.t. the space constraint *S)* is also a cluster w.r.t. the space constraint *DB* (see Lemma 6). Otherwise, the current cluster may need to be merged with another cluster (see Lemmas 5 and 7). At the end of the expansion, *MC* will be appended to *L.* The retrieval of the *Eps-* neighborhood for a given object is performed by using a region query. In Euclidean space, e.g., this region is a circle. The center of this circle is the query object and the radius equals Eps. Obviously, the run-time of PartDBSCAN is $O(|S| *$ run-time of a region query): for every point in the partition *S* a region query is performed by using the dR*-tree. Since the partition *S* consists of only local points and the height of the dR*-tree is *O*(log *n),* for small *Eps* the run-time of a region query is *O*(log*n)* on the average. Hence, the run-time of PartDBSCAN is $O(|S|$ * log *n)* in the average case. In general, the run-time complexity of PartDBSCAN is equivalent to DBSCAN on partition *S* w.r.t. the number of accessed pages and passed messages.

Lemma 9. *For a given partition S, PartDBSCAN based on a dR*-tree has the same order of run-time complexity as DBSCAN based on the corresponding R*-tree w.r.t. the number of accessed pages and passed messages.*

Proof: Since the run-time complexity of PartDBSCAN is $O(|S|$ * run-time of a region query on the dR*-tree) and the run-time complexity of DBSCAN is $O(|S|$ * run-time of a region query on the R*-tree), according to Lemma 2, we have proven the lemma. □

The master PDBSCAN receives a list of merging candidates from every SLAVE who has at least one merging candidate. Each merging candidate represents a cluster found with respect to a space constraint which may now need to be merged with another cluster found with respect to an adjacent space constraint. PDBSCAN collects all the lists *L* it receives and assigns them to a list *LL.* If *LL* is non-empty, a merging function Merging (shown in figure 14) will be executed.

The function Merging goes through the list *LL* and merges all clusters (found with respect to their different constraints) if their intersections are non-empty. Therefore, we have the following lemma.

Lemma 10. *The clustering result obtained by PDBSCAN is the same as the clustering result obtained by DBSCAN.*

Proof: It is obvious according to Lemmas 5–8. □

5. Performance evaluation

In this section, we evaluate the performance of PDBSCAN with respect to its scaleup, speedup and sizeup. For this purpose, we implemented PDBSCAN on a cluster of HP/UX workstations consisting of 8 HP C160 workstations (with a 150 MHz PA8000 CPU) interconnected by a 10 MB LAN using C++ and PVM (Parallel Virtual Machine, Geist et al., 1996). In this section, we will only evaluate the efficiency of PDBSCAN and not its accuracy,

```
Function: Merging (LL)
 FOR i FROM 1 TO LL.size DO
   L1 := LL.get(i);
   FOR j FROM 1 TO L1.size DO
     C1 := L1.get(j);
     FOR k FROM i+1 TO LL.size DO
       L2 := LL.get(k);
       FOR m FROM 1 TO L2.size DO
         C2 := L2.get(m);
         IF $C1 \cap C2 \neq \emptyset$ THEN
           C1 := $C1 \cup C2$;
           L2.remove(C2);
```

Figure 14. Merging function.

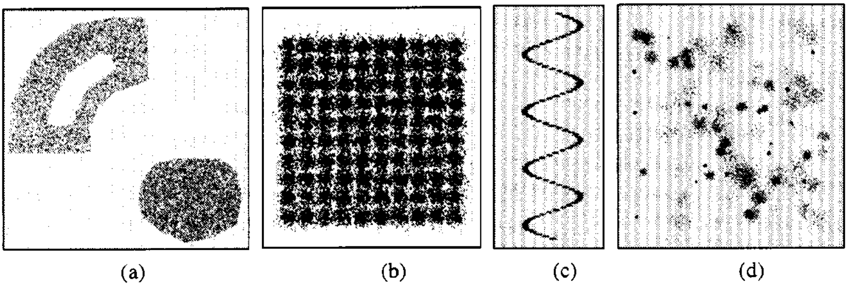


Figure 15. Synthetic data sets.

because PDBSCAN produces always the same results as DBSCAN (cf. Lemma 10). The parameters *(Eps* and *MinPts)* were chosen by using the heuristic presented in (Sander et al., 1998). A more detailed report on experimental and theoretical evaluation of PDBSCAN can also be found in (Xu, 1999).

For the following evaluations, we used both synthetic and real databases. The first synthetic data set we used is depicted in figure 15(a), which represents two clusters, each containing 10,000 points. For scaleup and sizeup experiments, we generated a series of these synthetic databases of varying size from 10,000 points with one cluster to 1 million points with 100 clusters by concatenating the original data file depicted in figure 15(a). The 1000k data set (1 million points) simply contains 50 of these two cluster sets. The databases are named according to the number of points contained. So 100k is a database containing 100,000 points.

We also used three synthetic data sets which were used for the evaluation of the BIRCH algorithm (see Zhang et al., 1998). The data sets birch1, birch2 and birch3 contain 100,000

Table 1. Synthetic and real databases.

Name	Number of points	Dimensions	Number of actual clusters	Size (in Kbytes)	Runtime for one processor (sec)
1000k	1,000,000	2	100	21,632	4199
birch1	100,000	2	100	2248	607
birch2	100,000	2	100	2248	551
birch3	100,000	2	100	2248	369
seq_534k	534,363	5	9	18,976	8916

2-dimensional points which were divided into 100 actual clusters. The structure of these data sets can be seen in figures 15 (b)–(d).For scaleup and sizeup experiments a series of data sets, beginning with 100,000 points and ending with 800,000 points, was generated. They were named birchx_y, where *x* denotes the data set type and y denotes the size. So birch3_500k would be the data set birch3 concatenated five times and containing 500,000 points.

As real-world data sets we used the raster data of the SEQUOIA 2000 benchmark database (Stonebraker et al., 1993), which is representative for Earth Science tasks. The raster data contains 5-dimensional points obtained from several satellite images of a region on the surface of the earth covering California. Every 1000 by 1000 meter area on the surface corresponds to a 5-dimensional vector, containing information for 5 different spectral channels. Finding clusters in such feature spaces is a common task in remote sensing digital image analysis (Richards, 1983) for the creation of thematic maps in geographic information systems. (Sander et al., 1998) describe the application of DBSCAN to the SEQUOIA data set. We used the same data set where identical points were removed (seq534k). For scaleup and sizeup experiments, we used a series of data sets, beginning with 100,000 points and ending with 800,000 points. They were named seq_y, where y denotes the size. seq_100k corresponds to the data subset of 100,000 points of the data set sequoia. seq_500k is five times the concatenation of this subset, so it contains 500,000 points. The characteristics of the databases we used are summarized in Table 1.

We report on the results of a series of experiments described in Section 5.1. In Section 5.2, we study the *I/O* and communication cost factors in our experiments. The purpose was to explore further possibilities to improve the efficiency of the PDBSCAN algorithm.

5.1. Experimental performance evaluation

Below, we examine the speedup, scaleup and sizeup characteristics of the PDBSCAN algorithm.

To measure the speedup, we keep the database constant and increase the number of computers in the system. More formally, we apply our clustering algorithm to a given database in a system consisting of one computer (server). Then we increase the number of computers in the system from 1 to *m,* and cluster the database in the system with *m* computers. The speedup given by the larger system with *m* computers is measured as:

Speedup(m) = run-time on one compute/run-time on *m* computers

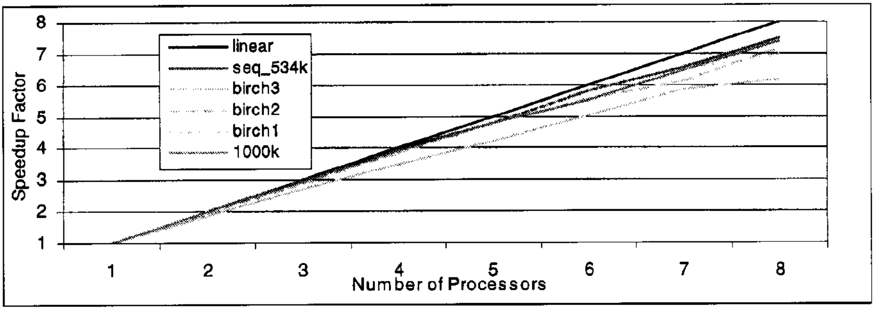


Figure 16. Speedup.

The ideal parallel algorithm demonstrates linear speedup: a system with m times the number of computers yields a speedup of *m*. However, linear speedup is difficult to achieve because of the communication cost and the skew of the slaves, i.e. the problem that the slowest slave determines the total time needed. If not every slave needs the same time, we have this skew problem.

We have performed the speedup evaluation on databases with quite different sizes and structures. The number of computers varied from 1 to 8. Figure 16 shows the speedup for these databases. As the graphs show, PDBSCAN has a very good speedup performance. This performance is almost the same for databases with very different sizes and shapes. Birch3 has a slightly lower speedup curve, because there are many clusters with varying density which sometimes even overlap. It is very hard to distribute the R-tree data pages to the slaves in a manner that one slave has only neighboring points. Therefore, the skew in this case is higher than in the other data sets and the total speedup is lower.

Speedup analysis holds the database size constant and grows the system. Scaleup measures the ability to grow both the system and the database size. Scaleup is defined as the ability of an *m*-times larger system to perform an m-times larger job in the same run-time as the original system. The scaleup metric is:

Scaleup(DB, m) = run-time for clustering *DB* on 1 computer/run-time for clustering $m * DB$ on *m* computer

To demonstrate how well the PDBSCAN algorithm handles larger databases when more computers are available, we have performed scaleup experiments where we have increased the size of the databases in direct proportion to the number of computers in the system. For the data set birch1, e.g., 100,000 points are clustered on 1 computer and 800,000 points are clustered on 8 computers. Figure 17 shows the performance results of the databases. Clearly, the PDBSCAN algorithm scales very well.

Sizeup analysis holds the number of computers in the system constant, and grows the size of the databases by the factor *m*. Sizeup measures how much longer it takes on a given system, when the database size is *m*- times larger than the original database. The sizeup

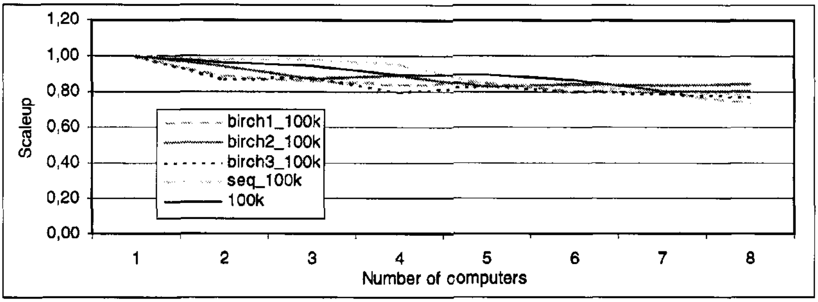


Figure 17. Scaleup.

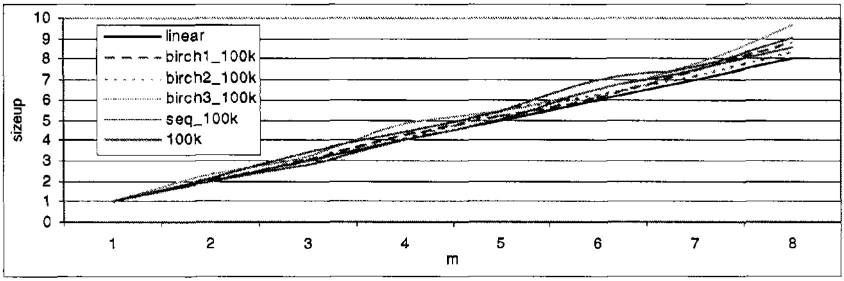


Figure 18. Sizeup.

metric is defined as follows:

Sizeup(DB, m) = run-time for clustering $m * DB$/ run-time for clustering *DB*

To measure the performance of sizeup, we have fixed the number of computers to 1, 2, 4, and 8 respectively. Figure 18 shows the results on 8 computers. The graphs show that PDBSCAN has a very good sizeup performance. An 8 times larger problem needs about 8 to 10 times more time.

5.2. A Study on I/O and communication cost

In this section, we study the two main cost factors of PDBSCAN, i.e. *I/O* cost and communication cost. Typically, the *I/O* cost and the communication cost are measured by the number of pages accessed and the number of messages passed, respectively.

To analyze the impact of *I/O* cost on the performance of PDBSCAN, we fix the database size and grow the system by increasing its number of computers. Figure 19 shows the *I/O* cost as a function of the number of computers. The vertical axis is scaled logarithmically. The graphs show that PDBSCAN has a very good *I/O* cost performance. The *I/O* cost is

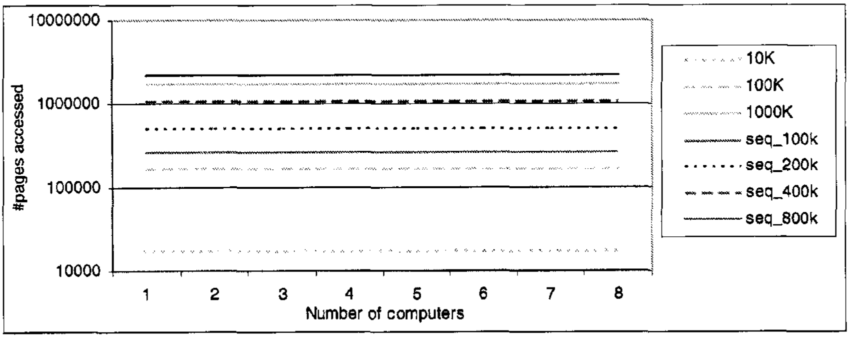


Figure 19. *I/O* cost.

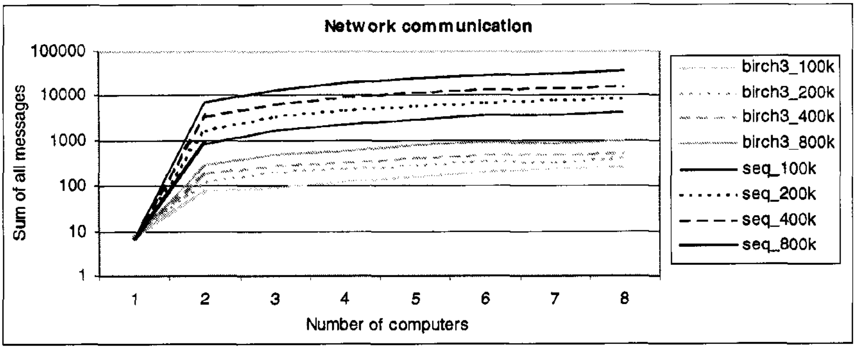


Figure 20. Communication cost.

almost constant as the number of computers in the system increases. This shows also that the dR*-tree has a good scalability w.r.t. the number of computers used.

To see the impact of the communication cost on the performance of PDBSCAN, again we fixed the database size and increased the number of computers in the system. Figure 20 shows the communication cost, i.e. the number of messages passed, as a function of the number of computers for the databases birch and SEQUOIA. The graph plots indicate that the communication cost increases linearly w.r.t. the number of computers used.

Therefore, the performance of PDBSCAN can be further improved if we use a high speed network, e.g. FDDI (100 Mbit/sec) or HiPPI (800 Mbit/sec to 1.6 Gbit/sec) instead of Ethernet (10 Mbit/sec) in our shared-nothing system.

6. Conclusions

In this paper, we proposed the parallel clustering algorithm PDBSCAN for mining in large distributed spatial databases. The main characteristics are:

1. an R*-tree based data placement strategy which minimizes the interference between slaves.
2. a distributed spatial access method dR*-tree which meets the requirements for parallel clustering with respect to load balancing, and minimizes communication cost as well as distributed data access.
3. an efficient parallel clustering algorithm PDBSCAN which is based on DBSCAN. Our theoretical analysis and the experimental evaluation show that PDBSCAN has very good performance w.r.t. speedup, scaleup and sizeup. We have also proven that the clustering result of PDBSCAN is the same as that of DBSCAN.

There are also some interesting issues for future research. This paper has shown that the distributed access method is a very powerful structure for data mining in the distributed environment. We can also use the same idea to extend other spatial access methods of the R-tree family, such as the X-tree, to distribute spatial index structures for high-dimensional data.

Other data placement strategies should also be considered and compared with our method. Furthermore, the parallelization of other spatial data mining methods should be considered in the future.

Appendix

Proof of Lemma 2: Since a dR*-tree has the same structure as the corresponding R*-tree except that the leaf nodes are distributed on different severs, a query processed using a dR*-tree accesses the same number of pages as a query using its corresponding R*-tree. We denote the number of accessed pages by *#accessed_pages.* We use *#accessed_data_pages* to denote the number of accessed data pages. It is obvious that *#accessed_data_pages* < #accessed-pages, Two messages are passed for each accessed data page. Therefore, the run-time *QT* of a query on a dR*-tree can be expressed as follows:

$$\begin{aligned} QT &= \text{time for } I/O + \text{time for communication cost} \\ &= O\,(\#accessed\_pages) + o\,(2 \text{ x } \#accessed\_data\_pages) \\ &= O(\#accessed\_pages) \end{aligned}$$

Hence, the run-time of a query on a dR*-tree is of the same order of complexity as on an R*-tree with respect to the number of accessed pages and the number of messages passed. □

Proof of Lemma 6: (1) Maximality: Let $p \in C$ and let be q density-reachable from p w.r.t. the space constraint *DB, Eps* and *MinPts,* i.e. $q >_{DB} p$,. According to Definition 7, there is a chain of points $p_1, \ldots, p_n, p_i = p, p_n = q$ such that p_{i+1} is directly density-reachable from p_i w.r.t. the space constraint *DB, Eps* and *MinPts,* and $p_i \in DB$. Since p_i, $i = 1, \ldots, n - 1$ are core points, $p \in C$. Hence, also q E C. (2) Connectivity: for $\forall p\ q, \in C$, since p is density-connected to q w.r.t. the space constraint *S, Eps* and *MinPts,* where $S \subseteq DB$, then p is also density-connected to q w.r.t. the space constraint *DB, Eps* and *MinPts.* □

Proof of Lemma 7: Let $p \in \{o \in C_1 \cap C_2 \cap (S_1 \cup S_2)$ $Curd(N_{Eps}(o)) \geq MinPts]$. It follows that $p \in (C_1 \cap C_2 \cap S_1) \cup (C_1 \cap C_2 \cap S_2)$ and $Curd(N_{Eps}(p)) \geq MinPts$. Since $S_1 \cap S_2 = \emptyset$ there are only two possibilities:

(1) $p \in C_1 \cap C_2 \cap S_1$. In this case, we have $p \in C_1 \cap S_1$ and p E C_2. This implies that p must be directly density-reachable from a point q w.r.t. the space constraint S_2 in C2. It follows that $dist(p, q) \leq Eps$. Therefore, $q \in NEps(p)\backslash S1$. This means that N_{Eps} $(p)\backslash S_1 \neq \emptyset$ To summarize, p satisfies the conditions for , S_1): $p \in C_1 \cap S_1$, $Card(N_{Eps}(p)) \geq MinPts$ and $N_{Eps}(p)\backslash S_1 \neq \emptyset$

(2) $p \in C_1 \cap C_2 \cap S_2$. In this case, we have $p \notin$ S1 and p E C_1. It follows that p must be directly density-reachable from a point q w.r.t. the space constraint S_1 in C_1 such that $Curd(N_{Eps}(q)) \geq MinPts$ and $p \in N_{Eps}(q)\backslash S_1$. To summarize, p satisfies the conditions for $MC(C_1, S_1)$: $\exists q \in C_1 \cap S_1$ *such* that $Card(N_{Eps}(q)) \geq MinPts$ and $p \in N_{Eps}(q)\backslash S_1$.
Therefore, $p \in MC(C_1, S_1)$. □

Proof of Lemma 8: Let $p \in MC(C_1, S_1) \cap MC(C_2, S_2)$. It follows that $p \in MC(C_1, S_1)$. According to the definition of $MC(C_1, S_1)$, p is either a core point in S_1 or a point outside of S_1. If p is a core point in S_1, then according to Lemma 5 $C_1 \cup C_2$ is a cluster w.r.t. the space constraint $S_1 \cup S_2$, Eps and $MinPts$. If p is outside of S_1, then p must be in S_2 and $p \in MC(C_2, S_2)$. Therefore, p will be a core point in S_2. According to Lemma 5, we have proven the lemma. □

References

Agrawal, R. and Shafer, J.C. 1996. Parallel mining of association rules: design, implementation, and experience. IBM Research Report.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The R*-tree: an efficient and robust access method for points and rectangles. Proc. ACM SIGMOD Int. Conf. on Management of Data. Atlantic City, NJ, pp. 322–331.

Berchtold S., Keim D.A., and Kriegel, H.-P. 1996. The X-tree: an index structure for high-dimensional data. Proc. 22nd Int. Conf. on Very Large Data Bases, Bombay, India, Morgan Kaufmann, pp. 28–39.

Bially, T. 1969. Space-filling curves: their generation and their application to bandwidth reduction. IEEE Trans. on Information Theory, IT-15(6):658–664.

Cheung, D.W., Han, J., Ng, V.T., Fu, A.W., and Fu, Y. 1996. A fast distributed algorithm for mining association rules. Proc. Int. Conf. on Parallel and Distributed Information System (PDIS'96). Miami Beach, FL, USA.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, OR, pp. 226–231.

Ester, M., Kriegel, H.-P., and Xu, X. 1995. A database interface for clustering in large spatial databases. Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining. Montreal, Canada, 1995, pp. 94–99.

Faloutsos, C. and Roseman, S. 1989. Fractals for secondary key retrieval. Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 247–252.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. 1996. Knowledge discovery and data mining: towards a unifying framework. Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. Portland, OR, pp. 82–88.

Geist, A., Beguelin, A., Dongama, J., Jiang, W., Manchek, R., and Sunderam, V. 1996. PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing. Cambridge, MA, London, England: The MIT Press, 3rd printing.

Gueting, R.H. 1994. An introduction to spatial database systems. The VLDB Journal, 3(4):357–399.

Jaja, J. 1992. An Introduction to Parallel Algorithms. Addison-Wesley Publishing Company, pp. 61–65.

Kamel, I. and Faloutsos, C. 1993. On packing R-trees. Proc. 2nd Int. Conf. on Information and Knowledge Management (CIKM).

Li, X. and Fang, Z. 1989. Parallel clustering algorithms. Parallel Computing, 11:275–290.

Matheus, C.J., Chan, P.K., and Piatetsky-Shapiro, G. 1993. Systems for knowledge discovery in databases. IEEE Transactions on Knowledge and Data Engineering, 5(6):903–913.

Mehta, M. and DeWitt, D.J. 1997. Data placement in shared-nothing parallel database systems. VLDB Journal, 6:53-72.

Olson, C.F. 1995. Parallel algorithms for hierarchical clustering. Parallel Computing, 21(8):1313–1325.

Park, J.-S., Chen, M.-S., and Yu, P.S. 1995. An effective hash based algorithm for mining association rules. Proc. ACM SIGMOD Int. Conf. on Management of Data. San Jose, CA, pp.175–186.

Pfitzner, D.W., Salmon, J.K., and Sterling, T. 1998. Halo World: Tools for Parallel Cluster Finding in Astrophysical N-body Simulations. Data Mining and Knowledge Discovery. Kluwer Academic Publishers, Vol. 2, No. 2, pp. 419-438.

Rasmussen, E.M. and Willett, P. 1989. Efficiency ofhierarchical agglomerative clustering using the ICL distributed array processor. Journal of Documentation, 45(1): 1–24.

Richards, A.J. 1983. Remote Sensing Digital Image Analysis. An Introduction, Berlin: Springer.

Sander, J., Ester, M., Kriegel, H.-P., and Xu, X. 1998. Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications. Data Mining and Knowledge Discovery, Kluwer Academic Publishers, vol. 2, pp. 1–27.

Stonebraker, M. 1986. The case for shared nothing. Database Eng., 9(1).

Stonebraker, M., Frew, J., Gardels, K., and Meredith, J. 1993. The SEQUOIA 2000 storage benchmark. Proc. ACM SIGMOD Int. Conf. on Management of Data. Washington, DC, pp. 2–11.

Xu, X. 1999. Efficient Clustering for Knowledge Discovery in Spatial Databases. Shaker, Germany: Aachen.

Xu, X., Ester, M., Kriegel, H.-P., and Sander, J. 1998. A distribution-based clustering algorithm for mining in large spatial databases. 14th Int. Conf. on Data Engineering (ICDE'98). Orlando, FL.

Zhang, T., Ramakrishnan, R., and Livny, M. 1998. BIRCH: A New Data Clustering Algorithm and Its Applications, Kluwer Academic Publishers, pp. 1-40,

Xiaowei Xu is a research scientist in the Siemens AG, Corporate Technology. His research interests are in data mining and knowledge discovery in databases, particularly in scalable data mining algorithms, parallel and distributed computing, and efficient data and index structures. He received his M.S. in 1987 from Shenyang Institute for Computing Technology, Chinese Academy of Sciences and his Ph.D. in 1998 from the University of Munich, Germany.

Jochen Jäger is a graduate student with the Institute for Computer Science at the University of Munich. His research interests include data mining, especially in biological data, parallel computing and efficient data and index structures.

Hans-Peter Kriegel is a full professor for database systems in the Institute for Computer Science at the University of Munich. His research interests are in spatial databases, particularly in query processing, performance issues, similarity search, high-dimensional indexing, and in parallel systems. Data Exploration using visualization led him to the area of knowledge discovery and data mining. Kriegel received his M.S. and Ph.D. in 1973 and 1976, respectively, from the University of Karlsruhe, Germany. Hans-Peter Kriegel has been chairman and program committee member in many international database conference. He has published over 150 refereed conference and journal papers.

Data Mining and Knowledge Discovery, 3,291–314 (1999)

Effect of Data Distribution in Parallel Mining of Associations

DAVID W. CHEUNG dcheung@csis.hku.hk
YONGQIAO XIAO
Department of Computer Science, The Universiry of Hong Kong, Hong Kong

Editors: Yike Guo and Robert Grossman

Abstract. Association rule mining is an important new problem in data mining. It has crucial applications in decision support and marketing strategy. We proposed an efficient parallel algorithm for mining association rules on a distributed share-nothing parallel system. Its efficiency is attributed to the incorporation of two powerful candidate set pruning techniques. The two techniques, distributed and global prunings, are sensitive to two data distribution characteristics: data skewness and workload balance. The prunings are very effective when both the skewness and balance are high. We have implemented FPM on an IBM SP2 parallel system. The performance studies show that FPM outperforms CD consistently, which is a parallel version of the representative Apriori algorithm (Agrawal and Srikant, 1994). Also, the results have validated our observation on the effectiveness of the two pruning techniques with respect to the data distribution characteristics. Furthermore, it shows that FPM has nice scalability and parallelism, which can be tuned for different business applications.

Keywords: association rules, data mining, data skewness, workload balance, parallel mining, parallel computing

1. Introduction

Association rule discovery has attracted a lot of attention from the research and business communities (Agrawal et al., 1993; Agrawal and Srikant, 1994; Brin et al., 1997). An association rule is a rule which implies certain association relationships among a set of objects (such as "occur together" or "one implies the other") in a database. The intuitive meaning of an association $X \Rightarrow Y$, where X and Y are set of items, is that transactions of the database which contain X tend to contain Y. A classical example is that 98% of customers that purchase tires and automobile accessories in a department store also have automotive services carried out. This example is a typical association in a basket database which sounds like common sense knowledge; however, there could be a lot of associations among the data which may not be able to deduce from common knowledge. Therefore, efficient automated technique to discover this type of rules is a very important area of research in data mining (Agrawal and Srikant, 1994; Cheung et al., 1996; Fayyad et al., 1995; Park et al., 1995a; Savasere et al., 1995). Applications of association rule mining range from decision support to product marketing and consumer behavior prediction.

Previous studies examined efficient mining of association rules from many different angles. An influential association rule mining algorithm, Apriori (Agrawal and Srikant, 1994), has been developed for rule mining in large transaction databases. The scope of the study has also been extended to efficient mining of sequential patterns (Srikant and Agrawal, 1996a), generalized association rules (Srikant and Agrawal, 1995), multiple-level association rules (Han and Fu, 1995), quantitative association rules (Srikant and Agrawal, 1996b), constrainted association rules (Ng et al., 1998) etc. Although these studies are on sequential data mining techniques, algorithms for parallel mining of association rules have also been proposed (Agrawal and Shafer, 1996; Park et al., 1995b; Shintani and Kitsuregawa, 1996; Zaki et al., 1996).

The development of parallel systems for mining of association rules has its unique importance—databases or data warehouses (Silberschatz et al., 1995) have been used more often to store a huge amount of data; data mining in such databases require substantial processing power, and parallel system is a possible solution. This observation motivates us to study efficient parallel algorithms for mining association rules in large databases. In this work, we study the problem on parallel system with distributed share-nothing memory such as the IBM SP2 (1995). In this model, the database is partitioned and distributed across the local disks of the processors; and the processors communicate via a fast network.

It has been well studied that the major cost of mining association rules is the computation of the set of *large itemsets* (i.e., *frequently occurring sets of items,* see Section 2.1) in the database (Agrawal et al., 1993; Agrawal and Srikant, 1994). An itemset (a set of items) is *large* if the percentage of transactions that containing all these items is greater than a given threshold. The most representative parallel algorithm for mining association rules is the CD algorithm (Count Distribution), which is designed for share-nothing parallel systems (Agrawal and Shafer, 1996). It extends directly the basic technique of Aprori to parallel system. Our proposed algorithm, FPM (Fast Parallel Mining), has the following distinct feature in comparison with CD: FPM has explored an important property between *locally large itemsets* (those that are large with respect to the partition of a processor) and *globally large itemsets* (those that are large with respect to the entire database) to develop two powerful pruning techniques, *distributed pruning* and *global pruning,* which can reduce the number of candidate sets at each individual processor. Since the number of candidate sets is a dominant parameter of the computation cost, with a substantially smaller candidate sets, FPM performs much better than CD.

Another contribution of this work is the discovery that the effectiveness of the two aforementioned pruning techniques, and hence the performance of the parallel mining, depends on the data distribution characteristics in the database partitioning. We have captured the distribution characteristics in two factors: *data skewness* and *workload balance.* These two factors are orthogonal properties. Intuitively, a partitioned database has high data skewness if most globally large itemsets are locally large only at a very few partitions. On the other hand, a partitioned database has a high workload balance if all the processors have similar number of locally large itemsets in their partitions. (More precise definitions of skewness and workload balance will be given in Sections 3 and 4.) We have defined metrics to measure data skewness and workload balance. We found out that both the distributed and global prunings have super performance in the best case of high data skewness and high workload balance. The combination of high balance with moderate skewness is the second best case. On the other hand, the high skewness, moderate balance combination only provide moderate improvement over CD, while the combination of low skewness and low balance is the worst case in which only marginal improvement can be found.

We have implemented FPM on an IBM SP2 parallel machine with 32 processors. Extensive performance studies have been carried out. The results confirm our observation on the relationship between pruning effectiveness and data distribution.

The rest of this paper is organized as follows. Section 2 overviews the parallel mining of association rules. The techniques of distributed and global prunings, together with the FPM algorithm are described in Section 3. In the same section, we have also investigated the relationship between the effectiveness of the prunings and the two data distribution characteristics of data skewness and workload balance. In Section 4, we define metrics to measure data skewness and workload balance of a data partition. Section 5 reports the result of an extensive performance study. In Section 6, we discuss a few issues including possible extensions of FPM to enhance its scalability. Section 7 is the conclusion.

2. Parallel mining of association rules

2.1. Sequential algorithmfor mining association rules

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of *items.* Let D be a database of transactions, where each transaction T consists of a set of items such that $T \subseteq I$. Given an *itemset* $X \subseteq I$, a transaction T *contains* X if and only if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \emptyset$ (Agrawal and Srikant, 1994). The association rule X $\Rightarrow$ Y holds in D with *confidence c* if the probability of a transaction in D which contains X also contains Y is C. The association rule $X \Rightarrow Y$ has *support s* in D if the probability of a transaction in D contains both X and Y is s. The task of mining association rules is to find all the association rules whose support is larger than a *minimum support threshold* and whose confidence is larger than a *minimum confidence threshold.*

For an itemset X, its *support* is the percentage of transactions in D which contains X, and its *support count,* denoted by $X_{.sup}$, is the number of transactions in D containing X. An itemset X is *large* (or more precisely, *frequently occurring)* if its support is no less than the minimum support threshold. An itemset of size k is called a *k-itemset.* It has been shown that the problem of mining association rules can be reduced to two subproblems (Agrawal et al., 1993): (1) *find all large itemsetsfor a given minimum support threshold,* and (2) *generate the association rulesfrom the large itemsetsfound.* Since (1) dominates the overall cost of mining association rules, the research has been focused on developing efficient methods to solve the first subproblem (Agrawal and Srikant, 1994).

An interesting serial algorithm, *Apriori* (Agrawal and Srikant, 1994), has been proposed for computing large itemsets at mining association rules in a transaction database, which is outlined as follows (Agrawal and Srikant, 1994).

> The large itemsets are computed through iterations. At each iteration, Apriori scans the database once and finds all the large itemsets of the same size. At the kth iteration, Apriori creates the set of candidate sets $C_{(k)}$ by applying the candidate set generating function *Apriori_gen* on $L_{(k-1)}$, where $L_{(k-1)}$ is the set of all large $(k-1)$-itemsets found at the $(k-1)$st iteration, and Apriori_gen generates only those k-itemset candidates whose every $(k-1)$-itemset subset is in $L_{(k-1)}$.

```
1) $C_k$ = apriori_gen$(L_{k-1})$;
2) scan partition $D_i$ to find the local support count $X_{.sup(i)}$ for all $X \in C_k$;
3) exchange $\{X_{.sup(i)} \mid X \in C_k\}$ with all other processors to get global support counts $X_{.sup}$, for all $X \in C_k$;
4) $L_k = \{X \in C_k \mid X_{.sup} \geq minsup \times |D|\}$
```

Figure 1. Count distribution algorithm.

2.2. Count distribution algorithm for parallel mining

CD (Count Distribution) is a parallel version of Apriori. It is one of the earliest proposed and representative parallel algorithms for mining of association rules (Agrawal and Shafer, 1996). We describe here briefly its steps for comparison purpose. The database D is partitioned into $D_1, D_2, \ldots, D_n$ and distributed across n processors. The program fragment of CD at processor p_i, $1 \leq i \leq n$, for the kth iteration is outlined in figure 1. (For convenience, we use $X_{.sup(i)}$ to represent the local support count of an itemset X in partition D_i.) In step 1, every processor computes the same candidate set C_k by applying the Apriori_gen function on L_{k-1}, which is the set of large itemsets found at the $(k-1)$th iteration. In step 2, local support counts (support in D_i) of candidates in C_k are found. In steps 3 and 4, local support counts are exchanged with all other processors to get global support counts (support in D) and globally large itemsets (large with respect to D) L_k are computed independently by each processor. CD repeats steps 14 until no more candidate is found. We have implemented CD on an IBM SP2 using the MPI (Message Passing Interface) (1994).

3. Pruning techniques and the FPM algorithm

3.1. Distributed pruning

It is important to observe some interesting properties related to large itemsets in a parallel environments since such properties may substantially reduce the number of candidate sets. (The preliminary form of the results in this section have been developed in Cheung et al., (1996) and extended here.) First, there is an important relationship between large itemsets and the processors in the database: *every globally large itemsets must be locally large at some processor(s).* If an itemset X is *both globally large and locally large* at a processor p_i, X is called gl-large at processor p_i. The set of gl-large itemsets at a processor will form a basis for the processor to generate its own candidate sets.

Second, a gl-large itemset at a processor has the following monotonic subset relationship property: *ifan itemset is gl-large at a processor p_i, all ofits subsets are also gl-large at p_i.* Combining these two properties, we have the following results.

Lemma 1. *Ifan itemset X is globally large, there exists aprocessor pi,* $(1 \leq i \leq n)$, *such that X and all its subsets are gl-large atprocessor p_i.*[1]

We use GL_i to denote the set of gl-large itemsets at processor p_i, and $GL_{i(k)}$ to denote the set of gl-large k-itemsets at processor p_i. It follows from Lemma 1 that if $X \in L_{(k)}$, then there exists a processor p_i, such that all its size-$(k-1)$ subsets are gl-large at processor p_i, i.e., they belong to $GL_{i(k-1)}$.

In a straightforward adaptation of Apriori, the set of candidate sets at the *kth* iteration, denoted by $CA_{(k)}$, which stands for size-k candidate sets from Apriori, would be generated by applying the Apriori_gen function on $L_{(k-1)}$. That is,

$$CA_{(k)} = \text{Apriori\_gen}(L_{(k-l)}).$$

At each processor p_i, let $CG_{i(k)}$ be the set of candidates sets generated by applying Apriori_gen on $GL_{i(k-1)}$, i.e.,

$$CG_{i(k)} = \text{Apriori\_gen}(GL_{i(k-l)}),$$

where CG stands for candidate sets generated from gl-large itemsets. Hence $CG_{i(k)}$ is generated from $GL_{i(k-1)}$. Since $GL_{i(k-1)} \subseteq L_{(k-1)}$, $C_{Gi(k)}$ is a subset of $CA_{(k)}$. In the following, we use $CG_{(k)}$ to denote the set $\cup_{i=1}^{n} CG_{i(k)}$.

Theorem 1. *For every $k > 1$, the set ofall large k-itemsets $L_{(k)}$ is a subset of $CG_{(k)} = \bigcup_{i=1}^{n} CG_{i(k)}$, where $CG_{i(k)}$ = Apriori_gen ($GL_{i(k-1,)}$.*

Proof: Let $X \in L_{(k)}$. It follows from Lemma 1 that there exists aprocessor p_i, $(1 \geq i \geq n)$, such that all the *size-*$(k-1)$ subsets of X are gl-large at processor p_i. Hence $X \in CG_{i(k)}$. Therefore,

$$L_{(k)} \subseteq CG_{(k)} = \bigcup_{i=1}^{n} CG_{i(k)} = \bigcup_{i=1}^{n} \text{Apriori\_gen}(GL_{i(k-1)}) \qquad \square$$

Theorem 1 indicates that $CG_{(k)}$, which is a subset of $CA_{(k)}$ and could be much smaller than $CA_{(k)}$, can be taken as the set of candidate sets for the size- k large itemsets. In effect, the set of candidates in $CA_{(k)}$ has been pruned down to those in $CG_{(k)}$—we called this technique *distributed pruning.* This result forms a basis for the reduction of the set of candidate sets in the algorithm FPM. First the set of candidate sets $CG_{i(k)}$ can be generated locally at each processor p_i at the kth iteration. After the exchange of support counts, the gl-large itemsets $GL_{i(k)}$ in $CG_{i(k)}$ can be found at the end of that iteration. Based on $GL_{i(k)}$, the candidate sets at processor p_i for the $(k+1)$st iteration can then be generated according to Theorem 1. According to our performance studies, the number of candidate sets generated by distributed pruning can be substantially reduced to about 10–25%of that generated in CD.

Example 1 illustrates the effectiveness of the reduction of candidate sets using distributed pruning.

Example 1. Assuming there are 3 processors in a parallel system in which the database D has been partitioned into D_1, D_2 and D_3, Suppose the set of large 1-itemsets (computed at the first iteration) $L_{(1)} = \{A, B, C, D, E, F, G, H\}$, in which A, B, and C are locally large at processor p_1, B, C, and D are locally large at processor p_2, and E, F, G, and H are locally large at processor p_3. Therefore, $GL_{1(1)} = \{A, B, C\}$, $GL_{2(1)} = \{B, C, D\}$, and $GL_{3(1)} = \{E, F, G, H\}$.

Based on Theorem 1, the set of size-2 candidate sets at processor p_1 is $CG_{1(2)} =$ Apriori_gen $(GL_{1(1)}) = \{AB, BC, AC\}$. Similarly, $CG_{2(2)} = \{BC, CD, BD\}$, and $CG_{3(2)} = \{EF, EG, EH, FG, FH, GH\}$. Hence, the set of candidate sets for large 2-itemsets is $CG_{(2)} = CG_{1(2)} \cup CG_{2(2)} \cup CG_{3(2)}$, total 11 candidates.

However, if Apriori_gen is applied to $L_{(1)}$, the set of candidate sets $CA_{(2)} =$ Apriori_gen $(L_{(1)})$ would have 28 candidates. This shows that it is very effective to apply distributed pruning to reduce the candidate sets.

3.2. Global pruning

As a result of the count exchange, the local support counts $X_{.sup(i)}$, for all processor pi, $(1 \leq i \leq n)$, are also available at every processor. With this information, another powerful pruning technique called *global pruning* can be developed. Let X be a candidate *k-itemset.* At each partition D_i, $X_{.sup(i)} \leq Y_{.sup(i)}$, if $Y. \subset X$. Therefore the local support count of X, $X_{.sup(i)}$, is bounded by the value $min[Y_{.sup(i)} \mid Y$ c X, and $|Y| = k - 1\}$. Since the global support count of X, $X_{.sup}$, is the sum of its local support count at all the processors, the value

$$X_{.\mathrm{maxsup}} = \sum_{i=1}^{n} X_{.\mathrm{maxsup}(i)},$$

where

$$X_{.maxsup(i)} = \min\{Y_{.sup(i)} \mid Y \subset X, \text{ and } |Y| = k - 1\},$$

is an upper bound of $X_{.sup}$. If $X_{.maxsup} <$ minsup x $|D|$, then X can be pruned away. This technique is called *global pruning.* Note that global pruning requires no additional information except the local support counts resulted from count exchange in the previous iteration.

Table 1 gives an example to show that global pruning can pruning away candidates which cannot be pruned by distributed pruning. Suppose the global support count threshold is 15 and the local support count threshold at each processor is 5. Distributed pruning cannot prune away CD, as C and D are both gl-large at processor 2. Whereas global pruning can Prune away CD, as $CD_{maxsup} = CD_{.maxsup(1)} + CD_{.maxsup(2)} + CD_{.maxsup(3)} = 1 + 12 + 1 = 14 < 15$. EF can also be pruned, because $EF_{.maxsup} = 1 + 1 + 12 = 13 < 15$. However, AB would survive global pruning. From this example, it is clear that global pruning is more effective than distributed pruning, i.e., what can pruned away by distributed pruning will be pruned away by global pruning. The three pruning techniques, the one in apriori_gen, the distributed and global prunings, have increasing pruning power, and the latter ones subsume the previous one.

Table1. High data skewness and high workload balance case.

Items	A	B	C	D	E	F
Localsupport atprocessor 1	13	33	1	2	2	1
Localsupport atprocessor2	3	3	12	34	1	4
Local support atprocessor 3	2	3	2	1	12	33
Global support	18	39	15	37	15	38
gl-large at processor 1	✓	✓	×	×	×	×
gl-large at processor 2	×	×	✓	✓	×	×
gl-large at processor 3	×	×	×	×	✓	✓

```
1) compute candidate sets $CG_k = \cup_{i=1}^{n}$ apriori_gen($GL_{k-1(i)}$); (distributed pruning)
2) prune candidates in $CG_k$ by global pruning;
3) scan partition $D_i$ to find the local support count $X.sup_{(i)}$ for all remaining candidates $X \in CG_k$;
4) exchange {$X_{sup(i)}$ 1 $X \in CG_k$} with all other processors to get global support counts $X.sup$, for all $X \in CG_k$;
5) compute $GL_{k(i)} = \{X \in CG_k \mid X.sup \geq$ minsup x $|D|$, $X.sup_{(i)} \geq$ minsup x $|D_i|\}$, for all i, $1 \leq i \leq n$;
6) return $L_k = \cup_{i=1}^{n} GL_{k(i)}$.
```

Figure 2. The FPM algorithm.

3.3. *Fast parallel mining algorithm (FPM)*

We present the FPM algorithm in this section. It is an enhancement of CD. The simple support counts exchange scheme in CD is retained in FPM. The main difference is the incorporation of both the distributed and global prunings in FPM to reduce the candidate set size.

The first iteration of FPM is the same as CD. Each processor scans its partition to find out local support counts of all size-I itemsets and use one round of count exchange to compute the global support counts. At the end of the 1st iteration, in addition to L_1, each processor also finds out the gl-large itemsets $GL_{1(i)}$, for $1 \leq i \leq n$.

For the kth iteration of FPM, $k > 1$, the program fragment executed at processor i, $1 \leq i \leq n$, is described in figure 2.

Similar to CD, FPM is also implemented by collective communication operations of MPI on the SP2. In order to compare the effects of distributed and global pruning, we have also implemented a variant FNG (FPM with no global pruning) of FPM. FNG does not perform the global pruning, i.e., it's procedure is the same as that in figure 2, except step 2 is removed.

4. Data skewness and workload balance

In a database partition, two data distribution characteristics, *data skewness* and *workload balance,* have orthogonal effects on prunings and hence performance of FPM.

Intuitively, the data skewness of a partitioned database is high if the supports of most large itemsets are clustered in a few partitions. It is low if the supports of most large itemsets are distributed evenly across the processors. In Table 1, it is clear that all the itemsets have high skewness.

For a partition with high skewness, even though the support of each large itemset is clustered at a few processors, the clusterings of different large itemsets may be distributed evenly across the processors or concentrated on a few of them. In the first case, the clusterings of the large itemsets are distributed evenly among the processors; hence, each processor would have similar number of locally large itemsets. We characterise this case as high workload balance. In the second case, the clusterings would be concentrated on a few processors; hence some processors would have much more locally large itemsets than the others. This is the low workload balance case. For example, the itemsets in Table 1 not only have high skewness, it also has a good workload balance; because *A, B* are locally large at processor 1, and *C, D* at processor 2, whereas *E, F* at processor 3.

It follows from our discussion of the pruning techniques that high data skewness would increase the chance of candidate set pruning; however, it is not the only factor, workload is another critical factor. In the following, we will see that given a good data skewness, if the distribution of the clusterings amount the processors are not even, then the pruning effects would be reduced significantly, and, to aggravate the problem more, the work of computing the large itemsets would be concentrated on a few processors which is a very troublesome issue for parallel computation.

Example 2. As explained above, Table 1 is a case of high data skewness and high workload balance. The supports of each itemset are distributed mostly in one partition; hence, the skewness is high. On the other hand, every partition has the same number of locally large itemsets; therefore, the workload balance is also high. In this case, CD will generate $\binom{6}{2} = 15$ candidates in the second iteration. Whereas, the distributed pruning will generate only three candidates *AB, CD* and *EF,* which shows that the pruning has good effect for this distribution.

Table 2 is an example of high data skewness but low workload balance. The thresholds are the same as that in Table 1, i.e., the global support threshold is 15 and the local support threshold at each processor is 5. The support count distribution of each item is the same as

Table2. High data skewness and low workload balance case.

Items	A	B	C	D	E	F
Localsupport atprocessor 1	13	33	12	34	2	1
Local support at processor 2	1	3	1	2	1	4
Localsupportat processor3	2	1	2	1	12	33
Global support	16	37	15	37	15	38
gl-large at processor 1	✓	✓	✓	✓	×	×
gl-large at processor 2	×	×	×	×	×	×
gl-large at processor 3	×	×	×	×	✓	✓

Table3. Low data skewness and high workload balance case.

Items	*A*	*B*	*C*	*D*	*E*	*F*
Localsupportatprocessor 1	6	12	4	13	5	12
Local support at processor 2	6	12	5	12	4	13
Local support at processor 3	4	13	6	12	6	13
Global support	16	37	15	37	15	38
gl-large at processor 1	√	√	×	√	√	√
gl-large at processor 2	√	√	√	√	×	√
gl-large at processor 3	×	√	√	√	√	√

that in Table 1 except that items *A, B, C* and *D* are now locally large together at processor 1 instead of distributed between processors 1 and 2. In this lower workload balance case, distributed pruning will generate 7 size-2 candidates, namely *AB, AC, AD, BC, BD, CD* and *EF,* while CD will still have 15 candidates. Thus, the distributed pruning remains to be very effective, but not as good as that in the high workload balance case (Table 1).

Table 3 is an example of low data skewness and high workload balance. The support counts of the items *A, B, C, D, E* and *F* are almost equally distributed over the 3 processors. Hence, the data skewness is low. On the other hand, the workload balance is high, because the number of locally large itemsets in each processor is almost the same. In this case, both CD and distributed pruning generate the same 15 candidate sets; hence, if we restrict pruning to the distributed pruning, then it has no advantage over CD in this case. However, global pruning can prune away the candidates *AC, AE* and *CE.* In other words, FPM still has a 20% of improvement over CD in this pathological case of low skewness and high balance.

Following Example 2, it is observed that global pruning is more effective than distributed pruning and can perform significant candidates reduction even in the moderate data skewness or low workload balance cases. As a note, low skewness and low balance cannot occur together. Also, according to our analysis, distributed pruning can prune away almost $\frac{n-1}{n}$ (*n* is the number of partitions) of all the size-2 candidates generated by CD in the high data skewness and high workload balance case.

In summary, distributed pruning is very effective when a database is partitioned with high skewness and high balance. On the other hand, in the worst cases of high skewness with low balance or high balance with low skewness, the effect of distributed pruning is degraded to the level in CD, however, global pruning may still perform better than CD. To strengthen our studies, we investigated the problem of defining metrics to measure skewness and balance.

4.1. Data skewness metric

We have developed a skewness metric based on the well established notion of entropy (Cover and Thomas, 1991). Given a random variable *X,* it's entropy is a measurement on how even or uneven its probability distribution is over its values. If a database is partitioned over n processors, the value $px(i) = \frac{X_{.sup(i)}}{X_{.sup}}$ can be regarded as the probability of occurrence of an itemset X in partition D_i, $(1 \leq i \leq$ n). The entropy $H(X) = -\sum_{i=1}^{n}(p_X(i)$ x $\log(px(i)))$ is a measurement of the distribution of the local supports of X over the partitions. For example, if X is skewed completely into a single partition D_k, $(1 \leq k \leq$ n), i.e., it only occurs in D_k, then $px(k) = 1$ and $px(i) = 0, \forall i \neq k$. The value of $H(X) = 0$ is the minimal in this case. On the other hand, if X is evenly distributed among all the partitions, then $px(i) = \frac{1}{n}, 1 \leq i \leq n$, and the value of $H(X) = \log(n)$ is the maximal in this case. Therefore the following metric can be used to measure the skewness of a data partition.

Definition 1. Given a database partition D_i, $(1 \leq i \leq n)$, the skewness $S(X)$ of an itemset is defined by $S(X) = \frac{H_{\max} - H(X)}{H_{\max}}$, where $H(X) = -\sum_{i=1}^{n}(p_X(i)$ x $\log(px(i)))$ and $H_{\max} = \log(n)$.

The skewness $S(X)$ has the following properties:

- $S(X) = 0$, when all $px(i)$, $1 \leq i \leq n$, are equal. So the skewness is at its lowest value when X is distributed evenly in all partitions.
- $S(X) = 1$, if $\exists k \in [1, n]$ such that $px(k) = 1$, and $px(i) = 0$ for $\forall i \neq k$, $1 \leq i \leq n$. So the skewness is at its highest value when X occurs only in one partition.
- $0 < S(X) < 1$, in all the other cases.

It follows from the property of entropy that $S(X)$ increases with respect to the skewness of X; hence, it is a suitable metric for the skewness of an individual itemset. Table 4 shows the skewness of the large itemsets in Tables 1–3.

In addition to measuring the skewness of an itemset, we also need a metric to measure the skewness of the database partition. We define the skewness of a database partition as a weighted sum of the skewness of all the large itemsets. In other words, the skewness of a partition is a measurement of the total skewness of all the large itemsets.

Definition 2. Given a database partition D_i, $(1 \leq i \leq n)$, the skewness $TS(D)$ of the partition is defined by $TS(D) = \sum_{X \in L_S}$ S(X) x $w(X)$, where L_s is the set of all the large itemsets, $w(X) = \frac{X_{.sup}}{\sum_{Y \in L_S} Y_{.sup}}$ is the weight of the support of X over all the large itemsets in L_s, and $S(X)$ is the skewness of X.

$TS(D)$ has some properties similar to $S(X)$.

- $TS(D) = 0$, when the skewness of all the itemsets are at its minimal value.
- $TS(D) = 1$, when the skewness of all the itemsets are at its maximal value.
- $0 < TS(D) < 1$, in all the other cases.

In Table 4, the skewness $TS(D)$ of the partitions for the three situations have computed. (For illustration purpose, we only have computed $TS(D)$ with respect to the skewness of all the size-1 large itemsets.) Note that we have ignored the small itemsets in the computation of the skewness of a partition. Since the purpose of our task is to investigate the effect of data

Table4. Data skewness and workload balance.

	Itemset	A	B	C	D	E	F	Workload W_i
Table 1 High data skewness, high workload balance	Local count at processor 1	13	33	1	2	2	1	0.329
	Local count at processor 2	1	3	12	34	1	4	0.348
	Local count at processor 3	2	1	2	1	12	33	0.322
	S(X)	0.452	0.633	0.429	0.697	0.429	0.586	
	TS(D)				0.494			
	TB(D)				0.999			
Table 2 High data skewness, low workload balance	Local count at processor 1	13	33	12	34	2	1	0.601
	Local count at processor 2	1	3	1	2	1	4	0.076
	Local count at processor 3	2	1	2	1	12	33	0.323
	S(X)	0.452	0.633	0.429	0.697	0.429	0.586	
	TS(D)				0.494			
	TB(D)				0.789			
Table 3 Low data skewness, high workload balance	Local count at processor 1	6	12	4	13	5	12	0.329
	Local count at processor 2	6	12	5	12	4	13	0.329
	Local count at processor 3	4	13	6	12	6	13	0.342
	S(X)	0.015	0.001	0.012	0.001	0.012	0,001	
	TS(D)				0.005			
	TB(D)				0.999			

skewness on candidate sets pruning, and this only involves large itemsets, this restriction would in fact make the metric more relevant to candidate set pruning.

4.2. Workload balance metric

Workload balance is a measurement of the distribution of the support clusterings of the large itemsets over the partitions at the processors. Based on the definition of *w(X)* in Definition 2 and that of *px(i)* in Definition 1, we define $W_i = \sum_{X \in L_s} w(X)\ x\ px(i)$ to be the *itemset workload* in a partition $D_{i,}$ where *Ls* is the set of all the large itemsets. Intuitively, the workload W_i in partition D_i is the ratio of the total supports of the large itemsets in D_i over all the partitions. Note that $\sum_{i=1}^{n} W_i = 1$.

A partition has high workload balance if W_i are the same for all partitions D_i, $1 \leq i \leq n$. On the other hand, if distribution of W_i over the partitions are very uneven, then the workload balance is low. As has been pointed out, the workload balance has important bearing on the pruning and performance of parallel mining. In parallel to the metric for data skewness, we also define a metric workload balance factor to measure the workload balance of a partition, which is based also on entropy.

Definition 3. For a database partition D_i, $1 \leq i \leq n$, of a database D, the workload balance factor $TB(D)$ of the partition is given by $TB(D) = \frac{-\sum_{i=1}^{n} W_i \log(W_i)}{\log(n)}$.

The metric $TB(D)$ has the following properties:

- $TB(D) = 1$, when the workload across all processors are the same;
- $TB(D) = 0$, when the workload is concentrated on one processor;
- $0 < TB(D) < 1$, in all the other cases.

In Table 4, the workload W_i of the first and last cases (Tables 1 and 3) have a high balance, and the values of $TB(D)$ are almost equal to 1. In the second case (Table 2), the workload at processor 2 has been shifted to processor 1, and hence created an unbalance case; the value of $TB(D)$ thus has been reduced to 0.789, which indicates a moderate workload balance.

The data skewness metric and workload balance factor are not independent. Theoretically, each one of them could have values range from 0 and 1. However, some combinations of their values are not admissable. First, let us consider some boundary cases.

Theorem 2. Let $D_1, D_2, \ldots, D$, be a partition of a database D.

1. If $TS(D) = 1$, then the admissable values of $TB(D)$ range from 0 to 1. If $TS(D) = 0$, then $TB(D) = 1$.
2. If $TB(D) = 1$, then the admissable values of $TS(D)$ range from 0 to 1. If $TB(D) = 0$, then $TS(D) = 1$.

Proof:

1. By definition $0 \leq TB(D) \leq 1$. What we need to prove is that the boundary cases are admissable when $TS(D) = 1$. $TS(D) = 1$ implies that $S(X) = 1$, for all large itemsets X. Therefore, each large itemset is large at one and only one processor. If all the large itemsets are large at the same processor i, ($1 \leq i \leq$ n), then $W_i = 1$ and $W_k = 0$, ($1 \leq k \leq n, k \neq i$). Thus $TB(D) = 0$ is admissable. On the other hand, if each processor has the same number of large itemsets, then $W_i = \frac{1}{n}$, ($1 \leq i \leq n$), and $TB(D) = 1$. Furthermore, if $TS(D) = 0$, then $S(X) = 0$ for all large itemsets X. This implies that W_i are the same for all $1 \leq i \leq n$. Hence $TB(D) = 1$.
2. It follows from the first result of this theorem that both $TS(D) = 0$ and $TS(D) = 1$ are admissable when $TB(D) = 1$. Therefore the first part is proved. Furthermore, if $TB(D) = 0$, there exists a partition D_i, $1 \leq i \leq n$, such that $W_i = 1$ and $W_k = 0$, ($1 \leq k \leq n, k \neq i$). This implies that all large itemsets are locally large at only D_i. Hence $TS(D) = 1$. □

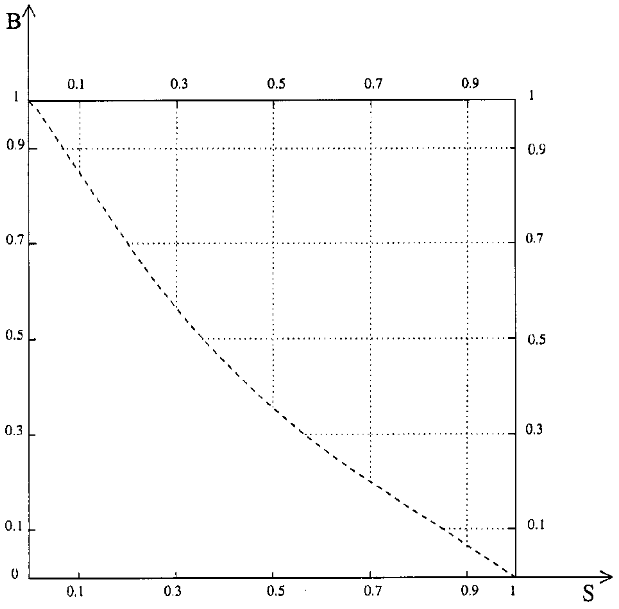


Figure 3. Admissable combinations of skewness *(S)* and balance *(B)*.

Even though, 0 ≤ *TS(D)* ≤ 1 and 0 ≤ *TB(D)* ≤ 1, we have shown in Theorem 2 that not all possible combinations are admissable. In general, the admissable combinations will be a subset of the unit square such as the one in figure 3. It always contains the two segments *TS(D)* = 1 (*S* = 1 in figure 3) and *TB(D)* = 1 (*B* = 1 in figure 3), but not the origin, (*S* = 0, *B* = 0). After defining the metrics and studying their characteristics, we can validate our observation on the relationship between data skewness, workload balance and candidates pruning effect in our performance studies.

5. Performance studies

In order to confirm our analysis that the proposed FPM is an efficient algorithm for mining associations in a parallel system, we have implemented all the algorithms on an IBM SP2 and carried out a substantial performance evaluation and comparison.

We have the following three goals in our studies: (1) to verify that FPM is faster than the representative algorithm CD, and confirm that the major performance gain is from the two pruning techniques; (2) to confirm the observation that both data skewness and workload balance are two critical factors in the performance of FPM; (3) to demonstrate that FPM has good parallelism and scalability as a parallel algorithm.

Table 5. Synthetic database parameters.

D	Number of transactions in each partition
T	Average size of the transactions
1	Average size of the maximal potentially large itemsets
L	Number of maximal potentially large itemsets
N	Number of items
S	Partition skewness
B	Workload balance
n	Number of partitions

Table6. Attributes of synthetic databases.

	TS(D)						*TB(D)*					
Name	B100	B90	B70	B50	B30	B10	B100	B90	B70	B50	B30	B10
D3278K.T5.12.S90	0.86	0.85	0.86	0.87	0.85	0.88	0.99	0.88	0.69	0.47	0.29	0.08
D3278K.T5.12.S70	0.74	0.72	0.71	0.72	0.74	-	0.98	0.87	0.68	0.48	0.27	-
D3278K.T5.12.S50	0.46	0.45	0.47	0.47	-	-	0.98	0.88	0.67	0.47	-	-
D3278K.T5.12.S30	0.24	0.26	0.25	-	-	-	0.99	0.87	0.66	-	-	-
D3278K.T5.12.S10	0.07	0.08	-	-	-	-	0.99	0.92	-	-	-	-

We implemented FPM, its variant FNG, and CD. The IBM SP2 parallel system we used has 32 POWER2 processors (66.7 MHz) with 64 MB main memory, running the AIX operating system. Communication between processors are through a high performance switch with an aggregated peak bandwidth of 40 MBps and a latency of about 40 microseconds. Data was allocated to the local disk in each processor, and the database partition on each node is about 100 MB in size.

In order to be able to control the experiments to test different data distributions and scenarios, many works (Agrawal and Srikant, 1994; Agrawal and Shafer, 1996; Han and Fu, 1995; Park et al., 1995a; Park et al., 1995b) in mining association rules have adopted the standard technique introduced in (Agrawal and Srikant, 1994) to generate the database. We have enhanced the technique for the generation of database partitions and introduced parameters to control the skewness and workload balance. Table 5 is a list of the parameters used in our synthetic databases. Details of the data generation technique is in the appendix.

5.1. Relative performance

In order to compare the performance between FMP, FNG, and CD, a databases and twenty data sets have been generated. The data sets generated and their skewness and balance factors are listed in Table 6. The number of partitions in each case is 16 ($n = 16$), and the size of each

Table7. Performance improvement of FPM and FNG over CD.

Response Time	FPM/CD						FNG/CD					
Ratio	B100	B90	B70	B50	B30	B10	B100	B90	B70	B50	B30	B10
D3278K.T5.12.S90	2.10	1.69	1.36	1.23	1.14	1.06	1.79	1.35	1.17	1.10	1.05	1.01
D3278K.T5.12.S70	2.07	1.41	1.23	1.13	1.06	-	1.43	1.17	1.08	1.05	1.01	-
D3278K.T5.12.S50	1.88	1.22	1.11	1.06	-	-	1.25	1.10	1.05	1.01	-	-
D3278K.T5.12.S30	1.55	1.17	1.08	-	-	-	1.08	1.06	1.01	-	-	-
D3278K.T5.12.S10	1.36	1.09	-	-	-	-	1.03	1.02	-	-	-	-

partition is about 100 MB. The name of a partition is denoted by Dx.Ty.Iz.Sr.Bl, where x is the number of transactions in each partitions, y is the average size of the transactions, z is the average size of the itemsets. The two parameters Sr and B1 are two important parameters used to control the skewness and workload balance in the data generation. (The B1 values are listed separately in the table.) In Table 6, we have also computed the measured skewness *TS(D)* and the balance factor *TB(D)* of the partitions generated. It is important to note that these measured skewness and workload are very close to the values of the controlled parameters, i.e., the values of S and B are good approximations of values of *TS(D)* and *TB(D)*. In addition, they cover a wide range of skewness and balanace. Thus, our synthesized data partitions are good simulation of data partitions of various distribution characteristics. We believe this is technically valuable because even real data may not be general enough to cover all possible distributions.

We ran FPM, FNG and CD on the database partitions in Table 6. The minimum support threshold is 0.5%. The improvement of FPM and FNG over CD in response time are recorded in Table 7, and the result is very encouraging. FPM and FNG are consistently faster than CD in all cases. In the following, we analyze the performance gain of FPM and FNG in three aspects: (1) improvement when the workload balance is high, and the skewness varies from high to moderate; (2) improvement when the skewness is high, and the workload balance varies from high to moderate; (3) desirable and undesirable combinations of skewness and workload balance values.

Figure 4 is the relative performance between FPM, FNG and CD on partitions with different skewness and a high balance value $(B = 100)$. FNG performs much better than CD when the skewness is relatively high $(s > 0.5)$. On the other hand, FPM outperforms CD significantly even when the skewness is in the moderate range, $(0.1 \leq s \leq 0.5)$. When $B = 90$, the result in Table 7 shows that FPM is again much faster than CD. This demonstrates that FPM outperforms CD consistently given a high workload balance and at least a moderate skewness.

Figure 5 is the relative performance given a high skewness $(S = 90)$ and different workload balance. Both FPM and FNG perform much better than CD when the workload balance is relatively high $(B > 0.5)$; however, the improvement in the range of moderate balance, $(0.1 \leq B \leq 0.5)$, is marginal. This confirms our observation that workload balance is an essential requirement. It shows that a high skewness has to accompany by a high workload

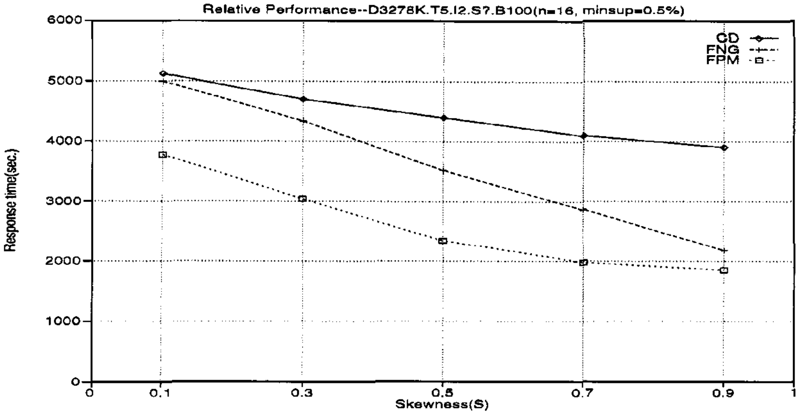


Figure 4. Relative performance on databases with high balance and different skewness.

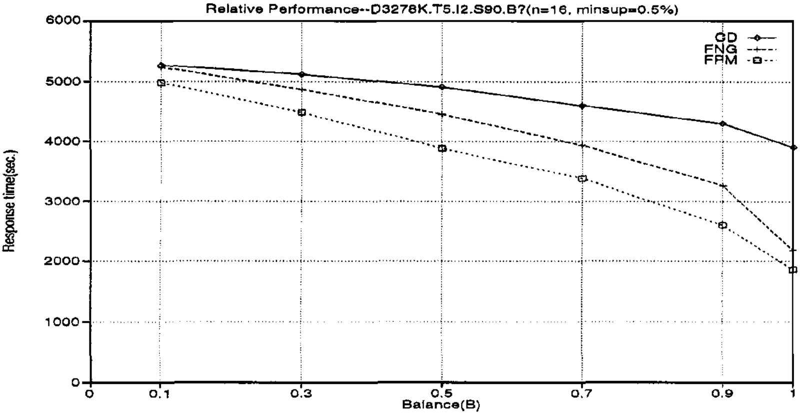


Figure 5. Relative performance on databases with high skewness and different workload balance.

balance. The effect of a high skewness with a moderate balance may not be as good as that of a high balance with a moderate skewness.

In figure 6, we vary both the skewness and balance together from a low value range to a high value range. The trend shows that the improvement of FPM over CD increases faster when both values approach the high value range.

Combining the observations in the above three cases together with the results in Table 7, we can divide the area covering all the admissable combinations of skewness and balance in our experiments into four performance regions as shown in figure 7. Region A is the most favorable region in which the balance is high and the skewness varies from high to

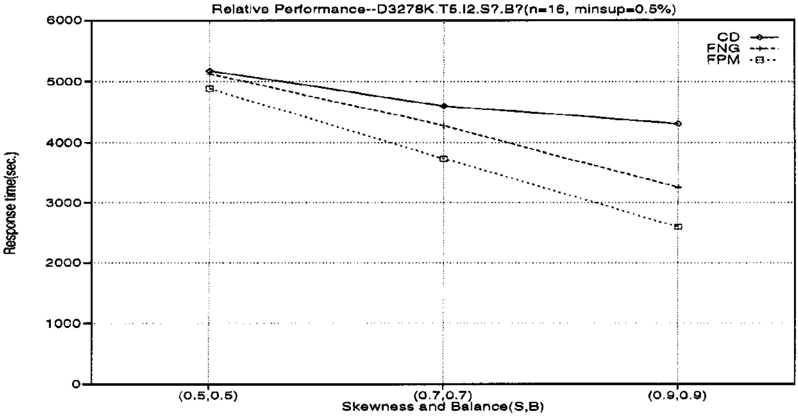


Figure 6. Relative performance on databases when both skewness and balance change.

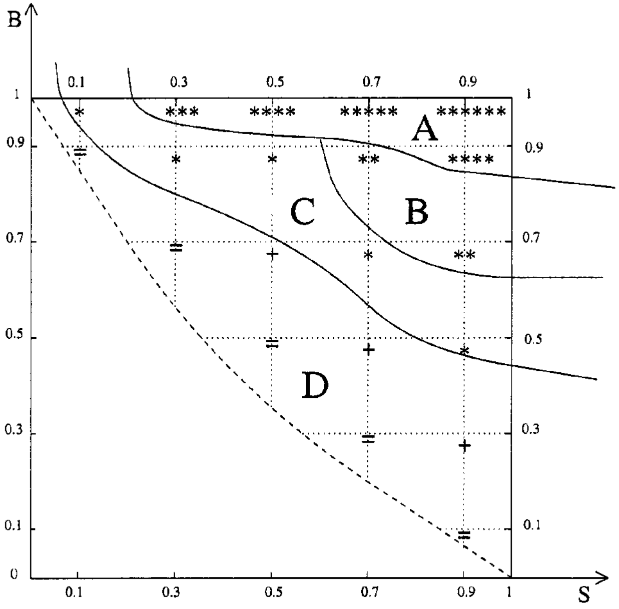


Figure 7. Performance regions (FPM/CD) in the admissible combinations of skewness and workload balance.

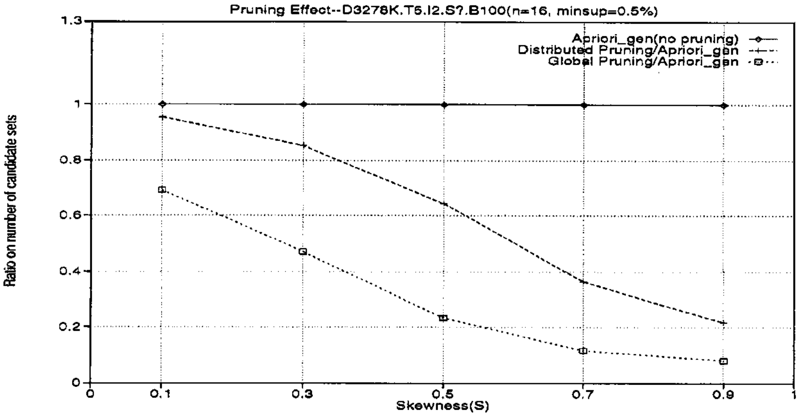


Figure 8. Pruning effect on databases in figure 4.

moderate. FPM in general is 50 to 100% faster than CD. In region B, the workload balance value has degraded moderately and the skewness remains high; in this case, the gain in FPM over CD is around 50%. Region C covers combinations that have very low workload balance; the gain in FPM falls into a moderate range of about 30%. Region D contains the most undesirable combinations; FPM only has marginal performance gain.

Figure 8 provides us another view to understand the candidates pruning effects. It shows the ratio on the number of candidate sets between FPM (FNG) and CD for the same experiments in figure 4. The reduction ratios for the runs in the database D3278K.T5.12.Sr.B100, *(r* = 90,70, 50, 30, 10), are in the first graph. When the skewness is high, *(s* = 0.9), distributed pruning has a 79.2% of reduction in candidate sets comparing with CD, and global pruning has a 93.9% reduction. When the skewness is low, *(s* = 0.1), distributed pruning only has a 6.6% reduction, but global pruning has a 30.7% reduction. This confirms our observation on the effect of high balance combined with high or moderate skewness.

5.2. Scalability andparallelism: Speedup and scaleup

In order to study the efficiency ofFPM as aparallel algorithm, we investigate its *speedup* and *scaleup* against CD. Speedup is the reduction in response time vs. the number ofprocessors, given that the total size of the database remains unchanged. The more processors are used, the faster the computation should be. The ideal speedup is a linear function on the number of processors. Scaleup is the performance vs. the number ofprocessors when the database size is scaled up proportional to the number ofproccesors. Ifthe algorithm has high efficiency and low overhead, its performance would maintain uniform when both the number ofprocessors and the size of the database scaled up proportionally.

In the speedup experiment, we execute the algorithms on a fixed size database with various number of processors and partitions. We selected the database with high skewness

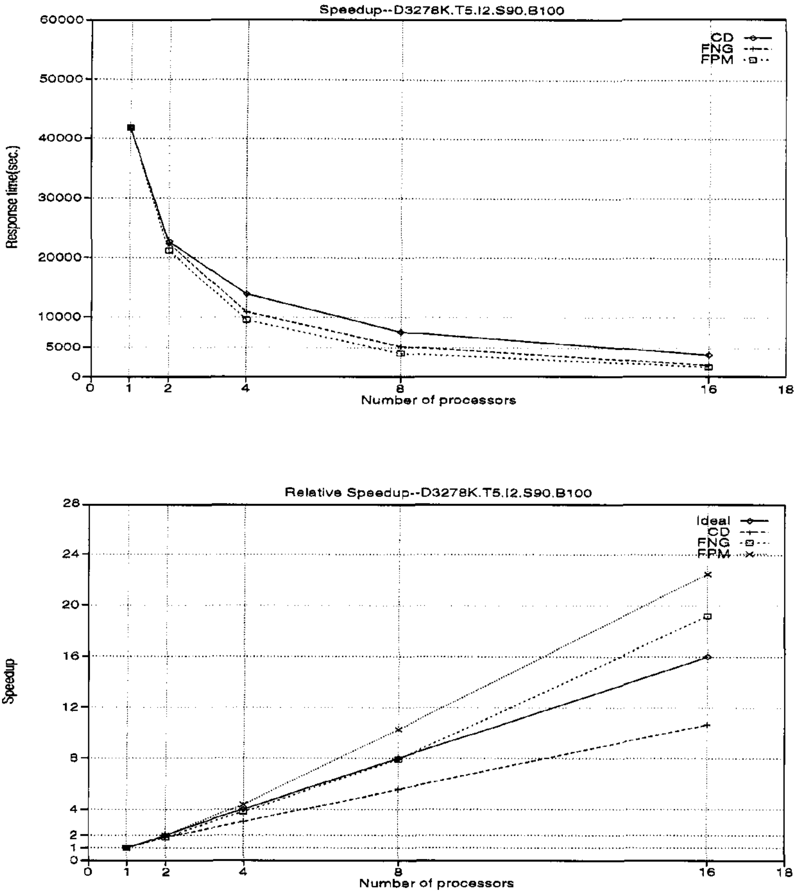


Figure 9. Speedup on a Database *(S* = 90, *B* = 100).

and balance as a representative to perform the study. The database is listed in Table 8. It has a total size of 1.6 GB, and was first divided into 16 partitions. Subsequently, we combined the partitions to form databases with 8, 4, 2, and zero partitions.

Figure 9 is the execution times and speedups of FPM, FNG, and CD on the databases. The speedups are also shown in Table 8. FNG had a linear speedup and FPM achieved a remarkable superlinear speedup. The reason behind FPM’s superlinear speedup is the increase in the skewness when the number of partitions increases.

In the scaleup experiment, both the database size and the number of processors are scaled up proportionally. The number of processors involved were increased from 1 to 16, and the sizes of the databases were increased correspondingly from 100 MB to 1.6 GB. Each

Table 8. Speedup on five databases with different distribution characteristics.

Databases	Speedup of FPM				SpeedupofFNG				Speedup of CD			
	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 2$	$n = 4$	$n = 8$	$n = 16$
D3278K.T5.12.S90,B100	1.97	4.37	10.29	22.50	1.85	3.79	7.91	19.19	1.85	3.01	5.54	10.70

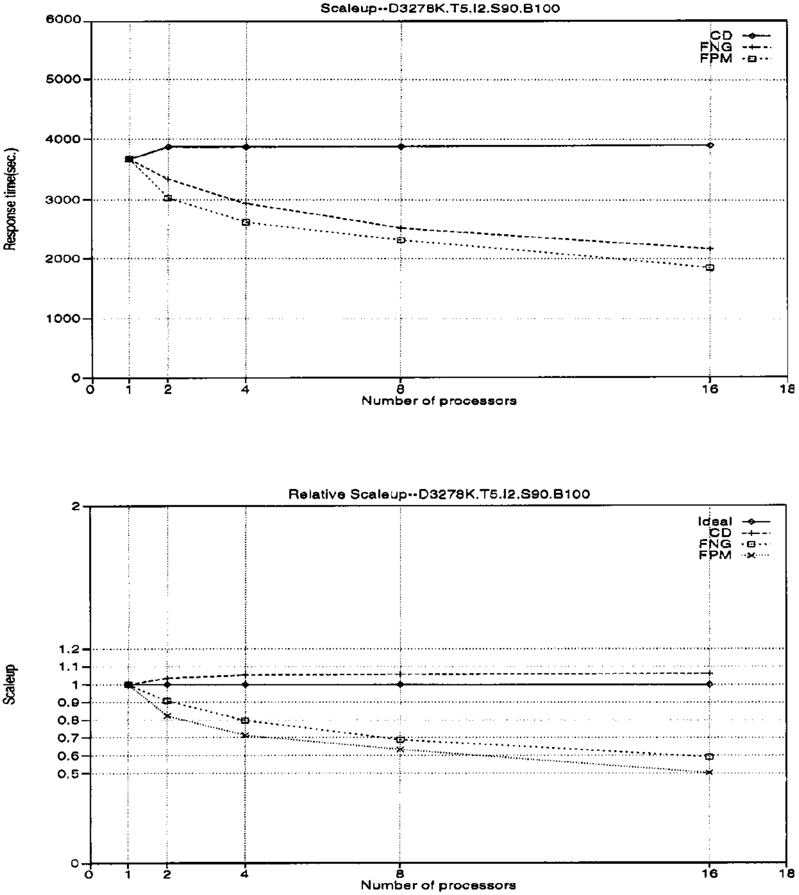


Figure 10. Scaleup on a database $(S = 90, B = 100)$.

database were partitioned according to the number of processors such that every partition is maintained at the size of 100 MB. We performed the experiment based on the database D3278K.T5.12.S90.B100, i.e., the databases are generated with the same parameters. Figure 10 shows the result of the experiment. Surprisingly, both FPM and FNG not only can maintain the performance, their response time in fact had gone down when the database was scaled up. The prime reason for this is the increase in pruing capability when the number of partitions increases.

6. Discussion

To restrict the search of large itemsets in a small set of candidates is essential to the performance of mining association rules. The pruning techniques we proposed are theoretically interesting, because they have effect only in the parallel case but not the serial case. Both distributed and global pruning provide significant amount of pruning power, in particular, when the data distribution is in a favorable situation, i.e. when workload balance is high and the skewness is at least at a moderate level. It is important to study partition techniques that can deliver a good data distribution. Random approaches in general will deliver partitions which have good balance. However, the skewness would be difficult to guarantee together with good workload balance. Clustering technique such as the *k*-means algorithm (MacQueen, 1967). will give good skewness. It remains an open problem how to modify clustering technique to generate partitions which have good skewness and also good workload balance.

7. Conclusion

A parallel algorithm FPM for mining association rules has been proposed. A performance study carried out on an IBM SP2 shared-nothing memory parallel system shows that FPM consistently outperforms CD. It also has nice scalability in terms of speedup and scaleup. The gain in performance in FPM is due mainly to the pruning techniques incorporated. It was discovered that the effectiveness of the pruning techniques depend highly on the data distribution characteristics, which can be measured by two metrics: data skewness and workload balance. Our analysis and experiment results show that the pruning techniques are very sensitive to workload balance, though good skewness will also have important positive effect. The techniques are very effective in the best case of high balance and high skewness. The combination of high balance and moderate skewness is the second best case. In the worst case of low balance and low skewness, FPM can only deliver the performance close to that of CD. Since mining associations has many interesting applications, important future works would include fine tuning of the proposed parallel techniques on real business cases.

Appendix

Synthetic databases generation

The synthetic databases used in our experiments are generated using similar techniques introduced in (Agrawal and Srikant, 1994). We have enhanced it to generate data partitions and introduced two parameters to control the skewness and workload balance. Table 5 is a list of the parameters used in our synthetic databases.

The synthetic database partitions are generated from a pool of potentially large itemsets. The first step is to generate the relative weights of these large itemsets. These weights are then broken down into smaller weights with respect to the partitions. Therefore, every itemset in the pool has *n* weights associated with it. Each one of these weights corresponds to the probability of occurrence of the itemset in a partition. The weight of each itemset in the pool is picked from an exponential distribution with unit mean. A skewness level s for the itemset is then drawn from a normal distribution with mean *S* and variance 0.1. Following that, *n* probability values from an exponential distribution with variance equal to s are picked. These *n* values are normalize so that their sum equals to 1. The skewness of these *n* probability values are computed according to the skewness metric in Definition 1. We repeat this random process until a set of *n* values, whose skewness falls into the permitted range of *s* ± 0.02, is generated. These *n* probability values are then randomly assigned to the *n* partitions. Eventually, the weight of the itemset is broken down into *n* weights by multiplying it with then probability values. We repeat this process to generate the weights of all the itemsets and their breakdowns. If we use these generated weights as the distribution of the support counts of the itemsets, a high workload balance among the partitions will be resulted because of the randomness in the process. In order to control the balance factor, we redistribute the weights among the partitions starting from this high balance configuration. Firstly, we randomly determine a nondescending order for the workloads of the partitions. We will reshuffle the weights among the partitions according to this order. Secondly, we pick an itemset from the pool randomly, shuffle its *n* weights among the partitions such that they will be ordered according to the determined workload order. We check the workload balance according to Definition 3, and repeat the process until the balance value is in the permitted range of *B* ± 0.02. Since the balance value starts at the highest value, it will converge into the expected range. At the end of this step, both the balance value and the workloads of all the partitions are determined.

The second step is to generate the itemsets in the pool. We first divide the *N* items into *n* disjoint ranges whose lengths are proportional to the workloads of the corresponding partitions. In order to control the skewness of the itemsets, we regard the *ith* $(1 \leq i \leq n)$ probability values generated for an itemset in the previous step as the probability of choosing items from the ith range to put into the itemset. For an itemset whose weights have been determined in the previous step, we first determine the ranges in which the items are picked. These ranges are determined by tossing a *n*- side weighted coin, where the weight of side *i* is the ith probability of the *n* probability values of the itemset. Once a range has been determined, an item is picked randomly in it. Repeating this procedure, items are picked until the number is equal to the size of the itemset. Some items of the subsequent itemsets are copied from the previous itemset according to the correlation level as in (Agrawal and Srikant, 1994), while the remaining items are picked in the same way as in the first itemset.

The last step is to generate the transactions in all the partitions. The technique follows primary the one introduced in (Agrawal and Srikant, 1994) with the following modification. After we shuffled the weights to control the workload balance factor, the workload will be different at different partitions. We add a dummy weight to each one of those partitions whose workload is less than the maximum workload to make the workload of all of them equal. The dummy weight corresponds to the weights of the small itemsets in each partition. Therefore, it won't affect the true workload balance. In generating the transactions for partition *i*, $(1 \leq i \leq n)$, we normalize the ith weights of all the itemsets so that their sum equals to 1 and use the normalized *ith* weight as the probability of occurrence of the associated itemset in partition i. Each transaction in the partition is assigned a series of large itemsets, which are chosen by tossing an $(L + 1)$-side weighted coin, (1 extra side is for the dummy weight), where the weight for a side is the ith weight of the associated itemset. In the special case that a dummy weight is selected, a dummy which corresponds to small itemsets will be inserted into the transactions. Since the dummy represents small itemsets, it won't be counted into the large itemsets found.

Note

1. This result is stronger than that in (Savasere et al., 1995) — the result there states that a globally large itemset is locally large in some partition; while Lemma 1 states that all its subsets must be locally large together at the same partition.

References

Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining association rules between sets of items in large databases. Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data. pp. 207–216.

Agrawal, R. and Shafer, J.C. 1996. Parallel mining of association rules: Design, implementation and experience. Special Issue in Data Mining, IEEE Trans. on Knowledge and Data Engineering, IEEE Computer Society, 8(6):962–969.

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. Proc. 1994 Int. Conf. Very Large Data Bases. Santiago. Chile, pp. 487-499.

Brin, S., Motwani, R., Ullman, J., and Tsur, S. 1997. Dynamic itemset counting and implication rules for market basket data. Proc. of 1997 ACM-SIGMOD Int. Conf. On Management of Data. Tucson, Arizona, pp. 255–264.

Cheung, D.W., Han, J., Ng, V.T., Fu, A.W., and Fu. Y. 1996. A fast distributed algorithm for mining association rules. Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems. Miami Beach, FL, pp. 31–43.

Cheung, D.W., Han, J., Ng, V.T., and Wong, C.Y. 1996. Maintenance of discovered association rules in large databases: An incremental updating technique. Proc. 1996 IEEE Int. Conf. on Data Engineering. New Orleans, Louisiana.

Cover T.M. and Thomas, T.A. 1991. Elements of Information Theory. John Wiley & Sons.

Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R. 1995. Advances in Knowledge Discovery and Data Mining. AAAI/MIT Press.

Han J. and Fu, Y. 1995. Discovery of multiple-level association rules from large databases. Proc. 1995 Int. Conf. Very Large Data Bases. Zurich, Switzerland, pp. 420-431.

Han, E., Karypis G., and Kumar, V. 1997. Scalable parallel data mining for association rules. Proc. of 1997 ACM-SIGMOD Int. Conf. On Management of Data.

Int'l Business Machines. 1995. Scalable POWERparallel Systems, GA23-2475-02 edition.

MacQueen, J.B. 1967. Some methods for classification and analysis of multivariate observations. Proceedings of the 5th Berkeley symposium on mathematical statistics and probability, pp. 281–297.

Message Passing Interface Forum. 1994. MPI: A Message-Passing Interface Standard.

Ng, R., Lakshmanan, L., Han J., and Pang, A. 1998. Exploratory mining and pruning optimizations of constrainted association rules. Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data. Seattle, WH.

Park, J.S., Chen, M.S., and Yu, P.S. 1995a. An effective hash-based algorithm for mining association rules. Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data. San Jose, CA, pp. 175–186.

Park, J.S., Chen, M.S., and Yu, P.S. 1995b. Efficient parallel data mining for association rules. Proc. 1995 Int. Conf. on Information and Knowledge Management. Baltimore, MD.

Savasere, A., Omiecinski, E., and Navathe, S. 1995. An efficient algorithm for mining association rules in large databases. Proc. 1995 Int. Conf. Very Large Data Bases. Zurich, Switzerland, pp. 432-444.

Shintani, T. and Kitsuregawa, M. 1996. Hash based parallel algorithms for mining association rules. Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems.

Silberschatz, A., Stonebraker, M., and Ullman, J. 1995. Database research: achievements and opportunities into the 21st century. Report of an NSF Workshop on the Future of Database Systems Research.

Srikant R. and Agrawal, R. 1995. Mining generalized association rules. Proc. 1995 Int. Conf. Very Large Data Bases. Zurich, Switzerland, pp. 407–419.

Srikant R. and Agrawal, R. 1996a. Mining sequential patterns: Generalizations and performance improvements. Proc. of the 5th Int. Conf. on Extending Database Technology. Avignon, France.

Srikant R. and Agrawal, R. 1996b. Mining quantitative association rules in large relational tables. Proc. 1996 ACM-SIGMOD Int. Conf. on Management of Data. Montreal, Canada.

Zaki, M.J., Ogihara, M., Parthasarathy, S., and Li, W. 1996. Parallel data mining for association rules on shared-memory multi-processors. Supercomputing’96, Pittsburg, PA, Nov. 17–22.

David Wai-lok Cheung received the M.Sc. and Ph.D. degrees in computer science from Simon Fraser University, Canada, in 1985 and 1989, respectively. He also received the B.Sc. degree in mathematics from the Chinese University of Hong Kong. From 1989 to 1993, he was with Bell Northern Research, Canada, where he was a member of the scientific staff. Since 1994, Dr. Cheung has been faculty member of the department of computer science and information systems in The University of Hong Kong. His research interest includes data mining, data warehousing, Web databases, multimedia databases and database concurrency control. Dr. Cheung has served as program committee members in numerous international conferences including VLDB’97, VLDB’99, ICDE’99, KDD’97, DASFAA’99, PAKDD’97, PAKDD’98, and PAKDD’99.

Yongqiao Xiao was a Ph.D. student at The University of Hong Kong.

Data Mining and Knowledge Discovery, 3, 315–339 (1999)

Parallel Learning of Belief Networks in Large and Difficult Domains

Y. XIANG* yxiang@cs.uregina.ca
Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2

T. CHU
Avant Corporation, Sunnyvale, CA, USA

Editors: Yike Guo and Robert Grossman

Abstract. Learning belief networks from large domains can be expensive even with single-link lookahead search (SLLS). Since a SLLS cannot learn correctly in a class of problem domains, multi-link lookahead search (MLLS) is needed which further increases the computational complexity. In our experiment, learning in some difficult domains over more than a dozen variables took days. In this paper, we study how to use parallelism to speed up SLLS for learning in large domains and to tackle the increased complexity of MLLS for learning in difficult domains. We propose a natural decomposition of the learning task for parallel processing. We investigate two strategies for job allocation among processors to further improve load balancing and efficiency of the parallel system. For learning from very large datasets, we present a regrouping of the available processors such that slow data access through the file system can be replaced by fast memory access. Experimental results in a distributed memory MIMD computer demonstrate the effectiveness of the proposed algorithms.

Keywords: belief networks, parallel implementation of data mining

1. Introduction

Probabilistic belief networks (Pearl, 1988; Jensen, 1996) have been widely used for inference with uncertain knowledge in artificial intelligence. As an alternative to elicitation from domain experts, learning belief networks from data has been actively studied (Cooper and Herskovits, 1992; Heckerman et al., 1995; Herskovits and Cooper, 1990; Lam and Bacchus, 1994; Spirtes and Glymour, 1991; Xiang et al., 1997). Since the task is NP-hard in general (Chickering et al., 1995), it is justified to use heuristics in learning. Many algorithms developed use a scoring metric combined with a single-link lookahead search (SLLS), where alternative network structures differing from the current structure by *one* link are evaluated exhaustively before one of them is adopted. Although the complexity is polynomial on the number of variables of the problem domain, the computation is still expensive for large domains. Furthermore, a class of domain models termed pseudo-independent (PI) models cannot be learned correctly by a SLLS (Xiang et al., 1997). One alternative is to use a multi-link lookahead search (MLLS) (Xiang et al., 1997), where consecutive structures

*Author to whom all correspondence should be addressed.

differ by multiple links. However, the complexity of a MLLS is higher. In our experiment (Section 11), learning a 35 variable PI domain model (containing two small PI submodels) took about two and half days, and learning a 16 variable PI domain model (containing a slightly larger PI submodel) took about 25 days.

In this paper, we study parallel learning to speed up computation during SLLS in large domains and to tackle the increased complexity during MLLS in potential PI domains. We focus on learning decomposable Markov networks (DMNs) (Xiang et al., 1997) and show that the lessons we learned are applicable to learning Bayesian networks (BNs) (Pearl, 1998). To the best of our knowledge, this is the first investigation on parallel learning of belief networks. As learning graphical probabilistic models has become an important subarea in data mining and knowledge discovery, this work extends parallel data mining to learning these models. We focus on multiple instruction multiple data (MIMD) distributed-memory architecture for it is available to us, and we discuss the generalization of our lessons to other architectures.

The paper is organized as follows: To make it self-contained, we briefly introduce PI models and MLLS in Sections 2 and 3. In Sections 4 through 9, we propose parallel algorithms for learning DMNs and their refinements. We present experimental results in Sections 10 and 1 1. Graph-theoretic terms unfamiliar to some readers and a list of frequently used acronyms are included in Appendix.

2. Pseudo-independent models

Let N be a set of discrete variables in a problem domain. A *tuple* of $N^1 \subseteq N$ is an assignment of values to every variable in N^1. *Aprobabilistic domain model* (PDM) over N determines the probability of every tuple of N^1 for each $N^1 \subseteq N$. For disjoint sets X, Y and Z of variables, X and Y are *conditionally independent* given Z if $P(X \mid Y, Z) = P(X/Z)$ *whenever* $P(Y, Z) > 0$, which we shall denote by $I(X, Z, Y)$. If $Z = ø$, X and Y are *marginally independent,* denoted by $I(X, ø, Y)$.

Table 1 shows a PDM over four binary variables. The PDM satisfies $I(u, \{v, x\}, y)$. In the subset $\{v, x, y\}$, each pair is marginally dependent, e.g., $P(v, x) \neq P(v)P(x)$, and is dependent given the third, e.g., $P(v/x, y) \neq P(u/y)$. However in the subset $\{u, v, x\}$, although each pair is dependent given the third, e.g., $P(u/v, x) \neq .P(u/v)$, we have $I(u, ø, v)$ and $I(u, ø, x)$. Hence u, v and x are said to be *collectively dependent* even though u and v are marginally independent (so are u and x). This PDM is a PI model. In general, a PI model is a PDM where proper subsets of a set of collectively dependent variables display marginal independence (Xiang et al., 1997). Example PI models include *parity* and *modulus addition*

Table 1. A PI model.

(u, v, x, y)	*P(N)*	*(u, v, x, y)*	*P(N)*	*(u, v, x, Y)*	*P(N)*	*(u, v, x, y)*	*P(N)*
(0,0,0,0)	0.0225	(0,1,0,0)	0.0175	(1,0,0,0)	0.02	(1,1,0,0)	0.035
(0,0,0,1)	0.2025	(0,1,0,1)	0.0075	(1,0,0,1)	0.18	(1,1,0,1)	0.015
(0,0,1,0)	0.005	(0,1,1,0)	0.135	(1,0,1,0)	0.01	(1,1,1,0)	0.12
(0,0,1,1)	0.02	(0,1,1,1)	0.09	(1,0,1,1)	0.04	(1,1,1,1)	0.08

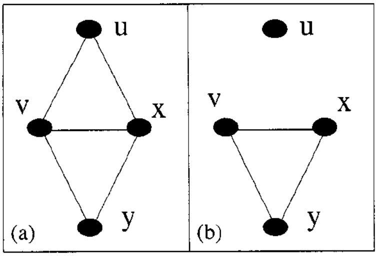


Figure 1. (a) Minimal I-map of PDM in Table 1. (b) Network structure learned by a SLLS.

problems (Xiang et al., 1997). PI models have also been found in real datasets. Analysis of data' from 1993 General Social Survey (conducted by Statistics Canada) on Personal Risk has discovered two PI models, one on *harmful drinking* and the other on *accidentprevention* (Hu, 1997).

For disjoint subsets X, Y and Z of nodes in an undirected graph G, we use $<X|Z|Y>_G$ to denote that nodes in Z intercept all paths between X and Y. A graph G is an *I-map* of a PDM over N if there is an one-to-one correspondence between nodes of G and variables in N such that for all disjoint subsets X, Y and Z of N, $<X|Z|Y>_G \Longrightarrow I(X, Z, Y)$. G is a *minimal* I-map if no link can be removed such that the resultant graph is still an I-map. The minimal I-map of the above PDM is shown in figure 1(a).

Several algorithms for learning belief networks have been shown being unable to learn correctly when the underlying PDM is PI (Xiang et al., 1996). Suppose learning starts with an empty graph (with all nodes but without any link). A SLLS will not connect u and v since $I(u, \emptyset, v)$. Neither will u and x be connected. This results in the learned structure in figure 1(b), which is incorrect. On the other hand, if we perform a double link search after the single-link search, which can effectively test whether $P(u/u, x) = P(u/v)$ holds, then the answer will be negative and the two links (u,v) and (u, x) will be added. The structure in figure 1(a) will be learned.

3. A sequential MLLS algorithm

The parallel learning algorithms presented in the paper are based on the sequential MLLS algorithm Seq (Xiang et al. 1997), which learns the structure (a chordal graph) of a DMN using K-L cross entropy (Kullback and Leibler, 1951) as scoring metric. Once the structure is learned, numerical parameters can be easily estimated from the same dataset. Search is organized into *levels* (the outer*for* loop) and the number of lookahead links is identical in the same level. Each level consists of multiple *pusses* (the *repeat* loop). In each pass at the same level, alternative structures that differ from the current structure by the same number i of links are evaluated. Search at each pass selects i links that decrease the cross entropy maximally after evaluating all distinct and valid combinations of i links. If the corresponding entropy decrement is significant, the i links will be adopted and the next pass at the same level starts. Otherwise, the first pass at the next higher level starts.

Algorithm 1 (Seq).

```
Input: A dataset D over a set N ofvariables, a maximum size n ofclique, a
        maximum number $K \leq n(n-1)/2$ oflookahead links, and a threshold $\delta h$.
begin
    initialize an empty graph $G = (N, E)$, $G^1 := G$;
  fori = 1 to K, do
        repeat
            initialize the entropy decrement $dh^1 := 0$;
          for each set L ofi links ($L \cap E = \emptyset$), do
                ifG* = $(N, E \cup L)$ is chordal and
                L is implied by a single clique ofsize $\leq n$, then
                    compute the entropy decrement $dh^*$;
                    ifdh* > $dh^1$, then $dh^1 := dh^*$, $G^1 := G^*$;
            ifdh$^1 > \delta h$, then $G := G^1$,done :=false; else done := true;
        until done = true;
    return G;
end
```

Note that each intermediate graph is chordal as indicated by the *if* statement in the innermost loop. The condition that *L* is implied by a single clique *C* means that all links in *L* are contained in the subgraph induced by *C.* It helps reduce search space. Note also that the algorithm is greedy while the learning problem is NP-hard. Hence, a link committed early in the search is not necessarily contained in a corresponding minimal I-map.

Figure 2 illustrates Seq with a dataset over variables *{u, v, x, y}.* A SLLS is performed for simplicity. Search starts with an empty graph in (a). Six alternative graphs in (b) through

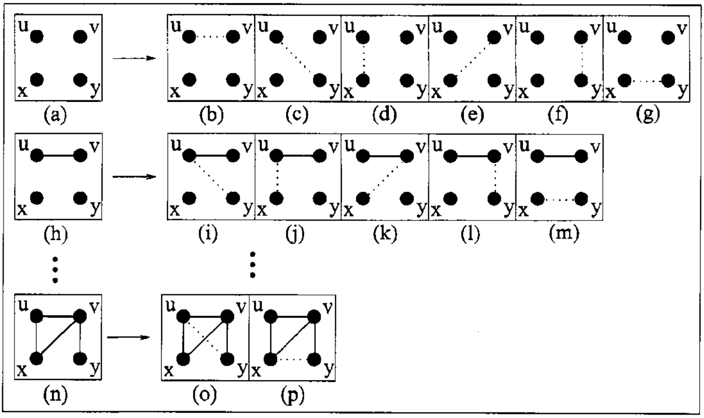


Figure 2. An example of sequential learning.

(g) are evaluated before, say, (b) is selected. The next pass starts with (b) as the current strucuture (redrawn as (h)) and graphs in (i) through (m) are evaluated. Repeating the above process, suppose eventually the graph in (n) is obtained. In the last pass, suppose none of the graphs in (o) and (p) decreases the cross entropy significantly. Then the graph in (n) will be the final result.

4. Task decomposition for parallel learning

In algorithm Seq, for each pass at level 1, $O(|N|^2)$ structures are evaluated before a link is added. $O(|N|^{2m})$ structures are evaluated before m links are added in a pass at level m. To tackle the complexity of MLLS and to speed up SLLS in large domains, we explore parallelism. To this end, we decompose the learning task based on the following observation: At each pass of search, the exploration of alternative structures are coupled only through the current structure. Given the current structure, evaluation of alternative structures are independent, and hence the evaluation can be performed in parallel.

As mentioned earlier, this study is performed using an architecture where processors communicate through message passing (vs. shared memory) only. We partition the processors as follows: One processor is designated as the search *manager* and the others are structure *explorers.* The manager executes Mgr1 (Algorithm 2). For each pass, it generates alternative graphs based on the current graph. It then partitions them into n sets and distributes one set to each explorer.

Algorithm 2 (Mgr1).

```
Input: N,D,n,K, δh as algorithm Seq, and the total number n of explorers.
begin
  send N, D and n to each explorer;
  initialize an empty graph G = (N, E), G' := G;
  for i = 1 to K, do
    repeat
      initialize the cross entropy decrement dh' := 0;
      partition all graphs that differ from G by i links into n sets;
      send one set of graphs and G to each explorer;
      for each explorer
        receive dh* and G*;
        if dh* > dh' then dh' := dh*, G' :=G*;
      if dh' > δh, then G := G', done := false; else done := true;
    until done = true;
  send a halt signal to each explorer;
  return G;
end
```

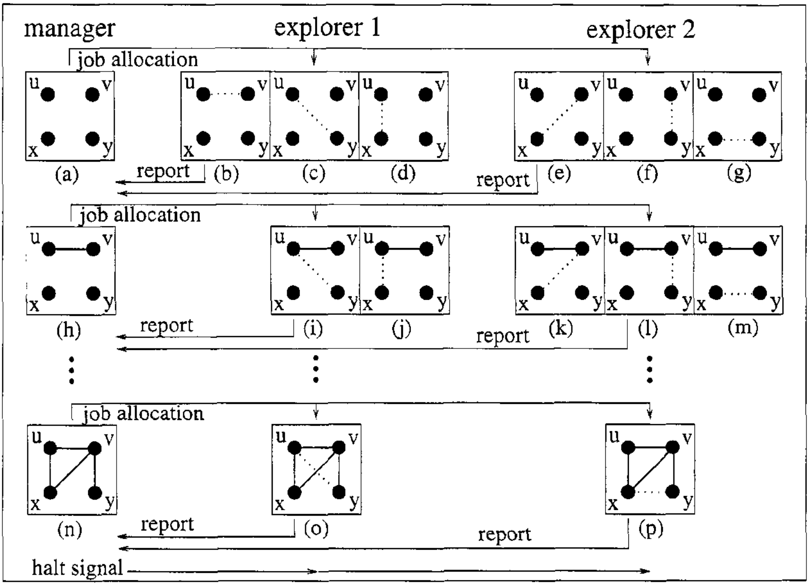


Figure 3. An example of parallel learning.

Each explorer executes Epr1. It checks chordality for each graph received and computes *dh** for each chordal graph. It then chooses the best graph *G** and reports *dh** and *G** to manager. Manager collects the reported graphs from all explorers, selects the global best, and then starts the next pass of search.

Figure 3 illustrates the parallel learning with two explorers and a dataset over variables *{u, v, x, y}*. A SLLS is performed for simplicity. Manager starts with an empty graph in (a).

Algorithm 3 (Epr1).

```
begin
    receive N, D and nfrom the manager:
    repeat
        receive G = (N, E) and a set ofgraphsfrom the manager;
        initialize dh* := 0 and G* := G;
        for each received graph G' = (N, L ∪ E), do
            ifG' is chordal and L is implied by a single clique ofsize ≤ n, then compute dh';
            ifdh' > dh*, then dh* := dh', G* := G';
        send dh* and G* to the manager;
    until halt signal is received;
end
```

It sends six alternative graphs in (b) through (g) to explorers 1 and 2. Explorer 1 checks graphs in (b), (c) and (d). Suppose the one in (b) is selected and reported to manager. Suppose explorer 2 reports the one in (e), After collecting the two graphs, manager chooses the one in (b) as the new current graph. It then sends graphs in (i) through (m). Repeating the above process, manager finally gets the graph in (n) and sends graphs in (o) and (p) to explorers. Suppose none of them decreases the cross entropy significantly. Then manager chooses the graph in (n) as the final result and terminates explorers.

5. Issue of load balancing

In algorithm Mgr1, alternative graphs are *evenly* allocated to explorers. However, the amount of computation in evaluating each graph tends to swing between two extremes. If a graph is non-chordal, it is discarded immediately without further computation. On the other hand, if a graph is chordal, its cross entropy decrement will be computed. Figure 4(a) shows an example graph. There are six supergraphs (graphs with *more* links) that differ by one link. If any of the dotted links in (b) is added to (a), the resultant graph is non-chordal. If any of the dashed links in (c) is added to (a), the resultant graph is chordal. Since the complexity of checking chordality is $O(|N| + |E|)$,where $|E|$ is the number of links in the graph, the amount of computation is very small. Since the complexity of computing cross entropy decrement is $O(|D| + \eta\ (\eta \log \eta + 2^{\eta}))$ (Xiang et al., 1997), where $|D|$ is the number of distinct tuples appearing in the dataset, the amount of computation is much greater. As a result, even job allocation may cause significant fluctuation among explorers in the amount of computation. As manager must collect reports from all explorers before the new current graph can be selected, some explorers will be idle while others are completing their jobs.

Figure 5 shows the time taken by each of six explorers in a particular pass in learning from a dataset over 37 variables, where a distributed memory MIMD computer was used. Explorer 1 took much longer than others did.

The above analysis implies that more sophisticated job allocation strategy is needed to improve the efficiency of the parallel system. In the following sections, we propose two strategies: multi-batch allocation and two-stage allocation.

6. Multi-batch allocation

Multi-batch allocation is based on the idea of keeping some jobs unallocated in the initial allocation and allocating them later to explorers who finish early. The multi-batch allocation problem can be abstracted as follows:

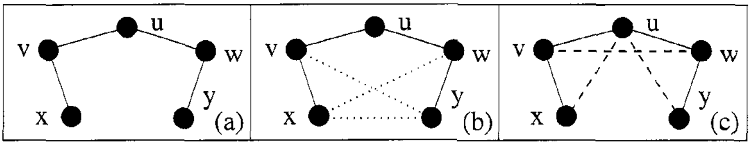


Figure 4. Chordal and nonchordal alternative structures.

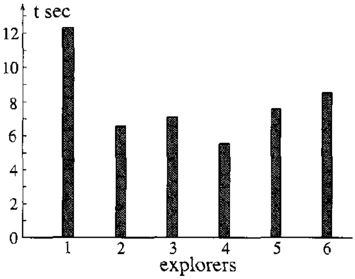


Figure 5. Job completion time of six explorers.

Let L_0 be the total number of job units, each of which corresponds to a graph to be evaluated. A job unit is either of type 0 (non-chordal) or of type 1 (chordal). It takes time T_0 to process a unit of type 0 job and T_1 for that of type 1. After an explorer has finished a given batch of job units, it takes time T_c to send another batch of job units (by one message) to the explorer. We shall refer to any batch sent to an explorer after the first batch as an *additional* batch. The goal is to find the proper size of each batch such that the sum of idle time of all explorers is reduced during the completion of L_0 job units.

In deriving the batch sizes, we make the following assumptions:

Assumption 1. T_0 and T_c are constants in a pass.

T_0 is the computation time to test the chordality of a graph. Since the complexity of checking chordality is $O(|N| + |E|)$, and each graph in the same pass has the identical number of nodes and links, T_0 can be treated as a constant.

T_c is the time for manager to send an additional batch to an explorer. An additional batch is much smaller (as will be seen) than the first batch. A message for an additional batch is thus very short. Messages are sent through communication channels (> 10M bps) within the parallel computer, and the actual data transfer is very fast. Consequently, T_c consists mainly of handshaking time and only varies slightly from message to message.

Assumption 2. T_1 is a constant in a pass and is much larger than T_0 and T_c.

T_1 is the computation time to process one unit of type 1 job which involves checking the chordality of a given graph and computing the cross entropy decrement of a chordal graph. It is much larger than T_0 and T_c. For example, in learning from a database with 37 variables, we found T_0 to be between 0.007 to 0.009 sec and T_c about 0.017 sec in our parallel computing environment. T_1 was at least 0.06 sec. However, the assumption that T_1 is a constant is less accurate. When the variation of clique sizes in a chordal graph is small, T_1 tends to be close to a constant. When the variation is large, T_1 tends to vary depending on specific job unit. Still, we found the assumption to be a useful approximation in deriving a simple method to determine the batch size.

Suppose the first batch allocated to each explorer has J_0 $(<L_0/n)$ units. Let Q_i (B_i) denote the number of type 1 (0) units in the batch assigned to explorer *i*. Let Q denote the

total number of type 1 units in the n batches. Let $\beta i = Q_i/J_0$ be the percentage of type 1 units in the batch to explorer i. Let $\beta = Q/(nJ_0)$ be the percentage of type 1 units in the n batches. Without losing generality, suppose $\beta_1 = \max_{i=1}^{n}(\beta_i)$ and we alternatively denote β_1 by β_{max}.

The time t_i taken by explorer i to process its first batch is

$$t_i = Q_i T_1 + B_i T_0 = \beta_i J_0 T_1 + (1 - \beta_i) J_0 T_0 = J_0(\beta_i(T_1 - T_0) + T_0). \tag{1}$$

Let T be the sum of the idle time of explorers 2 through n while explorer 1 is processing its first batch. We can derive

$$\begin{aligned} T &= \sum_{i=2}^{n}(t_1 - t_i) = \sum_{i=2}^{n} J_0((\beta_{\max}(T_1 - T_0) + T_0) - (\beta_i(T_1 - T_0) + T_0)) \\ &= \sum_{i=2}^{n} J_0\beta_{\max}(T_1 - T_0) - \sum_{i=2}^{n} J_0\beta_i(T_1 - T_0). \end{aligned} \tag{2}$$

Substituting $\sum_{i=2}^{n} J_0\beta_i = Q - Q_1 = nJ_0\beta - J_0\beta_{\max}$ in Eq. (2), we have

$$\begin{aligned} T &= (n - 1)J_0\beta_{max}(T_1 - T_0) - (nJ_0\beta - J_0\beta_{max})(T_1 - T_0) \\ &= nJ_0(\beta_{max} - \beta)(T_1 - T_0). \end{aligned} \tag{3}$$

To make use of the idle time T, we allocate the $L_0 - nJ_0$ (denoted by L_1) reserved job units in additional batches to explorers who finish their first batches before explorer 1. Denote the percentage of type 1 jobs in the L_1 units by β_r. Ideally, the L_1 units should be allocated to explorers 2 through n such that they will be fully engaged during the $[0, t_1]$ time period and all L_1 units will be completed at time t_1. Using the result in Eq. (1), this condition can be expressed as

$$T = L_1(\beta_r(T_1 - T_0) + T_0) + MT_c \tag{4}$$

where M is the total number of additional batches to allocate the L_1 units. The value of M depends on the actual size of each batch (including J_0) and its estimation will be discussed shortly.

Eqs. (3) and (4) imply

$$(L_0 - nJ_0)(\beta_r(T_1 - T_0) + T_0) + MT_c = nJ_0(\beta_{\max} - \beta)(T_1 - T_0). \tag{5}$$

Solving Eq. (5), J_0 can be expressed as

$$J_0 = \frac{L_0(\beta_r(T_1 - T_0) + T_0) + MT_c}{n((\beta_{\max} - \beta + \beta_r)(T_1 - T_0) + T_0)}. \tag{6}$$

To compute J_0, we need the values for β, β_{max}, β_r and M. However, they are unknown at the beginning of the search pass when J_0 is to be computed. The estimation of these values is discussed below:

The values of β and β_{max} can be estimated based on the following assumption:

Assumption 3. The difference between the values of β (β_{max}) in successive search passes is small.

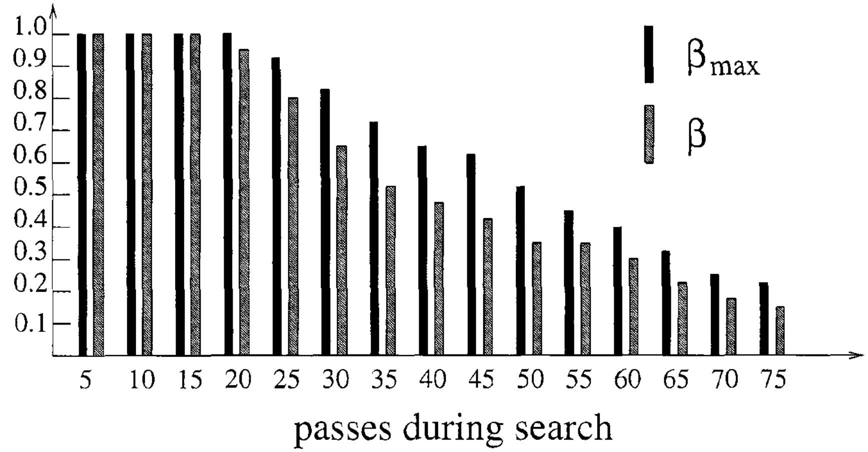


Figure 6. β and β_{max} values obtained with eight explorers.

Assumption 3 usually holds since the graphs involved in successive passes differ by only *i* links. Figure 6 shows the values of β and β_{max} from search pass 5 to 75 in learning from a dataset of 37 variables, which provides an empirical justification of the assumption.

The value of β_r usually varies from $\beta_{min} = \min_{i=1}^{n}(\beta_i)$ to β_{max}. We can approximate β_r of Eq. (6) by the average $\beta_{avg} = 0.5(\beta_{min} + \beta_{max})$.

By Eq. (6), estimation errors in β, β_{max} and β_r can make J_0 smaller or larger than the ideal value. If J_0 is smaller, more units will be reserved, resulting in more additional batches. On the other hand, if J_0 is larger, less units will be reserved and some explorers will be idle after all units have been allocated.

Finally, we consider the estimation of M. From the numerator of Eq. (6), the effect of estimation error in M is small because $\beta_r(T_1 - T_0) + T_0$ is larger than T_c and L_0 is much larger than M.

Based on Assumption 3 and the above analysis, manager can collect the values β', β'_{avg}, β'_{max} and M' from the previous pass of search to calculate the value of J_0 as follows:

$$J_0 \approx \frac{L_0(\beta'_{\text{avg}}(T_1 - T_0) + T_0) + M'T_c}{n((\beta'_{\text{max}} + \beta'_{\text{avg}} - \beta')(T_1 - T_0) + T_0)}. \qquad (7)$$

We have now determined the size of the first batch to each explorer.

Next, we determine the size for additional batches. As an example, consider a situation illustrated by figure 5. Suppose that the histogram depicts the computation time of the first batch by each explorer. Explorer 4 finishes the first. Let J_1 be the size of the second batch allocated to explorer 4. The most conservative batch size is $J_1 = L_1/(n - 1)$, which effectively assumes that every explorer (other than explorer 1) finishes at this moment. Usually other explorers will finish later and hence this size will under-allocate for explorer 4. However, the under-allocation will only slightly increases the number M of additional batches. Since T_c is very small, a few more additional batches will not affect the overall efficiency significantly. We have therefore adopted this conservative batch size.

In general, let L_2 be the remaining job units after the allocation of a batch of J_1 units to the explorer that finishes the first, L_3 be the remaining job units after the allocation of a batch of J_2 units to the explorer that finishes the second, and so on. The batch size allocated to the explorer that finishes the *ith* place will be

$$J_i = \begin{cases} \frac{L_i}{n-1} & \text{when } L_i \geq 2(n-1) \\ 1 & \text{when } L_i < 2(n-1) \end{cases} \tag{8}$$

where $i = 1,2,\ldots,$ and $L_{i+1} = L_i - J_i$. Note that after the number of remaining units drops below $2(n-1)$, jobs are allocated unit by unit to achieve high degree of load balancing.

Based on Eqs. (7) and (8), we modify Mgr1/Epr1 into algorithms Mgr2/Epr2.

Manager executes Mgr2. For each pass, it computes J_0 according to equation (7), and then sends the current graph G and a batch of J_0 graphs to each explorer. Each explorer executes Epr2. It checks chordality for each graph received and computes the entropy decrement for each chordal graph. The explorer then sends a signal to manager indicating

Algorithm 4 (Mgr2).

```
Input: $N$, $D$, $n$, $K$, $\delta h$ and $n$.
begin
        send $N$, $D$ and $n$ to each explorer;
        initialize an empty graph $G = (N, E)$, $G' := G$;
        set initial values for $\beta$, $\beta_{max}$, $\beta_{avg}$, $T_0$, $T_1$, $T_c$ and $M$;
       for $i = 1$ to $K$, do
              repeat
                    initialize the cross entropy decrement $dh' := 0$; $j := 0$;
                    send current graph $G$ and $J_j$ graphs to each explorer; $j++$;
                    repeat
                          receive a completion signalfrom an explorer;
                          if $L_j > 0$, then send $J_j$ graphs to the explorer; $j++$;
                          else send a report signal to the explorer;
                    until report signal has been sent to each explorer;
                   for each explorer $x$, do
                          receive $dh^*$, $\beta_x$, $T_0$, $T_1$ and $G^*$;
                          if $dh^* > dh'$, then $dh' := dh^*$, $G' := G^*$;
                    if $dh' > \delta h$, then $G := G'$, done := false; else done := true;
                    if done = true, then update $\beta$, $\beta_{max}$, $\beta_{avg}$, $T_O$, $T_1$ and $M = j$;
              until done = true;
        send a halt signal to each explorer;
        return $G$;
end
```

its completion of the batch. Upon receiving the signal, manager computes size J_j for an additional batch and sends the batch to the explorer. If no job units are left for this pass, manager will signal the explorer for report. After reports are collected from all explorers, manager updates the relevant search parameters and starts the next pass. Note that both T_0 and T_1 are updated to account for the inaccuracy of Assumptions 1 and 2.

7. Two-stage allocation

The two-stage allocation is based on the fact that a chordal structure and a non-chordal one require significantly different amount of computation in evaluation, and the difference is the major source of unbalanced load among processors in even allocation.

To improve load balancing, we modify even job allocation of Mgr1/Epr1 by allocating jobs in two stages as shown in algorithms Mgr3/Epr3. In the first stage, manager (see Mgr3) partitions alternative graphs *evenly* and distributes one set to each explorer. Each explorer (see Epr3) checks the chordality for each graph received and reports to manager valid candidates (chordal graphs). Since the complexity of checking chordality is $(|N| + |E|)$, and each graph has the identical number of nodes and links, the computation among explorers is *even.*

Algorithm 5 (Expr2).

```
begin
    receive N, D and nfrom manager;
    repeat
        receive G = (N, E) from manager;
        initialize dh* := 0 and G* := G;
        repeat
            receive a set ofgraphsfrom manager;
            for each received graph G' = (N, L ∪ E), do
                ifG' is chordal and L is implied by a single clique ofsize ≤ n,
                    then compute the entropy decrement dh';
                ifdh' > dh*, then dh* := dh', G* := G';
            send a completion signal to manager:
        until report signal is received;
        send dh*, $\beta_x$, $T_0$, $T_1$ and G* to manager;
    until halt signal is received:
end
```

In the second stage, manager partitions all received graphs *evenly* and distributes one set to each explorer. Each explorer computes entropy decrement for each graph received. It then chooses the best graph and reports it and its entropy decrement to manager. Manager collects the reported graphs, selects the best, and then starts the next pass. Since all graphs are chordal in the second stage, the degree of load balance mainly depends on the variability of the sizes of the largest cliques.

8. Comparison of allocation strategies

Compared with multi-batch allocation, two-stage allocation is much simpler. It only needs to partition and distribute job units twice. With the multi-batch allocation, multiple batches are sent to each explorer, resulting higher communication overhead. For example, in learning from a database of 37 variables with 12 explorers, we found that on average six batches are sent to each explorer. The data collection and computation involved in multi-batch allocation are also more expensive.

However, two-stage allocation suffers from variation in the amount of computation for calculating entropy decrements as each set L of new links forms new cliques whose sizes may vary significantly. On the other hand, the multi-batch allocation has the resistance to the variation in clique size since allocation is dynamically adapted to the *actual* amount of computation used for each batch.

We present the experimental comparison of the two strategies in Section 11.

Algorithm 6 (Mgr3).

```
Input: $N$, $D$, $n$, $K$, $\delta h$ and $n$.
begin
    send $N$, $D$ and $n$ to each explorer;
    initialize an empty graph $G = (N, E)$, $G' := G$;
    for $i = 1$ to $K$, do
        repeat
            initialize $dh' := 0$;
            partition all graphs that differ from $G$ by $i$ links into $n$ sets;
            send one set of graphs and $G$ to each explorer;
            receive a set of valid graphs from each explorer;
            partition all received graphs into $n$ sets;
            send one set of graphs to each explorer;
            for each explorer, do
                receive $dh^*$ and $G^*$;
                if $dh^* > dh'$, then $dh' := dh^*$, $G' := G^*$;
            if $dh' > \delta h$, then $G := G'$, done := false; else done := true;
        until done = true;
    send a halt signal to each explorer;
    return $G$;
end
```

Algorithm 7 (Erpr3).

```
begin
    receive $N$, $D$ and $n$ from manager;
    repeat
        receive current graph $G = (N, E)$ and a set of graphs from manager;
        initialize $dh^* := 0$ and $G^* := G$;
        for each received graph $G' = (N, L \cup E)$, do
            if $G'$ is chordal and $L$ is implied by a single clique of size $\leq n$,
                then mark it as valid;
        send all valid graphs to manager;
        receive a set of graphs from manager;
        for each received graph $G'$, do
            compute the entropy decrement $dh'$;
            if $dh' > dh^*$, then $dh^* := dh'$, $G^* := G'$;
        send $dh^*$ and $G^*$ to manager;
    until halt signal is received;;
end
```

9. Marginal servers

In order to learn a belief network with satisfactory accuracy, a dataset of large number of cases is preferred. During learning, the data will be frequently accessed by each explorer to obtain marginal probability distributions (marginals) of subsets of variables (for computing entropy decrements). Using a distributed-memory architecture, the available local memory to each processor is limited. If the dataset (with a proper compression) can be fit into the local memory such that each processor has one copy of the dataset, then data can be accessed effectively during learning. Otherwise, special measure has to be taken for data access.

One obvious solution is to access data through the file system. However file access is much slower than memory access. Even worse, many parallel computers have limited channels for file access, making it a bottleneck. For example, in the computer available to us, file access by all processors must be performed through a single host computer.

To achieve efficient data access, we propose an alternative using so called *marginal servers* to avoid file access completely during learning. The idea is to split the dataset so that each subset can be stored into the local memory of a processor. A group of (say *m)* such processors is then given the task of serving explorers in computing partial marginals from their local data.

In particular, the m servers are connected *logically* into a pipeline. The dataset is partitioned into $m + 1$ sets, where the size of each set may not be identical as we will discuss shortly. Each server stores one *distinct* set of data and each explorer *duplicates* one copy of the remaining set.

As an example, consider the computation of the marginal over two binary variables $\{x, y\} \subset N$. Suppose $|D| = 10000$ and there are one explorer and two marginal servers.

Table 2. Data storage using servers.

(x, y)	Tuples in explorer	Tuples in server 1	Tuples in server 2
(0, 0)	2000	1000	500
(0, 1)	1500	500	1000
(1, 0)	1000	500	500
(1, 1)	500	500	500

We store 5000 tuples in the explorer and 2500 in each server. Table 2 shows one possible scenario of how the tuples might be distributed according to *{x, y}*.

When the explorer needs to compute the marginal over *{x,* y}, it first sends *{x, y}* to servers, and then computes locally the*potential* (non-normalized distribution) (2000, 1500, 1000, 500). Requested by the explorer, server 1 computes the local potential (1000, 500, 500, 500) and sends to server 2. Server 2 computes its local potential, adds to the result from server 1 to obtain the sum (1500, 1500, 1000, 1000), and sends the sum to the explorer. The explorer adds the sum to its local potential to obtain (3500, 3000, 2000, 1500) and normalizes to get the marginal (0.35, 0.3, 0.2, 0.15).

Two-stage allocation enhanced by marginal servers is implemented in Mgr4, Epr4 and Svr. Multi-batch allocation can be enhanced accordingly.

Algorithm 8 (Mgr4).

```
Input: N, D, n, K, $\delta h$, n and m.
begin
    partition D into m + 1 sets;
    send one set to each server and broadcast the last set to explorers;
    initialize an empty graph G = (N, E), G' := G;
    fori = 1 to K, do
        repeat
            initialize dh' := 0;
            partition all graphs that differfrom G by i links into m + n sets;
            send one set ofgraphs and G to each explorer and each server;
            receive a set ofvalid graphsfrom each explorer and each server;
            partition all received graphs into n sets;
            send one set ofgraphs to each explorer;
            for each explorer, do
                receive dh* and G*;
                ifdh* > dh' then dh' := dh*, G' := G*;
            ifdh' > $\delta h$, then G := G', done :=false; else done := true;
            send an end-of-pass signal to each server;
        until done = true;
    send a halt signal to each explorer and each server;
    return G;
end
```

Manager executes Mgr4. It partitions data into $m + 1$ sets, distributes to explorers and servers, and starts the search process. In the first stage of each pass, manager generates alternative graphs based on the current graph. It partitions them into $m + n$ sets, distributes to explorers and servers, and receives reported valid graphs. In the second stage, manager partitions valid graphs into n sets and sends one set to each explorer.

Each explorer executes Epr4. In the first stage of each pass, it checks the chordality of each received graph and reports valid graphs to manager. In the second stage, the explorer receives a set of valid graphs from manager. For each graph received, it identifies the marginals (each over a subset $C c N)$ necessary in computing entropy decrement. For each marginal, it sends a request to servers, computes a local potential, receives a potential from a server (to be specified below), sums them up and obtains the marginal. After evaluating all valid graphs received, the explorer chooses the best graph and reports to manager. Manager collects reported graphs from all explorers, selects the best as the new current graph, sends a signal to each server to notify the end of the current pass, and then starts the next pass.

Each marginal server executes Svr. In the first stage of each pass, each server functions as an explorer (testing chordality). In the second stage, a server processes requests repeatedly until it receives a signal to end the current pass. For each request (a marginal over a subset $C c N)$, a server computes a local potential, adds to the potential from its

Algorithm 9 (Epr4).

```
begin
    receive n and a subset ofdata over N;
    repeat
        receive G = (N, E) and a set ofgraphsfrom manager;
        initialize dh* := 0 and G* := G;
        for each received graph G' = (N, L ∪ E), do
            ifG' is chordal and L is implied by a clique ofsize ≤ n, then mark G' valid;
        send all valid graphs to manager;
        receive a set ofgraphsfrom manager;
        for each received graph G' = (N, L ∪ E), do
            for each set C ofvariables involved in computing dh', do
                send C to each marginal server;
                compute local potential over C;
                receive a potential over Cfrom a server;
                compute marginal over C;
            compute the entropy decrement dh':
            ifdh' > dh*, then dh* := dh', G* := G';
        send dh* and G* to manager:
    until halt signal is received;;
end
```

predecessor if it is not the head of the pipeline, and sends the sum to the next server or the requesting explorer depending on whether it is the end of the pipeline.

To keep all processors fully engaged, the dataset D must be properly partitioned among explorers and servers. Since each server serves n explorers, the processing of one request by a server must be n times as fast as the local processing of a requesting explorer. This implies $nT_s = T_e$, where T_s and T_e are the time to process one marginal request by a server and an explorer, respectively. Let $|D_s|$,and $|D_e|$ be the number of tuples stored locally in each server and each explorer, respectively. T_s and T_e can be expressed as $T_s = k_d|D_s|$ and $T_e = k_g|N| + k_d|D_e|$, where k_d and k_g are coefficients, and $k_g|N|$ is the computation time to identify the marginals necessary in computing entropy decrement. Therefore, we have

$$nk_d|D_s| = k_g|N| + k_d|D_e|. \tag{9}$$

In algorithm Mgr4, D is partitioned into $m + 1$ sets and hence

$$|D| = m|D_s| + |D_e|. \tag{10}$$

Denoting $p = m + n$ and solving Eqs. (9) and (10), we obtain

$$n = \frac{p(\alpha|N| + |D_e|)}{\alpha|N| + |D|}, \quad m = \frac{p(|D| - |D_e|)}{\alpha|N| + |D|}, \quad |D_s| = \frac{\alpha|N| + |D|}{p},$$

where $\alpha = k_g/k_d$ with its value between 0.003 to 0.006 in our experimental environment. In practice, *m, n, ID,*| and | D_e| must be rounded to integers, and $|D_e|$ must be upper bounded

Algorithm 10 (Svr).

```
begin
    receive n and a subset ofdata over N;
    repeat
        receive $G = (N, E)$ and a set ofgraphsfrom manager;
        for each received graph $G' = (N, L \cup .E)$, do
            ifG' is chordal and L is implied by a clique ofsize $\leq \eta$, then mark G' valid;
        send all valid graphs to manager;
        repeat
            receive a set C ofvariablesfrom an explorer;
            compute localpotential over C;
            ifthis server is not head ofserverpipeline, then
                receive a potential over Cfrom predecessor server;
                sum localpotential with receivedpotential;
            ifthis server is not tail ofserverpipeline, then send sum to the next server;
            else send sum to the requesting explorer;
        until end-of-pass signal is received;;
    until halt signal is received;;
end
```

by the available local memory for data storage. As an example, suppose $|D| = 100k$, $|D_e| = 20$k, $|N| = 1000$, $p = 30$ and $\alpha = 0.005$. We have $n = 6$, $m = 24$ and $|D_s| \approx 3.334$k.

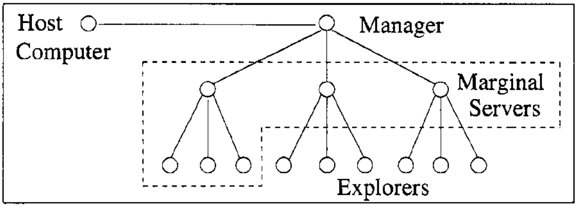


Figure 7. Ternary tree topology.

10. Experimental environment

The parallel algorithms presented have been implemented on an ALEX AVX Series 2 distributed memory MIMD computer. It contains *8* root nodes and 64 compute nodes, which may be partitioned among and used by multiple users at any time. Each root node is a *T*805 processor, which can be used to control the topology of compute nodes. Each compute node consists of an *i* 860 processor (40 Mhz) for computation and a *T* 805 processor for message passing with other nodes through four channels at each node. Data transmission rate is 10 Mbps in simplex mode and 20 Mbps in duplex mode. The *i*860 and *T* 805 processors at each node share 32 MB memory and the latter has its own additional *8* MB memory. All access to the file system is through a root node and a host computer.

We configure the available processors into a ternary tree (figure 7) to reduce the length of message passing path. The root is manager and non-root nodes are explorers/servers. Servers cooperate *logically* as a pipeline.

We tested our implementation using the *ALARM* network (Beinlich et al., *1989)* and four randomly generated networks *PIMi* $(i = 1, \ldots, 4)$ each of which is a PI model. *ALARM* has 37 variables. *PIM*1 has 26 variables and contains an embedded PI submodel over three variables. *PIM* 2 has 30 variables and contains two embedded PI submodels each of which is over three variables. *PIM* 3 has 35 variables and contains two embedded PI submodels similar to those of *PIM* 2. *PIM* 4 has 16 variables and contains one embedded PI submodel over four variables. Five datasets are generated by sampling the five control networks with 10000, 20000, 25000, 30000 and 10000 cases, respectively.

We measure the performance of our programs by *speed-up (S)* and *efficiency (E).* Given a task, let $T(1)$ be the execution time of a sequential program and $T(n)$ be that of a parallel program with n processors. Then $S = T(1)/T(n)$ and $E = S/n$.

11. Experimental results

We demonstrate the performance of multi-batch and two-stage allocation strategies and the benefit of using marginal servers.

The DMN learned from *ALARM* dataset is shown in figure 8 (left). Since the task decomposition that we used for parallelism does not introduce errors, the learning outcome is *identical* to what is obtained by Seq with the same learning parameters. Figure *8* (right)

Table 3. Experimental results for even and two-stage allocations.

	Even allocation			Two-stage allocation		
n	Time (s)	Speed-up	Efficiency	Time (s)	Speed-up	Efficiency
1	3160	1.0	1.0	3160	1.0	1.0
2	1750	1.81	0.903	1617	1.95	0.977
4	957	3.30	0.825	850	3.72	0.929
6	712	4.44	0.740	609	5.19	0.865
8	558	5.66	0.708	472	6.69	0.837
10	486	6.50	0.650	393	8.04	0.804
12	454	6.96	0.580	351	9.00	0.750

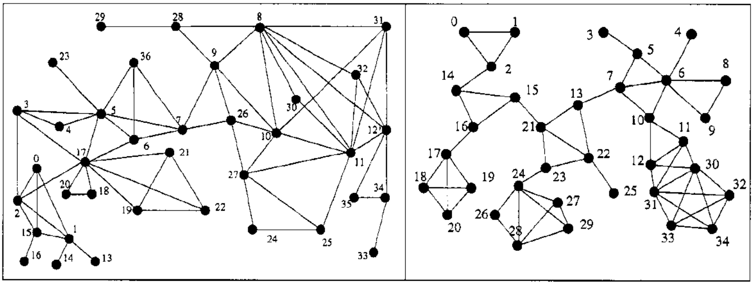


Figure 8. DMNs learned from data obtained from ALARM (left) and PZM3 (right).

shows the DMN learned from *PIM*3 dataset. Nodes labeled 6, 8 and 9 form a PI submodel in *PIM*3 and so do nodes labeled 14, 15 and 16.

In learning from the ALARM dataset, we compared even (Mgr1/Epr1), multi-batch (Mgr2/Epr2) and two-stage (Mgr3/Epr3) allocations. The dataset, after compression, was loaded into the local memory of each explorer. Table 3 shows experimental results for even and two-stage allocations as the number *n* of explorers increases from 1 to 12.

Columns 3 and 6 show that as *n* increases, speed-up increases as well when either allocation strategy is used. This demonstrates that the parallel algorithms can effectively reduce learning time and provides positive evidence that parallelism is an alternative to tackle the computational complexity in learning belief networks.

Comparing column 3 with 6 and column 4 with 7, it can be seen that two-stage allocation further speeds up learning and improves efficiency beyond that of even allocation. For example, when eight explorers are used, speed-up is 5.66 and efficiency is 0.708 for even allocation, and 6.69 and 0.837 for two-stage. Figure 9 plots the speed-up and efficiency for all three strategies for comparison.

Among the three strategies, even allocation has the lowest speed-up and efficiency, especially when *n* increases. There is no significant difference between multi-batch and two-stage

Table 4. Experimental results in learning PI models.

n		*PIM*1	*PIM*2	*PIM*3	*PIM*4
1	Time (min)	262.4	868.6	3555.4	36584
12	Time(min)	26.8	89.3	352.2	3382
	Speed-up	9.8	9.7	10.1	10.8
	Efficiency	0.82	0.81	0.84	0.90
24	Time(min)	17.2	54.2	179.4	1735
	Speed-up	15.3	16.0	19.8	21.1
	Efficiency	0.64	0.67	0.83	0.88
36	Time(min)	12.5	37.7	124.5	1197
	Speed-up	21.0	23.0	28.6	30.6
	Efficiency	0.58	0.64	0.79	0.85

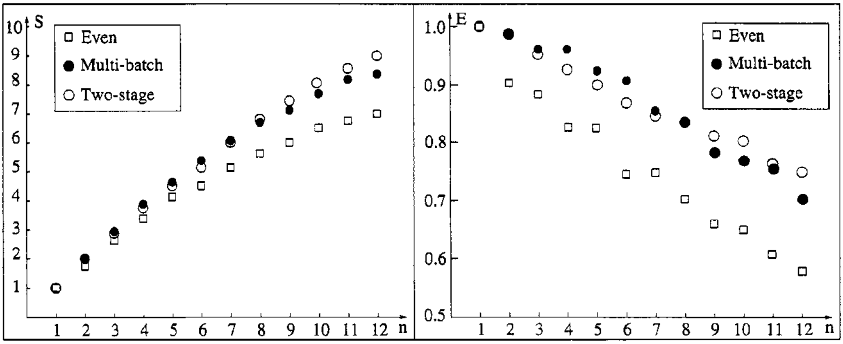


Figure 9. Speed-up (left) and efficiency (right) in learning from *ALARM* dataset.

allocations. For $n > 6$, multi-batch allocation is slightly better than two-stage allocation. As n increase beyond 9, two-stage performs better than multi-batch. This is because the overhead of multi-batch job allocation becomes more significant as the number of explorers increases.

The results also show a gradual decrease in efficiency as n increases. This decrease is due to allocation overhead. At the start of each pass, manager allocates jobs to explorers in sequence. Hence an explorer is idle between submission of its report in previous pass and receipt of the next batch of jobs. However, efficiency decrease will be less significant when learning is performed in large or PI domains as the proportion of message passing time in each pass will be much smaller than computation time. This is illustrated by our learning results in PI domains as follows:

Table 4 shows experimental results for learning *PI* models *PIMi* $(i = 1, \ldots, 4)$, where triple-link lookahead is used for learning *PIMi* $(i = 1, , \ldots, 3)$ and six-link lookahead is used

Table 5. Experimental results by using four marginal servers.

n+*m*	5	6	7	8	9	10	11	12
Time(s)	2870	1616	1166	1015	910	819	762	737
Speed-up	4.45	7.91	10.96	12.59	14.04	15.60	16.77	17.34
Efficiency	0.891	1.318	1.566	1.574	1.560	1.560	1.525	1.445

for learning *PIM* 4. The first column indicates the number of explorers used. As expected, speed-up is shown to increase with *n*.

The third column shows results in learning *PIM* 1. When 12 explorers are used, speed-up and efficiency are 9.8 and 0.82. The table shows rapid decrease of efficiency when 24 and 36 explorers are used. The similar trend can be seen in column 4 for learning *PIM* 2. This is because the two domains are relatively small (with 20 and 30 variables, respectively) and less complex (sparse, and with one and two small PI submodels, respectively). Message passing time is significant compared with computation time in these cases.

Column 5 shows results for learning *PIM* 3. The domain contains 35 variables and two PI submodels, and the control network is more densely connected. Significantly longer computation time (3555.4 min) was used by the sequential program. The last column shows results for learning *PIM* 4. Although its domain is not large (16 variables), the fact that it contains a *PI* sub-model with 4 variables and a six-link lookahead is needed to identify the sub-model makes its computation expensive. It took the sequential program over *25* days *(36584* min). Compared with *PIM* 1 and *PIM* 2, speed-up and efficiency in learning these two models are much better when larger number of explorers are used. Note that with 36 explorers, the time to learn *PIM* 4 is reduced from over 25 days to less than one day (1197 min).

Finally, we demonstrate the use of marginal servers by learning the *ALARM* network. Although *ALARM* is not very large and the dataset can be loaded entirely into the local memory of each explorer, we choose to use it for two reasons: First, domain size does not hinder demonstration of correctness of the server method. Second, if we decrease the available local memory below what we have, at some point, it would not be large enough to hold *ALARM* dataset. In that case, data access by file system would be necessary if the server method were not used. Hence, generality is not compromised by using *ALARM.*

To demonstrate the effect of using servers, we assume that the dataset cannot be loaded into local memory of explorers. Using data access by file system, it took 12780 sec for the sequential program to complete learning *ALARM.* Table 5 shows results of learning *ALARM* by using $m = 4$ servers. The number n of explorers ranges from one to eight. The data size stored in each explorer was twice as large as that in each server. Note that since marginal servers replace slow file access by fast memory access, the efficiency can be larger than 1.0 as shown in the table.

12. Looking beyond distributed memory MIMD

Flynn’s taxonomy (Moldovan, 1993) classifies hardware into SISD, SIMD, MISD and MIMD. MIMD computers can be further classified into shared or distributed memory. The following discussion extends our lessons from using distributed memory MIMD to the suitability of other architectures for parallel learning of belief networks. As SISD is incapable of true parallelism (Lewis and Rewini, 1992), we discuss only SIMD, MISD and MIMD.

An MISD computer applies multiple instructions to a single data stream. For example, it can perform matrix operations **A + B** and **A - B** simultaneously. The task of learning belief networks decomposes naturally into evaluation of alternative network structures (multiple data streams) as we have investigated in this study. Therefore, the MISD architecture appears unsuitable for this task.

SIMD computers consist of multiple arithmetic logic units (ALUs) under the supervision of a single control unit (CU). CU synchronizes all the ALUs by broadcasting control signals to them. The ALUs perform the same instruction on different data that each of them fetches from its own memory. For instance, CM-2 connect machine has 64K processors each of which is an one-bit CPU with 256K one-bit memory. Normal instructions are executed by a host computer and the vector instructions are broadcast by the host computer and executed by all processors.

In learning belief networks, each alternative network structure has a unique graphical topology and requires a unique stream of instructions for its evaluation. Therefore, SIMD computers do not appear suitable if the learning task is decomposed at the level of network structures. In other words, it appears necessary to decompose the task at a much *lower* abstraction level. One alternative is to partition the dataset into small subsets each of which is then loaded into the memory of one processor. Each marginal can then be computed by cooperation of multiple processors when requested by a host computer. However, the host computer must carry out all other major steps in evaluating each alternative structure. This is essentially the sequential learning (algorithm Seq) with parallelism applied to only marginal computation. The degree of parallelism is much reduced compared with what we have presented. Therefore, SIMD computers do not appear a better architecture than the MIMD that we have used.

In a MIMD computer, each processor can execute its own program upon its own data. Cooperation among processors is achieved by either shared memory or message passing (in distributed memory architectures). In a MIMD computer with shared memory, all programs and data are stored in *k* memories and are accessible by all processors with the restriction that each memory can be accessed by one processor at any time. This restriction tends to put an upper bound on the number of processors that can be effectively incorporated. Therefore, shared memory systems are efficient for small to medium number of processors.

For parallel learning of belief networks on a shared memory MIMD computer, our manager/explorer partition of processors can be used. Manager generates alternative structures and stores them in one memory. Each explorer can fetch one or more structures for evaluation at each time, which can be controlled by accessing a critical section. Hence job allocation can be performed similarly to our multi-batch or two-stage strategies. On the other hand, dataset access will become a bottleneck if a large number of processors want to access the same memory for data at the same time. The problem may be alleviated by duplicating the dataset in multiple memories. However, this may not be practical for large datasets due to limited total memory.

Based on our investigation using a distributed memory MIMD computer and the above analysis, we believe that this architecture is most suited to parallel learning of belief networks among the four architectures considered.

13. Conclusion

We have investigated parallel learning of belief networks as a way to tackle the computational complexity when learning in large and difficult (e.g., PI) problem domains. We have proposed parallel algorithms that decompose the learning task naturally for parallelism and they do not introduce errors compared with a corresponding sequential learning algorithm. We have studies multi-batch and two-stage job allocations which further improve the efficiency of the parallel system beyond the straightforward even allocation strategy. We found that multi-batch is more effective when the number of processors is small and two-stage is more effective when the number is large. We have proposed marginal server configuration to replace slow data access through file system by fast memory access. This allows parallel learning from very large datasets be performed effectively. We have implemented the algorithms in a distributed memory MIMD computer and our experimental results confirmed our analysis.

Our study has focused on learning DMNs. However, our results can be easily extended to learning Bayesian networks (BNs). This is because all known algorithms for learning belief networks (whether they are DMNs or BNs) are based on evaluation of alternative network structures (often using local computations) relative to the given dataset. Therefore, our results on task decomposition, job allocation strategies and use of marginal servers are applicable to learning any type of belief networks.

We have extended the lessons we learned from using the distributed memory MIMD system to other architectures based on Flynn's taxonomy. Our analysis of the features of each architecture and the features of learning belief networks makes us believe that the distributed memory MIMD architecture is most suited to this task.

Appendix A: Graph-theoretic terminology

Let G be an undirected graph. A set X of nodes in G is *complete* if each pair of nodes in X is adjacent. A set C of nodes is a *clique* if C is complete and no superset of C is complete. A *chord* is a link that connects two nonadjacent nodes. G is *chordal* if every cycle of length >3 has a chord.

A decomposable Markov network (DMN) over a set N of variables is a pair (G, P) where G is a chordal graph and P is a probability distribution over N. Each node in $G = (N, E)$ is labeled by an element of N. Each link in G signifies the direct dependence of its end nodes. For disjoint subsets X, Y and Z of nodes, $<X|Z|Y>_G$ signifies $I(X, Z, Y)$, and hence P can be factorized into marginal distributions over cliques of G.

Appendix B: Frequently used acronyms

BN: Bayesian network
DMN: decomposable Markov network

PDM: probabilistic domain model
PI: pseudo-independent
MIMD: multiple instruction, multiple data
MISD: multiple instruction, single data
MLLS: multi-link lookahead search
SISD: single instruction, single data
SIMD: single instruction, multiple data
SLLS: single link lookahead search

Acknowledgments

This work is supported by research grants from the Natural Sciences and Engineering Research Council of Canada and from the Institute for Robotics and Intelligent Systems in the Networks of Centres of Excellence Program of Canada.

Note

[1] The survey is over 469 variables. Analysis was performed only on data about some subtopics due to limited time, More PI models may be found if analysis is applied to the entire data.

References

1. Beinlich, LA., Suermondt, H.J., Chavez, R.M., and Cooper, G.F., 1989. The alarm monitoring system: a case study with two probabilistic inference techniques for belief networks. Technical Report KSL-88-84, Knowledge Systems Lab, Medical Computer Science, Stanford University.
2. Chickering, D., Geiger, D., and Heckerman, D. 1995. Learning Bayesian networks: serach methods and experimental results. In Proc. of 5th Conf. on Artificial Intelligence and Statistics. Ft. Lauderdale, Society for AI and Statistics, pp. 112–128.
3. Cooper, G.F. and Herskovits, E. 1992. A Bayesian method for the induction of probabilistic networks from data. Machine Learning, (9):309–347.
4. Heckerman, D., Geiger, D., and Chickering, D.M. 1995. Learning Bayesian networks: the combination of knowledge and statistical data. Machine Learning, 20: 197–243.
5. Herskovits, E.H. and Cooper, G.F. 1990. Kutato: an entropy-driven system for construction of probabilistic expert systems from database. Proc. 6th Conf. on Uncertainty in Artificial Intelligence. Cambridge, pp. 54–62.
6. Hu, J. 1997. Learning belief networks in pseudo indeependent domains. Master’s thesis. University of Regina.
7. Jensen, F.V. 1996. An Introduction to Bayesian Networks. UCL Press.
8. Kullback, S. and Leibler, R.A. 1951. On information and sufficiency. Annals of Mathematical Sratistics, 22:79–86.
9. Lam, W. and Bacchus, F. 1994. Learning Bayesian networks: an approach based on the MDL principle. Computational Intelligence, 10(3):269–293.
10. Lewis, T.G. and El-Rewini, H. 1992. Introduction to Parallel Computing. Prentice Hall.
11. Moldovan, D.I. 1993. Parallel Processing: From Applications To Systems. Morgan Kaufman.
12. Pearl, J. 1988. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann.
13. Spirtes, P. and Glymour, C. 1991. An algorithm for fast recovery of sparse causal graphs. Social Science Computer Review, 9(1):62–73.

14. Xiang, Y. 1997. Towards understanding of pseudo-independent domains. Poster Proc. 10th Inter. Symposium on Methodologies for Intelligent Systems.
15. Xiang, Y., Wong, S.K.M., and Cercone, N. 1996. Critical remarks on single link search in learning belief networks. Proc. 12th Conf. on Uncertainty in Artificial Intelligence, Portland, pp. 564–571.
16. Xiang, Y., Wong, S.K.M., andCercone, N. 1997. A 'microscopic' study of minimum entropy search in learning decomposable Markov networks. Machine Learning, 26(1):65–92.

Yang Xiang is an Associate Professor in the Department of Computer Science at University of Regina, Canada. He received his Ph.D. from University of British Columbia in 1992. He is a Principle Investigator of the Canadian Institute of Robotics and Intelligent Systems (IRIS). His main research interest concerns probabilistic reasoning with belief networks, knowledge discovery from data, distributed inference in multiagent systems, diagnosis and trouble-shooting. He developed the toolkit WEBWEAVR-III for normative decision support available from his homepage. He can be reached at yxiang@cs.uregina.ca.

Tongsheng Chu received his B.S. and Ph.D. in mechanical engineering from University of Jiaotong, China, and his M.S. in computer science from University of Regina, Canada. He was an Assistant Professor at Northwest Institute of Light-Industry, China and worked on computer-aided design and manufacturing in mechanical engineering. Since 1997, he has been a software engineer at Avant! Corporation, California. His current research interests include computer-aided design of VLSI circuits with emphasis on formal verification.