HIGH PERFORMANCE DATA MINING

Scaling Algorithms, Applications and Systems

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

Editorial ..... Yike Guo and Robert Grossman 1

Full Paper Contributors

Parallel Formulations of Decision-Tree Classification Algorithms .....
..... *Anurag Srivastava, Eui-Hong Han, Vipin Kumar and Vineet Singh* 3

A Fast Parallel Clustering Algorithm for Large Spatial Databases .....
..... *Hiaowei Xu, Jochen Jäger and Hans-Peter Kriegel* 29

Effect of Data Distribution in Parallel Mining of Associations .....
..... *David W. Cheung and Yongagao Xiao* 57

Parallel Learning of Belief Networks in Large and Difficult Domains .....
..... *Y. Xiang and T. Chu* 81

Data Mining and Knowledge Discovery, 3, 235–236 (1999)

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.

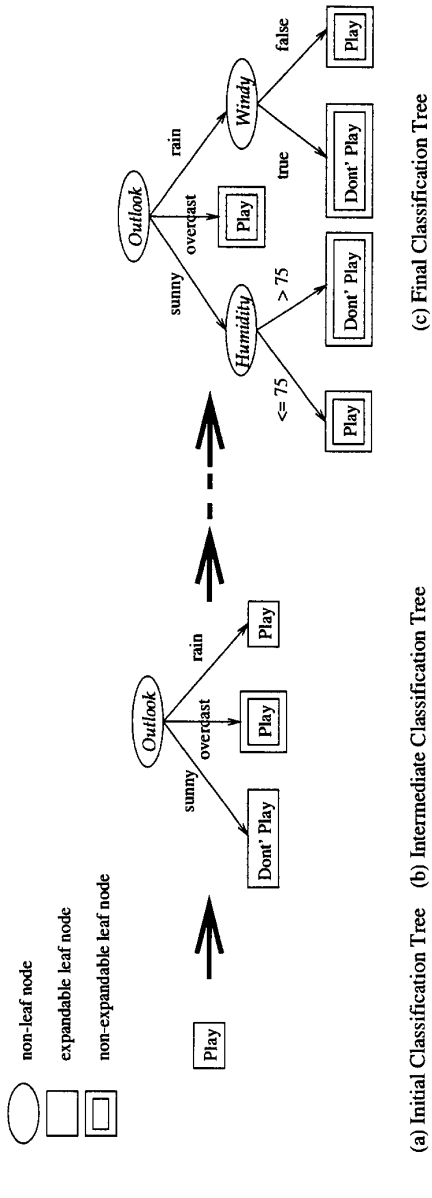
Case 3. T contains no cases. The decision tree for T is a leaf, but the class to be associated with the leaf must be determined from information other than T. For example, *C4.5* chooses this to be the most frequent class at the parent of this node.

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.

Table 1. 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

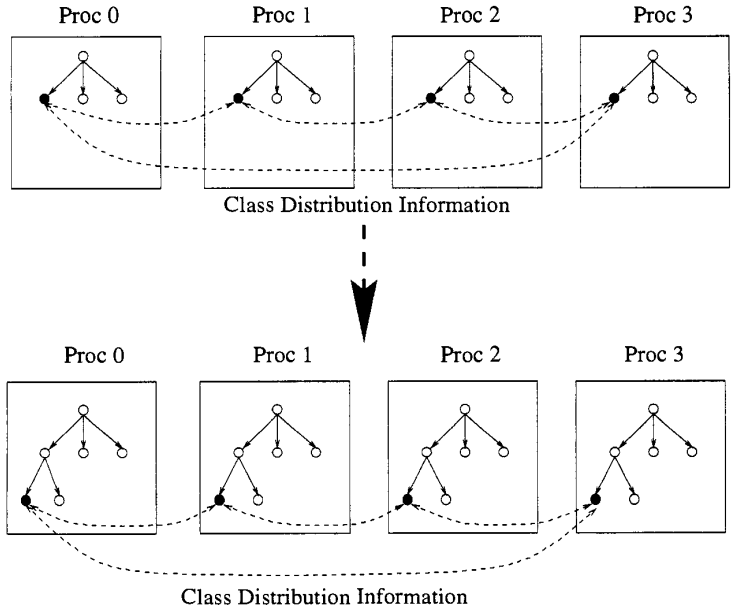


Figure 1. Synchronous tree construction approach with depth first expansion strategy.

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_m 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_m 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

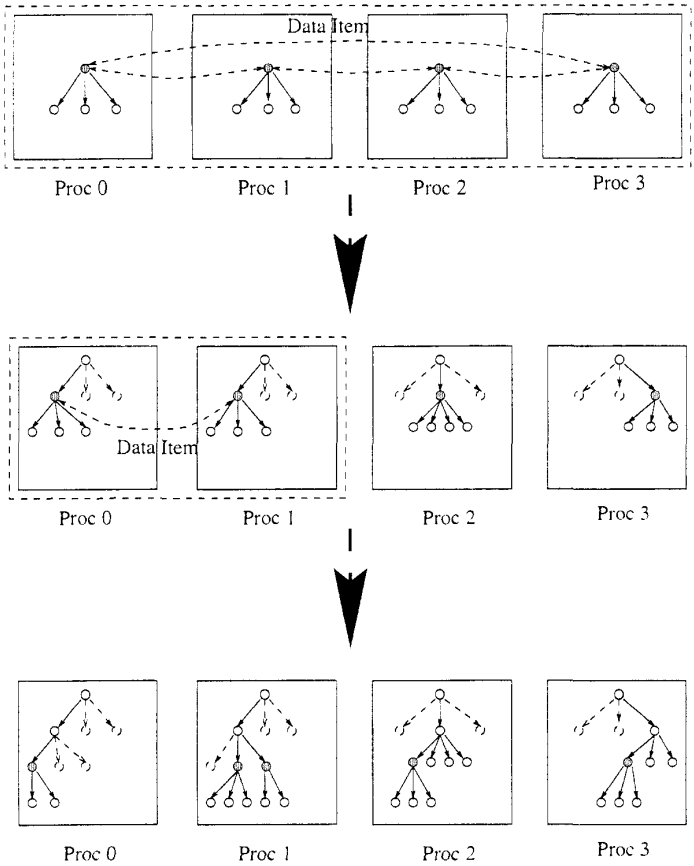


Figure 2. Partitioned tree construction approach

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.

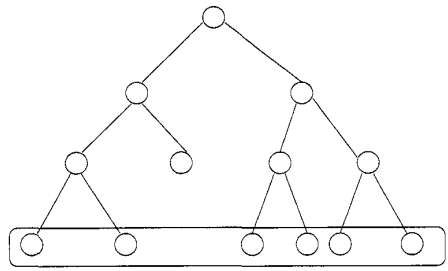


Figure 3. The computation frontier during computation phase.

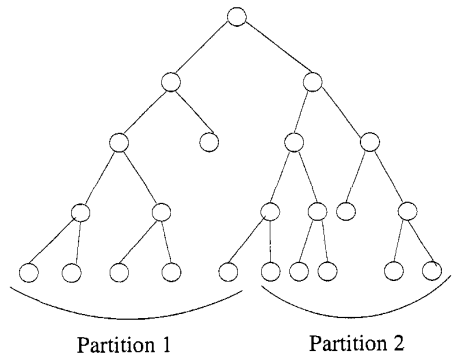


Figure 4. Binary partitioning of the tree to reduce communication costs.

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