, , 1–24 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Parallel Formulations of Decision-Tree Classification Algorithms

Anurag Srivastava[1] Eui-Hong Han[2] Vipin Kumar[2] Vineet Singh[3]

[1]Digital Impact
[2]Dept. of Computer Science & Engineering, University of Minnesota
[3]Information Technology Lab, Hitachi America, Ltd.

Editor:

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 [17], genetic algorithms [11], and decision trees [20] have been proposed. Decision trees are probably the most popular since they obtain reasonable accuracy [9] and they are relatively inexpensive to compute. Most current classification algorithms such as *C4.5* [20], and *SLIQ* [18] are based on the *ID3* classification decision tree algorithm [20].

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 [24, 5, 6, 7]. 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* [20], *CDP* [1], *SLIQ* [18], and *SPRINT* [21] use Hunt's method [20] 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 1 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

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	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Table 1. A small training data set [Qui93]

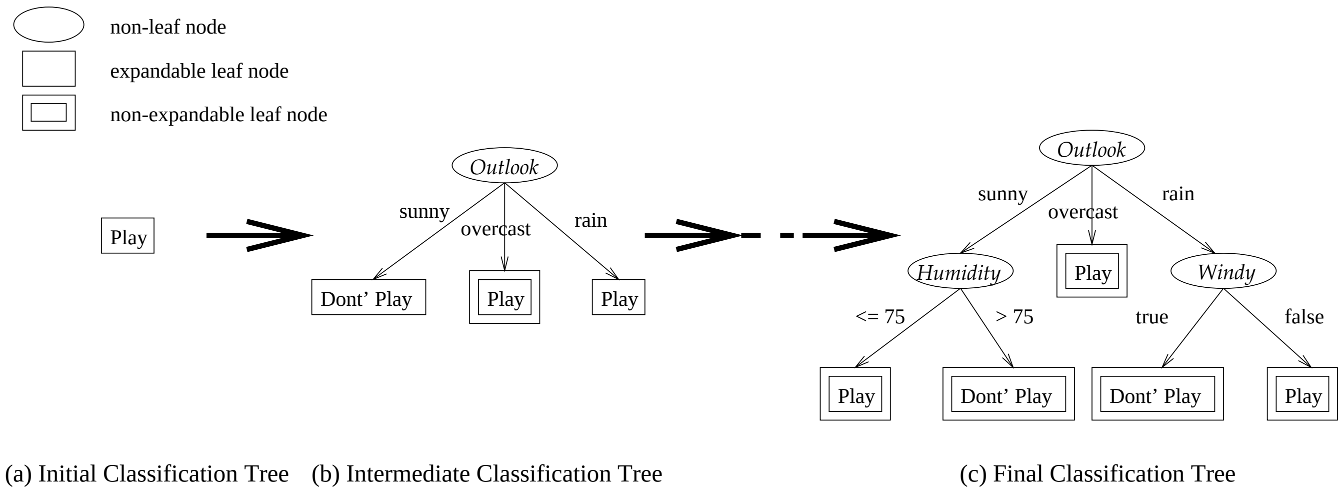

Figure 1. Demonstration of Hunt's Method

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.

Attribute Value	Class	
	Play	Don't Play
sunny	2	3
overcast	4	0
rain	3	2

Table 2. Class Distribution Information of Attribute *Outlook*

Attribute Value	Binary Test	Class	
		Play	Don't Play
65	$\leq$	1	0
	$>$	8	5
70	$\leq$	3	1
	$>$	6	4
75	$\leq$	4	1
	$>$	5	4
78	$\leq$	5	1
	$>$	4	4
80	$\leq$	7	2
	$>$	2	3
85	$\leq$	7	3
	$>$	2	2
90	$\leq$	8	4
	$>$	1	1
95	$\leq$	8	5
	$>$	1	0
96	$\leq$	9	5
	$>$	0	0

Table 3. Class Distribution Information of Attribute *Humidity*

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* [20] or *Gini Index* [4]. 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. 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* [18] and *SPRINT* [21] 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 [19]. 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 [8], a few general approaches for parallelizing C4.5 are discussed. In the Dynamic Task Distribution (DTD) scheme, a master processor allocates a subtree of the decision tree to an idle slave processor. This scheme does 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 [8] 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 [15]. This approach is similar to the DP-rec scheme [8] 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 [21] and ScalParC [13] use methods for partitioning work that is identical to the one used in the synchronous tree construction approach discussed in this paper. Serial SPRINT [21] 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 [16] 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 [13] 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, Aluru, and Ranka proposed the Concatenated Parallelism strategy for efficient parallel solution of divide and conquer problems [10]. 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 (eg. 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 [16] 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.

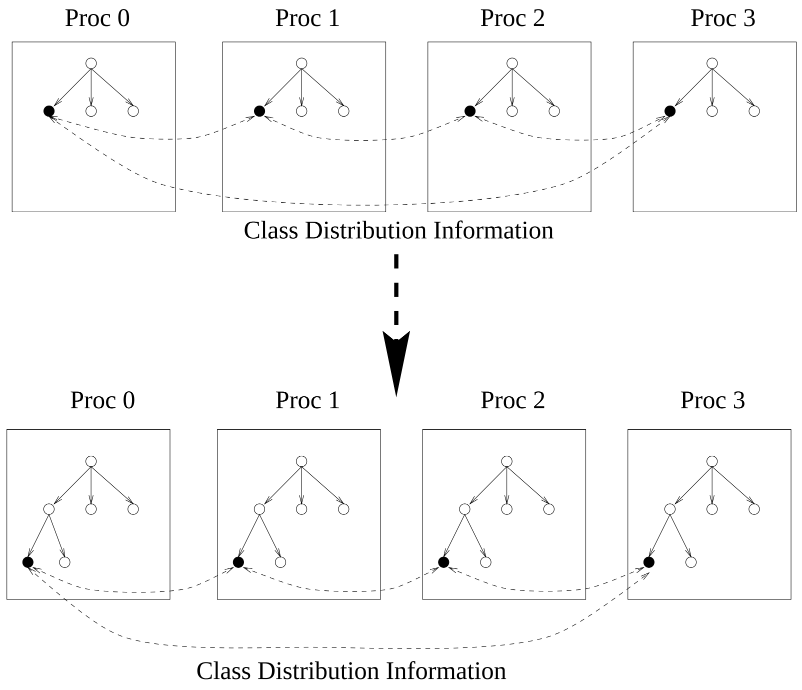
Proc 0
Proc 1
Proc 2
Proc 3
Class Distribution Information
Proc 0
Proc 1
Proc 2
Proc 3
Class Distribution Information

Figure 2. Synchronous Tree Construction Approach with Depth–First Expansion Strategy

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 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.

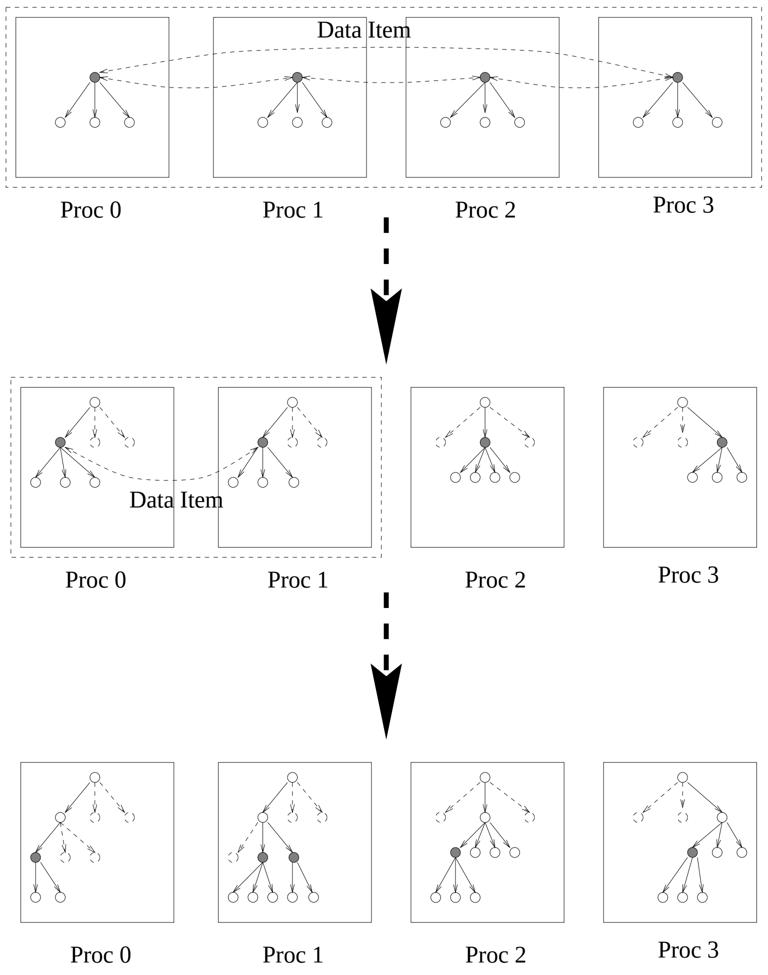

Figure 3. Partitioned Tree Construction Approach

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 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.

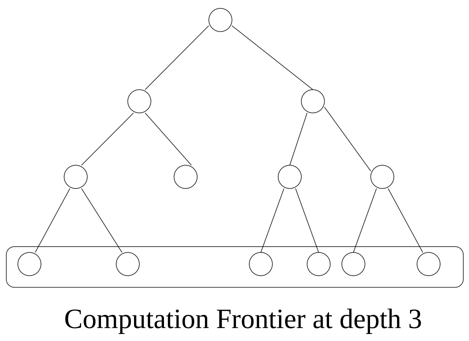

Figure 4. The computation frontier during computation phase

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

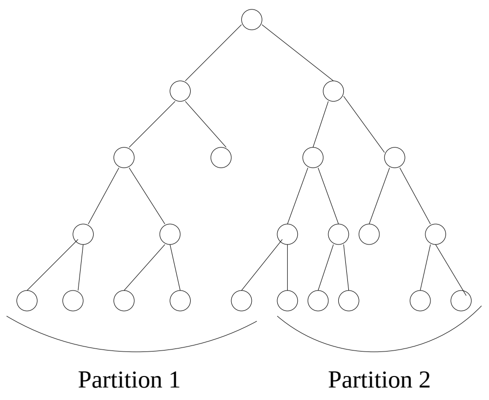

Figure 5. Binary partitioning of the tree to reduce communication costs

$$\sum(Communication\ Cost) \geq MovingCost + LoadBalancing$$

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.

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* [18], *SPRINT* [21], and *ScalParC* [13]. 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.

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 a decision tree
t_c	Unit computation time
t_s	Start up time of communication latency [KGGK94]
t_w	Per–word transfer time of communication latency [KGGK94]

Table 4. Symbols used in the analysis.

Existing parallel formulations of these schemes [21, 13] 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 [21, 13] 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 [12]. 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 [3] where quantiles [2] are used to discretize continuous attributes. The second example of this approach to discretize at each node is *SPEC* [23] 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 [23]. 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 [23].

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 [23]. The detailed study of the communication patterns used in this analysis can be found in [16]. 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 [16].
- 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:

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

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:

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

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 :

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

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 Equation 4 above is given in [23]. 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 [22] 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 [14]. 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 [14]. 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 [16]. 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_1. 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 Equations 3 and 4. The number of partition splits that a processor participates in is less than or equal to L_1- the depth of the tree.

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

Communication cost at each level is given by Equation 2 ($=\theta(\log P)$). 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 Equations 5 and 6.

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

Computation cost given by Equation 1 is:

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

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

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

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 Equation 9 to serial computation time.

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

Therefore, the isoefficiency function is $N = \theta(P \log P)$. Isoefficiency is $\theta(P \log P)$ 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 [18] for all our experiments. Ten classification functions were also proposed in [18] 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 [21] 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 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

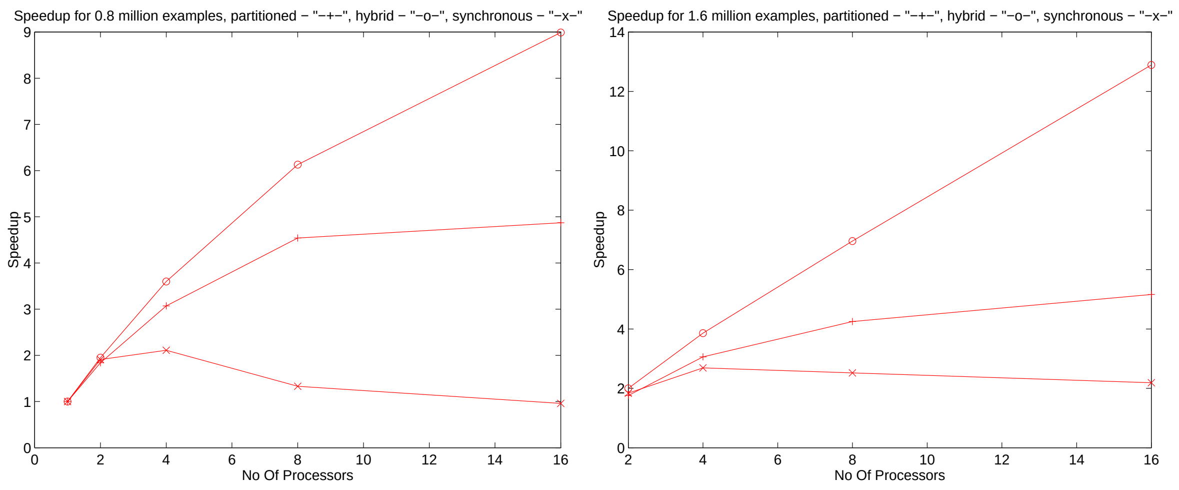

Figure 6. Speedup comparison of the three parallel algorithms.

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.,

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

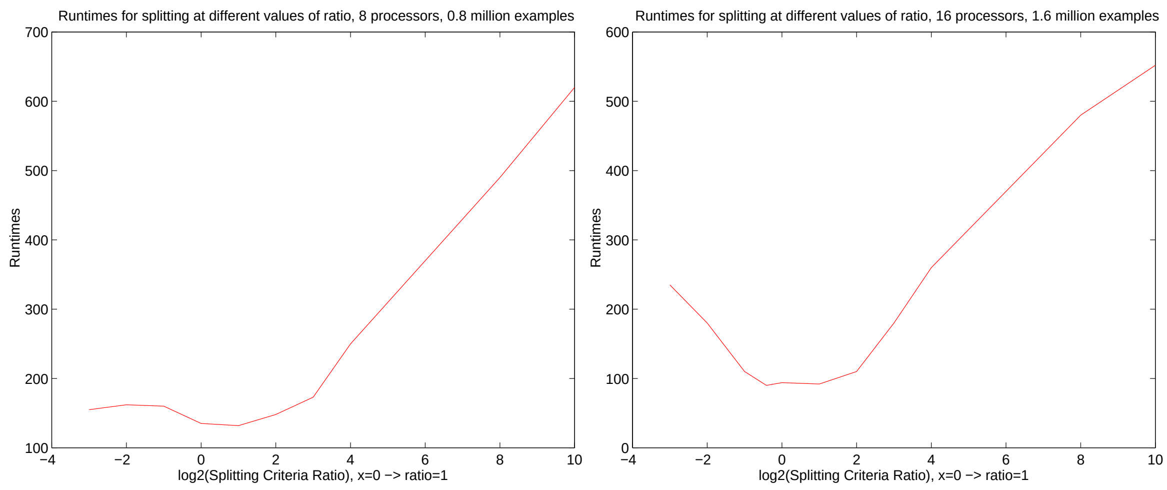

Figure 7. Splitting criterion verification in the hybrid algorithm.

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 technique to discretize continuous attributes at each decision tree node [23]. Note that the parallel formulation gives *almost identical* performance as the serial algorithm in terms of accuracy and classification tree size [23]. 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.

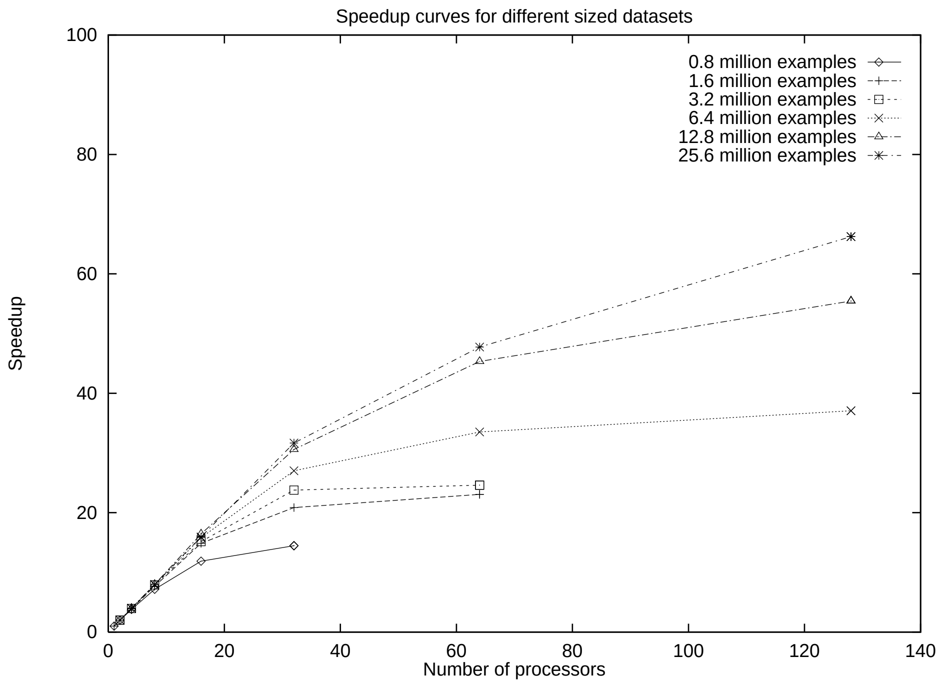

Figure 8. Speedup of the hybrid approach with different size datasets.

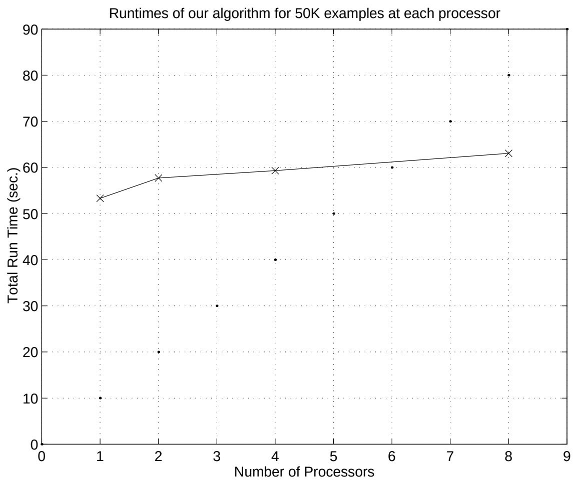

Figure 9. Scaleup of our algorithm

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-9634719, 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.

Notes

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 [16] section 3.7 for details.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Eng.*, 5(6):914–925, December 1993.
2. K. Alsabti, S. Ranka, and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *Proc. of the 23rd VLDB Conference*, 1997.
3. K. Alsabti, S. Ranka, and V. Singh. CLOUDS: Classification for large or out-of-core datasets. *http://www.cise.ufl.edu/~ranka/dm.html*, 1998.
4. L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth, Monterrey, CA, 1984.
5. J. Catlett. *Megainduction: Machine Learning on Very Large Databases. PhD thesis.* University of Sydney, 1991.
6. Philip K. Chan and Salvatore J. Stolfo. Experiments on multistrategy learning by metalearning. In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, pages 314–323, 1993.
7. Philip K. Chan and Salvatore J. Stolfo. Metalearning for multistrategy learning and parallel learning. In *Proc. Second Intl. Conference on Multistrategy Learning*, pages 150–165, 1993.
8. J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H. Huning, M. Kohler, J. Sutiwaraphun, H.W. To, and D. Yang. Large scale data mining: Challenges and responses. In *Proc. of the Third Int'l Conference on Knowledge Discovery and Data Mining*, 1997.
9. D.J. Spiegelhalter D. Michie and C.C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.
10. S. Goil, S. Aluru, and S. Ranka. Concatenated parallelism: A technique for efficient parallel divide and conquer. In *Proc. of the Symposium of Parallel and Distributed Computing (SPDP'96)*, 1996.
11. D. E. Goldberg. *Genetic Algorithms in Search, Optimizations and Machine Learning.* Morgan-Kaufman, 1989.
12. S.J. Hong. Use of contextual information for feature ranking and discretization. *IEEE Transactions on Knowledge and Data Eng.*, 9(5):718–730, September/October 1997.
13. M.V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proc. of the International Parallel Processing Symposium*, 1998.

14. George Karypis and Vipin Kumar. Unstructured tree search on simd parallel computers. *Journal of Parallel and Distributed Computing*, 22(3):379–391, September 1994.
15. R. Kufrin. Decision trees on parallel processors. In J. Geller, H. Kitano, and C.B. Suttner, editors, *Parallel Processing for Artificial Intelligence 3*. Elsevier Science, 1997.
16. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley, Redwod City, 1994.
17. R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(22), April 1987.
18. M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.
19. R.A. Pearson. A coarse grained parallel induction heuristic. In H. Kitano, V. Kumar, and C.B. Suttner, editors, *Parallel Processing for Artificial Intelligence 2*, pages 207–226. Elsevier Science, 1994.
20. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
21. J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd VLDB Conference*, 1996.
22. R. Shankar, K. Alsabti, and S. Ranka. Many-to-many communication with bounded traffic. In *Frontiers '95, the fifth symposium on advances in massively parallel computation*, McLean, VA, February 1995.
23. Anurag Srivastava, Vineet Singh, Eui-Hong Han, and Vipin Kumar. An efficient, scalable, parallel classifier for data mining. Technical Report TR-97-010,http://www.cs.umn.edu/∼kumar, Department of Computer Science, University of Minnesota, Minneapolis, 1997.
24. J. Wirth and J. Catlett. Experiments on the costs and benefits of windowing in ID3. In *5th Int'l Conference on Machine learning*, 1988.