Parallel Formulations of Inductive Classification Learning Algorithm

Vineet Singh
IBM T.J. Watson Research Center

Eui-Hong (Sam) Han, Anurag Srivastava, Vipin Kumar
Dept. of Computer & Information Sciences
4-192 EECS Bldg., 200 Union St. SE
University of Minnesota
Minneapolis, MN 55455, USA

Abstract

One of the important problems in data mining [SAD+93] is the classification–rule learning. The classification–rule learning involves finding rules or decision trees that partition given data into predefined classes. For any realistic problem domain of the classification–rule learning, the set of possible decision trees is too large to be searched exhaustively. In fact, the computational complexity of finding an optimal classification decision tree is NP–*hard*. All of the existing algorithms, like $C4.5$ [Qui93], CDP [AIS93] and $SLIQ$ [MAR96], use local heuristics to handle the computational complexity. The computational complexity of these algorithms ranges from $O(ANlogN)$ to $O(AN(logN)^2)$ with N training data items and A attributes. These algorithms are fast enough for application domains where N is relatively small. However, in the data mining domain where millions of records and a large number of attributes are involved, the execution time of these algorithms can become prohibitive, particularly in interactive applications.

In this paper, we present parallel formulations of classification–rule–learning algorithm based on induction. We describe two basic parallel formulation, 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.

Keywords: Data mining, parallel processing, classification, decision trees.

1 Introduction

One of the important problems in data mining [SAD+93] is the classification–rule learning. The classification–rule learning involves finding rules or decision trees that partition given data into predefined classes. For any realistic problem domain of the classification–rule learning, the set of possible decision trees is too large to be searched exhaustively. In fact, the computational complexity of finding an optimal classification decision tree is NP–$hard$. All of the existing algorithms, like $C4.5$ [Qui93], CDP [AIS93] and $SLIQ$ [MAR96], use local heuristics to handle the computational complexity. The computational complexity of these algorithms ranges from $O(ANlogN)$ to $O(AN(logN)^2)$ with N training data items and A attributes. These algorithms are fast enough for application domains where N is relatively small. However, in the data mining domain where millions of records and a large number of attributes are involved, the execution time of these algorithms can become prohibitive, particularly in interactive applications.

In this paper, we present parallel formulations of classification–rule–learning algorithm based on induction. We describe two basic parallel formulation, 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.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the existing algorithms for classification–rule learning. Section 3 describes the two basic parallel formulation. The hybrid method is described in Section 4 and its analysis is shown in Section 5. Section 6 presents our conclusions.

2 Sequential Classification Rule Learning Algorithms

Most of the existing induction–based algorithms like $C4.5$ [Qui93], CDP [AIS93] and $SLIQ$ [MAR96], use Hunt's method [Qui93] 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 one or more 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 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.

Case 3 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 $\{O_1, O_2, \ldots, O_n\}$. 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.

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]

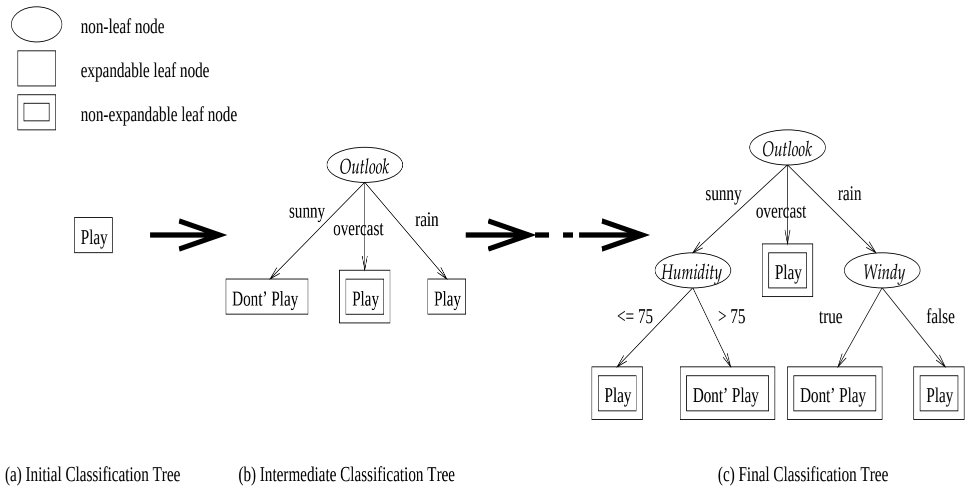

Figure 1: Demonstration of Hunt's Method

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 the Hunt's method works with the training data set. In the case 3 of the 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*. For a continuous attribute, binary test involving all the distinct values of the attribute is considered. Table 3 shows the class distribution information of data attribute *Humidity*. Once the class distribution information of all the attributes are gathered, the entropy is calculated based on either *information theory* [Qui93] or *Gini Index* [BFOS84]. One attribute with the most entropy gain 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 recursive partitioning of 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 values 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 attributes.

$SLIQ$ introduced a novel pre–sorting techniques to avoid costly sorting at each node. $SLIQ$ keeps a separate sorted list for each continuous attribute and a separate list called class list. An entry in the class list corresponds to a data item, and has a class label and name of the node it belongs in the decision tree. An entry in the sorted attribute list has an attribute value and the index of data item in the class list. $SLIQ$ grows the decision tree in breadth–first manner. For each attribute, it scans the corresponding sorted list and calculate entropy values of each distinct values of all the nodes in the frontier of the decision tree simultaneously. After the entropy values have been calculated for each attribute, one attribute is chosen for a split for each nodes in the current frontier, and they are expanded to have a new frontier. Then one more scan of the sorted attribute list is performed to update the class list for the new nodes.

3 Basic Parallel Formulation

Initially N training data items are randomly distributed to P processors such that each processor has N/P data items. At this point, all the processors cooperate to expand the root node of a decision tree. For this, processors need to decide on an attribute to use to generate child nodes of the root. Discrete and continuous attributes are treated differently.

For discrete attributes, there are three steps. In the first step, every processor collects the class distribution information of the local data for the discrete attributes only. In the second step, the processors exchange the local class distribution information using global reduction [KGGK94]. Finally, each processor can simultaneously compute entropy gains of the discrete attributes and find the best discrete attribute.

For continuous attributes, there are three steps for each attribute. In the first step, every processor separates the local continuous attribute values of each attribute into separate buckets, one for each class. In the second step, all processors cooperate to cluster all attribute values in each class separately using a standard clustering algorithm such as k-means [?] for a single pass over the data[1]. At the end of this step, all processors have the cluster results for each attribute for each class. In the third step, each processor finds the best entropy split for each continuous attribute using the clusters as opposed to the raw data.

After processing both discrete and continuous attributes, we can find the best attribute for tree expansion overall using either discrete or continuous attributes.

There are two approaches for further progress. In *Synchronous Tree Construction Approach*, the entire set of processors synchronously expand one node of the decision tree at a

[1] The k-means algorithm is modified slightly to make it more suitable for a one-pass scheme and enable it to run in parallel.

time. In *Partitioned Tree Construction Approach*, each new generated node is expanded by a subset of processors that helped the expansion of the parent node.

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. Figure 2 shows the overall picture. Major steps for the approach are shown below:

1. Select a node to expand according to a decision tree expansion strategy (eg. Depth–First, Breadth–First or Best–First), and call that node as the current node.

2. Perform the three steps given above for discrete and continuous attributes (separate steps for the two cases).

3. Based on the attribute values and the splitting condition (subset condition for discrete attributes or splitting value for continuous attributes), create child nodes and partition the data accordingly.

4. Based on the attribute values, create child nodes and partition the data according to the values of the attribute chosen.

5. Repeat the above steps (1–5) until no more nodes are available for the expansion.

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 reading in the local data, all the processors need to synchronize to construct the clusters. 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 by at most 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

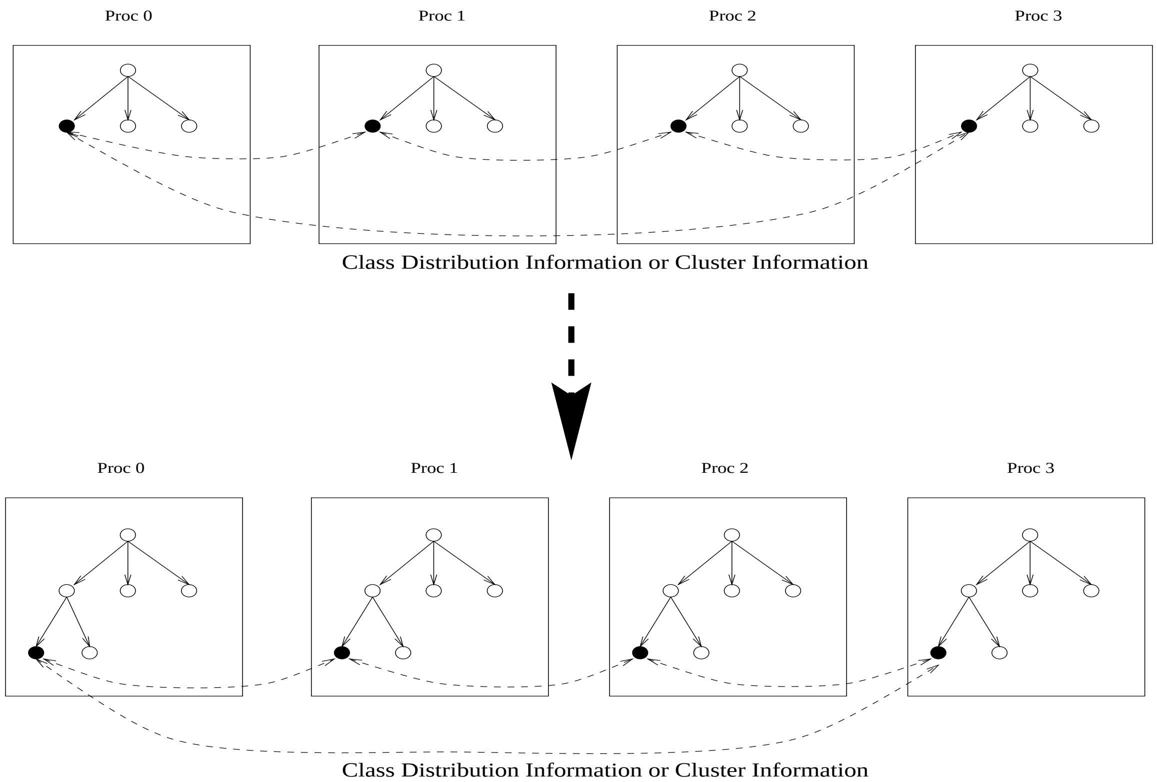

Figure 2: Synchronous Tree Construction Approach with Depth–First Expansion Strategy

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. Now the only source of load imbalance is when some leaf nodes become terminal nodes. This load imbalance can be further minimized if the training data set is distributed randomly.

3.2 Partitioned Tree Construction Approach

In this approach, each leaf node, n, of the frontier of the decision tree is handled by a distinct subset of processors $\mathcal{P}_{(n)}$. Once the node n is expanded into child nodes, $n_1, n_2, \ldots, n_k$, the processor group $\mathcal{P}_{(n)}$ is also partitioned into k parts, $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$, such that $\mathcal{P}_i$ handle node n_i. All the data items are shuffled such that the processors in group $\mathcal{P}_i$ have data items that belong to the leaf n_i only. Figure 3 shows the overall picture. Major steps for the approach is shown below:

1. Expand a node based on the method discussed in the beginning of the Section 3 for expanding the root node.

2. (a) If the number of leaf nodes is less than $|\mathcal{P}_{(n)}|$,

      i. Assign a subset of processor to each leaf node such that number of processors assigned to a leaf node is proportional to the number of the data items contained in the node.

ii. Shuffle the training data such that each subset of processors has data item that belongs to the leaf nodes it is responsible for.

iii. Processor subsets assigned to different nodes develop subtrees of the responsible nodes independently, by following the above steps recursively.

(b) Otherwise,

i. Partition leaf nodes into $|\mathcal{P}_{(n)}|$ groups such that each group has about the equal number of data items. Assign each processor to one node group.

ii. Shuffle the training data such that each processor has data item that belongs to the leaf nodes it is responsible for.

iii. Now the expansion of the subtrees rooted at a node group proceeds completely independently at each process.

3. At the end, the whole decision tree is constructed by combining subtrees of each processor.

The advantage of this approach is that once a processor becomes solely responsible for a node, it can develop a subtree of the decision tree independently without any communication overhead. There are a number of disadvantages of this approach. First disadvantage is that it requires data movement after each node expansion until one process becomes responsible for an entire subtree. The communication cost is particularly expensive in the expansion of the upper part of the decision tree. Once the number of nodes in the frontier exceeds the number of processors, then the communication cost becomes zero. Second disadvantage is due to load balancing. The load balancing at the time of node expansion is done based on the number of items on the leaf nodes. However, after an expansion of child nodes, subtree on a processor might be completely done while the other subtrees on other processors are still being processed. Then the processor which finished early does not have any work to do. Hence, we need to perform dynamic load balancing to divide up the work of other processors. This requires additional movement of data items.

4 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 number of processors as well as the current frontier of the decision tree is 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.

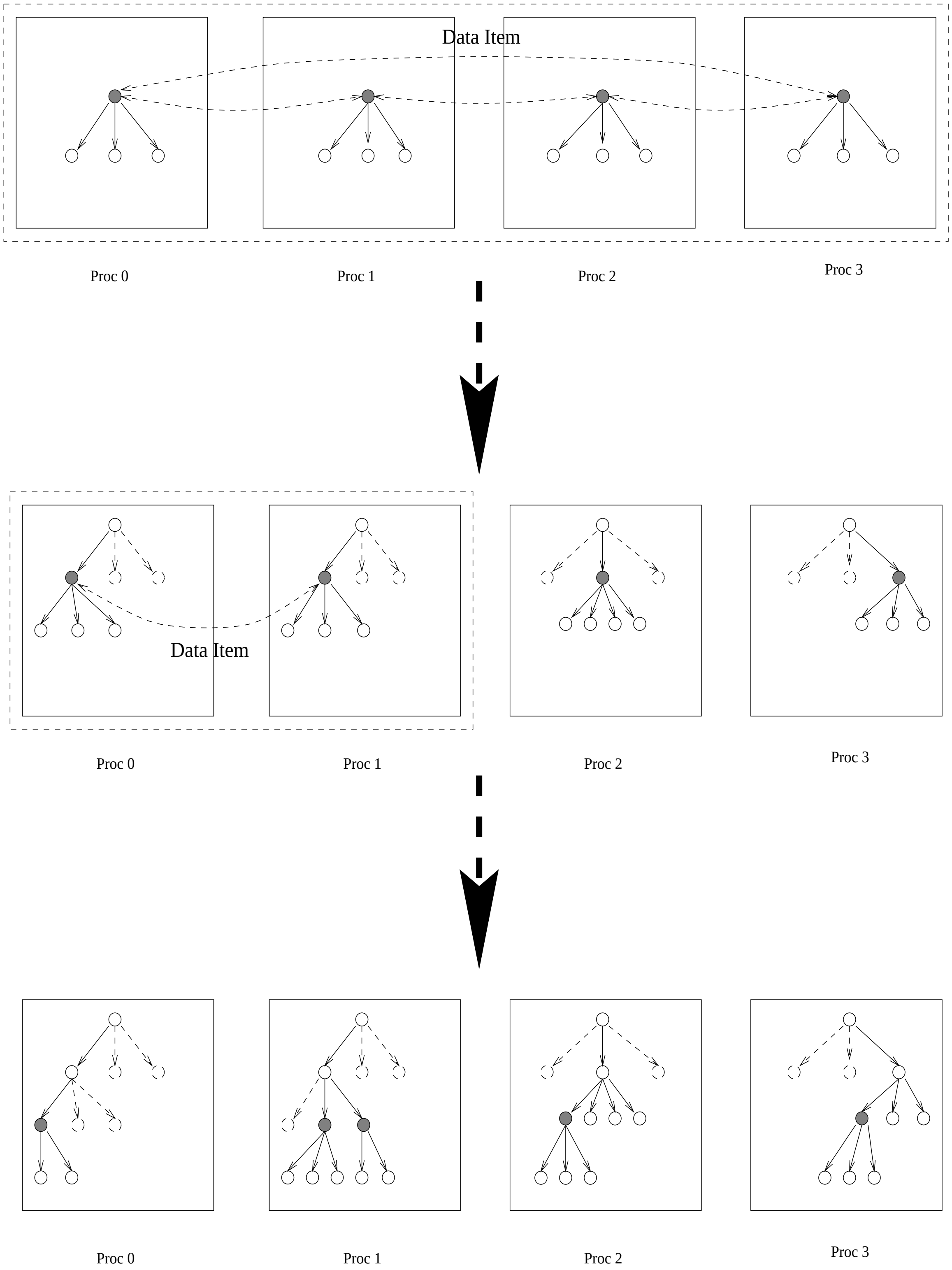

Figure 3: Partitioned Tree Construction Approach

- The test cases on which the classification algorithm has to be run are randomly sampled from the database at each processor. Thus if N is the total number of training cases, each processor has N/P training cases locally.
- Each processor works on its training cases and helps expand the decision tree common to the partition to which it belongs. Initially all processes belong to one global partition.

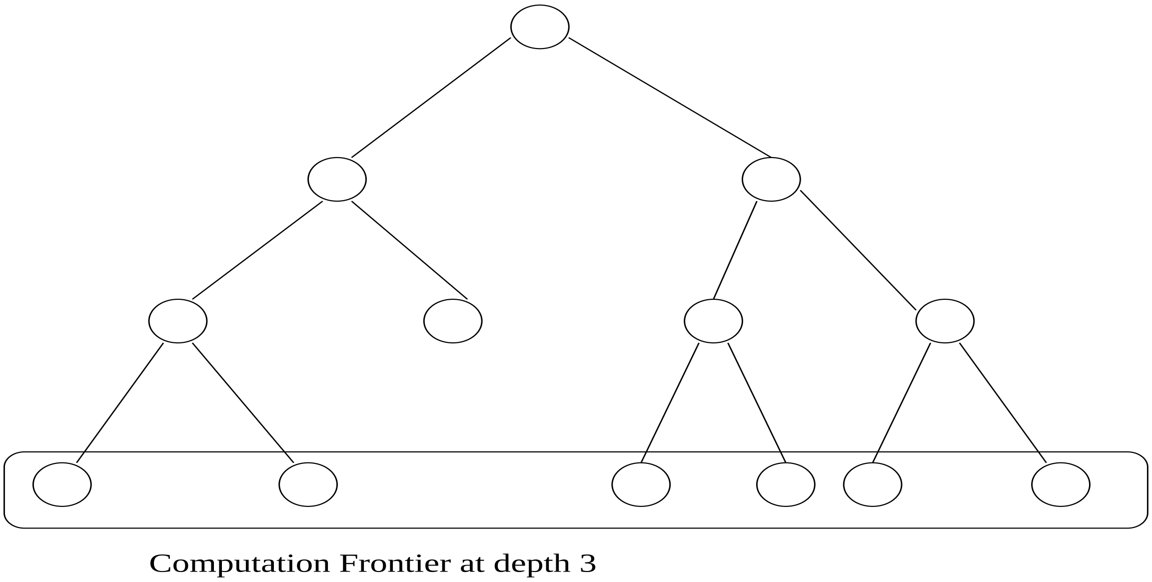

Figure 4: The computation Frontier during computation phase

- Broadly, the processing within each partition is divided into a series of computation and communication phases. Note that processors in different partition work asynchronously.
- In the computation phase, all the nodes at the frontier of computation, which encompass a level of the decision tree, that belong to one partition, are processed at a time. The class distribution information for all the nodes at a level of the decision tree are computed locally within each processor.
- At this point a global reduction communication operation, on the local training case counts at each processor is performed on all processors within a partition. Based on the global training case counts appropriate decision variables are chosen for expanding each frontier node in the decision tree located at each partition.
- As the depth of the tree increases the volume of statistics gathered at each level also increases. This is because statistics are generated for each node at a particular level. Very soon 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 data items in each partition is about the same.

- On a hypercube, each of the two partitions naturally correspond to a sub cube. First, corresponding processors within the two sub cubes exchange the data items. After this exchange, processors within each sub cube collectively have all the training items for their partition, but the number of training data items 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 equal number of data items.

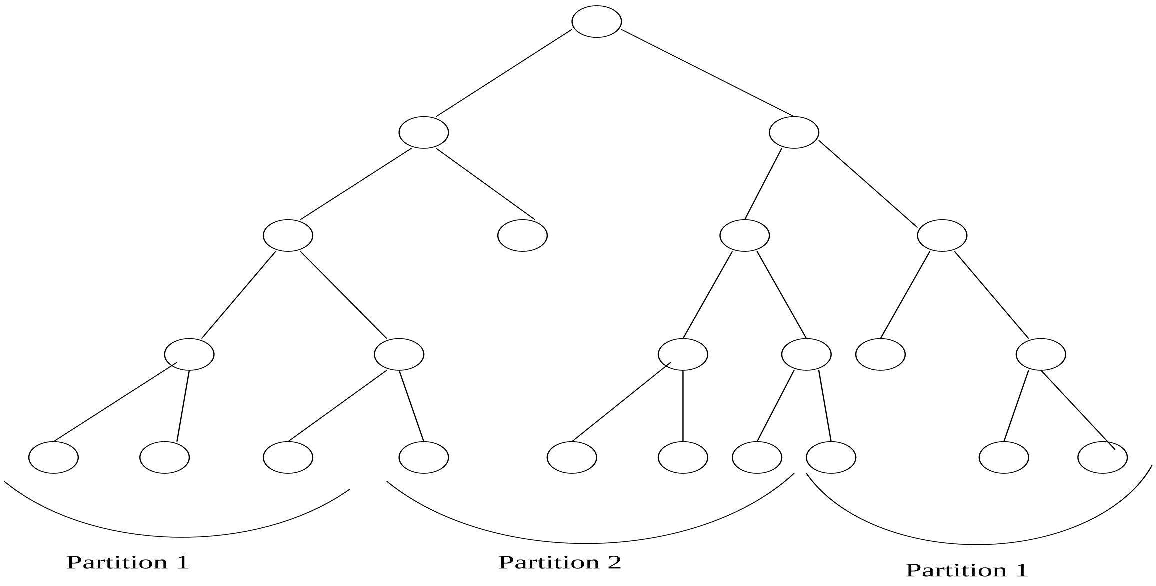

Figure 5: Binary partitioning of the tree to reduce communication costs

- 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 till a complete decision 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 samples for each processor in the donor partition to a processor in the receiving partition.

4.1 Reducing Processor idling

All the training cases, which have still to be classified by the decision tree, will be encountered on some node at the given level of the tree. Therefore in the compute phase all the processors have almost equal work to do at each level.

If a partition of processors becomes idle then it combines with a partition that is performing computation on a part of the tree. This combination is done with another partition of equal size.

symbol	definition
N	Total number of training sample
P	Number of processors
A	Number of attributes in the database
C	Number of classes in the database
M	Average number of distinct values in the discrete attributes
S	Average number of distinct values in the continuous attributes
L	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.

4.2 Reducing Communication Costs

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 decision tree). If partitioning is done too frequently, then the hybrid scheme will approximate the partitioned tree construction algorithm, 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.

Message buffers can grow to large sizes very soon as the number of leaves start doubling as one level of the decision tree is traversed. Count informations have to be sent for each leaf node. To reduce this message size only top few entries, with least entropies, are sent at each level. This can reduce the message size dramatically as the whole table need not be sent for each computation on a leaf. The combining of these top few entries has to be carefully done in order to minimize the chances of errors.

When communication costs become prohibitive, even after using the above mentioned optimization, binary partitioning of the tree is done so that the information exchange is localized to the processors in the partition. This partitioning is done, as shown in Figure 5, when the tree is of a sizeable depth.

5 Analysis of the Hybrid Algorithm

In this section, we provide the analysis of the hybrid algorithm proposed in Section 4. We provide a separate analysis for two different data attribute types, discrete and continuous. The analysis of the mixture of discrete and continuous attributes could be done easily based on the analysis provided here. The detailed study of the communication patterns used in this analysis can be found in [KGGK94]. Table 4 describes the symbols used in this section.

5.1 Case with Discrete Attributes Only

Here we give a detailed analysis of splitting for the case when only discrete attributes are present. At each leaf of a level, there are A attribute tables that need to be communicated. The size of each of these tables is the product of class size and the mean number of attribute values. Thus at each leaf the size of the communication buffer is :

$$\text{Communication buffer size } = C * A * M$$

The number of leaves at level L is M^L. Thus the size of the total data to be sent from each processor is :

$$\text{Processor Communication Buffer size} = C * A * M * M^L$$

At level L, the computation cost involves updating the count tables for each attribute, calculating entropy and then finding the best attribute for split is:

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

where t_c is the unit of computation for processing a single tuple.

At the end of local computation a synchronization involves a global reduction. The communication cost[2] is :

$$\text{Per level Communication cost} = (t_s + t_w * C * A * M^{L+1}) * \log P$$

In case a splitting of processors takes place, the processor space is divided into two equal partitions. Each processor sends records to a corresponding processor in the second partition. Due to an initial randomized distribution, on an average half the records move from one node to the other.

$$\text{Moving Cost} = t_s + t_w * \frac{N * K_1}{2 * P}$$

where $\frac{N*K_1}{2*P}$ represents the maximum number of excess elements present on a node. K_1 is a small constant, due to initial randomized distribution. In any case, K_1 is between 0 and 2.

After the partitions have been created, load balancing has to be done in each partition. This involves two communication operations. In the first step an all to all broadcast of the number of records at each node is done. After this, using a bin packing algorithm for ensuring equal number of records at each node, the data to be shipped out is found out at each node. The bin packing algorithm is so designed that it minimizes the data movement costs and distribute communication overhead equally among all processors. One of the schemes could be to separately sort the receivers and senders of data. This sorting and matching of receivers and senders can be done in order to reduce communication costs.The number of records to be sent out or received is on the average a fixed fraction of local records.

[2] 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 [KGGK94] section 3.7 for details.

Note that before the load balancing step, the number of items at each processor range from 0 to $2 * \frac{N}{P}$ and the average number of items desired at the end at each processor is $\frac{N}{P}$. Hence no processor has to send or receive more that $\frac{N}{P}$ elements. In absence of conflict all the data arrangement can be done in time $\frac{N}{P} * t_w$ (ignoring t_s term) time. However conflict can arise if more than one processor tries to send the data to the same processor at the same time. In this case one of the processor has to wait. We show that using a simple scheme which separates senders and receivers of data, the communication time can be bound by $\frac{2*N}{P} * t_w$.Initially senders and receivers are listed in order. A group of consecutive receivers are allocated to every sender.

A sender may have to send to an arbitrary number of receivers (up to P). However if a sender sends data to more than 2 receivers, then it can have contention with other senders only at the first and the last receivers in its list. Each sender can send out a maximum of $\frac{N}{P}$ data. Let time spent by a sender on the first receiver be :

$$C_1 * \frac{N}{P} * t_w, \text{where } C_1 \leq 1.$$

Note that in the worst case almost all of the time might be spent by this sender might be waiting on the receiver which is receiving data from other senders. Let time spent by a sender on the middle receivers be:

$$C_2 * \frac{N}{P} * t_w, \text{where } C_2 \leq 1.$$

During all this time the sender sends actual data. When the sender reaches the last receiver, two cases are possible:

- Receiver is busy receiving some data. The receiver can receive a maximum data of size $\frac{N}{P}$. Since the receiver must have been receiving data for time $(C_1 + C_2) * \frac{N}{P} * t_w$. It can receive data from this sender for :

  $$\text{Max time spent by the sender waiting/sending to last receiver} = (1 - C_1 - C_2) * \frac{N}{P} * t_w$$

  $$\text{Total Time for any sender} = \frac{N}{P} * t_w$$

- Receiver is idle. The sender can send :

  $$\text{Max data that can be sent out from sender} = (1 - C_2) * \frac{N}{P}$$

  since $C_2 * \frac{N}{P}$ amount of data has already been sent to middle receivers and in the worst case very little data might have been sent to the first receiver.

  $$\text{Total time for any sender} = (1 + C_1) * \frac{N}{P} * t_w$$

So the upper bound to the load balancing cost is $2 * \frac{N}{P}$, since the maximum value of C_1 is 1. When sorting is done the worst case bound remains the same but we expect to reduce the number of receivers to whom the data has to be sent by a single sender.

$$\begin{aligned}\text{Communication Cost for Load Balancing} &= \text{all to all broadcast} + \text{movement of records} \\ &= t_s \log P + t_w * (P-1) + k_1 * (t_s + t_w * \frac{N}{k_2 * P})\end{aligned}$$

where k_1 and k_2 are constants and $\frac{k_1}{k_2} \leq 2$ and $k_1 \leq P$ from the above discussion.

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

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

This splitting is recursive and is applied as many times as required. Once splitting is done, the above computations are applied to each group. When a group of processors starts to idle then it sends a request to a busy group about its idle state. This request is sent to a group of processors of roughly the same size as the idle group. During the next round of splitting the idle group is included as a part of the busy group and the computation proceeds on as described above.

5.2 Case with Continuous Attributes Only

We handle continuous attributes by clustering attribute values and using the centroids of clusters as landmarks in finding possible splitting points. So in our parallel formulation, where breadth first expansion is assumed, following steps are taken for each level:

```
1 for each node in the frontier
2    read in data block of size B;
3    for each attribute
4        partition the data according to class and sort these data globally
5        select K attribute values per class as initial centroids globally
6        construct cluster tree per class with these K centroids as leaves;
7    while data is available
8        read in data block of size B and update the cluster trees;
9        send/receive K leaves of the each cluster tree using global reduction;
10       update/reconstruct the cluster trees;
11   calculate the entropy for all the attributes;
12   find the best attribute to cut;
```

The dominant cost of steps 4 to 6 is global sorting involving P processors and B number of items per processor. Using parallel sample sort [KGGK94], the overall computation cost of step 4 is:

$$2^L A(B \log B + P(\log P)^2 + P \log B)$$

and the overall communication cost is:

$$2^L A(B + P \log P)$$

The size of the communication buffer for step 9 is $2^L ACK$, since there are 2^L number of nodes in level L of the frontier and there are AC number of cluster trees with K leaves on each node. The number of loop iteration of step 8 is $\frac{N}{P2^L B}$ assuming that each node has $\frac{N}{P2^L}$ data items. The overall communication cost of step 9 is:

$$\frac{N}{P2^L B}(t_s + t_w 2^L ACK) \log P$$

Then, the overall communication costs of step 4 and step 9 is:

$$\text{Communication Cost} \quad = \quad 2^L A(B + P \log P) + \frac{N}{P2^L B}(t_s + t_w 2^L ACK) \log P$$

Assuming that each node has $\frac{N}{P2^L}$ data items and there are AC number of cluster trees per node, the overall computation cost of step 8 is:

$$2^L AC \frac{N}{P2^L} \log K \quad = \quad AC \frac{N}{P} \log K$$

The overall computation cost of step 11 and 12 is $2^L ACK$. This gives the overall computation cost as follow:

$$\text{Computation Cost} \quad = \quad (2^L A(B \log B + P(\log P)^2 + P \log B) + AC\frac{N}{P} \log K + 2^L ACK)t_c$$

The cost of splitting leaf nodes and moving data for load balancing is the same as that of the analysis of the discrete attributes only.

6 Conclusion

In this paper, we proposed three parallel formulations of inductive-classification-learning algorithm. The *Synchronous Tree Construction Approach* performs well if the classification tree is shallow. In the shallow level of the tree, there are relatively large number of training data items at the nodes and thus the communication overhead is relatively small. However, as the tree deepens, the number of training data items at the nodes decreases. Hence this approach suffers from high communication overhead and load imbalance. The *Partitioned Tree Construction Approach* works the best if the tree is deep. This approach pays a big communication overhead in the shallow levels of the tree as it has to shuffle lots of training data items to different processors. Once every node is solely assigned to 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 partition of the data was done statically.

The hybrid approach combines the good features of these two approaches to reduce communication overhead and load imbalance. In this approach, when the shallow levels of decision tree is constructed, the *Synchronous Tree Construction Approach* is used. Since there are relatively large number of the training data items in the nodes, 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*. The load imbalance is handled by allowing an idle processor group to join other processor groups when the partition of training data items is performed.

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 might be uniform in depth while other trees might be skewed in one part of the tree. The hybrid approach adapts well to all different shapes of the classification tree. If the decision tree is shallow, 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* if the tree is deep. If the tree has a big variance in depth, the hybrid approach will perform dynamic load balancing with processor groups such that there are not many idle processors.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engg.*, 5(6):914–925, December 1993.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterrey, CA, 1984.

[Hon94] S.J. Hong. Use of contextual information for feature ranking and discretization. Technical Report RC 19664, IBM Research Division, 1994.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley, Redwod City, 1994.

[KK92] G. Karypis and V. Kumar. Unstructured tree search on SIMD parallel computers. Technical Report TR-92-021, Department of Computer Science, University of Minnesota, Minneapolis, 1992.

[MAR96] 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.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[SAD+93] M. Stonebraker, R. Agrawal, U. Dayal, E. J. Neuhold, and A. Reuter. DBMS research at a crossroads: The vienna update. In *Proc. of the 19th VLDB Conference*, pages 688–692, Dublin, Ireland, 1993.

Contents

List of Figures

List of Tables