An Efficient, Scalable, Parallel Classifier for Data Mining

Anurag Srivastava†
Dept. of Computer Science
University of Minnesota

Vineet Singh
IBM T.J. Watson Research Center

Eui-Hong (Sam) Han, Vipin Kumar
Dept. of Computer Science
University of Minnesota

Abstract

One of the important problems in data mining is classification. Recently there has been a lot of interest in classifying disk resident datasets. Experiments show that classifying large datasets leads to improvement in accuracy. Most of the current algorithms are suited for small datasets and are inefficient for large disk resident data sets. SPRINT is a state of the art decision tree based algorithm for disk resident data. We propose a new decision-tree-based algorithm which achieves essentially similar accuracy as SPRINT and gives much better performance. Our algorithm has absolutely no memory restrictions compared to SPRINT which has some memory restrictions. Our algorithm also does significantly less I/O than SPRINT.

We also present a parallel formulation of our serial algorithm which is inherently more scalable than parallel-SPRINT. An analysis of the cost of computation and communication of our parallel formulation is done. Our analysis and experimental results on IBM SP2 show that our parallel formulation exhibits superior performance and scalability than parallel-SPRINT.

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

†**Contact Address:**

Dept. of Computer Science
4-192 EE/CS Bldg., 200 Union St. SE
University of Minnesota
Minneapolis, MN 55455, USA
Tel: (612) 626–7515
Fax: (612) 624–6539
Email: anurag@cs.umn.edu

Contents

1 Introduction

Classification is an important and a well studied data mining problem. Recently there has been a lot of interest in classifying disk resident datasets because working with large dataset has been shown to improve the accuracy of classification [CS93a], [CS93b].

A classification problem has an input dataset called the training set which consists of a number of examples each having a number of attributes. Each example in the training set has a class label associated with it. The attributes are of two kinds *continuous*, having an ordering among themselves and *categorical*, which have no inherent order in them. The objective is to build a classification model that can be used to classify new data, based upon the attribute values. Applications include fraud detection from credit card data, bankruptcy prediction, etc..

Several classification models like neural networks [Lip87], genetic algorithms [Gol89], decision trees [Qui93] have been proposed. Decision trees are perhaps the most popular classification models. They are relatively computationally inexpensive to construct. They also obtain comparable accuracy on datasets compared to other methods [MST94].

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. Parallelization of these algorithms is also extremely important to be able to handle huge amounts of data within reasonable time.

One way to reduce the computational complexity of decision tree classifier on large datasets is to use only a small sample of training data to construct the decision tree. Such methods do not yield the same classification accuracy as a decision tree classifier that uses the entire data set. [WC88].

Most current algorithms classification algorithm are based on ID3 algorithm [Qui93]. C4.5 [Qui93] is one of the commonly used algorithms. But it is not suitable for large disk resident data sets because it requires all the training set to be memory resident and requires sorting at each node. SLIQ classifier [MAR96] is designed to be more suitable for large datasets. But the algorithm requires a data structure proportional to the number of records to stay memory resident. This restriction puts a hard limit on the amount of data that can be handled by this classifier. SPRINT [SAM96] algorithm is a recently developed classification algorithm which is more scalable than SLIQ. But it also has a hash table data structure proportional to the *number of tuples associated with the present decision tree node*. This data structure should stay memory resident; otherwise the algorithm suffers serious I/O overheads. The parallelization of this algorithm is inherently unscalable as it requires expensive global communication (proportional to number of examples) at each level of the decision tree.

We present a new algorithm that is completely scalable, as it removes any memory restrictions on the algorithm. The accuracy obtained by our decision tree classifier is essentially identical to the general class of decision tree classifiers including SPRINT. Analysis as well as experiments show that our algorithm gives better performance than SPRINT incurring much less I/O cost. The design of the algorithm makes it highly parallel. We describe two schemes of parallelizing the tree building process and then come up with a hybrid of these two schemes which shows excellent scaleup, speedup and size up properties.

The rest of the paper is organized as follows. We briefly explain decision tree classifiers in section 2 and also give a brief overview of SPRINT. Section 3 talks in detail about our algorithm and the results, after that we compare our algorithm to SPRINT in section 4. Section 5 describes the parallel formulations, we present the results for parallel formulation in Section 6 and conclude with the conclusions in section 7.

2 Decision-Tree Classification

Decision tree classifiers are class discriminators which recursively partition the training set to get nodes belonging to a single class. Here we provide a quick overview of the ID3 type decision tree classifier. For more details, the reader is referred to [Qui93], [SAM96]. Figure 1 shows a decision tree constructed based on the training set shown. Each node of the decision-tree is either a

1. a leaf indicating a class, or

2. a decision node which specifies a test on a single attribute value.

tid	*Car*	*Age*	*Salary*	*Class*
0	family	30	65	G
1	sports	23	15	B
2	sports	40	75	G
3	family	55	40	B
4	truck	55	100	G
5	family	45	60	G

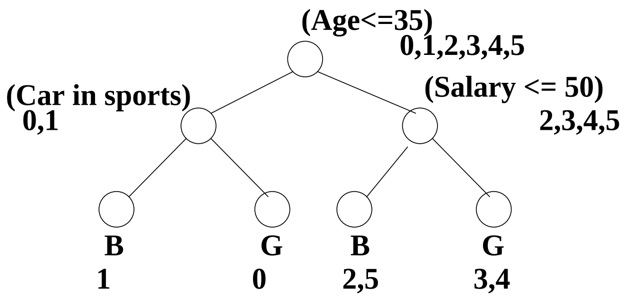

Figure 1: Decision tree approach

The two major phases of classification process are tree building and pruning. In tree building two major operations are done: i) evaluation of split point; ii) distributing the examples among the children nodes created based on the splitting test.

The splitting attribute is chosen so as to minimize the combined entropies of the new nodes created as a result of the split. The combined entropy is calculated as the weighted sum of the individual entropies of child nodes weighted with proportion of original examples contained by each one of them. The evaluation of split point for categorical and continuous attributes is shown in Figure 2.

2.1 How the Split is Performed ?

- *Categorical attributes* Class histogram is built up, and all possible subsets of values are evaluated for finding a least entropy split point. Split test is of the form "value of attribute A is a subset of possible values of attribute A". The splitting can be binary or multi way.

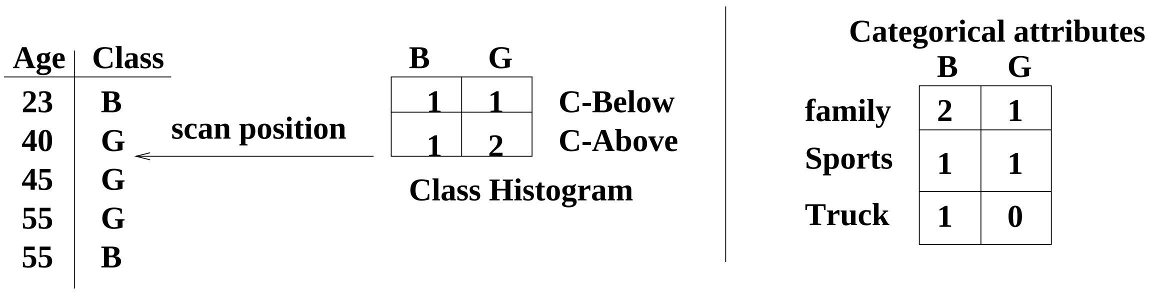

Figure 2: Finding Split points for categorical and continuous attributes

- *Continuous Attributes* A continuous splitting attribute has the test of form "value of attribute A $\leq$ some constant". Most of the current algorithms like SLIQ, SPRINT, C4.5 sort the data and then explore each possible distinct point as a possible splitting value. The splitting value with the least entropy is chosen.

Figure 2 shows the processing for categoric and continuous attributes.

Almost all of the decision tree algorithms use sorting of continuous attribute to consider each possible split point of data. We refer to these schemes as *sorting based schemes.* Let a potential split point divide a dataset D of size C into datasets D_1 and D_2 of sizes C_1 and C_2. The entropy of this split is $Entropy_{split}(D) = \frac{C_1}{C} Entropy(D_1) + \frac{C_2}{C} Entropy(D_2)$.

The second phase called the pruning phase prunes the tree to avoid the tree from over-fitting the data in order to remove statistical variations and noise. We will not discuss this further in this paper as the pruning phase of our new algorithm is similar to that in SPRINT.

2.2 SPRINT Overview

Since SPRINT is the state of the art for large data sets, it will be used as a comparative algorithm throughout. In SPRINT sorting of continuous attributes is done only once in the beginning of the tree building process. As a node is partitioned into child nodes, the values for a continuous attributes remain sorted.Attributes are stored in form of attribute lists which are stored on a disk for large datasets and are brought into main memory when required. All the attributes lists are accessed in sequential manner while they are being brought from disk.

tid	*Car*	*Age*	*Salary*	*Class*
0	**family**	**30**	**65**	**G**
1	**sports**	**23**	**15**	**B**
2	**sports**	**40**	**75**	**G**
3	**family**	**55**	**40**	**B**
4	**truck**	**55**	**100**	**G**
5	**family**	**45**	**60**	**G**

Age	**Tid**	**Class**
23	**1**	**B**
30	**0**	**G**
40	**2**	**G**
45	**5**	**G**
55	**3**	**B**
55	**4**	**G**

Figure 3: Attribute lists in SPRINT

Each row in the attribute list table has a class label and a tid in addition to the attribute value. The class label is required during the split evaluation phase, to calculate entropy for the splitting point under consideration. Since the attributes of a tuple are separated into attribute lists and these lists are reordered due to pre-sorting, a tid has to be stored in each row. These tid's can be used to reconstruct the original example, so that the complete example can be assigned to a decision tree node during partitioning of the data. Figure 3 shows the attribute lists.

When the partitioning of datasets corresponding to the nodes is done,the splitting attribute can be directly split by applying the splitting test to it. For splitting the rest of the attributes a tid hash table (Figure 4) is created using the splitting attribute. This tid hash table is used to find to which child node an example goes. The tid hash table should fit into the memory or else repeated accesses to the disk resident hash table may be required.

Tid	Child
0	**Left**
1	**Right**
2	**Right**
3	**Left**
4	**Left**
5	**Left**

Figure 4: Hash tables in SPRINT

3 Our New Formulation

We first give an idea of how our algorithm works and then present details later.

3.1 Overview

Our algorithm deals with the problem of finding split points for continuous attributes in a different manner than used in sorting based algorithms such as SPRINT. For categorical attributes, it builds up the class histogram and evaluates the possible subsets as is done in other algorithms like C4.5, SLIQ and SPRINT. For continuous attributes it performs clustering separately on each classes and then finds cutpoints with least entropy measure. Only binary splits are allowed in the current version of our algorithm. The nodes are processed one at a time. The tree is grown in breadth first manner.

3.2 Storage and Data Structures

The table containing input data is vertically partitioned into separate attribute columns and a class column. These are stored sequentially on disk. Note that these "attribute lists" are not same as those used in SPRINT. Each attribute list just contains a single attribute value for each example as shown in Figure 5. We do not need to keep tid's as the data is not reordered. A single class list is used for all attributes.

Windy?	Temp	Class
true	75	P
false	85	DP
true	80	P
true	60	P
false	65	P
true	72	DP
true	70	DP

Windy?
true
false
true
true
false
true
true

Temp
75
85
80
60
65
72
70

Class
P
DP
P
P
P
DP
DP

Figure 5: Vertical partitioning of dataset into attributes lists and class list

In order to evaluate the split points, class histograms have to be maintained. For categorical attributes class histogram are maintained for split point evaluation phase as shown in Figure 2.

These hold the class distribution for an attribute after the corresponding attribute and class lists are scanned. For continuous attributes two class histograms for values below and above a certain cutpoint are maintained as shown in Figure 2. Only one set of these class histograms for categorical and continuous attributes required at a time as the nodes are processed one after the other. These histograms are reused for other nodes.

A clustering module for each class of a continuous attribute holds the cluster information for a fixed number (16 in our implementation) of centroids. Each centroid structure holds value of the centroid, variance, count of elements and class labels.

3.3 Processing of Data for Finding Split Point at Each Node

- A dataset is separated into "attribute lists" for the attributes and the class.
- Each node is processed one attribute at a time.
- The tree is expanded in a breadth first manner.
- Each attribute list is processed by reading the attribute list and the class list sequentially. Note that a tuple has its attribute and its class value at the same position in both the lists, as the original order has not been violated.
- If the attribute is categoric then the class histogram is updated based on attribute and class value. Once the attribute list has been completely read in, a sub setting algorithm, on the distinct values, is run to find the least entropy test.
- If the data is continuous it is processed as explained in Section 3.4.

3.4 Processing of Continuous Attributes at a Node

While processing each continuous attribute, separate buckets are created for each class. An element is assigned to a particular bucket based on its class. This element is processed inside its bucket by a one pass clustering algorithm.

The clustering algorithms we have used is a modified version of K-means clustering [KR90]. The element is assigned to one of a fixed number of cluster centroids based on its distance from the centroids. These centroids are rearranged and processed after a fixed block of data has been read. If a centroid has number of elements below a certain threshold, another centroid with large variance and large number of elements is split to create two centroids. The elements of the centroid being split are equally split between the two rearranged centroids. The elements of the old centroid are merged with a centroid which is nearest to them.

At the end, clusters for each class are collected and projected on the number line. Each point has with it its centroid value, variance, count of elements in it and its class. A cutpoint which minimizes entropy based on these clusters is found. If the clusters do not overlap then the local minimas for entropy will occur at the separation of the clusters representing different classes. In our implementation we consider the midpoints of the projected centroids as potential cut points. Generally clusters overlap so while evaluating a cutpoint, the variances of clusters in its neighborhood are also taken into account. A uniform probability distribution of elements around the centroid is assumed for entropy calculation. Figure 6 graphically depicts the continuous attribute processing.

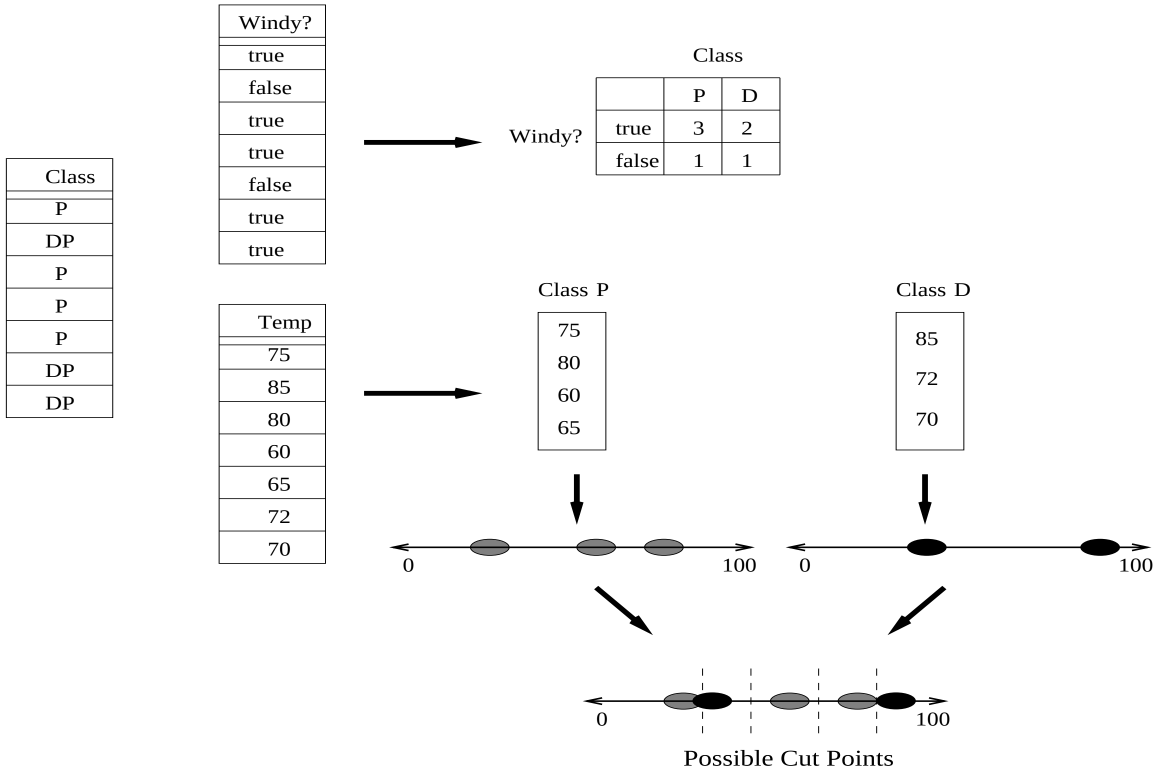

Figure 6: Getting split point for continuous attributes

3.5 A discussion of our cutpoint finding scheme

The process of classification using entropy minimization techniques is done so as to reduce the entropy or randomness present in the initial data. The intial dataset has random occurence of different classes. A decision tree classifier tries to partition the data so that the datapoints belonging to different classes get "separated".

A test on a numeric value is of the form attribute $A_i \leq X$, where X is a constant. Consider the sorted list of values for a particular continuous attribute, intuitively this number X should lie on a boundary point separating runs of two different classes. This is required so that "maximum advantage" can be got out of this test. Conversely consider the case when X occurs at an intermediate point in a class run R_x. It can be seen that atleast one more decision node tree will be required in order to separate elements of this class run R_x as shown in Figure 7. This extra decision tree node increases the size of the decision tree without gaining anything by the way of accuracy.

[FI93] shows, based on the mathematical definition of entropy, that there is no need for evaluating each distinct value of the sorted continuous attribute list as a potential cutpoint. Entropy can be minimized by only considering those points which lie at the crossover of class runs.

Our cutpoint algorithm for finding least entropy tests for numerical attributes is based on the concept of clustering. Our clustering scheme reduces to the above described sorting based schemes if we consider every possible run of each class as a cluster. For a given dataset at a particular level, clustering with a fixed number of centroids makes the assumption that relatively small sized runs of a class can be ignored. It considers significantly large runs for finding the least entropy cutpoint. This cluster could be a single large run or could be formed by merging close together small runs of a particular class to form a single large run for that class.

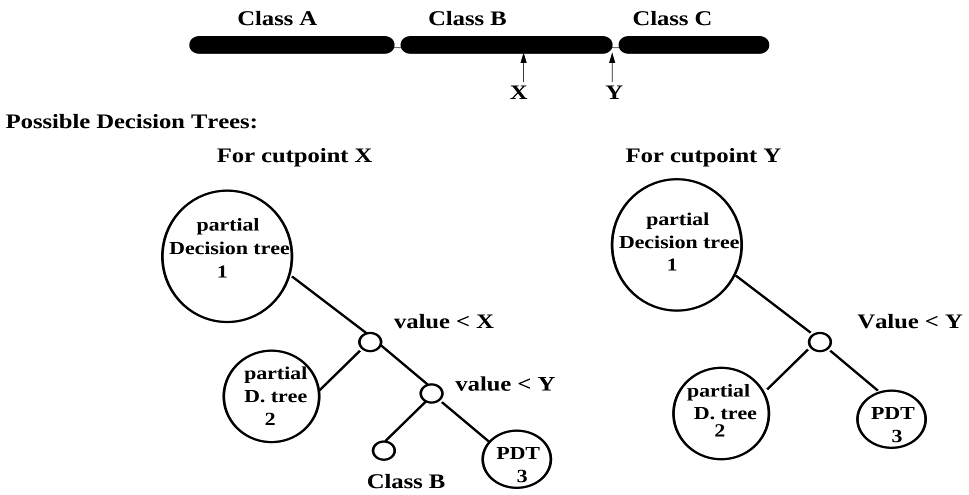

Figure 7: A comparison of decision trees generated by different cut points on the numerical attributes

3.6 Partitioning the Data

The splitting attribute is partitioned based on the splitting test. It generates a string of 0s and 1s to be used as splitting table for other attributes. Note that this strings needs to be scanned only serially for the non splitting attributes in order to split them.

4 Implementation Status and Results

In our implementation we have not optimized performance by fitting the disk resident data structures into main memory. We expect the performance of our algorithm to improve further when we utilize the total memory of the system. As of now the implementation uses the least memory possible. The disk resident data structures are implemented as buffered files with a fixed buffer of 4k. Full MDL based pruning in the SLIQ paper[MAR96] is used on the decision trees generated. The accuracy, decision tree size, and the run times of other algorithms has been taken from the SLIQ paper [MAR96] for comparison purposes. All the experiments were run on a 66MHz RS6000 workstation with 64MB main memory running AIX 4.1.

The results are shown for the statlog data sets, a commonly used benchmarking dataset for classification algorithms.

4.1 Results

Table 1 shows classification accuracy. Our algorithm performs essentially identical to all the other widely used tree classifiers. It's accuracy is almost identical (within 0.9%) to SPRINT.

Tree Sizes are compared in Table 2. We have used Full MDL pruning strategy [MAR96]; therefore the results of the rest of the algorithms are also for the same pruning scheme. The final tree sizes after pruning are essentially similar to SPRINT and better than the others.

Table 3 shows the execution times. The execution time comparison is shown to give a relative idea. The comparison is not entirely valid, as the the results for other algorithms are for an experimental set up of an IBM RS/6000 250 workstation with 64 MB main memory and executing the AIX 3.2.5 OS. These timings are again taken from [MAR96]. Since we did not have access to SPRINT we could not run it to get the run times on our system. These datasets have only been

Datasets	IND-Cart	IND-C4	SPRINT	Ours
Australian	85.3	84.4	84.9	84.6
Diabetes	74.6	70.1	75.4	74.5
DNA	92.2	92.5	92.1	92.3
Letter	84.7	86.8	84.6	84.6
Satimage	85.3	85.2	86.3	85.8
Segment	94.9	95.9	94.6	95.0
Shuttle	99.9	99.9	99.9	99.9

Table 1: Classification accuracy for different algorithms [MAR96]

Datasets	IND-Cart	IND-C4	SPRINT	Ours
Australian	5.2	85	14.6	13
Diabetes	11.5	179.7	35	33
DNA	35	171	55	54
Letter	1199.5	3241.3	1141	1089
Satimage	90	563	159	133
Segment	52	102	18.6	19
Shuttle	27	57	29	21

Table 2: Pruned Tree Sizes (Full Mdl Pruning)[MAR96]

used for accuracy and tree size comparisons since these are small datasets which will not show the effect of I/O improvements evident for large data sets.

Datasets	IND-Cart	IND-C4	SLIQ	Ours
Australian	2.1	1.5	7.1	6.7
Diabetes	2.5	1.4	1.8	1.4
DNA	33.4	9.21	19.3	21
Letter	251.3	53.08	39.0	61.0
Satimage	224.7	37.06	16.5	21.1
Segment	30.2	9.7	5.2	12.1

Table 3: Run Times[MAR96]

The tests on synthetic data sets were carried out for the synthetic datasets proposed in [AIS93] and also used by SPRINT. The accuracies were essentially similar. The run times for two different sizes are as follows: i) For 0.4 million examples SPRINT: 1000sec, Ours: 405 sec, ii) 0.8 million examples SPRINT 2100 sec, Ours 766 sec.. Again the runtime comparison is not easy as the machines have different OS versions and might be different in some other respects also. It can be seen that for similar architectures our algorithm is 2.5 - 3.0 times faster on the two datasets for which figures are given above.

5 Comparison of SPRINT and Our Serial Algorithm

5.1 I/O Requirements

We analyze the Storage and I/O requirements for our algorithm and compare it with SPRINT.

- I/O requirements of SPRINT.
  In SPRINT for processing each tuple for each attribute in the split point finding phase, attribute list row has to be read from the disk.

  Finding Split point for nodes

  I/O per tuple per attribute =Sizeof(Attribute Value)+Sizeof(tid)+Sizeof(Class Value).

  Splitting Phase

  for *partitioning a non splitting attribute.*
  We assume the best case scenario that multiple scans of the hash table are not done).

  I/O done per tuple per attribute = I/O for one attribute list row+ I/O for hash entry

  = SizeOf(Attribute Value)+Sizeof(tid)+Sizeof(Class Value) + Sizeof(tid) +Sizeof(splitting boolean value).

  Writing requires I/O per tuple= Sizeof(Attribute Value) + Sizeof(tid) + Sizeof(Class Value);

  The Attribute list structure and hash tables for SPRINT are shown in figure 3 and 4.

- I/O requirement for our algorithm.

  Finding Split point for nodes

  I/O per tuple per attribute = Sizeof(Attribute Value)+Sizeof(Class Value);

  Splitting Phase

  for *partitioning a non splitting attribute.*

  I/O per tuple per attribute is =Sizeof(Attribute Value)+Sizeof(splitting boolean).

  Writing requires I/O per tuple =Sizeof(Attribute Value);

If there are m attributes I/O per tuple is approximately m times sum of the above three terms for each case.

Our I/O requirements are much less than SPRINT as can be seen from the above expressions. Assuming the size of data objects as present in available machines we read $\frac{1}{2} to \frac{1}{3}$ of the data read by SPRINT.

Attribute lists maintained in SPRINT are about 3 times larger compared to our attribute lists. This increases the storage requirements for SPRINT.

5.2 Absence of Hash Tables and No Pre-sorting

Our algorithm does not require pre-sorting of continuous attribute. This presorting is expensive if data does not fit in the main memory.

SPRINT requires multiple scans of disk resident hash tables, if they do not fit into main memory. We have no such problems in our formulation. This is possible because the relative order of examples in all the "attribute lists" is the same and thus no hash tables are created.

5.3 Accuracy

We get similar accuracy for the datasets that we have tested although it may seem that our approximations for continuous attribute should lower the accuracy. We think that the sorting based techniques tend to perform an "overkill" by trying to look at every distinct value of the continuous attribute. Typically different entropy measures like Gini Index (SLIQ, SPRINT), Information gain(C4.5) are used by the algorithms. Even the exact least entropy points found by these schemes are not the same when they work on a single dataset. We think therefore that our approach does not suffer any loss in accuracy.

5.4 CPU Requirements

The processing for categorical attributes is the same in almost all of the decision based algorithms as a class histogram has to be constructed. In case of continuous attributes we use clustering at every node of the decision tree, our algorithm performs on average about log(c) operations per attribute value where c is the number of centroids in the cluster. This is the time required to assign a attribute to a centroid nearest to it. We are working with 16 clusters for all our experiments.No major change in accuracy was seen as cluster size was varied from 8-64 in our experiments. The CPU processing for *sorting based* schemes is on average equal to the number of classes as entropy value has to be calculated at each point. This requires looking at all the classes in the class histogram.

5.5 New Data Layout to Reduce Storage Requirements

Instead of separating input dataset into attribute lists, another way is to store the whole records sequentially. This will further decrease the I/O cost as the partitioning of the example can be done for all attributes based on the splitting attribute as all the attributes for an example are stored contiguously. But this scheme requires class histograms and clustering structures for all attributes, though only for a single node as attributes are processed one at a time, to be active at the same time during the process of split point finding process. This could become a memory limitation if attributes are large in number. Our implementation uses separate "attribute lists" for each attribute.

6 Parallel Formulation

We now present the parallelization of our algorithm on a multi-processor system. We concentrate on the parallelization of the decision tree building process. Pruning the decision tree built, takes a small amount of time so it is done on a single processor after the decision tree building has been completed. The parallelization presented is valid for a shared nothing parallel environment where each processor has access to a local disk system. The parallelization was specifically done on an IBM SP2 system. We describe two parallel formulations for generating the decision tree. In *Synchronous Tree Construction Approach*, the entire set of processors synchronously expand the decision tree level by level. Each processor works on expanding all the nodes at a level cooperatively with other processors. In *Partitioned Tree Construction Approach*, each new generated node is expanded by a subset of processors that helped the expansion of the parent node. As the tree expansion takes place the tree is partitioned and allocated to a subset of processors which work on expanding them independent of other processors. Every time the tree is partitioned some data relocation has to be done. We have implemented the *Hybrid Formulation* which has elements of both schemes to reduce communication overheads. This formulation leads to a highly efficient and scalable parallel formulation. We explain the common elements of all these parallel formulations, the two parallel approaches and then explain the hybrid formulation.

6.1 Data Distribution and decision tree node expansion

The table containing the entire dataset is horizontally partitioned into P partitions, where P is the number of processors being used. Each of the processors get N/P training data items. At this point, all the processors cooperate to expand the root node of a decision tree. The two basic steps for decision tree node expansion i) Evaluation of split point for a node ii) Partitioning the data at the node to the child nodes. We explain how each one of these steps is done.

6.2 Finding splitting attributes for nodes

For splitting, processors need to decide on an attribute to split a decision tree node and to generate its child nodes. Discrete and continuous attributes are treated differently.

Categorical Attributes

For discrete attributes, there are three steps. In the first step, every processor computes the class histogram for its local data. In the second step, the processors combine their local class distribution information using global reduction [KGGK94] to get a class histogram for all the examples at a particular node. Finally, processors calculates compute entropy gains of the discrete attributes and find the best discrete attribute.

Numerical Attributes

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, This clustering could be done in two ways.

- Each processor clusters its local data, independent of data at other processors and then at the end these clusters are combined.
- all processors cooperate to cluster all attribute values in each class separately using a standard clustering algorithm such as k-means [KR90] 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 node expansion overall using either discrete or continuous attributes. Partitioning of the data is done by applying the splitting test to the local data.

6.3 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 8 shows the overall picture. Major steps for the approach are shown below:

1. Let the current frontier consist of the root of the decision tree.

[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.

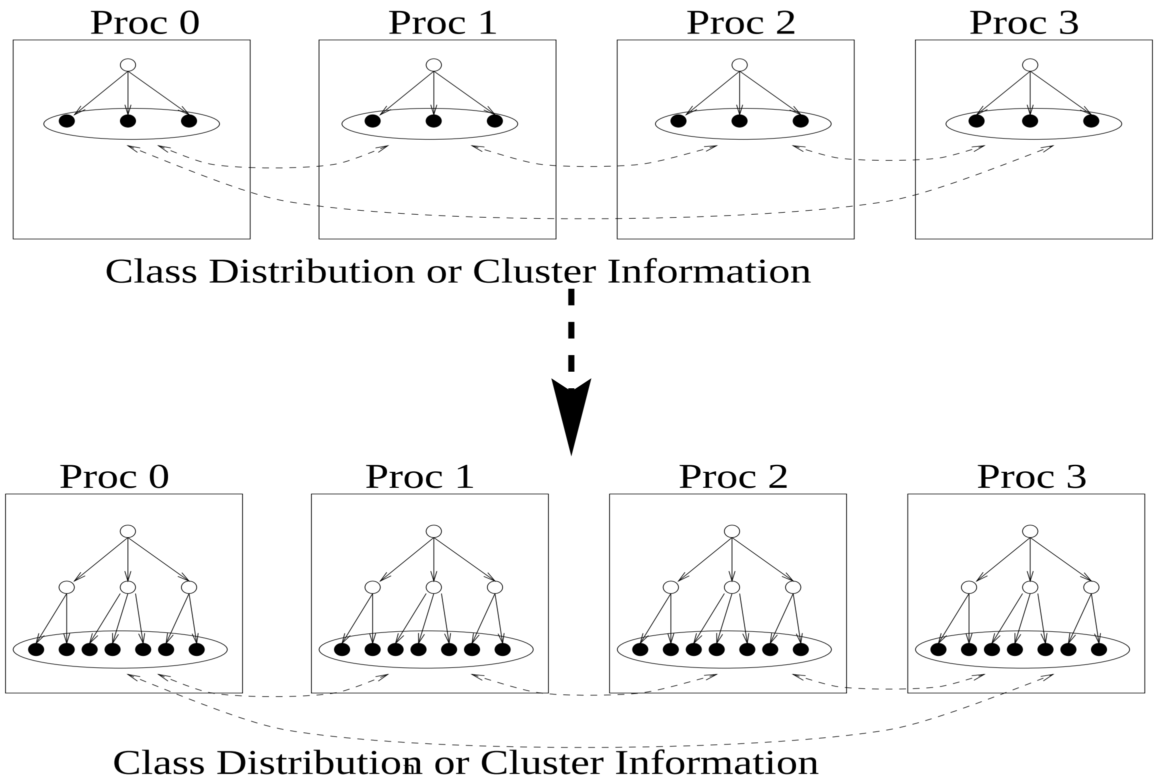

Figure 8: Synchronous Tree Construction Approach with Depth–First Expansion Strategy

2. For each node in the current frontier of the decision tree, 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 conditions (subset condition for discrete attributes or splitting value for continuous attributes), create child nodes and partition the data accordingly for all the nodes in the frontier.

4. Set the current frontier to consist of the nodes generated in the previous step.

5. Repeat the above steps (2–4) until the frontier is empty.

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. After collecting class distribution information for discrete attributes and cluster information for continuous attributes of all the nodes in the current frontier, all the processors have to communicate. The amount of communication is directly proportional to the number of nodes in the frontier, because each node has its own class distribution and cluster information. At the nodes of shallow depth, the communication overhead is relatively small, because the number of nodes in the frontier is small. But as the decision tree grows and deepens, the number of nodes in the frontier increases and the communication overhead becomes large.

6.4 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

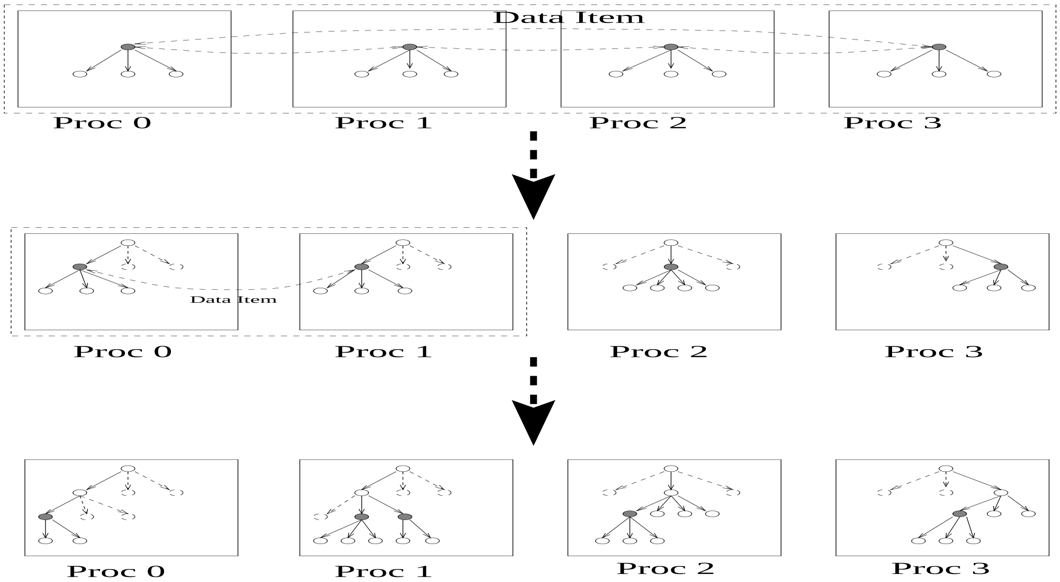

Figure 9: Partitioned Tree Construction Approach

$\mathcal{P}_{(n)}$ 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 9 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 6 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.

6.5 Hybrid Parallel Formulation

Our hybrid parallel formulation has elements of both schemes. The *Synchronous Tree Construction Approach* in Section 6.3 incurs high communication overhead as the frontier gets larger. The *Partitioned Tree Construction Approach* of Section 6.4 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.

- The test cases on which the classification algorithm has to be run are randomly selected 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.

- 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 attributes 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.

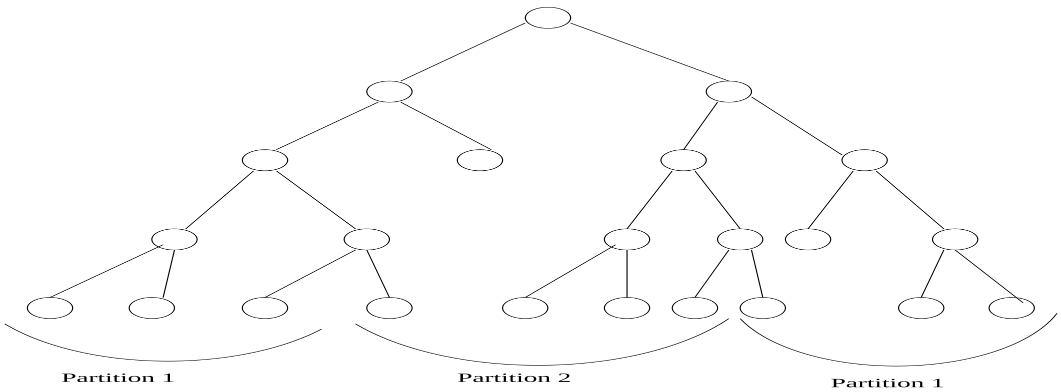

Figure 10: Binary partitioning of the tree to reduce communication costs

- 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.

- 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.

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. 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

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

A detailed analysis of the hybrid algorithm is done in the Appendix . The algorithm is highly scalable because the communication is reduced by partitioning the tree generation among processors. All the other components of the algorithm are totally parallelizable.

7 Implementation and Results

We have implemented the hybrid formulation using the MPI programming library. The parallel formulation gives *almost identical* performance as the serial algorithm in terms of accuracy and classification tree size. Note that only the split finding phase for continuous attributes in the parallel version is different from the serial algorithm. The parallel clustering algorithm produces similar but not exactly same clusters as in the serial case.

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.

7.1 Description of the dataset and Architecture

In the absence of any benchmark having large datasets, we have used the synthetic dataset proposed by in the SLIQ paper [MAR96] for all our experiments. Ten classification functions were also proposed in [MAR96] for these functions. We have used the function 2 dataset for our algorithm. In this dataset each record consists of 9 attributes having 3 categoric and 9 continuous attributes. There are two classes. The same dataset was also used by the SPRINT algorithm for evaluating its performance.

Experiments were done on an IBM SP2. The results are reported for parallel runs on 1, 2, 4, 8, 16, 32, 64 and 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).

7.2 Speedup and runtimes

For measuring the speedups we worked with different sized datasets. We increased the processors from 1 to 128. The results in fig 11 show the runtimes. Efficiency falls slightly as the processors are increased. It falls to 91% with 8 processors for the case of 0.8 million examples. The results confirm the that the parallelization is indeed very effective.

7.3 Scaleup

We kept the dataset size at each processor constant at 50,000 examples and increased the number of processors. Figure 12 shows the speedups.

They are very close to the the ideal case of a horizontal line. The deviation from the ideal case is due to the fact that communication overhead increases as the number of processors increase.

7.4 Comparison with parallel SPRINT

SPRINT paper [SAM96] contains details of SPRINT parallelizations. Although the design of SPRINT has been done to make it scalable, Some components of it are not very scalable. The reasons SPRINT is not scalable are

- During decision tree building process at each level of the tree, a costly all-to-all broadcast [KGGK94] of a data structure (hash table), which is proportional to number of total examples in the input data, is done.
- There is some serial work which is not parallelizable in SPRINT. The hash tables have the same size at each processor independent of the number of processors.

Our algorithm gives superior performance compared to parallel SPRINT. The runtimes of SPRINT are shown in 13. These runtimes are for the same data set that we have used. Our runtimes are 3.5-4.0 times less on similar architecture and the efficiencies are better than SPRINT. Note that *actual runtimes cannot be strictly compared* as these results were done on a different version of the

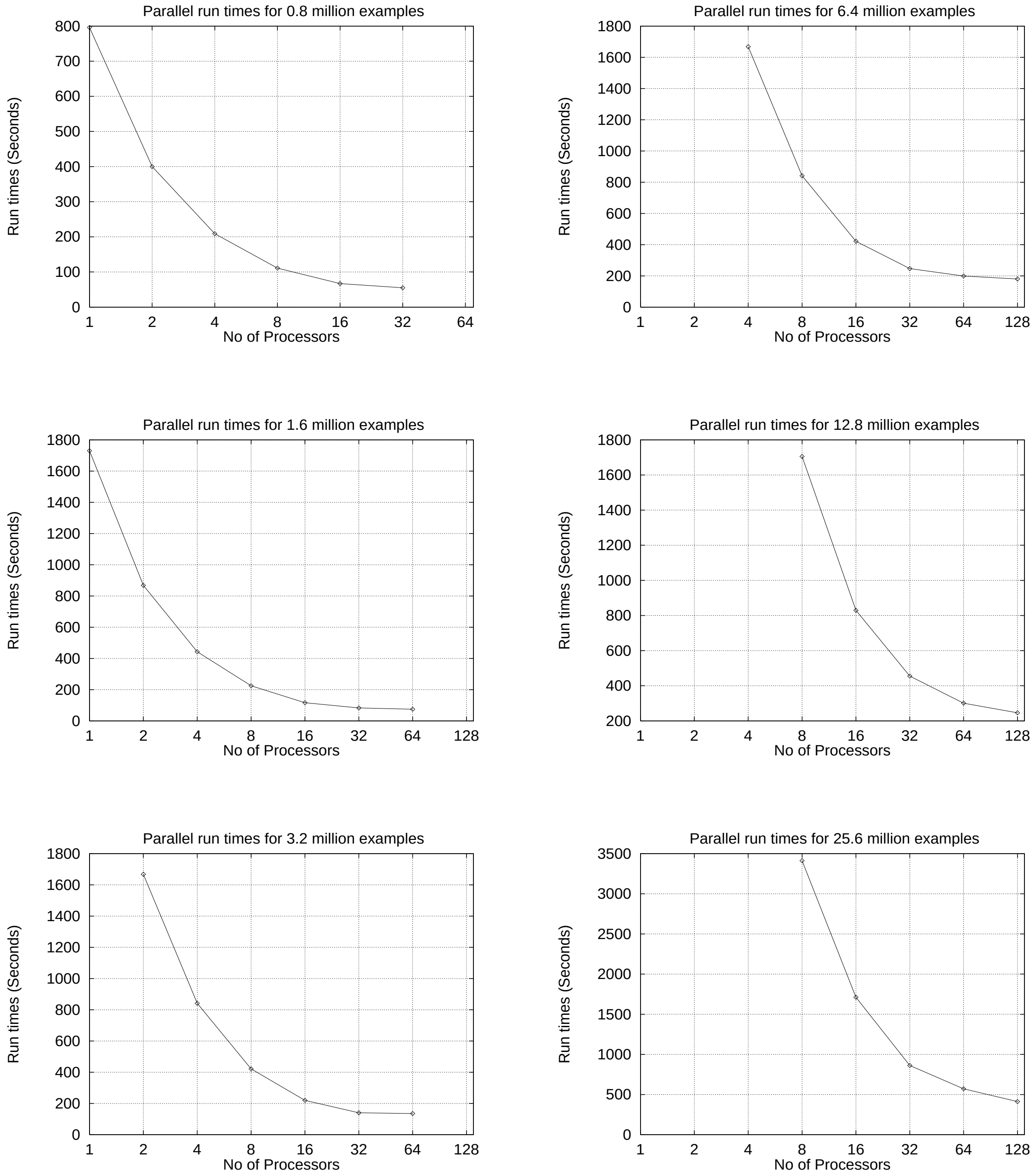

Figure 11: Parallel runtimes of our algorithm

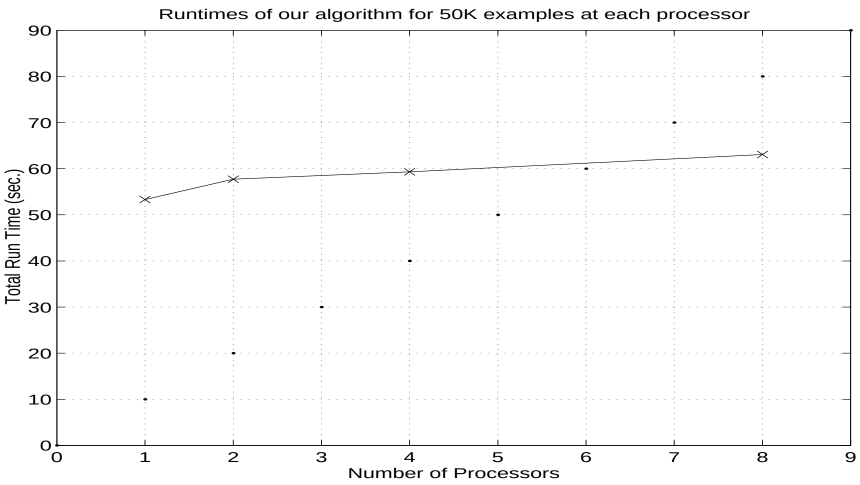

Figure 12: Scaleup of our algorithm

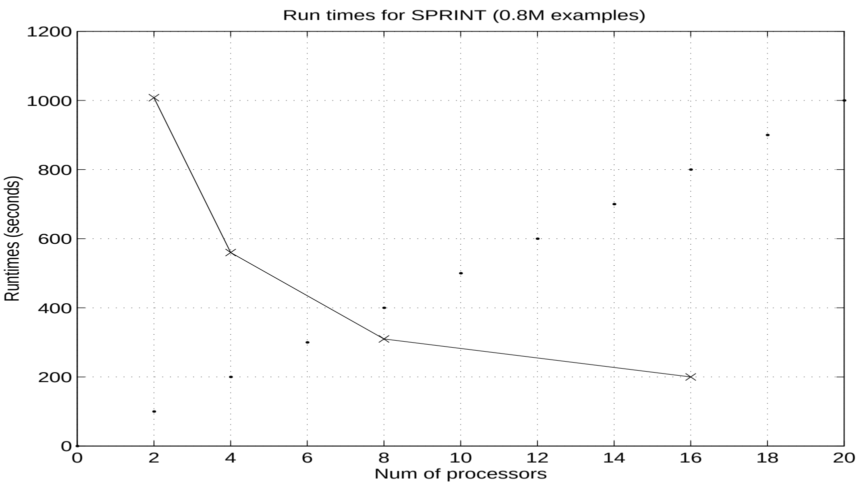

Figure 13: Parallel runtimes of SPRINT[JSM96]

IBM SP2(Model 9076). Also SPRINT results are with all data stored in memory. Although with this I/O cost become smaller part of overall execution time. It also hides the cost of multiple scans of database which would be done in SPRINT if hash tables did not fit in memory.

8 Conclusions

In this paper we presented a new classification algorithm for dealing with large disk resident datasets. The algorithm introduced a novel way of handling continuous attributes . The serial algorithm was shown to achieve essentially identical accuracy as achieved by all the other decision based classifiers. This algorithm was shown to be highly efficient and capable of handling extremely large datasets. We also presented a comparative analysis with SPRINT, which has also been designed for large datasets. Analysis and experimental results showed superior performance superior performance of our algorithm as compared to SPRINT.

We also presented three possible parallel parallelization schemes for our algorithm. The hybrid scheme was implemented on the IBM SP2. The analysis and results for our parallel version show that it extremely scalable and can can handle very large datasets. Speedups got were very near to the ideal case. The algorithm also showed nice scaleup.

Some recent papers [KS96], [FI93] results have shown that considering the continuous attributes as categorical attributes by considering multiple split points and sub setting on these split points improves the accuracy of classification. This capability is difficult to achieve on sorting based algorithms as it requires multiple scans of the disk resident attribute lists. But it can be easily done in our clustering based approach. We plan to achieve this capability in our system in the future.

We intend to run our algorithm on larger datasets on a larger number of processors. We also intend to port the classifier onto other parallel machines like CrayT3E, SGI SMP clusters etc.

References

[AIS93] 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.

[CS93a] 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.

[CS93b] 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.

[FI93] U.M. Fayyad and K.B Irani. Multi–interval discretization of continuous–valued attributes for classification learning. In *Proceedings of the 13th Int. Joint Conference on Artificial Intelligence*, pages 1022–1027, 1993.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimizations and Machine Learning.* morgan-kaufman, 1989.

[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.

[KR90] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis.* John Wiley & Sons, 1990.

[KS96] R. Kohavi and M. Sahami. Error–based and entropy–based discretization of continuous features. In *Proc. of the Second Int'l Conference on Knowledge Discovery and Data Mining*, pages 114–119, Portland, OR, 1996.

[Lip87] R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(22), April 1987.

[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.

[MST94] D. Michie, D.J. Spiegelhalter, and C.C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

symbol	definition
N	Total number of training sample
P	Number of processors
A_d	Number of categorical attributes in the database
A_c	Number of continuous attributes in the database
K	Number of clusters for each class
C	Number of classes in the database
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.

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

[SAM96] J. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *22nd VLDB conference*, 1996.

[WC88] J. Wirth and J. Catlett. Experiments on the costs and benefits of windowing in ID3. In *5th Int'l Conference on Machine learning*, 1988.

Appendix: Analysis of the Hybrid Formulation

In this section, we provide the analysis of the hybrid algorithm proposed in Section 6.5. We provide a separate analysis for two different data attribute types, discrete and continuous. 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.

Case with Discrete Attributes Only

Here we give a detailed analysis of splitting for the case for discrete attributes. 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 :

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

The number of leaves at level L is 2^L. Thus the size of the total data to be sent from each processor is :

$$\text{Processor Communication Buffer size} = C * A_d * M * 2^L$$

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:

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

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 :

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

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.

$$\text{Moving Cost} = t_s + t_w * \frac{N * K_1}{2 * P} \tag{3}$$

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.

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:

[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.

- 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.

$$\begin{aligned} \text{Communication Cost for Load Balancing} &= \text{all-to-all broadcast} + \text{movement of records} \\ &= t_s \log P + t_w * (P - 1) + k_1 * (t_s + t_w * \frac{N}{k_2 * P}) \end{aligned} \tag{4}$$

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.

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. This clustering is done locally at each processor and finally these clusters accumulated at a single processor. So in our parallel formulation, where breadth first expansion is assumed, following steps are taken for each level:

```
for each node in the frontier
   read in data block of size B;
   for each attribute
       partition the data according to class and sort this data locally
```

```
5        select K attribute values per class as initial centroids locally
6        construct cluster tree locally 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        update/reconstruct the cluster trees locally;
10   Accumulate the local clusters of each processor at a single processor
11   calculate the entropy of an attribute based on accumulated clusters ;
12   find the best attribute for a decision tree node;
```

The size of the communication buffer for step 10 is $2^L * A_c * C * K$, since there are 2^L number of nodes in level L of the frontier and there are $A_c * C$ number of cluster trees with K leaves on each node. After local clustering the clusters are accumulated at a single processor in an operation which is the reverse of one-to-all personalized [KGGK94] communication. The overall communication cost of step 10 is

$$t_s + t_w * 2^L * A_c * C * K * P \tag{5}$$

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

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

The overall computation cost of step 11 and 12 is $2^L * A_c * C * K * P$. This gives the overall computation cost as follow:

$$\text{Computation Cost} = (A_c * \frac{N}{P} * \log K + 2^L * A_c * C * K * P)t_c \tag{6}$$

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.

9 Isoefficiency analysis of our algorithm

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.

From Equation 3 in the appendix moving cost at any level takes time: $\theta(\frac{N}{P})$.This moving cost is the same irrespective of the size of the subset of processor in which the data is moved. Note that when the subset under consideration is of size $\frac{P}{2^i}$ the size of dataset at each processor has the same size $\frac{N}{P}$ as was present in the beginning and therefore the moving cost is the same.

From Equation 4 a single load balancing step in any subset of processors of size P_i takes time: $\theta(P_i) + \theta(\frac{N}{P})$

Total moving, load balancing costs for the entire tree building process equals Total number of movements * (Moving + Load Balancing cost)

$$\text{Total moving, load balancing cost} \leq L_1 * (\theta(\frac{N}{P})) + \sum_{i=0}^{NoofMoves} \frac{P}{2^i}$$

$$\text{Total moving, load balancing cost} = \theta(\frac{N}{P}) + \theta(P)$$

For the discrete attributes only the class histogram has to be communicated. This requires a global reduction among all the processors. At any level L a subset P_i of processors works on expanding 2^L of the nodes. Communication for discrete attributes at each level equals $t_s + 2^L * t_w * A_d * C * S * \log P_i$ from Equation 2.

$$\text{Total Communication cost for discrete attributes} = \theta(\log P)$$

For the continuous attributes number of clusters proportional to P_i have to be processed. Each processor comes up with a fixed number of clusters from its data. These clusters are accumulated at a single node in a operation which is the reverse of one to all personalized communication [KGGK94]. So at each level communication time for continuous attribute equals $t_s + 2^L * t_w * A_c * C * K * P_i$ from Equation 5.

$$\text{Total Communication cost for continuous attributes} = \theta(P)$$

Total communication cost for all the attributes considering both discrete and continuous attributes is :

$$\theta(P) + \theta(\log P)$$

The total overhead incurred is the sum of costs of rearranging data, cost of communication for categoric and continuous attributes and the cost of processing larger number of clusters for numeric data. So the total communication cost is:

$$\theta(P) + \theta(\log P) + \theta(\frac{N}{P}) \tag{7}$$

Total computation done is the scanning of the dataset present at each processor of size $\frac{N}{P}$ once for every level . Cutpoint finding time for continuous variables takes time proportional to number of processors and processing of discrete variables takes a constant amount of time. So total computation time is given by the sum of Equation 1 and Equation 6.
Total computation time = No of levels $*$ Computation at each level

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

Total time for the parallel case = Communication time + Computation time.

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

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 overhead function using Equation 9 to serial computation time.

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

$$N = \theta(P^2)$$

therefore the isoefficiency function is $N = \theta(P^2)$.

Isoefficiency is $\theta(P^2)$ because the communication cost for continuous attributes are proportional to P. This is because the number of clusters to be accumulated at a single processor are proportionalto P.

The hybrid scheme does not performs superior in asymptotic terms to the synchronous tree construction approach but it significantly reduces the communication cost. As soon as the tree is partitioned the the communication is reduced by a factor of four since half the number of processors are working now on half the previous number of decision tree nodes. Note that at a certain depth there is no interprocessor communication in the hybrid scheme as each processor works individually on its set of decision tree nodes.