Communication and Memory Efficient Parallel Decision Tree Construction *

Ruoming Jin
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210

jinr@cis.ohio-state.edu

Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210

agrawal@cis.ohio-state.edu

ABSTRACT

Decision tree construction is an important data mining problem. In this paper, we revisit this problem, with a new goal, i.e. *Can we develop an efficient parallel algorithm for decision tree construction that can be parallelized in the same way as algorithms for other major mining tasks ?*.

We report a new approach to decision tree construction, which we refer to as SPIES (Statistical Pruning of Intervals for Enhanced Scalability). This approach combines RainForest based AVC groups with sampling to achieve memory efficient processing of numerical attributes. Overall, this algorithm has the following properties: 1) no preprocessing or sorting of input data is required, 2) the size of the data-structure required in the main memory is very small, 3) the only disk-traffic required is one pass for splitting nodes for each level of the tree, and no writing-back of data, 4) very low communication volume when this algorithm is parallelized, and 5) the same level of accuracy as an algorithm that does not use sampling or pruning.

We show that this algorithm can be efficiently parallelized using the same high-level interface and runtime support that was previously used to parallelize association mining and clustering algorithms. This, we believe, is an important step towards offering high-level interfaces for parallel data mining. Moreover, we have efficiently parallelized this algorithm on a cluster of SMPs, i.e. combining shared memory and distributed memory parallelism, and over disk-resident datasets.

Key-words: Decision tree construction, parallelization, cluster of SMPs, sampling, algorithms for streaming data

1. INTRODUCTION

Decision tree construction is an important data mining problem. With the availability of large datasets, decision tree construction over disk-resident datasets and on parallel machines has received considerable attention [6, 7, 8, 14, 16, 18, 20, 21].

In this paper, we revisit this problem, with a new goal, i.e. *Can we develop an efficient parallel algorithm for decision tree construction that can be parallelized in the same way as algorithms for other major mining tasks ?*.

To motivate this problem, we consider SLIQ and SPRINT, which are the two major parallelizable algorithms suitable for disk-resident datasets. SLIQ [16] requires sorting of ordered at-

tributes and separation of the input dataset into attribute lists. This imposes a significant computational and memory overhead. Sorting of disk-resident datasets is an expensive operation and moreover, creation of attribute lists can increase the total memory requirements up to three times. In addition, SLIQ requires a data-structure called *class list*, whose size is proportional to the number of records in the dataset. Class list is accessed frequently and randomly, therefore, it must be kept in the main memory. SPRINT [18] is quite similar. It also requires separation of input dataset into attribute lists, and sorting of ordered attributes. SPRINT does not require class lists, but instead requires a hash table which is proportional to the number of records associated with a node in the decision tree. Moreover, SPRINT requires partitioning of attribute lists whenever a node in the decision tree is split. Thus, it can have a significant overhead of rewriting disk-resident datasets.

In comparison, algorithms for other mining tasks, such as association mining, clustering, and k-nearest neighbor search, can be parallelized without any expensive preprocessing of datasets or requiring datasets to be written back [2, 22, 4]. As we have demonstrated in our recent work, these algorithms can be parallelized on both shared memory and distributed memory configurations and over disk-resident datasets, using a common interface and runtime support [11, 10, 12].

This paper presents a new algorithm for decision tree construction. Besides being more memory efficient than the existing algorithms, this algorithm can be efficiently parallelized in the same fashion as the common algorithms for other major mining tasks. Our approach derives from the RainForest approach for scaling decision tree construction [7], as well as the recent work on one pass decision tree algorithms for streaming data [5].

RainForest is a general framework for scaling decision tree construction [7]. Unlike SPRINT and SLIQ, it does not require sorting or creation of attribute lists. The main memory data-structure required (AVC group) is proportional to the number of distinct values of the attributes. Within this framework, a number of algorithms have been proposed. Of particular interest to us is RF-read, in which the dataset is never partitioned. All nodes at any given level of the tree are processed in a single pass if the AVC group for all the nodes fit in the main memory. If not, multiple passes over the input dataset are made to split nodes at the same level of the tree.

The biggest challenge in using RainForest based approach comes because of the memory requirements for AVC groups for numerical attributes. The total number of distinct values for a numerical attribute is typically very large. This has two implications. First, in sequential processing, the total memory requirements for AVC groups of all nodes at a level of the tree can be very high. As a

result, the input data may have to be read multiple times to process one level of the tree, or may have to be partitioned and written back. Second, the total communication volume required can severely limit parallel performance.

We propose an approach which overcomes this limitation by using sampling based upon Hoeffding's bound [9], which is also used in recent work on one pass decision tree construction [5]. We call this approach SPIES (Statistical Pruning of Intervals for Enhanced Scalability). In this approach, the range of any numerical attribute which has a large number of distinct values is partitioned into *intervals*. The construction of AVC groups is done using two, or in some cases three, steps. The first step is the sampling step, where we use a sample to record class histograms for the intervals. Using this information, we prune the intervals that are not likely to contain the split point. Then, a complete pass over the data is taken. During this step, we update the class histograms for the intervals, and also compute class histograms for the candidate split points that had not been pruned after the sampling step. However, since this pruning was based upon a sample, it could have incorrectly pruned some intervals. If this is the case, we need to perform an additional pass over the data and record class frequencies for previously incorrectly pruned split points. Our sample size is chosen using Hoeffding's inequality, which gives a bound on the probability of requiring the additional pass. With the use of pruning, the memory requirements for AVC groups are reduced drastically.

Overall, the SPIES algorithm has the following properties: 1) no preprocessing or sorting of input data is required, 2) the size of the data-structure required in the main memory is very small, 3) the only disk-traffic required is one pass for splitting nodes for each level of the tree, and no writing-back of data, 4) very low communication volume when this algorithm is parallelized, and 5) the same level of accuracy as an algorithm that does not use sampling or pruning.

We have parallelized this approach using the FREERIDE (Framework for Rapid Implementation of Datamining Engines) middleware system we had reported in our earlier work [11, 12]. This system offers an interface and runtime support to parallelize a variety of mining algorithms, and has previously been used for association mining, clustering, and k-nearest neighbor search algorithms. The target environment of this system is a cluster of SMPs, which have shared memory parallelism within each node and distributed memory parallelism across the nodes.

Our detailed experimental results demonstrate the following. First, the memory requirements for holding AVC groups are reduced by between 95% and 85%, depending upon the number of intervals used. In addition, because of a smaller number of candidate splitting conditions, the computation time is also reduced by up to 20% as compared to the RainForest based algorithms. It is also notable that in all our experiments, the extra pass was never required. In performing distributed memory parallelization, near linear speedups are achieved, particularly with using 100 or 500 intervals. When the implementation is not I/O or memory bound, good shared memory speedups are also obtained.

To summarize, this paper makes the following contributions.

- We have presented a new (sequential) algorithm for decision tree construction, which combines RainForest like AVC group based approach with sampling. The properties of this algorithm are listed above.

- We have shown how this algorithm can be efficiently parallelized in the same fashion as the algorithms for other common mining tasks. We believe that this is an important step towards offering high-level interfaces for developing parallel

data mining implementations.

- We have parallelized a decision tree construction algorithm on a cluster of SMPs, i.e., combining shared memory and distributed memory parallelization, and on disk-resident datasets. We are not aware of any previous reporting this.

The rest of this paper is organized as follows. In Section 2, we further define the decision tree construction problem and then given an overview of the RainForest approach. Our new algorithm is presented in Section 3. Our middleware and parallelization of our new algorithm are presented in Section 4. Experimental evaluation of our work is presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2. DECISION TREE CONSTRUCTION

In this section, we focus on the problem of decision tree construction, existing scalable algorithms, and the RainForest based approach.

2.1 Problem Definition

The input to a decision tree construction algorithm is a database of *training records*. Each record has several *attributes*. An attribute whose underlying domain is totally ordered is called a *numerical* attribute. Other attributes are called *categorical* attributes. One particular attribute is called the *class label*, and typically can hold only two values, true and false. All other attributes are referred to as the *predictor* attributes.

A decision tree construction algorithm processes the set of training records, and builds a model which is used for predicting the class label of new records. Decision tree algorithms proceed by recursively partitioning the dataset, till the records associated with a leaf node either all have the same class label, or their cardinality is below a threshold. A node is partitioned using a *splitting* condition. For a categorical attribute, the splitting condition is typically a subset predicate, e.g., if $x_i \in \{red, blue\}$. For a numerical attribute, the splitting condition is a typically a range, e.g., if $x_i \leq 50$. In partitioning any node, a number of different candidate splitting conditions are evaluated. The best splitting condition is typically chosen to maximize a *gain function*, which are based upon impurity measurements such as gini or entropy.

There are two major challenges in scaling a decision tree construction algorithm. The first comes because of a very large number of candidate splitting conditions associated with every numerical attribute. The second arises because of the recursive nature of the algorithm. As we are partitioning a non-root node, we want to be able to access and process the records associated with this node. However, this can require us to partition and write-back the data, or maintain special data-structures.

One of the first decision tree construction methods for disk-resident datasets was SLIQ [16]. To find splitting points for a numerical attribute, SLIQ requires separation of the input dataset into attribute lists and sorting of attribute lists associated with a numerical attribute. An attribute list in SLIQ has a record-id and attribute value for each training record. To be able to determine the records associated with a non-root node, a data-structure called *class list* is also maintained. For each training record, the class list stores the class label and a pointer to the current node in the tree. The need for maintaining the class list limits the scalability of this algorithm. Because the class list is accessed randomly and frequently, it must be maintained in main memory. Moreover, in parallelizing the algorithm, it needs to be either replicated, or a high communication overhead is incurred.

A somewhat related approach is SPRINT [18]. SPRINT also requires separation of the dataset into class labels and sorting of attributes lists associated with numerical attributes. The attribute lists in SPRINT store the class label for the record, besides the record-id and attribute value. SPRINT does not require class list data-structure. However, the attribute lists must be partitioned and written-back when a node is partitioned. Thus, it can have a significant overhead of rewriting a disk-resident dataset.

In the related work section, we will further compare our work with newer approaches like the CLOUDS method [3], the algorithm by Srivastava *et al.* [20], and the BOAT algorithm by Gehrke [6].

2.2 RainForest Based Approach

RainForest is a general framework for scaling decision tree construction [7]. RainForest scales decision tree construction to larger datasets, while also effectively exploiting the available main memory. This is done by isolating an AVC (Attribute-Value, Classlabel) set for a given attribute and a node being processed. AVC set for an attribute simply records the the count of occurrence of each class label for each distinct value the attribute can take. The size of the AVC set for a given node and attribute is proportional to the product of the number of distinct values of the attribute and the number of distinct class labels. The AVC set can be constructed by taking one pass through the training records associated with the node.

Given a node of the decision tree, AVC group is the combination of AVC set for all attributes. The key observation is that though AVC group does not contain sufficient information to reconstruct the training dataset, it contains all information that is required for selecting the criteria for splitting the node. One can expect the AVC group for a node to easily fit in main memory, though the RainForest framework includes algorithms that do not require this. With the above observation, processing for selecting the splitting criteria for the root node can be easily performed even if the dataset is disk-resident. The algorithm initiates by reading the training dataset once, and constructing the AVC group of the root node. Then, the criteria for splitting the root node is selected.

A number of algorithms have been proposed within the RainForest framework to split decision tree nodes at lower levels. In the algorithm RF-read, the dataset is never partitioned. The algorithm progresses level by level. In the first step, AVC group for the root node is built and a splitting criteria is selected. At any of the lower levels, all nodes at that level are processed in a single pass if the AVC group for all the nodes fit in main memory. If not, multiple passes over the input dataset are made to split nodes at the same level of the tree. Because the training dataset is not partitioned, this can mean reading each record multiple times for one level of the tree.

Another algorithm, RF-write, partitions and rewrites the dataset after each pass. The algorithm RF-hybrid combines the previous two algorithms. Overall, RF-read and RF-hybrid algorithms are able to exploit the available main memory to speedup computations, but without requiring the dataset to be main memory resident.

3. NEW ALGORITHM FOR HANDLING NUMERICAL ATTRIBUTES

In this section, we present our SPIES (Statistical Pruning of Intervals for Enhanced Scalability) approach for making decision tree construction more memory (and communication) efficient. Here, we focus on the sequential algorithm. Parallelization of the algorithm is presented in the next section.

3.1 Overview of the Algorithm

SPIES-Classifier(Level $\mathcal{L}$, Dataset $\mathcal{D}$)
  foreach ($Node$ $\mathcal{N} \in Level$ $\mathcal{L}$)
    { *Sampling Step* }

    $Sample$ $\mathcal{S} \leftarrow Sampling(\mathcal{D})$;
    $Build\_Small\_AVCGroup(\mathcal{S})$;
    $Build\_Concise\_AVCGroup(\mathcal{S})$;
    $g' \leftarrow Find\_Best\_Gain(AVCGroup)$;
    $Partial\_AVCGroup$
      $\leftarrow Pruning(G', Concise\_AVCGroup)$;

    { *Completion Step* }

    $Build\_Small\_AVCGroup(\mathcal{D})$;
    $Build\_Concise\_AVCGroup(\mathcal{D})$;
    $Build\_Partial\_AVCGroup(\mathcal{D})$;
    $g \leftarrow Find\_Best\_Gain(AVCGroup)$;

    if $False\_Pruning(G, Concise\_AVCGroup)$
      { *Additional Step* }
      $Partial\_AVCGroup$
        $\leftarrow Pruning(G, Concise\_AVCGroup)$;
      $Build\_Partial\_AVCGroup(\mathcal{D})$;
      $G \leftarrow Find\_Best\_Gain(AVCGroup)$;
    if not $satisfy\_stop\_condition(\mathcal{N})$
      $Split\_Node(\mathcal{N})$;

Figure 1: SPIES Algorithm (Sequential) for Decision Tree Construction

---

The algorithm is presented in Figure 1. Our method is based on AVC groups, like the RainForest approach. The key difference is in how the numerical attributes are handled. In our approach, the AVC group for a node in the tree comprises three sub-groups:

Small AVC group: This is primarily comprised of AVC sets for all categorical attributes. The main idea is that the number of distinct elements for a categorical attribute is not very large, and therefore, the size of the AVC set for each attribute is quite small. In addition, we also add the AVC sets for numerical attributes that only have a small number of distinct elements. These are built and treated in the same fashion as in the RainForest approach.

Concise AVC group: The range of numerical attributes which have a large number of distinct elements in the dataset is divided into *intervals*. The number of intervals and how the intervals are constructed are important parameters to the algorithm. In our current work, we have used *equal-width* intervals, i.e. the range of a numerical attribute is divided into intervals of equal width. The impact of number of intervals used is evaluated in our experiments. The concise AVC group records the class histogram (i.e. the frequency of occurrence of each class) for each interval.

Partial AVC group: Based upon the estimate of the concise AVC group, our algorithm computes a subset of the values in the range of the numerical attributes that are likely to be the split point. The partial AVC group stores the class histogram for the points in the range of a numerical attribute that has been determined to be a candidate for being the split condition.

Our algorithm uses two steps to efficiently construct the above AVC groups. The first step is the *sampling step*. Here, we use a sample from the dataset to *estimate* small AVC groups and concise numerical attributes. Based on these, we can get an estimation of the best (highest) gain, denoted as g'. Then, by using g', we will prune the intervals that do not appear likely to include the split

point. The second step is the *completion step*. Here, we use the entire dataset to construct all of the three sub AVC groups. The partial AVC groups will record the class histogram for all of the points in the unpruned intervals.

After that, we get the best gain g from these AVC groups. Because our pruning is based upon the estimates of small and concise AVC groups, we could have a situation we refer to as *false pruning*. False pruning occurs when an interval in the range of a numerical attribute does not appear likely to include the split point after the sampling step, but could include the split point after more information is made available during the completion phase.

False pruning is detected using the updated values of small and concise AVC groups that are available after the completion step. We see if there are any differences between the pruning that may be obtained now and the pruning obtained using the estimates of small and concise AVC groups after the sampling step. If false pruning does not occur, a node can be partitioned using the available values of small, partial, and concise AVC groups. However, if false pruning has occurred, we need to make another pass on the data to construct partial AVC groups for points in false pruned intervals.

We can see that effectiveness and accuracy of pruning is the key to the performance of our algorithm. On one side, we will like the pruning to eliminate the intervals which don't include the split point, reduce the size of the AVC groups, and therefore, reduce both memory and computational costs. At the same time, we will like to minimize the possibility of false pruning, thus to reduce the necessity of the additional scan.

3.2 Technical Details

In this subsection, we discuss the details of how pruning, sampling, and testing for false pruning is done. Initially, we focus on pruning.

For our discussion here, we for now assume that we have processed the entire dataset, and are trying to prune the space of potential split points from the range of a numerical attribute. Decision tree construction algorithms typically use a function for measuring the *gain* achieved from a predicate used to split a node. Our approach is independent of the specific function used.

Assume that g is the best (highest) gain possible from any splitting condition associated with a node of the decision tree. Suppose $[x_i, x_{i+1})$ is the ith interval on the numerical attribute X. Let the class distribution of the interval i be

$$\vec{H}_i = (h_i^1, h_i^2, \ldots h_i^c)$$

where $h_i^j, 1 \leq j \leq c$ is the number of training records with the class label j that fall into the interval i, and c is the total number of class labels in the training data.

We want to determine if any point within this interval can provide a higher gain than g. For this, we need to determine the highest possible value of gain from any point within this interval. For the boundary point x_i, we define the cumulative distribution function

$$\vec{N}_{x_i} = (n_{x_i}^1, \ldots, n_{x_i}^c)$$

where,

$$n_{x_i}^k = \sum_{j=1}^{i-1} h_j^k, 1 \leq k \leq c$$

Here, $n_{x_i}^k$ is the number of training records with the class label k such that their value of the numerical attribute under consideration is less than x_i. Similarly, for the boundary point x_{i+1}, we have

$$\vec{N}_{x_{i+1}} = (n_{x_i}^1 + h_i^1, \ldots, n_{x_i}^c + h_i^c)$$

Now, consider any point y between x_i and x_{i+1}. Let the cumulative distribution function be

$$\vec{N}_y = (n_y^1, \ldots, n_y^c)$$

Clearly, the following property holds.

$$\forall j, \ 1 \leq j \leq c, \ n_{x_i} \leq n_y \leq n_{x_{i+1}} \ \ (a)$$

If we do not have any further information about the class distribution of the training records in the interval, all points y satisfying the above constraint need to be considered in determining the highest gain possible from within the interval $[x_i, x_{i+1})$. Formally, we define the set P of possible internal points as all values within the interval $[x_i, x_{i+1})$ that satisfy the constraint a.

The number of points in the set P can be very large, making it computationally demanding to determine the highest possible gain from within the interval. However, we are helped by a well-known mathematical result. To state this result, we define a set S, comprising of *corner points* within the interval. Formally,

$$S = \{s | s \in [x_i, x_{i+1}] \wedge \vec{N}_s \ satisfies \ that$$

$$\forall j, 1 \leq j \leq c, \ (n_s^j = n_{x_i}^j) \vee (n_s^j = n_{x_{i+1}}^j)\}$$

It is easy to see that the set S has 2^c points. Now, the following result allows us to compute the highest possible gain from the interval very efficiently.

LEMMA 1. *Let f be a concave gain function. Let P be the set of possible internal points of interval i, and S be the set of corner points. Then*

$$\max_{p \in P} (f(\vec{N}_p)) \ \leq \ \max_{s \in S} (f(\vec{N}_s))$$

The above lemma comes from a general mathematical theorem in Magasarian [15] and was also previously used by Gehrke *et al.* in the BOAT approach [6]. Note that the gain functions used for decision tree construction are concave functions.

This lemma tells us that we only need to check the 2^c corner points to determine the upper bound on the best gain possible from within a given interval i. By recording the frequency of intervals, computing gains at the interval boundaries, and applying this lemma, we can compute the upper bound u_i of the gain possible from the interval. If we already know the best gain g, we can compare u_i and g. The interval can contain the split point only if $u_i \geq g$, and can be pruned if this is not the case.

As described above, this method is useful only if we have already scanned the entire dataset, know the value of the best gain g as well as the gains at the interval boundaries. However, we perform this pruning after the sampling step, i.e., by examining only a fraction of the training records. To effectively perform pruning using only a sample, we use an important result from statistics, by Hoeffding [9].

As we had described earlier in this section, we use a sample and compute small and concise AVC groups. Thus, for numerical attributes with a large number of distinct values, we are recording the class frequencies for only the intervals, and only using the sample. As defined above, g is the best gain possible using the entire dataset and all possible splitting conditions. Now, let $\bar{g}$ be the best gain computed using the sampling step. Note that we are not computing either g or $\bar{g}$ here. Let $g\prime$ be the best gain noted after the sampling step and using class histograms for the intervals. This is the value actually computed by our algorithm.

We have,

$$g' \leq \bar{g}$$

This follows simply from the fact that g' is computed from among a subset of the split points that $\bar{g}$ would be computed from.

Using the values of gain at interval boundaries, and using the Lemma 1, we can estimate the upper bound on the gain possible from a split point within a given interval i. We denote this value as $\bar{u}_i$, and is an estimate (using the sample) of the value u_i that could be computed using the entire dataset.

We focus on the function

$$\triangle = g - u_i$$

If we have computed g and u_i using the entire dataset, we can prune an interval i if $\triangle > 0$. However, using sampling, we can only have estimate of this function. Suppose, we consider

$$\bar{\triangle} = \bar{g} - \bar{u}_i$$

An application of statistics result from Hoeffding [9] gives that if $\bar{\triangle} \geq \epsilon$, then with probability $1 - \delta$, we have $\triangle > 0$, where,

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$$

Here, R is the length of the range of the function $\triangle$ and n is the number of samples used.

Thus, pruning using the condition $\bar{\triangle} > \epsilon$ gives us a probabilistic bound on the accuracy of pruning, i.e., we could still prune incorrectly, but the probability is low and bounded as defined above.

Further, since we do not even have the value of $\bar{g}$, we use g'. Since $g' \leq \bar{g}$, we have

$$(g' - \bar{u}_i) > \epsilon \implies (\bar{g} - \bar{u}_i) > \epsilon$$

Thus, after the sampling step, we perform pruning using the condition $g' - \bar{u}_i > \epsilon$. We call this method *Hoeffding bound based pruning*.

We call an interval to be *falsely pruned* if it is pruned using the statistical estimates described above, but after the completion step, it turns out that $u_i \geq g$.

LEMMA 2. *The probability of falsely pruning an interval using the Hoeffding bound based pruning is bounded by δ, where δ is as defined above.*

Proof: This follows from our discussion above. □

Now, we briefly discuss how we test for false pruning. We compute the best gain from the concise and partial AVC group. This value is denoted as g. Now, the condition we need to check for is if any of the pruned intervals can have a gain greater than g. For this, we compute u_i using the class histograms for intervals and the Lemma 1. We check if $u_i \geq g$. If so, we conclude that false pruning has occurred.

3.3 Algorithm Correctness

In this part, we focus on the algorithm correctness, i.e. argue how the new algorithm always generates the same decision tree as an algorithm using the full AVC groups. To prove that, we first show that the new algorithm will get the same best gain for each node as obtained using the full AVC groups.

In the completion step, we compute the best gain g from AVC groups which is build from the entire dataset. However, this is computed using only the interval boundaries and other split points within unpruned intervals. If our test for false pruning returns negative, we know that no point inside a pruned interval could achieve a higher gain. Therefore, we know that g is the best gain possible from using the full AVC group.

If false pruning has occurred and we find intervals that could have a higher gain than g, we perform an additional step. Let $g1$ denote the new value of the best gain, after new points are added in partial AVC group. If this $g1$ is the same as g, then again we have the point with the highest gain. If not, it is a value greater than the one previously computed, and therefore, any interval which is still pruned could not have a higher gain.

LEMMA 3. *For splitting any decision tree node N, the new algorithm will achieve the same best again as obtained using full AVC groups.*

PROOF. Follows from our discussion above. □

Assume that there is only one point that could achieve the best gain for a decision tree node. In such a case, our algorithm will compute the same splitting condition for any node that is split, as obtained by the RainForest method using full AVC groups. If more than one split conditions give the same best gain, then again we can use the same rule that is used in the RainForest Framework to choose the split condition.

THEOREM 1. *The new algorithm generates the same decision tree as any algorithm that is part of the RainForest Framework and uses full AVC groups.*

4. FREERIDE MIDDLEWARE AND PARALLELIZATION OF SPIES

In this section, we describe the parallelization of the SPIES algorithm using the FREERIDE middleware we have reported in our earlier work and used for parallelizing association mining and clustering algorithms [11, 10, 12].

Initializing, we briefly describe the functionality and interface of our middleware. Then, we describe the parallelization of the SPIES algorithm.

4.1 FREERIDE Functionality and Interface

The FREERIDE (Framework for Rapid Implementation of Datamining Engines) is based on the observation that a number of popular data mining algorithms share a relatively similar structure. Their common processing structure is essentially that of *generalized reductions*. During each *phase* of the algorithm, the computation involves reading the data instances in an arbitrary order, processing each data instance, and updating elements of a *reduction object* using associative and commutative operators.

In a distributed memory setting, such algorithms can be parallelized by dividing the data items among the processors and replicating the reduction object. Each node can process the data items it owns to perform a local reduction. After local reduction on all processors, a global reduction can be performed. In a shared memory setting, parallelization can be done by assigning different data items to different threads. The main challenge in maintaining the correctness is avoiding race conditions when different threads may be trying to update the same element of the reduction object. We have developed a number of techniques for avoiding such race conditions, particularly focusing on the memory hierarchy impact of the use of locking. However, if the size of the reduction object is relatively small, race conditions can be avoided by simply replicating the reduction object.

Our middleware incorporates techniques for both distributed memory and shared memory parallelization and offers a high-level programming interface. For distributed memory parallelization, the

interface requires the programmers to specify pairs of local and global reduction functions, and an iterator that invokes these and checks for termination conditions. For shared memory parallelization, the programmers are required to identify the reduction object, and also the updates to those, and also specify a way in which different copies of the reduction object could be merged together.

A particular feature of the system is the support for efficiently processing disk-resident datasets. This is done by aggressively using asynchronous operations for reading *chunks* or disk-blocks from the dataset. However, no support is available for writing back data.

4.2   Parallelization of SPIES Approach

Parallel SPIES(Level $\mathcal{L}$, dataset $\mathcal{D}$)

{ *Sampling Step*}

foreach $(chunk\ C \in Sample\ \mathcal{S})$
  $Reduce2AVCGroups(C)$;
$Perform\ Global\ Reduction\ to$
 $Update\ AVCGroups$;

foreach $(node\ \mathcal{N}\ at\ level\ \mathcal{L})$
  $g' \leftarrow best\_G(\mathcal{N})$;
  $Prune(\mathcal{N}.avc\_group, g')$;

{ *Completion Step*}

foreach $(chunk\ C \in \mathcal{D})$
  $Reduce2AVCGroups(C)$;
$Perform\ Global\ Reduction\ to$
 $Update\ AVCGroups$;

if $False\_Pruning$

  { *Additional Step*}

  foreach $(chunk\ C \in \mathcal{D})$
    $Reduce2AVCGroups(C)$;
  $Perform\ Global\ Reduction\ to$
   $Update\ AVCGroups$;

foreach $(node\ \mathcal{N}\ at\ level\ \mathcal{L})$
  $Determine\_best\_Split\_Condition(\mathcal{N})$
  foreach $(child(C) \in create\_children(node))$
    if not $satisfy\_stop\_condition(\mathcal{C})$
      $Add(\mathcal{C})\ to\ the\ set\ of\ nodes$
        $to\ be\ split\ at\ level\ \mathcal{L}+1$;

Figure 2: Parallelizing SPIES Using Our Middleware

The key observation in parallelizing the SPIES based algorithm is that construction of each type of AVC group (including small, concise, and partial) essentially involves a reduction operation. Each data item is read, and the class histograms for appropriate AVC sets are updated. The order in which the data items are read and processed does not impact the result, i.e. the value of AVC groups. Moreover, if separate copies of the AVC groups are initialized and updated by processing different portions of the dataset, a final copy can be created by simply adding the corresponding values from the class histograms.

Therefore, this algorithm can be easily parallelized using our middleware system. The overall algorithm is presented in Figure 2. We assume that the dataset is already distributed between the pro-

cessors. We also assume that with the use of pruning, the AVC groups for all nodes at any given level of the tree fit into the main memory. Therefore, only one pass on the dataset is required for each level of the tree, with the exception of the cases where false pruning occurs.

Initially, our discussion focuses on distributed memory parallelization. As we had noted while describing the sequential algorithm, there are up to three phases where the AVC groups are updated. These are, creating estimates of small and concise groups with the use of sampling, constructing small, concise, and partial groups during the completion phase, and finally, the update to the partial group if false pruning occurs. Each of these three phases is parallelized using a local reduction phase followed by a communication phase and a global reduction.

Each processor participates in the sampling step. Using a sample $\mathcal{S}$ drawn from the entire dataset $\mathcal{D}$, AVC groups for all nodes at the current level of the tree are updated. The data items within the sample are read one one chunk at a time, and AVC groups (in this case, the small and concise groups) are updated. After each processor finishes processing of all chunks corresponding to their respective samples, a communication and global reduction phase finalizes the values of the AVC groups. Each processor now holds the same set of values.

The pruning phase is replicated on all processors. After the pruning phase, all processors perform the completion phase, where they read all chunks in their portion of the dataset and update AVC groups. Again, a communication phase and a global reduction operation is used to finalize the values. The test for false pruning is replicated on all processors. Note that since all processors have the same AVC group, the result will be same on each processor.

If false pruning has occurred for any numerical attribute and for any node at the current level, all processors need to perform the additional step. Again, the processing here is similar to the sampling and completion phases.

Finally, in the end, the split condition for each node is chosen and the split is performed. If a child node created does not satisfy the stop condition (e.g. contain more than a specified number of data items), it is added to the set of nodes to be processed at the next level.

Shared memory parallelism can also be easily performed for each of the three reduction loops. As the data items are read from the disks or memory, they can be assigned to different threads. The threads can process these independently, provided the race conditions in updating the AVC groups are avoided. Because of the relatively low size of the reduction object here, we simply use replication to avoid race conditions, i.e., one copy of the AVC groups is created for each thread, which updates it independently. Later, the copies are merged together, by simply adding the corresponding values from class histograms.

In addition to shared memory parallelization of reductions, we also use task parallelism. This is done for the step where best split conditions are taken from different attributes, and for different nodes at the same level.

5.   EXPERIMENTAL EVALUATION

In this section, we describe the results from a series of experiments we conducted to evaluate the SPIES approach for sequential and parallel decision tree construction. Particularly, we carried out experiments with the following goals:

- Evaluating the distributed memory speedup obtained from parallelizing the existing RF-read algorithm (Section 5.2).

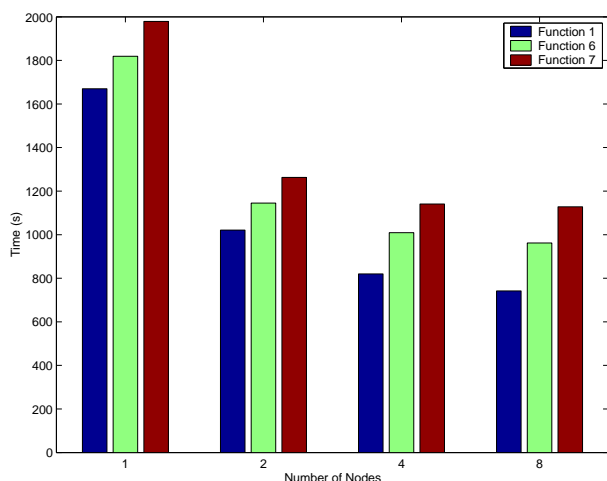- Examining the impact of the number of intervals used on the

Figure 3: Distributed Memory Speedup of RF-read (without intervals), 800 MB datasets, functions 1, 6, and 7

sequential and parallel execution times and memory requirements for the SPIES algorithm (Section 5.3).

- Evaluating the shared memory and distributed memory speedups obtained by parallelizing the SPIES algorithm (Section 5.4).

Before describing the experiments and the results obtained, we list the parallel configuration and the datasets used.

5.1 Experimental Set-up and Datasets

The experiments were conducted on a cluster of SMP workstations. We used 8 Sun Microsystem Ultra Enterprise 450's, each of which has 4 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each of the node have a 4 GB system disk and a 18 GB data disk. The data disks are Seagate-ST318275LC with 7200 rotations per minute and 6.9 millisecond seek time. The nodes are connected by a Myrinet switch with model number M2M-OCT-SW8.

We used two groups of datasets for our experiments, generated using a tool described by Agrawal *et al.* [1]. The size of each dataset in the first group is nearly 800 MB, and they have nearly 20 million records. The size of each dataset in the second group is nearly 1.6 GB, with nearly 40 million records. The size of the datasets in this group exceeds the available main memory on one node of the machines in our cluster. Each record in the above datasets has 9 attributes, of which 3 are categorical and other 6 are numerical. Every record belongs to 1 of the 2 classes. Agrawal *et al.* use a series of classification functions of increasing complexity to classify records into different groups. Tuples in the training set are assigned the group label (classes) by first generating the tuple and then applying the classification function to the tuple to determine the group to which the tuple belongs. Each group has three datasets corresponding to the use of functions 1, 6 and 7, respectively. The size of the decision tree resulting from the use of functions 1, 6, and 7 is in increasing order.

In our experiments, the *stop point* for the node size is 1,000,000, i.e., if the subtree includes fewer than 1,000,000 tuples, we do not expand it any further. The use of functions 1,6, and 7 results in trees with 27, 37, and 53 nodes, respectively. This is similar to what was done in evaluating the RainForest and BOAT. The reason is that after the size of records associated with a node becomes in-core, in-core algorithms can be used.

On the average, the size of the sample used in our experiments was 20% of the dataset. False pruning never occurred in our experiments by using this sample size.

5.2 Parallel Performance of RF-read

Before presenting results from our new algorithm, we evaluate the parallel performance from the RF-read, which is the first algorithm in the RainForest framework.

The results from the 800 MB dataset, and using functions 1, 6, and 7, are presented in Figure 3. The overall speedups on 8 nodes, and using functions 1, 6, and 7, are 2.25, 1.88, and 1.75, respectively. Thus, the speedups from parallelizing RF-read are very limited. Moreover, it is clear from this chart that almost no speedups are obtained in going from 4 to 8 nodes. Thus, we can expect parallel performance to be limited by a factor of nearly 2, irrespective of the number of nodes used. The reason for very limited parallel performance is the communication costs. The total volume of data communicated by each node to other nodes is 420 MB, 570 MB, and 660 MB with the use of function 1, 6, and 7, respectively.

An obvious question is, can we achieve better performance by using other algorithms that are part of the RainForest framework, such as RF-write and RF-hybrid. We believe that the answer is negative. RF-write and RF-hybrid can reduce the number of passes on the data, but do not reduce the size of the AVC groups, which will need to be communicated in a data parallel parallelization of RF-write or RF-hybrid. Moreover, in our experiments, RF-read required only pass on the input data for each level of the tree, since sufficient memory was always available for holding AVC groups for all nodes at any given level of the tree. In such a case, RF-hybrid simply reduces to RF-read, and RF-write will only incur extra costs of writing back input data. RF-write can potentially reduce the time required for determining the node with which a record is associated. However, this time was negligible in our experiments.

5.3 Impact of Number of Intervals on Sequential and Parallel Performance

We next focus on evaluating the impact that the number of intervals has on sequential and parallel performance of the SPIES algorithm. The results from 800 MB dataset, and using functions 1, 6, and 7, are presented in Figures 4, 5, and 6, respectively. We have presented the results using 100, 500, 1,000, 5,000, and 20,000 intervals for each numerical attribute, and using 1, 2, 4, and 8 nodes. Further, we have also included sequential and parallel performance from the original RF-read algorithm, i.e., using no intervals.

Initially, we focus on sequential execution times. The new algorithm, with the use of 100 intervals, reduces the sequential execution times by 13%, 15%, and 19%, with functions 1, 6, and 7, respectively. The use of 500 or 1000 intervals also gives better performance than the original RF-read algorithm. However, using 5,000 or 10,000 intervals, the sequential performance of SPIES algorithm is lower.

As we noted earlier, RF-read on our configuration only required a single pass of reading input data for each level of the tree. Thus, though the interval-based approach reduces the memory requirements also, any reduction in sequential execution time is purely because of the reduced computation and reduced number of cache misses. The reduction in the execution time is achieved because we have a lower number of candidate splitting conditions for numerical attributes, which reduces both the amount of computation in selecting a split condition, and cache misses in updating AVC groups.

In evaluating the sequential performance, it is interesting to note that using a small number of intervals gives better performance than
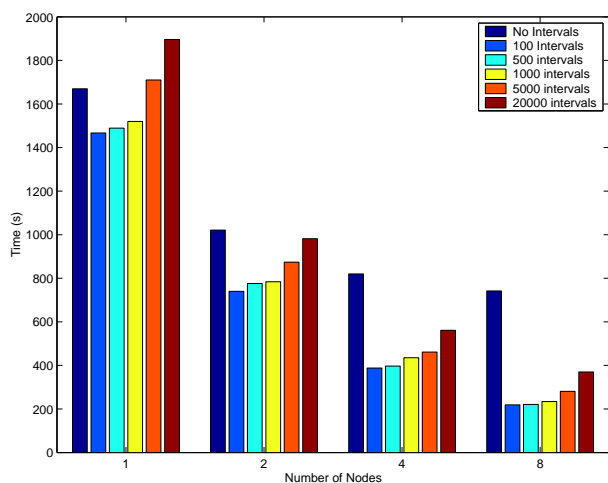
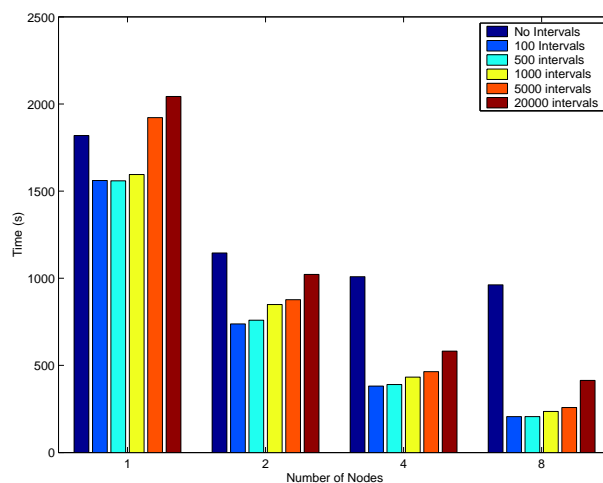Figure 4: Impact of Number of Intervals on Sequential and Parallel Performance, 800 MB dataset, function 1

Figure 5: Impact of Number of Intervals on Sequential and Parallel Performance, 800 M B dataset, function 6
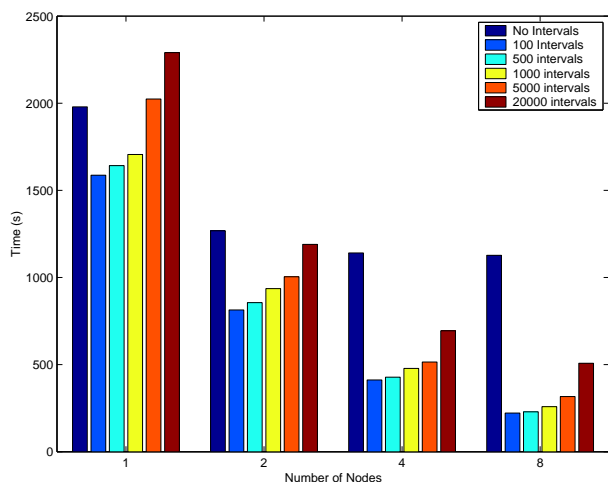

Figure 6: Impact of Number of Intervals on Sequential and Parallel Performance, 800 MB dataset, function 7
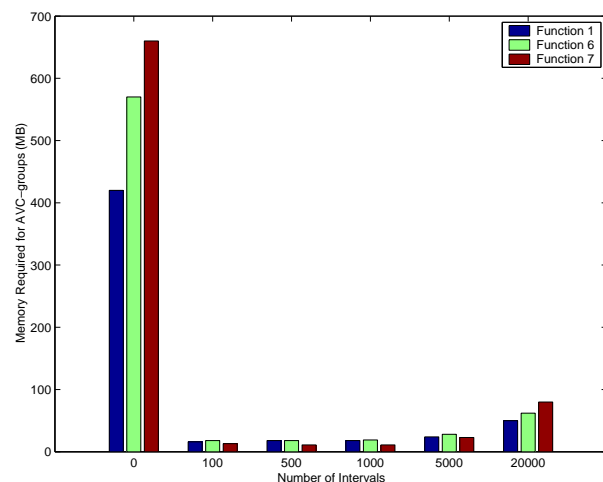

Figure 7: Impact of Number of Intervals on Memory Requirements, 800 MB datasets, functions 1, 6, and 7

using a larger number. This is because even with the use of 100 intervals, effective pruning and a significant reduction in memory requirements occurred. At the same time, the lower the number of intervals, the computation required is lower.

Next, we focus on parallel performance. The speedups on 8 nodes, and with the function 1, are 6.7, 6.74, and 6.50, with 100, 500, and 1000 intervals, respectively. The version with 100 intervals has the best performance on 2, 4, and 8 nodes. Execution time of parallel versions increases with the increasing number of intervals, because of both additional computation and higher communication volume. The same trend is seen with functions 6 and 7. In fact, because of higher ratio of computation, the speedups are higher. With the use of function 6, the speedups on 8 nodes are 7.58 and 7.57 with 100 and 500 intervals, respectively. With the use of function 7, the speedups on 8 nodes are 7.15 and 7.17, respectively.

We have also analyzed the variation in the total memory requirements for AVC groups as the number of intervals is increased. The results, for the functions 1, 6, and 7, and using 100, 500, 1,000, 5,000, and 20,000 intervals and no intervals are shown in Figure 7. The total memory requirements shown here are the sum of the sizes

of the AVC groups of all nodes of the decision tree that are expanded. Thus, this size also represents the total volume of interprocessor communication that each node has with all other processors. As can be seen from this figure, the total memory requirements without the use of interval-based approach are very high. As we described earlier, these result in very limited parallel speedups.

The use of interval based approach significantly reduces the memory requirements. The reduction in memory requirements are at least 85% for any of the values of the number of intervals we experimented with. All of this reduction comes from numerical attributes, which are 6 of the 9 attributes in our dataset. With the use of 100 intervals, the reduction in memory requirements are nearly 95%. Increasing the number of intervals of up to 1000 results in almost no difference in total memory requirements. With an increase in number of intervals to 5,000 and then 20,000, moderate increases in memory requirements are seen.

5.4 Scalability on Cluster of SMPs

We next focus on evaluating the overall scalability on cluster of SMPs, i.e., combining distributed memory and shared memory par-
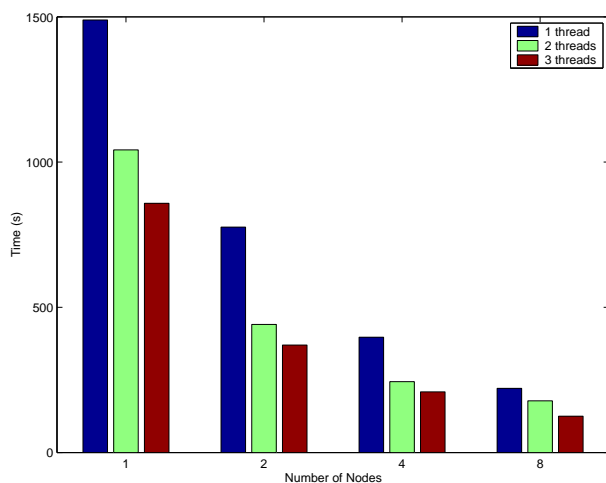
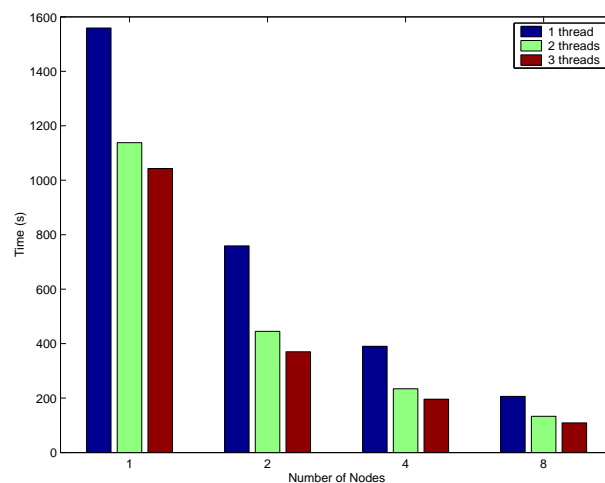**Figure 8: Shared Memory and Distributed Memory Parallel Performance, 800 MB dataset, function 1**

Figure 9: Shared Memory and Distributed Memory Parallel Performance, 800 MB dataset, function 6
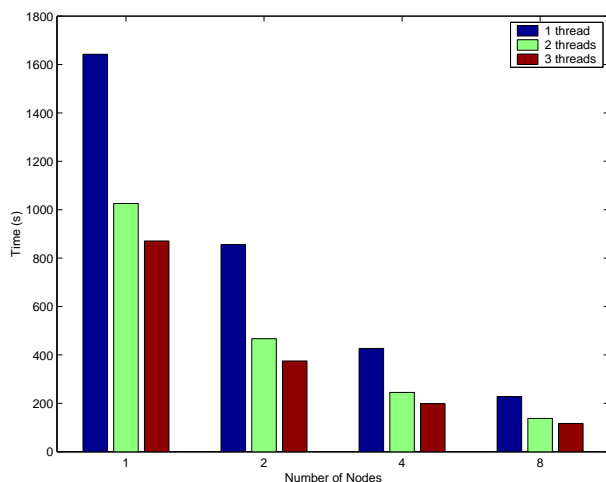

Figure 10: Shared Memory and Distributed Memory Parallel Performance, 800 MB dataset, function 7
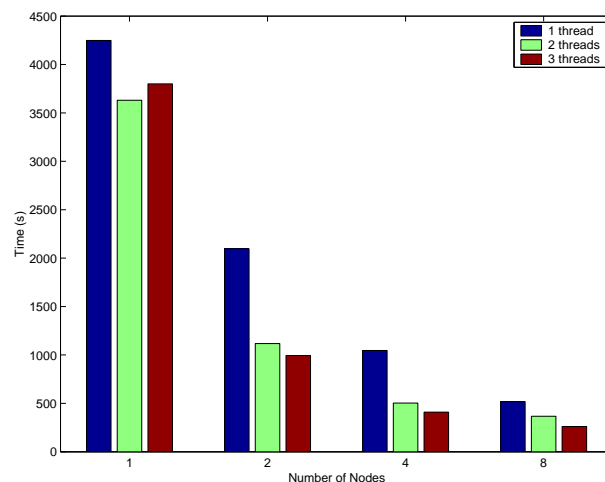

Figure 11: Shared Memory and Distributed Memory Parallel Performance, 1600 MB dataset, function 1

allelism. Each node in our cluster had 4 processors. We ran experiments using 1, 2, and 3 threads on each node. Each node typically had some background jobs, therefore, we could not use all processors for extended periods of time. We present results using the 800 MB and 1600 MB datasets, and with functions 1, 6, and 7.

The results from 800 MB dataset, and using functions 1, 6, and 7, are presented in Figures 8, 9, and 10, respectively. Using the function 1, the shared memory speedups using 3 threads are 1.74, 2.10, 1.90, and 1.77, on 1, 2, 4, and 8 nodes respectively. Using the function 6, the speedups using 3 threads are 1.48, 2.30, 2.18, 1.92, on 1, 2, 4, and 8 nodes, respectively. Using the function 7, the speedups on these 4 cases are 1.49, 2.05, 1.99, and 1.89. The shared memory speedups obtained depend upon a number of factors, but particularly, the amount of computation available and the ratio between computation and memory and/or disk accesses. When the data is available on a single node, the memory and disk accesses are the limiting factor on performance. Therefore, the use of additional threads gives only marginally better performance. The speedups are generally better with 2 and 4 nodes. However, the speedups are again limited with 8 nodes. This, we believe, is because the amount

of computation on each node is relatively low.

The results from 1600 MB dataset, and using functions 1, 6, and 7, are presented in Figures 11, 12, and 13, respectively. Using the function 1, the shared memory speedups using 3 threads are 1.18, 2.10, 2.55, and 2.0, on 1, 2, 4, and 8 nodes respectively. Using the function 6, the speedups using 3 threads are 1.17, 2.33, 2.29, and 2.12, on 1, 2, 4, and 8 nodes, respectively. Using the function 7, the speedups on these 4 cases are 1.09, 1.29, 2.4, and 2.28, respectively.

Note that with the use of one node, this dataset cannot be cached in the available main memory, whose size is bounded by 1 GB. Therefore, on one node, the implementation becomes I/O bound, and cannot benefit from the use of additional processors. However, as the data gets divided between the nodes, good shared memory speedups are obtained. The distributed memory speedups on 8 nodes with this dataset, and the functions 1, 6, and 7, are 8.20, 9.37, and 10.05, respectively. The super-linear speedups are because of memory caching of the datasets, as we explained above. With the use of 8 nodes and 3 threads per node, the speedups are 16.4, 19.8, and 23.1, with the functions 1, 6, and 7, respectively.
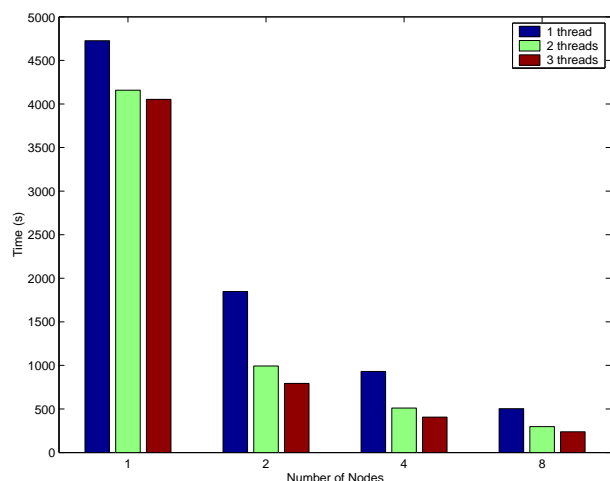
Figure 12: Shared Memory and Distributed Memory Parallel Performance, 1600 MB dataset, function 6

Figure 13: Shared Memory and Distributed Memory Parallel Performance, 1600 MB dataset, function 7

6. RELATED WORK

We now compare our work with related research efforts.

One effort particularly close to our work is the BOAT approach for decision tree construction, developed by Gehrke *et al.* [6]. BOAT uses a statistical technique called bootstrapping to reduce decision tree construction to as few as two passes over the entire dataset. In addition, BOAT can handle insertions and deletions of the data, which we cannot. The important contribution of our work is parallelization on cluster of SMPs and using the same interface and runtime support that has been used for other mining algorithms. In contrast, we are not aware of any work on parallelizing BOAT. Moreover, the two algorithms differ in how samples are used. Our approach offers a bound on the probability of requiring a second pass. In the future, we plan to consider parallelization of BOAT, as well as examine how we can reduce the number of passes on the data in our approach.

CLOUDS is another algorithm that uses intervals to speedup processing of numerical attributes [3]. However, their method does not guarantee the same level of accuracy as one would achieve by considering all possible numerical splitting points. It should be noted, however, that in their experiments, the difference is usually small. Further, CLOUDS always requires two scans over the dataset for partitioning the nodes at one level of the tree. In our experiments, our method only required one pass for each level of the tree.

As compared to earlier work on decision tree construction, like SLIQ [16] and SPRINT [18], the SPIES algorithm significantly reduces the size of the memory-resident data-structure, and eliminates the need for preprocessing and writing-back of datasets during the execution. Also, the speedups we are able to achieve are much higher than what was reported from these algorithms. Some other more recent efforts have focused on reducing the memory and I/O requirements of SPRINT [14, 20]. However, they do not guarantee the same precision from the resulting decision tree, and do not eliminate the need for writing-back the datasets.

The only previous work on shared memory parallelization of decision tree construction on disk-resident datasets is by Zaki *et al.* [21]. They have carried out a shared memory parallelization of SPRINT algorithm. Our work is distinct in parallelizing a very different method for decision tree construction. In parallelizing SPRINT, each attribute list is assigned to a separate processor. In comparison, we parallelize updates to reduction objects. Narlikar
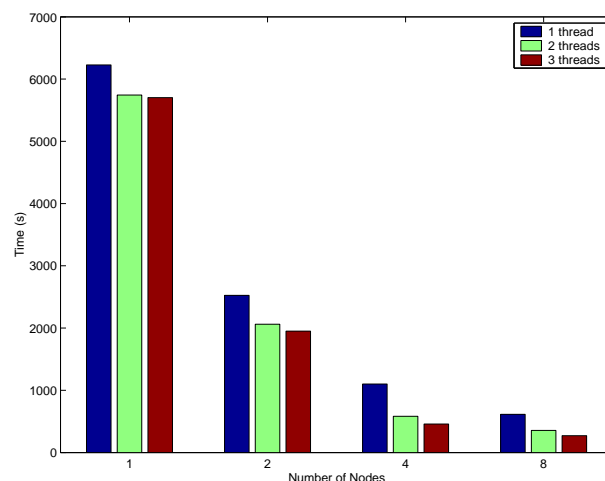
has used a fine-grained threaded library for parallelizing a decision tree algorithm [17], but the work is limited to memory-resident datasets.

Our middleware system, FREERIDE, is based upon the premise that a common set of parallelization techniques can be used for algorithms for a variety of mining tasks [11, 12]. However, we believe that Skillicorn was the first to recognize the similarity between the parallel versions of several mining algorithms [19]. Our previous work on this middleware system did not report distributed memory parallelization of any decision tree construction algorithm. In a recent paper, we performed shared memory parallelization of the RF-read algorithm [13].

7. CONCLUSIONS

In this paper, we have presented and evaluated a new approach for decision tree construction, with a particular focus on parallel efficiency. The key idea in our approach is to combine RainForest like AVC-groups with sampling.

Our approach achieves three important properties. The first is very low communication volume when the algorithm is parallelized. This, as we have shown, results in almost linear speedups. The second property is that we do not need to sort or preprocess the training records, and there is no need to partition and write-back input data during the execution. As the technology trends are creating an increasing difference between the CPU speedups and the disk speeds, we believe that this is a very useful property. The third important property of our approach is that it could be efficiently parallelized using the same high-level interface and runtime support that we had earlier used to parallelize association mining and clustering algorithms. We consider this an important step towards offering high-level support for developing parallel mining implementations.

Another important contribution of this paper is that we have presented results from parallelization on a cluster of SMPs, i.e., combining shared memory and distributed memory parallelization, and on disk-resident datasets. We are not aware of any previous work that had achieved this for decision tree construction.

8. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on*

Knowledge and Data Eng., 5(6):914-925,, December 1993.

[2] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, June 1996.

[3] K. Alsabti, S. Ranka, and V. Singh. Clouds: Classification for large or out-of-core datasets. http://www.cise.ufl.edu/ ranka/dm.html, 1998.

[4] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *In Proceedings of Workshop on Large-Scale Parallel KDD Systems, in conjunction with the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 99)*, pages 47 – 56, August 1999.

[5] P. Domingos and G. Hulten. Mining high-speed data streams. In *SIGKDD00*, 2000.

[6] J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh. Boat– optimistic decision tree construction. In *In Proc. of the ACM SIGMOD Conference on Management of Data*, June 1999.

[7] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB*, 1998.

[8] S. Goil and A. Choudhary. Efficient parallel classification using dimensional aggregates. In *Proceedings of Workshop on Large-Scala Parallel KDD Systems, with ACM SIGKDD-99*. ACM Press, August 1999.

[9] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

[10] Ruoming Jin and Gagan Agrawal. An efficient implementation of apriori association mining on cluster of smps. In *Proceedings of the workshop on High Performance Data Mining, held with IPDPS 2001*, April 2001.

[11] Ruoming Jin and Gagan Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.

[12] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, April 2002.

[13] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Decision Tree Construction Using a General Middleware. In *Proceedings of Europar 2002*, 2002.

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

[15] O. L. Mangasarian. *Nonlinear Programming. Classics in Applied Mathematics.* Society for Industrial and Applied Mathematics., 1994.

[16] M. Mehta, R. Agrawal, and J.Rissanen. Sliq: A fast scalable classifier for data mining. In *In Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[17] G. J. Narlikar. A parallel , multithreaded decision tree builder. Technical Report CMU-CS-98-184, School of Computer Science, Carnegie Mellon University, 1998.

[18] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*,

pages 544–555, September 1996.

[19] David B. Skillicorn. Strategies for parallel data mining. *IEEE Concurrency*, Oct-Dec 1999.

[20] A. Srivastava, E. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. In *In Proc. of 1998 International Conference on Parallel Processing, 1998.*, 1998.

[21] M. J. Zaki, C.-T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. *IEEE International Conference on Data Engineering*, pages 198–205, May 1999.

[22] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14 – 25, 1999.