A Review of Distributed Decision Tree Induction

Gregory Gay

West Virginia University, Morgantown, WV, USA greg@greggay.com

Abstract. A common practice in the data mining field is *classification*, the identification of unknown data based on evidence gathered from its attributes. A popular type of classification is decision tree inference. Decision tree classifiers select a value for an attribute that will maximize the information gained. It then splits the remaining data based on that value. The algorithm will recursively repeat this process until each leaf only contains data fitting to a particular classification value.
The success of these data inference techniques has led to an expectation of fast and accurate classification on larger and larger data set. These data sources are often gigabytes to terabytes in size. To process these large, high-dimensional data sets within a useful time bound, it becomes desirable to parallelize the learning process. This can be done by distributing the data, the algorithm, or both. This paper presents a technical review of both paradigms.

1 Introduction

Data mining has become a major research field over the past twenty years. The overarching goal of data mining exercises is to discover interesting , yet previously unknown, patterns from massive caches of data. To this end, it is useful to know how to identify a piece of data. This is the practice of classification, to categorically sort data into a series of *classes*. Classification is heavily used in fields like financial analysis and medical diagnosis [18].

In classification, we are given a set of data that has been split into *testing* and *training* subsets. Each entry in a data set consists of several *attributes*, which can be numerical or categorical (from an unordered domain, i.e. job type or family name). One of these categorical attributes is specified as a *classification attribute*, and its values model the possible classifications of that data entry. The goal of such classification is to create a concise model of the classification attribute based on evidence gathered from the other attributes. The *classifier* uses the training set to build this model, which is then applied to the testing set. Decision tree classifiers, which are elaborated on in section 2, build these models by recursively splitting the training set such that all of the records within a branch have the same class label.

Classification is expected to be both fast and accurate, which is an increasingly difficult task as the size of the data grows. Large datasets, from gigabytes

to terabytes, are appearing more and more commonly [3, 9, 18]. Much of the focus of data mining research over the past few years has been in how to deal with these massive data collections. While data preprocessors can subsample from a large data cache [16, 20, 21], vital information can be lost. It is still desirable to have classifiers that can quickly and accurately assess these massive collections. Because of the large memory and computation requirements, parallelization of the data or the classification algorithm (or both) has become a viable approach.

In the following sections, I will examine both approaches.

2 Decision Tree Induction

Decision tree algorithms are a widely-used family of machine learning techniques [4, 15, 22] for the classification of data. These methods have several advantages over other algorithm families, including the ability to select from all attributes or some relevant subset of them, the ability to identify complex predictive relations among these attributes, and the ability to produce rules that are easily understood by humans. A large variety of these algorithms exist in the literature. However, most of them greedily and recursively elect the attribute that is used to partition the data set into subsets until each leaf node in the tree only contains data with the proper class label. This is known as the *tree growth phase*. Tree construction is often followed by a *pruning phase* in order to avoid overfitting.

Many classification algorithms, such as SPRINT [13] first partition the training set into *attribute lists*. Each entry in one of these lists contains the attribute value, a class label, and a record identifier that ties the entry back to the overall data set. The attribute lists for numerical attributes are independently sorted by the attribute values. Lists for discrete attributes are not sorted. Initially, the entire training set is associated with the root node. At each stage, the leaf node is split into two child nodes that partition the remaining data between them. This is done in a two-stages, where the algorithm *finds the best split point* and *performs the split*.

Decision tree classifiers, such as Ross Quinlan's [22] ID3 (Iterative Dichotomizer 3) and its more recent variants, search in a greedy fashion for attributes that yield the maximum amount of information, the *gain*, for determining the class membership of instances of some training set T. The construction of a decision tree is completed by recursively partitioning T into subsets based on the value of the selected *split point* until each leaf only contains instances of exactly one of the classes. At each stage of construction, an attribute is selected that maximizes the estimated expected information gained from knowing the value of said attribute.

Algorithms for decision tree induction differ from each other in the criterion that is used to evaluate the attributes for selecting the proper split point. The choice of the attribute at each node of the decision tree greedily maximizes (or minimizes) the chosen splitting criterion. Quinlan's ID3 and C4.5 algorithm split based on entropy [22], while the Gini index [15] is used by Breiman's CART

method, among others. Although those are the most common, other criterion [4] have been proposed.

Often, decision tree classifiers include a *pruning phase*. In this phase, the tree is trimmed in order to avoid data overfitting. For the sake of simplicity, and because the literature on parallelized decision tree induction often overlooks this phase, I will limit my discussion to the *tree growth* phase. It is, however, relatively straightforward to incorporate a wide variety of pruning methods into any of the parallel algorithms.

2.1 Finding a Split Point

Consider a set of instances T which is partitioned into N classes $C_1, C_2, ..., C_N$ such that $T = C_1 \cup C_2 ... \cup C_N$ and $C_i \cap C_j = \emptyset \forall i \neq j$. The probability that a randomly chosen entry $t \in T$ belong to class C_j is $P(j) = \frac{|C_j|}{|T|}$, where $|T|$ denotes the cardinality of set T.

The *entropy* of the set T is a measure of the expected information needed to identify the class of instances from set T. The entropy is defined as:

$$entropy(T) = -\sum_j \frac{|C_j|}{|T|} * log_2(\frac{|C_j|}{|T|}) \tag{1}$$

The estimated *Gini index* for the set T is defined as:

$$gini(S) = 1 - \sum_j (\frac{|C_j|}{|T|})^2 \tag{2}$$

Given an impurity measure such as the entropy or Gini index, or some other measure based on the probabilities $P(j)$, the estimated *information gain* for an attribute a, relative to a data set T, can be defined as follows:

$$gain(T, a) = I(T) - \sum_{v \in Values(a)} \frac{|T_v|}{|T|} * I(T_v) \tag{3}$$

In this equation, $Values(a)$ is the set of all values for the attribute a, T_v is a subset of T where all instances of attribute a have value v, and $I(T)$ is an impurity measure ($entropy(T), gini(T)$, or some other suitable measure).

Regardless of the splitting measure, the decision tree algorithm needs the same information from the underlying data set. In both the case of entropy and gini, it is necessary to gather the relative frequencies computed from the appropriate data entries. In fact, many classifiers can now incorporate additional splitting criteria beyond the impurity measure, provided that these measures can be calculated based purely on statistics gleaned from a single scan of the testing data. Some of these additional measures include misclassification rate, one-sided purity, and one-sided extremes [4]. This is actually quite useful, as different criteria often offer different insights about the data. This observation, that all of the measures we use to split the data are based on the same information,

is what allows us to efficiently construct decision trees from distributed data sources. This results in provably exact algorithms for decision tree classification from fragmented data sets [5].

2.2 Performing the Split

Once the best split point has been determined at a tree node, the data associated with that node is divided into two partitions, one for each new child node. Therefore, every attribute list must be partitioned into two new lists. The list of the splitting attribute is divided into two by scanning the original list and moving each entry into one of the two new lists based on the splitting condition. Basically, given an attribute a and split point v, data entries will go into the first list if $a \leq v$ and into the second list if $a > v$. A hash table is used to split the other attributes. While scanning the attribute list, a hash table is created on the record identifiers. The record id for each entry of the attribute list is inserted into this hash table along with the name of the child node where the corresponding data entry is placed. The other attribute lists are split be scanning each list, referencing the hash table using the identifiers, and moving the entries into one of the two new lists. If an attribute is numerical, rather than discrete, the lists are split in such a way that the order of the entries is maintained. Because of this, numerical attribute lists do not have to be resorted at each level of the tree.

3 Paradigm A: Data Parallelism

Advances in computing technology, as well as the growing power and reliability of data acquisition techniques, have made it possible to gather and store large volumes of data in digital archives. Given the large size of these data sets, gathering all of the data in a centralized location or in a single, massive file is neither desirable nor always feasible because of the bandwidth and storage requirements. In some domains, such as in the medical field, data may be fragmented in order to address privacy and security constraints. In these cases, there is a need for data inference algorithms that can learn from fragmented, parallelized data. In such settings, learning can be accomplished by combining the fragments in one location and parallelizing the algorithm (see the next section, *task parallelism*. Alternatively, the algorithm can be augmented to build a decision tree from the parallelized data. This is what is known as *data parallelism*. There are two common types of data parallelism: *horizontal fragmentation* where subsets of data entries are stored at different locations, and *vertical fragmentation* where subsets of attributes are stored at different sites.

Assuming that we have a partially constructed decision tree. We want to find the best splitting point. Let $A_j(P)$ denote the attribute at the jth node along the path P that started from root attribute $A_1(P)$. This path leads to the node $A_l(P)$ in question at depth l. Let $V(A_j(P))$ denote the value of the attribute corresponding to the jth node along path P. When adding a node below $A_l(P)$,

the set of examples to be considered satisfies the following constraints on the value of the attributes:

$$L(P) = [A_1(P) = V(A_1(P))] \wedge [A_2(P) = V(A_2(P))] ... \wedge [A_l(P) = V(A_l(P))] \quad (4)$$

where $[A_j(P) = V(A_j(P))]$denotes that the value of the jth attribute along the path P is $V(A_j(P))$.

It follows from this and the preceding discussion on splitting criteria that the information required for the construction of decision trees is the counts of entries that satisfy the specified constraints on the values of the particular attributes [5]. These counts must be obtained once for each node that is added to the tree. If distributed information extraction operators can be devised that obtain the necessary counts from distributed data sources, the decision tree algorithms will construct the exact same trees for distributed data sets that it would construct for an entirely localized data set. This holds as long as the same splitting criterion is used for both cases.

3.1 Horizontal Fragmentation

In data sets that have been horizontally fragmented, the entries are equally divided into a number of subsets equal to the number of different storage sites. In order to identify the best splitting point for the current node, all of the sites are visited and the counts corresponding to candidate splits of the node are accumulated. The learner uses these frequency counts to find the attribute that yields the most information gain to further partition the set of examples at that node. Given $L(P)$, as defined in equation 4, in order to split the node $A_l(P) = V(A_l(P))$, the algorithm must obtain the counts of examples that belong to each class for each possible value of each candidate attribute.

Caragea et.al. have analyzed both the time and communication complexity of such algorithms [5]. Let $|E|$ be the total number of examples in the data set T, $|A|$ be the total number of attributes, V the maximum number of possible values per attribute, M the number of data sites, N the number of classes, and $size(D)$ the total number of nodes in the decision tree D. For each node in D, the algorithm must scan the data at each site in order to calculate the corresponding counts. This sums to:

$$\sum_{i=1}^{n} |E_i| = |E| \quad (5)$$

Therefore, in the case where a single learner visits each site, the time complexity of the algorithm is $|E||A|size(D)$.

For each node in the decision tree D, each site must transmit counts based on the local data. These counts form a matrix of size $N*|A|*V$. This means that the total amount of information transmitted between sites, the communication complexity, is equal to $N * |A| * V * M * size(D)$. The bounds presented here may be improved by taking into account the height of the tree rather than the number of nodes. Techniques for this are presented in [12, 13].

Two algorithms designed to work with horizontally fragmented data sets are SLIQ and SPRINT [13]. When evaluating a split based on a continuous attribute, these algorithms sort the list of examples by the value of this attribute. In order to avoid sorting the examples every time that a continuous attribute is evaluated, they both use separate lists for each attribute. These lists are sorted once at the beginning of tree construction. Each processor is responsible for maintaining its own attribute lists. SLIQ uses a special list, called the *class list*. This list has the values of the class for each example. This class list is stored in memory for the entire tree construction phase in order to increase the efficiency of the algorithm. SPRINT eliminates this class list by adding the class label to every attribute list entry. The evaluation of the possible splits is performed by all processors, and they communicate amongst themselves to determine the best split. Each processor then splits its attribute lists based on the chosen attribute/value pair. Both of these algorithms report good performance and they seem to scale well to large data sets. However, the splitting operation requires a high communication load for both algorithms.

3.2 Vertical Fragmentation

In a system where the data set is vertically distributed, it is safe to assume that each example has a unique identifier associated with it. Individual attributes or subsets of attributes, along with a list of $(value, identifier)$ pairs for each are distributed across different sites. Correspondence between these subsets can be conducted using the included identifiers.

As in horizontal distribution, given $L(P)$ (equation 4), in order to split a node that holds to the rule $A_l = V(A_l(P))$, the algorithm must obtain the frequency counts of examples that belong to each class for each possible value of each chosen attribute. Since each site only has a subset of the total number of attributes, the set of identifiers for the set of examples that match $L(P)$ must be transmitted to the sites. For each node in the decision tree D, each site must use this table to compute the relative counts of examples that satisfy $L(P)$ for the attributes stored at *that site*. The size of the attribute-value table at each site is given as $|E|$ and the number of attributes is bounded by the total number of attributes at all sites, $|A|$. The time complexity works out to $|E|*|A|*M*size(D)$. Again, this can be improved by parallelizing the learning algorithm. For each node in the tree D, we must transmit the set of identifiers for the examples that satisfy constraint $L(P)$ to each site and get back the relevant frequency counts for the attribute stored at that site. The number of identifiers is bound by $|E|$ and the number of counts is bound by $N * |A| * V$. Thus, the communication complexity of these algorithms is given by $(|E| + N * |A| * V) * M * size(D)$ [5]. The authors of [12, 13] provide techniques that can improve these complexities.

Parallelism using vertical data fragmentation can often suffer from load imbalance [19]. This is due to the evaluation of continuous attributes, which require more processing power than the evaluation of discrete attributes. In order to avoid this, the continuous attributes must be evenly distributed among the different sites at the time where the data is split. The vertical approach often

leads to poor scalability. The number of attributes is rarely high; as the number of sites grows, the number of attributes at each site decreases. As this happens, the time spent in communication will quickly outgrow the time spent processing the data. This is why horizontal fragmentation is far more common.

3.3 Distributed vs. Centralized Learning

Even when the data set is distributed across multiple sites, it would be possible to collect those fragments and reassemble them in a centralized location. So, then, the question to ask is why we should leave the data at all of those sites? The most obvious situation is when privacy or security rules prevent transfer of the raw data. In these situations, it becomes necessary to have a learning algorithm that can perform its operations based solely on statistical summaries derived from each site. Even when it is possible to access the raw data, distributed versions of algorithms compare favorably with the corresponding centralized technique whenever its communication cost is less than the cost of collecting all of the data in one place [5]. From the proceeding analysis, it follows that in the case of horizontally fragmented data that a distributed algorithm has an advantage when $N * V * M * size(D) \leq |E|$ since the cost of transferring the data is given by its actual size, $|E| * |A|$. In the case of vertically fragmented data, the distributed algorithm is desirable when $size(D) \leq |A|$, as the cost of transmitting the data is given by the lower bound of $|E| * |A|$. In practice, high-dimensional large data sets often meet these conditions, making distributed algorithms desirable over centralized ones.

4 Paradigm B: Task Parallelism

The construction of decision trees in parallel is the *task parallelism* approach. This approach can be viewed as dynamically distributing the decision nodes among a set of processors for further expansion [19]. A single process containing the entire training set begins the construction process. When the number of child decision nodes equals the number of available processors, the nodes are split among them. Each processor then proceeds with the execution of the tree construction algorithm and builds sub-trees from that point.

Implementing parallel algorithms for decision tree classifcation is a difficult task for multiple reasons. First, the shape of the trees are highly irregular and are determinable only during runtime. Because of this, any static allocation scheme will probably suffer from a high load imbalance, and the amount of processing power needed at each node will vary significantly. The authors of [8] present an implementation that avoids this issue, but it requires the whole training set be mirrored in the memory of all of the processors. Another problem - even if the successors of a node are processed in parallel, their construction requires sharing data with the parent node. If the data is dynamically distributed, then it becomes necessary to implement a strategy for data movement. If the distribution is not handled properly, the performance of the parallelized algorithm may degrade due to the subsequent loss of locality [2, 10].

5 Paradigm C: Hybrid Parallelization

Most implementations of parallelized algorithms are not following strict *task parallelization*. They actually are implementations of a *hybrid parallelization*, they combine a parallelized algorithm with parallelized data (usually following a horizontal fragmentation scheme). The implementation of hybrid parallel algorithms is motivated by the balance between the amount of processing at each node and the required volume of communication. For the nodes that would cover a significant amount of data entries, data parallelism is useful to avoid the problems of load imbalance mentioned in the previous section. However, for the nodes containing fewer entries, the time spent busy in communicating can be higher than the time spent processing the examples. To avoid this problem, if the number of examples stored at a node is lower than a specific threshold, one of the processors continues with the construction of the tree rooted at this node. This is made possible through task parallelization.

Usually, the switch between data and task parallelization (if one exists) is performed when the communications cost overcomes the processing and data transfer costs. Some algorithms do not make a switch, like the SMP algorithm presented in [11]; they always operate in task parallelization mode on data split over multiple sites. In general, hybrid parallelization approaches faster runtime and efficiency results than other techniques [14]. The following subsections detail hybrid algorithms found in the literature.

5.1 C4.5 Parallel Implementation

The authors of [19] have designed a parallel implementation of Ross Quinlan's seminal C4.5 decision tree algorithm. Their design of the decision tree follows a breadth-first strategy. This implementation follows a hybrid parallelism approach with data parallelism at the beginning of the decision tree construction phase and task parallelism at the lower nodes of the tree that contain a number of examples that falls below a threshold (based on the balance concept described previously).

The data parallelism is implemented using a horizontal fragmentation scheme. This strategy is similar to the one employed by SLIQ [13]. Their implementation was designed to be executed in a distributed memory environment, where each processor has its own private memory. A uniform load balance is achieved by equally distributing data entries among the processors and using a breadth-first strategy to build the actual decision tree. To avoid the problem of load imbalance, when a process finished exploring it nodes, it sends a request for more nodes from the other processors.

Each processor is responsible for building its own attribute lists and class list from the subset of data assigned to it. The continuous attribute lists are globally sorted by the possible values of that attribute using a sorting algorithm from [6]. For each continuous attribute, this gives each processor a list of sorted values where the first processor has the lower values of the attribute and the subsequent processors have higher values. After this sort, each processor updates its class

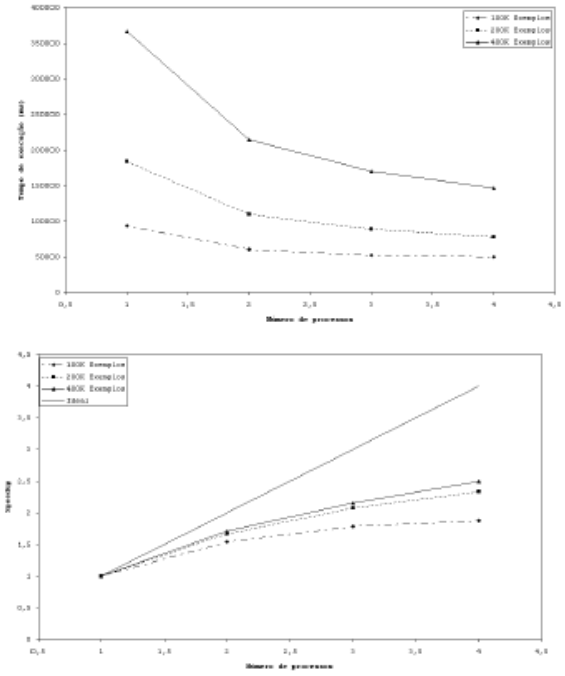

Fig. 1. Speedup results for the parallel C4.5. The x-axis is number of processors, y-axis is time in seconds.

list with the information corresponding to the new values of the continuous attributes that were not previously assigned to it.

Each processor has a distinct subset of the overall data set in their globally sorted lists. Before each processor starts evaluating the possible split points in their entries, the distributions must reflect the data entries assigned to the other processors. For discrete attributes, all of the distributions are gathered in all processors that locally will compute their gain. For continuous attributes, the gain is calculated based on the distributions before and after the split point. Which distributions come before and after is based on the "rank" of the processor (i.e. what subset of the sorted data that it has). After evaluating all of the local data for split points, the processors communicate among themselves in order to find the best split from the N local best splits.

Once this point is agreed upon, the split is performed by creating the child nodes and dividing the set of examples among them. This requires that each processor update the pointers to the nodes in their class lists and divide the attribute list, assigning the new lists to the child nodes.

The decision tree is constructed using horizontal data fragmentation as long as the number of examples covered by the nodes stays above a pre-defined thresh-

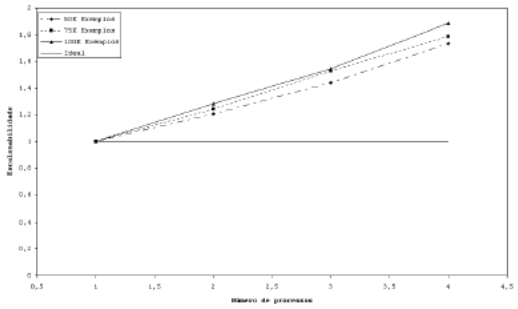

Fig. 2. Scale-up results for the parallel C4.5. The x-axis is number of processors, y-axis is time in seconds.

old. When the number of data entries in the remaining nodes to be explored falls below this line, they are evenly distributed among the processors. Deciding what this threshold will be - that is, when to switch between data and task parallelism - is one of the most important considerations in the performance of an algorithm [14]. If the switch happens too early, it will lead to load balance issues. If it happens too late, the performance will suffer from high communication costs. This hybrid C4.5 implementation makes the switch whenever the total communication cost for building a node overcomes the cost of building it locally at one process plus the cost of moving the set of examples between processors.

This parallel implementation preserves the main features of the C4.5 tree learner, while addressing flaws in that algorithm like a lack of support for missing or unknown attribute values. Since the same evaluation criteria (entropy) is considered in the parallel version, it obtains the same values when assessing split points. This leads to the same decision trees and same classification results as a localized C4.5. The parallel implementation is actually more efficient because it uses the attribute and class lists like in SLIQ [13]. The reason for uing separate lists for each attribute is to avoid sorting the examples every time that a continuous attribute is evaluated. Instead, the algorithm only has to sort the continuous attributes once at the beginning of the tree construction phase. Removing this inefficiency makes it possible to work on larger sets of data.

In tests on large synthetic data sets, the parallel C4.5 performed very well in terms of speed. It vastly outperformed the localized C4.5 on a data set with 400K examples. It also performed well on scalability, as more and more processors were added. However, there were limitations due to the communication overhead. Figures 1 and 2 show graphs of the learner's performance.

5.2 SMP Decision Tree Classifier

Clusters of shared-memory processors, in which shared-memory nodes with a small number of processors (usually 2-8) are connected together with a high-

speed interconnect, have become a popular alternative to distributed-memory and shared-memory machines [11]. An increasing number of these SMP clusters are being deployed in universities, research labs, and in private industry. These clusters provide a two-tiered architecture where a combination of shared-memory and distributed-memory algorithms can be deployed. The SMP algorithm proposed by [11] employs a hybrid approach and is based, in part, on the SPRINT distributed-memory algorithm [13] and the BASIC shared-memory algorithm [17].

The training dataset is horizontally fragmented across the SMP nodes so that each node carries out tree construction using an equal subset of the overall data set. Attribute lists are dynamically scheduled to take advantage of the parallel light-weight threads within the SMP node. If there are K processors in a SMP node, then K threads will spawn. One of these threads is designated as a master thread. Master threads are responsible for processing the attribute lists as well as exchanging data and information between SMP nodes. The numerical attribute lists are globally sorted such that the first node has the first portion of the data set, and subsequent nodes have subsequent portions. Thus, like in the C4.5 implementation above, sorting only happens once during the tree construction phase.

Following the partitioning of the training set, the entire decision tree is built in each SMP node. Initially, the root node of the tree in an SMP node is associated with the local portion of the attribute list in that node. When a tree node is split into its children, the local attribute lists are partitioned among those children. Thus, no communication costs are incurred due to the attribute lists as the tree is grown in each SMP node. However, some communication is required to find the global best split point. This communication causes a global synchronization among the tree nodes. Although each SMP node has the same number of data records, the number of records per *tree node* may differm which can decrease performance as more global synchronizations occur. In order to reduce the total number of synchronization points, the split points and histograms for all of the leaf nodes are updated in each SMP node *before* performing any sort of synchronization operation. The algorithm will proceed in a breadth-first manner, processing all of the leaf nodes before processing any of the new child nodes.

Before finding the new best split point, the attribute lists in a SMP node are inserted into a queue shared by all of the threads running on the SMP node. This queue is used to schedule attribute lists to threads. Any idle thread will receive an attribute list from the queue to process. A mutual exclusion lock prevents two threads from trying to process the same attribute. Two histograms, C_{below} and C_{above} are used to find the ideal split point for a numerical attribute. The histograms are initialized to reflect the distribution of class labels below and above the current SMP node in the list of clusters. These histograms are calculated when the parent node is split. After the histogram is initialized, attribute are scheduled to the threads for processing. Each leaf node calculates and stores the best split point along with its *Gini value*. Each thread compares its local best

split points with all of those along the leaf nodes, replacing their own value if a better one is found. Mutual exclusion locks ensure that only one update can occur at a time. At the end of processing all of the attribute lists, each SMP node has the best split point for all of the local numerical attributes and partial frequency count tables for discrete attributes. The master threads on SMP nodes perform a global minimum operation on the local split points and a global sum operation on the count tables so that SMP node 0 can compute the overall best split point for all attributes. It will broadcast this point to all of the SMP nodes.

Once the best split point is determined, leaf nodes will be divided into child nodes. Each processor exchanges record identifiers and the corresponding child destination with the other processors to create a new hash table. Partitioning of the attributes is carried out by dynamically scheduled threads, like what is done when finding the best split point. When a numerical attribute is partitioned, the corresponding local histograms are initialized for the next level of the tree.

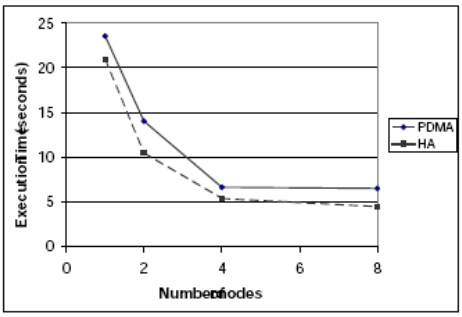

Fig. 3. Execution times for finding the best split point

In order to evaluate the performance of their SMP algorithm, they compared their hybrid shared & distributed-memory approach with another purely distributed-memory algorithm. They conducted a variety of experiments on synthetic data sets with up to 800K entries and varied the number of processor nodes from two to eight.

Some of their results can be seen in figures 3, 4, 5, and 6. These figures show (in order) results for execution time for calculating the split, communications time, execution time for performing the split, and total execution time for generating the decision tree. The SMP algorithm performs similarly, but slightly faster, when finding the split. Their algorithm is much better when it comes to communication time because of the increased reliance on inter-processor communication. Both algorithms perform similarly on time to split the attribute list. This is because SMP provides no real benefit for intensive I/O operations. The SMP algorithm also performs well on total execution time, with the performance gap increasing as the number of nodes increases.

5.3 INDUS

The premise of learners that work on distributed data sources assumes that it is possible to extract from those sources the information needed for classification (that is, counts of instances that satisfy specific constraints on the values of

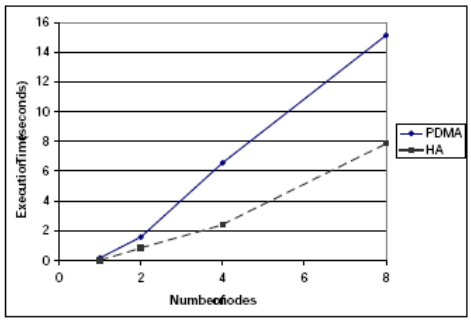

Fig. 4. Change in communication time as the number of nodes increases. The x-axis is number of processors, y-axis is time in seconds.

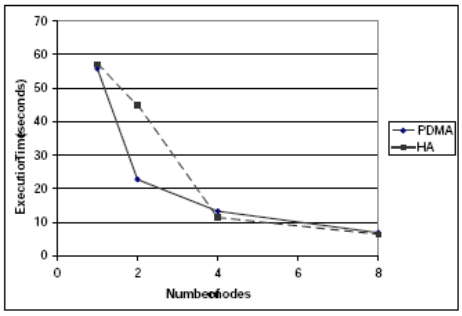

Fig. 5. Execution time for performing the split. The x-axis is number of processors, y-axis is time in seconds.

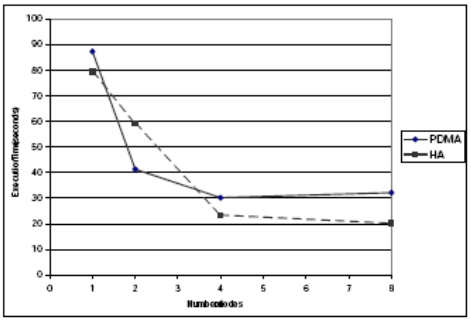

Fig. 6. Total execution time for building the decision tree. The x-axis is number of processors, y-axis is time in seconds.

the attributes). This is a straightforward process when the data has a homogeneous structure and straightforward semantics. However, this task tends to be far more complex when given the heterogeneous semantics of many real-world data collections [23].

INDUS is a multi-agent system used for knowledge acquisition from such heterogeneous data sources [5]. This system stands in contrast to many other hybrid parallelized algorithms because it has been specifically designed to deal rely only on statistical summaries of the data. Many other algorithms proposed in the literature [1,7,24] are only designed to handle horizontally fragmented data. INDUS can use data fragmented in either manner, and can work with summaries of the data in situations where the raw data is inaccessible. It was designed to provide a unified query interface over a set of distributed, heterogeneous, and autonomous data sources as if the dataset were a table. For the purposes of decision tree learning, the learning algorithm formulates queries whose execution (as performed by an intelligent agent sent to that site) returns tables containing data of interest. These queries are driven by an internal tree learner that uses them whenever a new node needs to be added to the tree.

The multi-agent design of INDUS and it's data-independent properties are incredibly interesting. However, as the authors provide no details on its actual performance, there is little to recommend it over the other algorithms proposed in literature.

6 Conclusions and Summary

The overarching goal of data inference activities is to discover interesting , yet previously unknown, patterns from massive caches of data. To this end, it is useful to know how to identify a piece of data. This is the practice of classification, to categorically sort data into a series of *classes*. Classification is heavily used in fields like financial analysis and medical diagnosis [18].

One popular type of classification is *decision tree induction*. This family of algorithms will take a body of data, find the point that gives the most *information gain* when the data is split there, and divide out into tree branches based on that split. This tree is finished when a child node contains only examples of a particular class label.

Classification is expected to be both fast and accurate, which is an increasingly difficult task as the size of the data grows. Much of the focus of data mining research over the past few years has been in how to deal with these massive data collections. Because of the large memory and computation requirements, parallelization of the data or the classification algorithm (or both) has become a viable approach.

Data can be parallelized in two fashions - horizontally or vertically. Horizontal fragmentation is where data entries are split equally among several sites. Vertical fragmentation is where columns of attributes are sent to different sites. Horizontal fragmentation is far more common, and is easier to program for.

Task parallelism is where the learning algorithm is parallelized. As it builds the tree, the algorithm spawns copies of itself to build the subbranches. Often, a hybrid approach is employed where the algorithm parallelizes itself over fragmented data and builds subtrees at each site.

This paper took a look at both major paradigms of parallel data mining. It elaborated on some of the algorithms being used, including SLIQ, SPRINT [13], INDUS [5], a parallelized C4.5 [19], and a SMP-based decision tree learner [11]. Many of these algorithms have reported promising results, and data mining on parallel-systems appears to be a booming research field.

References

1. S. Stolfo A. Prodromidis, P. Chan, *Meta-learning in distributed data mining systems: Issues and approaches*, in Advances of Distributed Data Mining, 2000.
2. V. Kumar A. Srivastrava, E. Han and V. Singh, *Parallel formulations of decision-tree classification algorithms*, Data Mining and Knowledge Discovery, (1999).
3. A. Thakar A.S. Szalay, P. Kunszt and J. Gray, *Designing and mining multiterabyte astronomy archives: The sloan digital sky survey*, Technical Report MS-TR-99-30, Microsoft Research, (1999).
4. A. Buja and Y. Lee, *Data mining criteria for tree-based regression and classification*, (2000).
5. A. Silvescu D. Caragea and V. Honavar, *Decision tree induction from distributed, heterogeneous, autonomous data sources*, in Conference on Intelligent Systems Design and Applications, 2003.
6. J.F. Naughton D. J. DeWitt and D.A. Schneigder, *Parallel sorting on a shared-nothing architecture using probabilistic splitting*, in Proceedings of the 1st International Conference on Parallel and Distributed Information Systems, 1991.
7. P. Domingos, *Knowledge acquisition from examples via multiple models*, in Proc. of the 14th ICML, 1997.
8. J. Darlington et. al., *Parallel induction algorithms for data mining*, in Proc. of the 2nd Intelligent Data Analysis Conference, 1997.
9. U. Fayyad, *Taming the giants and the monsters: mining large databases for nuggets of knowledge*, 1998.
10. A. A. Freitas and S.H. Lavington, *Mining very large databases with parallel processing*, 1998.
11. J. Saltz H. Andrade, T. Kurc and A. Sussman, *Decision tree construction for data mining on clusters of shared memory multiprocessors*, in HPDM: High Performance, Pervasive, and Data Stream Mining, 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM: PDS'03). In conjunction with Third International SIAM Conference on Data Mining, May 2003.
12. R. Ramakrishnan J. Gehrke, V. Ganti and W. Loh, *Boat - optimistic decision tree construction*, 1999.
13. M. Mehta J. Schafer, R. Agrawal, *Sprint: A scalable parallel classifier for data mining*, in VLDB'96, Proceedings of the 22nd International Conference on VLDB, September 1996.
14. R. Kufrin, *Decision trees on parallel processors*, Parallel Processing for Artificial Intelligence, 3 (1997), pp. 279–306.

15. Friedman R. Olshen L. Breiman, J and C. Stone, *Classification and regression trees*, 1984.
16. Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic, and Yue Jiang, *Implications of ceiling effects in defect predictors*, in PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering, New York, NY, USA, 2008, ACM, pp. 47–54.
17. C.T. Ho M.J. Zaki and R. Agrawal, *Parallel classification for data mining on shared-memory multiprocessors*, in IEEE International Conference on Data Engineering, March 1999, pp. 198–205.
18. S.K. Murthy, *Automatic construction of decision trees from data: A multi-disciplinary study*, Data Mining and Knowledge Discovery, 2 (1998), pp. 345–389.
19. J. Gama N. Amado and F. Silva, *Exploiting parallelism in decision tree induction*, in Parallel and Distributed Computing for Machine Learning. In conjunction with the 14th European Conference on Machine Learning (ECML'03) and the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'03), December 2003.
20. C.R. Palmer and C. Faloutsos, *Density biased sampling: An improved method for data mining and clustering*, in 2000 ACM SIGMOD International Conference on Management of Data, May 2000.
21. F. Provost and V. Kolluri, *A survey of methods for scaling up inductive algorithms*, Data Mining and Knowledge Discovery, (1999).
22. R. Quinlan, *Induction of decision trees*, Machine Learning, 1 (1986), pp. 81–106.
23. J. Reinoso-Castillo D. Caragea C. Andorf V. Honavar, A. Silvescu and D. Dobbs, *Ontology-driven information extraction and knowledge acquisition from heterogeneous, distributed biological data sources*, in Proceedings of the IJCAI - 2001 Workshop on Knowledge Discovery From Heterogeneous, Distributed, Autonomous, Dynamic Data and Knowledge Sources, 2001.
24. Dagger W. Davies, P. Edwards, *A new approach to combining multiple models learned from disjoin subsets*, in ML99, 1999.