Abstract

Over the recent past, organizations and other users have been capturing increasingly large amounts of data that they wish to analyze. This data is to be used for discovering concepts, patterns and to improve the process of decision making. To address this problem machine learning has focussed on developing automatic data analysis techniques. The runtime of these learning algorithms depends on the size of data (the number of data points and the number of features for each data point). As a result, the learning time can become very large for larger datasets. One of the defining challenges for the machine learning community is to scale-up the learning algorithms to handle large volumes of data. This work focuses on parallelizing the decision tree learning algorithm in an attempt to reduce the learning time. We are especially interested in performing such work on simple network of workstations. Our results show small but significant gain in process time, but suggest that further work must be done.

Acknowledgements

I am grateful to my advisor Dr. Rich Maclin for providing me an opportunity to work with him and the valuable guidance that he provided. I also thank Dr. Masha Sosonkina Driver and Dr. Robert McFarland for their co-operation.

I am indebted to my parents, brother, sister and friends, without their support nothing would have been possible.

Contents

List of Figures

List of Tables

1 Introduction

Machine learning is the field of computer science in which computers are programmed to learn and thereby improve automatically with experience. It addresses the question of how to build computer programs that improve their performance at some task through experience. A precise definition would be:

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Machine learning involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner.

To design a machine learning algorithm we must determine: the type of training experience; the learned target function and the representation of the learned target function. This set of design choices constrains the learning task in a number of ways. Based on different choices we can design different types of algorithms for the same learning task. Different hypothesis representations are appropriate for learning different kinds of target functions. For each of the hypothesis representations, the corresponding learning algorithm takes advantage of a different underlying structure to organize the search through the hypothesis space. Linear functions, logical descriptions, decision trees and artificial neural networks are some examples of hypothesis representation. Learning can be characterized by the search strategies used to search through the hypothesis space and by the underlying structure of the search spaces.

1.1 Concept Learning

The problem of inducing general functions from specific training examples is central to learning. *Concept learning* is acquiring the definition of a general category given a sample of positive and negative training examples of the category. It is the problem of automatically inferring the general definition of some concept, given examples labeled as members or non-members of the concept. Given a set of training examples of the target concept c, the problem faced by the learner is to hypothesize, or estimate c. Although the learning task is to determine a hypothesis h which is the best estimate of target concept c, the only information available about c is its value over the training examples. Learning algorithms can best guarantee that the output hypothesis fits the training data perfectly. The hypothesis cannot guarantee results on unseen data examples. The fundamental assumption of *inductive learning* is, the best hypothesis regarding unseen instances is the hypothesis that “best” fits the training data. The inductive learning hypothesis can be more precisely stated as,

Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over unobserved examples.

1.2 Motivation

Over the recent past, the volume of data used for learning purposes has increased enormously. Many organizations have begun to routinely capture huge volumes of data describing their products, customers and operations. Scientists and engineers in various fields have been capturing increasingly complex experimental data. Machine learning helps answer the question of how best to make use of this data and to discover useful concepts, knowledge and patterns that can be used for future decisions. Machine learning algorithms are being used in practical applications as analyzing medical outcomes [Kononenko et al., 1984] and credit-card fraud detection [Stolfo et al., 1997].

Machine learning until recently was focussed on development and the use of automatic data analysis techniques. Various learning models like decision trees, artificial neural networks and genetic algorithms are available to learn from data. But now there is a new area of focus and that is to scale-up these learning algorithms to huge amounts of data. Most learning algorithms are so computationally intensive that they cannot be used in practice for such large volumes of data. One of the defining challenges for the machine learning community today is to enable these learning algorithms to work efficiently and produce accurate results on large volumes of data.

Machine learning algorithms need to scale-up to very large datasets for several reasons, including increasing accuracy and discovering infrequent special cases. These tasks are infeasible for learning algorithms running on sequential machines. We need to look into other techniques for learning algorithms so that they can be converted into some kind of a parallel implementation. Several techniques exist for performing tasks in parallel, like shared memory and messages passing.

We are interested in combining both of these methods so that we have a learning algorithm that runs in parallel on a number of machines and thus effectively reduces the time required to examine the dataset.

1.3 Thesis Statement

We intend to exploit the inherent parallelism in the decision tree learning algorithm and hence develop an algorithm for building the tree in parallel. We choose to use the decision tree learning algorithm as it is a good learning algorithm and has been successfully applied to a broad range of tasks. Also we see an inherent parallelism in the construction of the tree that we can exploit. The runtime of most machine learning algorithms is greatly influenced by the number of data points examined. Our technique will try to improve the runtime of the decision tree learning algorithm by examining only a subset of data points at a host (machine) and do this process in parallel several machines. We plan to use a master-slave type of computing environment in which the master controls the tree building algorithm and the slave provides intermediate results.

Statement of thesis: To parallelize the decision tree learning for an environment and check that it runs efficiently and produces accurate results.

The thesis will attempt to answer the following questions:

- *Question 1:* Can the decision tree algorithm be parallelized by exploiting the natural parallelism involved in building a decision tree? Namely, can we build the nodes of a tree in parallel?

- *Question 2:* How effective will the resulting algorithm be for a network of workstations?

- *Question 3:* What is the best mechanism for passing information about a node to be processed?

1.4 What Follows

In the following chapters this thesis will cover the necessary background about decision trees and parallel processing, the techniques that we developed, the results of our experiments followed by some concluding remarks and answers to the questions that the thesis attempts to address. Chapter 2 covers topics related to decision tree learning: tree representation, the basic tree learning algorithm, ID3 [Quinlan, 1990], and a discussion about basic parallel processing. Chapter 3 discusses the various techniques that we used for parallelizing the tree building algorithm. The chapter also covers the assumptions and the environment on which our implementations are based. Chapter 4 describes the results of using our implementations. The chapter also covers comparison between various types of implementations. Chapter 5 discusses the conclusions for the thesis based on the results and presents directions for future research.

2 Background

2.1 Decision Tree Learning

Decision tree learning [Quinlan, 1993, Breiman et al., 1984] is one of the most widely used and practical methods for inductive inference [Stolfo et al., 1997]. It is a method for approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions. The function learned from the training examples is represented in the form of a tree. Learned trees can also be represented as sets of if-then rules to improve human readability. This learning method is one of the most popular inductive inference algorithms and has been successfully applied to a broad range of learning tasks including medical diagnoses and credit risks assessment of loan applicants [Kononenko et al., 1984].

2.1.1 Decision Trees

Each node in the tree specifies a test of some attribute of the instance; each branch descending from the node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node and then moving down the branch corresponding to the value of the attribute. This process is repeated for the subtree rooted at the new node until a leaf is encountered. Thus decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance.

Figure 1 illustrates a typical learned decision tree. Outlook, Humidity and Wind are the attributes of the data that form the internal nodes of the tree. The branches correspond to their respective values, (e.g, attribute *Outlook* can have value Sunny, Overcast or Rain and there is a branch for each of these attributes).

This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance
$< Outlook = Sunny, Temperature = Hot, Humidity = High, Wind = Strong >$
would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative example (i.e., the tree predicts $PlayTennis = -$). Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree corresponds to a disjunction of these conjunctions. For example, the decision tree shown in Figure 1 corresponds to the expression

$$\begin{aligned} & (Outlook = Sunny \wedge Humidity = Normal) \\ \vee\ & (Outlook = Overcast) \\ \vee\ & (Outlook = Rain \wedge Wind = Weak) \end{aligned}$$

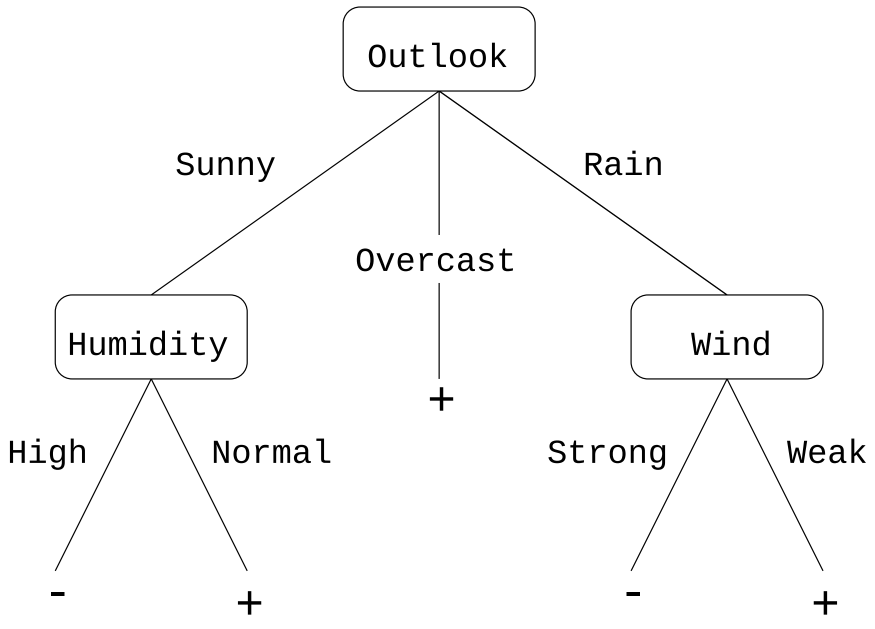


Figure 1: A decision Tree for the concept *PlayTennis*

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances, where this expression is true when the tree would classify an instance as positive(+). Each conjunctive clause of this disjunctive expression represents a path down the tree to a positive(+) leaf, (e.g., the conjunction $Outlook = Sunny \wedge Wind = Weak$ corresponds to the right most path of the tree, which has as leaf a positive (+) classification).

2.1.2 Building Decision Trees

Most tree learning algorithms use a top-down greedy search through a space of possible decision trees. The tree is empty initially and the algorithm starts building it from the root and adds internal nodes or leaf nodes as it goes down each branch of the tree. The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. The attribute selected must be most useful for classifying examples. The root node is the *best* attribute that singly classifies the entire training set. The best attribute is used as the test at the node. Each attribute value for the attribute of a node forms a branch. At each branch all the examples going down that

Table 1: The basic ID3 decision tree learning algorithm

```
ID3(Examples, Target_attribute, Attributes)
Examples are the training examples. Target_attribute is the attribute whose value is
to be predicted by the tree. Attributes is a list of other attributes that may be tested
by the learned decision tree. Returns a decision tree that correctly classifies the given
Examples.

• Create a Root node for the tree
• If all Examples are positive, Return the single-node tree Root, with label = +
• If all Examples are negative, Return the single-node tree Root, with label = −
• If Attributes is empty, Return the single-node tree Root, with label = most
  common value of Target_attribute in Examples
• Otherwise
    * $A \leftarrow$ the attribute from Attributes that best classifies Examples
    * The decision attribute for Root $\leftarrow A$
    * For each possible value $v_i$ of $A$,
        ∗ Add a new tree branch below Root corresponding to the test $A = v_i$
        ∗ Let $Examples_{vi}$ be the subset of Examples that have value $v_i$ for $A$
        ∗ if $Examples_{vi}$ is empty
            · Then below this new branch add a new leaf node with label =
              most common value of Target_attribute in Examples
            · Else below this new branch add the subtree
              ID3($Examples_{vi}$, Target_attribute, Attributes - $\{A\}$)
• Return Root
```

branch (i.e., examples based on their attribute values) are used to evaluate the best attribute for classification. The entire process is then repeated using the training examples associated with the descendant node to select the best attribute to test at that point in the tree. This is a *greedy* approach, as the algorithm never backtracks to reconsider previous decisions inorder to modify the learnt tree. A simplified version of the algorithm (ID3) [Quinlan, 1990], specialized to learning boolean-values functions (i.e., concept learning) is described in Table 1.

2.1.3 Which is the best attribute?

The central choice in building a tree is selecting which attribute to test at each node in the tree. The selected attribute must be most useful for classifying examples. The ID3 algorithm uses a statistical property called the *information gain* [Quinlan, 1990], that measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

To use information gain precisely a term called *entropy* is defined. *Entropy* characterizes the (im)purity of an arbitrary collection of examples. It is the expected number of bits required to encode an output class of a randomly drawn member of a collection of examples. Entropy is a measure of the expected amount of information conveyed by an as-yet-unseen message from a known set. The expected amount of information conveyed by any message is the sum over all possible messages, weighted by their probabilities. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this boolean classification is

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \tag{1}$$

where $p_{\oplus}$ is the proportion of positive examples in S and $p_{\ominus}$ is the proportion of negative examples in S. In all calculations involving entropy $0 \log_2 0$ is set equal to 0.

To illustrate, suppose S is a collection of 14 examples of some boolean concept that has 9 positive and 5 negative examples. Then the entropy of S relative to this boolean classification is

$$\begin{aligned} Entropy([9+, 5-]) &= (-9/14) \log_2(9/14) - (5/14) \log_2(5/14) \\ &= 0.940 \end{aligned} \tag{2}$$

The entropy is 0 if all members of S belong to the same class. For example, if all members are positive ($p_{\oplus} = 1$), then $p_{\ominus} = 0$, and

$$Entropy(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2 0 = -1 \cdot 0 - 0 = 0 \tag{3}$$

The entropy is 1 when the collection of examples in S contains an equal number of positive and negative examples. If the collection contains unequal number of positive

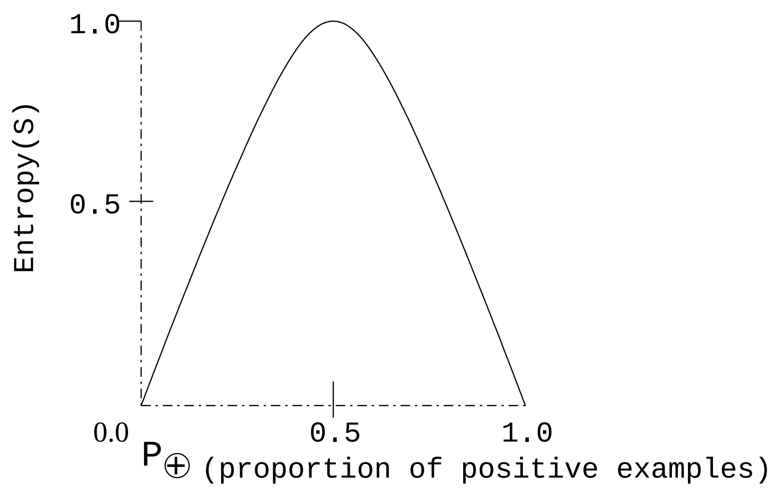


Figure 2: Entropy function relative to a boolean classification

and negative examples, the entropy is between 0 and 1. Figure 2 shows the graph of the entropy function relative to a boolean classification as $p_\oplus$ varies from 0 to 1.

Entropy can also be interpreted as the number of bits required to encode the classification of any arbitrary example in S. If the examples have more than two classes, then the formula for entropy is generalized to

$$Entropy(s) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \tag{4}$$

where we have c possible classes. The logarithm is still base 2 as entropy is the measure of the expected encoding in number of *bits.* According to information theory optimal length code assigns $-log_2 p$ to a message having probability p. Also if target classification can take c possible values, the entropy can be as large as $\log_2 c$.

Information gain is the expected reduction in entropy caused by partitioning examples according to a particular attribute. It is used to measure the effectiveness of an attribute in classifying the training data. The information gain, $Gain(S, A)$ of an attribute A, relative to a collection of examples S, is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{5}$$

where $Values(A)$ is the set of all possible values for attribute A, and S_v is the subset of S for which attribute A has value v (i.e., $S_v = \{s \in S | A(s) = v\}$). The first term in Equation 5 is the entropy of the original collection S, and the second term is the expected value of the entropy after S is partitioned using attribute A. The expected entropy is the sum of the entropies of each subset S_v, weighted by the fraction of examples that belong to S_v. $Gain(S, A)$ is the information provided about the *target value function*, given the value of attribute A. The value of $Gain(S, A)$ can also be expressed as the reduction in the number of bits required to encode the target value of an arbitrary member of S, by knowing the value of attribute A.

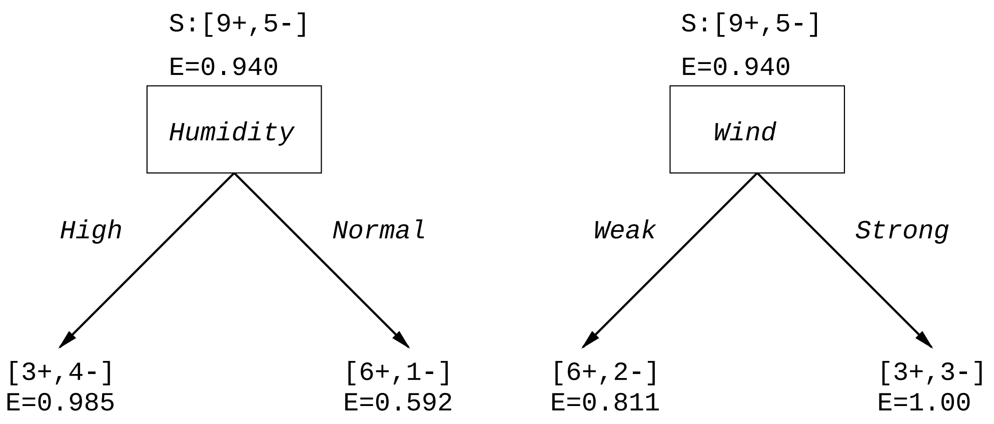
For example, suppose S is a collection of training examples with 9 positive and 5 negative examples. *Wind* is one of the attributes for the examples and takes attribute values *Weak* and *Strong*. Of the 14 examples suppose 6 of the positive and 2 of the negative examples have *Wind* = *Weak*, and the remainder have *Wind* = *Strong*. The information gain of the attribute *Wind* then is

$$\begin{aligned}
Values(Wind) &= Weak, Strong \\
S &= [9+, 5-] \\
S_{Wind=Weak} &\leftarrow [6+, 2-] \\
S_{Wind=Strong} &\leftarrow [3+, 3-] \\
Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - (8/14) Entropy(S_{Wind=Weak}) \\
&\quad -(6/14) Entropy(S_{Wind=Strong}) \\
&= 0.940 - (8/14)0.811 - (6/14)1.00 \\
&= 0.048
\end{aligned}$$

Attribute *Humidity* provides greater information gain than attribute *Wind*, relative to the target classification. E stands for entropy and s for collection of examples. The information gain of *Humidity* is 0.151 and that of *Wind* is 0.048.

Similarly, the information gain is calculated for all the attributes that are not tested in the current branch. In the greedy approach, the attribute with the best gain is the best attribute for classification. This attribute will be the attribute to be tested at that node. The same process is followed at each level of the tree for each branch. A node that has all examples with the same target class is a *leaf* node with the corresponding class as it's classification.

Figure 3 shows an example for comparing to different attributes on their information gain values. The information gain for attributes Humidity and Wind over the set of examples S is 0.151 and 0.048 respectively. From these values, the attribute Humidity is a better attribute than Wind.



$Gain(S, Humidity)$
$= .940 - (7/14).985 - (7/14).592$
$= 0.151$

$Gain(S, Wind)$
$= .940 - (8/14).811 - (6/14)1.00$
$= 0.048$

Figure 3: Which attribute is the best classifier?

2.1.4 ID3 Features

Some of the important features of the ID3 algorithm are:

- ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypotheses searched by ID3 is the set of possible decision trees. ID3 starts from an empty tree and progressively considers more elaborate hypotheses in search of a decision tree that correctly classifies the training data. ID3 performs a simple to complex, hill-climbing search through the hypothesis space. The idea behind hill-climbing is to find a solution then and move to a better solution in the next step, based on some reward function. In ID3 the reward function is the information gain.

- ID3's hypothesis space of all decision trees is a *complete* space of all functions, relative to the available attributes. Because every target function can be represented by some decision tree, ID3 avoids the risk of searching an incomplete hypothesis search space.

- ID3 maintains only a singly current hypothesis as it searches through the space of decision trees. This contrasts with other examples of learning methods which maintain the set of all hypotheses consistent with the available training examples. Due to this, ID3 loses the capability to look at alternate tree to resolve competing hypotheses. As the algorithm only considers the current *best* gain it misses horizon effects, where an optimal tree may be built by selecting an attribute that is not the best at some stage. ID3 performs no backtracking. Once an attribute is selected as an internal node the algorithm does not go back to change this decision. Therefore, it is susceptible to the usual risks of hill-climbing search without backtracking; converging to locally optimal solutions that are not globally optimal. These locally optimal decision trees may be less desired than trees that would have been encountered along a different branch of the search.

2.1.5 Inductive Bias in Decision Tree Learning

Inductive bias is the method by which the learner generalizes beyond the observed training data. It is the set of assumptions that together with the training data, justify the classifications assigned by the learner to future instances. Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 is to describe the basis by which it chooses one of the consistent hypotheses over the others. ID3 chooses the first consistent tree that it encounters in the greedy hill-climbing search. ID3 roughly speaking (a) selects in favor of shorter trees over longer ones, and (b) selects trees that place attributes with higher information gain values nearer to the root of the tree.

An approximate inductive bias of ID3 is : *Shorter trees are preferred over longer trees. Trees that place high information gain attributes near the root are preferred over those that do not.* The basis for this bias is that there are fewer short hypotheses than long ones and it is less likely that one will find a hypothesis that coincidentally fits the data.

2.1.6 Issues in Decision Tree Learning

Following are some of the issues and extensions related to ID3. ID3 has also been modified and extended to account for some of these issues and has been renamed $C4.5$ [Quinlan, 1993].

2.1.6.1 Overfitting the Data and Pruning

The ID3 algorithm grows each branch of the tree so that it classifies all the training examples. In cases where the number of training examples are too few or the training data contains noise, this may not be desirable. The tree may have few examples to fully detect the concept or the tree might be too complex and will fit the noise. In either of these cases the tree is said to *overfit* the data. A hypothesis is said to overfit the training examples if some other hypothesis that fits the training examples less well actually performs better over the entire distribution of instances (i.e., training and testing examples). Overfitting reduces the accuracy of the learned decision tree and also increases the size of the tree.
Methods for avoiding overfitting in decision tree learning [Mitchell, 1997] are:

- Stop growing the tree before it reaches as point where it overfits the data. It is difficult to estimate precisely when to stop growing the tree without having seen the whole tree or future data.

- Let the tree overfit the data and then use *pruning* techniques to reduce overfitting. Each decision node is considered for pruning. Pruning a node consists of removing the subtree rooted at that node, making a leaf node and assigning it the most common classification of the examples in that node. Some methods for pruning are :

  - Reduced Error Pruning [Quinlan, 1987a]
    The tree is grown till it fits the entire dataset. Each decision node in the tree is considered for pruning. Nodes are removed if the resulting tree performs better than the original. The algorithm starts from the lower level nodes and prunes the tree moving towards the root node.
  - Rule Post-Pruning [Quinlan, 1993]
    In this approach the decision tree is grown till it fits the entire dataset. The tree is converted to a set of *rules*, by created one rule for each path from root to leaf of tree. Each rule can be pruned by removing any preconditions that result in improving its estimated accuracy. The rules are sorted by estimated accuracy and considered in a sequence when classifying subsequent examples.

2.1.6.2 Handling Continuous-Values Attributes

The basic ID3 algorithm assumes the target attribute and the attributes tested in the decision tree to be discrete-valued. The first restriction is difficult to overcome as each leaf node holds a single classification. To overcome it the tree has to be constructed such that we have a leaf that covers a range of values or have a leaf node for each possible value, which is not practical. Also the best attribute during tree building is decided using the target classifications. If we use non-discrete target values we will have to complicate the process of using those values in the same manner as the discrete target classifications.

The second restriction is easier to overcome and has been implemented as part of the C4.5 algorithm. This is accomplished by dynamically creating a new discrete valued attribute that partitions the continuous valued attribute into a discrete set of intervals. In particular, for a continuous valued attribute A, the algorithm creates a threshold value c and partitions the data into two parts. All examples with a value of attribute A less than c are in one part of the partition and rest in the second partition. Effectively, that attribute can treated as a discrete attribute with values less than threshold value and greater than or equal to the threshold value. There are different ways in which the *threshold* value can be chosen to divide the data. We would like to pick a threshold, c, that produces the greatest information gain. By sorting the examples according to the continuous attribute and the identifying adjacent examples that differ in classification we can generate candidate thresholds. Also different implementations of the algorithm may split the attribute into more than two parts. We can have more than a single threshold value and have intervals between which values can lie (e.g., if we have two thresholds $t1$ and $t2$, we can handle values that lie in any of the following intervals, '$< t1$' or '$>= t1 \& < t2$' or '$>= t2$'. We can compare this situation, with a discrete-valued attribute which can handle three different values.) The dynamically created discrete-valued attribute then is evaluated as any other discrete attribute and it is used to calculate the information gain.

For example, consider a *Temperature* attribute that has continuous values and the values of the attribute Temperature at a certain node and the corresponding classifications for target *PlayTennis* are as following.

Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

It can be shown that the value of c that maximizes information gain must lie at such a boundary [Fayyad, 1991]. In the example shown, there are two candidate thresholds, corresponding to the values of *Temperature* at which value of *PlayTennis* changes: $(48+60)/2$, and $(80+90)/2$. Information gain can then be calculated for each of the candidate attributes, $Temperature_{>54}$ and $Temperature_{>85}$.

2.1.6.3 Handling Examples with Missing Values

In many cases the data available may not have attribute values for all attributes for all examples. The basic algorithm is extended so that these *unknown* values are replaced by some estimate. This estimate is based on other examples for which the attribute has a known value. Some of the methods used are:

- Assign to the missing value the most common value among the training examples at the particular node under consideration.
- Assign the most common value among training examples at the node, that have the same classification as that of the example that has the unknown value.
- Use a more complex procedure that assigns a probability to each of the possible values of the attribute rather than a single value [Quinlan, 1993]. This probability is calculated using the frequencies of the known examples at the node. A similar approach can be used to classify unseen examples that have unknown values. In this case the classification of the new instance the most probable classification, is determined by adding the weights of the examples at each leaf node.

2.2 Parallel and Distributed Processing

2.2.1 Parallelism

Running programs simultaneously to decrease runtime and to efficiently use available resources is one of the important objectives of parallel computing. The programs that are executed in parallel could be the same program running on more than one processor, or multiple programs running on multiple processors or multiple programs running on the same processor. A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem. This definition is broad enough to include parallel supercomputers that have hundreds or thousands of processors, networks of workstations, multiple-processor workstations, and embedded systems [Foster, 1995]. Parallel computing denotes a computing environment in which some number of processors are running asynchronously but communicating with each other to avoid conflicts or to exchange intermediate answers.

2.2.2 Parallel Programming Models

A broad classification of parallel programming models [Foster, 1995] is presented below:

2.2.2.1 Message Passing

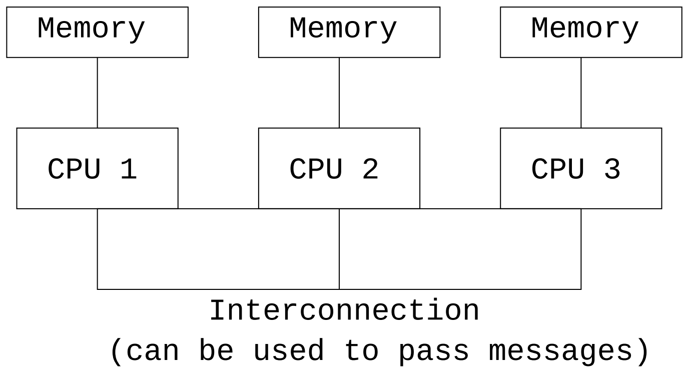


Figure 4: Message Passing System

Message passing is probably the most widely used parallel programming model today. In this scenario a particular task is broken down into multiple tasks and these tasks communicated by passing messages (e.g., addition of arrays on different machines and the results can be collected as messages). Most message passing systems use the *Single Program Multiple Data* model as each task executes the same program but on different data. Task-specific code can also be added to the program, so that each task only executes it's own code. Messaging systems do not discourage the creation of new processes at runtime or execution of different programs on different processors, but most current systems setup the number of processors and the program before execution starts. In most cases this is sufficient for parallel algorithms.

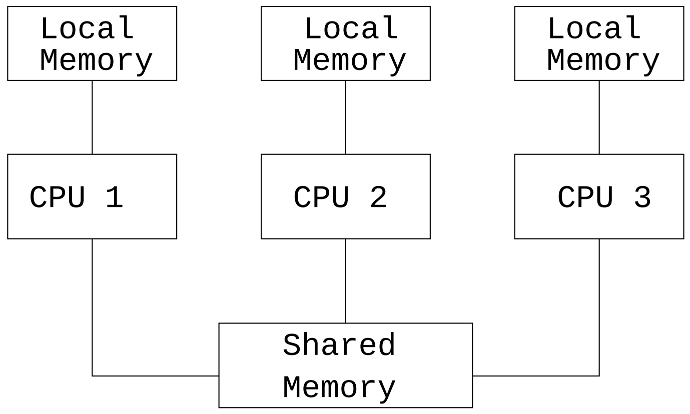


Figure 5: Shared Memory Processing System

2.2.2.2 Shared Memory

In the shared-memory model tasks share a common address space where all tasks can read and write asynchronously. Various mechanisms like *locks* and *semaphores* are used to control access to shared data. As a common address space is present data transfer to different tasks is not an issue, but writing deterministic programs may become difficult.

2.2.2.3 Data Parallelism

This model exploits the concurrency that is present due to applying the same operations on a set of data (e.g., add 5 to all elements of the array [Shafer et al., 1996]). If each operation to be performed on the data is independent, the data can be distributed over many processors and the each works on the available data independently. Issues in this type of system are how to distribute to data over different processors and issues relating to transfer of the data.

2.3 Scaling Up Inductive Learning Algorithms

As datasets grow in size, the inductive learning algorithms have to be enabled (scaled-up) to handle these large datasets. One of the reasons cited for using large datasets is that often accuracy of learned classifiers increases with increase in training data [Catlett, 1991]. More data can help in discovering more interesting concepts. Having a large number of examples introduces potential problems with both time and space complexity. To scale-up an algorithm so that it processes large datasets efficiently and produces accurate results, many different techniques have been proposed and implemented. Some of the approaches that have been followed are [Provost and Kolluri, 1999]:

1. *Design a fast algorithm*
   The most straight forward approach to scaling-up inductive learning is to produce efficient algorithms. This approach includes a wide variety of algorithm design techniques for reducing complexity, optimizing search and representation, using approximate intermediate results and taking advantage of task's inherent parallelism. Some techniques used are:

   (a) *Restricted Model Space*
   In this approach the algorithm uses a restricted hypothesis space, thus reducing the total number of hypotheses that can fit the data. Examples of the these algorithms are ones used to build two-level decision trees and decision stumps [Holte, 1993, Iba and Langley, 1992]. These classifiers have shown high accuracy in many datasets.

   (b) *Powerful Search Heuristics*
   In cases where a restricted model space does not work, using powerful search heuristics can help scale-up the learning algorithm. ID3 uses a greedy hill-climbing approach to build the tree and produces highly accurate results. Other approaches would be to avoid growing the tree completely before pruning [Quinlan, 1987b] and incremental reduced error pruning [Furukranz and Widmer, 1994].

   (c) *Algorithm/Programming Optimizations*
   Learning algorithms can be optimized using efficient data structures (e.g., bit vectors, hash tables, binary search trees) combined with good programming techniques. Some other techniques like bookkeeping and presorting can also be used to reduce the time required to process data [Almuallim et al., 1995, Aronis and Provost, 1997].

   (d) *Parallelism*
   Learning algorithms can also be parallelized in order to scale-up. The space can be searched in parallel [Cook and Holder, 1990] and as each processor works with a restricted hypothesis space the algorithm can be made more efficient. The task of building nodes at different branches can also be spread across more than one processor, with a master processor coordinating the process of accumulating results and building the

tree [Kumar and Rao, 1987]. But parallelism also has issues like overhead of communication between nodes, availability of special parallel hardware and synchronizing between hosts.

2. *Partition the data*

   The runtime of learning algorithms increases as the size of data increases. For large datasets the data can be partitioned and processed in parallel to improve runtime. The data can be partitioned in many ways including: *instance* [Catlett, 1991] and *feature* sampling [Musick et al., 1993]. In instance sampling the entire dataset is broken down into smaller subsets to be processed independently. As each subset can be processed in parallel, the process aims to speed-up the learning algorithm. Another approach to splitting the data is based on the features or attributes of the examples. Different processors work in parallel and on different subsets of features. The runtime of algorithms grows with the increase in features, as this increases the attributes that need to be processed at each node in the tree. The above approach tries to reduce this using a smaller set of features at each processor. In both of the above methods for partitioning data, the parallel learners need to coordinate with each other to produce the final classifier. Different approaches exist for interaction between parallel learning algorithms. Learners can learn their own classifier and then combine them or a learner can give feedback about its classifier to other learners and let them use his results. A generic parallel learning algorithm is shown in Figure 6.

3. *Use a relational representation for data*
   As data files get bigger, storing them as flat files is not the best possible approach. The complexity of storing and manipulating data could be delegated to another level. Data can be stored in distributed databases [Ribeiro et al., 1996] or relational databases [Agarwal and Shim, 1996]. for efficient access and operations on data. Access to relational databases through *SQL* queries, computations inside the DBMS and utilizing parallel database engines can give efficient data access, which can lead to faster learning algorithms.

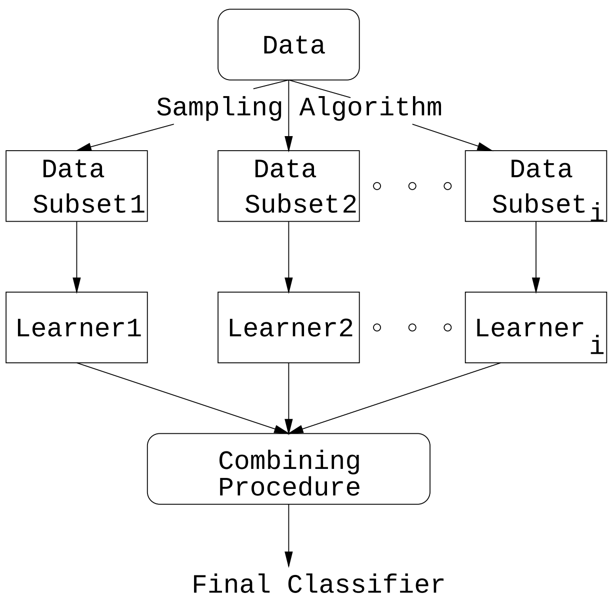


Figure 6: An ensemble learner based on partitioning the data.

3 Method

The goal of scaling-up an inductive algorithm, is to process large amounts of data efficiently and to produce accurate results. In this section we present the techniques that we implemented to scale-up the decision tree building algorithm.

3.1 Assumptions and Environment

Before we start discussing the actual techniques, we will discuss the type of hardware and software used for the implementation as well as other underlying assumptions.

3.1.1 Intent

The decision tree is built by examining the data points at a node and selecting the best attribute for partitioning the data and continuing till each branch terminates in a *leaf*. We will parallelize the process by evaluating branches of a node in parallel. Each branch of a particular node has to be evaluated over a set of attributes and examples to determine the next best attribute or whether the branch terminates in a leaf. We parallelize this process by evaluating the branches of a particular node in parallel. For this we need a parallel processing environment.

3.1.2 Network of Workstations

We have not used any special parallel hardware, such as a shared memory processing system or a parallel machine with more than one processor, for our tests. We have tried to use commonly available workstations on a network working in parallel. The workstations on the network act as hosts that can be used for different processing tasks in a parallel manner. We are not making any assumptions about the traffic on the network and hope to get better results than the serial algorithm when the network has a relatively light load.

3.1.3 Accessing Data

The data (i.e., the set of examples to be used for learning) is assumed to be present at each node for processing. Before performing any computations each host reads in data from the data files into local memory as part of the learning algorithm. By making this assumption we are avoiding the issues involved in transferring data from one host to another for every computation. In an alternative case, the data points at each stage of the tree would have to be determined by the master, and that subset of data passed to the concerned host. The above assumption avoids passing data examples between hosts. Instead each host, based on certain parameters (e.g., list of attributes and corresponding values), constructs the dataset that it must use for its current computation. We look at two different ways of passing information: vectors as messages and using files.

3.1.4 Message Passing Interface

The Message Passing Interface [Snir et al., 1996, Pacheo, 1996, Gropp et al., 1999] (MPI) is a set of core library routines which are used to write portable message passing programs. Message passing is a parallel programming paradigm that can be used effectively on parallel computers and network of workstations (NOWs). MPI provides source-code level portability, that is the same code can be executed on a variety of machines as long as the MPI library is available. MPI also has the ability to run transparently on heterogeneous systems (i.e., a collection of processors with distinct architectures).

MPI is used to pass messages among participating hosts (machines). The messages that are exchanged between machines have information regarding what portion of the dataset to use. The hosts also transmit messages for results after computation of a certain portion of the dataset. MPI gives a relatively simple abstraction of how machines in a network can be used in parallel. All the machines that participate in the running program have an assigned *rank*, which is an integer. Hosts are distinguished using these ranks and the ranks are used as identifiers to pass messages among different hosts. This is a simple abstraction for communication between hosts, but other techniques (e.g., low level sockets) exist for communication between different machines. These alternative methods may perform better, but we choose to use a relatively simple model and concentrate instead on experimenting with different learning algorithms.

3.2 Basic Setup

The different variations of the learning algorithms have a common setup. Each implementation is based on the setup shown in Figure 7.

For all the implementations there exists a machine (host) that acts as the master node and in terms of the MPI ranking scheme, is the *Rank 0* machine. All the other machines in the pool for a particular execution of the program act as slaves. They receive specific *messages* from the master node, when they are required to carry out a request and also get necessary information for executing that request. Based on this

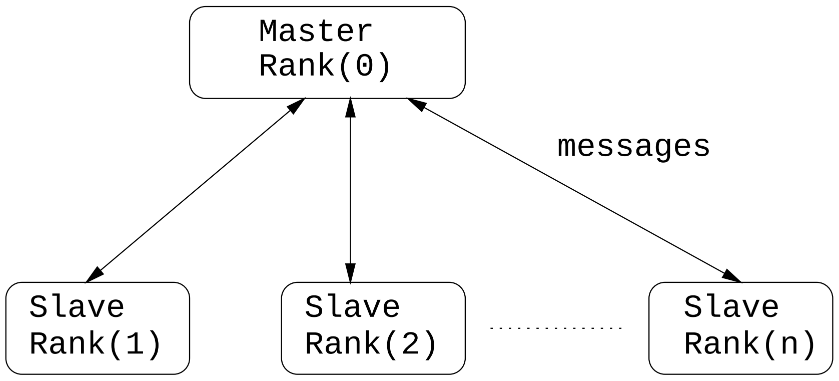


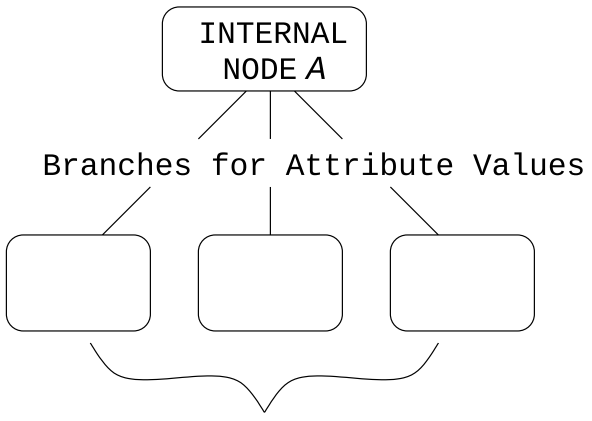
Figure 7: The master-slave scenario for building the tree in parallel. The master host controls the algorithm and builds the tree based on results obtained from the slaves.

information that the hosts receive, they perform certain computations and send the results in the form of messages back to the master.

3.2.1 Main Idea

In the tree building algorithm, the time spent in finding out the *best* node at different stages of the tree is the most important factor in determining the runtime. As we saw earlier, to determine the best attribute for a node or to make it a leaf node requires that a set of attributes and all the data points at that node in the tree be considered. For each node under consideration we have to find out its *information gain* and then compare values of all attributes to determine the best attribute. The node may be a *leaf* when all examples have the same classification or all attributes have been already tested in a branch. There can be other conditions for determining leaf nodes, for example, the ratio of data points in each class can be used to find out if a split is required or a minimum number of data points can be set to create an internal node.

Other than cases when a node is a leaf, the operation of computing the information gain is performed on all the unused attributes in the branch under consideration. Calculating the best attribute, involves traversing the set of examples (data points) at that node for each remaining attribute. For a node that is internal to the tree, each value that it can take forms a branch. We try to change the serial algorithm to parallel by processing each node down a different branch in parallel. So instead of finding the attribute/leaf for all the nodes down the branches of a particular parent attribute in turn, we try to do all branches at the same time.



Each node at a branch is independent and can be built in parallel.

Figure 8: The idea behind parallelizing the tree building process is to evaluate branches of a node in parallel.

As shown in Figure 8 if we have an internal node A and it can take three values, then it has three branches corresponding to these values. The examples that correspond to each branch are used in determining the best attribute of an internal node or leaf node at that point of the tree. As the same process has to be carried out (finding best attribute or leaf) for each branch, we can do this process in parallel. The set of examples for each branch are independent, as they are sorted based on the attribute values. Building nodes in different branches is an independent process for each node (i.e., the results of a node need not be passed to another node down a different branch).

3.3 The Basic Message Passing Parallel Algorithm

By constructing the nodes in parallel we are trying to reduce the time required for multiple passes over the data. To achieve this parallelism each host needs to know what set of data it has to process in order to come up with a best attribute for an internal node or a leaf node. This information is passed to the *slave* hosts by the *master*. The master knows the current state of the tree and based on which branch a particular host is supposed to look at, it sends the necessary information to the corresponding host. There are two ways that we have tried to pass this information between the master and the slaves. One is directly passing information vectors and other is a file based technique. Both techniques are discussed in the following sections.

The basic algorithm for all these variations is as shown in Table 2. The basic difference between the implementations is the manner in which the information is transmitted to the hosts.

Table 2: The basic master-slave algorithm for building decision trees in parallel.

- At program start-up the user specifies the number of processors and which hosts should run the program. The machines on which the program should run in parallel can be specified in a file that is read by default by all MPI programs.
- Starting at the first slave, the algorithm creates a *Root* node for the tree, considering the complete training set for this purpose.
- If *Root* node is a *leaf* stop algorithm.
- Else
  - For Each Branch (i.e., each distinct value of attribute in internal node)
    - The master locates a host to process the current branch.
    - The master sends necessary information to the slave host (regarding which data points to consider for computation of information gain).
    - The slaves use the information from master to create the subset of data to be considered.
    - The slaves then use this subset to determine best attribute or that the node is a leaf.
  - The master waits for replies from the slaves (the replies will contain information about nodes at each branch).
  - The master uses replies from slave hosts to update the decision tree.
  - The algorithm is called to create a node for the examples corresponding to each branch.
  - Master stops requesting slaves for evaluating branches when all internal nodes are evaluated.

3.3.1 The First Combined Message Passing Method

In this variation we are trying to use the parallel method for high level nodes, that is nodes that are nearest to the root. In most cases the nodes near the root are ones that have the majority of the data. The data splits into branches as we move down the tree. The number of examples associated with a node that is nearer to a the root is generally much greater than that of a node farther away from the root. When we send messages and try to parallelize the computations, sending and receiving messages adds to the runtime of the algorithm. The nodes that are farther away from the root or have a smaller number of associated examples may not be worth this effort. The time required to send and receive messages and get the computation done at a different host may require more time than using a serial algorithm. We will try to verify this assumption using our implementations. Based on the assumption, this method modifies the basic algorithm shown in Table 2 by placing a constraint on the number of examples required at a node to evaluate its branches in parallel. The basic parallel algorithm is used if the number of examples that are in a node is greater than a *threshold* value. The value of threshold on the number of examples can be set while starting up the program by the user. If a node has a smaller number of examples than the threshold, then the algorithm executes as a normal serial version on the tree algorithm.

3.4 The File Method

In the previous two techniques messages that are sent from the master to slaves are in the form of a vector. The vectors indicate which data points to consider and what attribute values are present in the current path. These vectors are array structures that are passed and values are set to identify the attributes and their values in the current path of the tree.

Main Idea: The implementation is based on a network of workstations which share a file system. The file technique tries to make use of this fact and share files for communication among all the machines in the current pool.

In this variation, instead of passing vectors of information regarding the current path and then building the subset of data that is associated with the branch we send a filename. The filename corresponds to a physical file that holds flagged values for each training example and is called the *datamap*. If an example is to be considered its entry in the datamap is set to a value greater than zero. This value is based on other attribute values which bring the example to the current branch. If an example is not be considered its corresponding value in the file is set to zero. Because the data at child nodes is from the parent nodes, *datamap* files for child nodes can be created from the parent datamap entries. As we can reuse parent datamaps, we can create the new datamap files for the new nodes relatively fast. The slaves that read these datamap files look up the flag for each example and determine whether they need to use it. This saves the time required to go through the list of attributes and their values to create the set of data points for computation. The set can be created using the datamap by grouping all non-zero flagged examples together.

An example of a datamap for a parent node and its corresponding child is as shown in Table 3. The parent (internal) node is the attribute *Temperature* and the values it can take are *Hot, Mild* and *Cool.* Each of these values correspond to a data map file. As the example shows the datamap for the child nodes can be formed by looking at

Table 3: Example datamap file entries for an internal node (Temperature) and its child nodes which correspond to the values it can take (Hot, Mild and Cool).

Example Number	Temperature	Hot	Mild	Cool
1	1	0	0	1
2	1	1	0	0
3	0	0	0	0
4	1	0	0	1
5	1	0	1	0
6	0	0	0	0
7	1	0	1	0
8	1	0	0	1
9	0	0	0	0
10	0	0	0	0
11	1	1	0	0
12	1	0	0	1
13	0	0	0	0
14	1	1	0	0

those data examples that are flagged non-zero in the datamap for the parent. The reuse of datamap helps in decreasing the runtime as we are not creating the subsets of data from scratch everytime based on the path the current node is located.

3.5 The Second Combined File Method

As in the first combined technique, the files method can also be changed to take into account the number of examples at each node. The datamap files for a node and its corresponding attribute values are created only if the number of examples in a node is greater than a *threshold* value. If the number of examples is less than the threshold, the serial learning algorithm is executed. We are trying to avoid creating files which have only a few flagged (non-zero) values. The cost for creating these files and reading them in for getting the set of current data points may be substantially more than the normal serial algorithm. This threshold value can be controlled by the user when he executes the program. This technique is expected to perform better than the previous all parallel files method.

3.6 Parameters

Parameters that can be varied to influence how the program executes are:

- Number of processors on which the program can execute
- The type(s) of machines program can execute on
- The pruning depth of the tree

4 Experiments

In this section we present the experiments we performed on the different implementations of the tree building algorithm. All the runs are on a network of workstations and no special parallel hardware is used for running the programs. The network used is public access and has varying degrees of traffic. The values obtained will be used to compare how the different implementations perform relative to each other.

4.1 Datasets used for experiments

Table 4 shows a list of datasets that we used for our experiments. The datasets used are part of the UCI Machine Learning Repository [Blake and Merz, 1998]. These datasets have been contributed by the machine learning research community and are widely used for testing. We use data sets to compare results with other experiments using the same datasets. The datasets are also related to real-world problems and have a wide variety of features.

Each dataset is characterized by the number of attributes, the number of possible output classes and the number of examples present in the dataset. Each attribute is a *discrete* or *continuous* attribute. Values of discrete attributes are from a set of values that are defined for that attribute (e.g., a discrete attribute *Temperature* might have values *Hot, Mild* or *Cold*). Continuous valued attributes hold real numbers as values (e.g., Continuous attribute *Wage* can take values like $1000, 2000.78, 909.63$). In some cases the attribute value of an attribute for a data point may be *unknown.*

4.2 Program Execution

A set of parameters can be varied to change the manner in which the learning algorithm is executed. The parameters that can control the algorithm are:

- *Number of workstations*
  As we run our experiments on a network of workstations, the number of processes used in the parallel algorithm will affect the runtime of the program. By changing the number of processes running on different workstations we can study the relation between number of processes and the speed of execution.

- *Threshold to control parallel/serial execution* As mentioned in sections 3.3.1 and 3.5 we can set a *threshold* for the number of examples for evaluating nodes in parallel. By varying this threshold value we expect the algorithm to produce

Table 4: The datasets used in my experiments. Each dataset is characterized by the number of attributes, the number of possible output classes and the number of examples present in the dataset.

Name of Dataset	Number of Attributes		Number of Output Classes	Number of Examples
	Discrete	Continuous		
hypo	22	7	5	3772
ionosphere	-	34	2	351
segmentation	-	19	7	2310
sick	22	7	2	3772
sonar	-	60	2	208
splice	60	-	3	3190
vehicle	-	18	4	846
letter-recognition	-	16	26	20000

different runtime values. The value of threshold can be used to check till what point the tree can be built effectively in parallel.

Results are averaged over a set of 10 tests and represent the time taken to build one tree. The time taken over the 10 tests is divided by 10 to obtain the average.

4.3 Results

Tables 5, 6, 7 and 8 show the results over some datasets and the various parameters that control the execution. The results shown are for the *Basic Message Passing, First Combined Message Passing, Basic File* and *Second Combined File* methods. The results shown are performed using 3 and 5 processors. Based on these values we can compare the results of the serial algorithm and the different parallel techniques.

Table 5: Results of the Serial, Basic Message Passing and First Combined Message Passing with threshold methods on some datasets using 3 processors.The second column shows the runtime using the serial algorithm. The third column shows the results using the Basic Message Passing method. The fourth column is the threshold value for the Combined Message Passing method and the corresponding runtime is displayed in the fifth column.

(Message Passing methods) Number of processors = 3				
Dataset	Serial	No Threshold	Threshold	Runtime(sec)
hypo	153.3	159.6	50	138.5
			100	133.4
			200	128.5
ionosphere	15.5	11.5	50	12.8
			75	11.4
			100	11.0
segmentation	162.1	164.4	50	147.5
			100	147.1
			200	146.3
sick	204.7	231.7	100	194.3
			200	194.2
			300	193.6
sonar	7.5	7.5	25	7.4
			45	7.3
			75	7.2
splice	6.4	7.3	100	5.48
			150	5.06
			200	3.56
vehicle	23.2	27.9	50	19.2
			100	18.6
			150	18.2

Table 6: Results of the Serial, Basic Message Passing and First Combined Message Passing with threshold methods on some datasets using 5 processors.The second column shows the runtime using the serial algorithm. The third column shows the results using the Basic Message Passing method. The fourth column is the threshold value for the Combined Message Passing method and the corresponding runtime is displayed in the fifth column.

(Message Passing methods) Number of processors = 5				
Dataset	Serial	No Threshold	Threshold	Runtime(sec)
hypo	153.3	159.6	50	127.7
			100	127.1
			200	125.0
ionosphere	15.5	11.5	50	11.3
			75	10.7
			100	10.5
segmentation	162.1	163.7	50	145.1
			100	144.6
			200	142.6
sick	204.7	231.7	100	162.2
			200	161.1
			300	158.4
sonar	7.5	6.2	25	5.9
			45	6.0
			75	6.6
splice	6.4	7.3	100	3.14
			150	3.01
			200	2.89
vehicle	23.2	27.9	50	17.9
			100	17.8
			150	17.5

Table 7: Results of the Serial, Basic File and Second Combined File with threshold methods on some datasets using 3 processors.The second column shows the runtime using the serial algorithm. The third column shows the results using the Basic File method. The fourth column is the threshold value for the Combined File method and the corresponding runtime is displayed in the fifth column.

(File methods) Number of processors = 3				
Dataset	Serial	No Threshold	Threshold	Runtime(sec)
hypo	153.3	137.8	50	126.8
			100	123.4
			200	120
ionosphere	15.5	14.1	50	12.8
			75	12.2
			100	12.0
segmentation	162.1	163.7	50	150
			100	149.5
			200	147.8
sick	204.7	193.8	100	181.1
			200	181.6
			300	182.1
sonar	7.5	7.8	25	8.6
			45	8.3
			75	8.2
splice	6.4	10.8	100	9.5
			150	8.2
			200	7.8
vehicle	23.2	25.4	50	25.1
			100	24.6
			150	23.8

Table 8: Results of the Serial, Basic File and Second Combined File with threshold methods on some datasets using 5 processors.The second column shows the runtime using the serial algorithm. The third column shows the results using the Basic File method. The fourth column is the threshold value for the Combined File method and the corresponding runtime is displayed in the fifth column.

(File Methods) Number of processors = 5				
Dataset	Serial	No Threshold	Threshold	Runtime(sec)
hypo	153.3	125.3	50	124.8
			100	124
			200	122.5
ionosphere	15.5	13.4	50	13.2
			75	13.1
			100	12.8
segmentation	162.05	159.2	50	156.4
			100	153.7
			200	151.2
sick	204.7	190.2	100	180.2
			200	180.9
			300	181.4
sonar	7.5	7.1	25	7.0
			45	6.7
			75	7.1
splice	6.35	9.0	100	7.8
			150	6.9
			200	6.5
vehicle	23.18	24.8	50	24.4
			100	24.1
			150	23.5

4.4 Critical Path

The tree building algorithm starts with an empty tree and adds nodes and leaves as the depth of the tree increases. Only when a node is created can we decide the examples associated with its branches and work on the child nodes. This aspect is an important part of the runtime for the algorithm. As we are building nodes in parallel, we can build only those nodes whose parents are already determined. Consider an internal node *A* that has two branches *B1* and *B2* and both are executed in different hosts. It might so happen that *B1* gets evaluated first and *B2* takes more time. If the scheduling algorithm schedules the child of the internal node at branch B2 on the same machine as *B1*, the computation has to wait till *B1* finishes. So even though the algorithm tries to schedule the process in parallel, there are limitations based on the time taken by the previous request executing on the host.

Figure 9 shows the execution of the Basic Message Passing algorithm on the dataset *hypo* using 3 processors. Processor 1 is scheduled to decide the *Root* node, till this node is computed we cannot schedule other requests. Once the *Root* is decided, processor 2 works on its first branch while processor 1 works on the second branch. Looking at the time sequence, evaluating examples associated with branch 2 gets done before branch 1. According to the way we schedule hosts the next host to get a request is processor 2. Even though branch 2 is evaluated sooner, we cannot schedule a request on processor 2 as it is not finished evaluated the previous request. This kind of a scenario is an important factor in deciding the runtime of the algorithm and how much speed-up we can obtain using our methods.

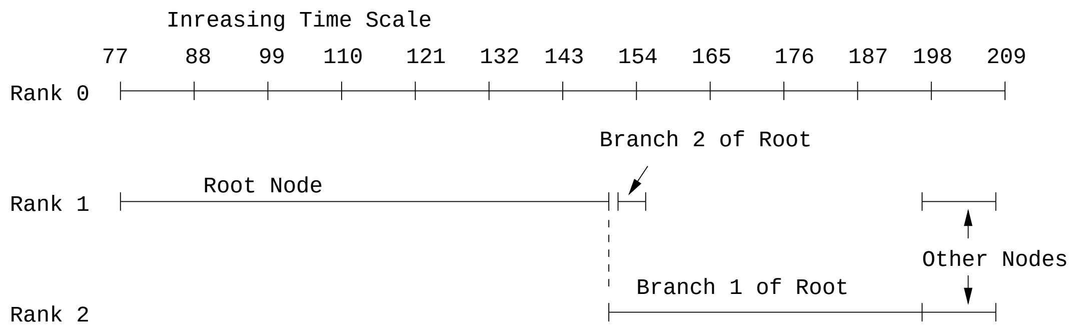


Figure 9: Experiment using dataset *hypo* on 3 processors. Shown is the execution time on the 2 processors in an increasing time sequence. The later nodes in the tree cannot be evaluated before their parent nodes.

We can try to improve runtime to some extent in such situations, by having a better scheduling mechanism which keeps track of all idle hosts, and can schedule requests even if another branch of its parent is not finished.

4.5 Comparison Charts

Figures 10, 11, 12, 13, 14 and 15 show a set graphs for the comparison of runtime values of different datasets. Using these graphs we can compare the runtime values of the different methods that we have implemented. Each chart displays the runtime along the y-axis and the *threshold* on the number of examples for the Combined techniques on the x-axis. The serial and the basic parallel techniques do not depend on the *threshold* value and are shown as a straight line on the graphs. The runtime values of the combined techniques vary with the *threshold* and in most cases show some kind of a gradient. Results of executions on a dataset using both the Message passing and the File techniques are shown.

4.6 Observations

Some observations based on the results:

- The basic parallel algorithm generally performs worse than the serial algorithm and only in a few cases slightly better than the serial algorithm. Most of the datasets that we tested had the same trend in both the message passing and the file techniques. There was no speedup derived from the basic parallel technique.
- The Combined message passing and file techniques perform better than the basic parallel techniques. In almost all cases the combined techniques have runtime less than the complete parallel techniques.
- As compared to the serial algorithm, the combined techniques show some definite gains in runtime. But there are a few cases in the combined file method that have runtime larger than the serial algorithm.
- As the number of processors increase, in the results shown 0, 3 and 5, the runtime over the datasets decreases. This trend is consistent over both the simple message passing and the file techniques.
- The two Combined techniques have nearly the same runtime. The Combined Message Passing method has better runtime values for most cases.

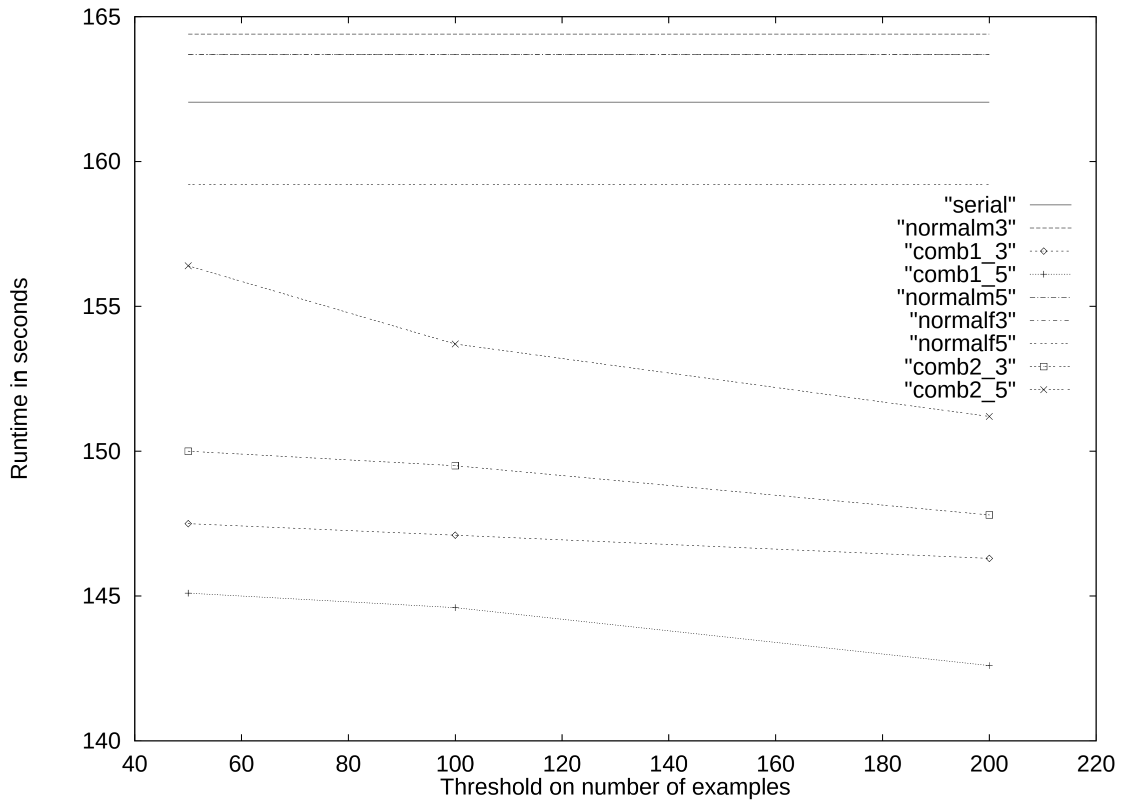


Figure 10: Comparison of runtime for dataset segmentation using the Message and File techniques on 3 and 5 processors. The words in the legend of the graph correspond to different runtime values and the names stand for different method which are: *serial* - the serial algorithm, *normalm3* - the basic message passing method on 3 processors, *normalm5* - the basic message passing method on 5 processors, *normalf3* - the basic File method on 3 processors, *normalf5* - the basic File method on 5 processors, *comb1_3* - the First Combined Message method on 3 processors, *comb1_3* - the First Combined Message method on 5 processors, *comb2_3* - the Second Combined Message method on 3 processors, *comb2_5* - the Second Combined Message method on 5 processors.

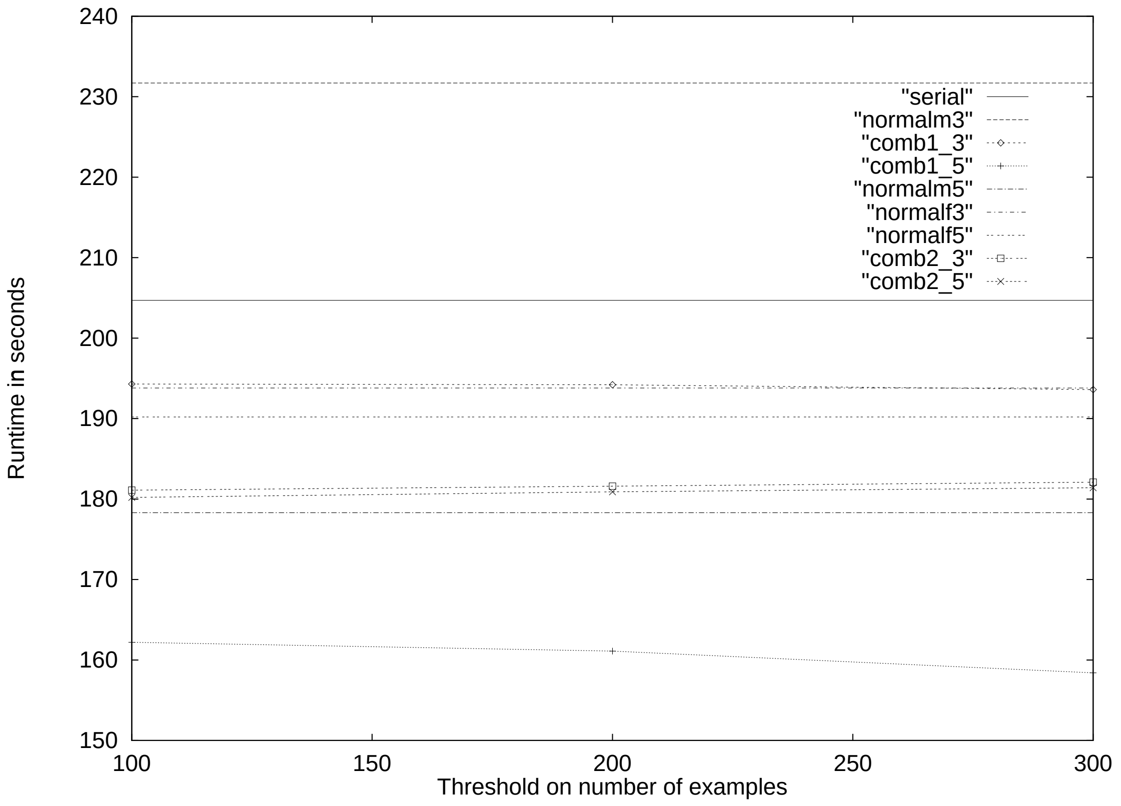


Figure 11: Comparison of runtime for dataset sick using the Message and File techniques on 3 and 5 processors. Refer to Figure 10 for legend description.

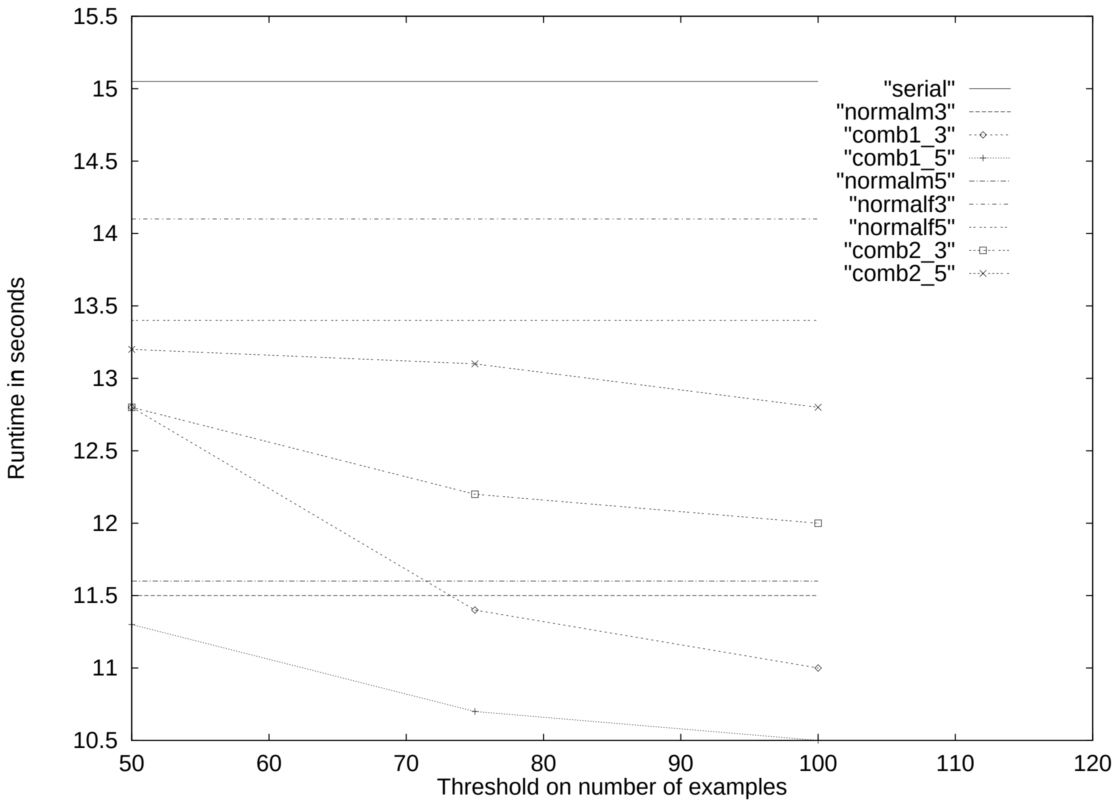


Figure 12: Comparison of runtime for dataset ionosphere using the Message and File techniques on 3 and 5 processors. Refer to Figure 10 for legend description.

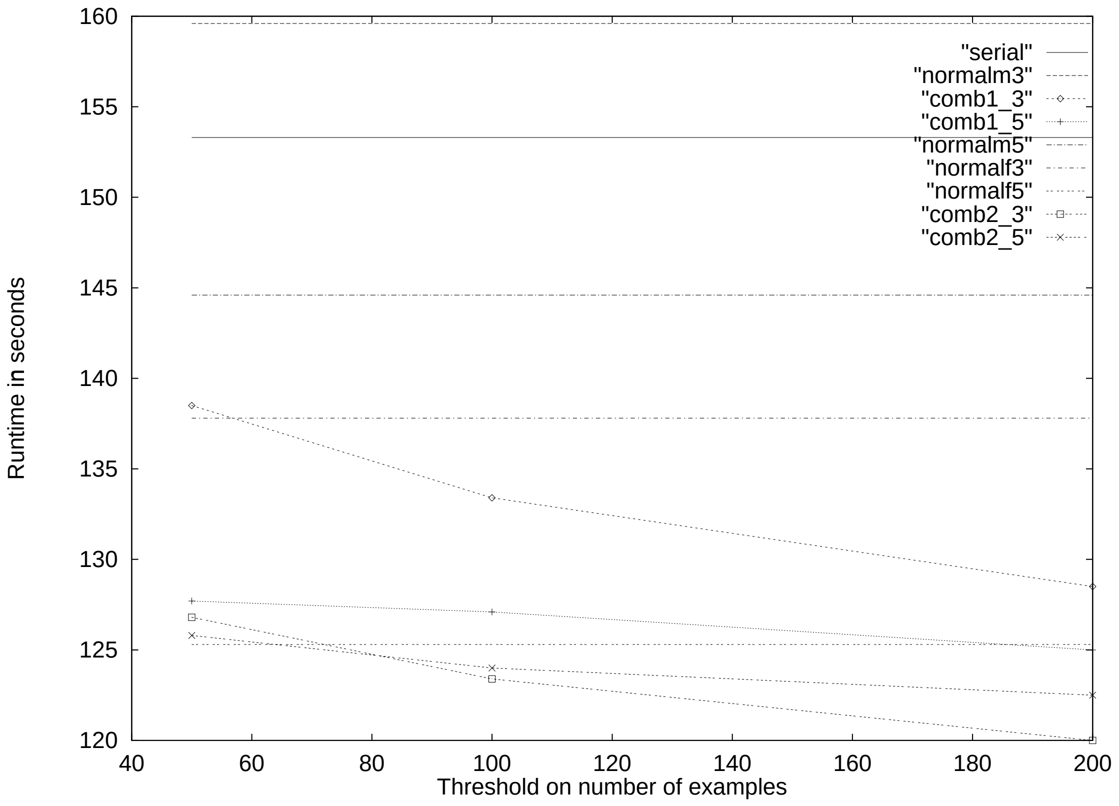


Figure 13: Comparison of runtime for dataset hypo using the Message and File techniques on 3 and 5 processors. Refer to Figure 10 for legend description.

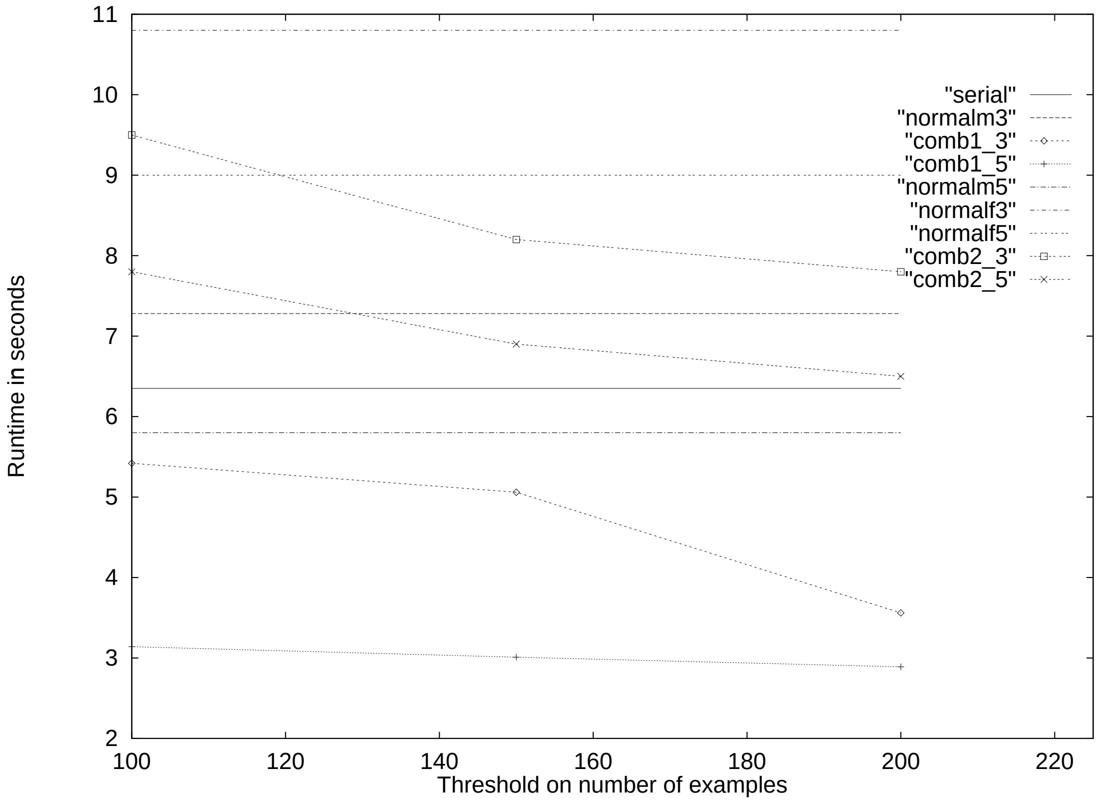


Figure 14: Comparison of runtime for dataset splice using the Message and File techniques on 3 an 5 processors. Refer to Figure 10 for legend description.

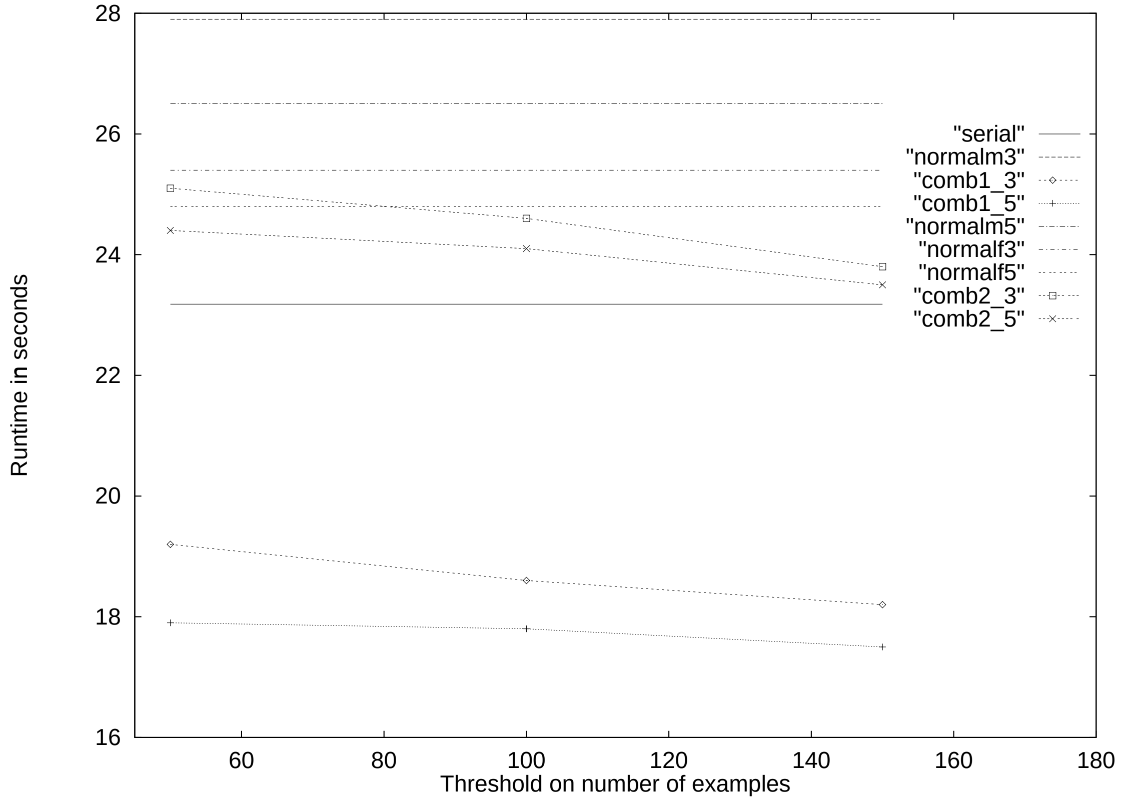


Figure 15: Comparison of runtime for dataset vehicle using the Message and File techniques. Refer to Figure 10 for legend description.

4.7 Remarks

Based on our results we can make the following conclusions.

- Of the four different parallel techniques that we tried, the two basic parallel techniques can be considered not helpful in terms of decreasing the runtime. The other two combined techniques showed gains in terms of decreasing the runtime but the gains are not very large.
  We can attribute these results to the following:

  * *Creating subsets of data*
    One of the dominant parts of the algorithm is the creation of data subsets based on information vectors. These subsets are the data points at a node and are to used for extending the tree from that node. Our implementation assumes that all of the data is present at each host and each host has to create its own data subset. The time required for creation of the subset depends on the number of total data points in the data set and the number of attributes tested in the current branch. As this process is carried out for all the nodes in the tree, it plays a major role in the overall time taken by the algorithm. The subset creation from information vectors is a dominant factor in the Message passing techniques and also to some extent in the File techniques.
  * *File Input-Output*
    In the File method, each data point has an entry in a datamap file to be read by the host for its subset of data. In this case a file has to created for each value that an attribute can hold (e.g., if *Temperature* is the best attribute at a node and it can take three values *Hot, Mild and Cold*, there will be three files created for this attribute). The time taken for this operation is directly proportional to the number of examples in a dataset and the number of different attribute values the attributes can hold. As the number of values an attribute can have, increases the number files that have to be created and read, resulting in more file input-output. So this also is a dominant factor in the time for execution in the File technique.
  * *Message Passing*
    We are using MPI for communicating between machines (processors). MPI provides a simple abstraction for communicating between processors and works well with shared memory processors and special parallel machines. But where data has to passed between nodes often and in our case only *information* data, the message passing may cause additional overhead. MPI

    has its own error-checking mechanism for exchange of data and has its own internal structure for managing the set of processors in an execution. These features will also certainly contribute to the runtime of the algorithm.

  * *Network of workstations*
    All our implementations run on a network of workstations. So when two hosts communicate the message passes through a (public) network before it reaches the other end. The time required for this communication is directly related to the type of network connection and the load at that time on the network. We would predict that with better network connections and with high bandwidth or special hardware such as shared memory processors or a parallel machine there will certainly be better results than that are obtained with the current setup.

- The size of dataset is an important factor in the parallelizing process. If a dataset is too small, the overhead of communication and creating subsets of data at different hosts will not be justified. The results in such cases are not very interesting. Of interest are the larger datasets, where the overhead of creating data subsets will be more, but that will probably be nullified by the fact that many such hosts are doing the same tasks in parallel on reasonably large amounts of data.

5 Conclusions

In this thesis, we implemented four different parallel decision tree learning algorithms and evaluated each of them using sets of learning data drawn from the UCI repository. Based on the experiments that we conducted, our results show small but significant gains in the process time of the algorithms.
The four techniques we experimented with are: Basic Message Passing, First Combined Message Passing, Basic File and the Second Combined File methods. The two combination methods are ones that have a parallel and a serial component.
In the preliminary tests, the runtime of the Basic Message Passing and File methods was compared with the runtime of the serial algorithm. These two basic techniques proved not to be very helpful in reducing the runtime of the tree building algorithm. The runtime for these methods was nearly same as that of the serial approach and in some cases more than the serial approach.
The other two techniques: the First Combined Message Passing Method and the Second Combined File Method proved to be useful and showed some significant gain as compared to the serial approach. Comparing the two approaches individually, both of them had nearly the same runtime values. Both the combined approaches performed better than the serial algorithm in almost all cases and significantly better than the Basic Message Passing method and the Basic File method.

Based on the experiments that we performed on our implementations, we will try to answer the questions we asked in Chapter 1.

Question 1: *Can the decision tree algorithm be parallelized by exploiting the natural parallelism involved in building a decision tree?*

We have implemented four different techniques for building a tree in parallel. The implementations prove that the inherent parallelism involved in the tree building algorithm of constructing nodes in parallel is one of the ways to parallelize the algorithm.

Question 2: *How effective will the resulting algorithm be for a network of workstations?*

Of the four implementations, the two basic algorithms show no significant gains. The First Combined Message Passing method and the Second Combined File method show gains in the runtime and considering that the implementations are based on a network of workstations they are significant.

Question 3: *What is the best mechanism for passing information about a node to be processed?*

We have tried two approaches: sending information vectors as messages and making datamaps to store data subsets for the hosts. Both the approaches are designed using the fact that we are running our experiments on a network of workstations and we rely on the MPI libraries for exchange of messages. These two are not the best techniques in general, but they provide an useful insight into how parallel algorithms can be designed.

Based on the results that we obtained we can conclude that the runtime of our approaches depended primarily on the following factors: creating subsets of data at hosts, file input-output, message passing and traffic on the network of workstations. We predict that the results that we obtained can definitely be enhanced in better computing environments.

5.1 Future Work

We have tried to parallelize the tree building algorithm by building nodes in parallel and assuming that all data exists at all participating hosts. Other possibilities for parallelizing include partitioning the data based on its features or just selecting a subset of important features. We can try methods in which the features are split across more than a single machine. In this case each machine works only with a subset of features and effectively will process less data. Using this technique will help processing nodes nearer to the *root*, as nodes near the *root* have larger number of data points. Using a high-bandwidth network, we can pass data to hosts for processing and save time required to create datasets at hosts. We can also use alternative computing environments like, special high speed parallel machines or shared memory systems. We might also be helped by using a different message passing model other than MPI, which could reduce the overhead in passing messages between hosts.

We conclude from the thesis that, even though not very significant gains were obtained from the implementations, they gave an insight into issues related to parallel learning algorithms. More work has to be done to explore different possibilities for parallelizing the algorithm along with new computing environments for parallel processing.

References

[Agarwal and Shim, 1996] Agarwal, R. A. and Shim, K. (1996). Developing tightly-coupled data mining applications on a relational database system. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining(KDD–96)*, pages 287–290, Portland, Oregon. AAAI Press.

[Almuallim et al., 1995] Almuallim, H., Akiba, Y., and Kaneda, S. (1995). On handling tree structure attributes in decision tree learning. In *Proceedings of Twelfth International Conference on Machine Learning (ML–93)*. Morgan Kaufmann.

[Aronis and Provost, 1997] Aronis, J. M. and Provost, F. J. (1997). Efficient data mining with or without hierarchical background knowledge. In *Proceedings of Third International Conference on Knowledge Discovery and Data Mining (KDD'97)*.

[Blake and Merz, 1998] Blake, C. and Merz, C. (1998). UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[Breiman et al., 1984] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, P. J. (1984). *Classification and Regression Trees.* Wadsworth International Group, CA.

[Catlett, 1991] Catlett, J. (1991). *Megainduction: Machine Learning on Very Large Databases.* PhD thesis, Dept. of Computer Science, University of Sydney, Australia.

[Cook and Holder, 1990] Cook, D. and Holder, L. (1990). Accelerated learning on the connection machine. In *Proceedings of the Second IEEE Symposium on Parallel and Distribued Processing*, pages 448–454.

[Fayyad, 1991] Fayyad, U. M. (1991). *On the induction of decision trees for multiple concept learning.* PhD thesis, Electrical Engineering and Computer Science Department, University of Michigan.

[Foster, 1995] Foster, I. T. (1995). *Designing and Building Parallel Programs.* Addison-Wesley.

[Furukranz and Widmer, 1994] Furukranz, J. and Widmer, G. (1994). Incremental reduced error pruning. In *Machine Learning: Proceedings of the Eleventh Annual Conference*, New Brunswick, NJ.

[Gropp et al., 1999] Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, 2nd edition.

[Holte, 1993] Holte, R. C. (1993). Very simple classification rules perfrom well on most commonly used datasets. *Machine Learning*, 3:63–91.

[Iba and Langley, 1992] Iba, W. F. and Langley, P. (1992). Induction of one-level decision trees. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 233–240. Morgan Kaufmann.

[Kononenko et al., 1984] Kononenko, I., Bratko, I., and Roskar, E. (1984). Experiments in automatic learning of medical diagnostic rules. (Technical Report), Jozef Stefan Institute, Ljubjjana. Yugoslavia.

[Kumar and Rao, 1987] Kumar, V. and Rao, V. (1987). Parallel depth-first search, Part 2: Analysis. In *International Journal of Parallel Programming*, pages 16,pp:501–519.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning.* WCB/McGraw-Hill.

[Musick et al., 1993] Musick, R., Catlett, J., and Russell, S. (1993). Decision theoretic subsampling for induction on large databases. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 212–219, San Mateo, CA.

[Pacheo, 1996] Pacheo, P. S. (1996). *Parallel Programming with MPI.* Morgan Kaufmann.

[Provost and Kolluri, 1999] Provost, F. and Kolluri, V. (1999). A Survey of Methods for Scaling Up Inductive Algorithms. *Knowledge Discovery and Data Mining (1999).*

[Quinlan, 1987a] Quinlan, J. R. (1987a). Rule induction with statistical data–A comparison with multiple regression. *Journal of the Operational Research Society*, 38,347–352.

[Quinlan, 1987b] Quinlan, J. R. (1987b). Simplifying decision trees. *International Journal of Man-Machine Studies*, pages 27:221–234.

[Quinlan, 1990] Quinlan, J. R. (1990). Induction of Decision Trees. In *Readings in Machine Learning.* Morgan Kaufmann. Originally published in *Machine Learning* 1:81–106, 1986.

[Quinlan, 1993] Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Mogran Kaufmann.

[Ribeiro et al., 1996] Ribeiro, J. S., Kaufmann, K. A., and Kerschberg, L. (1996). Knowledge discovery from multpile databases. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining(KDD–96)*, pages 240–245, Menlo Park, CA. AAAI Press.

[Shafer et al., 1996] Shafer, J., Agarwal, R., and Mehta, M. (1996). SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proceedings of the 22nd VLDB Conference*, Mumbai, India.

[Snir et al., 1996] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1996). *MPI: The Complete Reference.* The MIT Press.

[Stolfo et al., 1997] Stolfo, S. J., Fan, D. W., Lee, W., Prodromidis, A. L., and Chan, P. K. (1997). Credit Card Fraud Detection Using Meta Learning: Issues and Initial Results. Issues and initial results. Working notes of AAAI Workshop on AI Approaches to Fraud Detection and Risk Management 1997.