Chapter 1

Parallel Classification on Shared-Memory Systems

MOHAMMED J. ZAKI[†], CHING-TIEN HO[‡], RAKESH AGRAWAL[‡]

[†]Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180, USA

[‡]IBM Almaden Research Center
San Jose, CA 95120, USA

Email: *zaki@cs.rpi.edu, ho@almaden.ibm.com, ragrawal@almaden.ibm.com*

1.1 Introduction

An important task of data mining can be thought of as the process of assigning things to predefined categories or classes – a process called *Classification*. Since the classes are predefined this is also known as *Supervised Induction*. The input for the classification system consists of a set of example records, called a *training set*, where each record consists of several fields or *attributes*. Attributes are either *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. One of the attributes, called the *classifying* attribute, indicates the *class* or label to which each example belongs. The induced model consists of patterns that are useful in class discrimination. Once induced, the model can help in the automatic prediction of new unclassified data. Classification has been identified as an important problem in the emerging field of data mining (Agrawal, Imielinski, & Swami 1993). It has important applications in diverse domains like retail target marketing, customer retention, fraud detection and medical diagnosis (Michie, Spiegelhalter, & Taylor 1994).

Classification is a well-studied problem (see (Weiss & Kulikowski 1991; Michie, Spiegelhalter, & Taylor 1994) for excellent overviews) and several models have been proposed over the years, which include neural networks (Lippmann 1987), statistical models like linear/quadratic discriminants (James 1985), decision trees (Breiman *et al.* 1984; Quinlan 1993) and genetic algorithms (Goldberg 1989). Among these models, decision trees are particularly suited for data mining (Agrawal, Imielinski, & Swami 1993; Mehta, Agrawal, & Rissanen 1996). Decision trees can be constructed relatively fast compared to other methods. Another advantage is that decision tree models are simple and easy to understand (Quinlan 1993). Moreover, trees can be easily converted into SQL statements that can be used to access databases efficiently (Agrawal *et al.* 1992). Finally, decision tree classifiers obtain similar, and sometimes better, accuracy when compared with other classification methods (Michie, Spiegelhalter, & Taylor 1994). We have therefore focused on building scalable and parallel decision-tree classifiers.

While there has been a lot of research in classification in the past, the focus had been on memory-resident data, thus limiting their suitability for mining over large databases. Recent work has targeted the massive databases usual in data mining. Classifying larger datasets can enable the development of higher accuracy models. Various studies have confirmed this hypothesis (Catlett 1991; Chan & Stolfo 1993a; 1993b). Examples of fast scalable classification systems include SLIQ (Mehta, Agrawal, & Rissanen 1996), which for the first time was successful in handling disk-resident data. However, it did require some hashing information to be maintained in memory, restricting its scalability. The SPRINT (Shafer, Agrawal, & Mehta 1996) classifier was able to remove all such restrictions. It was also parallelized on the IBM SP2 distributed-memory machine (Shafer, Agrawal, & Mehta 1996).

A continuing trend in data mining is the rapid and inexorable growth in the data that is being collected. The development of high-performance scalable data mining tools must necessarily rely on parallel computing techniques. Past work on parallel classification has utilized distributed-memory parallel machines. In such a machine, each processor has private memory and local disks, and communicates with other processors only via passing messages. Parallel distributed-memory machines are essential for scalable massive parallelism. However, shared-memory multiprocessor systems (SMPs), often called shared-everything systems, are also capable of delivering high performance for low to medium degree of parallelism at an economically attractive price. SMP machines are the dominant types of parallel machines currently used in industry. Individual nodes of parallel distributed-memory machines are also increasingly being designed to be SMP nodes. For example, an IBM SP2 parallel system may consist of up to 64 *high nodes*, where each high node is an 8-way SMP system with PowerPC 604e processors (IBM). A shared-memory system offers a single memory address space that all processors can access. Processors communicate through shared variables in memory and are capable of accessing any memory location. Synchronization is used to co-ordinate processes. Any processor can also access any disk attached to the system.

This paper presents fast scalable decision-tree-based classification algorithms targeting shared-memory systems, the first such study. The algorithms are based on the sequential SPRINT classifier, and span the gamut of data and task parallelism. The data parallelism is based on attribute scheduling among processors. This is extended with task pipelining and dynamic load balancing to yield more complex schemes. The task parallel approach uses dynamic subtree partitioning among processors. These algorithms are evaluated on two SMP configurations: one in which data is too large to fit in memory and must be paged from a local disk as needed and the other in which memory is sufficiently large to hold the whole input data and all temporary files. For the local disk configuration, the speedup ranged from 2.97 to 3.86 for the build phase and from 2.20 to 3.67 for the total time on a 4-processor SMP. For the large memory configuration, the range of speedup was from 5.36 to 6.67 for the build phase and from 3.07 to 5.98 for the total time on an 8-processor SMP.

The rest of the paper is organized as follows. We review related work in Section 1.2. In Section 1.3 we describe the sequential SPRINT decision-tree classifier, which forms the backbone of the new algorithms. This section is adapted from (Shafer, Agrawal, & Mehta 1996). Section 1.4 describes our new SMP algorithms based on various data and task parallelization schemes. We give experimental results in Section 1.5 and conclude with a summary in Section 1.6.

1.2 Related Work

Random sampling is often used to handle large datasets when building a classifier. Previous work on building tree-classifiers from large datasets includes Catlett's study of two methods (Catlett 1991; Wirth & Catlett 1988) for improving the time taken to develop a classifier. The first method used data sampling at each node of the decision tree, and the second discretized continuous attributes. However, Catlett only considered datasets that could fit in memory; the largest training data had only 32,000 examples. Chan and Stolfo (Chan & Stolfo 1993a; 1993b) considered partitioning the data into subsets that fit in memory and then developing a classifier on each subset in parallel. The output of multiple classifiers is combined using various algorithms to reach the final classification. Their studies showed that although this approach reduces running time significantly, the multiple classifiers did not achieve the accuracy of a single classifier built using all the data. Incremental learning methods, where the data is classified in batches, have also been studied (Quinlan 1979; Wirth & Catlett 1988). However, the cumulative cost of classifying data incrementally can sometimes exceed the cost of classifying the entire training set once. In (Agrawal *et al.* 1992), a classifier built with database considerations, the size of the training set was overlooked. Instead, the focus was on building a classifier that could use database indices to improve the retrieval efficiency while classifying test data.

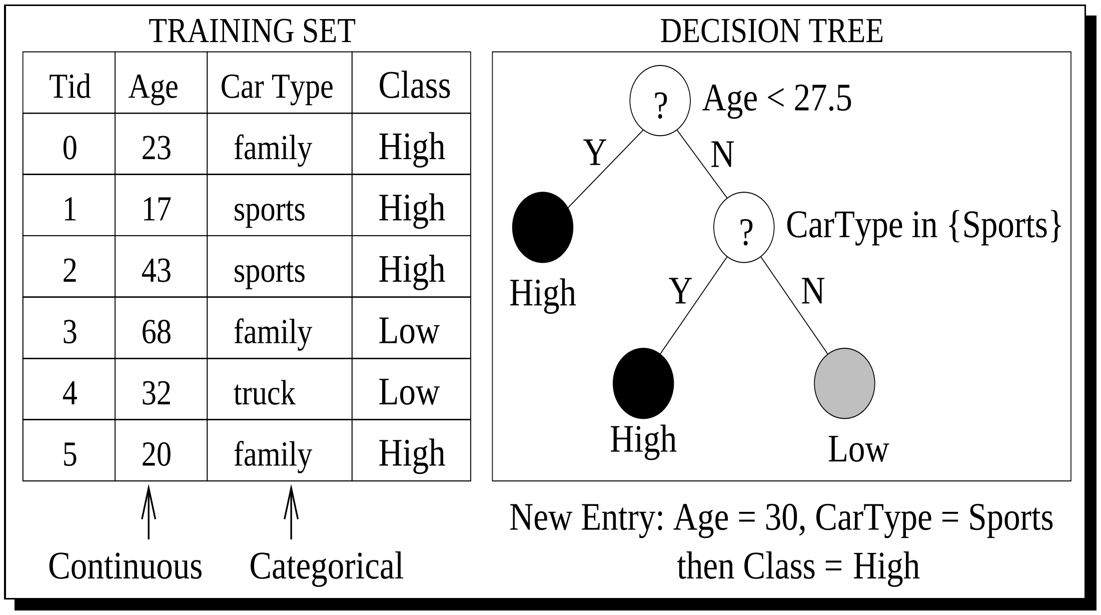
Work by Fifield in (Fifield 1992) examined parallelizing the ID3 (Quinlan 1986) decision-tree classifier, but it assumes that the entire dataset can fit in memory and does not address issues such as disk I/O. The algorithms presented there also require processor communication to evaluate any given split point, limiting the number of possible partitioning schemes the algorithms can efficiently consider for each leaf. The Darwin toolkit from Thinking Machines also contained a parallel implementation of the decision-tree classifier CART (Breiman *et al.* 1984); however, details of this parallelization are not available in published literature.

The recently proposed SLIQ classification algorithm (Mehta, Agrawal, & Rissanen 1996) addressed several issues in building a fast scalable classifier. SLIQ gracefully handles disk-resident data that is too large to fit in memory. It does not use small memory-sized datasets obtained via sampling or partitioning, but builds a single decision tree using the *entire* training set. However, SLIQ does require that some data per record stay memory-resident all the time. Since the size of this in-memory data structure grows in direct proportion to the number of input records, this limits the amount of data that can be classified by SLIQ. Its successor, the SPRINT classifier (Shafer, Agrawal, & Mehta 1996) removed all memory restrictions and was fast and scalable. It was also parallelized on the IBM SP2 distributed-memory system. ScalParC (Joshi, Karypis, & Kumar 1998) is also a parallel classifier for the Cray T3D distributed-memory machine. Other recent work on scalable classification includes CLOUDS (Alsabti, Ranka, & Singh 1998) and its distributed-memory parallelization pCLOUDS (Sreenivas, Alsabti, & Ranka 1999), PUBLIC (Rastogi & Shim 1998), Rainforest (Gehrke, Ramakrishnan, & Ganti 1998).

As noted earlier, our goal in this paper is to study the efficient implementation of SPRINT on shared-memory systems. These machines represent a popular parallel programming architecture and paradigm, and have very different characteristics. A shorter version of this paper appears in (Zaki, Ho, & Agrawal 1999).

1.3 Serial Classification

A decision tree contains tree-structured nodes. Each node is either a *leaf*, indicating a class, or a *decision node*, specifying some test on one or more attributes, with one branch or subtree for each of the possible outcomes of the split test. Decision trees successively divide the set of training examples until all the subsets consist of data belonging entirely, or predominantly, to a single class. Figure 1.1 shows a decision-tree classifier developed from the example training set. ($Age < 27.5$) and ($CarType \in \{sports\}$) are two split points that partition the records into *High* and *Low* risk classes. The decision tree can be used to screen future insurance applicants by classifying them into the High or Low risk categories.



TRAINING SET

Tid	Age	Car Type	Class
0	23	family	High
1	17	sports	High
2	43	sports	High
3	68	family	Low
4	32	truck	Low
5	20	family	High

Figure 1.1: Car Insurance Example

A decision tree classifier is usually built in two phases (Breiman *et al.* 1984; Quinlan 1993): a growth phase and a prune phase. The tree is grown using an elegantly simple divide and conquer approach. It takes as input a set of training examples S. There are basically two cases to be considered. If all examples in S entirely or predominantly (a user specified parameter) belong to a single class, then S becomes a leaf in the tree. If on the other hand it contains a mixture of examples from different classes, we need to further partition the input into subsets that tend towards a single class. The data is partitioned based on a test on the attributes, and can take the form of a binary or k-ary split. We will consider only binary splits because they usually lead to more accurate trees; however, our techniques can be extended to handle multi-way splits. Based on the outcomes, the data is partitioned into two subsets S_1 and S_2, which in turn serve as inputs to the recursive process. This tree growth phase is shown in Figure 1.2.

```
Partition(Data $S$)
  if (all points in $S$ are of the same class) then
     return;
  for each attribute $A$ do
     evaluate splits on attribute $A$;
  Use best split found to partition $S$ into $S_1$ and $S_2$;
  Partition($S_1$);
  Partition($S_2$);

Initial call: Partition(TrainingData)
```

Figure 1.2: General Tree-growth Algorithm

The tree built using the recursive partitioning approach can become very complex, and as such can be thought of as being an “overfit” of the data. Remember that the goal of classification is to predict new unseen cases. The tree pruning phase tries to generalize the tree by removing dependence on statistical noise or variation that may be particular only to the training set. This step requires access only to the fully grown tree, while the tree growth phase usually requires multiple passes over the training data, and as such is much more expensive. Previous studies from SLIQ suggest that usually less than 1% of the total time needed to build a classifier was spent in the pruning phase. In this paper we will therefore only concentrate on the computation and I/O intensive tree growth phase. We use a Minimum Description Length (Rissanen 1989) based algorithm for the tree pruning phase. (see (Mehta, Agrawal, & Rissanen 1996) for additional details). There are two major issues that have critical performance implications in the tree-growth phase.

1. How to find split points that define node tests.

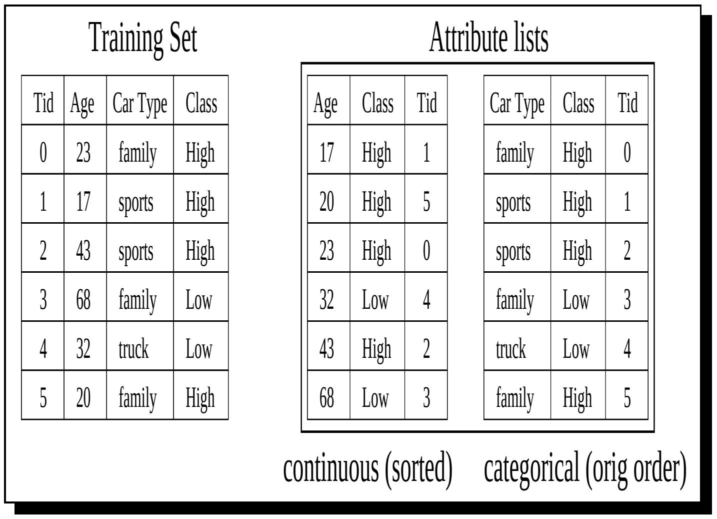
2. Having chosen a split point, how to partition the data.

We will first look at the data structures used in SPRINT, and then describe how it handles the two steps above.

1.3.1 Data Structures

The SPRINT classifier was designed to be disk-based. It builds the tree breadth-first and uses a one-time pre-sorting technique to reduce the cost of continuous attribute evaluation. In contrast to this, the well-known CART (Breiman *et al.* 1984) and C4.5 (Quinlan 1993) classifiers, grow trees depth-first and repeatedly sort the data at every node of the tree to arrive at the best splits for continuous attributes.

Attribute lists SPRINT initially creates an disk-based *attribute list* for each attribute in the data. Each entry in the list is called an *attribute record*, and consists of an attribute value, a class label, and a record identifier or *tid*. Initial lists for continuous attributes are sorted by attribute value when first created. The lists for categorical attributes remain in unsorted order. Figure 1.3 shows the initial attribute lists for our example training set. The initial lists created from the training set are associated with the root of the classification tree. As the tree is grown and is split into two subtrees, the attribute lists are also split at the same time.



Training Set

Tid	Age	Car Type	Class
0	23	family	High
1	17	sports	High
2	43	sports	High
3	68	family	Low
4	32	truck	Low
5	20	family	High

Attribute lists

continuous (sorted)

Age	Class	Tid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

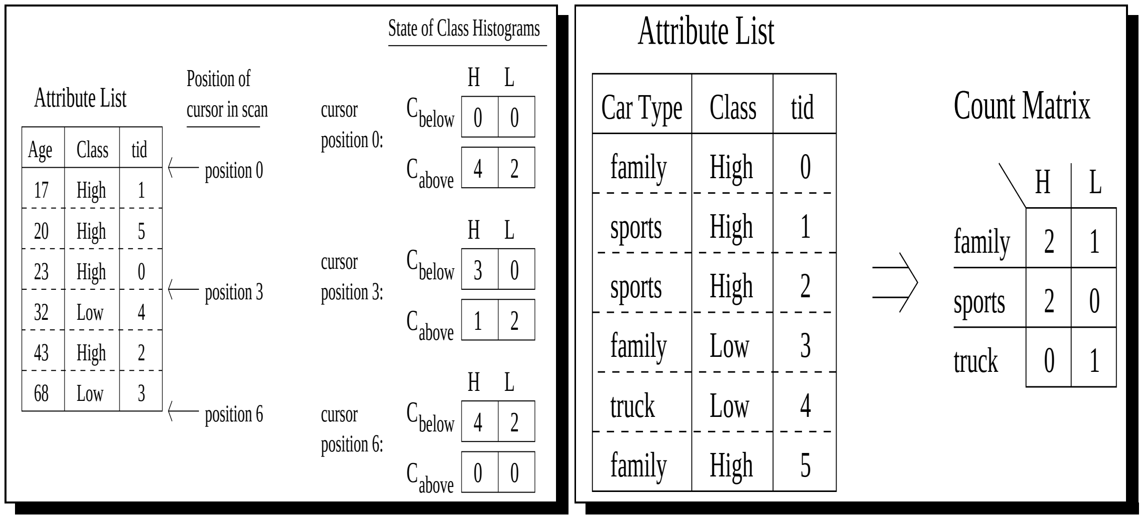
categorical (orig order)

Car Type	Class	Tid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Figure 1.3: Attribute Lists

By simply preserving the order of records in the partitioned lists, they don't require resorting. Figure 1.5 shows an example of the initial sorted attribute lists associated with the root of the tree and also the resulting partitioned lists for the two children.

Histograms SPRINT uses histograms tabulating the class distributions of the input records. For continuous attributes, two histograms are maintained – C_{below} keeps the counts for examples that have already been processed, and C_{above} for those that have not been seen yet. For categorical attributes one histogram, called the *count matrix*, is sufficient for the class distribution of a given attribute. There is one set of histograms for each node in the tree. However, since the attribute lists are processed one after the other, only one set of histograms need be kept in memory at any given time. Example histograms are shown in Figure 1.4.



Attribute List

Age	Class	tid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

State of Class Histograms

Position of cursor in scan		H	L
cursor position 0:	C_{below}	0	0
	C_{above}	4	2
cursor position 3:	C_{below}	3	0
	C_{above}	1	2
cursor position 6:	C_{below}	4	2
	C_{above}	0	0

Attribute List

Car Type	Class	tid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Count Matrix

	H	L
family	2	1
sports	2	0
truck	0	1

Figure 1.4: Evaluating a) Continuous and b) Categorical Split Points

1.3.2 Finding good split points

The form of the split used to partition the data depends on the type of the attribute used in the split. Splits for a continuous attribute A are of the form $value(A) < x$ where x is a value in the domain of A. Splits for a categorical attribute A are of the form $value(A) \in X$ where $X \subset domain(A)$.

To build compact trees modeling the training set, one approach would be to explore the space of all possible trees and select the best one. This process is unfortunately NP-Complete. Most tree construction methods therefore use a *greedy* approach, the idea being to determine the split point that "best" divides the training records belonging to that leaf. The "goodness" of the split obviously depends on how well it separates the classes. Several splitting indices have been proposed in the past to evaluate the goodness of the split. SPRINT uses the *gini* index (Breiman *et al.* 1984) for this task. For a data set S containing examples from n classes, $gini(S)$ is defined as

$$gini(S) = 1 - \sum p_j^2$$

where p_j is the relative frequency of class j in S. If a split divides S into two subsets S_1 and S_2, with n_1 and n_2 classes, respectively, the index of the divided data $gini_{split}(S)$ is given by

$$gini_{split}(S) = \frac{n_1}{n} gini(S_1) + \frac{n_2}{n} gini(S_2)$$

The computation of the gini index only requires the class distributions on each side of a candidate partition point. This information is kept in the class histograms described above. Once this information is known, the best split can be found using a simple approach. Each node's attribute lists are scanned, the histograms are updated, and the gini value for different split points is calculated. The attribute with the minimum gini value and the associated split point is used to partition the data.

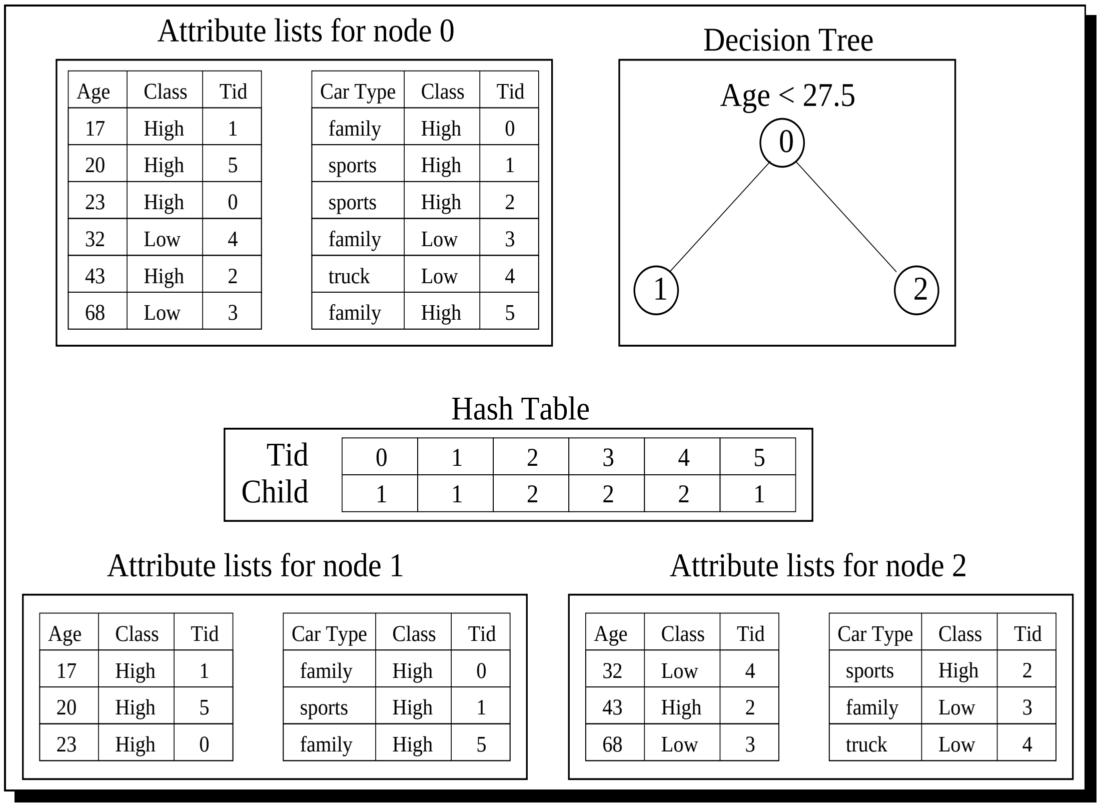
Continuous attributes For continuous attributes, the candidate split points are mid-points between every two consecutive attribute values in the training data. Initially C_{below} has all zeros, while C_{above} has the class distributions of the current node under consideration. For each attribute record, the histograms are updated and the gini index is calculated. The current minimum gini value, also called the *winning* split point is saved during this process. Since all the continuous values are sorted one scan over the attribute lists is sufficient. Figure 1.4a illustrates the histogram update for continuous attributes.

Categorical attributes For categorical attributes, all possible subsets of the attribute values are considered as potential split points. If the cardinality is too large a greedy subsetting algorithm (initially used in IND (NAS 1992)) is used. The histogram updation is shown in Figure 1.4b.

1.3.3 Splitting the data

Once the winning split point has been found for a node, the node is split into two children, along with the division of the node's attribute lists into two. Figure 1.5 shows this process. The attribute list splitting for the winning attribute (Age in our example) is quite straightforward. We simply scan the attribute records, and apply the split test. Those records satisfying the test go to the left child, and those that fail the test go to the right child. For the remaining "losing" attributes ($CarType$ in our example) more work is needed. While dividing the winning attribute SPRINT also constructs a probe structure (bit mask or hash table) on the *tids*, noting the child where a particular record belongs. To split the other attributes now only requires a scan of each record and a probe to determine the child where this record should be placed. This probe structure need not be memory-resident. If it occupies too much memory the splitting takes multiple steps. In each step only a portion of the attribute lists are partitioned. At the same time the split happens, SPRINT also collects the new class distribution histograms for the children nodes.

Avoiding multiple attribute lists Recall that the attribute lists of each attribute are stored in disk files. As the tree is split, we would need to create new files for the children, and delete the parent's files. File creation is usually an expensive operation, and this process can add significant overhead. For example, Figure 1.6 shows that if we create new files at each level, then if the tree has N levels, we would potentially require 2^N files. Rather than creating a separate attribute list for each attribute for each node, SPRINT actually uses only four physical files per attribute. Since we are dealing with binary splits, we have one attribute file for all leaves that are "left" children (file $L0$) and one file for all leaves that are "right" children (file $R0$). We also have two more list files per attribute that serve as alternates (files $L1$ and $R1$). All the attribute records for a


Attribute lists for node 0
Decision Tree
Age < 27.5
0
1
2
Hash Table
Attribute lists for node 1
Attribute lists for node 2

Attribute lists for node 0

Age	Class	Tid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

Car Type	Class	Tid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Hash Table

Tid	0	1	2	3	4	5
Child	1	1	2	2	2	1

Attribute lists for node 1

Age	Class	Tid
17	High	1
20	High	5
23	High	0

Car Type	Class	Tid
family	High	0
sports	High	1
family	High	5

Attribute lists for node 2

Age	Class	Tid
32	Low	4
43	High	2
68	Low	3

Car Type	Class	Tid
sports	High	2
family	Low	3
truck	Low	4

Figure 1.5: Splitting a Node's Attribute Lists

node are kept in a contiguous section within one of the two primary files (files $L0$ and $R0$). When reading records for a particular node, we read the corresponding portion of the attribute's left or right file. When splitting a node's attribute records, we append the left and right child's records to the end of the alternate left and right files (files $L1$ and $R1$).

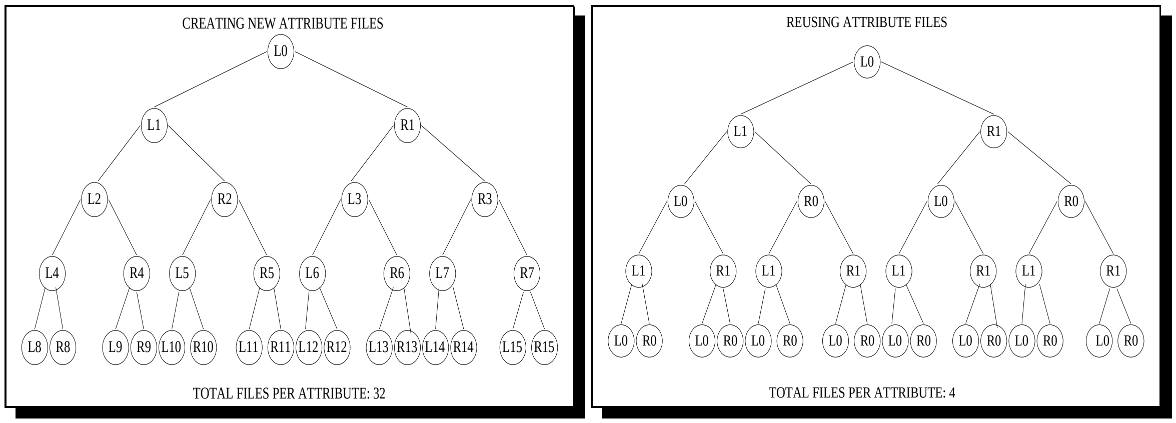


Figure 1.6: Avoiding Multiple Attribute Files

After splitting each node, all the training data will reside in the alternate files. These become the primary files for the next level. The old primary lists are cleared and they become new alternate files. Thus, we never have to pay the penalty of creating new attribute lists for new leaves; we can simply reuse the ones we already have. By processing tree nodes in the order they appear in the attribute files, this approach also avoids any random seeks within a file to find a node's records — reading and writing remain sequential operations. This optimization can be done with virtually no overhead and with no modifications to the SPRINT algorithm. It also has important implications for the parallelization strategies presented below.

1.4 Parallel Classification on Shared-memory Systems

We now turn our attention to the problem of building classification trees in parallel on SMP systems. We will only discuss the tree growth phase due to its compute and data-intensive nature. Tree pruning is relatively inexpensive (Mehta, Agrawal, & Rissanen 1996), as it requires access to only the decision-tree grown in the training phase.

1.4.1 SMP Schemes

While building a decision-tree, there are three main steps that must be performed for each node at each level of the tree:

1. Evaluate split points for each attribute (denoted as step $\mathcal{E}$).
2. Find the winning split-point and construct a probe structure using the attribute list of the winning attribute (denoted as step $\mathcal{W}$).
3. Split all the attribute lists into two parts, one for each child, using the probe structure (denoted as step $\mathcal{S}$).

Our parallel schemes will be described in terms of these steps. Our prototype implementation of these schemes uses the POSIX threads (pthread) standard (Lewis & Berg 1996). A thread is a light weight process. It is a *schedulable* entity that has only those properties that are required to ensure its independent flow of control, including the stack, scheduling properties, set of pending and blocking signals, and some thread-specific data. To keep the exposition simple, we will not differentiate between threads and processes and pretend as if there is only one process per processor. We propose two approaches to building a tree classifier in parallel: a *data parallel* approach and a *task parallel* approach.

Data parallel In data parallelism the P processors work on distinct portions of the datasets and synchronously construct the global decision tree. It essentially exploits the intra-node parallelism, i.e. that available within a decision tree node. There are two kinds of data parallelism possible in classification. In the first case, called *attribute parallelism*, the attributes are divided equally among the different processors so that each element is responsible for $1/P$ attributes. In the second case, called *record parallelism*, we split the attribute lists evenly among all the processors. Each processor is responsible for $1/P$ records from each attribute list. The implementation of this scheme on the IBM SP2 was presented in (Shafer, Agrawal, & Mehta 1996).

The record parallelism approach doesn't look very promising on an SMP system. Some of the reason are: First, for each continuous attribute, SPRINT sorts the attribute list then partitions the sorted list into P "continuous" sublists of the same size so that each processor has the same amount of work. As the tree grows, different sublists with varying numbers of records are split into a left or right child. When some of these nodes become pure (i.e., contain records belonging to the same class), the work associated with that node disappears. This creates load imbalance, especially when the data distribution is skewed and the tree is large. Performing dynamic load balancing with record data parallelism is quite complex and likely to introduce extra overhead.

Second, for each categorical attribute, its class histogram will have to be replicated on all processors, so that they can independently process their local attribute lists. If there are many categorical attributes or if they have a large domain of possible values, the incurred space overhead can be significant. Without such replication, the synchronization (say, of acquiring and releasing a lock) will have to be done at the record level.

Finally, the entire hash probe will have to be replicated on all processes. Furthermore, at every tree level, the information stored in each bit probe needs to be copied to all other bit probes. Thus, there are associated space and time overheads. Alternatively, if the bit probe is not replicated, then either a single global bit probe has to be lock-protected (due to concurrent writes at the record level) which is unattractive performance-wise, or a complicated scheduling, of when to write to bit probe and by which process, has to be devised to avoid sequential bottleneck. Having said this, we will implement the record parallel approach in the future, to experimentally confirm these points.

In summary, we see that one of the main reasons why record parallelism or a horizontal data partitioning is not good on a SMP, has to do with the fact that we don't want to allow more than one processor to access the same attribute file. Doing so will play havoc with the common bus and will result in disk contention, as more than one process tries to read/write to a different file location at the same time. A vertical partitioning using attribute partitioning eliminates this problem, since each attribute file can be the sole responsibility of a single processor. The other main reason is the replication of data structures needed to implement the record parallelism. On a distributed memory machine each processor gets a local copy of the data structures anyway, but on a SMP all the copies must reside on the same machine, which consumes precious memory resources. This is undesirable. Thus, in this paper we only deal with the attribute parallelism approach.

Task Parallelism The other major kind of parallelism is provided by the task parallel approach. It exploits the inter-node parallelism, i.e. different portions of the decision tree can be built in parallel among the processors.

1.4.2 Attribute Data Parallelism

We first describe the Moving-Window-K algorithm (MWK) based on attribute data parallelism. For pedagogical reasons, we will introduce two intermediate schemes called BASIC and Fixed-Window-K (FWK) and then evolve them to the more sophisticated MWK algorithm. MWK and the two intermediate schemes utilize dynamic attribute scheduling. In a static attribute scheduling, each process gets d/P attributes where d denotes the number of attributes. However, this static partitioning is not particularly suited for classification. Different attributes may have different processing costs because of two reasons. First, there are two kinds of attributes – continuous and categorical, and they use different techniques to arrive at split tests. Second, even for attributes of the same type, the computation depends on the distribution of the record values. For example, the cardinality of the value set of a categorical attribute determines the cost of gini index evaluation. These factors warrant a dynamic attribute scheduling approach.

The Basic Scheme (BASIC)

Figure 1.7 shows the pseudo-code for the BASIC scheme. A barrier represents a point of synchronization. At each level a processor evaluates the assigned attributes, which is followed by a barrier.

```
// Starting with the root node execute the
// following code for each new tree level
forall attributes in parallel (dynamic scheduling)
    for each leaf
        evaluate attributes (E)

barrier
if (master) then
    for each leaf
        get winning attribute; form hash-probe (W)

barrier
forall attributes in parallel (dynamic scheduling)
    for each leaf
        split attributes (S)
```

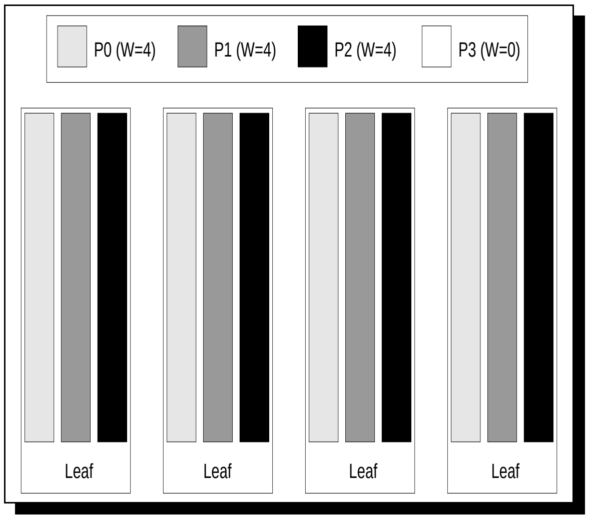


Figure 1.7: The BASIC Algorithm (4 Processors, 3 Attributes)

Attribute scheduling Attributes are scheduled dynamically by using an attribute counter and locking. A processor acquires the lock, grabs an attribute, increments the counter, and releases the lock. This method achieves the same effect as self-scheduling (Tang & Yew 1986), i.e., there is lock synchronization per attribute. For typical classification problems with up to a few hundred attributes, this approach works fine. For thousands of attributes self-scheduling can generate too much unnecessary synchronization. The latter can be addressed by using guided self-scheduling (Polychronopoulos & Kuck 1987) or its extensions, where a processor grabs a dynamically shrinking chunk of remaining attributes, thus minimizing the synchronization. Another possibility would be to use affinity scheduling (Markatos & LeBlanc 1994), where attention is paid to the location of the attribute lists so that accesses to local attribute lists are maximized.

Finding split points ($\mathcal{E}$) Since each attribute has its own set of four reusable attribute files, as long as no two processors work on the same attribute at the same time, there is no need for file access synchronization. To minimize barrier synchronization the tree is built in a breadth-first manner. The advantage is that once a processor has been assigned an attribute, it can evaluate the split points for that attribute for all the leaves in the current level. This way, each attribute list is accessed only once sequentially during the evaluation for a level. Once all attributes have been processed in this fashion, a single barrier ensures that all processors have reached the end of the attribute evaluation phase. In contrast, depth-first tree growth would require a barrier synchronization once per leaf, which could become a significant source of overhead in large trees. As each processor works independently on the entire attribute list, it can independently carry out gini index evaluation to determine the best split point for each attribute assigned to it.

Hash probe construction ($\mathcal{W}$) Once all the attributes of a leaf have been processed, each processor will have what it considers to be the best split for that leaf. We now need to find the best split point from among each processor's locally best split. We can then proceed to scan the winning attribute's records and form the hash probe.

The breadth-first tree construction imposes some constraints on the hash probe construction. We could keep separate hash tables for each leaf. If there is insufficient memory to hold these hash tables in memory, they would have to be written to disk. The size of each leaf's hash table can be reduced by keeping only the smaller child's *tids*, since the other records must necessarily belong to the other child. Another option is to maintain a global bit probe for all the current leaves. It has as many bits as there are tuples in the training set. As the records for each leaf's winning attribute are processed, the corresponding bit is set to reflect whether the record should be written to a *left* or *right* file. A third approach is to maintain an index of valid *tids* of a leaf, and relabel them starting from zero. Then each leaf can keep a separate bit probe.

BASIC uses the second approach, that maintains a global bit vector, due to its simplicity. Both the tasks of finding the minimum split value and bit probe construction are performed serially by a pre-designated master processor. This step thus represents a potential bottleneck in this BASIC scheme, which we will eliminate later in MWK. During the time the master computes the hash probe, the other processors enter a barrier and go to sleep. Once the master finishes, it also enters the barrier and wakes up the sleeping processors, setting the stage for the splitting phase.

Attribute list splitting ($\mathcal{S}$) The attribute list splitting phase proceeds in the same manner as the evaluation. A processor dynamically grabs an attribute, scans its records, hashes on the *tid* for the child node, and performs the split. Since the files for each attribute are distinct there is no read/write conflict among the different processors.

Figure 1.7 shows a small example of how BASIC works. In the example there are four processors and three attributes per leaf. There are four leaves in the current level. Assuming equal work for all attributes, there is always one processor that is idle at each leaf. The figure shows one worst case where the first three processors get $W = 4$ units of work each counting a leaf as one unit, while the last processor has no work to do. This example illustrates the limitation of the BASIC scheme.

The Fixed-Window-K Scheme (FWK)

We noted above that the winning attribute hash probe construction phase $\mathcal{W}$ in BASIC is a potential sequential bottleneck. The Fixed-Window-K (FWK) scheme shown in Figure 1.8 addresses this problem. The basic

```
// Starting with the root node execute the
// following code for each new tree level
forall attributes in parallel (dynamic scheduling)
    for each block of K leaves
        for each leaf i
            evaluate attributes (E)
        if (last leaf of block) then
            barrier
        if (last processor finishing on leaf i) then
            get winning attribute; form hash-probe (W)

barrier
forall attributes in parallel (dynamic scheduling)
    for each leaf
        split attributes (S)
```

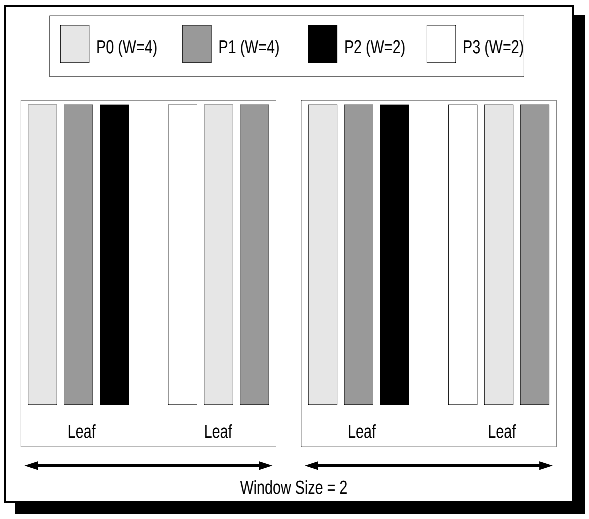


Figure 1.8: The FWK Algorithm (4 Processors, 3 Attributes)

idea is to overlap the $\mathcal{W}$-phase with the $\mathcal{E}$-phase of the next leaf at the current level, a technique called *task pipelining*. The degree of overlap can be controlled by a parameter K denoting the window of current overlapped leaves. Let $\mathcal{E}_i$, $\mathcal{W}_i$, and $\mathcal{S}_i$ denote the evaluation, winning hash construction, and partition steps for leaf i at a given level. Then for $K = 2$, we get the overlap of $\mathcal{W}_0$ with $\mathcal{E}_1$. For $K = 3$, we get an overlap of $\mathcal{W}_0$ with $\{\mathcal{E}_1, \mathcal{E}_2\}$, and an overlap of $\mathcal{W}_1$ with $\mathcal{E}_2$. For a general K, we get an overlap of $\mathcal{W}_i$ with $\{\mathcal{E}_{i+1}, \cdots, \mathcal{E}_{K-1}\}$, for all $1 \leq i \leq K-1$.

The attribute scheduling, split finding, and partitioning remain the same. The difference is that depending on the window size K, we group K leaves together. For each leaf within the K-block (i.e., K leaves of the same group), we first evaluate all attributes. At the last leaf in each block we perform a barrier synchronization to ensure that all evaluations for the current block have completed. The hash probe for a leaf is constructed by the last processor to exit the evaluation for that leaf. This ensures that no two processors access the hash probe at the same time.

Managing attribute files There are four reusable files per attribute in the BASIC scheme. However, if we are to allow overlapping of the hash probe construction step with the evaluation step, which uses dynamic attribute scheduling within each leaf, we would require K distinct files for the current level, and K files for the parent's attribute lists, that is $2K$ files per attribute. This way all K leaves in a group have separate files for each attribute and there is no read/write conflict. Another complication arises from the fact that some children may turn out to be pure (i.e., all records belong to the same class) at the next level. Since these children will not be processed after the next stage, we have to be careful in the file assignment for these children. A simple file assignment, without considering the child purity, where children are assigned files from $0, \cdots, K-1$, will not work well, as it may introduce "holes" in the schedule (see Figure 1.9). However, if we knew the pure children of the next level, we can do better.

The class histograms gathered while splitting the children are adequate to determine purity. We add a pre-test for child purity at this stage. If the child will become pure at the next level, it is removed from the list of valid children, and the files are assigned consecutively among the remaining children. This insures that there are no holes in the K block, and we get perfect scheduling. The two approaches are contrasted in Figure 1.9. The bold circles show the valid children for the current and next level. With the simple labeling scheme the file labels for the valid children are $L0, L0, R0, R0, R0$. With a window of size $K = 2$, there is only one instance where work can overlap, i.e., when going from $L0$ to $R0$. However, if we relabel the valid children's files then we obtain the perfectly schedulable sequence $L0, R0, L0, R0, L0$.

Note that the overlapping of work is achieved at the cost of increased barrier synchronization, one per each K-block. A large window size not only increases the overlap but also minimizes the number of synchronizations. However, a larger window size requires more temporary files, which incurs greater file creation

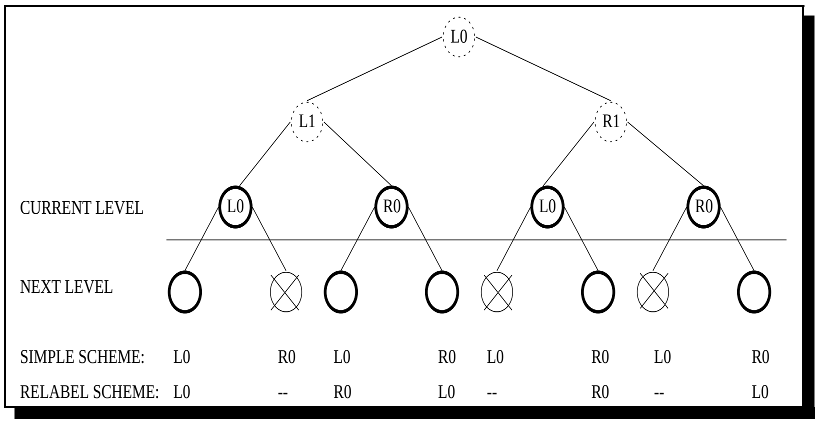

L0
L1
R1
CURRENT LEVEL
L0
R0
L0
R0
NEXT LEVEL
SIMPLE SCHEME: L0 R0 L0 R0 L0 R0 L0 R0
RELABEL SCHEME: L0 -- R0 L0 -- R0 -- L0

Figure 1.9: Scheduling Attribute Files

overhead and tends to have less locality. The ideal window size is a trade-off between the above conflicting goals.

Figure 1.8 shows an illustration of the algorithm. With a window size of two we count the attributes in two leaves as a single block. The example shows a possible assignment of work with round-robin scheduling (i.e., assuming all leaves have equal work) of attributes within a block. FWK is clearly an improvement over BASIC. We have two processors that have 4 units of work and two that have 2 units of work. No processor is completely idle as in BASIC.

The Moving-Window-K Algorithm (MWK)

We now describe the Moving-Window-K (MWK) algorithm which eliminates the serial bottleneck of BASIC and exploits greater parallelism than FWK. Figures 1.10 shows the pseudo-code for MWK.

```
// Starting with the root node execute the
// following code for each new tree level
forall attributes in parallel (dynamic scheduling)
    for each block of K leaves
        for each leaf i
            if (last block's i-th leaf not done) then
                wait
            evaluate attributes (E)
            if (last processor finishing on leaf i) then
                get winning attribute; form hash-probe (W)
                signal that i-th leaf is done

barrier
forall attributes in parallel (dynamic scheduling)
    for each leaf
        split attributes (S)
```

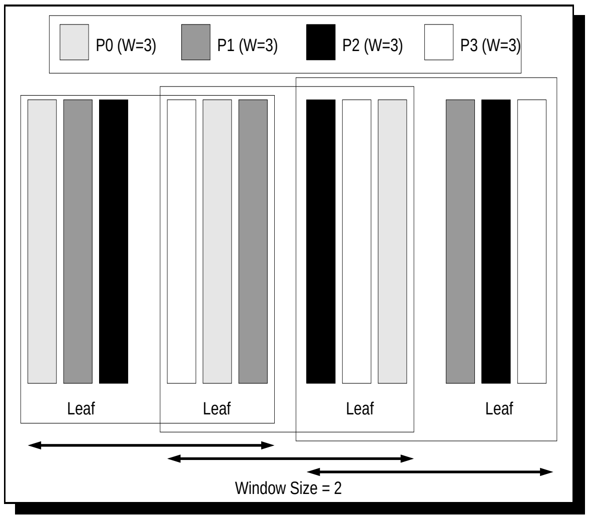


Figure 1.10: The MWK Algorithm (4 Processors, 3 Attributes)

Consider a current leaf frontier: $\{L0_1, R0_1, L0_2, R0_2\}$. With a window size of $K = 2$, not only is there parallelism available for fixed blocks $\{L0_1, R0_1\}$ and $\{L0_2, R0_2\}$ (used in FWK), but also between these two blocks, $\{R0_1, L0_2\}$. The MWK algorithm makes use of this additional parallelism.

This scheme is implemented by replacing the barrier per block of K leaves with a wait on a *conditional variable*. Before evaluating leaf i, a check is made whether the i-th leaf of the previous block has been processed. If not, the processor goes to sleep on the conditional variable. Otherwise, it proceeds with the current leaf. The last processor to finish the evaluation of leaf i from the previous block constructs the hash probe, and then signals the conditional variable, so that any sleeping processors are woken up.

It should be observed that the gain in available parallelism comes at the cost of increased lock synchronization per leaf (however, there is no barrier anymore). As in the FWK approach, the files are relabeled by eliminating the pure children. A larger K value would increase parallelism, and while the number of synchronizations remain about the same, it will reduce the average waiting time on the conditional variable. Like FWK, this scheme requires $2K$ files per attribute, so that each of the K leaves has separate files for each attribute and there is no read/write conflict.

Figure 1.10 shows an illustration of MWK. With a window size of 2, any two consecutive leaves are treated as a single block of schedulable attributes. The block shifts one leaf to the right each time ultimately encompassing all the leaves. The net effect is that all attributes in that level form one logical block. With this moving window scheme we see that all processors get equal (three units) amount of work, showing the benefit of MWK over BASIC and FWK.

1.4.3 Task Parallelism — The Subtree Algorithm (SUBTREE)

The data parallel approaches target the parallelism available among the different attributes. On the other hand the task parallel approach is based on the parallelism that exists in different sub-trees. Once the attribute lists are partitioned, each child can be processed in parallel. One implementation of this idea would be to initially assign all the processors to the tree root, and recursively partition the processor sets along with the attribute lists. Once a processor gains control of a subtree, it will work only on that portion of the tree. This approach would work fine if we have a full tree. In general, the decision trees are imbalanced and this static partitioning scheme can suffer from large load imbalances. We therefore use a dynamic subtree task parallelism scheme.

The pseudo-code and illustration for the dynamic SUBTREE algorithm is shown in Figure 1.11. To implement dynamic processor assignment to different subtrees, we maintain a queue of currently idle processors, called the *FREE* queue. Initially this queue is empty, and all processors are assigned to the root of the decision tree, and belong to a single group. One processor within the group is made the master (we chose the processor with the smallest identifier as the master). The master is responsible for partitioning the processor set.

```
SubTree (Processor Group P = {p1, p2, ..., px},
         Leaf Frontier L = {l1, l2, ..., ly})

   apply BASIC algorithm on L with P processors
   NewL = {l1, l2, ..., lm}   //new leaf frontier

   if (NewL is empty) then
      put self in FREE queue

   elseif (group master) then
      get FREE processors; NewP = {p1, p2, ..., pn}
      if (only one leaf remaining) then
         SubTree (NewP, l1)
      elseif (only one processor in group) then
         SubTree (p1, NewL)
      else   //multiple leaves and processors
         split NewL into L1 and L2
         split NewP into P1 and P2
         SubTree (P1, L1)
         SubTree (P2, L2)

      wakeup processors in NewP

   else //not the group master
      go to sleep
```

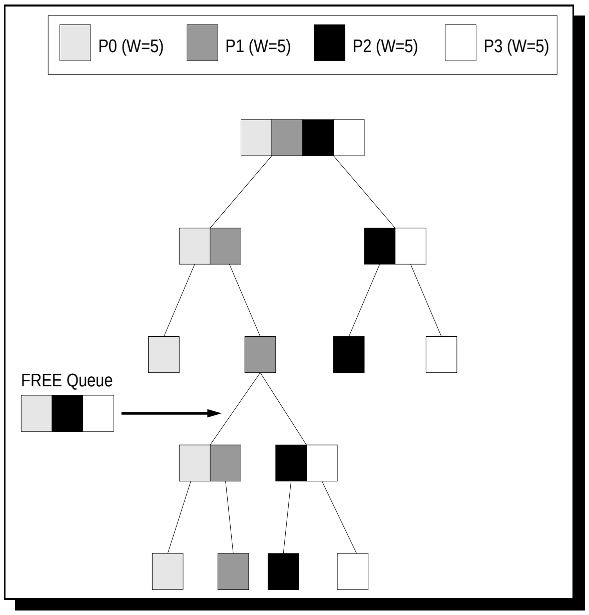


Figure 1.11: The SUBTREE Algorithm (4 Processors)

At any given point in the algorithm, there may be multiple processor groups working on distinct subtrees. Each group independently executes the following steps once the BASIC algorithm has been applied to the current subtree level. First, the new subtree leaf frontier is constructed. If there are no children remaining, then each processor inserts itself in the *FREE* queue, ensuring mutually exclusive access via locking. If there is more work to be done, then all processors except the master go to sleep on a conditional variable. The group master checks if there are any new arrivals in the *FREE* queue and grabs all free processors in the queue. This forms the new processor set.

There are three possible cases at this juncture. If there is only one leaf remaining, then all processors are assigned to that leaf. If there is only one processor in the previous group and there is no processor in the *FREE* queue, then it forms a group on its own and works on the current leaf frontier. Lastly, if there are multiple leaves and multiple processors, the group master splits the processor set and the leaves into two parts. The two newly formed processor sets become the new groups, and work on the corresponding leaf sets.

Finally, the master wakes up the all the relevant processors—those in the original group and those acquired from the *FREE* queue. Since there are P processors, there can be at most P groups, and since the attribute files for all of these must be distinct, this scheme requires up to $4P$ files per attribute.

Figure 1.11 shows how the algorithm works. We have four processors all of which are initially assigned to the root node. At the next level $P0$ and $P1$ work on the left child and the other two on the right. At the subsequent level each processor works independently on one portion of the emerging dynamic decision tree. In our example only $P1$'s subtree survives, and the remaining subtrees (actually leaves) turn out to be pure. Thus $P0, P2$, and $P3$ insert themselves into the *FREE* queue. When $P1$ expands the leaf to the next level there are two children. $P1$ checks the queue and grabs all idle processors. Thus there are now four available processors and two new leaves. $P0$ and $P1$ work on the left child and $P2$ and $P3$ work on the right. Finally at the next level all leaves turn out to be pure and the computation ends. This example illustrates how we achieve dynamic processor group assignments of good load-balance. As we can see each processor ends up working on an equal number of leaves ($W = 5$).

1.4.4 Discussion

We now qualitatively discuss the relative merits of each of the proposed algorithms. The MWK scheme eliminates the hash-probe construction bottleneck of BASIC via task pipelining. Furthermore, it fully exploits the available parallelism via the moving window mechanism, instead of using the fixed window approach of FWK. It also eliminates barrier synchronization completely. However, it introduces a lock synchronization per leaf per level. If the tree is bushy, then the increased synchronization could nullify the other benefits. A feature of MWK and FWK is that they exploit parallelism at a finer grain. The attributes in a K-block may be scheduled dynamically on any processor. This can have the effect of better load balancing compared to the coarser grained BASIC approach where a processor works on all the leaves for a given attribute. While MWK is essentially a data parallel approach, it utilizes some elements of task parallelism in the pipelining of the evaluation and hash probe construction stages.

The SUBTREE approach is also a hybrid approach in that it uses the BASIC scheme within each group. In fact we can also use FWK or MWK as the subroutine. The pros of SUBTREE are that it has only one barrier synchronization per level within each group and it has good processor utilization. As soon as a processor becomes idle it is likely to be grabbed by some active group. Some of the cons are that it is sensitive to the tree structure and may lead to excessive synchronization for the *FREE* queue, due to rapidly changing groups. Another disadvantage is that it requires more memory, because we need a separate hash probe per group.

As described above, our SMP algorithms ostensibly can create a large number of temporary files ($2Kd$ for MWK and $4dP$ for SUBTREE). However, it is possible to have a little more complex design so that lists for different attributes are combined into the same physical file. Such a design will reduce the number of temporary files to $2K$ for MWK and $4P$ for SUBTREE. The essential idea is to associate physical files for writing attribute lists with a processor (rather than with an attribute). In the split phase, a processor now writes all attribute lists to the same two physical files (for the left and right children). Additional bookkeeping data structures keep track of the start and end of different attribute lists in the file. These data structures are shared at the next tree level by all processors to locate the input attribute list for each dynamically assigned attribute. Note that this scheme does not incur additional synchronization overhead because a processor starts processing a new attribute list only after completely processing the one on hand.

1.5 Experimental Results

The primary metric for evaluating classifier performance is *classification accuracy* — the percentage of *test* examples (different from training examples used for building the classifier) that are correctly classified. The other important metrics are time to build the classifier and the *size* of the decision tree. The ideal goal for a decision tree classifier is to produce compact, accurate trees in a short time.

The accuracy and tree size characteristics of our SMP classifier are identical to SLIQ and SPRINT since they consider the same splits and use the same pruning algorithm. SLIQ's accuracy, execution time, and tree size have been compared with other classifiers such as CART (Breiman *et al.* 1984) and C4 (a predecessor of C4.5 (Quinlan 1993)). This performance evaluation, available in (Mehta, Agrawal, & Rissanen 1996), shows that compared to other classifiers SLIQ achieves comparable or better classification accuracy, but produces small decision trees and has small execution times. We, therefore, focus only on the classifier build time in our performance evaluation.

1.5.1 Experimental Setup

Machine Configuration Experiments were performed on two SMP machines with different configurations shown in Table 1.1. On both machines, each processor is a PowerPC-604 processor running at 112 MHz with a 16 KB instruction cache, a 16 KB data cache, and a 1 MB L2-Cache. These two machines represent two possible scenarios. With Machine A, the amount of memory is insufficient for training data, temporary files, and data structures to fit in memory. Therefore, the data will have to be read from and written to the disk for almost every new level of the tree. Machine B has a large memory relative to the size of the data. Therefore, all the temporary files created during the run are cached in memory. The first case is of greater interest to the database community and we present a detailed set of experiments for this configuration. However, due to the decreasing cost of RAM, the second configuration is also increasingly realizable in practice. We present this case to study the impact of large memories on the performance of our algorithms.

Machine Name	Number Processors	Main Memory	Disk Space Available	Access Type	Operating System
Machine A	4	128 MB	300 MB	local disk	AIX 4.1.4
Machine B	8	1 GB	2 GB	main-memory(cached)	AIX 4.1.5

Table 1.1: Machine Configurations

Function 2 (F2) - Group A:

$((\textbf{age} < 40) \ \wedge \ (50K \leq \textbf{salary} \leq 100K)) \vee$
$((40 \leq \textbf{age} < 60) \ \wedge \ (75K \leq \textbf{salary} \geq 125K)) \vee$
$((\textbf{age} \geq 60) \ \wedge \ (25K \leq \textbf{salary} \leq 75K))$

Function 7 (F7) - Group A:

$\textbf{disposable} > 0$
where $\textbf{disposable} = (0.67 \times (\textbf{salary} + \textbf{commission}))$
$- (0.2 \times \textbf{loan} \ - \ 20K)$

Figure 1.12: Classification Functions for Synthetic Data

Datasets An often used classification benchmark is STATLOG(Michie, Spiegelhalter, & Taylor 1994). Its largest dataset contains about 57,000 records. In our performance study we are interested in evaluating the SMP algorithms on large out-of-core data. We therefore use the synthetic benchmark proposed in (Agrawal *et al.* 1992) and used in several past studies. Example tuples in this benchmark have both continuous and categorical attributes. The benchmark gives several classification functions of varying complexity to generate synthetic databases. We present results for two of these functions, which are at the two ends of the complexity spectrum. Function 2 is a simple function to learn and results in fairly small decision trees, while Function 7 is the most complex function and produces large trees (see Table 1.2). Both these functions divide the database into two classes: Group A and Group B. Figure 1.12 shows the predicates for Group A for each function. For each of Functions 2 and 7, we try 3 different databases: 8 attributes with 1 million records, 32 attributes with 250K records, and 64 attributes with 125K records. The database parameters are shown in Table 1.2. The notation Fx-Ay-DzK is used to denote the dataset with function x, y attributes and $z \cdot 1000$ example records. The above choices allow us to investigate the effect of different data characteristics such as number of tuples and number of attributes. Also note that while the initial input size of the ASCII databases is around 60MB, the final input size after the creation of attribute lists is roughly 4 times more, i.e., around 240MB. Since Machine A has only 128MB main memory, the databases will be disk resident.

Dataset						Corresponding Tree	
Dataset Notation	Func.	No. Attr.	No. Tuple	Initial Size	Final Size	No. Levels	Max. No. Leaves/Level
F2-A8-D1000K	F2	8	1000K	61 MB	240MB	4	2
F2-A32-D250K	F2	32	250K	57.3 MB	225MB	4	2
F2-A64-D125K	F2	64	125K	56.6 MB	225MB	4	2
F2-A128-D64K	F2	128	64K	55.6 MB	225MB	4	2
F7-A8-D1000K	F7	8	1000K	61 MB	240MB	60	4662
F7-A32-D250K	F7	32	250K	57.3 MB	225MB	59	802
F7-A64-D125K	F7	64	125K	56.6 MB	225MB	55	384
F7-A128-D64K	F7	128	64K	55.6 MB	225MB	73	194

Table 1.2: Dataset Characteristics

Algorithms Our initial experiments confirmed that MWK was indeed better than BASIC as expected, and that it performs as well or better than FWK. Thus, we will only present the performance of MWK and SUBTREE.

We experimented with window sizes of 2, 4 and 8 for MWK. A larger window size implies more overhead on the file creation and managing related data structures. On the other head, a smaller window size may not have enough parallelism, especially when there are many processors and relatively few attributes. We found for our experiments a window size of 4 to be a good overall choice unless the ratio of the number of attributes to the number of processors is small (less than 2) and in that case we use a window size of 8 (which performs better than a window size of 4 by as much as 9%). In general, a simple rule of thumb for the window size is that if the number of attributes is at least twice the number of processors (which is typically the case for a real world run), a window size of 4 should be chosen. In the rare case when $d/P < 2$, we choose the smallest W such that $W * d/P \geq 8$.

1.5.2 Initial Setup and Sort Time

Table 1.3 shows the uniprocessor time spent in the initial attribute list creation phase (*setup* phase), as wells as the time spent in one-time sort of the attribute lists for the continuous attributes (*sort* phase). The time spent in these two phases as a fraction of total time to build a classifier tree depends on the complexity of the input data for which we are building the classification model. For simple datasets such as F2, it can be significant, whereas it is negligible for complex datasets such as F7.

We have not focussed on parallelizing these phases, concentrating instead on the more challenging build phase. There is much existing research in parallel sorting on SMP machines (Bitton *et al.* 1984). The creation of attribute lists can be speeded up by essentially using multiple input streams and merging this phase with the sort phase. In our implementation, the data scan to create attribute lists is sequential, although we write attribute lists in parallel. Attribute lists are sorted in parallel by assigning them to different processors. When we present the speedup graphs in the next section, we will show the speedups separately for the build phase as well as for the total time (including initial setup and sort times). There is obvious scope for improving the speedups for the total time.

Dataset	Setup Time (seconds)	Sort Time (seconds)	Total Time (seconds)	Setup %	Sort %
F2-A8-D1000K	721	633	3597	20.0%	17.6%
F2-A32-D250K	685	598	3584	19.1%	16.6%
F2-A64-D125K	705	626	3665	19.2%	17.1%
F2-A128-D64K	706	624	3802	18.6%	16.4%
F7-A8-D1000K	989	817	23360	4.2%	3.5%
F7-A32-D250K	838	780	24706	3.4%	3.2%
F7-A64-D125K	672	636	22664	3.0%	2.8%
F7-A128-D64K	707	719	25021	2.8%	2.9%

Table 1.3: Sequential Setup and Sorting Times

1.5.3 Parallel Build Performance: Local Disk Access

We consider four main parameters for performance comparison: 1) number of processors, 2) number of attributes, 3) number of example tuples, and 4) classification function (Function 2 or Function 7). We first study the effect of varying these parameters on the MWK and SUBTREE algorithms on Machine A, which has less main memory than the database size, so that disk I/O is required at each level while building the tree.

Figure 1.13 shows the parallel performance and speedup of the two algorithms as we vary the number of processors for the two classification functions F2 and F7, and using the dataset with eight attributes and one million records (*A8-D1000K*). Figures 1.14, 1.15 and 1.16 show similar results for datasets *A32-D250K*, *A64-D125K*, and *A128-D64K* respectively. The SUBTREE times for 4 processors are missing in *A128-D64K* since we exceeded the number of open files limit in Unix. The speedup chart in the bottom right part of each figure shows the speedup of the total time (including setup and sort time), while the speedup chart to the left of it and the two top bar charts show only the build time (excluding setup and sort time).

Considering the build time only, the speedups for both algorithms on 4 processors range from 2.97 to 3.32 for function F2 and from 3.25 to 3.86 for function F7. For function F7, the speedups of total time for both algorithms on 4 processors range from 3.12 to 3.67. The important observation from these figures is that both algorithms perform quite well for various datasets. Even the overall speedups are good for complex datasets generated with function F7. As expected, the overall speedups for simple datasets generated by function F2, in which build time is a smaller fraction of total time, are relatively not as good (around 2.2 to 2.5 on 4 processors). These speedups can be improved by parallelizing the setup phase more aggressively.

MWK's performance is mostly comparable or better than SUBTREE. The difference ranges from 8% worse than SUBTREE to 22% better than SUBTREE. Most of the MWK times are within 10% better than SUBTREE.

An observable trend is that having greater number of processors tends to favor SUBTREE. In other words, the advantage of MWK over SUBTREE tends to decrease as the number of processors increases. This is can be seen from figures for both $F2$ and $F7$ by comparing the build times for the two algorithms first with 2 processors, then with 4 processors. This is because after about $\log P$ levels of the tree growth, the only synchronization overhead for SUBTREE, before any processor becomes free, is that each processor checks the FREE queue once per level. On the other hand, for MWK, there will be relatively more processor synchronization overhead, as the number of processors increases, which includes acquiring attributes, checking on conditional variables, and waiting on barriers. As part of future work we plan to compare these algorithms on larger SMP configurations.

1.5.4 Parallel Build Performance: Main-Memory (Cached) Access

We next compare the parallel performance and speedups of the algorithms on Machine B. This configuration has 1 GB of main-memory available. Thus after the very first access the data will be cached in main-memory, leading to fast access times. Machine B has 8 processors. Figures 1.17, 1.18, and 1.19 show three sets of timing and speedup charts for the *A8-D1000K*, *A32-D250K* and *A64-D125K* datasets, on Functions 2 and 7, and on 1, 2, 4 and 8 processors.

Considering the build time only, the speedups for both algorithms on 8 processors range from 5.46 to 6.37 for function F2 and from 5.36 to 6.67 (and at least 6.22 with 32 or 64 attribute) for function F7. For

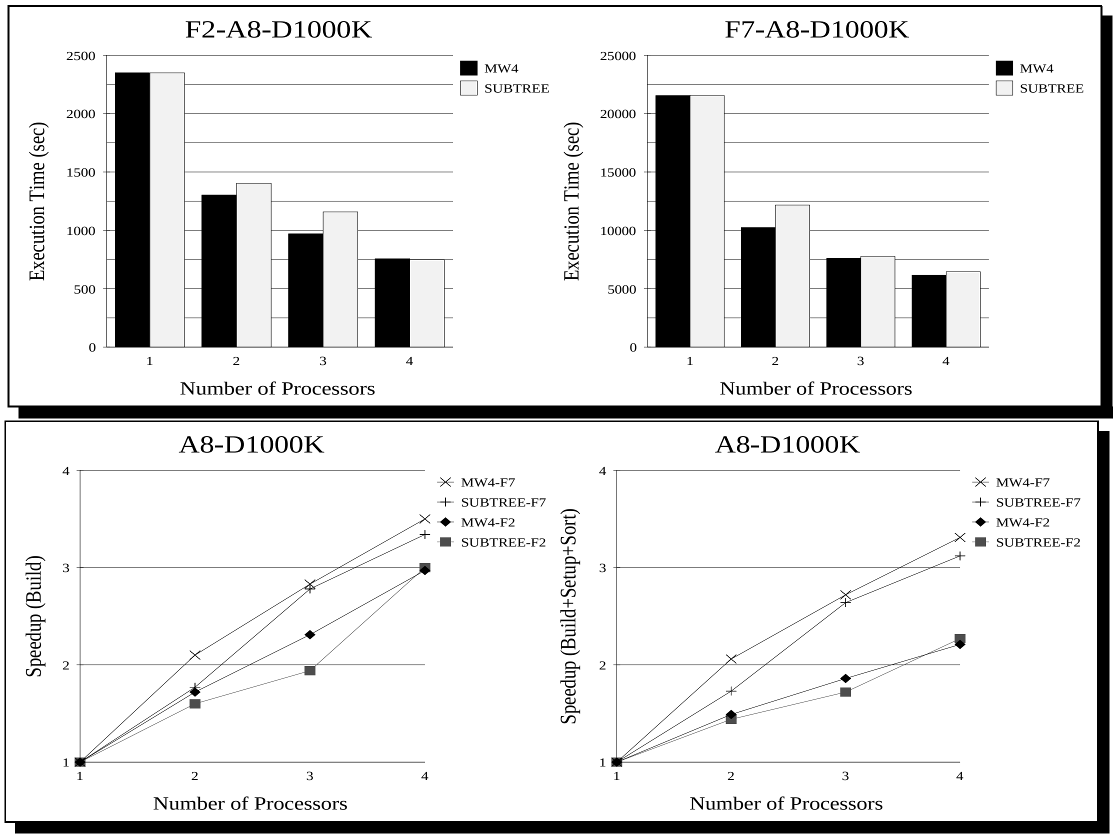


Figure 1.13: Local Disk Access: Functions 2 and 7; 8 Attributes; 1000K Records

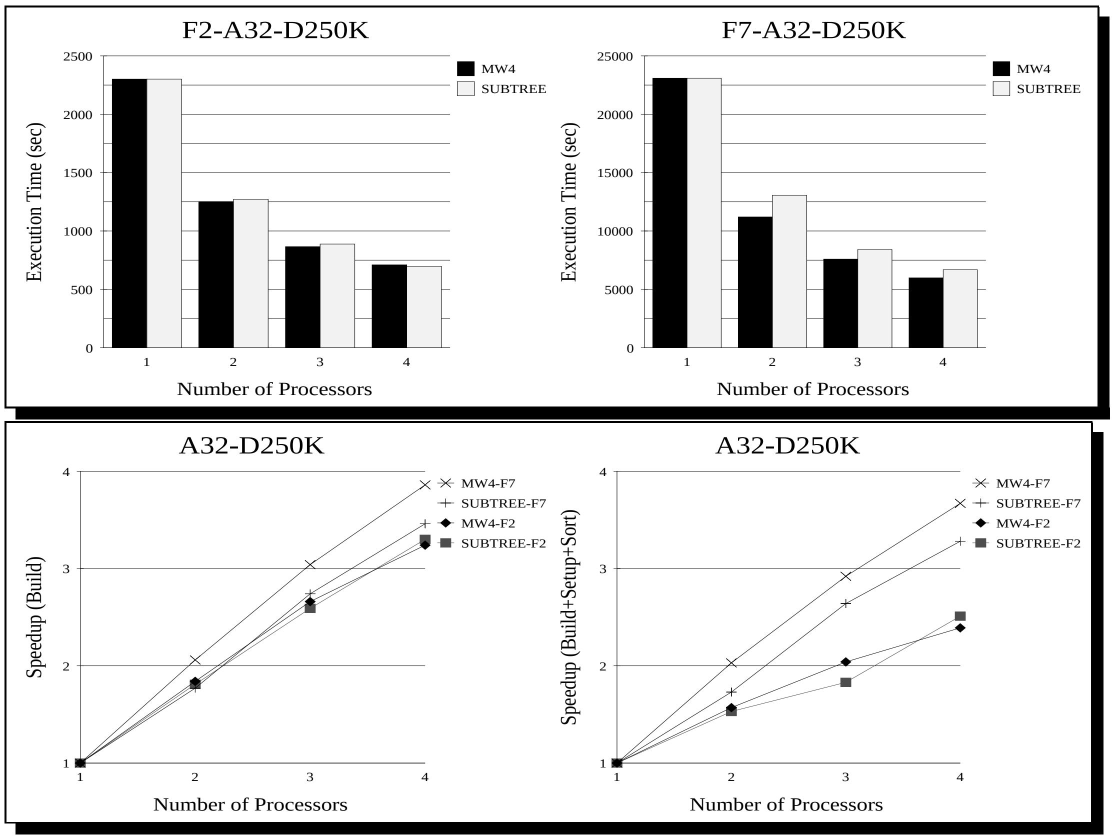


Figure 1.14: Local Disk Access: Functions 2 and 7; 32 Attributes; 250K Records

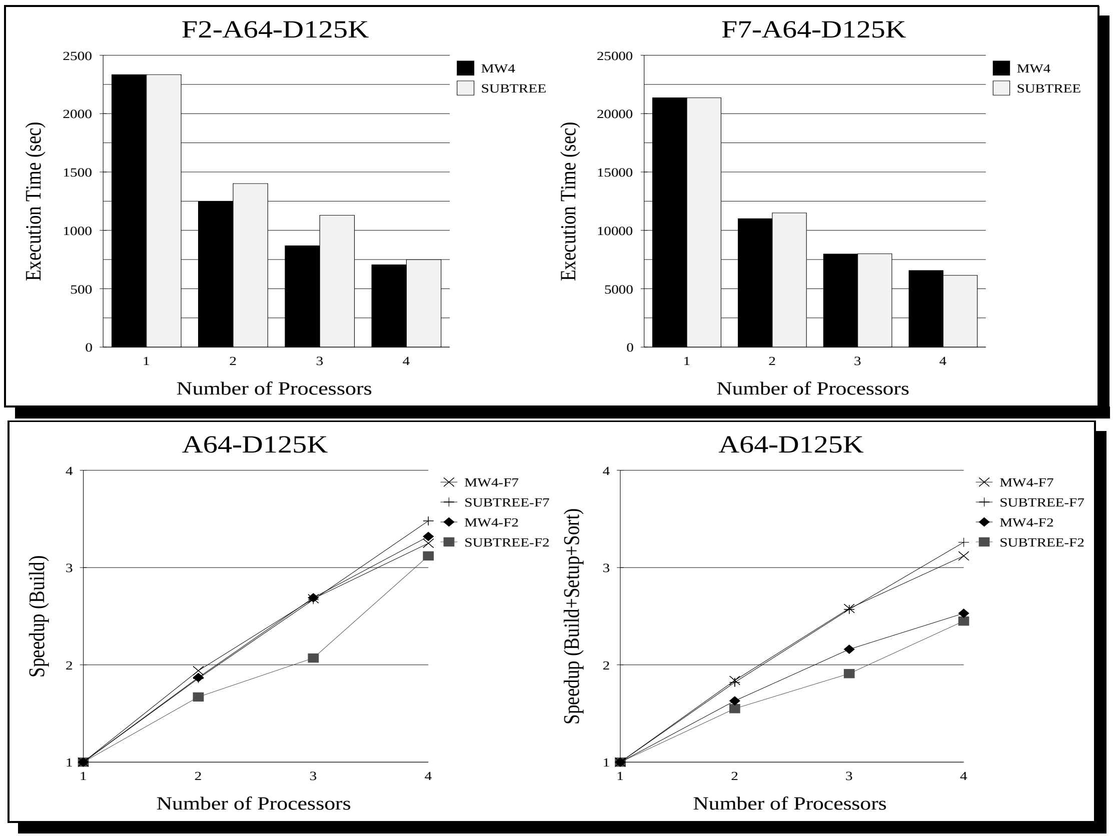


Figure 1.15: Local Disk Access: Functions 2 and 7; 64 Attributes; 125K Records

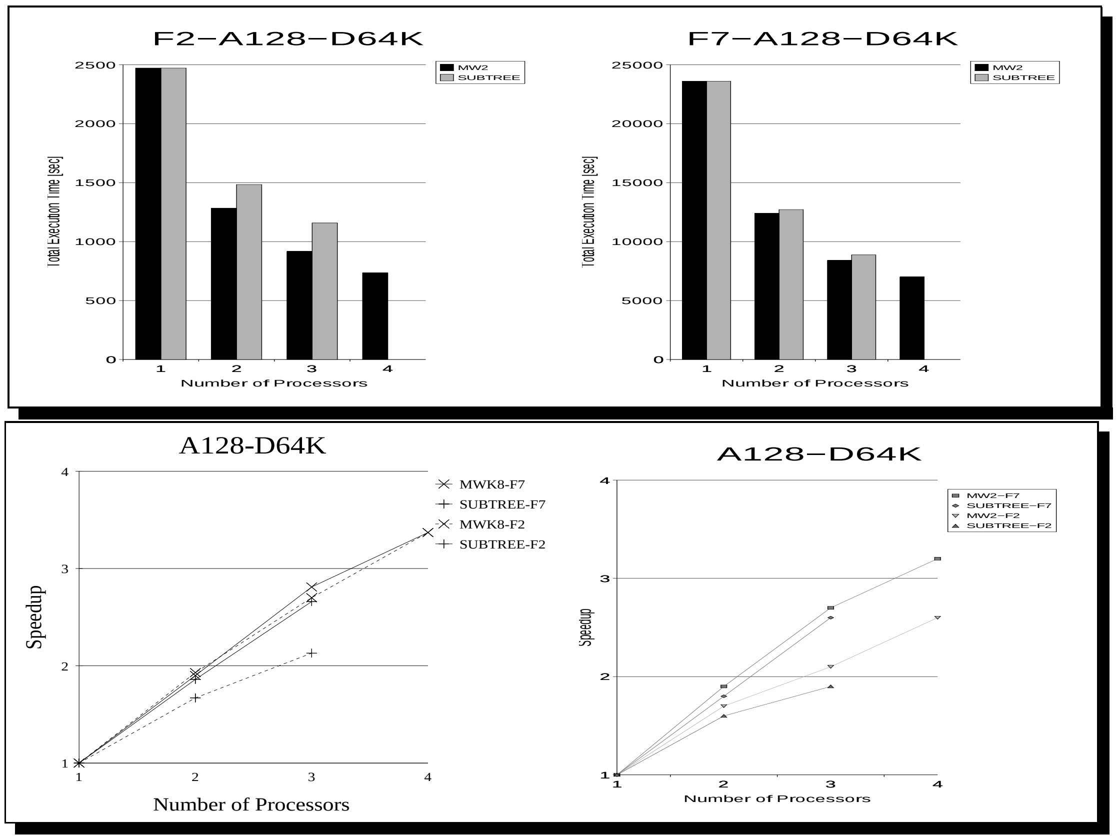


Figure 1.16: Local Disk Access: Functions 2 and 7; 128 Attributes; 64K Records

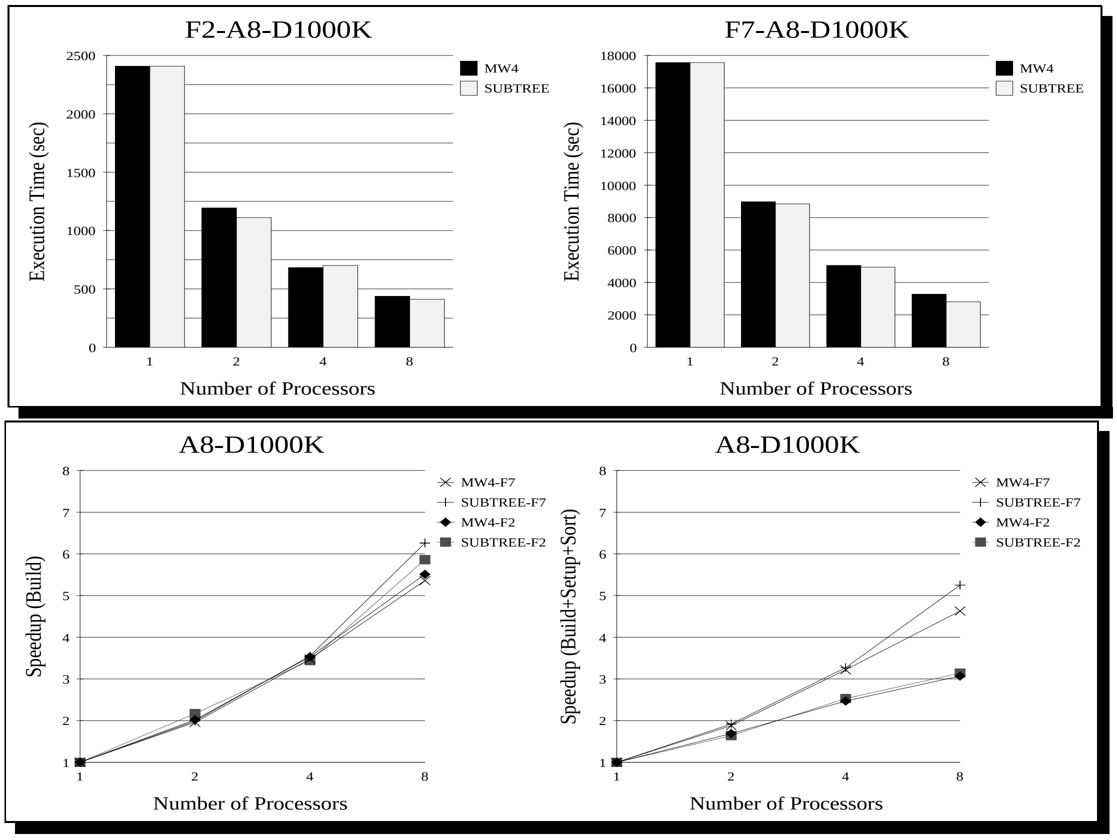


Figure 1.17: Main-memory Access: Functions 2 and 7; 8 Attributes; 1000K Records

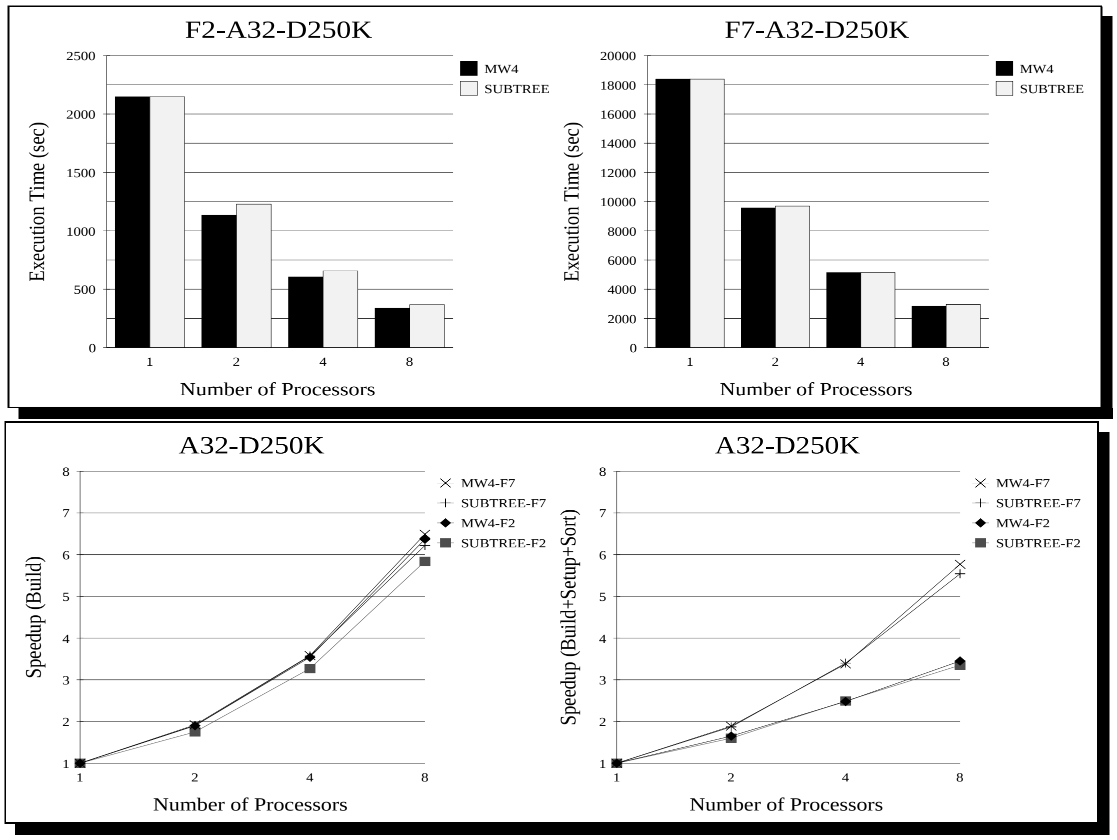


Figure 1.18: Main-memory Access: Functions 2 and 7; 32 Attributes; 250K Records

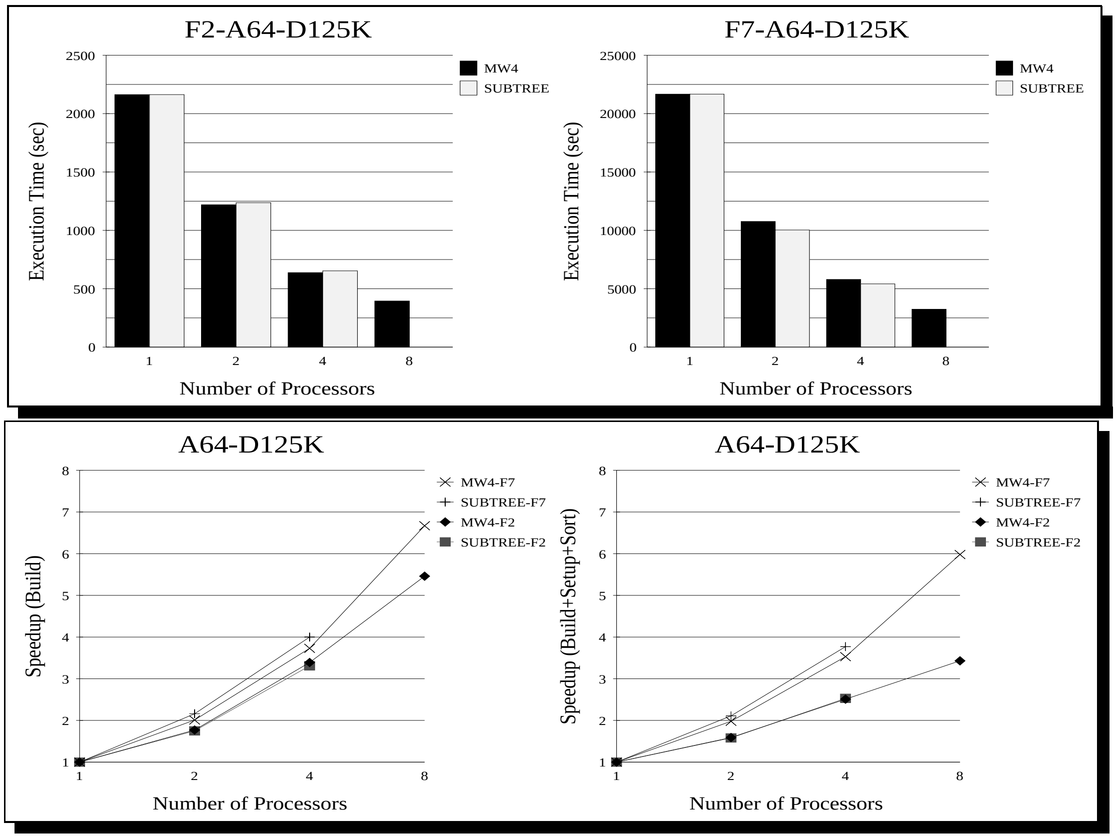


Figure 1.19: Main-memory Access: Functions 2 and 7; 64 Attributes; 125K Records

function F7, the speedups of total time for both algorithms on 8 processors range from 4.63 to 5.77 (and at least 5.25 for 32 or 64 attributes). Again, the important observation from these figures is that both algorithms perform very well for various datasets even up to 8 processors. The advantage of MWK over SUBTREE is more visible for the simple function F2. The reason is that F2 generates very small trees with 4 levels and a maximum of 2 leaves in any new leaf frontier. Around 40% of the total time is spent in the root node, where SUBTREE has only one process group. Thus on this dataset SUBTREE is unable to fully exploit the inter-node parallelism successfully. MWK is the winner because it not only overlaps the $\mathcal{E}$ and $\mathcal{W}$ phases, but also manages to reduce the load imbalance.

The overall trends observable from these figures are similar to those for the disk configuration. First, shallow trees (e.g., generated by F2) tend to hurt SUBTREE, 2) Greater number of processors tends to favor SUBTREE more, 3) Having a small number of attributes tends to hurt MWK.

1.6 Conclusions

We presented parallel algorithms for building decision-tree classifiers on SMP systems. The proposed algorithms span the gamut of data and task parallelism. The MWK algorithm uses data parallelism from multiple attributes, but also uses task pipelining to overlap different computing phases within a tree node, thus avoiding potential a sequential bottleneck for the hash-probe construction in the split phase. The MWK algorithm employs a conditional variable, instead of a barrier, among leaf nodes to avoid unnecessary processor blocking time. It also exploits dynamic assignment of attribute files to a fixed set of physical files, which maximizes the number of concurrent accesses to disk without file interference. The SUBTREE algorithm uses recursive divide-and-conquer to minimize processor interaction, and assigns free processors dynamically to busy groups to achieve load balancing.

Experiments show that both algorithms achieve good speedups in building the classifier on a 4-processor SMP with disk configuration and on an 8-processor SMP with memory configuration, for various numbers of attributes, various numbers of example tuples of input databases, and various complexities of data models. The performance of both algorithms are comparable, but MWK has a slight edge on the SMP configurations we looked at. These experiments demonstrate that the important data mining task of classification can be effectively parallelized on SMP machines.

References

Agrawal, R.; Ghosh, S.; Imielinski, T.; Iyer, B.; and Swami, A. 1992. An interval classifier for database mining applications. In *18th VLDB Conference*.

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

Alsabti, K.; Ranka, S.; and Singh, V. 1998. Clouds: A decision tree classifier for large datasets. In *4th Int'l Conference on Knowledge Discovery and Data Mining*.

Bitton, D.; DeWitt, D.; Hsiao, D. K.; and Menon, J. 1984. A taxonomy of parallel sorting. *ACM Computing Surveys* 16(3):287–318.

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification and Regression Trees*. Belmont: Wadsworth.

Catlett, J. 1991. *Megainduction: Machine Learning on Very Large Databases*. Ph.D. Dissertation, University of Sydney.

Chan, P. K., and Stolfo, S. J. 1993a. Experiments on multistrategy learning by meta-learning. In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, 314–323.

Chan, P. K., and Stolfo, S. J. 1993b. Meta-learning for multistrategy and parallel learning. In *Proc. Second Intl. Workshop on Multistrategy Learning*, 150–165.

Fifield, D. J. 1992. Distributed tree construction from large data-sets. Bachelor's Honours Thesis, Australian National University.

Gehrke, J.; Ramakrishnan, R.; and Ganti, V. 1998. Rainforest - a framework for fast decision tree construction of large datasets. In *24th Intl. Conf. Very Large Databases*.

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Morgan Kaufmann.

IBM Corp. *See http://www.rs6000.ibm.com/hardware/largescale/index.html, IBM RS/6000 SP System*.

James, M. 1985. *Classificaton Algorithms*. Wiley.

Joshi, M.; Karypis, G.; and Kumar, V. 1998. ScalParC: A scalable and parallel classification algorithm for mining large datasets. In *Intl. Parallel Processing Symposium*.

Lewis, B., and Berg, D. J. 1996. *Threads Primer*. New Jersey: Prentice Hall.

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

Markatos, E., and LeBlanc, T. 1994. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 5(4).

Mehta, M.; Agrawal, R.; and Rissanen, J. 1996. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*.

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

NASA Ames Research Center. 1992. *Introduction to IND Version 2.1*, GA23-2475-02 edition.

Polychronopoulos, C. D., and Kuck, D. J. 1987. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers* C-36(12):1425–1439.

Quinlan, J. R. 1979. Induction over large databases. Technical Report STAN-CS-739, Stanfard University.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning* 1:81–106.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufman.

Rastogi, R., and Shim, K. 1998. Public: A decision tree classifier that integrates building and pruning. In *24th Intl. Conf. Very Large Databases*.

Rissanen, J. 1989. *Stochastic Complexity in Statistical Inquiry*. World Scientific Publ. Co.

Shafer, J.; Agrawal, R.; and Mehta, M. 1996. Sprint: A scalable parallel classifier for data mining. In *22nd VLDB Conference*.

Sreenivas, M.; Alsabti, K.; and Ranka, S. 1999. Parallel out-of-core divide and conquer techniques with application to classification trees. In *13th International Parallel Processing Symposium*.

Tang, P., and Yew, P.-C. 1986. Processor self-scheduling for multiple nested parallel loops. In *International Conference On Parallel Processing*.

Weiss, S. M., and Kulikowski, C. A. 1991. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufman.

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

Zaki, M. J.; Ho, C.-T.; and Agrawal, R. 1999. Parallel classification for data mining on shared-memory multiprocessors. In *15th IEEE Intl. Conf. on Data Engineering*.