Parallel Classification for Data Mining on Shared-Memory Multiprocessors

Mohammed J. Zaki* Ching-Tien Ho Rakesh Agrawal
IBM Almaden Research Center, San Jose, CA 95120

Abstract

We present parallel algorithms for building decision-tree classifiers on shared-memory multiprocessor (SMP) systems. The proposed algorithms span the gamut of data and task parallelism. The data parallelism is based on attribute scheduling among processors. This basic scheme is extended with task pipelining and dynamic load balancing to yield faster implementations. The task parallel approach uses dynamic subtree partitioning among processors. We evaluate the performance of these algorithms on two machine 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 cache the whole data. This performance evaluation shows that the construction of a decision-tree classifier can be effectively parallelized on an SMP machine with good speedup. 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.

1 Introduction

Classification is recognized to be a primary data mining task [2] [9]. The input to a classification system consists of a set of example tuples, called a *training set*, where each tuple consists of several *attributes*. Attributes can be *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. One of the attributes, called the *class* attribute, indicates the class to which each example belongs. The goal of classification is to induce a model from the training set, that can be used to predict the class of a new tuple that does not have a class label. Classification has applications in diverse fields such as retail target marketing, customer retention, fraud detection, and medical diagnosis [15].

Amongst the several classification methods proposed over the years [20] [15], decision trees are particularly suited for data mining applications [14]. Decision trees can be constructed relatively fast compared to other methods and they are easy to interpret [17]. Moreover, trees can be converted into SQL statements that can be used to access databases efficiently [1]. Finally, decision-tree classifiers obtain similar, and often better, accuracy when compared to other classification methods [15].

Prior to interest in classification for database-centric data mining, it was tacitly assumed that the training sets could fit in memory. Recent work has targeted the massive training sets usual in

*Current Affiliation: Department of Computer Science, University of Rochester, Rochester, NY 14627, zaki@cs.rochester.edu.

data mining. Developing classification models using larger training sets can enable the development of higher accuracy models. Various studies have confirmed this hypothesis [5] [6] [7]. Examples of recent classification systems that can handle disk-resident data include SLIQ [14] and its extension SPRINT [18].

A continuing trend in data management 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 research on parallel classification has been focussed on distributed-memory (also called shared-nothing) parallel machines. In such a machine, each processor has private memory and local disks, and communicates with other processors only via passing messages. An early work in this area examined parallelizing the ID3 decision-tree classifier [10]. It assumed that the entire dataset could fit in memory and did not address issues such as disk I/O. The algorithms required processor communication to evaluate any test condition for a decision-tree node, limiting the number of possible tests the algorithm could consider. The Darwin toolkit from Thinking Machines, a distributed-memory parallel machine, also contained a parallel implementation of the CART decision-tree classifier [4]. The details of this parallelization are not available in published literature. The SPRINT algorithm mentioned earlier was also parallelized on the IBM SP2 parallel distributed-memory machine and was shown to achieve good scaleup and speedup [18].

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 even parallel distributed-memory machines are 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 [11]. 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. The programming architecture on these machines is quite different from that on distributed-memory machines.

This paper presents parallel algorithms for building decision-tree classifiers on shared-memory systems, the first such study to the best of our knowledge. The algorithms we propose span the gamut of data and task parallelism. The data parallelism is based on attribute scheduling among processors—scheduling work associated with different attributes to different processors. This basic scheme is extended with task pipelining and dynamic load balancing to yield more efficient 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

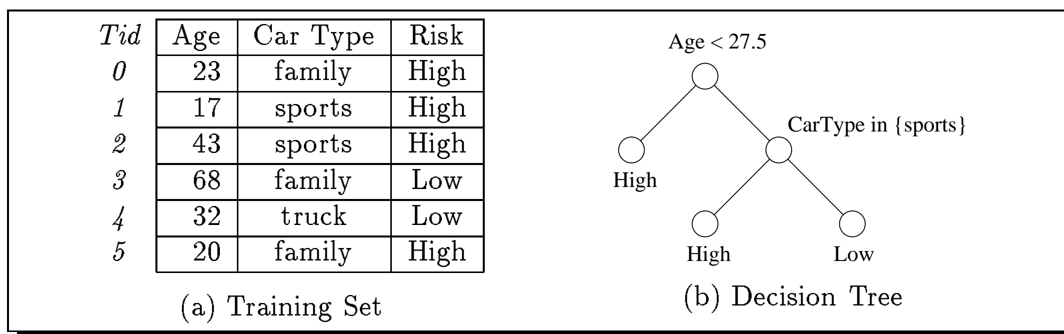
and from 3.07 to 5.98 for the total time on an 8-processor SMP.

It is fair to ask at this stage why one could not simply take the distributed-memory implementation of, say, SPRINT and adapt it to run on SMP machines. While it is feasible to do so, the performance penalty can be substantial. Essentially, we would be simulating a distributed-memory machine on an SMP machine by partitioning in software the main memory and disk storage. The result would be inefficient use of memory due to artificial replication of data structures. It would make the load balancing unnecessarily complex. The communication software for the distributed-memory system would also need efficient mapping into SMP communication primitives. We will discuss these points in detail later in the paper. Finally, the SMP architecture offers new challenges and trade-offs that are worth investigating in their own right.

The rest of the paper is organized as follows. In Section 2 we review how a decision-tree classifier, specifically SPRINT, is built on a uniprocessor machine. This section is adapted from [18]. Section 3 describes our new SMP algorithms based on various data and task parallelization schemes. We give experimental results in Section 4 and conclude with a summary in Section 5.

2 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(b) shows a decision-tree classifier developed from the training set shown in Figure 1(a).



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

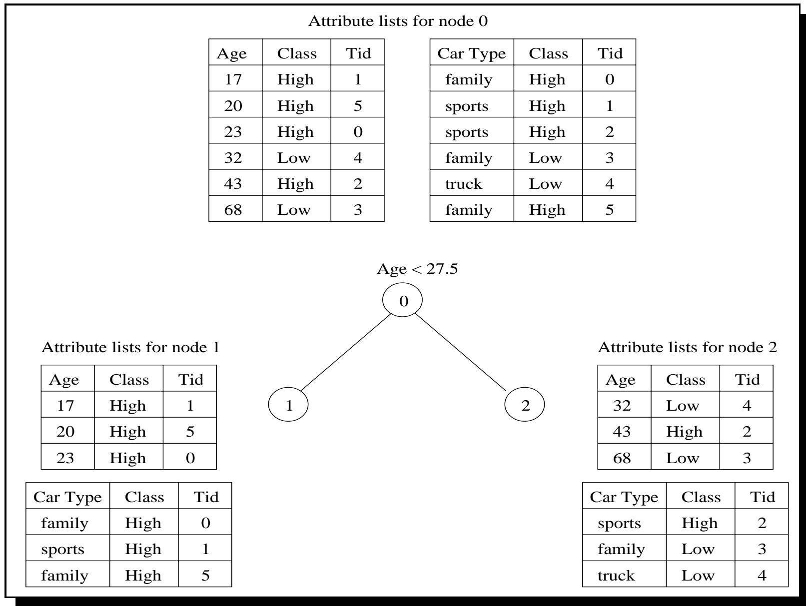
(a) Training Set

(b) Decision Tree

Figure 1: Car insurance example.

A decision-tree classifier is usually built in two phases [4] [17]: a growth phase and a prune phase. The tree is grown using a divide and conquer approach. Given a set of training examples S as input, there are two cases to consider. If all examples in S belong to a single class, then S becomes a leaf. On the other hand, if S contains a mixture of examples from different classes, the input is partitioned into subsets that tend towards a single class. The partitioning is based on a test on an attribute. The splits are usually binary as they lead to more accurate trees [4]. The above process is recursively applied using the partitioned datasets as inputs.

The tree thus built can "overfit" the data. The goal of classification is to predict new unseen cases. The prune phase generalizes the tree by removing subtrees corresponding to statistical noise



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

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 2: Splitting a node's attribute lists.

or variation that may be particular only to the training data. This phase 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 prune phase [14].

The performance of the tree growth phase depends on two factors: 1) how to find split points that define node tests, and 2) having chosen a split point, how to partition the data. We now describe how the above two steps are handled in serial SPRINT [18]. It builds the tree in a breadth-first order and uses a one-time pre-sorting technique to reduce the cost of continuous attribute evaluation. In contrast, the older, well-known CART [4] and C4.5 [17] classifiers grow trees depth-first and repeatedly sort the data at every node. Our SMP classifier uses the same basic computations as serial SPRINT, but the details of how computations are orchestrated are different.

2.1 Attribute Lists

SPRINT initially creates a disk-based *attribute list* for each attribute in the data. Each entry in the list consists of an attribute value, a class label, and a tuple identifier (tid) of the corresponding data tuple. We will refer to the elements of the attribute lists as "records" to avoid confusing them with "tuples" in the training data. Initial lists for continuous attributes are sorted by attribute value. The lists for categorical attributes stay in unsorted order. All the attribute lists are initially associated with the root of the classification tree. As the tree is grown and split into subtrees, the attribute lists are also split. By preserving the order of records in the partitioned lists, no re-sorting is required. Figure 2 shows an example of the initial sorted attribute lists associated with the root of the tree for the training data in Figure 1 and also the resulting partitioned lists for the two children.

Histograms SPRINT uses histograms tabulating the class distributions of the records in an attribute list to evaluate split points for the list. For continuous attributes, two histograms are maintained: C_{below} keeps the distributions for records that have already been processed, and C_{above} for the rest. For categorical attributes one histogram, called the *count matrix*, is maintained for the class distribution of a given attribute list. There is one set of histograms for each attribute list assigned to every node in the tree. C_{above} and *count matrix* are initialized as attribute lists are created. C_{below} and C_{above} are incrementally updated, as an attribute list for a continuous attribute is processed.

2.2 Finding Good Split Points

The form of the split test 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 $X \subset domain(A)$.

The split test is chosen to "best" divide the training records associated with a node. The "goodness" of the split 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 for this purpose. For a data set S containing examples from n classes, $gini(S)$ is defined to be $gini(S) = 1 - \sum p_j^2$ where p_j is the relative frequency of class j in S [4]. For continuous attributes, the candidate split points are mid-points between every two consecutive attribute values in the training data. 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 is used.) Note that the histograms contain the necessary information to evaluate a gini index.

2.3 Splitting the Data

Having found the winning split test for a node, the node is split into two children and the node's attribute lists are divided into two (Figure 2). The attribute list for the winning attribute (*Age* in our example at the root node) is partitioned simply by scanning the list and applying the split test to each record. For the remaining "losing" attributes (*CarType* in our example) more work is needed. While dividing the winning attribute a probe structure (bit mask or hash table) on the *tids* is created, noting the child where a particular record belongs. While splitting other attribute lists, this structure is consulted for each record to determine the child where this record should be placed. If the probe structure is too big to fit in memory, the splitting takes multiple steps. In each step only a portion of the attribute lists are partitioned.

Avoiding multiple attribute lists 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. Rather than creating a separate attribute list for each attribute for every node, only four physical files per attribute are needed. Since the splits are binary, there is one attribute file for all leaves that are "left" children and one file for all leaves that are "right" children. There are two more files

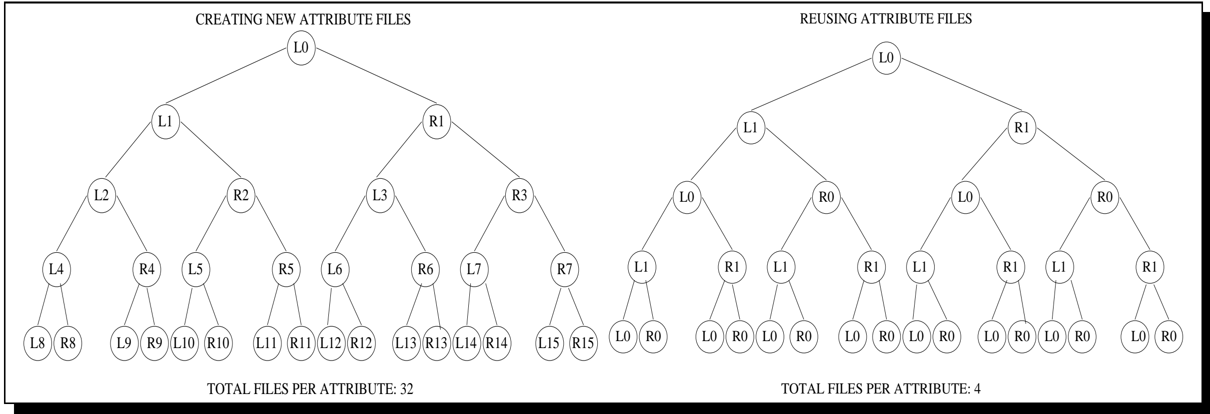


Figure 3: Avoiding multiple attribute files.

per attribute that serve as alternates. This scheme is illustrated in Figure 3. In fact, it is possible to combine the records of different attribute lists into one physical file, thus requiring a total of 4 physical files.

2.4 Parallel SPRINT on a Distributed-Memory Machine

The parallel implementation of SPRINT on IBM SP2 distributed-memory machine [18] is based on what can be called *record data parallelism*. Briefly, each processor is responsible for processing roughly $1/P$ fraction of each attribute list. All processors synchronously construct one global decision tree, *one node at a time*.

First, the training examples are distributed equally among all the processors. Each processor then generates its own attribute list partitions in parallel by projecting each attribute from training examples it was assigned. Lists for continuous attributes are further globally sorted using a parallel sort [8] and repartitioned amongst processors into equal-sized contiguous sorted sections. Lists for categorical attributes are not sorted/repartitioned. Note that the different attribute lists belonging to a processor no longer correspond to the same set of *tids*.

Finding split points in parallel SPRINT is similar to the serial case except that the class histograms are first created locally for the attribute-list partitions owned by a processor and then communicated between processors so that each processor has a global view of the class distributions. For a continuous attribute, each processor finds the the best local split point using its local partition of the attribute list and then the processors communicate to determine the overall best split point for the attribute. The best split point for a categorical attribute is determined by a co-ordinator using the global count matrix.

Each processor is responsible for splitting its own attribute-list partitions in the split phase. However, since different attribute lists belonging to a processor do not correspond to the same set of *tids*, each processor needs to build a probe structure covering all the *tids* of records in an attribute list. A processor accomplishes this by exchanging its own *tids* of the winning attribute list and their corresponding left/right labels with that of all other processors.

2.5 Why Not Simply Map the Distributed-Memory SPRINT?

Here, we examine why it is not attractive from a performance viewpoint to take the distributed-memory parallel SPRINT algorithm platform and map it to the SMP systems. This mapping would entail simulating a distributed-memory system on an SMP machine and creating software controlled memory and disk partitions. Artificial partitioning of SMP memory to run distributed-memory algorithm would result in very inefficient use of SMP memory. Several data structure will be unnecessarily replicated (e.g. probe structure) when a shared common data structure would have been sufficient. The communication software of the distributed-memory system will also need to be mapped into SMP communication primitives. In general, this approach is bringing in all the complexity of programming a distributed-memory system without benefiting from any of its scalability advantages.

Specifically, we argue that the record data parallelism approach of parallel SPRINT is not well-suited for an SMP system and will propose parallelization based on partitioning attributes. In parallel SPRINT, the processors simultaneously grow the same node of a tree by working in parallel on disjoint portions of the attribute list of a continuous attribute to find the best split point in their respective portions. While these portions start off being of equal size, as the tree grows and attribute lists are split, the sizes can start varying substantially resulting in load imbalances. Performing dynamic load balancing to correct this imbalance is quite complex and likely to introduce substantial overhead. The data parallelism based on attribute partitioning that we will describe next allows a processor to process all the attribute lists at the same level of tree for a given attribute. Assignment of attributes to processors is dynamic—as soon as a processor becomes free, it grabs the next available attribute simplifying load balancing.

3 Parallel Classification on Shared-memory Systems (SMP)

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 [14], as it requires access to only the decision-tree grown in the training phase.

3.1 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: (i) Evaluate split points for each attribute (denoted as step $\mathcal{E}$); (ii) Find the winning split-point and construct a probe structure using the attribute list of the winning attribute (denoted as step $\mathcal{W}$); and (iii) 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 [12]. 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. This approach exploits the intra-node parallelism, i.e. that available within a decision tree node. We use *attribute data parallelism*: the attributes are divided equally among the different processors so that each processor is responsible for $1/P$ attributes.

Task parallel The task parallelism exploits the inter-node parallelism: different portions of the decision tree are built in parallel among the processors.

3.2 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 partitioning approach.

3.2.1 The Basic Scheme (BASIC)

Figure 4 shows the pseudo-code for the BASIC scheme. A barrier represents a point of synchronization. While a full tree is shown here, the tree generally may have a sparse irregular structure. At each level a processor evaluates the assigned attributes, which is followed by a barrier.

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 [19], i.e., there is lock synchronization per attribute.[1]

[1] 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 [16] 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 [13], where

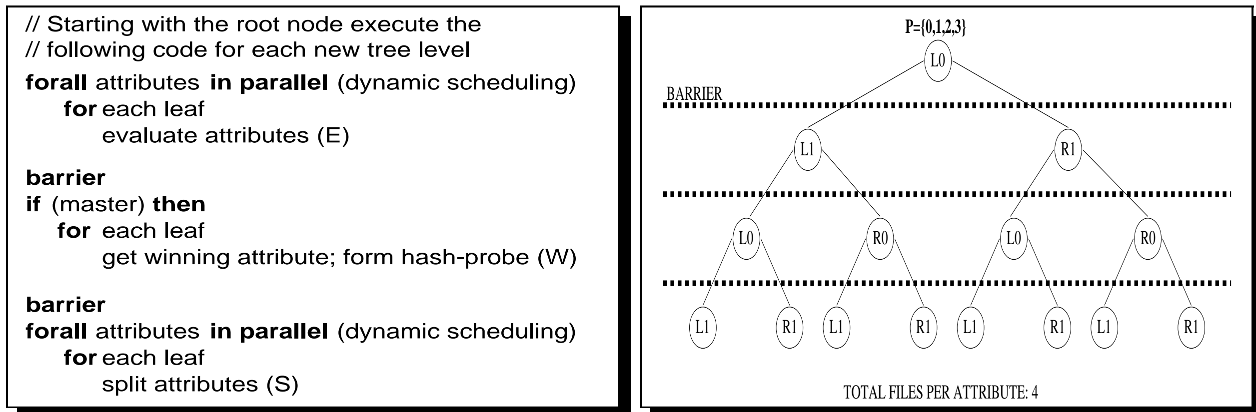


Figure 4: The BASIC algorithm.

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, they 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 for 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

attention is paid to the location of the attribute lists so that accesses to local attribute lists are maximized.

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.

3.2.2 The Fixed-Window-K Scheme (FWK)

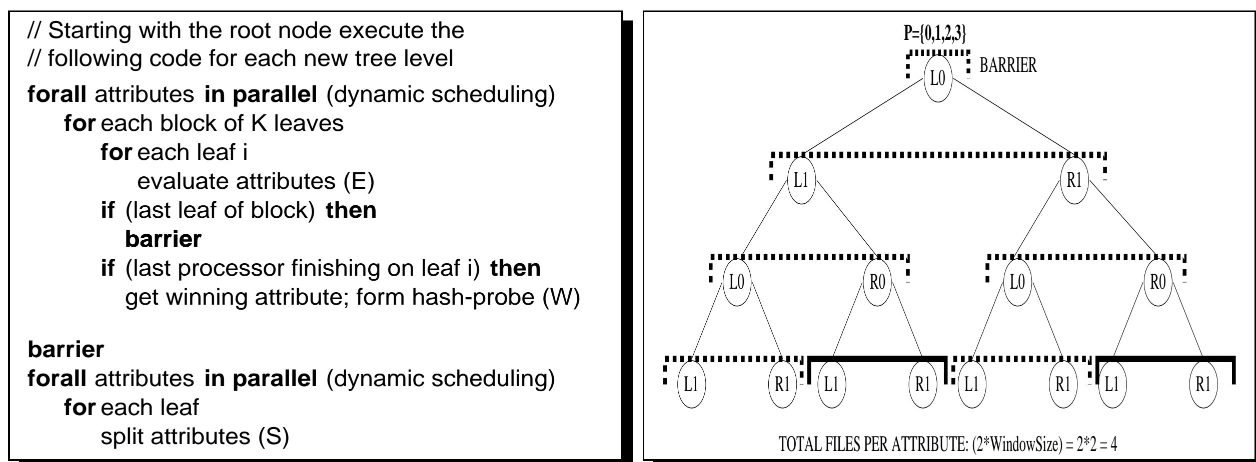


Figure 5: The FWK algorithm.

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 5 addresses this problem. The basic idea is to overlap the $\mathcal{W}$-phase with the $\mathcal{E}$-phase of the next leaf at the current level, thus realizing *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 were a set of 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

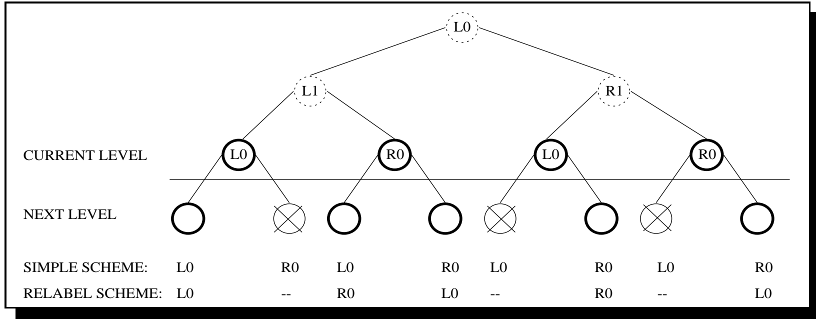


Figure 6: Scheduling attribute files.

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 6). However, if we knew which children will be pure in 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 6. 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 implies more temporary files, which incurs greater file creation overhead and tends to have less locality. The ideal window size is a trade-off between the above conflicting goals.

3.2.3 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 7 shows the pseudo-code for MWK.

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

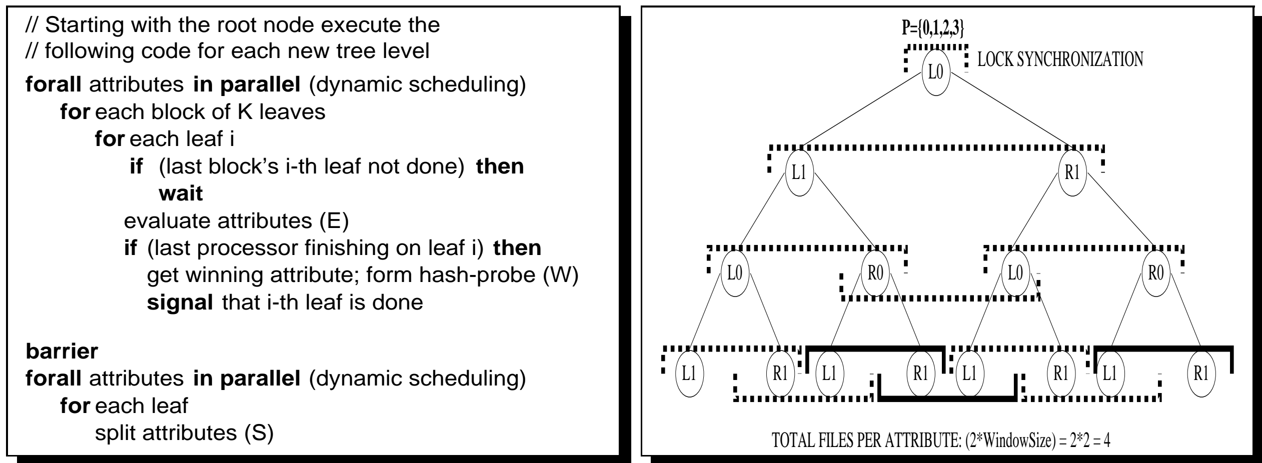


Figure 7: The MWK algorithm.

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.

3.3 Subtree 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 8. 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 group master (we chose the processor with the smallest identifier as the master). The master is responsible for partitioning the processor set.

At any given point in the algorithm, there may be multiple processor groups working on distinct

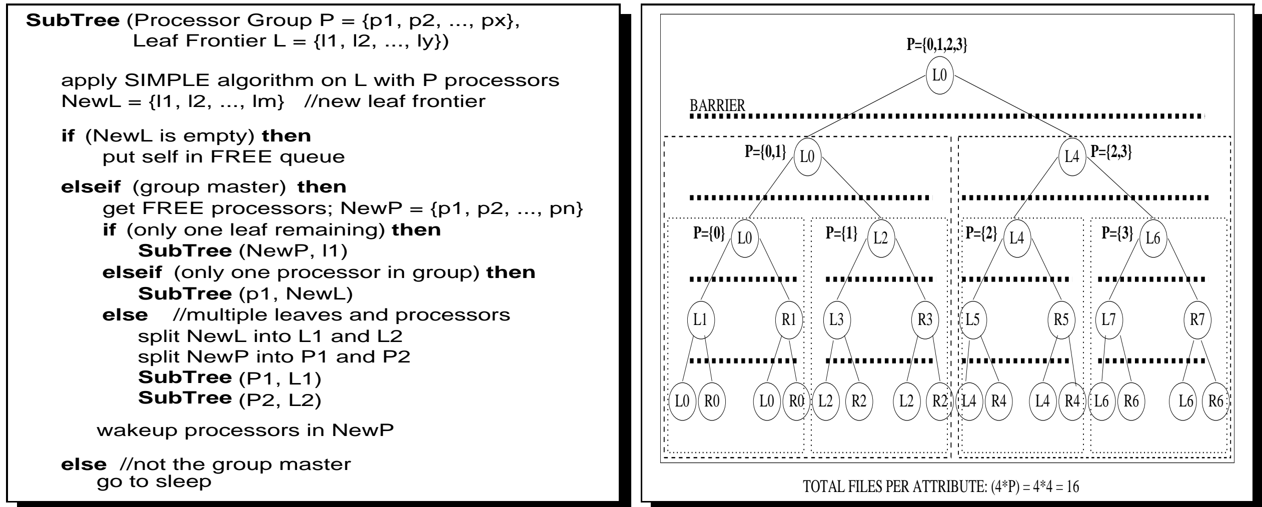


Figure 8: The SUBTREE algorithm.

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 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 into two parts, and also splits 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.

3.4 Discussion

We now qualitatively discuss the relative merits of each of the proposed algorithms. An experimental comparison will be presented in the next section.

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 this approach 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.

The reader may have observed that, as described above, our SMP algorithms 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 $4dP$. The essential idea is to associate physical files for writing attribute lists with processor (rather than with 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.

4 Performance Evaluation

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/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 [4] and C4 (a predecessor of C4.5 [17]). This performance evaluation, available in [14], 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.

4.1 Experimental Setup

Machine Configuration Experiments were performed on two SMP machines with different configurations shown in Table 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: Machine configurations.

Datasets An often used classification benchmark is STATLOG[15]. 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 [1] 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 2). Both these functions divide the database into two classes: Group A and Group B. 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 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.

Dataset					Corresponding Tree	
Dataset Notation	Function	Number of Attributes	Number of Tuples	Database Size	Number of Levels	Max No. of Leaves/Level
F2-A8-D1000K	F2	8	1000K	61 MB	4	2
F2-A32-D250K	F2	32	250K	57.3 MB	4	2
F2-A64-D125K	F2	64	125K	56.6 MB	4	2
F7-A8-D1000K	F7	8	1000K	61 MB	60	4662
F7-A32-D250K	F7	32	250K	57.3 MB	59	802
F7-A64-D125K	F7	64	125K	56.6 MB	55	384

Table 2: Dataset characteristics.

Algorithms Our initial experiments (not reported here for lack of space) 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 $W * d/p \geq 8$.

4.2 Initial Setup and Sort Time

Table 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 parallelizing sort including on SMP machines [3]. 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%
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%

Table 3: Sequential setup and sorting times.

4.3 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.

Figure 9 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 10 and 11 show similar results for datasets *A32-D250K* and *A64-D125K*, respectively. The speedup chart in the rightmost part of each figure shows the speedup of the total time (including the setup time and sort time), while the speedup chart to the left of it and the two leftmost bar charts show only the build time (excluding the setup time 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. We see two trends from these figures.

First, the overall 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 figures also show that on F2, increasing the number of attributes worsens the performance of SUBTREE. This is because a free processor can join a new group only at the end of a level. As each processor or group becomes free it waits in the FREE queue to rejoin the computation. However, it will not be assimilated into the new group until one of the existing group finishes working on all 64 attributes. Clearly, the larger the number of attributes the larger the wait, and this adversely impacts the performance of SUBTREE. On the other hand, MWK does not suffer from this phenomenon. It has the opposite trend, more attributes lead to a better attribute scheduling, which tends to minimize imbalance.

Another 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

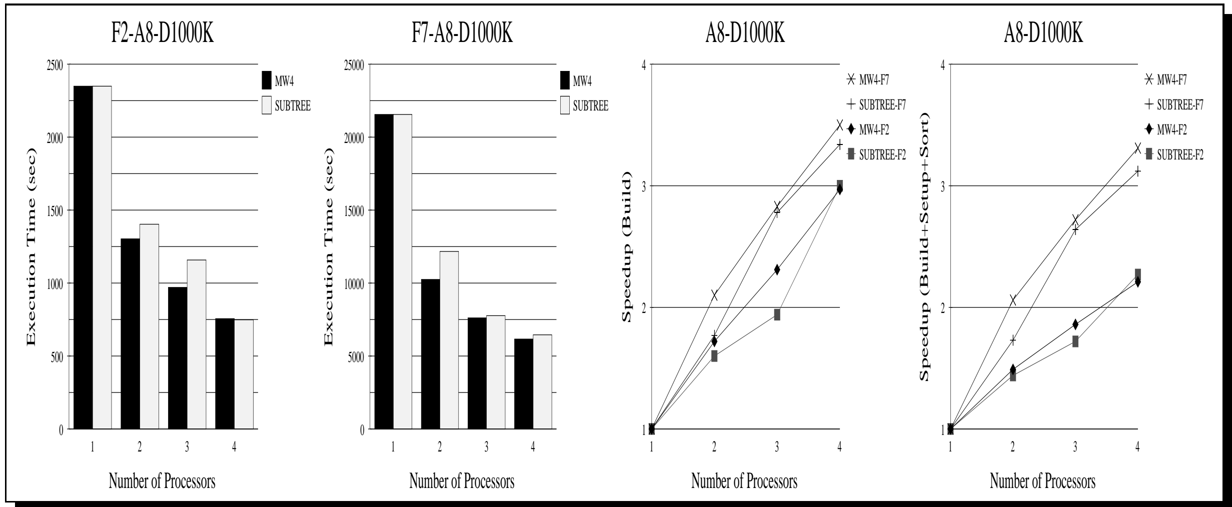


Figure 9: Local disk access: functions 2 and 7; 8 attributes; 1000K records.

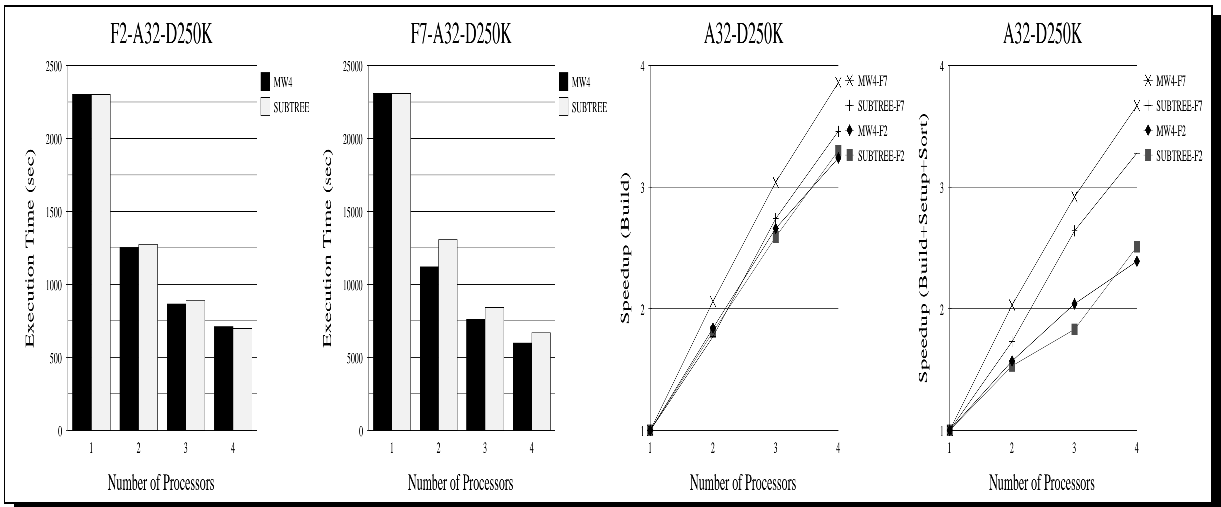


Figure 10: Local disk access: functions 2 and 7; 32 attributes; 250K records.

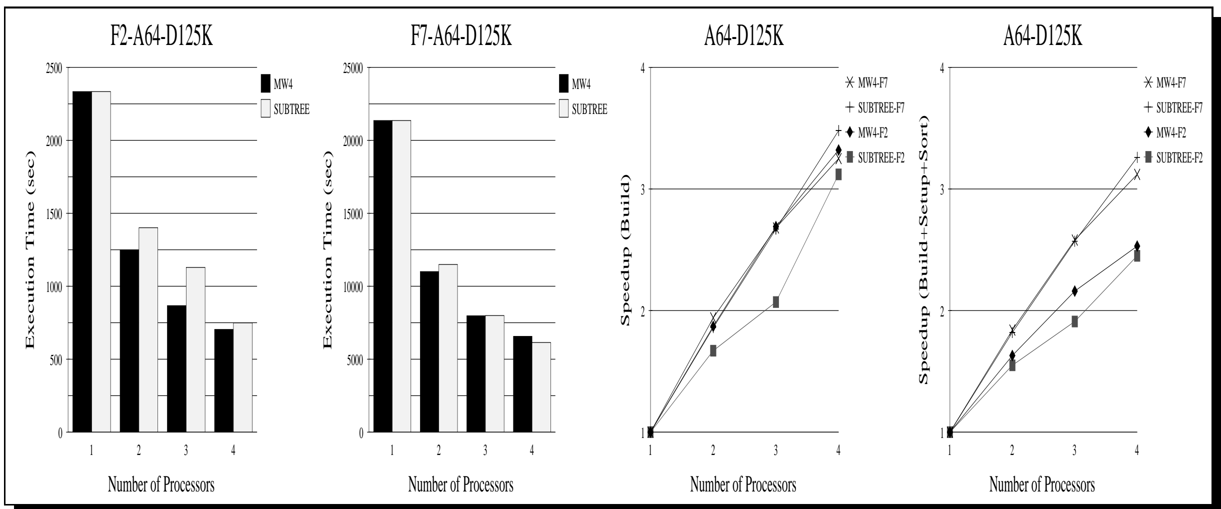


Figure 11: Local disk access: functions 2 and 7; 64 attributes; 125K records.

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.

4.4 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 12 to 14 show three sets of timing and speedup charts, corresponding to Figures 9 to 11, respectively. Note the numbers of processors on the X-axis are 1, 2, 4 and 8.

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 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 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. The combination of factors 2 and 3 can be seen from the 8-processor timings for the two *A8-D1000K* datasets.

5 Conclusion

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 phase within a tree node, thus avoiding potential sequential bottleneck for the hash-probe construction for the split phase. The MWK algorithm employs conditional variable, not barrier, among leaf nodes to avoid unnecessary processor blocking time at a barrier. 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 group" 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 overall has a slight edge. These experiments demonstrate that the important data mining task of classification can be effectively parallelized on SMP machines.

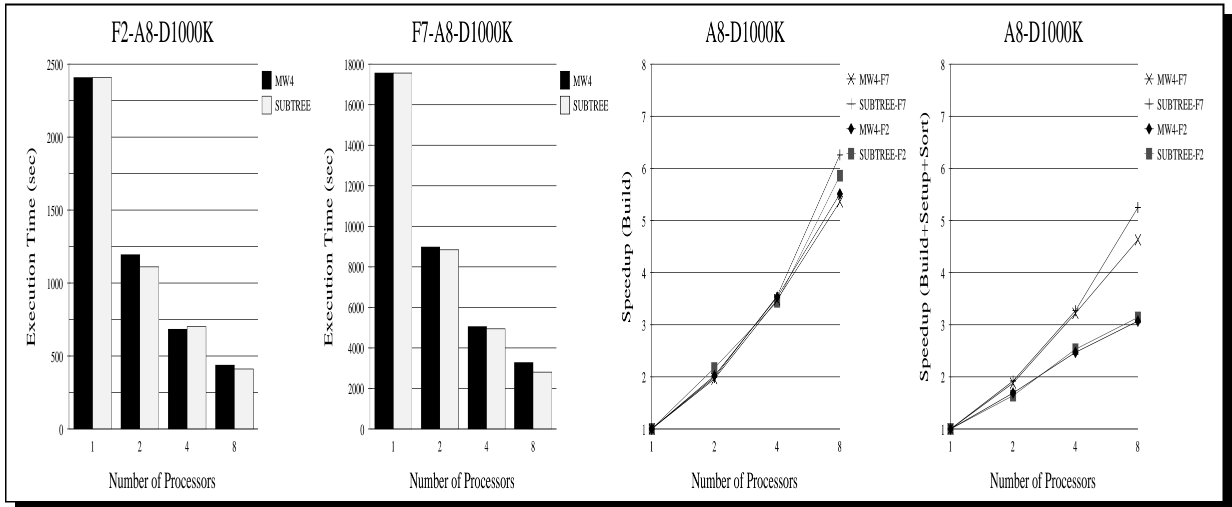


Figure 12: Main-memory access: functions 2 and 7; 8 attributes; 1000K records.

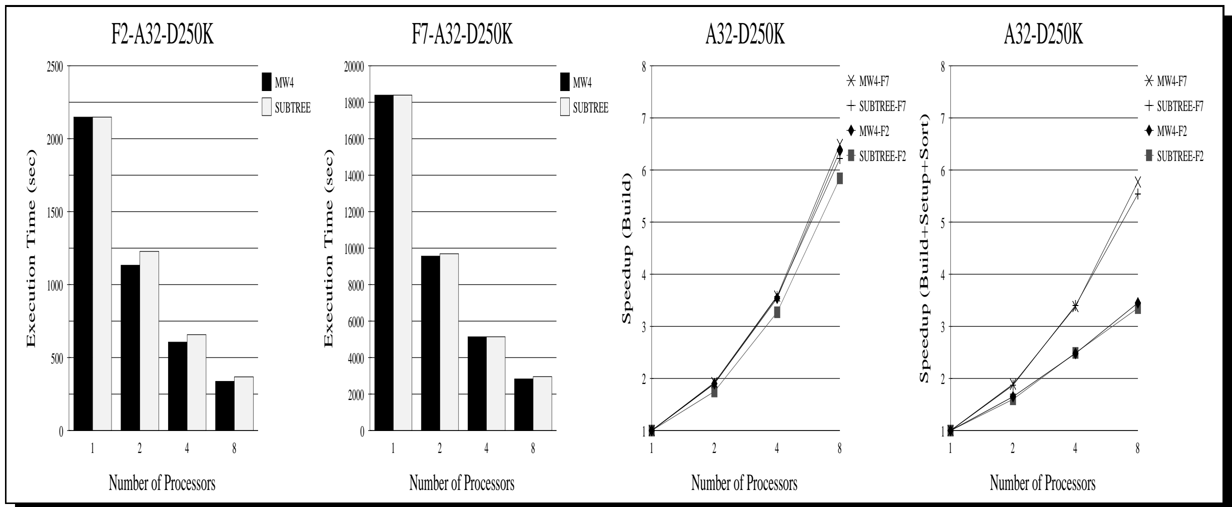


Figure 13: Main-memory access: functions 2 and 7; 32 attributes; 250K records.

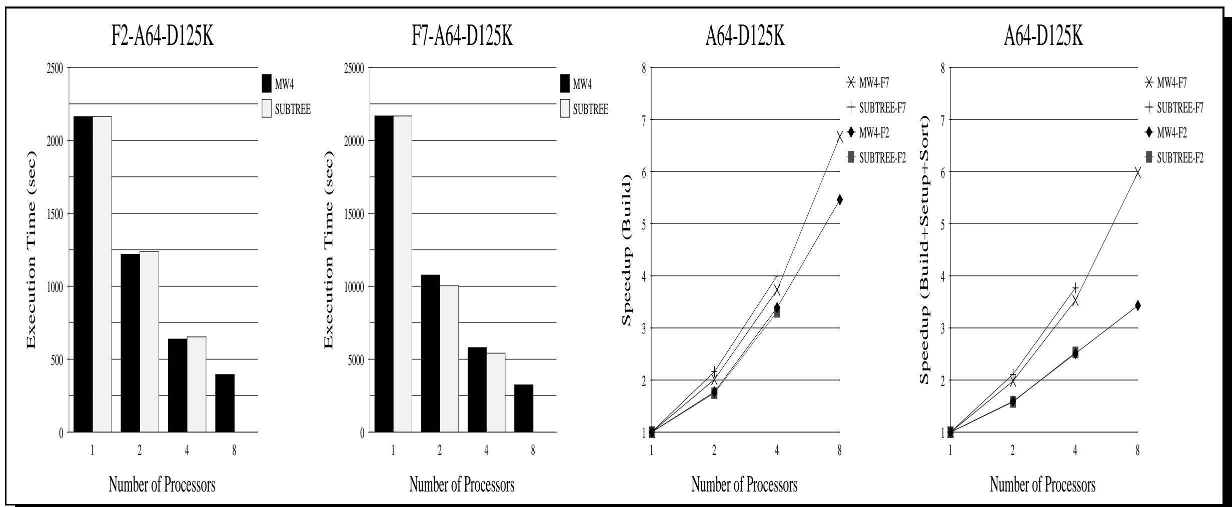


Figure 14: Main-memory access: functions 2 and 7; 64 attributes; 125K records.

Acknowledgment We would like to thank John Shafer for insightful discussions during all phases of this work.

References

[1] Rakesh Agrawal, Sakti Ghosh, Tomasz Imielinski, Bala Iyer, and Arun Swami. An interval classifier for database mining applications. In *Proc. of the VLDB Conference*, pages 560–573, Vancouver, British Columbia, Canada, August 1992.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[3] Dina Bitton, David DeWitt, David K. Hsiao, and Jaishankar Menon. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16(3):287–318, September 1984.

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

[5] Jason Catlett. *Megainduction: Machine Learning on Very Large Databases*. PhD thesis, University of Sydney, 1991.

[6] Philip K. Chan and Salvatore J. Stolfo. Experiments on multistrategy learning by meta-learning. In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, pages 314–323, 1993.

[7] Philip K. Chan and Salvatore J. Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. Second Intl. Workshop on Multistrategy Learning*, pages 150–165, 1993.

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. of the 1st Int'l Conf. on Parallel and Distributed Information Systems*, pages 280–291, December 1991.

[9] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

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

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

[12] Bil Lewis and Daniel J. Berg. *Threads Primer*. Prentice Hall, New Jersey, 1996.

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

[14] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

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

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

[17] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[18] John Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, September 1996.

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

[20] Sholom M. Weiss and Casimir A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufman, 1991.