Parallel Classification for Data Mining on Shared-Memory Multiprocessors

Mohammed J. Zaki
Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180
zaki@cs.rpi.edu

Ching-Tien Ho and Rakesh Agrawal
IBM Almaden Research Center
San Jose, CA 95120
{ho,ragrawal}@almaden.ibm.com

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. Our performance evaluation shows that the construction of a decision-tree classifier can be effectively parallelized on an SMP machine with good speedup.

1 Introduction

Classification is one of the primary data mining task [2]. The input to a classification system consists of example tuples, called a *training set*, with each tuple having several *attributes*. Attributes can be *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. A special *class* attribute indicates the label or category to which an 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. Classification has applications in diverse fields such as retail target marketing, fraud detection, and medical diagnosis [10]. Amongst many classification methods proposed over the years [13, 10], decision trees are particularly suited for data mining, since they can be built relatively fast compared to other methods and they are easy to interpret [11]. Trees can also be converted into SQL statements that can be used to access databases efficiently [1]. Finally, decision-tree classifiers obtain similar, and often better, accuracy compared to other methods [10].

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 data mining. Developing classification models using larger training sets can enable the development of higher accuracy models. Various studies have confirmed this [5, 6]. Recent classifiers that can handle disk-resident data include SLIQ [9], SPRINT [12], and CLOUDS [3].

As data continue to grow in size and complexity, 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) machines. Examples include parallel ID3 [7], which assumed that the entire dataset could fit in memory; Darwin toolkit with parallel CART [4] from Thinking Machine, whose details are not available in published literature; parallel SPRINT on IBM SP2 [12]; and ScalParC [8] on a Cray T3D.

While distributed-memory machines provide massive parallelism, shared-memory machines (also called shared-everything systems), are also capable of delivering high performance for low to medium degree of parallelism at an economically attractive price. Increasingly SMP machines are being networked together via high-speed links to form hierarchical clusters. Examples include the SGI Origin 2000 and IBM SP2 system which can have a 8-way SMP as one *high* node. A shared-memory system offers a single memory address space that all processors can access. Processors communicate through shared variables in memory. Synchronization is used to co-ordinate processes. Any processor can also access any disk attached to the system. The SMP architecture offers new challenges and trade-offs that are worth investigating in their own right.

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.

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. 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. A more detailed version of this paper appears in [14].

2 Serial Classification

Each node in a decision tree classifier 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 shows a decision-tree developed from the training set on its left.

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

Decision Tree

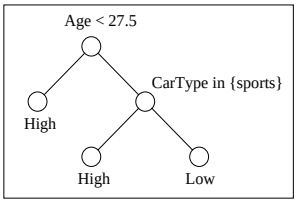

Figure 1. Car insurance example.

A decision-tree classifier is usually built in two phases [4, 11]: a *growth* phase and a *prune* phase. The tree is grown using a divide-and-conquer approach. If all examples in the training set 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, then S is partitioned into two subsets which serve as input for the recursive step.

The tree thus built can "overfit" the data. The prune phase generalizes the tree, and increases the classification accuracy on new examples, by removing statistical noise or variations. This phase requires access only to the fully grown tree, while the tree growth phase usually requires multiple passes over the training data. 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 [9]. We therefore concentrate on the tree building phase, which 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 [12]. It builds the tree in a breadth-first order and uses a one-time pre-sorting technique to reduce the cost of finding split point for a continuous attribute.

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, and the partitioned lists for its two children.

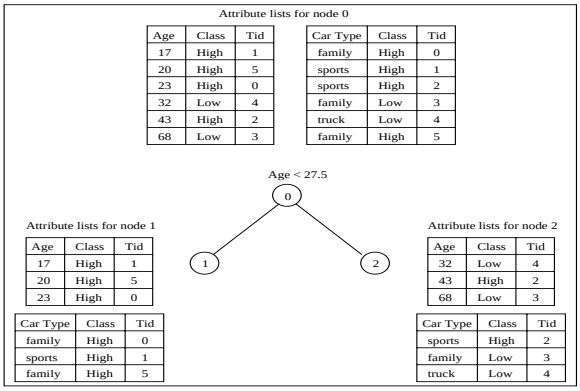

Figure 2. Splitting a node's attribute lists.

SPRINT also uses histograms tabulating the class distributions of the records in an attribute list to evaluate split points for the list. Since the attribute lists are processed one after the other, only one set of histograms are kept in memory at any given time.

2.2 Finding Good Split Points

The form of the split test used to partition the data depends on the type of the attribute. 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 per attribute that serve as alternates. 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.

3 Parallel Classification on SMP Systems

We now look at 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 [9], 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 $\mathcal{S}$). The parallel schemes will be described in terms of these steps.

Our prototype implementation of these schemes uses the POSIX threads (pthread), which are light weight processes. 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 parallelism: 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*, where the attributes are divided equally among the different processors so that each processor is responsible for $1/P$ attributes. The parallel implementation of SPRINT on an IBM SP2 [12] is based on *record data parallelism*, where each processor is responsible for processing roughly $1/P$ fraction of each attribute list. Record parallelism is not well suited to SMP systems since it is likely to cause excessive synchronization, and replication of data structures.

Task parallelism: 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 will describe the Moving-Window-K algorithm (MWK) based on attribute data parallelism. However, 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 3 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.

```
// 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 3. The BASIC algorithm.

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.

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. 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.

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 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)

```
// 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 4. 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 4 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 $0 \leq i \leq K-1$.

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.

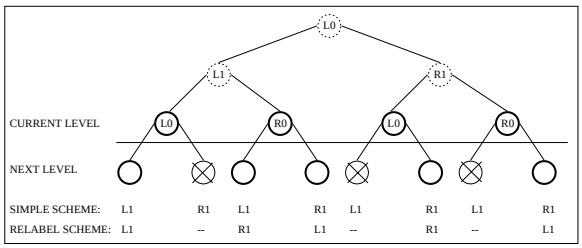

Figure 5. Scheduling attribute files.

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 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. 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 5. 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 $L1, L1, R1, R1, R1$. With a window of size $K = 2$, there is only one instance where work can overlap, i.e., when going from $L1$ to $R1$. However, if we relabel the valid children's files then we obtain the perfectly schedulable sequence $L1, R1, L1, R1, L1$.

The work overlap is achieved at the cost of increased barrier synchronization; one for each K-block. A large window size not only increases the overlap but also minimizes the number of synchronizations, but a larger window implies more temporary files, which incur greater file creation overhead and tend 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, shown in Figure 6. It eliminates the serial bottleneck of BASIC and exploits greater parallelism than FWK. Consider a 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 uses this extra parallelism.

```
// 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 6. The MWK algorithm.

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.

3.3 Task Parallel 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 subtrees. 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 for the dynamic SUBTREE algorithm is shown in Figure 7. 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.

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

    apply SIMPLE 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 7. The SUBTREE algorithm.

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 possibilities 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 relevant processors – from the original group and those acquired from the *FREE* queue. For P processors, there are at most P groups. Since the attribute files for all groups must be distinct, SUBTREE requires up to $4P$ files per attribute.

3.4 Qualitative Algorithm Comparison

The MWK scheme eliminates the hash-probe construction bottleneck of BASIC via task pipelining. 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.

4 Performance Evaluation

We use the execution time as the main metric of classifier performance, since [9] has shown that SLIQ/SPRINT achieve similar or better classification accuracy and produce smaller trees when compared to other classifiers like CART [4] and C4 (a predecessor of C4.5 [11]).

4.1 Experimental Setup

Machine Configuration: Experiments were performed on two SMP machines (with different configurations) with a 112 MHz PowerPC-604 processor, and a 1 MB L2-Cache. Machine A has 4 processors, 128 MB memory and 300 MB disk. The amount of memory is insufficient for training data, temporary files, and data structures altogether to fit in memory. Thus data reads/writes will go to disk each time. Machine B has 8 processors, 1 GB memory and 2 GB disk. All the temporary files created during the run will be cached in memory. Machine A is of greater interest to the database community and we present a detailed set of experiments for it. 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.

Datasets: We use synthetic datasets proposed in [1], using two classification functions of different complexity. These functions divide the database into two classes. 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). The notation *Fx-Ay-DzK* is used to denote the dataset with function x, y attributes and $z \cdot 1000$ example records. The database size shown in Table 1 is only the initial size. After SPRINT allocates temporary attribute files the final size is at least twice the initial size. Thus the datasets would require more than 192 MB disk space, and would be out-of-core on Machine A, which has only 128 MB memory.

Setup and Sort Time: Table 1 shows the time for creating the attribute lists and for sorting the continuous attributes. 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. Consequently for simple datasets such as F2 the setup and sort time can be significant. However, for the complex datasets such as F7 this time is small.

4.2 Parallel Performance: Local Disk Access

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 also found that a window size of 4 works well in practice. 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).

Figures 8 and 9 show the parallel performance and speedup of the two algorithms as we vary the number of processors on Machine A, for the two classification functions F2 and F7, on the datasets *A32-D250K* and *A64-D125K*. The rightmost charts on each row show the speedup with respect to total time (including setup and sort time), while the other charts show only the build 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 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 to or better than SUBTREE. The difference ranges from 8% worse to 22% better than SUBTREE. Overall, MWK is usually 10% better than SUBTREE.

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 at any level. 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 the attributes. The larger the number of attributes the larger the wait, adversely impacting the performance of SUBTREE. On the other hand, MWK has the opposite trend; more attributes lead to a better attribute scheduling, which tends to minimize imbalance.

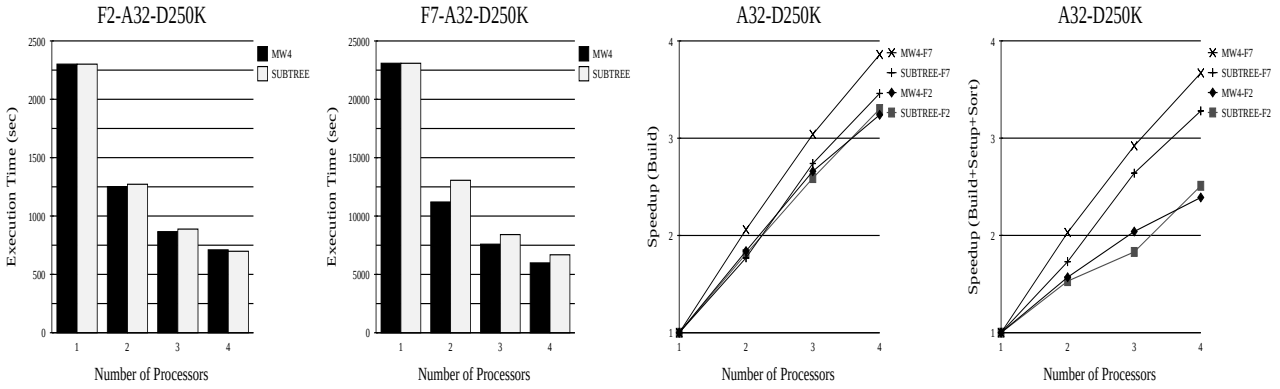

Figure 8. Local disk access: functions 2 and 7; 32 attributes; 250K records.

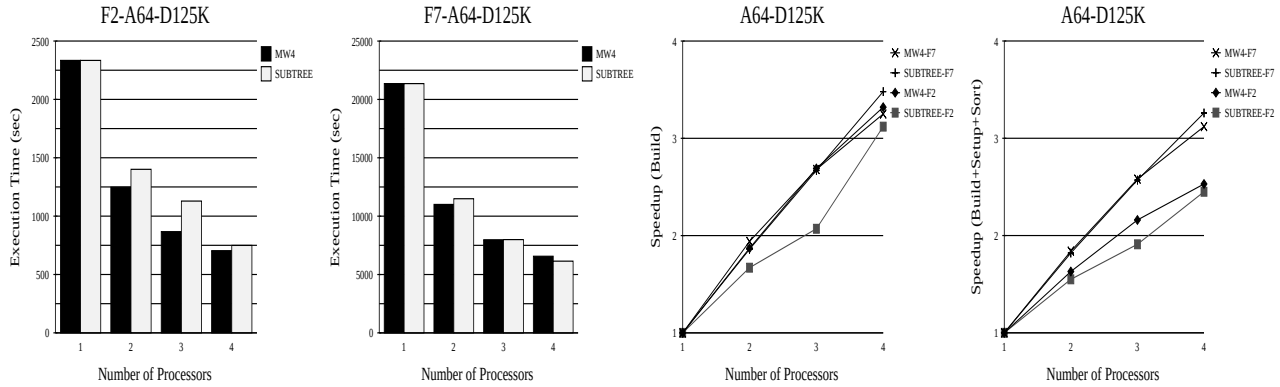

Figure 9. Local disk access: functions 2 and 7; 64 attributes; 125K records.

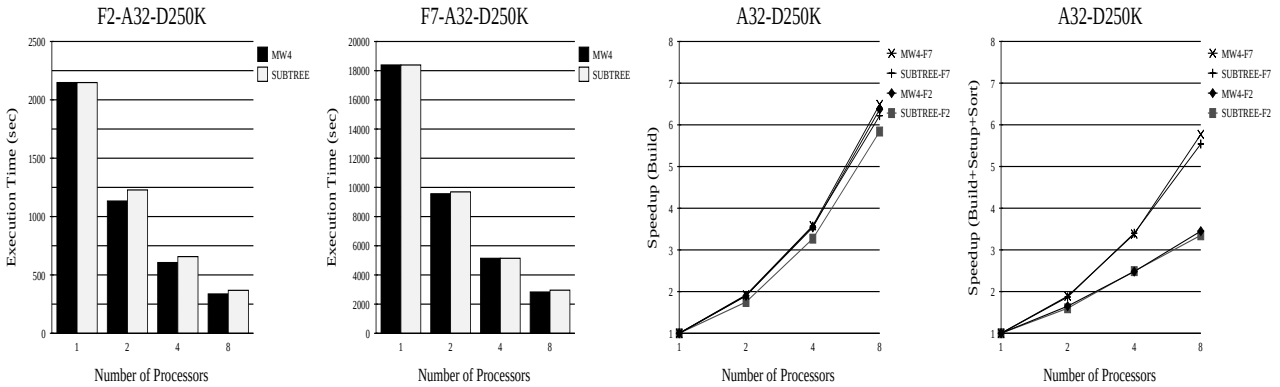

Figure 10. Main-memory access: functions 2 and 7; 32 attributes; 250K records.

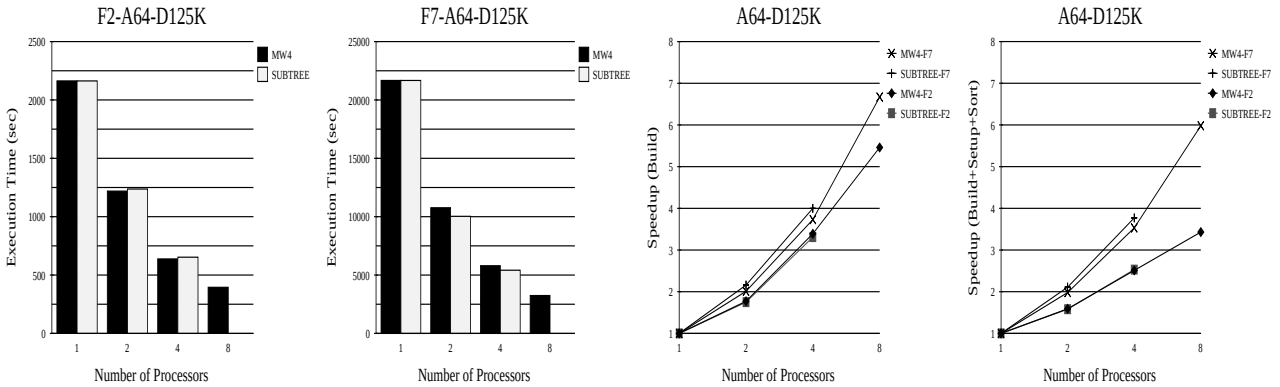

Figure 11. Main-memory access: functions 2 and 7; 64 attributes; 125K records.

Dataset	DB Size (MB)	Tree Size		Setup Time (seconds)	Sort Time (seconds)	Total Time (seconds)	Setup %	Sort %
		No. Levels	Max Leaves/Level					
F2-A32-D250K	96	4	2	685	598	3584	19.1%	16.6%
F2-A64-D125K	96	4	2	705	626	3665	19.2%	17.1%
F7-A32-D250K	96	59	802	838	780	24706	3.4%	3.2%
F7-A64-D125K	96	55	384	672	636	22664	3.0%	2.8%

Table 1. Dataset characteristics, and sequential setup and sorting times.

Another trend is that a large number of processors tends to favor SUBTREE. This 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 (P being the number of processors), 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.

4.3 Parallel Performance: Main-Memory Access

Machine B 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. Figures 10 and 11 show sets of timing and speedup charts. For 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 for function F7. The speedups of total time on F7 for both algorithms on 8 processors range from 4.63 to 5.77 Again, the important observation is that both algorithms perform very well for various datasets even up to 8 processors.

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 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 overall has a slight edge. These experiments demonstrate that the important data mining task of classification can be effectively parallelized on SMP machines.

Acknowledgment

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

References

[1] R. Agrawal, et al. An interval classifier for database mining applications. In *VLDB Conference*, Aug 1992.

[2] R. Agrawal, T. Imielinski, A. Swami. Database mining: A performance perspective. *IEEE Trans on Knowledge and Data Engg.* , 5(6):914–925, Dec 1993.

[3] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: A decision tree classifier for large datasets. In *4th Intl. Conf. on Knowledge Discovery and Data Mining*, Aug 1998.

[4] L. Breiman, et al. *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

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

[6] P. Chan, S. Stolfo. Experiments on multistrategy learning by meta-learning. In *2nd Intl. Conf. on Info. and Knowledge Mgmt.*, Nov 1993.

[7] D. Fifield. Distributed tree construction from large data-sets. Bachelor Thesis, Australian Natl. U., 1992.

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

[9] M. Mehta, R. Agrawal, J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *5th Intl. Conf. on Extending Database Technology*, March 1996.

[10] D. Michie, et al. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, 1994.

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

[12] J. Shafer, R. Agrawal, M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *22nd VLDB Conference*, Sept 1996.

[13] S. Weiss, C. Kulikowski. *Computer Systems that Learn*. Morgan Kaufman, 1991.

[14] M. Zaki, C-T. Ho, R. Agrawal. Parallel Classification for Data Mining on Shared-Memory Multiprocessors. IBM Technical Report, 1998. Available from www.almaden.ibm.com/cs/quest/publications.html.