Parallel Classification on SMP Systems

Mohammed J. Zaki [*], Ching-Tien Ho, and Rakesh Agrawal
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

Abstract

This paper presents fast scalable decision-tree-based classification algorithms targeting shared-memory systems. The algorithms are based on the sequential SPRINT classifier and span the gamut of data and task parallelism. The data parallelism is based on attribute scheduling among processors. This is extended with task pipelining and dynamic load balancing to yield more efficient schemes. The task parallel approach uses dynamic subtree partitioning among processors. These schemes are disk based and achieve excellent speedup, making them ideally suited for data mining in very large databases.

1 Introduction

An important task of data mining is to assign objects to predefined categories or classes – a process called *Classification*. The input to the classification system consists of a set of example records, called a *training set*, over several fields or *attributes*. Attributes are either *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. One of the attributes, called the *classifying* attribute, indicates the *class* or label to which each example belongs. The goal of classification is to induce a model from the training set, that can be used to predict the class of a new record. Classification has applications in diverse fields such as retail target marketing, customer retention, fraud detection and medical diagnosis [9].

While there has been a lot of research in classification in the past, the focus had been on developing classification models using training sets that 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 [4] [5] [6]. Examples of fast scalable classification systems include SLIQ [8], and SPRINT [11], which was disk-based. SPRINT was also parallelized on the IBM SP2 [7] parallel distributed-memory machine.

[*] Current affiliation: Computer Science Department, University of Rochester, Rochester, NY 14627

This paper presents fast scalable decision-tree-based classification algorithms targeting shared-memory systems, the first such study. The algorithms are based on the sequential SPRINT classifier, and span the gamut of data and task parallelism. The data parallelism is based on attribute scheduling among processors. This is extended with task pipelining and dynamic load balancing to yield more complex schemes. The task parallel approach uses dynamic subtree partitioning among processors. Our experiments show that we obtain a speedup from 3.0 to 3.9 for the tree growth phase, and from 2.2 to 3.7 for the total time, on a 4-processor SMP machine.

The rest of the paper is organized as follows. In Section 2 we review the serial SPRINT algorithm, which forms the backbone of the new SMP parallel algorithms. Section 3 describes our new SMP algorithms based on various data and task parallelization schemes. In Section 4, we give experimental results, and we conclude in Section 5.

2 Serial SPRINT Algorithm

A decision tree is a structure that is either a *leaf*, indicating a class, or a *decision node*, specifying some test on one or more attributes, with one branch for each outcome of the split test. A decision tree classifier is usually built in two phases [3] [10]: a growth phase and a prune phase. The growth phase takes the input training set, and recursively partitions it so that each leaf is composed entirely or predominantly of the same class. The tree pruning phase generalizes the tree by removing statistical noise or variation to avoid over-fitting the training set. Usually less than 1% of the total time needed to build a classifier is spent in the pruning phase [8]. We will therefore concentrate on the computation and I/O intensive tree growth phase.

Attribute lists SPRINT builds the tree breadth-first and uses a one-time pre-sorting technique to reduce the cost of continuous attribute evaluation. For each attribute, it initially creates a disk-based *attribute list* consisting of an attribute value, a class label, and a tuple identifier or *tid*. Initial lists for continuous attributes are sorted by attribute value when first created. The lists for categorical attributes remain in unsorted order. As the tree is split into subtrees, the attribute lists are also split. By preserving the order of records in the partitioned lists, they do not require resorting. Figure 1 shows an example of the initial sorted attribute lists associated with the root of the tree and also the resulting partitioned lists for the two children.

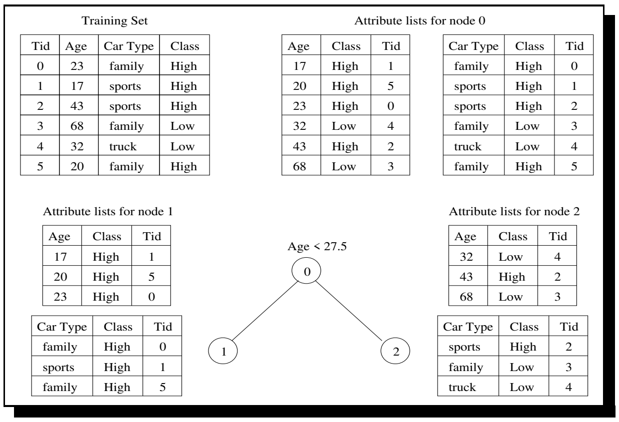

Figure 1. Splitting a node's attribute lists.

2.1 Finding Split Points and Splitting the Data

There are two major issues that have critical performance implications in the tree-growth phase: 1) How to find split points that define node tests, and 2) Having chosen a split point, how to partition the data. SPRINT uses the *gini* index [3] as a measure of split quality. The gini index is computed by scanning a node's attribute lists, and computing the class distributions on both sides of the split point. The attribute with the minimum gini value and the associated *winning* split point, is used to partition the data.

Figure 1 shows how the data is split once the winning attribute (Age) is found. To split the attribute list for the winning attribute (Age), we simply scan the list and apply the split test ($Age < 27.5$). For splitting a "losing" attribute ($CarType$) we constructs a bit or hash probe on the *tids*, noting the child where a particular record belongs. A simple scan and application of split test on the probe is used to split the other attributes.

Avoiding multiple attribute lists The attribute lists of each attribute are stored in disk files. Since file creation and deletion for each tree level is an expensive operation, SPRINT actually uses only four physical files per attribute. It has one attribute file for all left children, one file for all the right children, and separate files for the current and next level.

To split a node we read the files from the current level, and write to the left and right files for the next level. By processing tree nodes in the order they appear in the attribute lists, this approach also avoids any random seeks within a file to find a node's records — reading and writing remain sequential operations. This optimization is illustrated

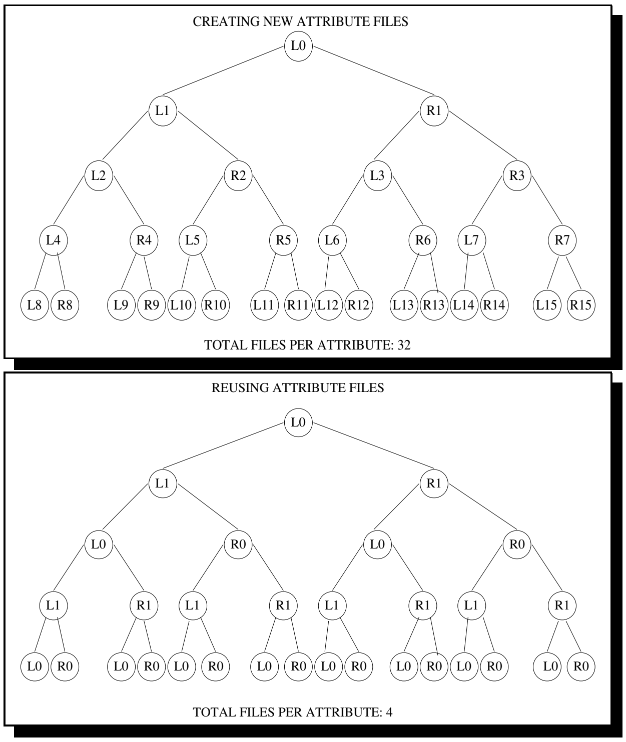

Figure 2. Avoiding multiple attribute files.

in Figure 2, and has important implications for the parallelization strategies presented below.

3 Parallel Classification on SMP Systems

We now describe different algorithms for parallelizing the compute- and data-intensive tree growth phase, on shared-memory systems. Recall that SPRINT has three main steps that are performed for each node at each level of the tree: 1) Evaluate split points for each attribute (denoted as step $\mathcal{E}$); 2) Find the winning split-point, i.e., the best among all the attributes, and then construct a bit probe from the winning attribute (denoted as step $\mathcal{W}$); and 3) Split all the attributes into two parts, one for each child, using the bit probe (denoted as step $\mathcal{S}$). Our parallel schemes will be described in terms of these steps. Our prototype implementation of these schemes uses POSIX standard pthreads.

There are two major ways of parallelizing classification – the *data parallel* approach and the *task parallel* approach. In data parallelism the P processors work on distinct attributes, and synchronously construct the global decision tree. It essentially exploits the intra-node parallelism, i.e. that available within a decision tree node. The task parallel approach exploits the inter-node parallelism, i.e. different subtrees can be grown in parallel among the processors.

3.1 Attribute Data Parallelism

We now 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. All schemes utilize dynamic attribute scheduling, since a static scheduling scheme is not particularly suited for classification due to non-uniform cost per attribute.

3.1.1 The BASIC Algorithm (BASIC)

In the BASIC approach, attributes are scheduled dynamically by using an attribute counter and simple locking. A processor acquires the lock, grabs an attribute, increments the counter, and releases the lock. The tree is built in a breadth-first manner so 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 also ensures that there is no attribute file sharing. A single barrier marks the end of the evaluation phase. Figure 3 illustrates the evaluation phase of BASIC.

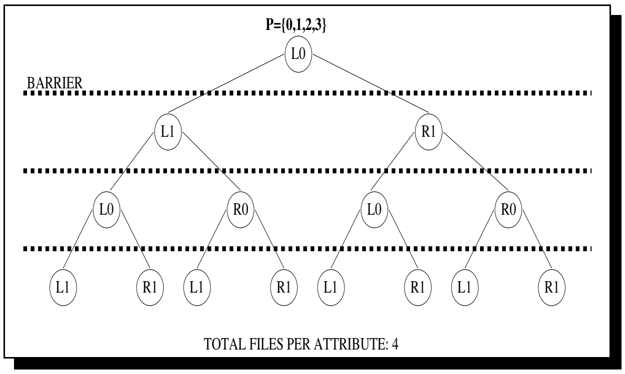

Figure 3. The BASIC Algorithm

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 minimum split value from among each processor's locally best split. We can then proceed to scan the winning attribute's records and form the hash probe. Both these tasks are performed serially by a pre-designated master processor. This step thus represents a potential bottle-neck in this algorithm.

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.

3.1.2 The Fixed-Window-K Algorithm (FWK)

The basic idea of the Fixed-Window-K (FWK) approach is to overlap the $\mathcal{W}$-phase with the $\mathcal{E}$-phase of the next leaf at the current level, thus realizing *pipelining*, and overcoming the bottleneck of BASIC. 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, however they are performed on a group of K leaves at a time. Another difference is that FWK requires $2K$ files per attribute instead of four so that the different leaves in a group can be processed independently. Figure 4 illustrates this scheme for $K = 2$.

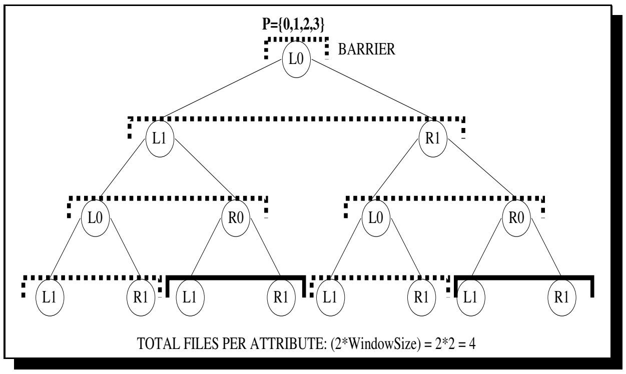

Figure 4. The FWK Algorithm

It should be noted that the overlapping of work is achieved at the cost of increased barrier synchronization, one per each K-block. We must therefore choose a sufficiently large window size, so that it not only increases the overlap, but also minimizes the number of barrier synchronizations. Note, however, that larger window size implies more temporary files, which incurs more file creation overhead and tends to have less locality.

3.1.3 The Moving-Window-K Algorithm (MWK)

Figure 5 shows the MWK algorithm, which extends the FWK algorithm by incorporating moving windows, and thus reduces processor idle time due to barriers. 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 $\{L0_2, R0_2\}$, but also between the leaves of these two blocks, $\{R0_1, L0_2\}$. The MWK algorithm makes use of this additional parallelism. The scheme can be 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 bit probe, and then wakes all processors sleeping on the conditional variable.

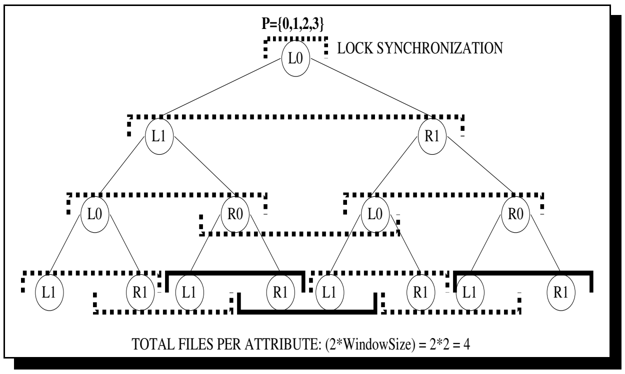

Figure 5. The MWK Algorithm

It should be observed that the gain in available parallelism is at the cost of increased lock synchronization per leaf (however, there is no barrier anymore). 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.

3.2 Subtree Task Parallel Algorithm (SUBTREE)

An illustration of the SUBTREE algorithm is provided in Figure 6. 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. The master is responsible for partitioning the processor set.

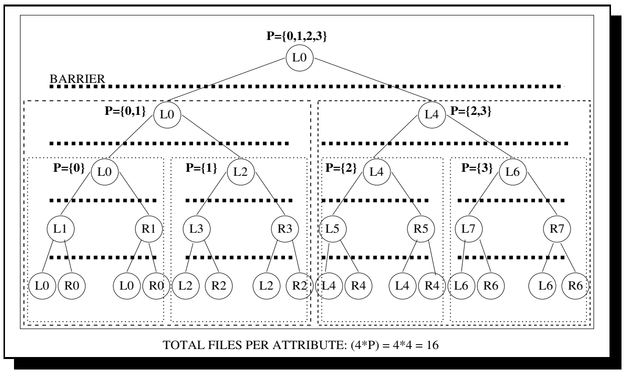

Figure 6. The SUBTREE Algorithm

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 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 remaining, then it forms a group on its own and works on the current leaf frontier. Lastly, if there are multiple leaves and 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 their attribute files must be distinct, this scheme requires up to $4P$ files per attribute.

4 Performance Evaluation

We use the tree build time as our performance metric for evaluating the proposed algorithms, since it has been shown that SLIQ/SPRINT achieves comparable or better classification accuracy than other classifiers [8].

4.1 Experimental Setup

Experiments were performed on a 4 processor SMP machine, with 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. We used different synthetic benchmark datasets (proposed in [1]) shown in Table 1. Two classification functions are used. Function F2 produces small trees, while function F7 produces very large trees.

Dataset					Corresponding Tree	
Dataset Notation	Function	No. Attributes	No. Tuples	Total Size	No. Levels	Max 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 1. Dataset characteristics.

Our initial experiments confirmed that MWK was indeed better than BASIC as expected, and that it performs as well or better than FWK. Thus, we will only present the performance of MWK and SUBTREE. We also found that the time spent in the initial setup and sort phase can be significant (upto 38%) for the datasets with small trees (function F2), whereas it is small (upto 8%) for complex datasets with large trees (function F7).

4.2 Parallel Performance

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).

Figure 7 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 8 and 9 show similar results for datasets *A32-D250K* and *A64-D125K*, respectively. Only the rightmost graphs uses the total time (including setup time and sort time), while the other graphs are based only on build time (excluding setup and sort phase).

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 internode 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 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 (where P is 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.

5 Conclusion

In this paper we presented fast scalable decision-tree-based classification algorithms targeting shared-memory systems. The algorithms span the spectrum of data and task parallelism. Our experiments confirmed that these schemes achieve good speedup, making them ideally suited for data mining in very large databases.

References

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

[2] D. Bitton et al. A taxonomy of parallel sorting. *ACM Computing Surveys*, 16(3):287–318, Sept 1984.

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

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

[5] P. K. Chan and S. J. Stolfo. Experiments on multistrategy learning by meta-learning. In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, 1993.

[6] P. K. Chan and S. J. Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. Second Intl. Workshop on Multistrategy Learning*, 1993.

[7] International Business Machines. *Scalable POWERparallel Systems*, GA23-2475-02 edition, Feb 1995.

[8] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Int'l Conference on Extending Database Technology (EDBT)*, Mar 1996.

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

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

[11] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *22nd Int'l Conference on Very Large Databases*, Sept 1996.

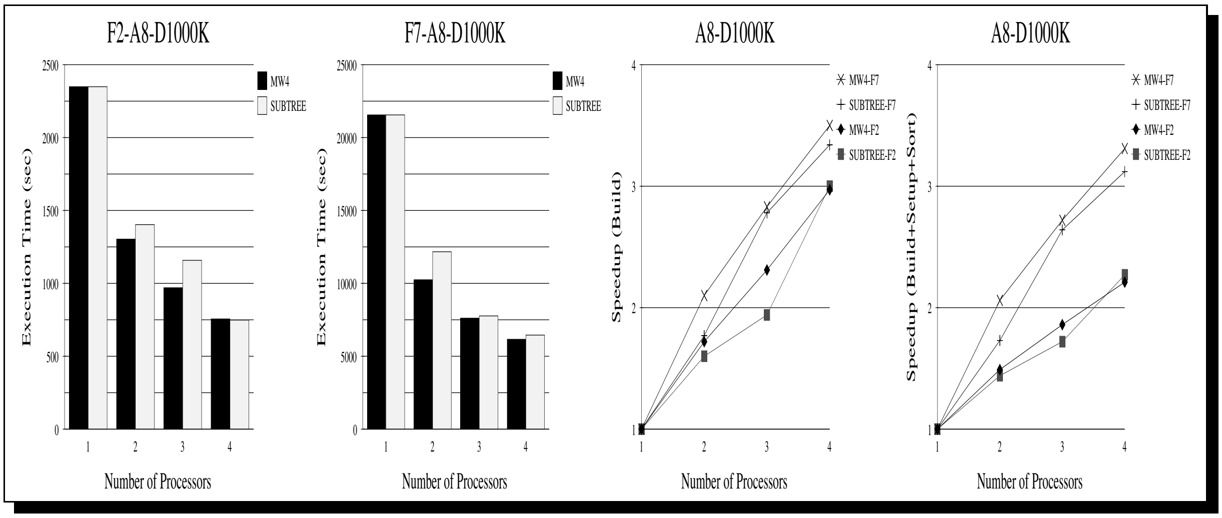

Figure 7. Parallel Performance: functions 2 and 7; 8 attributes; 1000K records.

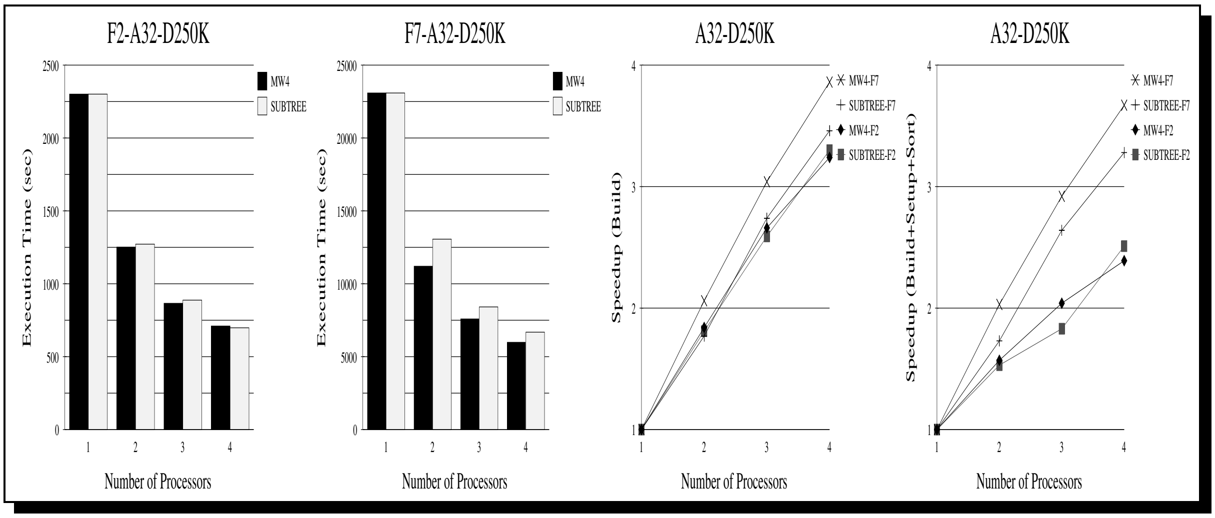

Figure 8. Parallel Performance: functions 2 and 7; 32 attributes; 250K records.

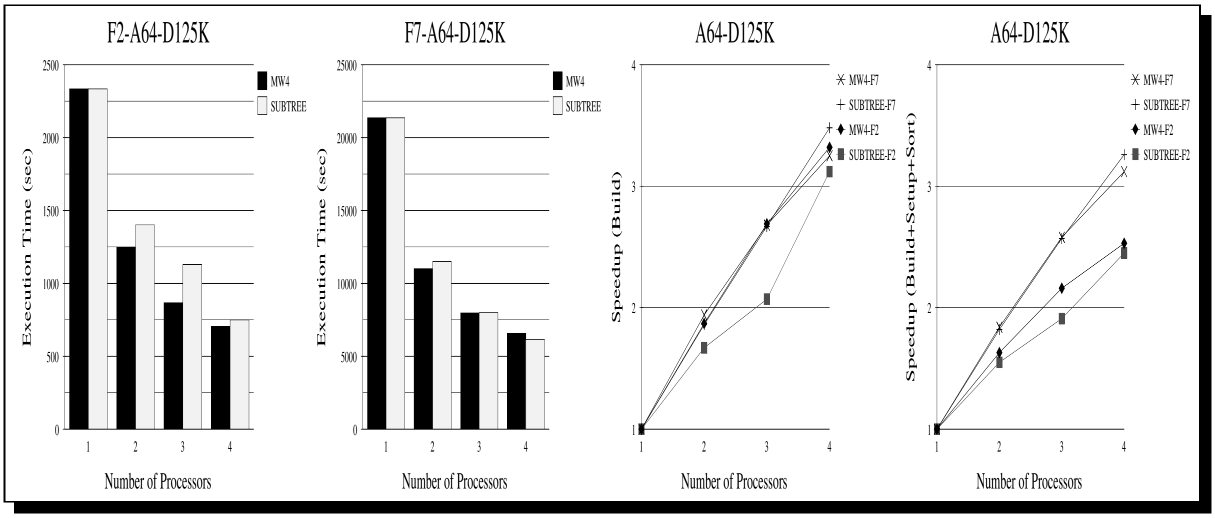

Figure 9. Parallel Performance: functions 2 and 7; 64 attributes; 125K records.