# Shared Memory Parallelization of Decision Tree Construction Using a General Data Mining Middleware[⋆]

Ruoming Jin and Gagan Agrawal

Department of Computer and Information Sciences
Ohio State University, Columbus, OH 43210
{jinr,agrawal}@cis.ohio-state.edu

## 1 Introduction

Decision tree construction is a very well studied problem in data mining, machine learning, and statistics communities [3,2,7,8,9]. The input to a decision tree construction algorithm is a database of *training records*. Each record has several *attributes*. An attribute whose underlying domain is totally ordered is called a *numerical* attribute. Other attributes are called *categorical* attributes. One particular attribute is called *class label*, and typically can hold only two values, true and false. All other attributes are referred to as *predictor* attributes.

A number of algorithms for decision tree construction have been proposed. In recent years, particular attention has been given to developing algorithms that can process datasets that do not fit in main memory [3,6,10]. Another development in recent years has been the emergence of more scalable shared memory parallel machines. To the best of our knowledge, there is only one effort on shared memory parallelization of decision tree construction on disk-resident datasets, which is by Zaki *et al.* [11].

In our previous work, we have developed a middleware for parallelization of data mining tasks on large SMP machines and clusters of SMPs. This middleware was used for apriori association mining, k-means clustering, and k-nearest neighbor classifiers [4,5]. In this paper, we demonstrate the use of the same middleware for decision tree construction. We particularly focus on parallelizing the *RainForest* framework for scalable decision tree construction [3].

The rest of the paper is organized as follows. We describe the original RainForest framework in Section 2. We review our middleware and parallelization techniques in Section 3. Our parallel algorithms and implementation are presented in Section 4. Section 5 presents the experimental results. We conclude in Section 6.

[⋆] This research was supported by NSF CAREER award ACI-9733520, NSF grant ACR-9982087, and NSF grant ACR-0130437.

## 2 Decision Tree Construction Using RainForest Framework

Though a large number of decision tree construction approaches have been used in the past, they are common in an important way. The decision tree is constructed in a top-down, recursive fashion. Initially, all training records are associated with the root of the tree. A criteria for splitting the root is chosen, and two or more children of this node are created. The training records are partitioned (physically or logically) between these children. This procedure is recursively applied, till either all training records associated with a node have the same class label, or the number of training records associated with a node is below a certain threshold. The different approaches for decision tree construction differ in the way criteria for splitting a node is selected, and the data-structures required for supporting the partitioning of the training sets.

RainForest is a general approach for scaling decision tree construction to larger datasets, while also effectively exploiting the available main memory. This is done by isolating an AVC (Attribute-Value, Classlabel) set for a given attribute and a node being processed. The size of the AVC-set for a given node and attribute is proportional to the number of distinct values of the attribute and the number of distinct class labels. For example, in a SPRINT like approach, AVC-set for a categorical attribute will simply be the count of occurrence of each distinct value the attribute can take. Therefore, the AVC-set can be constructed by taking one pass through the training records associated with the node.

A number of algorithms have been proposed within the RainForest framework to split decision tree nodes at lower levels. In this paper, we will mainly focus on parallelizing the RF-read algorithm. In the algorithm RF-read, the dataset is never partitioned. The algorithm progresses level by level. In the first step, AVC-group for the root node is built and a splitting criteria is selected. At any of the lower levels, all nodes at that level are processed in a single pass if the AVC-group for all the nodes fit in main memory. If not, multiple passes over the input dataset are made to split nodes at the same level of the tree. Because the training dataset is not partitioned, this can mean reading each record multiple times for one level of the tree.

## 3 Middleware and Parallelization Techniques

Our middleware is based upon the observation that a number of popular algorithms for association mining, clustering, and classification have a common structure. Specifically, the core of the computing in these algorithms follows a canonical loop that is shown in Figure 1. In our earlier work, we have argued how various association mining, clustering and classification algorithms have such a structure [5].

We next describe the parallelization techniques we use.

**Full Replication:** One simple way of avoiding race conditions is to replicate the reduction object and create one copy for every thread. The copy for each

```
{* Outer Sequential Loop *}
While() {
    {* Reduction Loop *}
    Foreach(element e) {
        (i, val)   =  Compute(e) ;
        RObj(i)    =  Reduc(RObj(i),val) ;
    }
}
```

**Fig. 1.** Canonical Loop Depicting the Structure of Common Data Mining Algorithms

thread needs to be initialized in the beginning. Each thread simply updates its own copy, thus avoiding any race conditions. After the local reduction has been performed using all the data items on a particular node, the updates made in all the copies are *merged*.

We next describe the locking schemes.

**Full Locking:** One obvious solution to avoiding race conditions is to associate one lock with every element in the reduction object. After processing a data item, a thread needs to acquire the lock associated with the element in the reduction object it needs to update.

This scheme has overheads of three types, doubled memory requirements, increased number of cache misses due to accessing locks and elements, and false sharing.

**Optimized Full Locking:** Optimized full locking scheme overcomes the the large number of cache misses associated with full locking scheme by allocating a reduction element and the corresponding lock in consecutive memory locations. This results in at most one cache miss when a reduction element and the corresponding lock is accessed.

**Cache-Sensitive Locking:** The final technique we describe is *cache-sensitive locking*. Consider a 64 byte cache block and a 4 byte reduction element. We use a single lock for all reduction elements in the same cache block. Moreover, this lock is allocated in the same cache block as the elements. So, each cache block will have 1 lock and 15 reduction elements.

Cache-sensitive locking reduces each of three types of overhead associated with full locking. This scheme results in lower memory requirements than the full locking and optimized full locking schemes. Each update operation results in at most one cache miss, as long as there is no contention between the threads. The problem of false sharing is also reduced because there is only one lock per cache block.

## 4 Parallel RainForest Algorithm and Implementation

In this section, we will present the algorithm and implementation details for parallelizing RF-read using our middleware. The algorithm is presented in Figure 5.

The algorithm takes several passes over the input dataset $D$. The dataset is organized as a set of chunks. During every pass, there are a number of nodes that are *active* or belong to the set $AQ$. These are the nodes for which AVC-group is built and splitting criteria is selected.

This processing is performed over three consecutive loops. In the first loop, the chunks in the dataset are read. For each training record or *tuple* in each chunk that is read, we determine the *node* at the current level to which it belongs. Then, we check if the node belongs to the set $AQ$. If so, we increment the elements in the AVC-group of the node.

The second loop finds the best splitting criteria for each of the active nodes, and creates the children. Before that, however, it must check if a *stop condition* holds for this node, and therefore, it need not be partitioned. For the nodes that are partitioned, no physical rewriting of data needs to be done. Instead, just the tree should be updated, so that future invocations to *classify* point to the appropriate children. The nodes that have been split are removed from the set $AQ$ and the newly created children are added to the set $Q$.

At the end of the second loop, the set $AQ$ is empty and the set $Q$ contains the nodes that still need to be processed. The third loop determines the set of the nodes that will be processed in the next phase. We iterate over the nodes in the set $Q$, remove a node from $Q$ and move it to $AQ$. This is done till either no more memory is available for AVC-groups, or $Q$ is empty.

The last loop contains only a very small part of the overall computing. Therefore, we focus on parallelizing the first and the second loop. Parallelization of the second loop is straight-forward and discussed first.

A simple multi-threaded implementation is used for the second loop. There is one thread per processor. This thread gets a node from the set $AQ$ and processes the corresponding AVC-group to find the best splitting criteria. The computing done for each node is completely independent. The only synchronization required is for getting a node from $AQ$ to process. This is implemented by simple locking.

Next, we focus on the first loop. Note that this loop fits nicely with the structure of the canonical loop we had shown in Figure 1. The set of AVC-groups for all nodes that are currently active is the reduction object. As different consumer threads try to update the same element in a AVC-set, race conditions can arise. The elements of the reduction object that are updated after processing a tuple cannot be determined without processing the tuple.

Therefore, the parallelization techniques we had presented in the last section are applicable to parallelizing the first loop. Both memory overheads and locking costs are important considerations in selecting the parallelization strategy. At lower levels of the tree, the total size of the reduction object can be very large. Therefore, memory overhead of the parallelization technique used is an important consideration. Also, the updates to the elements of the reduction object are fine-grained. After getting a lock associated with an element or a set of elements, the only computing performed is incrementing one value. Therefore, locking overheads can also be significant.

Next, we discuss the application of the techniques we have developed to parallelization of the first loop. Recall that the memory requirements of the three techniques are very different. If $R$ is the size of reduction object, $N$ is the size of consumer threads, and $L$ is the number of elements per cache line, the memory requirement of full replication, optimized full locking and cache sensitive locking are $N \times R$, $2 \times R$, and $\frac{N}{N-1} \times R$, respectively. This has an important implication for our parallel algorithm. Choosing a technique with larger memory requirements means that the set $AQ$ will be smaller. In other words, a larger number of passes over the dataset will be required.

An important property of the reduction object in RF-read is that updates to each AVC-set are independent. Therefore, we can apply different parallelization techniques to nodes at different levels, and for different attributes. Based upon this observation, we developed a number of approaches for applying one or more of the parallelization techniques we have. These approaches are, *pure*, *horizontal*, *vertical*, and *mixed*.

In the pure approach, the same parallelization approach is used for all AVC-sets, i.e., for nodes at different levels and for both categorical and numerical attributes.

The vertical approach is motivated by the fact that the sum of sizes of AVC-groups for all nodes at a level is quite small at upper levels of the tree. Therefore, full replication can be used for these levels without incurring the overhead of additional passes. Moreover, because the total number of elements in the reduction object is quite small at these levels, locking schemes can result in high overhead of waiting for locks and coherence cache misses. Therefore, in the vertical approach, replication is used for the first few levels (typically between 3 to 5) in the tree, and either optimized full locking or cache-sensitive locking is used at lower levels.

In determining the memory overheads, the cost of waiting for locks, and coherence cache misses, one important consideration is the number of distinct values of an attribute. If the number of the distinct values of an attribute is small, the corresponding AVC-set is small. Therefore, the memory overhead in replicating such AVC-sets may not be a significant consideration. At the same time, because the number of elements is small, the cost of waiting for locks and coherence cache misses can be significant. Note that typically, categorical attributes have a small number of distinct values and numerical attributes can have a large number of distinct values in a training set.

Therefore, in the horizontal approach, full replication is used for attributes with small number of distinct values, and one of the locking schemes is used for attributed with a large number of distinct values. For any attribute, the same technique is used at all levels of the tree.

Finally, the mixed strategy combines the two approaches. Here, full replication is used for all attributes at the first few levels, and for attributes with small number of distinct values at the lower levels. One of the locking schemes is used for the attributes with a large number of distinct values at lower levels of the tree.

## 5 Experimental Results

We used a Sun Fire 6800. Each processor in this machine is a 64 bit, 750 MHz Sun UltraSparc III. Each processor has a 64 KB L1 cache and a 8 MB L2 cache. The total main memory available is 24 GB. The Sun Fireplane interconnect provides a bandwidth of 9.6 GB per second. We experimented with up to 8 consumer threads on this machine.

The dataset we used for our experiments was generated using a tool described by Agrawal *et al.* [1]. The dataset is nearly 1.3 GB, with 32 million records in the training set. Each record has 9 attributes, of which 3 are categorical and other 6 are numerical. Every record belongs to 1 of 2 classes.

In Section 4, we had described *pure*, *vertical*, *horizontal*, and *mixed* approaches for using one or more of the parallelization techniques we support in the middleware. Based upon these, a total of 9 different versions of our parallel implementation were created. Obviously, there are three pure versions, corresponding to the use of full replication (`fr`), optimized full locking (`ofl`) and cache sensitive locking (`csl`). Optimized full locking can be combined with full replication using vertical, horizontal, and mixed approach, resulting in three versions. Similarly, cache sensitive locking can be combined with full replication using vertical, horizontal, and mixed approach, resulting in three additional versions, for a total of 9 versions.

Figure 2 shows the performance of pure versions. With 1 thread, `fr` gives the best performance. However, the parallel speedups are not good. This is because the the use of full replication for AVC-sets at all levels results in very high memory requirements. Locking schemes result in a 20 to 30% overhead on 1 thread, but the relative speedups are better. Using 8 threads, the relative speedups for `ofl` and `csl` are 5.37 and 4.95, respectively.

Figure 3 shows the experimental results from combining `fr` and `ofl`. As stated earlier, the two schemes can be combined in three different ways, horizontal, vertical, and mixed. The performance of these three versions is quite similar.

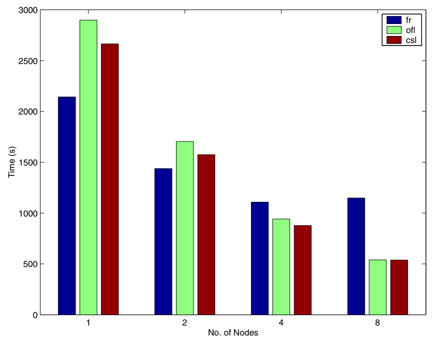

**Fig. 2.** Performance of *pure* versions

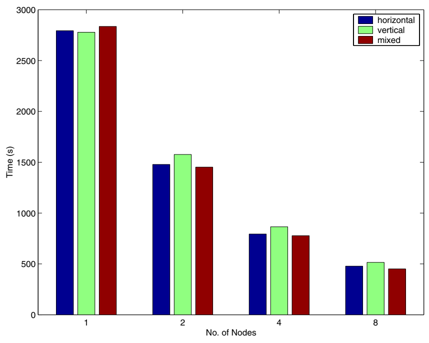

**Fig. 3.** Combining full replication and full locking, `dataset 2`

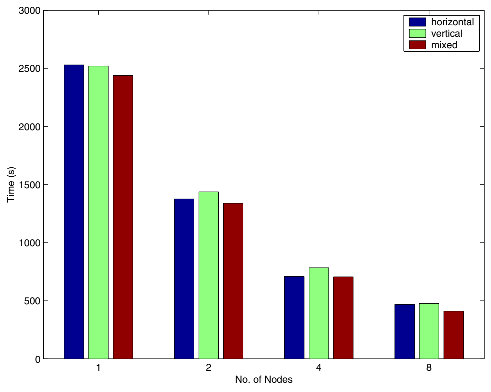

**Fig. 4.** Combining full replication and cache-sensitive locking

`vertical` is the slowest on 2, 4, and 8 threads, whereas `mixed` is the best on 2, 4, and 8 threads.

Figure 4 presents the experimental results from combining `fr` and `csl`. Again, the `mixed` version is the best among the three versions, for 2, 4, and 8 threads.

## 6 Conclusions

In this paper, we have presented a shared memory parallelization of a RainForest based decision tree construction algorithm, using a middleware framework we had developed in our earlier work.

Our work has lead to a number of interesting observations. First, we have shown that a RainForest based decision tree construction algorithm can be parallelized in a way which is very similar to the way association mining and clustering algorithms have been parallelized. Therefore, a general middleware framework for decision tree construction can simplify the parallelization of algorithms for a variety of mining tasks. Second, unlike the algorithms for other mining tasks, a combination of locking and replication based techniques results in the best speedups for decision tree construction. Thus, it is important that the framework used supports a variety of parallelization techniques.

## References

1. R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Eng., 5(6):914-925,*, December 1993.
2. J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh. Boat– optimistic decision tree construction. In *In Proc. of the ACM SIGMOD Conference on Management of Data*, June 1999.
3. J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB*, 1996.

```
**RF-Read**(dataset $\mathcal{D}$)
 **global** $Tree\ root$, $Queue\ \mathcal{Q}$ , $\mathcal{AQ}$;
 **local** $Node\ node$;
 $\mathcal{Q} \leftarrow NULL$; $\mathcal{AQ} \leftarrow NULL$;
 $add(root, \mathcal{AQ})$;
 **while** **not** ($empty(\mathcal{Q})$ **and** $empty(\mathcal{AQ})$)
      { Loop1: build AVC-group for nodes in $AQ$ }
      **foreach** ($chunk\ C \in \mathcal{D}$)
           **foreach** ($tuple\ t \in \mathcal{C}$)
                $node \leftarrow classify(root, t)$;
                **if** $node \in \mathcal{AQ}$
                     **foreach** ($attribute\ a \in\ t$)
                        $reduction(node.avc\_group, a, t.class)$;
      { Loop2: split the nodes in $AQ$ }
      **foreach** ($node \in \mathcal{AQ}$)
           **if** **not** $satisfy\_stop\_condition(node)$
                $find\_best\_split(node)$;
                **foreach** ($Node\ child \in\ create\_children(node)$)
                     $add(child, \mathcal{Q})$;
      { Loop3: build new $AQ$ }
      $\mathcal{AQ} \leftarrow NULL$;
      $done \leftarrow false$;
      **while** **not** $empty(\mathcal{Q})$ **and** **not** $done$
           $get(node, \mathcal{Q})$;
           **if** $enough\_memory(Q)$
                $remove(node, \mathcal{Q})$;
                $add(node, \mathcal{AQ})$;
             **else** $done \leftarrow true$;
```

**Fig. 5.** Algorithm for Parallelizing RF-read Using Our Middleware

4. Ruoming Jin and Gagan Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of the first SIAM conference on Data Mining*, April 2001.
5. Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, April 2002.
6. M. V. Joshi, G. Karypis, and V.Kumar. Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In *In Proc. of the International Parallel Processing Symposium*, 1998.
7. F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Knowledge Discovery and Data Mining*, 3, 1999.
8. J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

9. S. Ruggieri. Efficient c4.5. Technical Report TR-00-01, Department of Information, University of Pisa, February 1999.
10. J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, pages 544–555, September 1996.
11. M. J. Zaki, C.-T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. *IEEE International Conference on Data Engineering*, pages 198–205, May 1999.