Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance

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

Abstract

With recent technological advances, shared memory parallel machines have become more scalable, and offer large main memories and high bus bandwidths. They are emerging as good platforms for data warehousing and data mining. In this paper, we focus on shared memory parallelization of data mining algorithms.

We have developed a series of techniques for parallelization of data mining algorithms, including full replication, full locking, fixed locking, optimized full locking, and cache-sensitive locking. Unlike previous work on shared memory parallelization of specific data mining algorithms, all of our techniques apply to a large number of popular data mining algorithms. In addition, we propose a reduction-object based interface for specifying a data mining algorithm. We show how our runtime system can apply any of the technique we have developed starting from a common specification of the algorithm.

We have carried out a detailed evaluation of the parallelization techniques and the programming interface. We have experimented with apriori association mining, k-means clustering, k-nearest neighbor classifier, and decision tree construction. The main results from our experiments are as follows. 1) Among full replication, optimized full locking, and cache-sensitive locking, there is no clear winner. Each of these three techniques can outperform others depending upon machine and dataset parameters. These three techniques perform significantly better than the other two techniques. 2) Good parallel efficiency is achieved for each of the four algorithms we experimented with, using our techniques and runtime system. 3) The overhead of the interface is within 10% in almost all cases. 4) In the case of decision tree construction, combining different techniques turned out to be crucial for achieving high performance.

1 Introduction

With the availability of large datasets in application areas like bioinformatics, medical informatics, scientific data analysis, financial analysis, telecommunications, retailing, and marketing, it is becoming increasingly important to execute data mining tasks in parallel. At the same time, technological advances have made shared memory parallel machines commonly available to organizations and individuals. SMP machines with two to four processors are frequently used as desk-tops. Clusters of SMPs are very popular for high-end computing, and offer shared memory parallelism within each node. Shared memory machines are also becoming more scalable. Large shared memory machines

with high bus bandwidth and very large main memory are being sold by several vendors. Vendors of these machines are targeting data warehousing and data mining as major markets.

Using these machines for speeding up and scaling up data mining implementations, however, involves a number of challenges. First, the appropriate parallelization strategy could depend upon the data mining task and algorithm that is being parallelized. Second, with an increasingly complicated memory hierarchy, achieving high performance on SMP machines often requires subtle optimizations. Finally, maintaining, debugging, and performance tuning a parallel application is an extremely time consuming task. As parallel architectures evolve, or architectural parameters change, it is not easy to modify existing codes to achieve high performance on new systems. As new performance optimizations are developed, it is useful to be able to apply them to different parallel applications. Currently, this cannot be done for parallel data mining implementations without a high programming effort.

We believe that the above problems can be alleviated by developing parallelization techniques and runtime support that applies across a variety of data mining algorithms. In our research, we are developing a middleware for rapid development of data mining implementations on large SMPs and clusters of SMPs. Our system is called FREERIDE (FRamework for Rapid Implementation of Datamining Engines). It is based on the observation that parallel versions of several well-known data mining techniques, including apriori association mining [2], k-means clustering [21], k-nearest neighbor classifier [19] and artificial neural networks [19] share a relatively similar structure. The middleware performs distributed memory parallelization across the cluster and shared memory parallelization within each node. It enables high I/O performance by minimizing disk seek time and using asynchronous I/O operations. Thus, it can be used for developing efficient parallel data mining applications that operate on disk-resident datasets.

This paper focuses on shared memory parallelization. We have developed a series of techniques for runtime parallelization of data mining algorithms, including full replication, full locking, fixed locking, optimized full locking, and cache-sensitive locking. Unlike previous work on shared memory parallelization of specific data mining algorithms, all of our techniques apply across a large number of common data mining algorithms. The techniques we have developed involve a number of trade-offs between memory requirements, opportunity for parallelization, and locking overheads. Thus, the relative performance of these techniques depends upon machine parameters, as well as the characteristics of the algorithm and the dataset.

In addition, we present and evaluate the middleware interface, and the underlying runtime sup-

port for shared memory parallelization. We describe how a programmer can perform minor modifications to a sequential code and specify a data mining algorithm using the reduction object interface we have developed. We allow complex reduction objects and user-defined reduction functions, which are not available in OpenMP. We show how the runtime system can apply any of the technique we have developed starting from a common specification that uses the reduction object interface.

We initially evaluated our techniques and programming interface using our implementations of apriori association mining, k-means clustering, and k-nearest neighbor classifier. The main results from these experiments are as follows. 1) Among full replication, optimized full locking, and cache-sensitive locking, there is no clear winner. Each of these three techniques can outperform others depending upon machine and dataset parameters. These three techniques perform significantly better than the other two techniques. 2) Our techniques scale well on a large SMP machine. 3) The overhead of the interface is within 10% in almost all cases.

We have also carried out a detailed case study of applying our techniques and runtime system for decision tree construction. We have particularly focused on parallelization of the *RainForest* approach originally proposed by Gehrke *et al.* [12]. Typically, the techniques used for parallelizing decision tree construction have been quite different than the techniques used for association mining and clustering. Here, we have demonstrated that the full replication and either optimized full locking or cache-sensitive locking can be combined to achieve an efficient parallel implementation for decision tree construction.

Overall, our work has shown that a common collection of techniques can be used to efficiently parallelize algorithms for a variety of mining tasks. Moreover, a high-level interface can be supported to allow the programmers to rapidly create parallel implementations.

The rest of this paper is organized as follows. Section 2 reviews parallel versions of several common data mining techniques. Parallelization techniques are presented in Section 3. The middleware interface and implementation of different techniques starting from the common specification is described in Section 4. Experimental results from association mining, clustering, and k-nearest neighbor search are presented in Section 5. A detailed case study, decision tree construction, is presented in Section 6. We compare our work with related research efforts in Section 7 and conclude in Section 8.

2 Parallel Data Mining Algorithms

In this section, we describe how several commonly used data mining techniques can be parallelized on a shared memory machine in a very similar way. Our discussion focuses on five important data mining techniques: apriori associating mining [2], k-means clustering [21], k-nearest neighbors [19], artificial neural networks [19], and bayesian networks [9].

2.1 Apriori Association Mining

Association rule mining is the process of analyzing a set of transactions to extract *association rules* and is a very commonly used and well-studied data mining problem [3, 48]. Given a set of transactions[1] (each of them being a set of items), an association rule is an expression $X \rightarrow Y$, where X and Y are the sets of items. Such a rule implies that transactions in databases that contain the items in X also tend to contain the items in Y.

Formally, the goal is to compute the sets L_k. For a given value of k, the set L_k comprises the frequent itemsets of length k. A well accepted algorithm for association mining is the *apriori* mining algorithm [3]. The main observation in the apriori technique is that if an itemset occurs with frequency f, all the subsets of this itemset also occur with at least frequency f. In the first iteration of this algorithm, transactions are analyzed to determine the frequent 1-itemsets. During any subsequent iteration k, the frequent itemsets L_{k-1} found in the $(k-1)^{th}$ iteration are used to generate the candidate itemsets C_k. Then, each transaction in the dataset is processed to compute the frequency of each member of the set C_k. k-itemsets from C_k that have a certain pre-specified minimal frequency (called the *support level*) are added to the set L_k.

A simple shared memory parallelization scheme for this algorithm is as follows. One processor generates the complete C_k using the frequent itemset L_{k-1} created at the end of the iteration $k-1$. The transactions are scanned, and each transaction (or a set of transactions) is assigned to one processor. This processor evaluates the transaction(s) and updates the counts of candidates itemsets that are found in this transaction. Thus, by assigning different sets of transactions to processors, parallelism can be achieved. The only challenge in maintaining correctness is avoiding the possible race conditions when multiple processors may want to update the count of the same candidate.

The same basic parallelization strategy can be used for parallelization of a number of other association mining algorithms and variations of apriori, including SEAR [30], DHP [34], Partition [41],

[1] We use the terms *transactions*, *data items*, and *data instances* interchangeably.

and DIC [8]. These algorithm differ from the apriori algorithm in the data structure used for representing candidate itemsets, candidate space pruning, or in reducing passes over the set of transactions, none of which requires a significant change in the parallelization strategy.

2.2 k-means Clustering

The second data mining algorithm we describe is the k-means clustering technique [21], which is also very commonly used. This method considers transactions or data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters.

Three steps in the sequential version of this algorithm are as follows: 1) start with k given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it, assign this point to the corresponding cluster, and then move the center of the cluster closer to this point; and 3) repeat this process until the assignment of points to cluster does not change.

This method can also be parallelized in a fashion very similar to the method we described for apriori association mining. The data instances are read, and each data instance (or a set of instances) are assigned to one processor. This processor performs the computations associated with the data instance, and then updates the center of the cluster this data instance is closest to. Again, the only challenge in maintaining correctness is avoiding the race conditions when multiple processors may want to update center of the same cluster.

2.3 k-nearest Neighbors

k-nearest neighbor classifier is based on learning by analogy [19]. The training samples are described by an n-dimensional numeric space. Given an unknown sample, the k-nearest neighbor classifier searches the pattern space for k training samples that are closest, using the euclidean distance, to the unknown sample.

Again, this technique can be parallelized as follows. Each training sample is processed by one processor. After processing the sample, the processor determines if the list of k current nearest neighbors should be updated to include this sample. Again, the correctness issue is the race conditions if multiple processors try to update the list of nearest neighbors at the same time.

2.4 Artificial Neural Networks

An artificial neural network is a set of connected input/output units where each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class labels of the input samples. A very commonly used algorithm for training a neural network is *backpropagation* [19]. For each training sample, the weights are modified so as to minimize the difference between the network's prediction and the actual class label. These modifications are made in the backwards direction.

The straight forward method for parallelizing this techniques on a shared memory machine is as follows. Each transaction is assigned to one processor. This processor performs the computations associated with this transactions and updates the weights for each connection in the network. Again, the only correctness consideration is the possibility of race conditions when the weights are updated.

2.5 Bayesian Network

Bayesian network is an approach to unsupervised classification [9]. Each transaction or data instance X_i is represented as an ordered vector of attribute values $\{X_{i1}, \ldots, X_{ik}\}$. Given a set of data instances, the goal is to search for the best class descriptions that predict the data. Class membership is expressed probabilistically, i.e., a data instance probabilistically belongs to a number of possible classes. The classes provide probabilities for all attribute values of each instance. Class membership probabilities are then determined by combining all these probabilities.

Two most time consuming steps in computing the classification are `update_wts` and `update_parameters`. `update_wts` computes the weight of each class, which is the sum of the probabilities of each data instance being in that class. `update_parameters` uses the weights computed to update the parameters for classification used during the next phase.

A parallelization strategy that can be used for both of these steps is as follows. The data instances are partitioned across processors. In the `update_wts` phase, each processor updates the weight of each class after processing each data instance. The sequential version of `update_parameters` is composed of three nested loops. The outer most loop iterates over all the classes, the next loop iterates over all attributes, and the inner most loop iterates over the data instances. The inner most loop uses the values of all data instances to compute the class parameters. Since the data instances have been partitioned across processors, parallelization is done at the inner most loop. Processors update class parameters after processing each data instance. For both the phases, the correctness challenge is the

race condition when weights of the class or class parameters are updated.

3 Parallelization Techniques

This section focuses on parallelization techniques we have developed for data mining algorithms.

3.1 Overview of the Problem

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

Figure 1: Structure of Common Data Mining Algorithms

In the previous section, we have argued how several data mining algorithms can be parallelized in a very similar fashion. The common structure behind these algorithms is summarized in Figure 1. The function *op* is an associative and commutative function. Thus, the iterations of the foreach loop can be performed in any order. The data-structure *Reduc* is referred to as the reduction object.

The main correctness challenge in parallelizing a loop like this on a shared memory machine arises because of possible race conditions when multiple processors update the same element of the reduction object. The element of the reduction object that is updated in a loop iteration (i) is determined only as a result of the processing. For example, in the apriori association mining algorithm, the data item read needs to matched against all candidates to determine the set of candidates whose counts will be incremented. In the k-means clustering algorithm, first the cluster to which a data item belongs is determined. Then, the center of this cluster is updated using a reduction operation.

The major factors that make these loops challenging to execute efficiently and correctly are as follows:

- It is not possible to statically partition the reduction object so that different processors update disjoint portions of the collection. Thus, race conditions must be avoided at runtime.
- The execution time of the function *process* can be a significant part of the execution time of an iteration of the loop. Thus, runtime preprocessing or scheduling techniques cannot be applied.

- In many of algorithms, the size of the reduction object can be quite large. This means that the reduction object cannot be replicated or privatized without significant memory overheads.
- The updates to the reduction object are *fine-grained*. The reduction object comprises a large number of elements that take only a few bytes, and the foreach loop comprises a large number of iterations, each of which may take only a small number of cycles. Thus, if a locking scheme is used, the overhead of locking and synchronization can be significant.

3.2 Techniques

We have implemented five approaches for avoiding race conditions as multiple threads may want to update the same elements in the reduction object. These techniques are, *full replication*, *full locks*, *optimized full locks*, *fixed locks*, and *cache-sensitive locks*.

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 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*.

The other four techniques are based upon locking. The memory layout for these four techniques is shown in Figure 2.

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. For example, in the apriori association mining algorithm, there will be a lock associated with the count for each candidate, which will need to be acquired before updating that count. If two threads need to update the count of the same candidate, one of them will need to wait for the other one to release the lock. In apriori association mining, the number of candidates considered during any iteration is typically quite large, so the probability of one thread needing to wait for another one is very small.

In our experiment with apriori, with 2000 distinct items and support level of 0.1%, up to 3 million candidates were generated. In full locking, this means supporting 3 million locks. Supporting such a large numbers of locks results in overheads of three types. The first overhead is the high memory requirement associated with a large number of locks. The second overhead comes from cache misses.

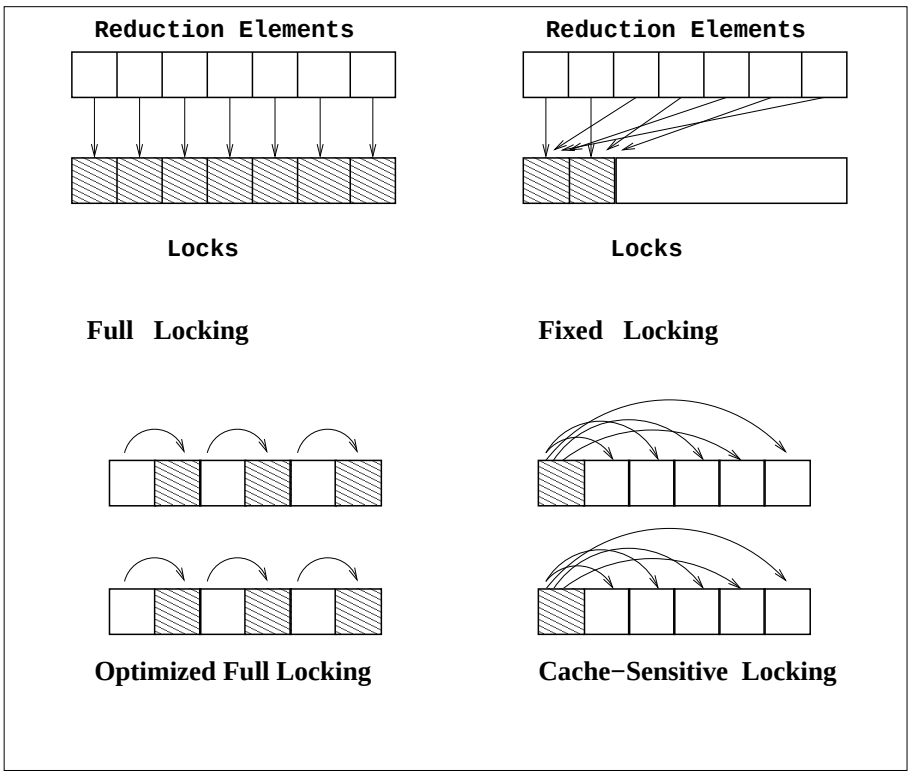

Figure 2: Memory Layout for Various Locking Schemes

Consider an update operation. If the total number of elements is large and there is no locality in accessing these elements, then the update operation is likely to result in two cache misses, one for the element and second for the lock. This cost can slow down the update operation significantly.

The third overhead is of *false sharing* [20]. In a cache-coherent shared memory multiprocessor, false sharing happens when two processors want to access different elements from the same cache block. In full locking scheme, false sharing can result in cache misses for both reduction elements and locks.

We have designed three schemes to overcome one or more of these three overheads associated with full locking. These three techniques are, *optimized full locks*, *fixed locks*, and *cache-sensitive locks*.

Optimized Full Locks: The next scheme we describe is optimized full locks. Optimized full locks 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, as shown in Figure 2. By appropriate alignment and padding, it can be ensured that the element and the lock are in the same cache block. Each update operation now results in at most one cold or capacity

cache miss. The possibility of false sharing is also reduced. This is because there are fewer elements (or locks) in each cache block. This scheme does not reduce the total memory requirements.

Fixed Locking: To alleviate the memory overheads associated with the large number of locks required in the full locking and optimized full locking schemes, we designed the fixed locking scheme. As the name suggests, a fixed number of locks are used. The number of locks chosen is a parameter to this scheme. If the number of locks is l, then the element i in the reduction object is assigned to the lock $i \ mod \ l$. So, in the apriori association mining algorithm, if a thread needs to update the support count for the candidate i, it needs to acquire the lock $i \ mod \ l$. In Figure 2, two locks are used. Alternate reduction elements use each of these two locks.

Clearly, this scheme avoids the memory overheads associated with supporting a large number of locks in the system. The obvious tradeoff is that as the number of locks is reduced, the probability of one thread having to wait for another one increases. Also, each update operation can still result in two cache misses. Fixed locking can also result in even more false sharing than the full locking scheme. This is because now there is a higher possibility of two processors wanting to acquire locks in the same cache block.

Cache-Sensitive Locking: The final technique we describe is *cache-sensitive locking*. This technique combines the ideas from optimized full locking and fixed 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.

This scheme results in lower memory requirements than the full locking and optimized full locking schemes. Similar to the fixed locking scheme, this scheme could have the potential of limiting parallelism. However, in cache-coherent multiprocessors, if different CPUs want to concurrently update distinct elements of the same cache block, they incur several cache misses. This is because of the effect of false sharing. This observation is exploited by the cache-sensitive locking scheme to have a single lock with all elements in a cache block.

Cache-sensitive locking reduces each of three types of overhead associated with full locking. 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.

One complication in implementation of cache-sensitive locking scheme is that modern processors

	Full Replication	Optimized Full Locks	Cache Sensitive Locks	Full Locks	Fixed Locks
Memory Requirement	very high	high	low	high	low
Parallelism	very high	high	medium	high	low
Locking Overhead	none	medium	high	medium	high
Cache Misses	low	medium	low	high	medium
False Sharing	none	yes	none	yes	yes
Merging Costs	yes	none	none	none	none

Table 1: Tradeoff Among the Techniques

have 2 or more levels of cache and the cache block size is different at different levels. Our implementation and experiments have been done on machines with two levels of cache, denoted by L1 and L2. Our observation is that if the reduction object exceeds the size of L2 cache, L2 cache misses are a more dominant overhead. Therefore, we have used the size of L2 cache in implementing the cache-sensitive locking scheme.

3.3 Comparing the Techniques

We now compare the five techniques we have presented along six criteria. The comparison is summarized in Table 1. The six criteria we use are:

- Memory requirements, denoting the extra memory required by a scheme because of replication or locks.
- Parallelism, denoting if parallelism is limited because of contention for locks.
- Locking overhead, which includes the cost of acquiring and releasing a lock and any extra computational costs in computing the address of the lock associated with the element.
- Cache misses, which only includes cold and capacity cache misses, and excludes coherence cache misses and false sharing.
- False sharing, which occurs when two processors want to access reduction elements or locks on the same cache block.
- Merge costs denotes the cost of merging updates made on replicated copies.

Full replication has the highest memory requirement. If the size of reduction object is S and there are T threads, then the total memory requirement is $S \times T$. Full locking and optimized full

locking are next in the level of memory requirement. If each lock takes the same number of bytes as each reduction element, the total memory requirement for both these schemes is $2 \times S$. Fixed locking and cache-sensitive locking have the lowest memory requirements. If one lock is used for every r reduction elements, total memory requirement in both the schemes is $(1 + 1/r) \times S$.

Full replication scheme does not limit parallelism in any way because each thread updates its own copy of reduction object. Full locks and optimized full locks are next in the level of parallelism. The probability of two threads trying to acquire the same lock is very small in both these schemes. Cache-sensitive and fixed locks can both limit parallelism because of sharing of locks.

The next criteria we address is the computation overhead because of locking. Full replication does not have such overhead. All locking schemes involve the cost of acquiring and releasing a lock. In addition, fixed locking and cache-sensitive locking require extra computation to determine the locks that needs to be used for an element.

Cache misses can be a significant cost when the total memory required by reduction object and locks is large. In full replication, there is no increase in working set size or the size of memory accessed by each processor because of locks. Only the reduction element needs to be accessed during the update operation. Replicating the reduction object does not increase the number of cold or capacity misses, because each processor has its own cache and accesses its own copy of the reduction object. As we discussed earlier, full locking scheme is the worst with respect to cold and capacity misses. The locks double the working set size at each processor. Further, each update operation results in accesses to two cache blocks.

We have denoted the cache misses for both fixed locking and optimized full locking as *medium*. In fixed locking, the total working set size increases only marginally because of locks. However, each update operation can still result in two cache misses. In optimized full locking, the working set size is the same as in full locking. However, each update operation needs to access only a single cache block.

Cache-sensitive locking is the best one among all locking schemes with respect to cache misses. Like fixed locking, the working set size is increased only marginally because of locks. Also, each update operation needs to access only a single cache block.

In our implementation, read-only data is segregated from the reduction object. Therefore, there is no false sharing in full replication. In full locking, optimized full locking, and fixed locking two threads can access the same cache block comprising either reduction elements or locks, and incur the

cost of false sharing. Cache-sensitive locking does not have any false sharing. This is because there is only one lock in a cache block and two threads cannot simultaneously acquire the same lock. Any cache misses incurred by a thread waiting to acquire a lock are considered a part of the waiting cost.

Our final criteria is the cost of merging. This cost is only incurred by the full replication scheme.

4 Programming Interface

In this section, we explain the interface we offer to the programmers for specifying a parallel data mining algorithm. We also describe how each of the five techniques described in the previous section can be implemented starting from a common specification.

```
void Kmeans::initialize() {
  for (int i = 0; i < k; i++) {
    clusterID[i]=reducobject->alloc(ndim + 2);
  }
  {* Initialize Centers *}
}
void Kmeans::reduction(void *point) {
  for (int i=0; i < k; i++) {
    dis=distance(point, i);
    if (dis < min) {
      min=dis;
      min_index=i;
    }
  }
  objectID=clusterID[min_index];
  for (int j=0; j< ndim; j++)
    reducobject->Add(objectID, j, point[j]);
  reducobject->Add(objectID, ndim, 1);
  reducobject->Add(objectID, ndim + 1, dis);
}
```

Figure 3: Initialization and Local Reduction Functions for k-means

4.1 Middleware Interface

As we stated earlier, this work is part of our work on developing a middleware for rapid development of data mining implementations on large SMPs and clusters of SMPs [23, 22]. Our middleware exploits the similarity in both shared memory and distributed memory parallelization strategy to offer a high-level interface to the programmers. For shared memory parallelization, the programmer is responsible for creating and initializing a reduction object. Further, the programmer needs to write a local reduction function that specifies the processing associated with each transaction. The

initialization and local reduction functions for k-means are shown in Figure 3.

As we discussed in Section 3.1, a common aspect of data mining algorithms is the *reduction object*. Declaration and allocation of a reduction object is a significant aspect of our middleware interface. There are two important reasons why reduction elements need to be separated from other data-structures. First, by separating them from read-only data-structures, false sharing can be reduced. Second, the middleware needs to know about the reduction object and its elements to optimize memory layout, allocate locks, and potentially replicate the object.

Consider, as an example, apriori association mining algorithm. Candidate itemsets are stored in a prefix or hash tree. During the reduction operation, the interior nodes of the tree are only read. Associated with each leaf node is the support count of the candidate itemset. All such counts need to be allocated as part of the reduction object. To facilitate updates to the counts while traversing the tree, pointers from leaf node to appropriate elements within the reduction object need to be inserted. Separate allocation of candidate counts allows the middleware to allocate appropriate number of locks depending upon the parallelization scheme used and optimize the memory layout of counts and locks. If full replication is used, the counts are replicated, without replicating the candidate tree. Another important benefit is avoiding or reducing the problem of false sharing. Separate allocation of counts ensures that the nodes of the tree and the counts of candidates are in separate cache blocks. Thus, a thread cannot incur false sharing misses while traversing the nodes of the tree, which is now a read-only data-structure. A disadvantage of separate allocation is that extra pointers need to be stored as part of the tree. Further, there is extra pointer chasing as part of the computation.

Two granularity levels are supported for reduction objects, the *group* level and the *element* level. One group is allocated at a time and comprises a number of elements. The goal is to provide programming convenience, as well as high performance. In apriori, all k itemsets that share the same parent $k-1$ itemsets are typically declared to be in the same group. In k-means, a group represents a center, which has $ndim + 2$ elements, where $ndim$ is the number of dimensions in the coordinate space.

After the reduction object is created and initialized, the runtime system may *clone* it and create several copies of it. However, this is transparent to the programmer, who views a single copy of it.

The reduction function shown in Figure 3 illustrates how updates to elements within a reduction object are performed. The programmer writes sequential code for processing, except the updates to elements within a reduction object are performed through member functions of the reduction object.

```
template<class T>
inline void ReducObject<T>::Reduc(int ObjectID,
            int Offset, void (*func)(void *, void *),
            int *param) {
  T *group_address =reducgroup[ObjectID];
  SWITCH (TECHNIQUE) {
    CASE FULL_REPLICATION:
      func(group_address[Offset],param);
      break;
    CASE FULL_LOCKS:
      offset = abs_offset(ObjectID,Offset);
      S_LOCK(&locks[offset]);
      func(group_address[Offset],param);
      S_UNLOCK(&locks[offset]);
      break;
    CASE FIXED_LOCKS:
      offset = abs_offset(ObjectID,Offset);
      S_LOCK(&locks[offset%num_locks]);
      func(group_address[Offset],param);
      S_UNLOCK(&locks[offset%num_locks]);
      break;
    CASE OPTIMIZE_FULL_LOCKS:
      S_LOCK(&group_address[Offset*2]);
      func(group_address[Offset*2+1],value);
      S_UNLOCK(&group_address[Offset*2]);
      break;
    CASE CACHE_SENSITIVE_LOCKS:
      cache_index = divide15(Offset);
      S_LOCK(&group_address[cache_index*16]);
      func(group_address[Offset+cache_index+1],value);
      S_UNLOCK(&group_address[cache_index*16]);
      break;
  }
}
```

Figure 4: Implementation of Different Parallelization Techniques Starting from a Common Specification

A particular element in the reduction object is referenced by a group identifier and an offset within the group. In this example, *add* function is invoked for all elements. Besides supporting the commonly used reduction functions, like addition, multiplication, maximum, and minimum, we also allow user defined functions. A function pointer can be passed a parameter to a generic reduction function. The reduction functions are implemented as part of our runtime support. Several parallelization strategies are supported, but their implementation is kept transparent from application programmers.

After the reduction operation has been applied on all transactions, a *merge* phase may required, depending upon the parallelization strategy used. If several copies of the reduction object have been created, the merge phase is responsible for creating a single correct copy. We allow the application programmer to choose between one of the standard merge functions, (like add corresponding elements from all copies), or to supply their own function.

4.2 Implementation From the Common Interface

Outline of the implementation of these five techniques is shown in Figure 4. Implementation of a general reduction function *reduc*, which takes a function pointer *func* and a pointer to a parameter

param, is shown in this figure.

The reduction element is identified by an *ObjectID* and an *Offset*. The operation $reducgroup[ObjectID]$ returns the starting address of the group to which the element belongs. This value is stored in variable $group\_address$. The function abs_offset returns the offset of an element from the start of allocation of first reduction group.

Implementation of full replication is straight forward. The function $func$ with the parameter *param* is applied to the reduction element.

Implementation of full locks is also simple. If $offset$ is the offset of an element from start of the allocation of reduction objects, $locks[offset]$ denotes the lock that can be used for this element. We use simple *spin locks* to reduce the locking overhead. Since we do not consider the possibility of executing more than one thread per processor, we do not need to block a thread that is waiting to acquire a lock. In other words, a thread can simply keep spinning till it acquires the lock. This allows us to use much simpler locks than the ones used in posix threads. The number of bytes taken by each spin lock is either 8 or the number of bytes required for storing each reduction element, whichever is smaller. These locks reduce the memory requirements and the cost of acquiring and releasing a lock.

The only difference in the implementation of fixed locking from full locking is that a *mod* operation is performed to determine which lock is used.

However, the implementations of *optimized full locking* and *cache-sensitive locking* are more involved. In both cases, each reduction group is allocated combining the memory requirements of the reduction elements and locks. In optimized full locking scheme, given an element with a particular *Offset*, the corresponding lock is at $group\_address + Offset * 2$ and the element is at $group\_address + Offset * 2 + 1$. In cache-sensitive locking, each reduction object is allocated at the start of a cache block. For simplification of our presentation, we assume that a cache block is 64 bytes and each element or lock takes 4 bytes. Given an element with a particular $Offset$, $Offset/15$ determines the cache block number within the group occupied by this element. The lock corresponding to this element is at $group\_address + Offset/15 \times 16$. The element itself is at $group\_address + Offset + Offset/15 + 1$.

Implementation of cache-sensitive locking involves a division operation that cannot be implemented using shifts. This can add significant overhead to the cache-sensitive locking scheme. To reduce this overhead, we use special properties of 15 (or 7) to implement a *divide15* (or *divide7*)

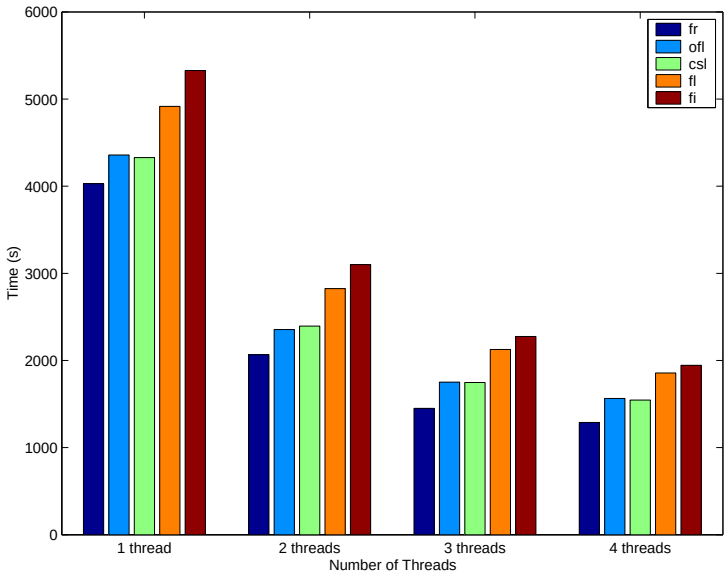

Figure 5: Comparing all Five Techniques for Apriori

function.

5 Experimental Results

We have so far implemented three data mining algorithms using our interface for shared memory parallelization. These algorithms are, apriori association mining, k-means clustering, and k-nearest neighbor classifier. We conducted a series of experiments to evaluate the following:

- Parallel performance achieved using the techniques we have developed for parallelization of data mining algorithms.
- The overhead introduced by the interface we have developed, i.e., the relative difference in the performance between the versions that use our interface and the versions that apply the same parallelization technique manually.

Through out this section, the program versions in which a parallelization technique was implemented by hand are referred to as *manual* versions, and versions where parallelization was done using the middleware interface are referred to as *interface* versions.

5.1 Experimental Platform

We used two different SMP machines for our experiments.

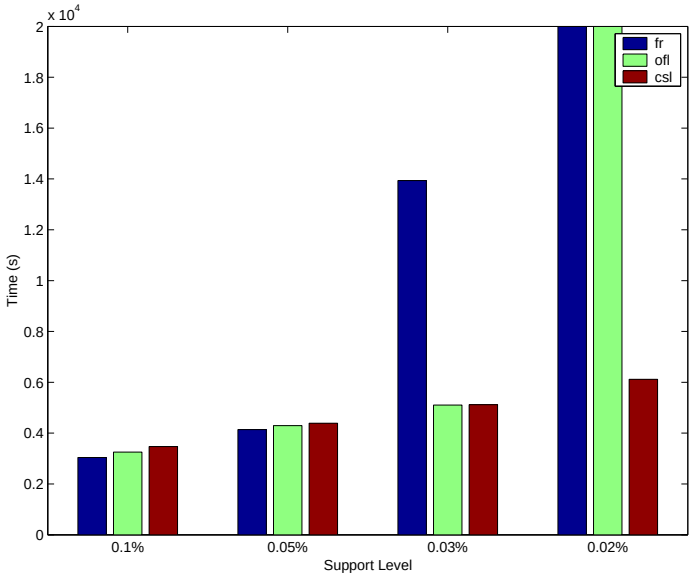

Figure 6: Relative Performance of Full Replication, Optimized Full Locking, and Cache-Sensitive Locking: 4 Threads, Different Support Levels

The first machine is a Sun Microsystem Ultra Enterprise 450, with 4 250MHz Ultra-II processors and 1 GB of 4-way interleaved main memory. This configuration represents a common SMP machine available as a desk-top or as part of a cluster of SMP workstations.

The second machine used for our experiments is a 24 processor SunFire 6800. Each processor in this machine is a 64 bit, 900 MHz Sun UltraSparc III. Each processor has a 96 KB L1 cache and a 64 MB L2 cache. The total main memory available is 24 GB. The Sun Fireplane interconnect provides a bandwidth of 9.6 GB per second. This configuration represents a state-of-the-art server machine that may not be affordable to all individuals or organizations interested in data mining.

On both the platforms, g++ compiler with -O3 option was used for all our experiments.

5.2 Results from Apriori

Our first experiment focused on evaluating all five techniques. Since we were interested in seeing the best performance that can be obtained from these techniques, we used only manual versions of each of these techniques. We used a 1 GB dataset. The total number of distinct items was 1000 and the average number of items in a transaction was 15. A confidence of 90% and support of 0.5 is used. The results are presented in Figure 5.

The versions corresponding to the full replication, optimized full locking, cache-sensitive locking, full locking and fixed locking are denoted by `fr`, `ofl`, `csl`, `fl`, and `fi`, respectively. Execution times

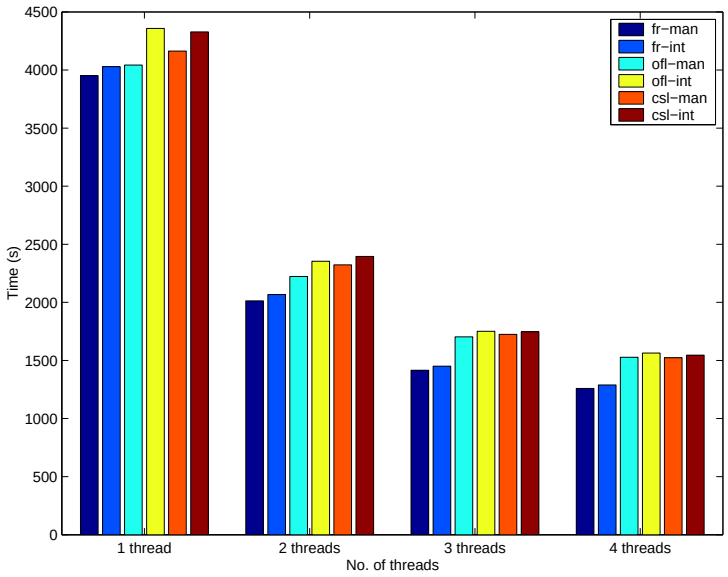

Figure 7: Scalability and Middleware Overhead for Apriori: 4 Processor SMP Machine

using 1, 2, 3, and 4 threads are presented on the 4 processor SMP machine. With 1 thread, `fr` does not have any significant overheads as compared to the sequential version. Therefore, this version is used for reporting all speedups. With 1 thread, `ofl` and `csl` are slower by nearly 7%, `fl` is slower by 22%, and `fi` is slower by 30%. For this dataset, even after replicating 4 times, the reduction object did not exceed the main memory. Therefore, `fr` has the best speedups. The speedup with 4 threads is 3.12 for `fr`, 2.58 with `ofl`, 2.60 with `csl`, 2.16 with `fl`, and 2.07 with `fi`.

From this experiment and our discussion in Section 3.3, `fl` and `fi` do not appear to be competitive schemes. Though the performance of `ofl` and `csl` is considerably lower than `fr`, they are promising for the cases when sufficient memory for supporting full replication may not be available. Therefore, in the rest of this section, we only focus on full replication, optimized full locking, and cache-sensitive locking.

Our second experiment demonstrates that each of these three techniques can be the winner, depending upon the problem and the dataset. We use a dataset with 2000 distinct items, where the average number of items per transaction is 20. The total size of the dataset is 500 MB and a confidence level of 90% is used. We consider four support levels, 0.1%, 0.05%, 0.03%, and 0.02%. Again, since we were only interested in relative performance of the three techniques, we experimented with manual version only.

The results are shown in Figure 6. We present results on the 4 processor SMP using 4 threads.

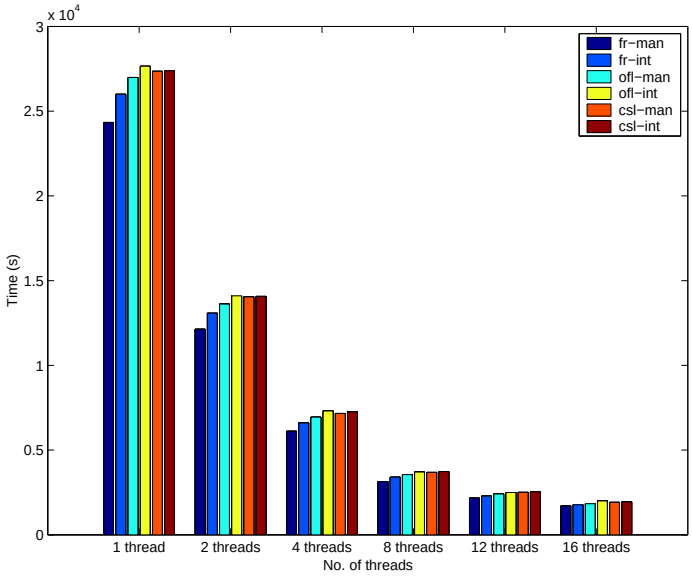

Figure 8: Scalability and Middleware Overhead for Apriori: Large SMP Machine

In apriori association mining, the total number of candidate item-sets increases as the support level is decreased. Therefore, the total memory requirement for the reduction objects also increases. When support level is 0.1% or 0.05%, sufficient memory is available for reduction object even after replicating 4 times. Therefore, `fr` gives the best performance. At the support level of 0.1%, `ofl` is slower by 7% and `csl` is slower by 14%. At the support level of 0.05%, they are slower by 4% and 6%, respectively. When the support level is 0.03%, the performance of `fr` degrades dramatically. This is because replicated reduction object does not fit in main memory and memory thrashing occurs. Since the memory requirements of locking schemes are lower, they do not see the same effect. `ofl` is the best scheme in this case, though `csl` is slower by less than 1%. When the support level is 0.02%, the available main memory is not even sufficient for `ofl`. Therefore, `csl` has the best performance. The execution time for `csl` was 6,117 seconds, whereas the execution time for `ofl` and `fr` was more than 80,000 seconds.

The next two experiments evaluated scalability and middleware overhead on 4 processor and large SMP, respectively. We use the same dataset as used in the first experiment. We created manual and interface versions of each of the three techniques, full replication, optimized full locking, and cache sensitive locking. Thus, we had six versions, denoted by `fr-man`, `fr-int`, `ofl-man`, `ofl-int`, `csl-man`, and `csl-int`.

Results on 4 processor SMP are shown in Figure 7. The results of manual versions are the same

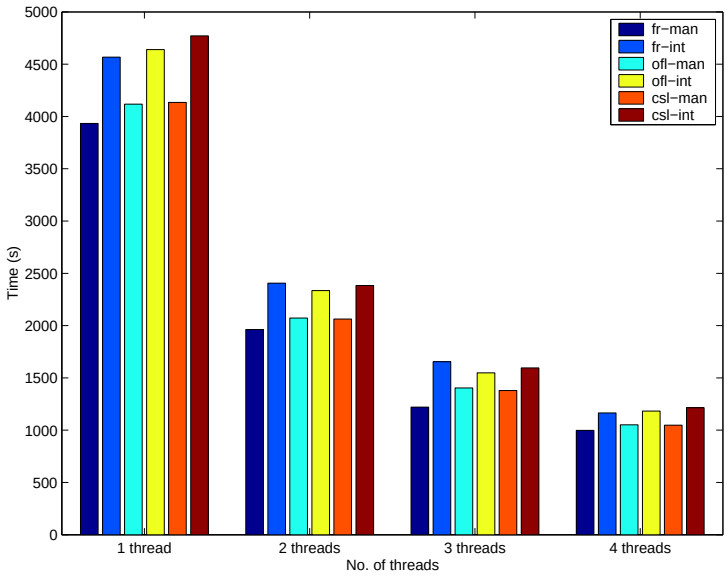

Figure 9: Scalability and Middleware Overhead for k-means: 4 Processor SMP Machine

as ones presented in Figure 5. The overhead of middleware's general interface is within 5% in all but two cases, and within 10% in all cases. The overhead of middleware comes primarily because of extra function calls and pointer chasing.

Results on the large SMP machine are shown in Figure 8. We were able to use only up to 16 processors of this machine at any time. We have presented experimental data on 1, 2, 4, 8, 12, and 16 threads. Because the total memory available is very large, sufficient memory is always available for `fr`. Therefore, `fr` always gives the best performance. However, the locking versions are slower by at most 15%.

One interesting observation is that all versions have a uniformly high relative speedup from 1 to 16 threads. The relative speedups for six versions are 14.30, 14.69, 14.68, 13.85, 14.29, and 14.10, respectively. This shows that different versions incur different overheads with 1 thread, but they all scale well. The overhead of middleware is within 10% in all cases. In some cases, it is as low as 2%.

5.3 Results from k-means

For evaluating our implementation of k-means, we used a 200 MB dataset comprising three-dimensional points. The value of k we used was 1000. The total size of the reduction object is much smaller in k-means as compared to apriori.

We focused on three techniques that produced competitive performance for apriori, i.e. full replication, optimized full locking, and cache-sensitive locking. We conducted experiments to eval-

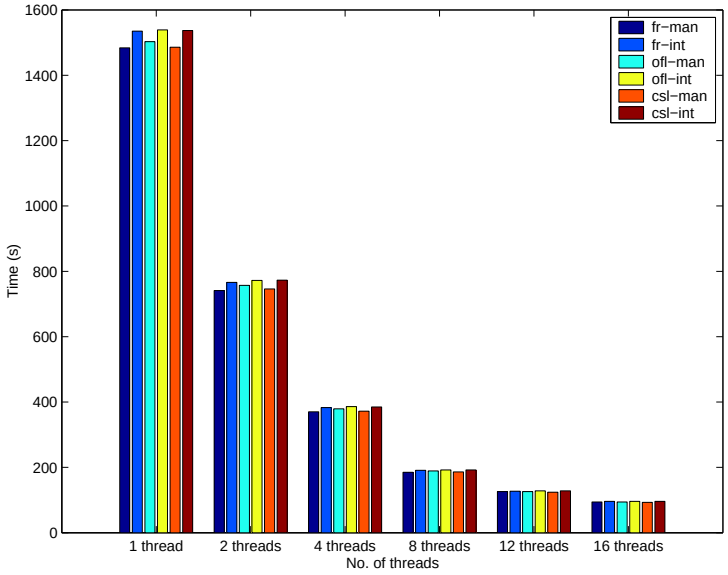

Figure 10: Scalability and Middleware Overhead for k-means: Large SMP Machine

uate scalability, relative performance, and middleware overheads on the 4 processor and large SMP machines.

The results on 4 processor machine are presented in Figure 9. As the memory requirements of reduction object are relatively small, full replication gives the best performance. However, the locking versions are within 5%. The relative speedups of six versions are 3.94, 3.92, 3.92, 3.93, 3.94, and 3.92, respectively. Thus, after the initial overhead on 1 thread versions, all versions scale almost linearly. The middleware overhead is up to 20% with k-means, which is higher than that from apriori. This is because the main computation phase of k-means involves accessing coordinates of centers, which are part of the reduction object. Therefore, extra point chasing is involved when middleware is used. The manual versions can simply declare an array comprising all centers, and avoid the extra cost.

The results on large SMP machine are presented in Figure 10. Six versions are run with 1, 2, 4, 8, 12, and 16 threads. Though full replication gives the best performance, locking versions are within 2%. The relative speedups in going from 1 thread to 16 threads for the six versions are 15.78, 15.98, 15.99, 16.03, 15.98, and 16.01, respectively. In other words, all versions scale linearly up to 16 threads. The overhead of the interface is significantly lower as compared to the 4 processor machine. We believe this is because the newer UltraSparc III processor performs aggressive out-of-order issues and can hide some latencies.

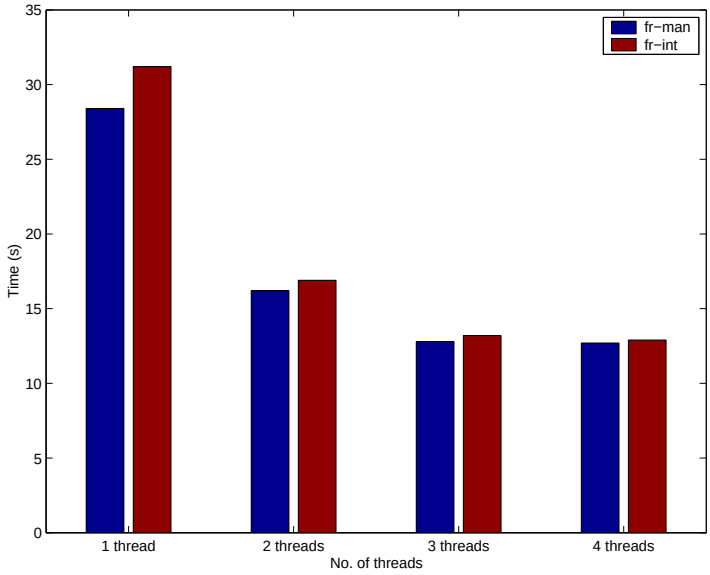

Figure 11: Parallelization of k-nearest Neighbors

5.4 Results from k-nearest Neighbors

The last data mining algorithm we consider is k-nearest neighbors. We have experimented with a 800 MB main memory resident dataset. The value of k used is 2000. The reduction object in this algorithm's parallel implementation is the list of k-nearest neighbors. This is considered a single element. Because of this granularity, only full replication scheme was implemented for this algorithm.

Figure 11 presents experimental results from `fr-man` and `fr-int` versions. The speedups of manual version are 1.75, 2.22, and 2.24 with 2, 3, and 4 threads, respectively. The speedups are limited because only a small amount of computation is associated with each transaction. The overhead of the use of the interface is within 10% in all cases. Because of the limited computation in this code, we did not experiment further with the large SMP machine.

6 A Detailed Case Study: Decision Tree Construction

We now describe our experience in parallelizing decision tree construction using the parallelization techniques and middleware described earlier in the paper.

Decision tree construction is a very well studied problem in data mining, machine learning, and statistics communities [12, 11, 31, 33, 37, 38, 39]. 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 an *ordered*, *numerical* or *continuous* 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 the *predictor* attributes. A decision tree construction algorithm processes the set of training records, and builds a model which is used for predicting the class label of new records. 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 [12, 24, 42].

A lot of effort has been put into developing parallel algorithms for decision tree construction [4, 13, 24, 29, 42, 44, 46]. Most of these efforts have targeted distributed 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.* [46].

Usually, the parallelization approach taken for decision tree construction is quite different than the approach taken for other common data mining algorithms, like association mining and clustering. Parallelization of decision tree construction typically involves sorting of numerical attributes and/or frequently writing back of input data. Therefore, an important question is, *"Can the parallelization techniques and runtime support that are suitable for association mining and clustering also be effective for decision tree construction?"*. Here, we demonstrate that this is indeed the case. We particularly focus on parallelizing the *RainForest* framework for scalable decision tree construction [12]. We believe that our effort is the first on parallelizing RainForest based decision tree construction.

6.1 Decision Tree Construction Using RainForest Framework

RainForest is a general framework for scaling decision tree construction [12]. The key observation that motivates this approach is as follows. 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 approach scales the decision tree construction process to larger (disk-resident) 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.

Given a node of the decision tree, AVC-group is the combination of AVC-set for all attributes. The key observation is that though AVC-group does not contain sufficient information to reconstruct the training dataset, it contains all information that is required for selecting the criteria for splitting the node. Since the number of attributes and the distinct values they can take is usually not very large, one can expect the AVC-group for a node to easily fit in main memory. With this observation, processing for selecting the splitting criteria for the root node can be easily performed even if the dataset is disk-resident. By reading the training dataset once, AVC-group of the root is constructed. Then, the criteria for splitting the node is selected.

A number of algorithms have been proposed within the RainForest framework to split decision tree nodes at lower levels. 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.

Another algorithm, RF-write, partitions and rewrites the dataset after each pass. The algorithm RF-hybrid combines the previous two algorithms. Overall, RF-read and RF-hybrid algorithms are able to exploit the available main memory to speedup computations, but without requiring the dataset to be main memory resident.

In our work, we will mainly focus on parallelizing the RF-read algorithm. There are several reasons for this. First, the only difference between RF-read, RF-write and RF-hybrid is the frequency of writing back. RF-read could be looked as the main computing subroutine to be called for every physical partition, and includes the dominant processing for them, i.e., building AVC-groups and finding the criteria for splitting. Secondly, the main memory size of SMP machines has been growing

at a rapid speed in recent years. This makes RF-read more practical even when the decision tree constructed is quite deep. Finally, because RF-read does not require the data to be written back, it fits into the structure of algorithms for association mining and clustering, and suits our middleware.

The next two subsections describe our existing middleware and the parallelization techniques supported, and our approach for parallelizing RF-read.

6.2 Parallel RainForest Algorithm and Implementation

In this subsection, we will present the algorithm and implementation details for parallelizing RF-read using our middleware. The algorithm is presented in Figure 12.

```
RF-Read(dataset D)
 global Tree root, Queue Q , AQ;
 local Node node;
 Q ← NULL; AQ ← NULL;
 add(root, AQ);
 while not (empty(Q) and empty(AQ))
      { Loop1: build AVC-group for nodes in AQ}
      foreach (chunk C ∈ D)
           foreach (tuple t ∈ C)
                node ← classify(root, t);
                if node ∈ AQ
                     foreach (attribute a ∈ t)
                          reduction(node.avc_group, a, t.class);
      { Loop2: split the nodes in AQ}
      foreach (node ∈ AQ)
           if not satisfy_stop_condition(node)
                find_best_split(node);
                foreach (Node child ∈ create_children(node))
                     add(child, Q);
      { Loop3: build new AQ}
      AQ ← NULL;
      done ← false;
      while not empty(Q) and not done
           get(node, Q);
           if enough_memory(Q)
                remove(node, Q);
                add(node, AQ);
             else done ← true;
```

Figure 12: Algorithm for Parallelizing RF-read Using Our Middleware

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.

Parallelization of the first loop is facilitated by the producer consumer framework we support. The producer thread reads the chunks, and put them in a queue. Consumer threads grab chunks from the queue and perform the computing associated with these. The main problem is ensuring correctness as consumer threads process training records from different chunks. Note that the first 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 have developed 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.

6.3 Experimental Results

In this subsection, we evaluate our implementation of decision tree construction. Since our primary consideration was evaluating scalability, we only evaluated the performance on SunFire 6800.

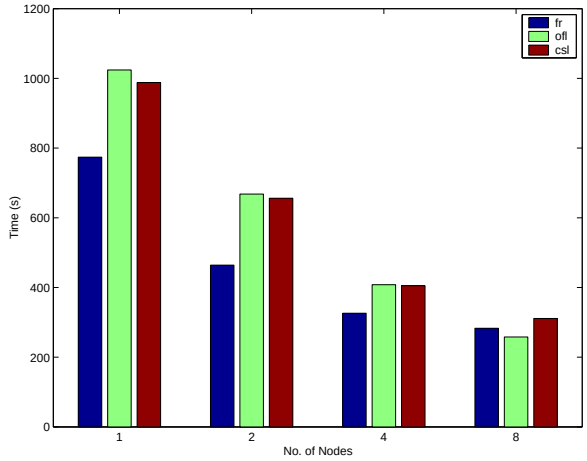

Figure 13: Performance of *pure* versions, `dataset 1`

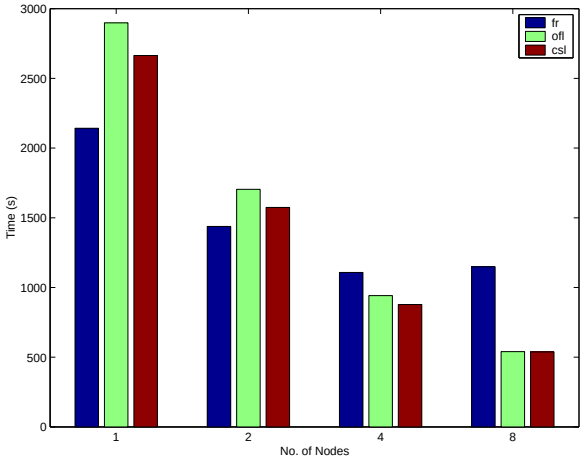

Figure 14: Performance of *pure* versions, `dataset 2`

We used two datasets for our experiments, generated using a tool described by Agrawal *et al.* [1]. Both the datasets were 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. Agrawal *et al.* use a series of classification functions of increasing complexity to classify records into different groups. Tuples in the training set are assigned the group label (classes) by first generating the tuple and then applying the classification function to the tuple to determine the group to which the tuple belongs. The two datasets we used correspond to the use of functions 1 and 7, respectively. The use of function 1 generates relatively small decision tree whereas the tree generated by function 7 is large. In our experiments, the *stop point* for the node size is 10,000, i.e., if the subtree includes fewer than 10,000 tuples, we do not expand it any further. The use of function 1 results in a tree with 10 levels, whereas the use of function 7 generates a tree with 16 levels. The datasets corresponding to the use of functions 1 and 7 are referred to as `dataset 1` and `dataset 2`, respectively.

In Section 6.2, 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.

Figures 13 and 14 show the performance of pure versions on two datasets. With 1 thread, `fr` gives the best performance for both the datasets. This is because there is no locking overhead. The relative speedups on 8 threads for `fr` are only 2.73 and 1.86 for the first and the second dataset, respectively. In fact, with the second dataset, there is a slow-down observed in going from 4 to 8 threads. This is because the the use of full replication for AVC-sets at all levels results in very high memory requirements. The second dataset produces a deeper tree, so the sum of the sizes of AVC-set for all nodes at a level can be even higher.

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, for the second dataset. However, for dataset 1, the relative speedups for both the versions are less than 4. There are two types of overhead in the use of locking schemes. The first is the additional cost of locking while incrementing every value in the AVC-set. Second, at the upper levels of the tree, the total number of elements associated with AVC-sets of all nodes at the level is quite small. This results in

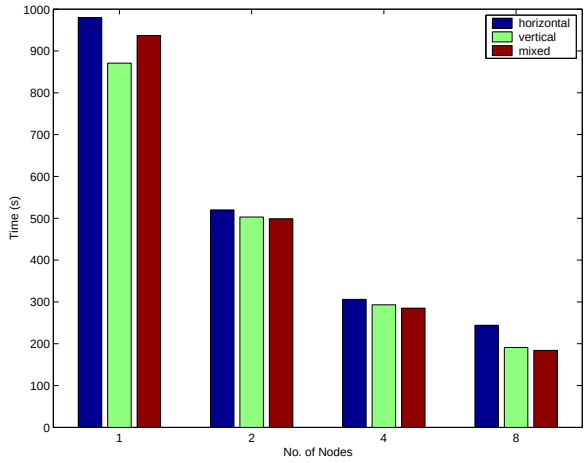

Figure 15: Combining full replication and full locking, `dataset 1`

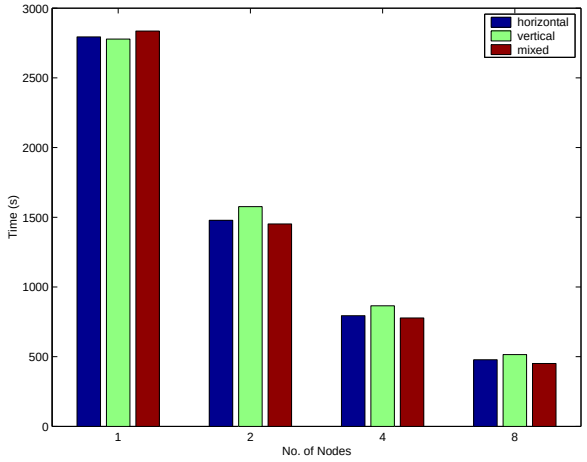

Figure 16: Combining full replication and full locking, `dataset 2`

waiting for locks and coherence cache misses when different threads want to update these elements. With dataset 1, the tree generated is not very deep, and therefore, a larger fraction of the time is spent doing the processing for the upper levels.

Figure 15 and 16 present experimental results from combining `fr` and `ofl`, from datasets 1 and 2, respectively. 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. With dataset 1, `horizontal` is consistently the slowest, and `mixed` gives the best performance on 2, 4, and 8 threads. With dataset 2, `vertical` is the slowest on 2, 4, and 8 nodes, whereas `mixed` is the best on 2, 4, and 8 nodes.

In the `horizontal` approach, the use of locking for continuous attributes at upper levels of the tree can slow down the computation because of waiting for locks and coherence cache misses. In contrast, in the `vertical` approach, the use of locking for categorical attributes at the lower levels results in waiting time for locks and coherence cache misses. As a result, the `mixed` approach results in the best performance on 2, 4, or 8 threads, for either dataset.

Figure 17 and 18 present experimental results from combining `fr` and `csl`, from datasets 1 and 2, respectively. Again, the `mixed` version is the best among the three versions, for 2, 4, and 8 threads, for either dataset.

It is interesting to compare the relative performance between combining optimized full locking with full replication and combining cache sensitive locking with full replication. As compared to optimized full locking, cache sensitive locking has lower memory requirements. However, the overhead

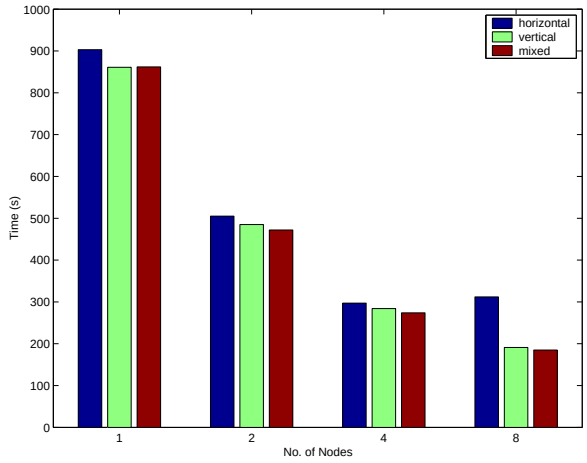

Figure 17: Combining full replication and cache-sensitive locking `dataset 1`

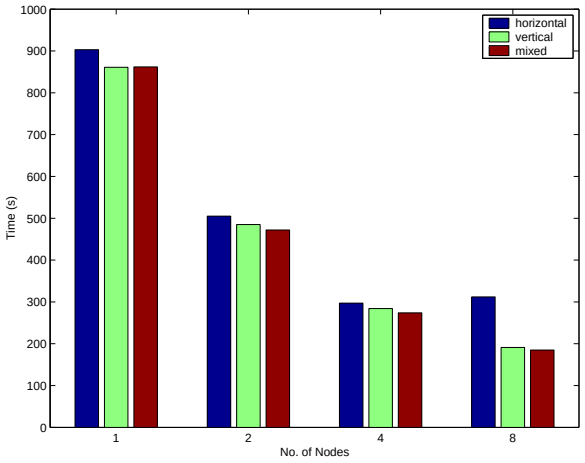

Figure 18: Combining full replication and cache-sensitive locking `dataset 2`

of waiting for locks could also be higher in cache sensitive locking.

For the first dataset, cache sensitive locking results in worse performance with `horizontal` approach and almost identical performance with `vertical` and `mixed` approaches. This is because the first dataset results in a tree with only 10 levels, and memory overhead is not a significant issue. With the second dataset, cache sensitive locking results in significantly better performance with all three approaches.

Combining cache sensitive locking and full replication using the mixed strategy results in the best performance. With the dataset 2, the relative speedup of this version on 8 threads is 5.9. Comparing this version against the best 1 thread version (which is `fr`), the speedup is 5.2.

6.4 Discussion

This case study 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.

The best relative speedup we obtained was 5.9 using 8 threads. This compares well with the

speedups that have been obtained by the researchers developing stand-alone shared memory or distributed memory decision tree implementations. Thus, our work also shows that a general framework for parallel data mining implementations can achieve high performance while significantly simplifying the programmer's task.

7 Related Work

We now compare our work with related research efforts.

Significant amount of work has been done on parallelization of individual data mining techniques. Most of the work has been on distributed memory machines, including association mining [2, 17, 18, 48], k-means clustering [10], and decision tree classifiers [4, 13, 24, 42, 44]. Recent efforts have also focused on shared memory parallelization of data mining algorithms, including association mining [47, 35, 36] and decision tree construction [46]. Our work is significantly different, because we offer an interface and runtime support to parallelize a number of data mining algorithms. Our shared memory parallelization techniques are also different, because we focus on a common framework for parallelization of a number of algorithms.

Since we have use apriori as an example in our implementation, we do a detailed comparison of our approach with the most recent work on parallelizing apriori on a shared memory machine [36]. One limitation of our approach is that we do not parallelize the candidate generation part of the algorithm in our framework. We have at least two advantages, however. First, we dynamically assign transactions to threads, whereas their parallel algorithm works on a static partition of the dataset. Second, our work on memory layout of locks and reduction elements also goes beyond their techniques. There are also many similarities in the two approaches. Both approaches segregate read-only data to reduce false sharing and consider replicating the reduction elements.

Decision tree construction for disk-resident datasets and on parallel machines has also been studied by many researchers. We describe these efforts and compare them with our approach.

One of the first decision tree construction methods for disk-resident datasets was SLIQ [29]. SLIQ requires sorting of ordered attributes and separation of the input dataset into attribute lists. In addition, it requires a data-structure called *class list*, whose size is proportional to the number of records in the dataset. Class list is accessed frequently and randomly, therefore, it must be kept in main memory. In parallelizing SLIQ, either the class list must be replicated, or a very high communication overhead is imposed.

A somewhat related approach is SPRINT [42]. SPRINT does not require class lists, but instead requires a hash table which is proportional to the number of records associated with a node in the decision tree. Like SLIQ, SPRINT requires attributes lists and sorting of ordered attributes. Moreover, SPRINT requires partitioning of attribute lists whenever a node in the decision tree is split. Thus, it can have a significant overhead of rewriting a disk-resident dataset.

The algorithm by Srivastava *et al.* [44] has many similarities with the RF-write algorithm. An important difference, however, is that it reduces the number of potential splitting points of ordered attributes by clustering their values. A somewhat similar idea is used in the CLOUDS method [4].

The only previous work on shared memory parallelization of decision tree construction on disk-resident datasets is by Zaki *et al.* [46]. They have carried out a shared memory parallelization of SPRINT algorithm. Our work is distinct in parallelizing a very different method for decision tree construction. In parallelizing SPRINT, each attribute list is assigned to a separate processor. In comparison, we parallelize updates to reduction objects, with fine sharing between the processors. Narlikar has used a fine-grained threaded library for parallelizing a decision tree algorithm [32], but the work is limited to memory-resident datasets.

Becuzzi *et al.* [5] have used a structured parallel programming environment PQE2000/SkIE for developing parallel implementation of data mining algorithms. However, they only focus on distributed memory parallelization, and I/O is handled explicitly by the programmers. The similarity among parallel versions of several data mining techniques has also been observed by Skillicorn [43]. Our work is different in offering a middleware to exploit the similarity, and ease parallel application development. Goil and Choudhary have developed PARSIMONY, which is an infrastructure for analysis of multi-dimensional datasets, including OLAP and data mining [14]. PARSIMONY does not focus on shared memory parallelization.

Some of the ideas in our parallelization techniques have been independently developed in the computer architecture field. Kagi *et al.* have used the idea of *collocation*, in which a lock and the reduction object are placed in the same cache block [26, 25]. Our focus has been on cases where a large number of small reduction elements exist and false sharing can be a significant problem. In addition, we have presented an interface for data mining algorithms, and evaluated different techniques specifically in the context of data mining algorithms.

OpenMP is the general accepted standard for shared memory programming. OpenMP currently only supports scalar reduction variables and a small number of simple reduction operations, which

makes it unsuitable for data mining algorithms we focus on. Inspector/executor approach for shared memory parallelization is based upon using an inspector that can examine certain values at runtime to determine an assignment of iterations to processors [6, 16, 27, 28, 15, 40, 45]. For data mining algorithms, the inspector will have to perform almost all the computation associated with local reductions, and will not be practical. Cilk, a language for shared memory programming developed at MIT [7], also does not target applications with reductions.

8 Conclusions

In this paper, we have focused on shared memory parallelization of data mining algorithms. By exploiting the similarity in the parallel algorithms for several data mining tasks, we have been able to develop a programming interface and a set of techniques. Our techniques offer a number of trade-offs between memory requirements, parallelism, and locking overhead. In using our programming interface, the programmers only need to make simple modifications to a sequential code and use our reduction object interface for updates to elements of a reduction object.

We have reported experimental results from implementations of apriori association mining, k-means clustering, k-nearest neighbors, and decision tree construction. These experiments establish the following: 1) Among full replication, optimized full locking, and cache-sensitive locking, there is no clear winner. Each of these three techniques can outperform others depending upon machine and dataset parameters. These three techniques perform significantly better than the other two techniques. 2) Good parallel efficiency is achieved for each of the four algorithms we experimented with, using our techniques and runtime system. 3) The overhead of the interface is within 10% in almost all cases. 4) In the case of decision tree construction, combining different techniques turned out to be crucial for achieving high performance.

Overall, our work has shown that a common collection of techniques can be developed to efficiently parallelize algorithms for a variety of mining tasks. Moreover, a high-level interface can be supported to allow the programmers to rapidly create parallel implementations.

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] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, June 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. conf. Very Large DataBases (VLDB'94)*, pages 487–499, Santiago,Chile, September 1994.

[4] K. Alsabti, S. Ranka, and V. Singh. Clouds: Classification for large or out-of-core datasets. http://www.cise.ufl.edu/ ranka/dm.html, 1998.

[5] P. Becuzzi, M. Coppola, and M. Vanneschi. Mining of association rules in very large databases: A structured parallel approach. In *Proceedings of Europar-99, Lecture Notes in Computer Science (LNCS) Volume 1685*, pages 1441 – 1450. Springer Verlag, August 1999.

[6] W. Blume, R. Doallo, R. Eigenman, J. Grout, J. Hoelflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[7] R. D. Blumofe and C. F. Joerg *et al.* Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM Conference on Principles and Practices of Parallel Programming (PPoPP)*, 1995.

[8] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[9] P. Cheeseman and J. Stutz. Bayesian classification (autoclass): Theory and practice. In *Advanced in Knowledge Discovery and Data Mining*, pages 61 – 83. AAAI Press / MIT Press, 1996.

[10] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *In Proceedings of Workshop on Large-Scale Parallel KDD Systems, in conjunction with the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 99)*, pages 47 – 56, August 1999.

[11] 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.

[12] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB*, 1996.

[13] S. Goil and A. Choudhary. Efficient parallel classification using dimensional aggregates. In *Proceedings of Workshop on Large-Scala Parallel KDD Systems, with ACM SIGKDD-99*. ACM Press, August 1999.

[14] Sanjay Goil and Alok Choudhary. PARSIMONY: An infrastructure for parallel multidimensional analysis and data mining. *Journal of Parallel and Distributed Computing*, 61(3):285–321, March 2001.

[15] E. Gutierrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *ICS00*, pages 78–87. ACM Press, May 2000.

[16] M. Hall, S. Amarsinghe, B. Murphy, S. Liao, and M. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, (12), December 1996.

[17] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. In *Proceedings of ACM SIGMOD 1997*, May 1997.

[18] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. *IEEE Transactions on Data and Knowledge Engineering*, 12(3), May / June 2000.

[19] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Inc., San Francisco, 2nd edition, 1996.

[21] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[22] Ruoming Jin and Gagan Agrawal. An efficient implementation of apriori association mining on cluster of smps. In *Proceedings of the workshop on High Performance Data Mining, held with IPDPS 2001*, April 2001.

[23] 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.

[24] 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.

[25] Alain Kagi. *Mechanisms for Efficient Shared-Memory, Lock-Based Sychronization.* PhD thesis, University of Wisconsin, Madison, 1999.

[26] Alain Kagi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 170–180. ACM Press, June 1997.

[27] Yuan Lin and David Padua. On the automatic parallelization of sparse and irregular Fortran programs. In *Proceedings of the Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers (LCR - 98)*, May 1998.

[28] Honghui Lu, Alan L. Cox, Snadhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 48–56. ACM Press, June 1997. ACM SIGPLAN Notices, Vol. 32, No. 7.

[29] M. Mehta, R. Agrawal, and J.Rissanen. Sliq: A fast scalable classifier for data mining. In *In Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[30] A. Mueller95. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, University of Maryland, College Park, August 1995.

[31] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.

[32] G. J. Narlikar. A parallel , multithreaded decision tree builder. Technical Report CMU-CS-98-184, School of Computer Science, Carnegie Mellon University, 1998.

[33] C. R. Palmer and C. Faloutsos. Density biases sampling: An improved method for data mining and clustering. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data.* ACM Press, June 2000.

[34] J. S. Park, M. Chen, and P. S. Yu. An effecitive hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[35] Srinivasan Parthasarathy, Mohammed Zaki, and Wei Li. Memory placement techniques for parallel association mining. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD)*, August 1998.

[36] Srinivasan Parthasarathy, Mohammed Zaki, Mitsunori Ogihara, and Wei Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, 2000. To appear.

[37] F. Provost and V. Kolluri. A survey of methods for scaling up inductive algorithms. *Knowledge Discovery and Data Mining*, 3, 1999.

[38] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[39] S. Ruggieri. Efficient c4.5. Technical Report TR-00-01, Department of Information, University of Pisa, February 1999.

[40] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[41] A. Savasere, E. Omiecinski, and S.Navathe. An efficient algorithm for mining association rules in large databases. In *21th VLDB Conf.*, 1995.

[42] 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.

[43] David B. Skillicorn. Strategies for parallel data mining. *IEEE Concurrency*, Oct-Dec 1999.

[44] A. Srivastava, E. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. In *In Proc. of 1998 International Conference on Parallel Processing, 1998.*, 1998.

[45] Hao Yu and Lawrence Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 66–75. ACM Press, May 2000.

[46] 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.

[47] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared memory multiprocessors. In *Proceedings of Supercomputing'96*, November 1996.

[48] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14 – 25, 1999.