

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

Ruoming Jin Gagan Agrawal †

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. Vendors of these machines are targeting data warehousing and data mining as the major markets.

In this paper, we focus on shared memory parallelization of data mining tasks. We consider three distinct aspects of this problem:

- What shared memory parallelization techniques can be used for data mining algorithms ?

*This research was supported by NSF CAREER award ACI-9733520, NSF grant ACR-9982087, and NSF grant ACR-0130437. The equipment for this research was purchased under NSF grant EIA-9703088.

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

- What kind of programming interface and runtime support can be offered to the application programmers for easing the development of shared memory data mining implementations ?
- What is the level of performance achieved using the various parallelization techniques ? What kind of overhead is introduced by the interface we have developed ?

This research is part of our overall effort in developing a middleware for rapid development of data mining implementations on large SMPs and clusters of SMPs [15, 14]. Our middleware is based on the observation that parallel versions of several well-known data mining techniques, including apriori association mining [1], k-means clustering [13], k-nearest neighbor classifier [11], artificial neural networks [11], and decision tree classifiers [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.

In this paper, we present and evaluate the middleware interface, and the underlying runtime support for 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.

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 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.

The rest of this paper is organized as follows. Our proposed parallelization techniques are presented in Section 2. The middleware interface and implementation of different techniques starting from the common specification is described in Section 3. Experimental results are presented in Section 4. We compare our work with related research efforts in Section 5 and conclude in Section 6.

2 Parallelization Techniques

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

2.1 Overview of the Problem

In the previous work, we have argued how several data mining algorithms can be parallelized in a very similar fashion [15, 14]. The common structure behind these

```

{* 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*

algorithms is summarized in Figure 1. This common structure applies to many popular data mining algorithms, including, but not limited to, apriori association mining, k-means clustering, bayesian networks, k-nearest neighbor classifier, and many decision tree construction algorithms. 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 be 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.

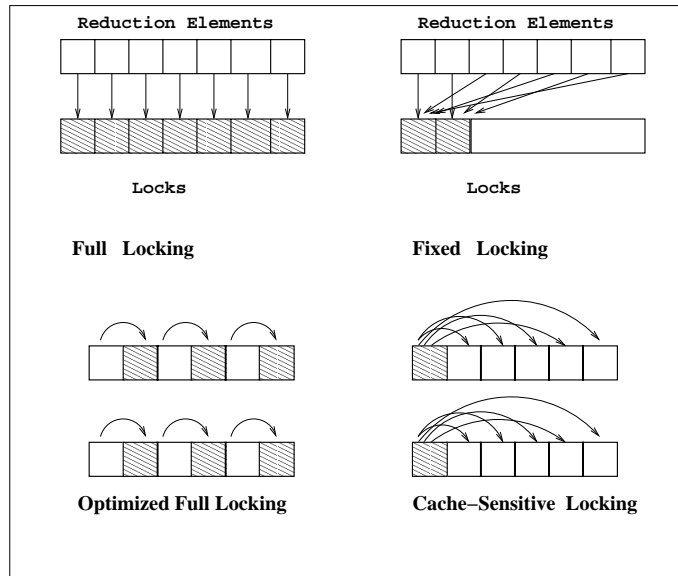


Figure 2. Memory Layout for Various Locking Schemes

2.2 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. 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* [12]. 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, as the element and the lock are in different cache blocks.

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 $i \bmod l$. 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 $i \bmod l$. 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

	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*

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.

2.3 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: 1) Memory requirements, denoting the extra memory required by a scheme because of replication or locks. 2) Parallelism, denoting if parallelism is limited because of contention for locks. 3) 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. 4) Cache misses, which only includes cold and capacity cache misses, and excludes coherence cache misses and false sharing. 5) False sharing, which occurs when two processors want to access reduction elements or locks on the same cache block, and 6) Merge costs denotes the cost of merging updates made on replicated copies.

3 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]=reducoobject->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++)
        reducoobject->Add(objectID, j, point[j]);
    reducoobject->Add(objectID, ndim, 1);
    reducoobject->Add(objectID, ndim + 1, dis);
}
```

Figure 3. Initialization and Local Reduction Functions for *k*-means

3.1 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 [15, 14]. 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 2.1, 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. 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.

3.2 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 func-

```

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

tion *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-

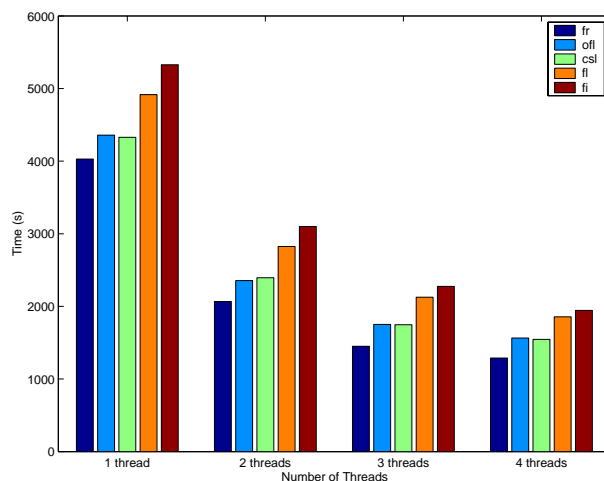


Figure 5. Comparing all Five Techniques for Apriori

sensitive locking scheme. To reduce this overhead, we use special properties of 15 (or 7) to implement a *divide15* (or *divide7*) function.

4 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.

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. The final answers obtained from the two versions were identical in all cases.

4.1 Experimental Platform

We used two different SMP machines for our experiments.

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 64 KB L1 cache and a 8 MB L2 cache. The total main memory available is 24 GB. The Sun Fireplane interconnect provides a bandwidth of 9.6 GB per second. 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.

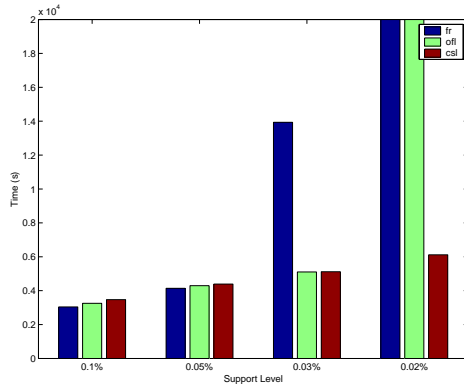


Figure 6. *Relative Performance of Full Replication, Optimized Full Locking, and Cache-Sensitive Locking: 4 Threads, Different Support Levels*

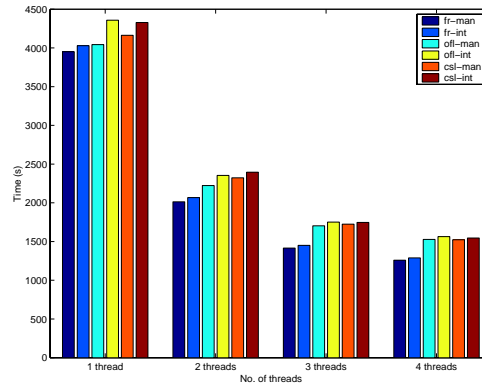


Figure 7. *Scalability and Middleware Overhead for Apriori: 4 Processor SMP Machine*

4.2 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**, **f1**, and **fi**, respectively. Execution times 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%, **f1** 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 **f1**, and 2.07 with **fi**.

From this experiment, **f1** 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.

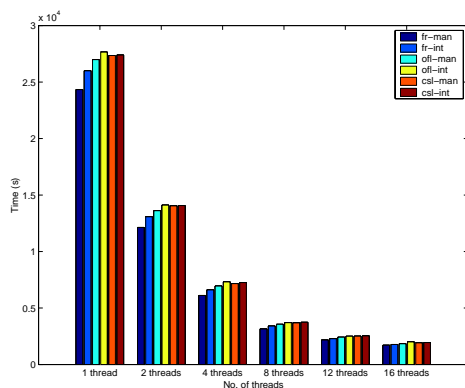


Figure 8. Scalability and Middleware Overhead for Apriori: Large SMP Machine

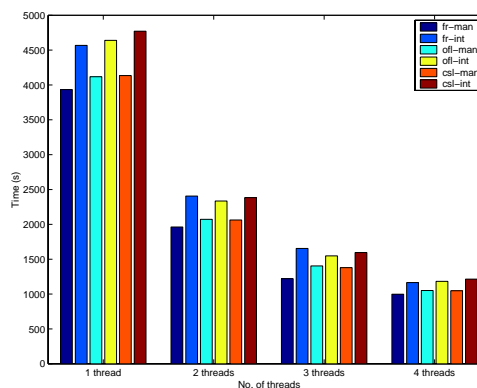


Figure 9. Scalability and Middleware Overhead for k-means: 4 Processor SMP Machine

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

4.3 Results from k-means

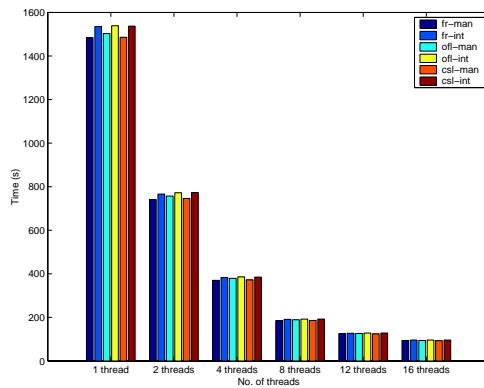


Figure 10. *Scalability and Middleware Overhead for k-means: Large SMP Machine*

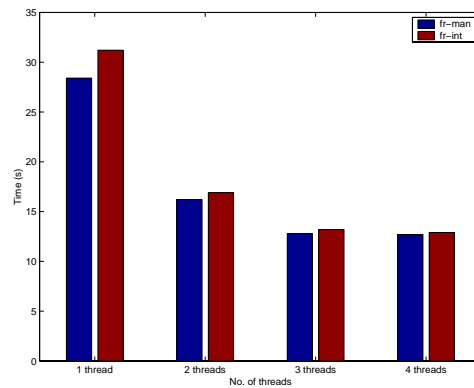


Figure 11. *Parallelization of k-nearest Neighbors*

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 evaluate 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.

4.4 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.

5 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 [1, 9, 10, 28], k-means clustering [6], and decision tree classifiers [2, 7, 16, 23, 25]. Recent efforts have also focused on shared memory parallelization of data mining algorithms, including association mining [27, 20, 21] and decision tree construction [26]. 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 [21]. 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.

Becuzzi *et al.* [3] 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 [24]. 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 [8]. 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 [18, 17]. 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 [5]. 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 [22]. 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 [4], also does not target applications with reductions.

6 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, and k-nearest neighbors. 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) Our techniques scale well on a large SMP machine. 3) The overhead of the interface is within 10% in almost all cases. Thus, our interface offers programming ease with only a very limited performance penalty.

Bibliography

- [1] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, June 1996.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.
- [6] 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.
- [7] S. Goil and A. Choudhary. Efficient parallel classification using dimensional aggregates. In *Proceedings of Workshop on Large-Scale Parallel KDD Systems, with ACM SIGKDD-99*. ACM Press, August 1999.
- [8] 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.
- [9] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. In *Proceedings of ACM SIGMOD 1997*, May 1997.
- [10] 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.
- [11] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.
- [12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Inc., San Francisco, 2nd edition, 1996.
- [13] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.
- [14] 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.
- [15] 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.
- [16] 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.

- [17] Alain Kagi. *Mechanisms for Efficient Shared-Memory, Lock-Based Synchronization*. PhD thesis, University of Wisconsin, Madison, 1999.
- [18] 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.
- [19] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2(4):345–389, 1998.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] David B. Skillicorn. Strategies for parallel data mining. *IEEE Concurrency*, Oct-Dec 1999.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14 – 25, 1999.