# Design and Evaluation of a High-level Interface for Data Mining[*]

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

## Abstract

*This paper presents a case study in developing an application class specific high-level interface for shared memory parallel programming. The application class we focus on is data mining. With the availability of large datasets in areas like bioinformatics, medical informatics, scientific data analysis, financial analysis, telecommunications, retailing, and marketing, data mining tasks have become an important application class for high performance computing.*

*Our study of a number of common data mining algorithms has shown that the same set of parallelization techniques can be applied to all of them. To exploit this, we have developed a reduction-object based interface to rapidly specify a shared memory parallel data mining algorithm. The set of parallelization techniques we target include full replication, optimized full locking, and cache-sensitive locking. We show how our runtime system can apply any of these technique starting from a common specification.*

*We have evaluated our high-level interface and the parallelization techniques using apriori association mining and k-means clustering algorithms. Our experimental results show that the overhead of the interface is within 10% and our parallelization techniques scale well.*

## 1 Introduction

Data mining is the process of extracting novel and useful patterns or models from large datasets. With the availability of large datasets in application areas like bioinformatics, medical informatics, scientific data analysis, financial analysis, telecommunications, retailing, and marketing, data mining tasks have become an important application class for high performance computing.

Recent technological advances have also made shared memory parallel machines commonly available to the end users. SMP machines with two to four processors are frequently used as desk-tops. Large shared memory machines with high bus bandwidths and very large main memories are being sold by several vendors. Vendors of these machines are targeting data warehousing and data mining as major markets.

This paper presents design and evaluation of a high-level interface for developing shared memory data mining algorithms. Our study of a number of popular data mining algorithms [13, 12] has shown that their structure can be described as comprising *generalized reduction operations* [10]. Processing for generalized reductions consists of three main steps: (1) retrieving data items of interest, (2) applying application-specific transformation operations on the retrieved input items, and, (3) mapping the input items to output items and aggregating all the input items that map to the same output data item. Most importantly, aggregation operations involve *commutative* and *associative* operations, i.e., the correctness of the output data values does not depend on the order input data items are aggregated. Further, the output of these operations are complex objects, and not scalars. Therefore, OpenMP's data parallel loops cannot be used for implementing data mining algorithms, as they currently only allow scalar reduction variables.

We have developed a reduction-object based interface to rapidly specify a shared memory parallel data mining algorithm. We describe how a programmer can perform minor modifications to a sequential code and specify a data mining algorithm using our interface. We have also developed a series of techniques for runtime parallelization of data mining algorithms, including full replication, full 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. 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 present experimental evaluation of our techniques and programming interface using our implementations of

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

apriori association mining and k-means clustering. The main results from our experiments are as follows. 1) The overhead of the interface is within 10% in almost all cases. Thus, our interface offers programming ease with only a limited performance penalty. 2) Our techniques scale well on a large SMP machine.

The rest of this paper is organized as follows. Section 2 reviews parallel versions of several common data mining techniques. The high-level interface is presented in Section 3. Parallelization techniques and their implementation from the common interface are presented in Section 4. Experimental results are presented in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

## 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 three important data mining techniques: apriori associating mining [1], k-means clustering [11], and k-nearest neighbors [8].

### 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 [2, 23]. 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 [2]. 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.

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

The same basic parallelization strategy can be used for parallelization of a number of other association mining algorithms and variations of apriori, including SEAR [15], DHP [16], Partition [19], and DIC [5]. 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 [11], 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 [8]. 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 condition is the race conditions if multiple processors try to update the list of nearest neighbors at the same time.

## 3 Programming Interface

In this section, we explain the interface we offer to the programmers for specifying a parallel data mining algorithm.

```
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 1. Initialization and Local Reduction Functions for k-means**

In using our interface, 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 1.

As we discussed earlier, 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 1 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.

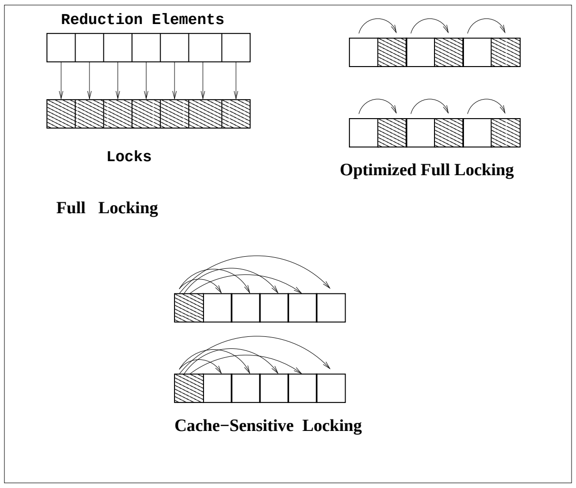

**Figure 2. Memory Layout for Various Locking Schemes**

## 4 Parallelization Techniques and Implementation

In this section, we describe the parallelization techniques we support and how they are implemented starting from the common interface.

### 4.1 Techniques

We focus on three approaches for parallelizing random write reductions. These techniques are, *full replication*, *optimized full locking*, and *cache-sensitive locking*. For motivating the optimized full locking and cache-sensitive locking schemes, we also describe a simple scheme that we refer to as *full locking*.

**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*.

We next describe the locking schemes. The memory layout of the three locking schemes, *full locking*, *optimized full locking*, and *cache-sensitive locking*, 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.

In our experiment with apriori, with 2000 distinct items and support level of 0.1%, up to 3 million candidates were generated [14]. In full locking, this means supporting 3 million locks. Supporting such a large numbers of locks results in overheads of three types. The first 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* [9]. 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.

**Optimized Full Locking:** Optimized full locking scheme overcomes the the large number of cache misses associated with full locking scheme by allocating a reduction element and the corresponding lock in consecutive memory locations, 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.

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

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

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

```
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 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 3. Implementation of Different Parallelization Techniques Starting from a Common Specification**

## 4.2 Implementation From the Common Interface

Outline of the implementation of these five techniques is shown in Figure 3. 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 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*) function.

## 5 Experimental Results

We conducted detailed experiments to evaluate 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.

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

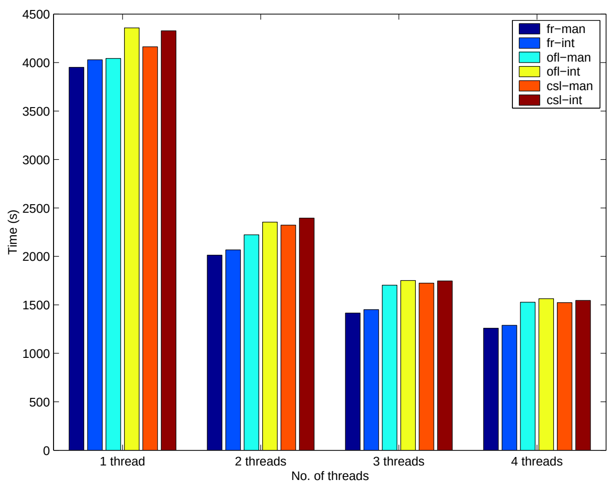

**Figure 4. Scalability and Middleware Overhead for Apriori: 4 Processor SMP Machine**

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.

The first data mining algorithm we focused on is apriori association mining. We did experiments to evaluate the scalability and middleware overhead on 4 processor and large SMP. 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 4. 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 5. 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

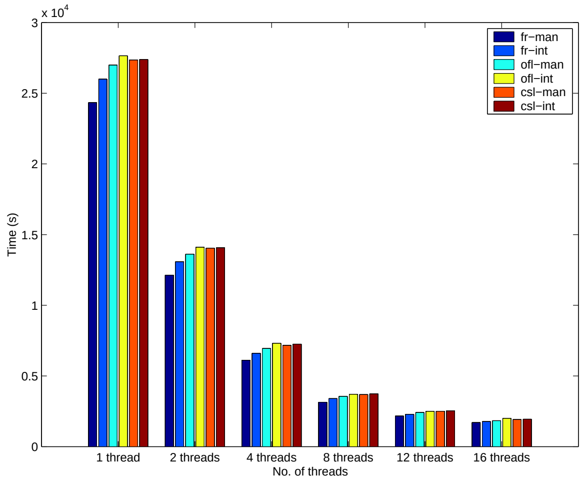

**Figure 5. Scalability and Middleware Overhead for Apriori: Large SMP Machine**

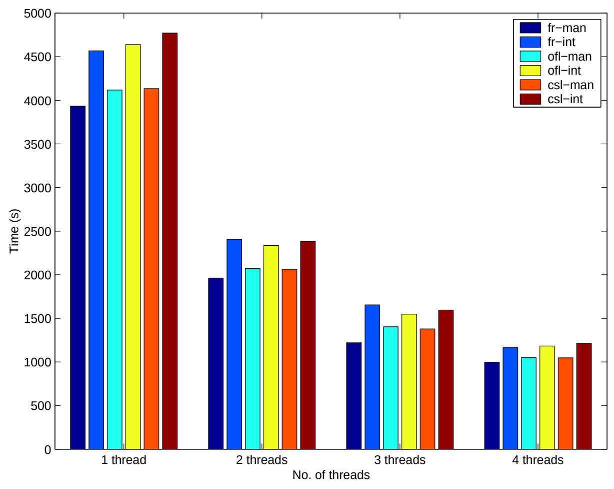

**Figure 6. Scalability and Middleware Overhead for k-means: 4 Processor SMP Machine**

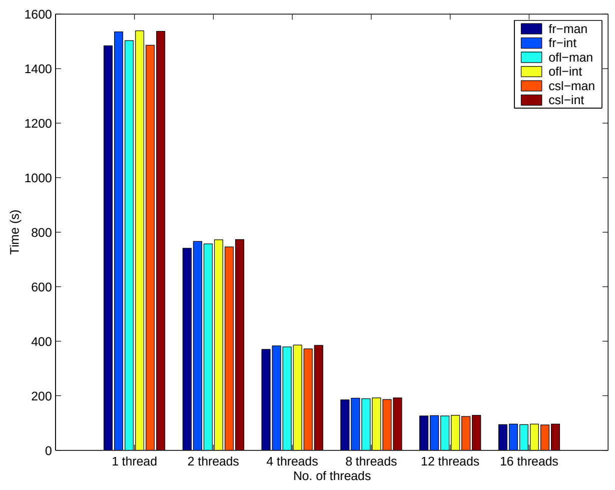

**Figure 7. Scalability and Middleware Overhead for k-means: Large SMP Machine**

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%.

The second data mining algorithm we used is k-means clustering. 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 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 6. 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 7. 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.

## 6 Related Work

We now compare our work with related research efforts.

Significant amount of work has been done on shared memory parallelization of data mining algorithms, including association mining [22, 17, 18] and decision tree construction [21]. 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.

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 [20]. 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 [7]. PARSIMONY does not focus on shared memory parallelization.

OpenMP is the general accepted standard for shared memory programming [6]. 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. Cilk, a language for shared memory programming developed at MIT [4], also does not target applications with reductions.

## 7 Conclusions

In this paper, we have presented a case study in developing an application class specific high-level interface for shared memory parallelization. We have exploited the commonality in a large number of popular data mining algorithms to develop a high-level interface for expressing parallel data mining algorithms. In using our interface, the programmers only need to make relatively small modifications to the sequential code. Our interface is supported through runtime techniques, which have a very small overhead.

Overall, we believe that our work demonstrates that application class specific high-level interfaces can be very effective. They alleviate the need for low-level programming and do not require extensive compiler or runtime support.

## References

[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] 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.

[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] 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.

[6] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1), 1998.

[7] 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.

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

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

[10] High Performance Fortran Forum. Hpf language specification, version 2.0. Available from http://www.crpc.rice.edu/HPFF/versions/hpf2/files/hpf-v20.ps.gz, January 1997.

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

[12] 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.

[13] 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.

[14] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. Submitted for Publication, 2001.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

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