

# A Middleware for Developing Parallel Data Mining Applications \*

Paper # 124

Ruoming Jin      Gagan Agrawal  
Department of Computer and Information Sciences  
University of Delaware, Newark DE 19716  
{jrm, agrawal}@cis.udel.edu

## Abstract

As the amount of information available for analysis is increasing, scalability of data mining applications is becoming a critical factor. To this end, parallel versions of most of the commonly used data mining algorithms have been developed in recent years. However, developing, maintaining, and optimizing a parallel data mining application on today's parallel systems is an extremely time-consuming task.

In this paper, we present the design and initial evaluation of a middleware system we have developed to enable rapid implementation of parallel data mining algorithms. Our middleware can help exploit parallelism on both shared memory and distributed memory configurations, while allowing efficient processing of disk resident data. Our work is based on the observation that parallel versions of a large number of common data mining algorithms including, apriori association mining, bayesian networks, k-means clustering, artificial neural networks, and k-nearest neighbors, share a similar structure. The middleware interface needs a high-level specification of a parallel data mining algorithm and does not require the application developer to specify or optimize file I/O, communication (for distributed memory parallelism), or threads (for shared memory parallelism).

We present a detailed performance study from implementations of apriori association mining and k-nearest neighbor algorithms using our middleware. Our results on a cluster of SMP workstations show that 1) distributed memory parallelization (across nodes of the cluster) achieves very high efficiency, 2) shared memory parallelization (using multiple processors on the same node) achieves good efficiency if the application is not I/O bound, and 3) the processing time required per data item remains almost the same as the dataset size is increased, establishing that efficient processing of disk resident datasets is possible.

## 1 Introduction

Data mining is an interdisciplinary field, having applications in diverse areas like bioinformatics, medical informatics, scientific data analysis, financial analysis, consumer profiling, etc. In each of these application domains, the amount of data available for analysis has exploded in recent years, making the scalability of data mining implementations a critical factor. To this end, parallel versions of most of the well-known data mining techniques have been developed in recent years. However, we believe that the following challenges still remain in effectively using parallel computing for scalable data mining:

**Ease of Development:** Developing efficient parallel applications is a difficult task on today's parallel systems. As clusters of workstations with off-the-shelf processors and networks are becoming common, careful optimization of locality and communication has become critical for performance. Clusters of SMP workstations, which have been gaining popularity in recent years, offer both distributed memory and shared memory parallelism, which makes application development even harder.

**Dealing with Large Datasets:** The datasets available in many application domains, like satellite data processing and medical informatics, easily exceed the total main memory on today's small and medium parallel systems. So, to be scalable to realistic datasets, the parallel versions need to efficiently access disk resident data.

---

\*This research was supported by NSF CAREER award ACI-9733520, NSF grant CCR-9808522, and NSF grant ACR-9982087. The equipment for this research was purchased under NSF grant EIA-9703088.

Optimizing I/O on parallel configurations is generally harder than on a uniprocessor, which further adds to the complexity of parallel data mining application development.

**Algorithm and Parallel Strategy Selection:** Selecting the most appropriate data mining techniques for a particular analysis task in a particular domain is typically done through trial and error, i.e., by applying various techniques on available datasets and viewing the results obtained. This selection process gets even more complex on parallel systems, as there often are several parallel algorithms or parallelization strategies for any given sequential technique. Therefore, it is important to be able to develop parallel versions of different data mining techniques without expending high effort, and to implement and evaluate different parallelization strategies in a rapid fashion.

**Maintaining and Performance Tuning Parallel Versions:** 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 I/O, communication, and synchronization 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.

In summary, there is a definite need for tools and techniques to be able to rapidly develop parallel and out-of-core versions of various data mining techniques [28].

In this paper, we present the design and initial performance evaluation of a middleware for enabling rapid development of parallel data mining applications. This middleware can help exploit parallelism on both shared memory and distributed memory configurations, while allowing efficient processing of disk resident data.

Our middleware is based on the observation that parallel versions of several well-known data mining techniques share a relatively similar structure. We have carefully studied parallel versions of apriori association mining [4], bayesian network for classification [14], k-means clustering [25], k-nearest neighbor classifier [24], and artificial neural networks [24]. In each of these methods, parallelization can be done by dividing the data instances (or records or transactions) among the nodes. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a *global reduction* is performed. This similarity in the structure can be exploited by the middleware system to execute the data mining tasks efficiently in parallel, starting from a relatively high-level specification of the technique.

Our middleware is particularly suited for a cluster of SMP workstations, which have emerged as a cost-effective and common parallel computing environment in recent years. Our middleware performs parallelization on different nodes of a cluster (which use message passing for communication) and on different processors on a node (which share a common memory). It enables high I/O performance by minimizing disk seek time and using asynchronous I/O operations. Thus, it can be used for rapidly developing efficient parallel data mining applications that operate on large datasets. I/O, communication, and synchronization optimizations can be implemented in such a middleware, enabling different parallel data mining applications to benefit from these. If the middleware is successfully ported on a new parallel configuration, all applications developed on top of it can be executed on the new configuration, without requiring any extra effort. Finally, such a middleware can also be used as a framework for experimenting with different parallelization strategies for data mining techniques.

We particularly focus on application domains where the datasets are stored as flat files, and not on top of a relational database. Our middleware is based upon the Active Data Repository (ADR) developed at the University of Maryland [8, 9]. ADR could only be used on a cluster of single-processor workstations, and is not tailored for data mining techniques. Our middleware has been tailored for data mining, and runs on a cluster of SMPs.

We have developed a *producer/consumer* framework for runtime parallelization of local reductions on a shared memory machine. A producer thread is responsible for managing I/O, communication, and doing work assignment. Consumer threads perform local reductions. We have implemented four techniques for avoiding race conditions among consumer threads trying to write to the same reduction object. These techniques are *full replication*, *partial replication*, *full locking*, and *fixed locking*. These techniques offer different trade-offs between memory requirements, locking overheads, and waiting time.

So far, we have used this middleware for parallelizing the apriori association mining algorithm [4] and the k-nearest neighbor algorithm [24]. In both cases, starting from a sequential version that assumed that all transactions were in main memory, we could develop an efficient parallel version using this middleware.

We conducted a series of experiments to evaluate the middleware system. Our experiments were conducted on a cluster of Sun Microsystem Ultra Enterprise 450's, each of which has 4 250MHz Ultra-II processors. We particularly experimented with large datasets, whose size exceeded the main memory of all nodes in the cluster. The main observations from our experiments include:

- Distributed memory parallelization (across nodes of a cluster) achieves very high efficiency. With the use of 8 nodes and 1 thread per node, parallel efficiency was always 95% or better.
- Shared memory parallelization (using multiple processors on the same node) gave good efficiency if the application was not I/O bound. Using 8 nodes and 3 consumer threads per node, speedups on apriori association mining were 19.32 and 21.98 for 2 GB and 8 GB datasets, respectively. The use of a fourth thread on each node gave only marginally better performance, because of the contention with the producer thread.
- The processing time required per data item remained almost the same as dataset size was increased from a main memory resident size to a disk resident size. This established that this middleware can be used for efficient processing of large and disk resident datasets.

The rest of this paper is organized as follows. Section 2 reviews parallel versions of apriori association mining, bayesian networks, k-means clustering, k-nearest neighbors, and artificial neural networks. The interface to the middleware is presented in Section 3. The details of the runtime support 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 parallel versions of several commonly used data mining techniques share a relatively similar structure. We focus on five important techniques: apriori associating mining [4], bayesian networks [14], k-means clustering [25], k-nearest neighbors [24], and artificial neural networks [24]. For each of these techniques, we describe the basic problem and then the parallelization strategy.

### 2.1 Apriori Association Mining

Given a set of transactions<sup>1</sup> (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  $Y$ . Association data mining is the process of analyzing a set of transactions to extract association rules.

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  $k$ . The most commonly used algorithm for association mining is the *apriori* mining algorithm [4]. 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 straight forward method for parallelizing the apriori association mining algorithm is *count distribution* [4]. Though a number of other parallelization techniques have been proposed [22, 41], the count distribution method is easy to implement and very efficient as long as the number of candidates does not become very large and/or sufficient memory is available on each node.

The outline of the count distribution parallelization strategy is as follows. The transactions are partitioned among the nodes. Each nodes generates the complete  $C_k$  using the frequent itemset  $L_{k-1}$  created at the end of the iteration  $k-1$ . Next, each node scans the transactions it owns to compute the count of local occurrences for each candidate k-itemset in the set  $C_k$ . After this *local* phase, all nodes perform a *global reduction* to compute the global count of occurrences for each candidate in the set  $C_k$ .

---

<sup>1</sup>In this paper, we use the terms *transactions*, *data items*, and *data instances* interchangeably.

## 2.2 Bayesian Network

Bayesian network is an approach to unsupervised classification [14]. Each transaction or data instance  $X_i$  is represented as an ordered vector of attribute values  $\{X_{i1}, \dots, X_{ik}\}$ . 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 [20]. The data instances are partitioned across nodes. In the `update_wts` phase, each node initially computes the *local weight* of each class, which is the sum of the probabilities of each locally owned data instance being in that class. *Global reduction* is then performed on each local weight to compute the final weight of each class.

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 nodes, parallelization is done at the inner most loop. On each node, the *local* sum of the class parameters is computed, and then a *global* reduction is performed.

## 2.3 k-means Clustering

The third data mining algorithm we describe is the k-means clustering technique [25]. 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:

- Start with  $k$  given centers for clusters,
- 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
- 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 previous two techniques [6, 17, 38]. The data instances are partitioned among the nodes. Each node processes the data instances it owns. Instead of moving the center of the cluster immediately after the data instance is assigned to the cluster, the *local sum* of movements of each center due to all points owned on that node is computed. A *global reduction* is performed on these local sums to determine the centers of clusters for the next iteration.

## 2.4 k-Nearest Neighbors

k-Nearest neighbor classifier is based on learning by analogy [24]. 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. The training samples are distributed among the nodes. Given an unknown sample, each node processes the training samples it owns to calculate the k-nearest neighbors *locally*. After this local phase, a global reduction computes the overall k-nearest neighbors from the k-nearest neighbors on each node.

## 2.5 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* [24]. 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 is as follows. The training data (transactions) are distributed among the nodes. Each node processes the transactions it owns and computes the *local* shifts in weights for each connection in the network. Then, a global reduction phase adds the shifts in weights for each connection calculated on all nodes.

### 3 Middleware Interface

In this section, we give an overview of the interface our middleware system supports. The interface exploits the similarity among parallel versions of several data mining techniques, as described in the previous section. It assumes that data instances have already been partitioned among the nodes.

The following functions need to be written by the application developer using our middleware. Most of these functions can be easily extracted from a sequential version that processes memory resident datasets.

- *Initial Processing*: Many data mining applications involve an initial processing of the data instances to remove noise or exceptional cases, or modify the format of certain fields, etc. This processing is performed independently on each data instance, and therefore, can be performed in parallel and in an arbitrary order on each processor.
- *Specifying the Subset of Data to be Processed*: In many case, only a subset of the available data needs to be analyzed for a given data mining task. For example, while creating associations rules from customer purchase record at a grocery store, we may be interested in processing records obtained in certain months, or for customers in a certain age groups, etc.
- *Local Reductions*: The data instances owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one data instance, a *reduction object* (declared by the programmer), is updated. This processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are read from the disks is determined by the runtime system. The reduction object is maintained in the main memory.
- *Global Reductions*: The reduction objects on all processors are combined using a global reduction function.
- *Iterator*: A parallel data mining application comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

### 4 Runtime Support

In this section, we describe the basic functionality of our middleware system. This system has been developed on top of the Active Data Repository (ADR) developed at University of Maryland [8, 9, 10]. ADR targeted strictly distributed memory parallel machines. We have implemented a framework for runtime parallelization on shared memory machines that allows us to use a cluster of SMP workstations. We are also working on modifying the interface of ADR to make it more suitable for parallel data mining algorithms.

We initially review the basic design and functionality of ADR. Then, we present our producer/consumer framework for runtime parallelization. The interface of our middleware was briefly described in the previous section.

#### 4.1 Active Data Repository

Active Data Repository (ADR) [8, 9, 10] is a runtime infrastructure that integrates storage, retrieval and processing of multi-dimensional datasets on a distributed memory parallel machine. ADR runtime support has been developed as a set of modular services implemented in C++. ADR allows customization for application specific processing, while leveraging the commonalities between the applications to provide support for common operations such as memory management, data retrieval, and scheduling of processing across a distributed memory parallel machine. Examples of data intensive applications implemented with ADR include Titan [11, 12, 35] for

satellite data processing, the Virtual Microscope [1, 18] for visualization and analysis of microscopy data, and coupling of multiple simulations for water contamination studies [27].

Customization in ADR is achieved through C++ class inheritance. That is, for each of the customizable services, ADR provides a set of C++ base classes with virtual functions that are expected to be implemented by derived classes. Adding an application-specific entry into a modular service requires the definition of a class derived from an ADR base class for that service and providing the appropriate implementations of the virtual functions. shared-nothing parallel architecture.

Any task is executed in ADR using two phases: *task planning* and *task execution*. The objective of task planning is to determine a schedule to efficiently process the computation, based on the amount of available resources in the parallel machine. A task plan specifies how parts of the final output are computed. The task execution service manages all the resources in the system and carries out the task plan generated by the task planning service. The primary feature of the task execution service is its ability to integrate data retrieval and processing for a wide variety of applications. This is achieved by pushing processing operations into the storage manager and allowing processing operations to access the buffer used to hold data arriving from disk. As a result, the system avoids one or more levels of copying that would be needed in a layered architecture where the storage manager and the processing belonged to different layers. To further reduce task execution time, the task execution service overlaps I/O, interprocessor communication and processing as much as possible. It does this by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switches between them as required. A dataset in ADR is partitioned into a set of chunks to achieve high bandwidth data retrieval. A chunk consists of one or more data items, and is the unit of I/O and communication.

## 4.2 Runtime Parallelization on SMPs

We now describe the framework we have implemented for efficiently using multiple processors available on each node of a SMP workstation for data mining applications.

We prefer to perform runtime parallelization within a single node for several reasons. First, it alleviates the need for writing explicitly parallel (threaded) programs. Second, dynamically assigning the tasks to different processors or threads allows good load-balancing. In some data mining applications, the amount of processing required for a data item can vary considerably, depending upon the values in that data item. Finally, the structure of data mining algorithms that this middleware is targeting (Section 2) makes runtime parallelization possible.

The processors available on each node need to perform the following tasks:

1. Manage disk operations and file I/O,
2. Manage communication with other nodes,
3. Execute the *Iterator* loop described in Section 3,
4. Perform runtime scheduling, i.e. assign local reductions on data items being processed by the node to different processors, and
5. Perform local reductions.

To perform the above tasks, we use one *producer* thread and one or more *consumer* threads. The producer thread is responsible for the tasks 1, 2, 3, and 4 in the list above. Typically, one consumer thread is scheduled on a single processor, and perform local reductions (task 5) on the data items assigned to it.

The producer/consumer framework is shown in Figure 1. As we mentioned earlier in this section, the unit for I/O in ADR is a *chunk*, which is typically one disk block or a small number of disk blocks. We use the same unit for dividing the local reductions among processors on a node. The idea is that chunk size can be chosen to be of sufficiently low granularity to allow effective load balancing at runtime and of sufficiently high granularity to keep the overhead of runtime scheduling acceptable.

In the set of data mining algorithms we are targeting, the local reductions on data items are independent operations, except for race conditions in updating the same reduction object. In the apriori association mining algorithm [4], counts for one or more candidates may be incremented after processing a data item. Similarly, in the k-means clustering algorithm [25], the shift for the center of one of the clusters is incremented after processing a data item. In the backpropagation based neural network training algorithm [24], weights associated with one or more connections are updated after processing a data item.

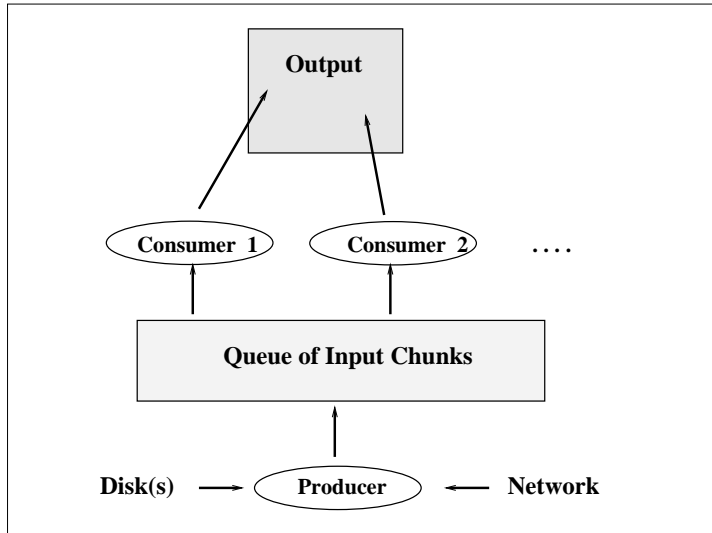


Figure 1: Producer / Consumer Framework for Runtime Parallelization

In each of these algorithms, the particular element(s) in the reduction object that need to be modified after processing a data item is not known until after performing the computation associated with the data item. For example, in the apriori association mining algorithm, the data item read from the disk needs to be matched against all candidates to determine the set of candidates whose counts will be incremented. In the k-means clustering algorithm, the cluster to which the data item belongs needs to be determined before it is known which center's shift will be updated. Therefore, it is not possible to assign data items to different threads in a manner that they do not update the same element in the reduction object.

The existing techniques for runtime parallelization cannot be used for data mining algorithms. Inspector/executor paradigm has been used for runtime parallelization and scheduling of loops [33]. This approach is based on using an inspector that can examine certain values (typically, contents of an indirection array) at runtime to determine an assignment of iterations to processors. The inspector must have a sufficiently low cost for this approach to be practical. For data mining algorithms, the inspector will have to perform almost all the computation associated with local reductions. The runtime parallelization approach taken by Rauchwerger and Padua [32] is also not applicable to our target algorithms, because their work focuses on cross-iteration dependencies.

We have implemented four different approaches for avoiding race conditions as different consumer threads may want to update the same elements in the reduction object. These techniques are, *full locking*, *fixed locking*, *full replication*, and *partial replication*.

**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, the consumer thread needs to acquire the locks associated with all elements 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 consumer 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. For example, for a 8 GB dataset we have used in our experiments, the average number of candidates during the 9 iterations was 139,000 and highest number of candidates considered during any iteration was 858,000. Supporting such a large number of locks, however, results in significant overheads. For example, with a large number of locks, operations for acquiring and releasing locks result in cache misses, slowing down the overall computation.

**Fixed Locking:** To alleviate the overheads associated with the large number of locks required in the full locking scheme, 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 consumer thread needs to update the support count for the candidate  $i$ , it needs to acquire the lock  $i \bmod l$ .

Clearly, this scheme avoids the 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.

**Full Replication:** The total size of the reduction object is typically not very large in data mining algorithms. For some of the techniques like k-means clustering and k-nearest neighbors, depending upon the value of  $k$ , it can be extremely small.

Therefore, one simple way of avoiding race conditions is to replicate the reduction object and create one copy for every consumer thread. The copy for each consumer thread needs to be initialized in the beginning. After the local reduction has been performed using all the data items on a particular node, the increments made in all the copies are *merged*. The *global reduction* function specified by the user can be used to perform this merge. After this merge, the global reduction still needs to be performed across the nodes.

**Partial Replication:** Full replication has the benefit of not requiring any locks and not requiring any thread to wait to acquire a lock. The disadvantages are memory requirements, the need for initializing the elements in the beginning, and performing the merge in the end. Consider the apriori association mining algorithm. Even if a consumer thread does not increment the count of a candidate in a particular iteration, the count of the candidate needs to be initialized to zero in the beginning, and needs to be copied during the merge phase.

To avoid this overhead, we designed the partial replication scheme. Instead of creating copies of the reduction object for each consumer thread, we create a buffer for every consumer thread. The consumer thread stores the updates to elements of the reduction object in this buffer. For example, in apriori association mining, the list of candidates whose count has been incremented can be buffered. In k-means clustering algorithm, a list of centers that have been shifted can be buffered, along with the value of the shifts. To keep the memory overhead low, there is a fixed maximum size associated with each buffer. A separate thread works on updating the reduction object using the values from the buffers. Only a single lock is associated with the entire reduction object, so values from only one buffer can be copied into the reduction object at any time.

## 5 Experimental Results

In this section, we evaluate our middleware and the set of runtime techniques we have presented here by a series of experiments. We have so far implemented two data mining algorithms using our middleware. The first is the apriori association mining algorithm [4], and the second is the k-nearest neighbors algorithm [24].

### 5.1 Experimental Platform

The experiments were conducted on a cluster of SMP workstations. We used 8 Sun Microsystem Ultra Enterprise 450's, each of which has 4 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each of the node have a 4 GB system disk and a 18 GB data disk. The data disks are Seagate-ST318275LC with 7200 rotations per minute and 6.9 milli-second seek time. The nodes are connected by a Myrinet switch with model number M2M-OCT-SW8. We believe that our cluster represents a common parallel processing configuration using off-the-shelf nodes and network.

### 5.2 Evaluating Shared Memory Parallelization Techniques

A major component of our runtime system is the producer/consumer framework for runtime parallelization of data intensive reductions. Within this framework, we have implemented four different techniques for avoiding race conditions for writes. These four techniques are *full replication*, *partial replication*, *full locking*, and *fixed locking*. Details of these techniques were presented in Section 4.

We designed a detailed experiment to both evaluate the overall efficiency of our producer/consumer framework and compare the four techniques. We used our implementation of apriori association mining in this experiment, because the k-nearest neighbor algorithm is I/O bound and has very little computation. We used a dataset with 8 million transactions, each with 20 items (on the average). The total number of distinct items in the dataset

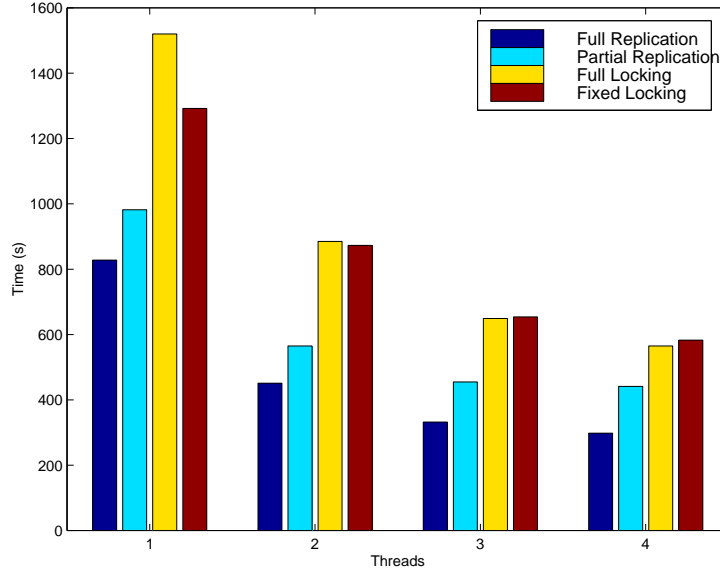


Figure 2: Comparing the Four Strategies for SMP Parallelization: 2 nodes, 800 MB dataset

is 1000. The total size of the dataset is 800 MB. Because our focus in this experiment is on evaluating SMP parallelization techniques, we choose a dataset that could fit in the main memory of a single node.

The association mining implementation developed on top of our middleware was executed using each of the four techniques, on 1, 2, 4, and 8 nodes of the cluster and using 1, 2, 3 or 4 consumer threads per node. The performance comparison on 2 nodes is shown in Figure 2 and the performance comparison on 8 nodes is shown in Figure 3. The sequential version took 1639 seconds. The threads per node listed in the Figures are consumer threads performing actual computations. In each case, there is a separate producer thread per node.

On 2 nodes and 1 thread per node, the speedups are 1.97 with full replication, 1.67 with partial replication, 1.07 with full locking, and 1.27 with fixed locking. The performance of full replication shows that distributed memory parallelization is working very well and resulting in almost perfect speedups. The performance of the other three schemes shows that there are significant overheads with shared memory parallelization using these schemes, even though only 1 thread is used on every node. The overhead of full locking is very high because of the large number of locks that need to be used. The performance of fixed locking (using 1024 locks) is significantly better, though not comparable with the performance of full replication or partial replication. With only 1 thread, the fixed locking scheme reduces the overhead associated with supporting a large number of locks in the system, but does not lose any parallelism.

On 2 nodes and 2 threads per node, the speedups are 3.63 with full replication, 2.9 with partial replication, 1.85 with full locking, and 1.87 with fixed locking. All the versions have significant speedups compared to the 1 thread versions, using the same scheme, on 2 nodes. The relative speedups compared to 1 thread versions are 1.83 with full replication, 1.73 with partial replication, 1.72 with full locking, and 1.47 with fixed locking. The performance of full replication version with 2 threads shows that the dynamic work assignment as part of the producer/consumer framework is working without any significant overheads. The partial replication version incurs some overheads for updating the main buffer and loses some performance. Its performance is still significantly better than the two locking versions. One main observation from the results with 2 threads per node is that the performance of fixed locking and full locking is very similar. Though fixed locking has lower overhead for supporting the locks, it also results in some loss of parallelism.

On 2 nodes and 3 threads per node, the speedups are 4.94 with full replication, 3.6 with partial replication, 2.52 with full locking, and 2.5 with fixed locking. The trends are very similar to the 2 threads per node case. All the schemes are successfully exploiting the extra thread to improve performance. Fixed locking actually performs worse than full locking in this case, though the difference is less than 1%.

On 2 nodes and 4 threads per node, the speedups are 5.5 with full replication, 3.71 with partial replication, 2.9 with full locking, and 2.81 with fixed locking. The relative performance improve because of adding an extra thread per node is relatively small for each of the versions. This is because each node has 1 producer thread, in

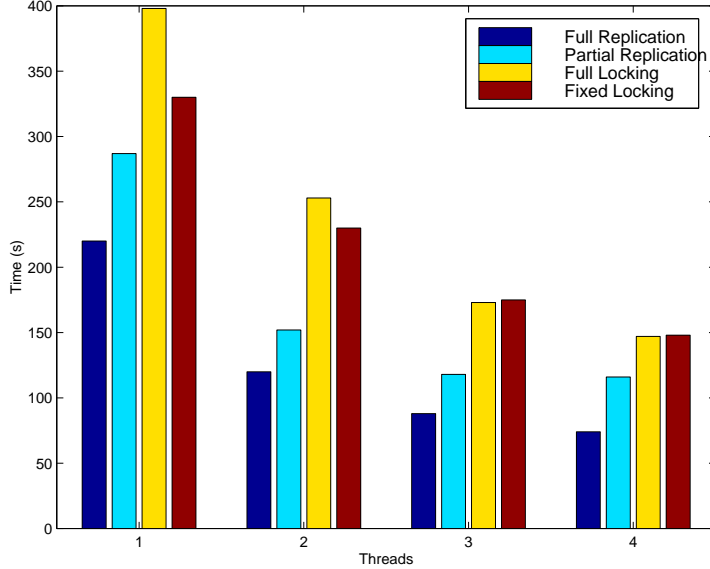


Figure 3: Comparing the Four Strategies for SMP Parallelization: 8 nodes, 800 MB dataset

addition to the 1, 2, 3 or 4 consumer threads. When 4 consumer threads are used, the total number of threads on 4 processors is 5, resulting in some contention. The fourth consumer thread still results in some additional speedups, which shows that the producer thread is not active all the time.

The results from comparing the 4 schemes on 8 nodes, with 1, 2, 3, and 4 threads per node, are shown in Figure 3. With 1 thread per node, the speedups are 7.45 with full replication, 5.7 with partial replication, 4.12 with full locking, and 4.96 with fixed locking. The relative performance of these four schemes on 8 nodes is very similar to that on 2 nodes. Again, with 1 thread per node, fixed locking performs significantly better than full locking. Neither of the locking schemes compares well against full or partial replication.

With 2 threads per node, the speedups are 13.66 with full replication, 10.72 with partial replication, 6.48 with full locking, and 7.12 with fixed locking. The performance of fixed locking is significantly better with 2 threads also. Again, substantial additional speedups occur when 3 threads are used per node. The speedups are 18.62 with full replication, 13.89 with partial replication, 9.47 with full locking, and 9.36 with fixed locking. With full replication, 24 processors (3 threads per node, 8 nodes) give a speedup of 18.62, or 77.6% parallel efficiency. Fixed locking results in marginally worse performance than full locking, which shows that some parallelism is being lost with fixed locking.

Finally, with 4 threads per node, the speedups are 22.1 with full replication, 14.1 with partial replication, 11.15 with full locking, and 11.07 with fixed locking. The parallel efficiency with the use of 32 processors is 69.1%, some-what lower than the efficiency with 24 processors. This is because of the contention with producer thread. Another noticeable factor from the results on 4 threads per node is that partial replication gave almost no performance improvement than the 3 threads per node case. This is because of the high overhead of updating the main buffer with the values cached in buffers associated with 4 threads.

### 5.3 Evaluating I/O Scalability

One of the obvious questions with the use of this middleware system is, “How does the performance scale with the increase in the size of the dataset?”. Particularly interesting is the time required per data item or transaction when the dataset is memory resident and when the dataset becomes disk resident.

We designed an experiment to evaluate this for our implementation of association mining. The main challenge in evaluating the effect of disk resident data on the performance is in keeping the amount of computation per transaction unchanged as the total number of transactions is increased. The amount of computation per transaction depends upon the number of candidates. To keep the number of candidates unchanged as the number of transactions is increased, we created 4 different datasets as follows. We initially created a 400 MB dataset, comprising 6 million transactions, with an average of 15 items per transaction. We then duplicated the data to create a 800 MB dataset, repeated it three times to create a 1.2 GB dataset, and finally, repeated it four times

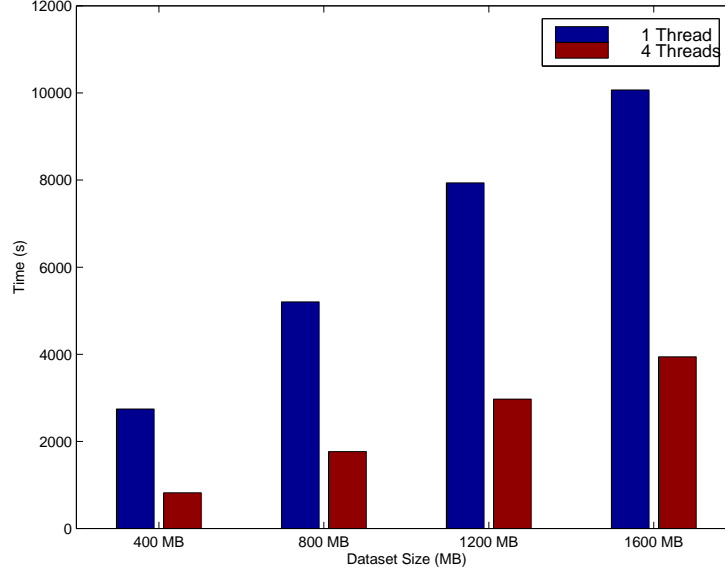


Figure 4: I/O Scalability of Association Mining Algorithm: 0.5% support level

to create a 1.6 GB dataset. If the same support percentage is used, the number of candidates considered during any iteration will be the same for these 4 datasets. The first and the second datasets are memory resident while the third and the fourth dataset exceed the main memory limit on 1 node.

Since we were only interested in evaluating the I/O performance in this experiment, we use a single node. We used 1 or 4 threads on this node. We used two different support levels, 0.5% and 1%, on the same 4 datasets. The total number of candidates that have to be considered with the support level of 0.5% is very large, and 13 iterations of the outer-loop in the apriori association mining algorithm are required. With the support level of 1%, only 4 iterations of the outer-loop are required.

The performance on the four datasets, with 1 and 4 threads on 1 node and the support level of 0.5%, is shown in Figure 4. With 1 thread, the ratio of the time spent on the 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 1.9, 2.9, and 3.67, respectively. This shows that the time required per transaction is not increasing as we move from memory resident dataset to disk resident dataset. Some of the computation in the code is independent of the number of datasets processed, which explains why the processing time increases by a factor less than linear on the size of the dataset.

With 4 threads, the ratio of the time spent on the 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 2.15, 3.62, and 4.80, respectively. As more I/O is required, the producer thread takes more cycles and slows down the computation when 4 threads are used.

The performance with 1 and 4 threads on 1 node and the support level of 1.0% is shown in Figure 5. With 1 thread, the ratio of the time spent on the 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 1.96, 3.10, and 3.93, respectively. With 4 threads, the ratio of the time spent on 800 MB, 1.2 GB, and 1.6 GB dataset to the time spent on the 400 MB dataset is 2.64, 5.04, and 6.43, respectively. As the code becomes more I/O dominated with the use of 1.0% support level, the processing time increases by more than a linear factor as the dataset size increases. However, the rate of increase is still within reasonable levels, and we believe that it shows that our system gives good performance on disk resident datasets.

## 5.4 Performance of Apriori Association Mining

In this subsection, we focus on the overall performance achieved on the implementation of apriori association mining. We have used two large datasets for this purpose. The first dataset, which is referred to as the 2GB dataset in our presentation, has 32 million transactions, each with (on the average) 20 items, and with 1000 distinct items. The total size of the dataset is 2.8 GB. Thus, this dataset does not fit in the main memory when the code is executed on 1 or 2 nodes. The second is referred to as the 8GB dataset. This dataset has 64 million transactions, with an average of 30 items per transaction. The total number of distinct items is 1000. The

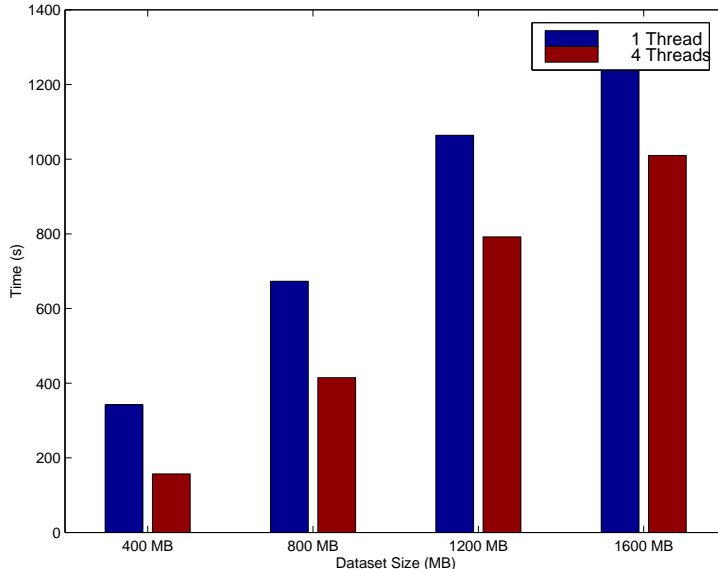


Figure 5: I/O Scalability of Association Mining Algorithm: 1.0% support level

support and confidence levels used in our experiments are 1% and 90%, respectively, for both the datasets. Since we wanted to see the best performance than can be achieved using our system, we have used the full replication scheme in all the experiments presented in this subsection.

The performance on the 2GB dataset is shown in Figure 6. Results from 1, 2, 4 and 8 nodes and 1, 2, 3, and 4 threads per node are presented. This dataset results in 8 iterations of the outer-loop in the apriori association mining algorithm. The number of candidates whose support is counted is 1,000, 278,735, 43,144, 4,354, 8,373, 949, 35, and 1, for the first through eighth iterations, respectively. Therefore, each node needs to send a total of nearly 1.3 MB of data (broken over 8 messages) to every other node, and needs to receive the same amount of data from every other node.

The middleware achieves high parallel efficiency for both distributed memory and shared memory parallelization. Using only 1 thread per node, the speedups on 2, 4, and 8 nodes are 1.93, 3.94, and 7.8, respectively. Note that the dataset owned by each node becomes main memory resident in going from 2 to 4 nodes. This results in a superlinear speedup in going from 2 to 4 nodes.

The shared memory parallelism is also exploited well on up to 3 threads per node. Using 3 threads per node, the speedups on 1, 2, 4, and 8 nodes are 2.43, 4.61, 9.31, and 19.32, respectively. Because of the producer thread, the fourth consumer thread does not result in significantly better performance. Using 4 threads per node, the speedups on 1, 2, 4, and 8 nodes are 2.53, 4.80, 9.35, and 21.04, respectively. The parallel efficiency in using 8 nodes and 3 threads per node is 81%, and the parallel efficiency in using 8 nodes and 4 threads per node is 66%.

The performance on the 8GB dataset is presented in Figure 7. This dataset resulted in 9 iterations of the outer-loop of the apriori association mining algorithm. The number of candidate whose support is counted during these iterations is 1,000, 344,912, 858,982, 25,801, 22,357, 14,354, 6,257, 55, and 1, for the first through ninth iterations, respectively. Each node has to broadcast and receive 5.1 MB of data (broken over 9 messages) during the course of the execution.

Again, high distributed memory and shared memory parallel efficiency is achieved. With the use of 1 thread per node, the speedups on 2, 4, and 8 nodes are 1.99, 4.05, and 8.07, respectively. Note that for this dataset, the data is not memory resident even on 8 nodes. However, as the data is distributed over multiple nodes, the amount of I/O needed on each node reduces, and helps achieve high speedups.

Use of up to 3 threads per node results in almost linear performance improvements. With 3 threads per node, the speedups on 1, 2, 4, and 8 nodes are 2.77, 5.50, 11.08, and 21.98, respectively. The parallel efficiency on 8 nodes with 3 thread per node is 91.6%. With 4 threads per node, the speedups on 1, 2, 4, and 8 nodes are 3.0, 6.5, 13.03, and 25.50, respectively. As the I/O requirements per node decrease, the producer thread consumes fewer cycles, resulting in more substantial performance gains with the use of the fourth consumer thread.

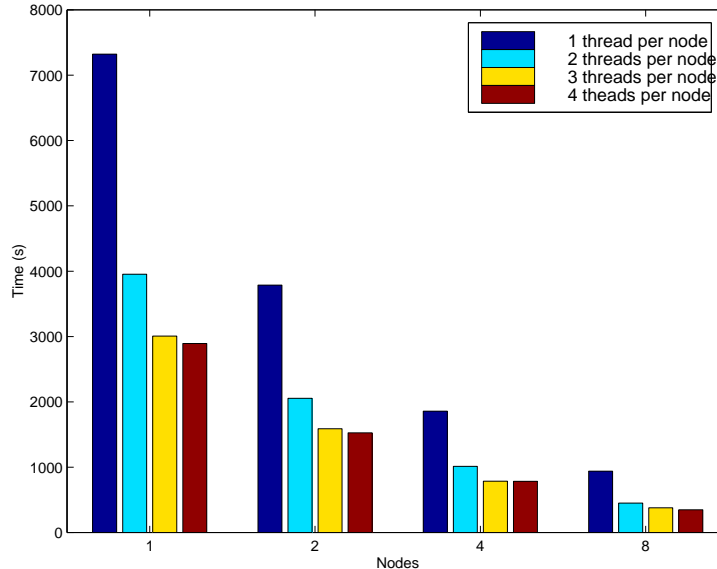


Figure 6: Performance of Apriori Association Mining: 2GB dataset

## 5.5 Performance of k-Nearest Neighbor

We now present experimental results from parallelization of k-nearest neighbors algorithm using our middleware. We used a 2.7 GB dataset with points in a 3-dimensional space for evaluating our implementation. The value of  $k$  used is 10, i.e., 10 nearest neighbors to a given point are searched by the algorithm.

The performance on 1, 2, 4, and 8 nodes, with 1 thread per node, is shown in Figure 8. This code is I/O bound, i.e., there is very little computation and most of the time is spent performing I/O. Therefore, no performance gains are possible by using additional threads for computation. The speedups on 2, 4, and 8 nodes are 1.93, 4.04, and 7.70, respectively. A superlinear speedup is observed in going from 2 to 4 nodes. This is consistent with what we observed from apriori association mining on the 2GB dataset, and is because data owned by each processor becomes memory resident in going from 2 to 4 nodes.

To verify that this algorithm is I/O bound, we measured the time taken by a version of the code that only performs I/O and no computation. The amount of time taken by that version is shown by a separate set of bars in Figure 8. The ratio of the time taken by I/O only version to the total time is 93.2%, 92%, 95.3%, and 93.5%, on 1, 2, 4, and 8 nodes, respectively. This clearly shows that the code is I/O bound and cannot benefit from additional threads for computation.

## 6 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 that can be parallelized through our approach. Most of the work has been on distributed memory machines, including association mining [4, 7, 22, 23], k-Means clustering technique [6, 17, 38], and bayesian networks [20]. Our work is significantly different, because we offer an interface and runtime support to parallelize each of these algorithms. Shared memory parallelization of association mining rules has also been an area of attention. Parthasarathy *et al.* have developed a number of modifications to the basic apriori algorithm for improving shared memory parallelization [29, 30]. The runtime parallelization techniques used in our middleware are significantly different, because we focus on techniques that can be used across a number of parallel data mining algorithms, or data intensive reduction operations in general.

One effort somewhat similar to our work is from Becuzzi *et al.* [7]. They use a structured parallel programming environment PQE2000/SkIE for developing parallel implementation of data mining algorithms. Darlington *et al.* [16] have also used structured parallel programming for developing data mining algorithms. Our work is distinct at least two important ways. First, they only target distributed memory parallelism (while they report

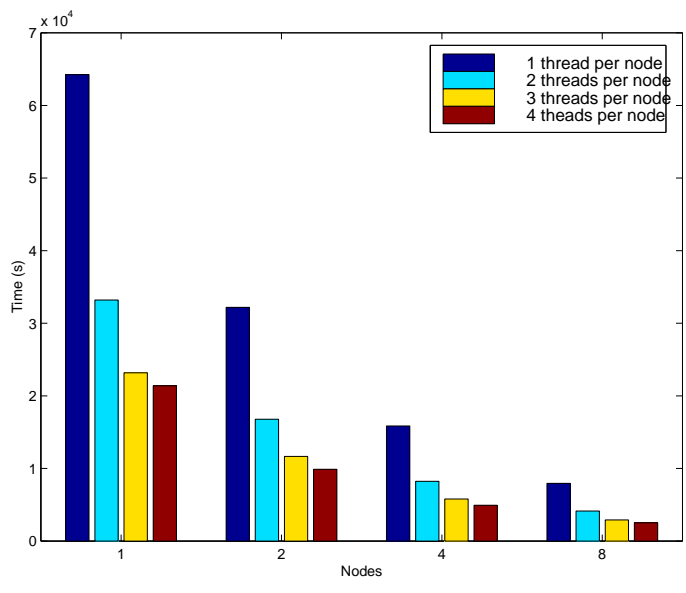


Figure 7: Performance of Apriori Association Mining: 8GB dataset

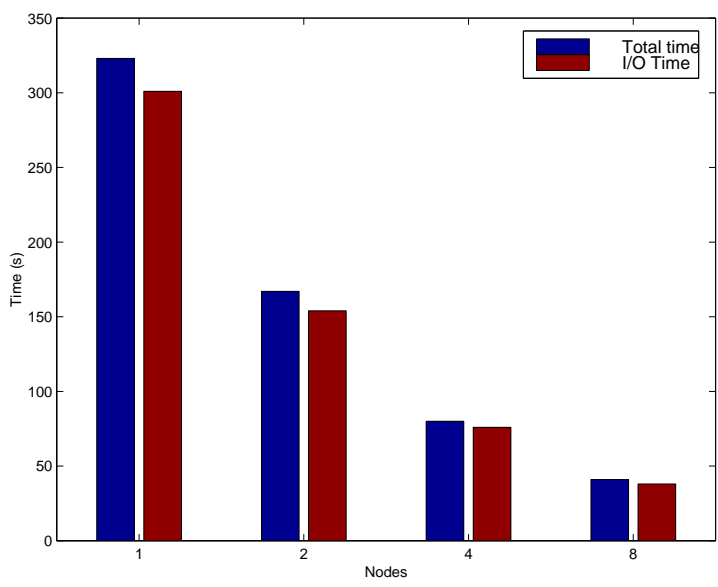


Figure 8: Performance of k-Nearest Neighbor

results on an SMP machine, it is using MPI). Second, I/O is handled explicitly by the programmers in their approach.

The similarity among parallel versions of different data mining techniques have also been observed by Skillicorn [37, 36]. Our work is different in offering a middleware to exploit the similarity, and ease parallel application development. The challenges in scalable and parallel data mining we listed in Section 1 have also been observed by a number of other authors [5, 13, 21, 26, 28, 31, 36].

Several runtime support libraries and file systems have been developed to support efficient I/O in a parallel environment [15, 34], most noticeable among these is the PASSION library designed by Alok Choudhary's group [39, 40]. They usually provide a collective I/O interface, in which all processing nodes cooperate to make a single large I/O request. With these collective I/O interfaces, the I/O requests still need to be inserted by the programmers, and data processing usually cannot begin until the entire collective I/O operation completes. The middleware we have presented is significantly different, because the computation is an integrated part of the specification. It is specifically targeted towards the kind of computations arising in data mining algorithms and data intensive reduction operations, whereas I/O libraries like PASSION are more general.

Our middleware system has been developed out the Active Data Repository (ADR) designed by Joel Saltz's group at University of Maryland [8, 9]. ADR could only be used on a cluster of single-processor workstations, and is not tailored for data mining techniques. Our middleware has been tailored for data mining, and runs on a cluster of SMPs.

## 7 Conclusions and Future Work

In this paper, we have presented a middleware system for enabling rapid development of parallel data mining implementations. The salient features of our middleware are:

- Our system exploits both shared memory and distributed memory parallelism. Thus, it is particularly well suited for clusters of SMP workstations. Such clusters have emerged as a cost-effective and common parallel processing configuration, but have not been targeted much in the current parallel data mining efforts.
- The parallel data mining implementations developed using our middleware can process disk resident datasets, thus enabling data mining on large and realistic datasets.
- Experimental results have clearly shown the effectiveness of our middleware. Specifically, we have shown that 1) distributed memory parallelization achieves very high efficiency, 2) shared memory parallelization achieves good efficiency if the application is not I/O bound, and 3) the processing time required per data item remains almost the same as the dataset size is increased, establishing that efficient processing of disk resident datasets is possible.

In the future, we plan to extend our work in many directions. In the near future, we will work on further evaluating our middleware by developing parallel implementations of k-means clustering, bayesian networks, and artificial neural networks, which have very similar structure to the apriori and k-nearest neighbors techniques. We also plan to generalize our middleware to implement data mining algorithms that use task parallelism or divide and conquer approach. Another generalization will be to allow experimentation with different parallelization strategies. Finally, we plan to use this middleware as the target for a data parallel compiler, to enable parallel data mining algorithms to be expressed in a high-level language. In a separate effort, we have developed a compiler for a data parallel dialect of Java [3, 2, 19]. The compiler generates ADR specification starting from data intensive computations expressed in such a dialect of Java. We plan to extend this compilation effort to express data mining techniques and automatically generate the code for middleware interface.

## Acknowledgments

This work builds on top of Active Data Repository (ADR) designed at the University of Maryland by the CHAOS group, lead by Joel Saltz and Alan Sussman. We particularly thank Chialin Chang and Renato Ferreira for helping us in getting started with ADR, and for their help with experiments during the course of this research. We also thank Sridhar Parthasarathy for giving us many pointers to data mining literature during the initial phases of this research.

## References

- [1] Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.
- [2] Gagan Agrawal, Renato Ferreira, Joel Saltz, and Ruoming Jin. High-level programming methodologies for data intensive computing. In *Proceedings of the Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 2000.
- [3] Gagan Agrawal, Renato Ferreira, and Joel Saltz. Language extensions and compilation techniques for data intensive computations. In *Proceedings of Workshop on Compilers for Parallel Computing*, January 2000.
- [4] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962 – 969, June 1996.
- [5] S. Anand. Designing a kernel for data mining. *IEEE Expert*, pages 65–74, March 1997.
- [6] R. Baraglia, D. Laforenza, S. Orlando, P. Palmerini, and R. Perego. Implementation issues in the design of i/o intensive data mining applications on clusters of workstations. In *Proceedings of Workshop on High Performance Data Mining IPDPS 2000, LNCS Volume 1800*, pages 350 – 357. Springer Verlag, 2000.
- [7] 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.
- [8] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998.
- [9] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, April 1999.
- [10] Chialin Chang, Tahsin Kurc, Alan Sussman, and Joel Saltz. Query planning for range queries with user-defined aggregation on multi-dimensional scientific datasets. Technical Report CS-TR-3996 and UMIACS-TR-99-15, University of Maryland, Department of Computer Science and UMIACS, February 1999.
- [11] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.
- [12] Chialin Chang, Alan Sussman, and Joel Saltz. Scheduling in a high performance remote-sensing data server. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.
- [13] Jaturon Chattratchat, John Darlington, Moustafa Ghanem, Yike Guo, Harald Huning, Martin Kohler, Janjao Sutiwaraphun, Hing Wing To, and Dan Yang. Large scale data mining: The challenges and the solutions. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, August 1997.
- [14] 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.
- [15] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.
- [16] John Darlington, Moustafa M. Ghanem, Yike Guo, and H. W. To. Performance models for co-ordinating parallel data classification. In *Proceedings of the Seventh International Parallel Computing Workshop (PCW-97)*, Canberra, Australia, September 1997.
- [17] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. 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.
- [18] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., October 1997.
- [19] Renato Ferreira, Gagan Agrawal, and Joel Saltz. Compiling object-oriented data intensive computations. In *Proceedings of the 2000 International Conference on Supercomputing*, May 2000.
- [20] D. Foti, D. Lipari, C. Pizzutti, and D. Talia. Scalable parallel clustering for data mining on multicomputers. In *Proceedings of the Workshop on High Performance Data Mining, IPDPS 2000, LNCS Volume 1800*, pages 390 – 398. Springer Verlag, May 2000.
- [21] A. Freitas and S. Lavington. *Mining very large databases with parallel processing*. Kluwer Academic Publishers, 1998.

- [22] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel datamining for association rules. In *Proceedings of ACM SIGMOD 1997*, May 1997.
- [23] 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.
- [24] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.
- [25] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.
- [26] Mahesh V. Joshi, Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar. Parallel algorithms for data mining. In J. Dongarra, I. Foster, G. Fox, K. Kennedy, and A. White, editors, *CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2000.
- [27] Tahsin M. Kurc, Alan Sussman, and Joel Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.
- [28] William A. Maniatty and Mohammed J. Zaki. A requirements analysis for parallel kdd systems. In *Proceedings of Workshop on High Performance Data Mining, IPDPS 2000, LNCS Volume 1800*, pages 358 – 365. IEEE Computer Society Press, May 2000.
- [29] 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.
- [30] 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.
- [31] J. Pei, R. Mao, K. Hu, and H. Zhu. Towards data mining benchmarking: A test bed for performance study of frequent pattern mining. In *Proceedings of 2000 ACM-SIGMOD Conference on Management of Data*. ACM Press, May 2000.
- [32] L. Rauchwerger and D.A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, February 1999.
- [33] 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.
- [34] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings Supercomputing '95*. IEEE Computer Society Press, December 1995.
- [35] Carter T. Shock, Chialin Chang, Bongki Moon, Anurag Acharya, Larry Davis, Joel Saltz, and Alan Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, January 1998.
- [36] David B. Skillicorn. Strategies for parallel data mining. *IEEE Concurrency*, Oct-Dec 1999.
- [37] D.B. Skillicorn. Strategies for parallelizing data mining. In *Proceedings of the Workshop on High-Performance Data Mining, in association with IPPS/SPDP 1998*, April 1998.
- [38] Kilian Stoffel and Abdelkader Belkoniene. Parallel k/h-means clustering for large datasets. In *Proceedings of Europar-99, Lecture Notes in Computer Science (LNCS) Volume 1685*, pages 1451 – 1454. Springer Verlag, August 1999.
- [39] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kutipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.
- [40] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.
- [41] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14 – 25, 1999.