Compiler and Middleware Support for Scalable Data Mining *

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

Abstract

High performance data mining is emerging as an important class of parallel applications. The expertise and effort currently required in implementing, maintaining, and performance tuning a parallel data mining application is currently an impediment in the wide use of parallel computers for data mining.

We have developed a data parallel dialect of Java that can be used for expressing common data mining algorithms at a high level. Our compiler generates a middleware specification from this dialect of Java. The middleware supports both distributed memory and shared memory parallelization, and performs a number of I/O optimizations to support efficient processing of disk resident datasets. Our final goal is to start from declarative mining operators, and translate them to data parallel Java.

In this paper, we describe the commonality among different data mining algorithms, the middleware and its interface, the data parallel dialect of Java, and the compilation techniques required for generating the middleware specification.

1 Introduction

Advances in information technology and a number of scientific and engineering domains have lead to *information explosion*. Very large datasets are available for analysis in a number of scientific fields, like genomics, geology and earth sciences, and computational simulations; and in commercial settings like large retail chains, financial institutions, insurance companies, etc. Knowledge discovery from these large data sets can lead to significant advances in scientific domains, as well as improvements in business practices.

As the amount of data available for analysis has exploded, scalability of data mining implementations has become a critical factor. To this end, parallel versions of most of the well-known data mining techniques have been developed in recent years [2, 20, 31, 37, 44, 45]. However, the expertise and effort currently required in implementing, maintaining, and performance tuning a parallel data mining application is a severe impediment in the wide use of parallel computers for scalable data mining.

Ideally, the end users will like to specify data mining tasks they want to perform at a very high-level (e.g., in a declarative or non-procedural fashion), and yet achieve scalable execution. Commercial products like IBM DB2 [23] and Oracle [30] are very successful in parallelizing relational queries. Like the relational database operations, commonly used data mining algorithms are well suited for parallel execution, but no tools and techniques are currently available for supporting their parallel execution.

1.1 Overview of the Project

We are developing a series of successively higher level interfaces to enable rapid specification or development of scalable data mining applications. The components of our research project are outlined in Figure 1.

Our target environment are clusters of SMPs, which are emerging as a cost-effective, flexible, and popular parallel processing configuration. Clusters of SMP workstations, where each workstation has an attached disk farm, offer both shared-nothing parallelism (across nodes of the cluster) and shared-memory parallelism (within a node). Thus, the tools and techniques we develop can also be used on purely shared memory configurations (like a large SMP machine) or purely distributed memory configurations (like a cluster of uniprocessor workstations).

Our work 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 [2], k-means clustering [24], k-nearest neighbor classifier [21], and decision tree classifier [33]. 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

*This work was supported by NSF grant ACR-9982087, NSF CAREER award ACI-9733520, and NSF grant CCR-9808522.

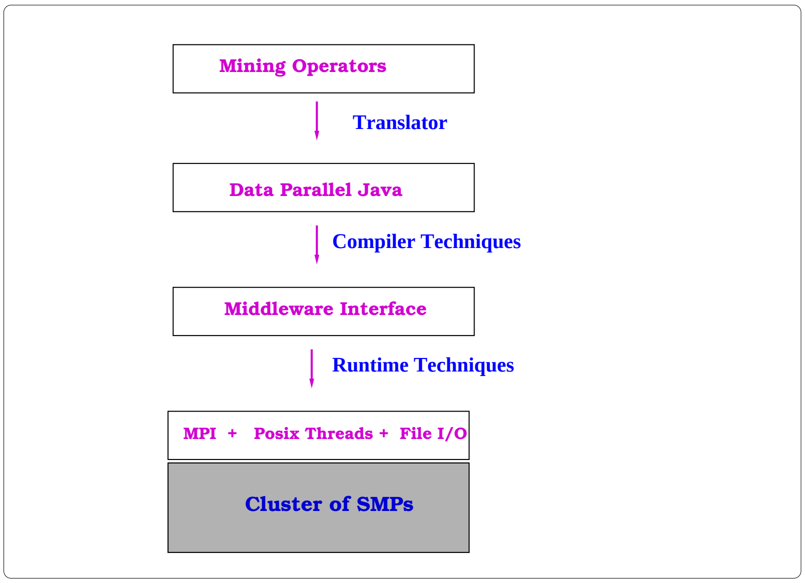

Figure 1: Overview of the Research Project

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.

Currently, developing a parallel data mining implementation on a clusters of SMPs involves programming using the Message Passing Interface (MPI), Posix threads, and file I/O, thereby requiring both high effort and expertise. We are developing the following series of successively higher-level interfaces for enabling much simpler development or specification of scalable data mining applications.

Middleware Interface: Our middleware interface exploits the similarity between a number of common data mining algorithms and allows a high-level specification of a parallel data mining algorithm. Runtime techniques achieve shared-nothing and shared-memory parallelization starting from such high-level specification. It enables high I/O performance by minimizing disk seek time and using asynchronous I/O operations, thereby achieving efficient execution on disk resident datasets. I/O, communication, and synchronization optimizations can be implemented in such a middleware, enabling different parallel data mining applications to benefit. If the middleware is successfully ported to a new parallel configuration, all applications developed on top of it can be executed on the new configuration without requiring any extra effort.

Data Parallel Java Interface: We are using a dialect of Java for expressing data mining algorithms at a higher level than our middleware interface. Our dialect of Java includes extensions for declaring multi-dimensional collections of objects, parallel loops over such collections, and a reduction interface. Starting from a sequential implementation that only processes main memory datasets, a data parallel version can be conveniently developed by inserting a few simple directives.

Mining Operators: The data mining community is working on developing a set of mining operators that will allow data mining tasks to be specified in a declarative fashion, similar to the relational queries [21]. We believe that by having a library of Java implementations of common data mining algorithms, we can easily create data parallel Java versions from such mining operators. By utilizing the compiler and runtime technology, supporting the data parallel Java dialect and the middleware interface, we can achieve scalable execution starting from mining operators.

1.2 Outline of the Paper

This paper focuses on the commonalities among different data mining algorithms, the middleware and its interface, the data parallel dialect of Java, and the compilation techniques required for generating the middleware specification.

The paper is organized as follows. In Section 2, we give an overview of parallelization approaches used for different data mining algorithms, and establish the commonality between those. Section 3 covers the middleware, including an overview, its interface, an example, and preliminary experimental results. Section 4 describes the language framework and compilation techniques. We compare our work with related research efforts in Section 5, and conclude in Section 6.

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 four important techniques: apriori associating mining [2], k-means clustering [24], k-nearest neighbors [21], and decision tree classifiers [33]. For each of these techniques, we describe the basic problem and then the parallelization strategy.

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. Given a set of transactions[1] (each of them being a set of items), an association rule is an expression $X \rightarrow Y$, where X and Y are the sets of items. Such a rule implies that transactions in databases that contain the items in X also tend to contain the items in Y.

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

A straight forward method for parallelizing the apriori association mining algorithm is *count distribution* [2]. Though a number of other parallelization techniques have been proposed [19, 45], 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.

2.2 k-means Clustering

The second data mining algorithm we describe is the k-means clustering technique, 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 count distribution method for apriori association mining [4, 16, 39]. 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.

2.3 k-nearest Neighbors

k-nearest neighbor classifier is based on learning by analogy [21]. 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.

2.4 Decision Tree Classifiers

The final set of data mining techniques we examine is decision tree classifiers [33]. In a decision tree, each leaf node is associated with a class label, and each internal node is associated with a split condition on an attribute. A decision tree is primarily built by recursively splitting the training dataset into partitions, until all or most of the records in the partitions have the same class label. Two most time consuming phases in a decision tree construction algorithm are: 1) finding the best split point for each internal node, and 2) performing the split, which means that the data items associated with the node split are divided into two partitions.

The parallelization strategy typically used for a decision tree classifier is also quite similar to the approach used for the previous five techniques. The training dataset is partitioned between the nodes. Each node processes

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

the data items it owns, to compute the *local* counts used for selecting between candidate splitting conditions. After local counts have been computed on each node, a *global reduction* needs to performed.

There are two major additional issues in implementing an efficient parallel version of a decision tree classifier. First, in distributing the data items among different nodes, numerical attributes are sorted. Second, the split operation can be I/O intensive, though it does not require communication between the nodes.

3 Middleware for Parallel Data Mining Implementations

Our middleware interface is the first high-level interface we offer for expressing parallel data mining tasks. We have designed the middleware and its interface with several goals in mind:

- Writing a middleware specification does not involve programming with the Message Passing Interface (MPI) (generally used for programming shared-nothing architectures), POSIX threads (generally used for programming shared-memory architectures), or using explicit I/O. Therefore, we believe that in the absence of higher-level interfaces, the middleware interface itself is usable by programmers who have no prior parallel programming experience.
- The middleware interface serves as a convenient target for a compiler for our data parallel Java dialect. As we will discuss in the next section, it very significantly simplifies the compilation task as compared to the general parallel compilation problem.
- By programming at the middleware level, we can evaluate and enhance our runtime parallelization techniques and I/O optimizations.
- As parallel architectures evolve, or architectural parameters change, we expect no, or at the most a very modest, change in the middleware interface. This will enable us to continue to support higher-level interfaces without having to modify the compiler for data parallel Java dialect or the translator from mining operators.

3.1 Middleware Interface

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 main memory resident datasets.

Initial Processing: Many data mining applications involve an initial processing of the data instances to modify the format of certain fields, or pre-process for some exceptional cases, 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. The result of 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.

3.2 Runtime Techniques

In this subsection, we discuss the runtime techniques implemented in the middleware for I/O performance enhancement, shared-nothing parallelization, and shared-memory parallelization.

3.2.1 Shared-nothing Parallelization

As we described in Section 2, the structure of parallel algorithms for common data mining techniques makes shared-nothing parallelization relatively simple. After data has been distributed (declustered) between different nodes, each node can execute initial processing and local reduction functions on data items it owns. After each invocation of local reduction function, local copies of reduction objects on each node are broadcasted to all other nodes, and local copies of reduction objects from all other nodes are received on each node. This communication is facilitated by the middleware. After the communication phase, global reduction function is invoked on each node.

3.2.2 I/O Performance Enhancement

Efficient I/O is critical for mining over disk resident datasets. For this purpose, a preprocessing phase, known as *task planning*, is used to create a schedule for each invocation of local reduction. Task planning is somewhat similar in nature to the *inspector* used as part of the inspector/executor paradigm for improving communication performance on distributed memory machines [35] or performing runtime parallelization [36]. A schedule or task plan specifies the order in which data items locally owned are accessed and processed. The task plan is created to minimize disk seek time on each disk, while maximizing parallelism in accessing each disk. Within each disk, the data items that are within the subset of data to be processed are accessed in the order in which they are stored. For achieving high bandwidth retrieval, the dataset owned by each node is partitioned into a set of *chunks*. A chunk is typically of the size of a small number of disk blocks and is the unit of I/O. During task processing, I/O and processing is overlapped as much as possible. It does this by maintaining explicit queues for both kind of operations (data retrieval and data processing) and switching between them as required.

One of the features of the our task processing is the integration of data retrieval and processing. 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 are not coupled together.

The I/O and shared-nothing parallelization support in our middleware has been developed on top of the Active Data Repository (ADR) developed at University of Maryland [10, 11, 12], which targets data intensive computations from domains like satellite data processing and medical informatics. Unlike our middleware, ADR targets strictly distributed memory parallel machines and is not specifically targeted towards data mining applications.

3.2.3 Runtime Shared-memory Parallelization

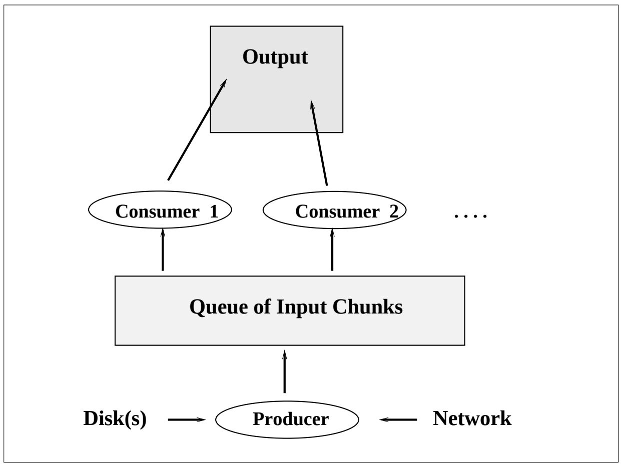

Figure 2: Producer / Consumer Framework for Runtime Parallelization

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.

The processors available on each node need to perform the following tasks: 1) perform task scheduling, 2) manage disk operations and file I/O, 3) manage communication with other nodes, 4) execute the *Iterator* loop described earlier, 5) assign local reductions on data items being processed by the node to different processors, and 6) 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, 4, and 5 in the list above. Typically, one consumer thread is scheduled on a single processor, and perform local reductions (task 6) on the data items assigned to it.

The producer/consumer framework is shown in Figure 2. As we mentioned earlier in this section, the unit for I/O in our middleware 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.

As we discussed earlier, in most of the parallel data mining algorithms, local reductions on data items are independent operations, except for race conditions in updating the same reduction object. In avoiding these race conditions, the main complication is that the particular element(s) in the reduction object that need to

```
LocalReduc(Chunk p, int k)
{
  for (each transaction t in p)
  {
    prefixtree.count(c,t,k);
  }
}

GlobalReduc(CountVector c, MessageReceiveBuffer buffer)
{
  combine(c,buffer);
}

Iterator()
{
  for (k = 1 ; not end_condition() ; k++)
  {
    for (each chunk p)
      LocalReduction(p,k);
    GlobalCombination(c,buffer);
    prefixtree.expand(k);
  }
}
Bool end_condition(k)
{
  return prefixtree.newcandidate == 0;
}
```

Figure 3: Main functions used in Middleware Specification

be modified after processing a data item is not known until after performing the computation associated with the data item. With this constraint, we have developed 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.

Fixed Locking: To alleviate the overheads associated with the large number of locks required in the full locking scheme, we have designed the fixed locking scheme. As the name suggests, a fixed number of locks are used. The number of locks chosen is a parameter to this scheme. If the number of locks is l, then the element i in the reduction object is assigned to the lock $i \ mod \ l$.

Full Replication: 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*.

Partial Replication: Instead of creating copies of the reduction object for each consumer thread (as in the full replication scheme), we create a buffer for every consumer thread. The consumer thread stores the updates to elements of the reduction object in this buffer. A separate thread works on updating the reduction object using the values from these buffers.

3.3 Case Study: Association Mining

We have so far used our middleware for implementing parallel versions of apriori association mining, k-means clustering, and k-nearest neighbors. In this subsection, we describe our implementation experience with apriori association mining, and illustrate several important aspects of the middleware interface.

We started from a sequential apriori code that processes datasets that fit in main memory. This sequential code was developed by Borgelt [9] and uses a prefix tree to represent candidate itemsets. The dataset is partitioned between different nodes in the cluster. On each node, the set of transactions is further divided into chunks, which is the unit for processing in our middleware.

The parallelization strategy we use across nodes is the same as the well known count distribution scheme [2], except that we use middleware for handling message passing.

The middleware's producer/consumer framework is used for parallelization within the node. The use of four mechanisms implemented in the middleware for avoiding race conditions while updating candidate counts leads to a family of new parallel apriori algorithms for hierarchical ((i.e. combining both shared-nothing and shared-memory parallelization) systems. Our shared-memory parallelization has some similarities with the Common Candidate Partitioned Database (CCPD) scheme developed by Zaki *et al.* [44], in the sense that there is only one itemset tree stored in the main memory. However, there are two main differences in our approach. First, the use of middleware and a producer thread allows asynchronous and efficient I/O, and therefore, good performance on disk resident datasets. Second, depending upon the scheme used for avoiding race conditions, the candidate counts may be fully or partially replicated across processors, enabling higher parallelism.

The major functions used in middleware specification are shown in Figure 3. The function `LocalReduc` just invokes the function `prefixtree.count` from the sequential implementation. After the local reduction, the reduction object on each processor (counts of all candidates) are broadcasted to all other processors. The processing in the `GlobalReduc` function is very simple. In the function shown in the figure, counts of all candidates received from another processor are combined with the local counts. This is repeated for the buffer received from each processor. The `Iterator` function invokes local and global reductions and adds candidates to the prefix tree.

We believe that our implementation is the first hierarchical parallel implementation of apriori on disk resident datasets.

3.4 Preliminary Experimental Data

In this section, we give preliminary evaluation of our middleware using implementations of association mining and k-nearest neighbor classifier.

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.

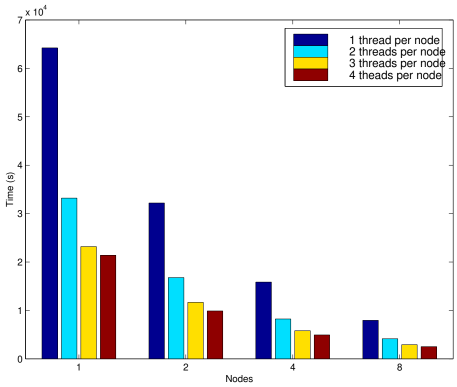

Figure 4: Performance of Apriori Association Mining

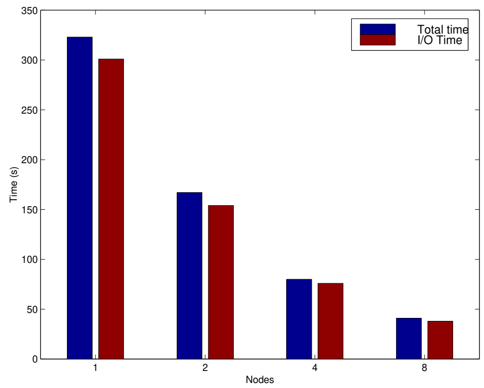

Figure 5: Performance of k-nearest Neighbor

The dataset we used for evaluating our implementation of association mining has 64 million transactions, with an average of 30 items per transaction. The total number of distinct items is 1000. The total size of the dataset is 8 GB, and exceeds the total main memory on 8 nodes. The support and confidence levels used in our experiments are 1% and 90%, respectively. Since we wanted to see the best performance than can achieved using our system, we have used the full replication scheme.

The performance on the `8GB` dataset is presented in Figure 4. 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.

Our implementation achieves high shared-nothing parallel efficiency. 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. As the data is distributed over multiple nodes, the amount of I/O needed on each node reduces, and helps achieve somewhat higher than linear speedups.

Use of up to 3 consumer threads per node results in almost linear performance improvements. With 3 consumer 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 consumer thread per node is 91.6%. With 4 consumer threads per node, the speedups on 1, 2, 4, and 8 nodes are 3.0, 6.5, 13.03, and 25.50, respectively, showing only marginal improvements from the results with 3 consumer threads. This is because each node has 1 producer thread, so the use of 4 consumer threads results in a total of 5 threads on 4 processors.

For evaluating k-nearest neighbor, we used a 2.7 GB dataset with points in a 3-dimensional space and 10 as the value of k. The performance on 1, 2, 4, and 8 nodes, with 1 thread per node, is shown in Figure 5. This code is I/O bound, i.e., there is very little computation and most of the time is spent in 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 because he data owned by each processor becomes memory resident in going from 2 to 4 nodes. The amount of time taken by a version that only performed I/O and no computation is shown by a separate set of bars in Figure 5.

```
Interface Reducinterface {
   {* Any object of any class implementing *}
   {* this interface is a reduction variable *}
}
public class Transaction {
   {* Implements a set of items *}
}
public class PrefixTree
               implements Reducinterface; {
   {* Implements a prefix tree of *}
   {* candidates with counts *}
}
```

```
static RectDomain[1] trans = [0:numTrans - 1];
static Transaction[2d] dataset = new Transaction[trans];
static PrefixTree prefixtree ;
public static void main(String[] args) {
   for (k = 1 ; !prefixtree.end_condition() ; k++) {
      foreach(t in trans) {
         prefixtree.count(dataset[t],k);
      }
      prefixtree.expand(k);
   }
}
```

Figure 6: Parallel Apriori Association Mining in Data Parallel Java

4 Data Parallel Language Support and Compiler Techniques

We now describe a data parallel dialect of Java that can be used for expressing parallel algorithms for common data mining techniques, including all the techniques we described in Section 2. This data parallel Java dialect offers a significantly higher level interface than the middleware interface described in the previous section. We also describe how innovative utilization of compiler technology will enable us to translate from this Java dialect to the middleware interface. At the end of this section, we describe how mining operators can be translated into data parallel Java code.

4.1 Java Extensions for Expressing Data Mining Algorithms

We use three main directives in our data parallel dialect. These are for specifying a multi-dimensional collections of objects, a parallel for loop, and a reduction interface. The first two have been commonly used in other object-oriented parallel systems like Titanium [43], HPC++ [7], and Concurrent Aggregates [14]. The concept of reduction interface is, to the best of our knowledge, novel to our approach.

Rectdomain: A rectdomain is a collection of objects of the same type such that each object in the collection has a *coordinate* associated with it, and this coordinate belongs to a pre-specified rectilinear section.

Foreach loop: A foreach loop iterates over objects in a rectdomain, and has the property that the order of iterations does not influence the result of the associated computations.

Reduction Interface: Any object of any class implementing the reduction interface acts as a *reduction variable* [22]. The semantics of a reduction variable are analogous to those used in version 2.0 of High Performance Fortran (HPF-2) [22]. A reduction variable has the property that it can only be updated inside a *foreach* loop by a series of operations that are associative and commutative. Furthermore, the intermediate value of the reduction variable may not be used within the loop, except for self-updates.

The goals of these extensions is to give the compiler information about independent collections of objects (with no aliasing between the elements), parallel loops, and reduction operations. We also require that no Java threads be spawned within such loop nests, and no memory locations read or written to inside the loop nests may be touched by another concurrent thread. Our compiler will also assume that no Java exceptions are raised in the loop nest.

The outline of the data parallel Java code for apriori association mining is shown in Figure 6. The outline of the middleware specification for the same data mining task was previously shown in Figure 3. We believe that programming in this data parallel dialect is much simpler than directly programming at the middleware interface. This is mainly because a programmer just needs to insert directives for parallel collections and loops in a sequential program, rather than having to generate separate functions for the middleware interface. The data parallel Java program written by a programmer assumes a single processor and a flat memory. It also assumes that the data is available as collections of objects, and is not in persistent storage.

4.2 Compilation Problem

In generating the code for the middleware interface (described in Section 3.1), the compiler needs to generate:

- An *Iterator* function that invokes local and global reduction functions.
- For each data parallel loop that updates a reduction object, we need to generate 1) a specification of the subset of the data processed in that loop, 2) a local reduction function, and 3) a global reduction function.

Extracting the iterator function is relatively simple, as each data parallel loop can be replaced by invocations to local and global reduction functions, and the resulting code is simply the iterator function.

foreach(r $\in \mathcal{R}$) {
  $O \ = \mathcal{F}(O, I[r])$
}

Figure 7: Canonical Loop for Parallel Data Mining Codes

Before describing how the processing range, the local reduction function, and the global reduction functions are extracted from each data parallel loop, we state the canonical form of loop. This form is shown in Figure 7, and is sufficient for expressing any of the parallel data mining algorithms described in Section 2. I is the collection of transactions or data items processed in the loop. O is the reduction object whose values is updated during the iterations of the loop. The range over this the loop iterates is denoted by $\mathcal{R}$.

Extracting the Processing Range: For extracting the description of the range over which the loop operates, we use the general technique of interprocedural static program slicing [42]. A number of techniques have been developed for slicing in the presence of object-oriented features like object references and polymorphism. Harrold *et al.* and Tonnela *et al.* have particularly focused on slicing in the presence of polymorphism, object references, and across procedure boundaries. Slicing in the presence of aliases and reference types has also been addressed [1]. Thus, the use of slicing for extracting range enables our compiler to work accurately in the presence of object-oriented features and frequent procedure boundaries.

Extracting the Local Reduction Function: Extracting the local reduction function is straight-forward. The computations performed in the loop are put in a separate function, which simply becomes the local reduction function.

Extracting the Global Reduction Function: Extracting the global reduction function from the computations in the loop is a more challenging problem. We describe our preliminary ideas for solving this problem here. Formally, we want to synthesize a function *op*, such that the computation in Figure 7 can be stated as $O = O \, op \, \mathcal{F}'(I[r])$.

This function *op* can be used as $O = O \ op \ O1$ to perform the global reduction, using the objects O and $O1$ computed on individual processors after the local reduction phase. This problem is significantly different from the previous work on analysis of reduction statements [6, 18, 29], because we are considering an object-oriented language where several members of an output object may be updated within the same function.

Consider any statement in the local reduction function that updates a data member of the reduction object O. If this statement includes any temporary variables that are defined in the local reduction itself, we perform forward substitution and replace these temporary variables. After such forward substitution, the update to the data member can be classified as being one of the following: 1) assignment to a *loop constant* expression, i.e., an expression whose value is constant within each invocation of the data intensive loop from which the local reduction function is extracted. 2) assignment to the value of another data member of the LHS object, or an expression involving one or more other data members, and loop constants, and 3) update using a commutative and associative function op_i, such that the data member $O.x$ is updated as $O.x = O.x \ op_i \ g(\ldots)$, where the function g does not involve any members of the reduction object O.

The set of statements in the local reduction function that update the data members of the LHS object is denoted by S. In synthesizing the function *op*, we start with the statements in the set S. The statements that fall in groups 1 or 2 above are left unchanged. The statements that fall in the group 3 are replaced by the statement of the form $O.x = O.x \ op_i \ O1.x$. Within the original local reduction function, we use the statements in the set S as the slicing criteria [42], and use program slicing to construct an executable function that will produce the same results (except as modified for the statements in the group 3) for these statements.

5 Related Work

One effort somewhat similar to our middleware effort is from Becuzzi *et al.* [5]. They use a structured parallel programming environment PQE2000/SkIE for developing parallel implementation of data mining algorithms. Darlington *et al.* [15] 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 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 that we described in Section 2 has also been observed by Skillicorn [38, 37]. Darlington *et al.* have also used cost models for choosing between different parallelization strategies, but only for a classification technique [15].

Our work at the data parallel Java level can be considered as developing an out-of-core Java compiler. Compiler optimizations for out-of-core data-structures have been considered by several projects. The PASSION project at Northwestern University has considered several different optimizations for improving locality in out-of-core applications [8, 28, 26, 25, 27, 40, 41]. Some of these optimizations have also been implemented as part of the Fortran D compilation system's support for out-of-core applications [34]. Mowry *et al.* have shown how a compiler can generate prefetching hints for improving the performance of a virtual memory system [32]. These projects have concentrated on on stencil computations written in Fortran. Our work is different in considering a different applications class, which has very different communication and data access patterns, a different language, and targeting an application-class specific middleware as the compiler output.

6 Summary

Our goal is to allow rapid development of parallel data mining implementations. For this purpose, we are developing a series of successively higher level interfaces. We have already developed a middleware that allows a high-level specification of a parallel data mining algorithm. The preliminary experimental results have demonstrated the middleware's ability to achieve efficient distributed memory and shared memory parallelization. We have proposed a data parallel dialect of Java that enables an even higher level specification of a data mining task. A compiler to translate from this data parallel dialect to our middleware interface is currently being developed.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification (TAV4)*, pages 60–73, 1991.

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

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

[4] 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.

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

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

[7] Francois Bodin, Peter Beckman, Dennis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[8] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[9] Christan Borgelt. Apriori. http://fuzzy.cs.Uni-Magdeburg.de/ borgelt/Software. Version 1.8.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A.A. Chien and W.J. Dally. Concurrent aggregates (CA). In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 187–196. ACM Press, March 1990.

[15] 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.

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

[17] 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.

[18] M. W. Hall, S. Amarsinghe, B. R. Murphy, S. Liao, and Monica Lam. Detecting Course-Grain Parallelism using an Interprocedural Parallelizing Compiler. In *Proceedings Supercomputing '95*, December 1995.

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

[20] 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.

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

[22] 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.

[23] IBM. Db2 universal database goes parallel with enterprise and enterprise-extended editions. Available at http://www-4.ibm.com/software/data/db2/udb/98eeebrochure, 1999.

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

[25] M. Kandemir, A. Choudhary, and A. Choudhary. Compiler optimizations for i/o intensive computations. In *Proceedings of International Conference on Parallel Processing*, September 1999.

[26] M. Kandemir, A. Choudhary, J. Ramanujam, and R. Bordawekar. Compilation techniques for out-of-core parallel computations. *Parallel Computing*, (3-4):597–628, June 1998.

[27] M. Kandemir, A. Choudhary, J. Ramanujam, and M. A.. Kandaswamy. A unified framework for optimizing locality, parallelism, and comunication in out-of-core computations. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):648–662, 2000.

[28] M. Kandemir, J. Ramanujam, and A. Choudhary. Improving the performance of out-of-core computations. In *Proceedings of International Conference on Parallel Processing*, August 1997.

[29] Bo Lu and John Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)*, April 1998.

[30] Tushar Mahapatra and Sanjay Mishra. *Oracle Parallel Processing*. O'reilly Publishers, 2000.

[31] 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.

[32] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and plementation (OSDI '96)*, Nov 1996.

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

[34] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118. IEEE Computer Society Press, February 1995.

[35] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[36] 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.

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

[38] 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.

[39] 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. Spring Verlag, August 1999.

[40] 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.

[41] Rajeev Thakur, Rajesh Bordawekar, and Alok Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *Proceedings of the IPPS'94 Second Annual Workshop on Input/Output in Parallel Computer Systems*, pages 54–72, April 1994. Also appears in ACM Computer Architecture News, Vol. 22, No. 4, September 1994.

[42] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[43] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Libit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience*, 9(11), November 1998.

[44] 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.

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