Combining Distributed Memory and Shared Memory Parallelization for Data Mining Algorithms *

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

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

ABSTRACT

In this paper, we focus on using a cluster of SMPs for scalable data mining. We have developed distributed memory and shared memory parallelization techniques that are applicable to a number of common data mining algorithms. These techniques are incorporated in a middleware called FREERIDE (FRamework for Rapid Implementations of Datamining Engines).

We present experimental evaluation of our techniques and framework using apriori association mining, k-means clustering, and a decision tree algorithm. We achieve excellent speedups for apriori and k-means, and good distributed memory speedup for decision tree construction. Despite using a common set of techniques and a middleware with a high-level interface, the speedups we achieve compare well against the reported performance from stand-alone implementations of individual parallel data mining algorithms. Overall, our work shows that a common framework can be used for efficiently parallelizing algorithms for different data mining tasks.

1. INTRODUCTION

In recent years, cluster of SMPs have emerged as a cost-effective, flexible, and popular parallel processing configuration. Clusters of SMPs offer both distributed memory parallelism (across nodes of the cluster) and shared-memory parallelism (within a node). This imposes an additional challenge in parallelizing any class of applications on these systems.

In this paper, we focus on using cluster of SMPs for data mining tasks. Our contributions are three fold. First, we have developed a set of techniques for distributed memory as well as shared memory parallelization that apply across a number of popular data mining algorithms. Second, we have incorporated these techniques in a middleware which offers a high-level programming interface to the application developers. Our middleware is called FREERIDE (FRamework for Rapid Implementation of Datamining Engines). Third, we present a detailed evaluation of our techniques and framework using three popular data mining algorithms, apriori association mining, k-means clustering, and a decision tree construction algorithm.

Our work is based on the observation that a number of popular data mining algorithms, including apriori association mining [2], k-means clustering [12], and decision tree classifiers [15] share a relatively similar structure. Their common processing structure is essentially that of *generalized reductions*. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and updating elements of a *reduction object* using associative and commutative operators.

In a distributed memory setting, such algorithms can be parallelized by dividing the data items (or records or transactions) among the nodes and replicating the reduction object. Each node can process the data items it owns to perform a local reduction. After local reduction on all nodes, a global reduction can be performed. In a shared memory setting, parallelization can be done by assigning different data items to different threads. The main challenge in maintaining the correctness is avoiding race conditions when different threads may be trying to update the same element of the reduction object. We have developed a number of techniques for avoiding such race conditions. Our middleware incorporates techniques for both distributed memory and shared memory parallelization and offers a high-level programming interface.

We present a detailed experimental evaluation of our techniques and the framework by parallelizing three popular data mining algorithms, apriori association mining, k-means clustering, and a decision tree construction algorithm. We achieve excellent speedups for apriori and k-means, and good distributed memory speedup for decision tree construction. Despite using a common set of techniques and a middleware with a high-level interface, the speedups we achieve compare well against the reported performance from stand-alone implementations of individual parallel data mining algorithms. Our work shows that a common framework can be used for efficiently parallelizing algorithms for different data mining tasks. Moreover, we have also demonstrated that clusters of SMPs are well suited for execution of mining algorithms.

The rest of this paper is organized as follows. Section 2 reviews parallel versions of several common data mining techniques. Techniques for both shared memory and distributed memory parallelization are presented in Section 3. Experimental results are presented in Section 4. We compare our work with related research efforts in Section 5 and conclude in Section 6.

2. PARALLEL DATA MINING ALGORITHMS

In this section, we describe how several commonly used data mining techniques can be parallelized in a very similar way.

Our discussion focuses on three important techniques: apriori associating mining [2], k-means clustering [12], and a decision tree construction algorithm [7].

2.1 Apriori Association Mining

Association rule mining is the process of analyzing a set of transactions to extract *association rules* and is a very commonly used and well-studied data mining problem [3, 23]. Given a set of transactions (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

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

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 distributed memory parallelization of apriori association mining algorithm is *count distribution* [2]. 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 k-itemset in the set C_k.

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

2.2 k-means Clustering

The second data mining algorithm we describe is the k-means clustering technique [12], 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.

In a distributed memory setting, this algorithm can be parallelized as follows. 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.

A simple shared memory parallelization strategy will be as follows. The data instances are read, and each data instance (or a set of instances) are assigned to one processor. This processor performs the computations associated with the data instance, and then updates the center of the cluster this data instance is closest to. Again, the only challenge in maintaining correctness is avoiding the race conditions when centers of clusters may be updated by different processors.

2.3 RainForest Based Decision Tree Construction

The final set of data mining techniques we examine is decision tree classifiers [15]. 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. The 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.

RainForest is a general approach for scaling decision tree construction to larger datasets, while also effectively exploiting the available main memory [7]. This is done by isolating an AVC (Attribute-Value, Classlabel) set for a given attribute and a node being processed. The size of the AVC-set for a given node and attribute is proportional to the number of distinct values of the attribute and the number of distinct class labels. Given a node of the decision tree, AVC-group is the combination of AVC-set for all attributes. The key observation is that though AVC-group does not contain sufficient information to reconstruct the training dataset, it contains all information that is required for selecting the criteria for splitting the node. Since the number of attributes and the distinct values they can take is usually not very large, one can expect the AVC-group for a node to easily fit in main memory. With this observation, processing for selecting the splitting criteria for the root node can be easily performed even if the dataset is disk-resident. By reading the training dataset once, AVC-group of the root is constructed. Then, the criteria for splitting the node is selected.

Parallelization of RainForest based algorithms is quite similar to the parallelization of apriori and k-means algorithms, on distributed memory as well as shared memory settings. The key observation is that updates to AVC-sets involve associative and commutative operators only. For distributed memory parallelization, the training records could be partitioned between the nodes. Each node can update its own copy of AVC-group, and these copies can be combined together during a global combination phase. For shared memory parallelization, training records can be assigned to different threads and AVC-sets can be updated. In doing this, race conditions must be avoided when more than one thread tries to update the same field of a AVC-set.

3. PARALLELIZATION TECHNIQUES

In this section, we focus on techniques for shared memory and distributed memory parallelization of the data mining algorithms we described in the previous section. Initially, we describe techniques for shared memory parallelization. Then, we focus on combining shared memory and distributed memory parallelization.

3.1 Shared Memory Parallelization

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

The main correctness challenge in parallelizing a loop like this on a shared memory machine arises because of possible race conditions when multiple processors update the same element of the

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

Figure 1: Structure of Common Data Mining Algorithms

reduction object. The element of the reduction object that is updated in a loop iteration (i) is determined only as a result of the processing. For example, in the apriori association mining algorithm, the data item read needs to matched against all candidates to determine the set of candidates whose counts will be incremented. In the k-means clustering algorithm, first the cluster to which a data item belongs is determined. Then, the center of this cluster is updated using a reduction operation.

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

- It is not possible to statically partition the reduction object so that different processors update disjoint portions of the collection. Thus, race conditions must be avoided at runtime.
- The execution time of the function $process$ can be a significant part of the execution time of an iteration of the loop. Thus, runtime preprocessing or scheduling techniques cannot be applied.
- In many of algorithms, the size of the reduction object can be quite large. This means that the reduction object cannot be replicated or privatized without significant memory overheads.
- The updates to the reduction object are *fine-grained*. The reduction object comprises a large number of elements that take only a few bytes, and the foreach loop comprises a large number of iterations, each of which may take only a small number of cycles. Thus, if a locking scheme is used, the overhead of locking and synchronization can be significant.

We focus on three techniques we have developed for parallelizing data mining algorithms. These techniques are, *full replication*, *optimized full locking*, and *cache-sensitive locking*. For motivating the optimized full locking and cache-sensitive locking schemes, we also describe a simple scheme that we refer to as *full locking*.

Full Replication: One simple way of avoiding race conditions is to replicate the reduction object and create one copy for every thread. The copy for each thread needs to be initialized in the beginning. Each thread simply updates its own copy, thus avoiding any race conditions. After the local reduction has been performed using all the data items on a particular node, the updates made in all the copies are *merged*.

We next describe the locking schemes. The memory layout of the three locking schemes, *full locking*, *optimized full locking*, and *cache-sensitive locking*, is shown in Figure 2.

Full Locking: One obvious solution to avoiding race conditions is to associate one lock with every element in the reduction object. After processing a data item, a thread needs to acquire the lock associated with the element in the reduction object it needs to update.

In our experiment with apriori, with 2000 distinct items and support level of 0.1%, up to 3 million candidates were generated [13]. In full locking, this means supporting 3 million locks. Supporting such a large numbers of locks results in overheads of three types. The first is the high memory requirement associated with a large number of locks. The second overhead comes from cache misses. Consider an update operation. If the total number of elements is large and there is no locality in accessing these elements, then the update operation is likely to result in two cache misses, one for the element and second for the lock. This cost can slow down the update operation significantly.

The third overhead is of *false sharing* [11]. In a cache-coherent shared memory multiprocessor, false sharing happens when two processors want to access different elements from the same cache block. In full locking scheme, false sharing can result in cache misses for both reduction elements and locks.

Optimized Full Locking: Optimized full locking scheme overcomes the the large number of cache misses associated with full locking scheme by allocating a reduction element and the corresponding lock in consecutive memory locations, as shown in Figure 2. By appropriate alignment and padding, it can be ensured that the element and the lock are in the same cache block. Each update operation now results in at most one cold or capacity cache miss. The possibility of false sharing is also reduced. This is because there are fewer elements (or locks) in each cache block. This scheme does not reduce the total memory requirements.

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

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

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

3.2 Distributed Memory Parallelization

The data mining algorithms we discussed in the previous section can all be parallelized in a distributed memory setting in a very similar fashion. The data items or training records are partitioned between the nodes and the reduction object is replicated. Each node can process the data items it owns to perform a local reduction. After local reduction on all nodes, a global reduction can be performed.

Based upon the three shared memory parallelization techniques we described in the previous subsection, we implemented four different versions for distributed memory parallelization.

Optimized Full Locking (without copying): This scheme is based upon using optimized full locking for shared memory parallelization. Recall that in this approach, locks are allocated on the same cache block as the reduction elements. In distributed memory parallelization, the entire reduction object (including the locks) are communicated for global reduction. Thus, the communication vol-

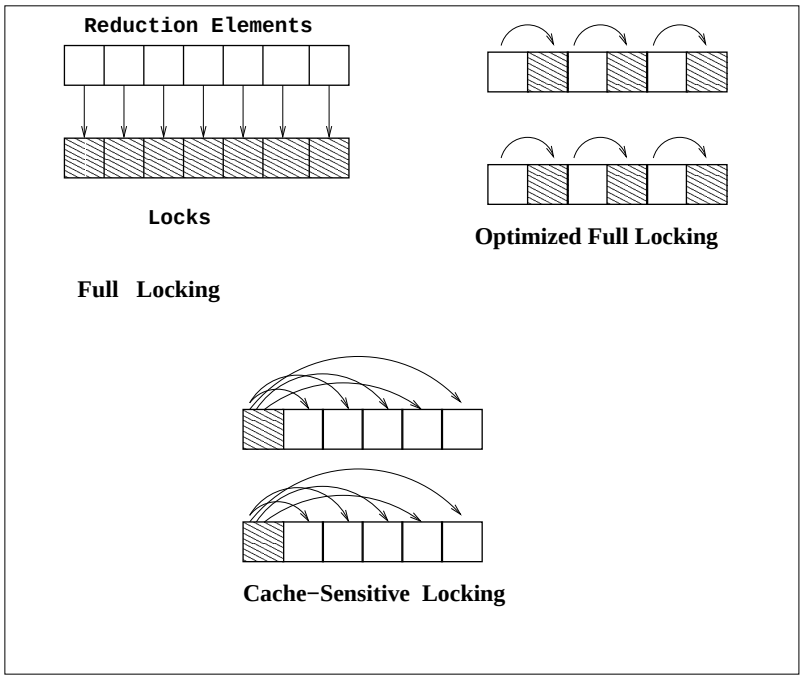

Figure 2: Memory Layout for Various Locking Schemes

ume increases two-fold. This approach is denoted as `ofl`.

Optimized Full Locking (with copying): To avoid the overhead of communicating the locks, the reduction object is initially copied on each node to extract only the reduction elements. Then, the copied reduction elements are communicated. Thus, this approach trades in extra copying for reducing communication volume. This approach is denoted as `ofl-c`.

Cache Sensitive Locking: The cache sensitive locking scheme involves only a small memory overhead of locks, as a single lock is used on each cache block. Therefore, for distributed memory parallelization, we communicate the entire reduction object, including the locks. We did not create separate versions with and without copying. This approach is denoted as `csl`.

Full Replication: After merging the replicated reduction objects within a node, the resulting reduction object is communicated. There is no extra communication or copying overhead because of the locks. This scheme is denoted as `fr`.

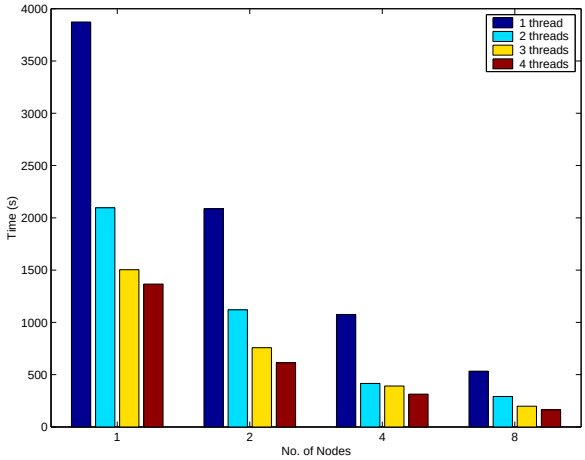

Figure 3: Performance of Apriori (`fr` version)

4. EXPERIMENTAL RESULTS

In this section, we evaluate our techniques and framework using three popular data mining algorithms that were described in Section 2. We created four versions for each of the three algorithms, corresponding to the four approaches described in Section 3.2.

We used 8 Sun Microsystem Ultra Enterprise 450's, each with 4 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each node has 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.

4.1 Results from Apriori Association Mining

We used a 1 GB dataset to evaluate our implementations of apriori association mining. The dataset was generated using a synthetic generator tool developed at IBM Almaden [2]. This tool has been widely used in evaluating previous work on association mining. In the dataset we used, the average transaction length was 10 and the number of distinctive items was 1000. A support level of 0.5% was used for our experiments.

The performance from four versions, full replication, optimized full locking without copying, optimized full locking with copying, and cache sensitive locking, are shown in Figures 3, 4, 5, 6, respectively. We show experimental results using 1, 2, 3, and 4 threads, and 1, 2, 4, and 8 nodes of the cluster. The 1 thread, 1 node version using full replication has no measurable overheads over the sequential version, and is used as calculating absolute speedups.

All versions achieve good shared memory and distributed memory speedups. In the `fr` version, the speedups using 1 thread on 2, 4, and 8 nodes are 1.85, 3.60, and 7.27, respectively. The speedups using 3 and 4 threads on 1 node are 2.58 and 2.84, respectively. The

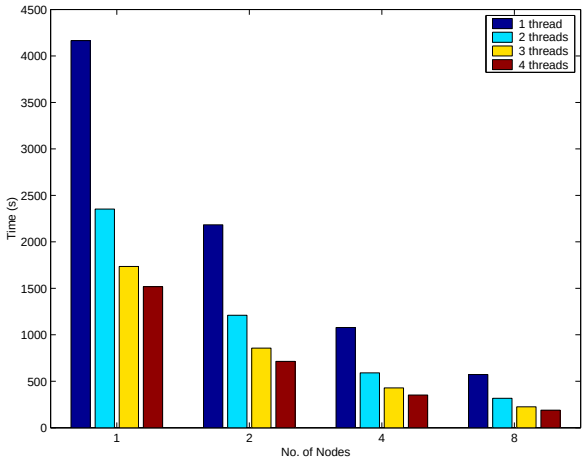

Figure 4: Performance of Apriori (`ofl` version)

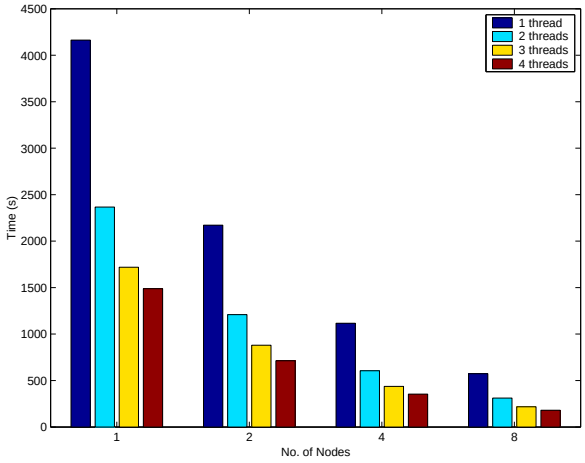

Figure 5: Performance of Apriori (`ofl-c` version)

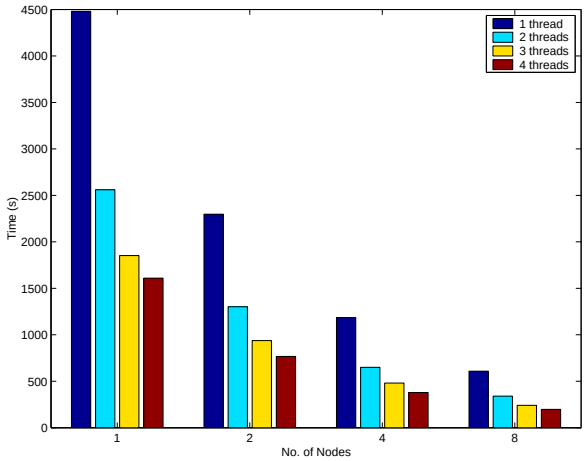

Figure 6: Performance of Apriori (`csl` version)

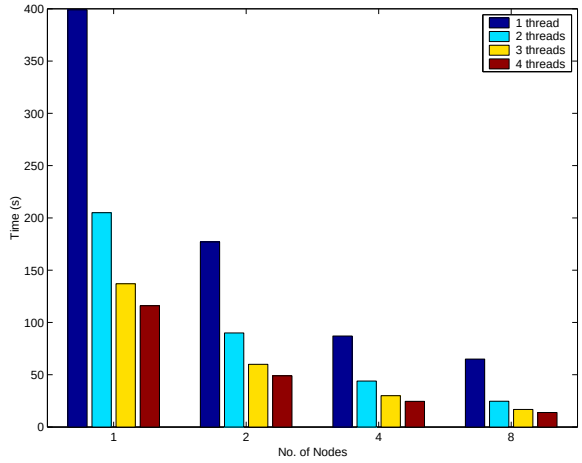

Figure 7: Performance of k-means (`fr` version)

overall speedup using 4 threads and 8 nodes is 23.76. Almost linear speedups are obtained in distributed memory settings and using up to 3 threads on shared memory settings. The use of the fourth thread on a 4 processor node does not give significant additional speedup. This is because our implementation uses an additional *producer* thread for managing I/O and coordinating between other threads.

The relative speedups obtained by the other three versions achieve very similar performance for apriori. However, they all involve an additional 5-15% overhead because of the use of locks. Because sufficient memory is available on each node to replicate the reduction object, full replication gives the best performance. For the dataset or support levels where the number of candidates becomes very large, we can expect full replication to have significant overheads because of thrashing. In comparing `ofl` and `ofl-c` versions, the performance is very similar. Because of the amount of computation involved, the additional costs of communicating the locks or extra copying are not significant.

4.2 Results from k-means Clustering

We next describe experimental results from our implementations of k-means clustering. We used a 200 MB datasets comprising three dimensional points, and 100 as the value of k.

The results from full replication, optimized full locking (without copying), optimized full locking (with copying), and cache sensitive locking are presented in Figures 7, 8, 9 and 10, respectively.

k-means clustering is a simpler code than apriori and decision tree classifier, and excellent speedups are seen from all versions, on both distributed memory and shared memory settings. Similar to apriori, the replicated reduction objects fits well in memory, and therefore, the `fr` versions gives the best performance. Using 8 nodes and 4 threads on each node, it achieves a speedup of 28.7. The relative speedups achieved by other versions are almost identical, but some initial overhead is incurred because of locking.

4.3 Results from Decision Tree Construction

Finally, we present experimental results from our implementation of RainForest based decision tree construction algorithm.

The dataset we used for our experiments was generated using a tool described by Agrawal *et al.* [1]. The datasets was nearly 600 MB, with 16 million records in the training set. Each record has 9 attributes, of which 3 are categorical and other 6 are numerical. Every record belongs to 1 of 2 classes.

The results from full replication, optimized full locking (without copying), optimized full locking (with copying), and cache sensitive locking are presented in Figures 11, 12, 13 and 14, respec-

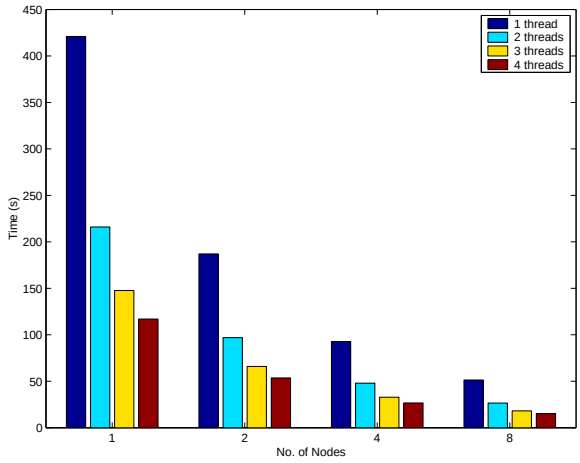

Figure 8: Performance of k-means (`ofl` version)

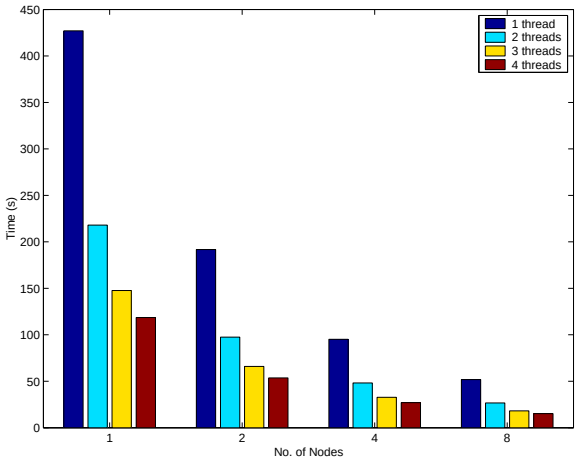

Figure 10: Performance of k-means (`csl` version)

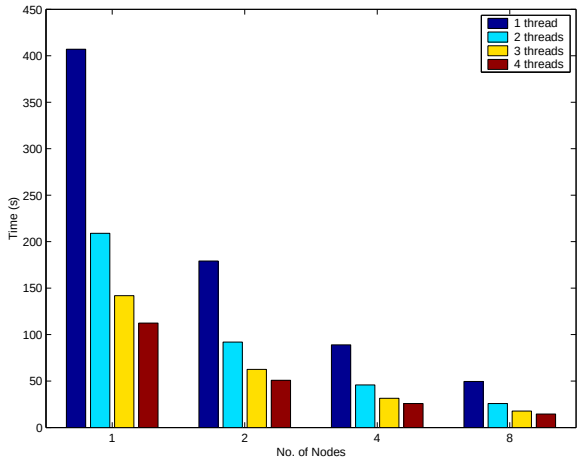

Figure 9: Performance of k-means (`ofl-c` version)

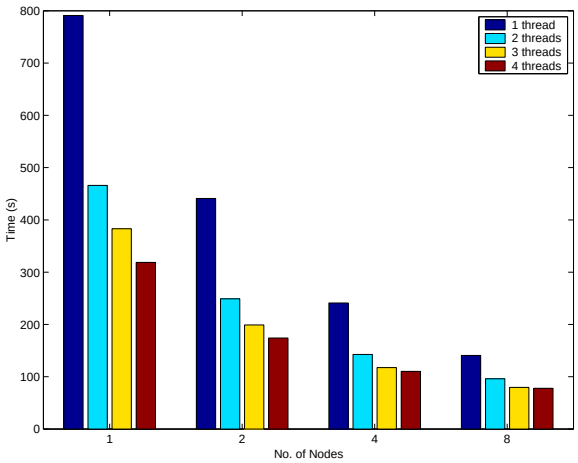

Figure 11: Performance of decision tree (`fr` version)

tively.

As compared to the other algorithms, this algorithm is I/O bound. As a result, shared memory scalability is limited. All the versions have a relative speedup of nearly 2.5 by using 4 threads on 1 node. Also, the communication volume is relatively high in decision tree construction. Therefore, distributed memory speedup is also limited to nearly 5 on 8 nodes. The best absolute speedup using 8 nodes and 4 threads on each node is nearly 10, and the best relative speedup is nearly 12.

In comparing across the different versions, `ofl-c` performs significantly better than `ofl` on 8 nodes. Because of the size of the reduction object, removing the overhead of communicating the locks results in significant improvements. `csl` version does not have the overhead of copying the reduction object or communication the locks. Therefore, it has higher relative distributed memory speedups than the over two locking versions. However, this version involves extra computation for calculating the address of the lock corresponding to an element. Therefore, the absolute speedups are not very high.

5. RELATED WORK

We now compare our work with related research efforts.

Significant amount of work has been done on parallelization of individual data mining techniques. Most of the work has been on distributed memory machines, including association mining [2, 9, 10, 23], k-means clustering [6], and decision tree classifiers [4, 8, 14, 18, 20]. Recent efforts have also focused on shared memory parallelization of data mining algorithms, including association mining [22, 16, 17] and decision tree construction [21].

We are not aware of any previous work on parallelizing k-means or decision tree construction on cluster of SMPs. There is a limited amount of work on parallelizing apriori association mining on cluster of SMPs [23]. In addition, our work is significantly different because we have developed parallelization techniques that are applicable across a number of data mining algorithms. Our shared memory parallelization techniques are also significantly different from the one previously developed for apriori association mining and decision tree construction.

Becuzzi *et al.* [5] have used a structured parallel programming environment PQE2000/SkIE for developing parallel implementation of data mining algorithms. However, they only focus on distributed memory parallelization. The similarity among parallel versions of different data mining techniques has also been observed by Skillicorn [19]. Our work is different in offering a middleware to exploit the similarity, and easing parallel implementations.

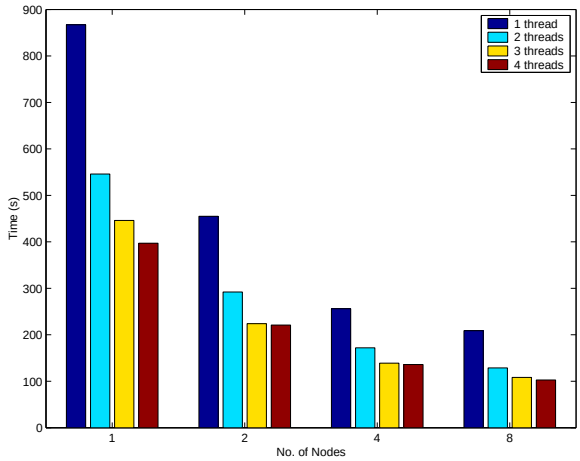

Figure 12: Performance of decision tree (`ofl` version)

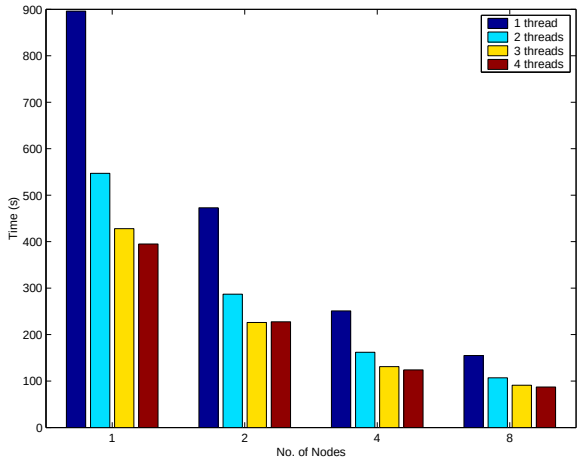

Figure 13: Performance of decision tree (`ofl-c` version)

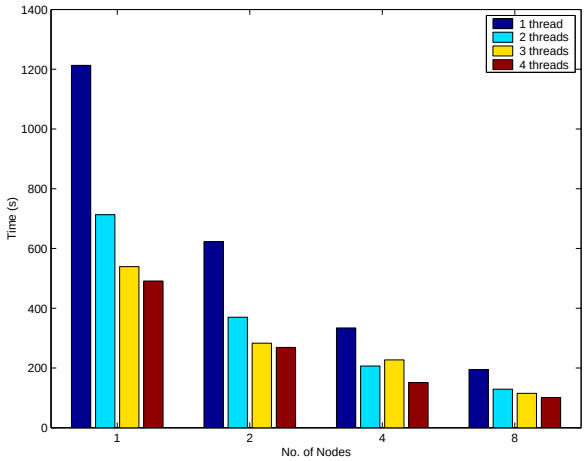

Figure 14: Performance of decision tree (`csl` version)

6. SUMMARY

In this paper, we focused on developing and evaluating techniques and a framework for parallelization of data mining algorithms. We have presented techniques for shared memory as well as distributed memory parallelization that apply across algorithms for a variety of data mining tasks.

Using 8 nodes and 4 threads on each node, we achieved a speedup of 28 on k-means clustering and 23 on apriori association mining. For decision tree construction, we are limited to distributed memory speedup of 5 on 8 nodes and shared memory speedup of 2.5 using 4 threads, because of high communication volume and I/O bandwidth limitations, respectively. In most cases, the four different approaches for combining shared memory and distributed memory parallelization we implemented had similar relative speedups.

In comparing our work with the existing work on parallel data mining, the speedups we achieve compare well against the reported performance from stand-alone implementations of individual parallel data mining algorithms, despite our using a common set of techniques and a middleware with a high-level interface. Thus, our work has shown that a common framework can be used for efficiently parallelizing algorithms for different data mining tasks.

7. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Eng., 5(6):914-925,*, December 1993.

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

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

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

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

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

[7] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *VLDB*, 1998.

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

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

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

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann,

Inc., San Francisco, 2nd edition, 1996.

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

[13] Ruoming Jin and Gagan Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the second SIAM conference on Data Mining*, April 2002.

[14] M. V. Joshi, G. Karypis, and V.Kumar. Scalparc: A new scalable and efficient parallel classification algorithm for mining large datasets. In *In Proc. of the International Parallel Processing Symposium*, 1998.

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

[16] 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.

[17] 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.

[18] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, pages 544–555, September 1996.

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

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

[21] M. J. Zaki, C.-T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. *IEEE International Conference on Data Engineering*, pages 198–205, May 1999.

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

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