A Localized Algorithm for Parallel Association Mining *

Mohammed Javeed Zaki, Srinivasan Parthasarathy, and Wei Li
Department of Computer Science, University of Rochester, Rochester, NY 14627
{zaki,srini,wei}@cs.rochester.edu

Abstract

Discovery of association rules is an important database mining problem. Mining for association rules involves extracting patterns from large databases and inferring useful rules from them. Several parallel and sequential algorithms have been proposed in the literature to solve this problem. Almost all of these algorithms make repeated passes over the database to determine the commonly occurring patterns or *itemsets* (set of items), thus incurring high I/O overhead. In the parallel case, these algorithms do a reduction at the end of each pass to construct the global patterns, thus incurring high synchronization cost.

In this paper we describe a new parallel association mining algorithm. Our algorithm is a result of detailed study of the available parallelism and the properties of associations. The algorithm uses a scheme to cluster related frequent itemsets together, and to partition them among the processors. At the same time it also uses a different database layout which clusters related transactions together, and selectively replicates the database so that the portion of the database needed for the computation of associations is local to each processor. After the initial set-up phase, the algorithm eliminates the need for further communication or synchronization. The algorithm further scans the local database partition only three times, thus minimizing I/O overheads. Unlike previous approaches, the algorithms uses simple intersection operations to compute frequent itemsets and doesn't have to maintain or search complex hash structures.

Our experimental testbed is a 32-processor DEC Alpha cluster inter-connected by the Memory Channel network. We present results on the performance of our algorithm on various databases, and compare it against a well known parallel algorithm. Our algorithm outperforms it by an more than an order of magnitude.

1 Introduction

Business organizations are increasingly turning to the automatic extraction of information from large volumes of routinely collected business data. Such high-level inference process may provide a host of useful information on customer groups, buying patterns, stock trends, etc. This process of automatic information inferencing is commonly known as Knowledge Discovery and Data mining (KDD). We look at one aspect of this process — mining for associations. Discovery of association rules is an important problem in database mining. The prototypical application is the analysis of sales or *basket* data [2]. Basket data consists of items bought by a customer along with the transaction identifier. Association rules have been shown to be useful in domains that range from decision support to telecommunications alarm diagnosis, and prediction.

1.1 Problem Statement

The problem of mining associations over basket data was introduced in [1]. It can be formally stated as: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. Each transaction T in the database $\mathcal{D}$ of transactions, has a unique identifier, and *contains* a set of items, called *itemset*, such that $T \subseteq \mathcal{I}$, i.e. each transaction is of the form <TID, $i_1, i_2, ..., i_k$>. An itemset with k items is called a *k-itemset*. A subset of length k is called a *k-subset*. An itemset is said to have a *support* s if $s\%$ of the transactions in $\mathcal{D}$ contain the itemset. An *association rule* is an expression $A \Rightarrow B$, where itemsets $A, B \subset \mathcal{I}$, and $A \cap B = \emptyset$. The *confidence* of the association rule, given as $support(A \cup B)/support(A)$, is simply the conditional probability that a transaction contains B, given that it contains A. The data mining task for association rules can be broken into two steps. The first step consists of finding all *frequent* itemsets, i.e., itemsets that occur in the database with a certain user-specified frequency, called *minimum support*. The second step consists of forming implication rules among the frequent itemsets [4]. The second step is relatively straightforward. Once the support of frequent itemsets is known, rules of the form $X - Y \Rightarrow Y$ (where $Y \subset X$), are generated for all frequent itemsets X, provided the rules meet the desired confidence. On the other hand the problem of identifying all frequent itemsets is hard. Given m items, there are potentially 2^m frequent itemsets. However, only a small fraction of the whole space of itemsets is frequent. Discovering the frequent itemsets requires a lot of computation power, memory and I/O, which can only be provided by parallel computers. Efficient parallel methods are needed to discover the relevant itemsets, and this is the focus of our paper.

1.2 Related Work

Sequential Algorithms Several algorithms for mining associations have been proposed in the literature [1, 10, 4, 8, 11,

*This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

9, 14, 2, 15]. The *Apriori* algorithm [10, 4, 2] was shown to have superior performance to earlier approaches [1, 11, 8, 9] and forms the core of almost all of the current algorithms. The key observation used is that all subsets of a frequent itemset must themselves be frequent. During the initial pass over the database the support for all single items (1-itemsets) is counted. The frequent 1-itemsets are used to generate candidate 2-itemsets. The database is scanned again to obtain their support, and the frequent 2-itemsets are selected for the next pass. This iterative process is repeated for $k = 3, 4, \cdots$, until there are no more frequent k-itemsets to be found. However, if the database is too large to fit in memory, these algorithms incur high I/O overhead for scanning it in each iteration. The *Partition* algorithm [14] minimizes I/O by scanning the database only twice. It partitions the database into small chunks which can be handled in memory. In the first pass it generates the set of all potentially frequent itemsets (any itemset locally frequent in a partition), and in the second pass their global support is obtained. Another way to minimize the I/O overhead is to work with only a small random sample of the database. An analysis of the effectiveness of sampling for association mining was presented in [17], and [15] presents an exact algorithm that finds all rules using sampling. The question whether one can efficiently extract all the rules in a single database pass has been addressed in [18]. They propose new algorithms which scan the database only once, generating all frequent itemsets. The performance gains are obtained by using efficient itemset clustering and candidate searching techniques.

Parallel Algorithms There has been relatively less work in parallel mining of associations. Three different parallelizations of *Apriori* on a distributed-memory machine (IBM SP2) were presented in [3]. The *Count Distribution* algorithm is a straightforward parallelization of *Apriori*. Each processor generates the partial support of all candidate itemsets from its local database partition. At the end of each iteration the global supports are generated by exchanging the partial supports among all the processors. The *Data Distribution* algorithm partitions the candidates into disjoint sets, which are assigned to different processors. However to generate the global support each processor must scan the entire database (its local partition, and all the remote partitions) in all iterations. It thus suffers from huge communication overhead. The *Candidate Distribution* algorithm also partitions the candidates, but it selectively replicates the database, so that each processor proceeds independently. The local portion is scanned once during each iteration.

The PDM algorithm [12] presents a parallelization of the DHP algorithm [11] on the IBM SP2. However, both PDM and DHP perform worse than *Count Distribution* [3] and *Apriori*. Distributed algorithms (DMA, FDM) are presented in [6, 5] which generate fewer candidates than *Count Distribution*, and use effective pruning techniques to minimize the messages for the support exchange step. In recent work we presented the CCPD parallel algorithm (based on *Apriori*) for shared memory machines [16]. It is similar in spirit to *Count Distribution*. The candidate itemsets are generated in parallel and are stored in a hash structure which is shared among all the processors. Each processor then scans its logical partition of the database and atomically updates the counts of candidates in the shared hash tree. There is no need to perform a sum-reduction to obtain global counts, but there is a barrier synchronization at the end of each iteration to ensure that all processors have updated the counts. The algorithm uses additional optimization such as computation balancing, hash-tree balancing and short-circuited subset counting to speed up performance [16].

1.3 Contribution

The main limitation of all the current parallel algorithms is that they make repeated passes over the disk-resident database partition, incurring high I/O overheads. Furthermore, the schemes involve exchanging either the counts of candidates or the remote database partitions during each iteration. This results in high communication and synchronization overhead. The previous algorithms also use complicated hash structures which entails additional overhead in maintaining and searching them, and typically also have poor cache locality [13].

The work in the current paper contrasts to these approaches in several ways. We present a new parallel algorithm – *Eclat* (Equivalence CLass Transformation), which clusters related frequent itemsets and transactions. It then distributes the work among the processors in such a way that each processor can compute the frequent itemsets independently, using simple intersection operations. The techniques help eliminate the need for synchronization after the initial set-up phase. The transaction clustering scheme which uses a vertical data layout enables us to scan the database only one more time after the initial phase, requiring only three database scans in all. This drastically cuts down the I/O overhead. Our experimental testbed is a 32-processor (8 nodes, 4 processors each) DEC Alpha cluster inter-connected by the Memory Channel [7] network. The Memory Channel allows a user-level application to write to the memory of remote nodes, thus allowing for very fast user-level messages and low synchronization costs. We experimentally compare our algorithm with previous approaches and show that it outperforms a well known parallel algorithm, *Count Distribution* by more than an order of magnitude.

The rest of the paper is organized as follows. We begin by providing more details on the sequential *Apriori* algorithm since all current parallel algorithms are based on it. Section 3 describes some of the previous parallel algorithms, namely the *Count Distribution* and *Candidate Distribution* algorithms. We present our itemset and transaction clustering techniques in section 4. Section 5 details the new *Eclat* algorithm. The implementation details for communication over the Memory Channel are provided in section 6. We then recapitulate the salient features of the new algorithm in section 7, before presenting the experimental results in section 8. Finally we present our conclusions in section 9.

2 Sequential Association Mining

In this section we will briefly describe the *Apriori* algorithm [2], since it forms the core of all parallel algorithms [3, 6, 5, 12, 16]. *Apriori* follows the basic iterative structure discussed earlier. Making use of the fact that any subset of a frequent itemset must also be frequent, during each iteration of the algorithm only candidates found to be frequent in the previous iteration are used to generate a new candidate set. A pruning step eliminates any candidate at least one of whose subsets is not frequent. The complete algorithm is shown in figure 1. It has three main steps. The candidates for the k-th pass are generated by joining L_{k-1} with itself, which can be expressed as

$$C_k = \{X = A[1]A[2]...A[k-1]B[k-1]\}$$

where $A, B \in L_{k-1}$, $A[1:k-2] = B[1:k-2]$, $A[k-1] < B[k-1]$, and $X[i]$ denotes the i-th item, while $X[i:j]$ denotes items at index i through j in itemset X. For example, let $L_2 = \{$AB, AC, AD, AE, BC, BD, BE, DE$\}$. Then $C_3 = \{$ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE$\}$.

```
$L_1 = \{$frequent 1-itemsets$\}$;
for ($k = 2; L_{k-1} \neq \emptyset; k++$)
  $C_k$ = Set of New Candidates;
  for all transactions $t \in \mathcal{D}$
    for all $k$-subsets $s$ of $t$
      if ($s \in C_k$) $s.count++$;
  $L_k = \{c \in C_k | c.count \geq$ minimum support$\}$;
Set of all frequent itemsets = $\bigcup_k L_k$;
```

Figure 1. The *Apriori* algorithm

Before inserting an itemset into C_k, *Apriori* tests whether all its $(k-1)$-subsets are frequent. This *pruning* step can eliminate a lot of unnecessary candidates. The candidates, C_k, are stored in a hash tree to facilitate fast support counting. An internal node of the hash tree at depth d contains a hash table whose cells point to nodes at depth $d+1$. All the itemsets are stored in the leaves. The insertion procedure starts at the root, and hashing on successive items, inserts the candidate in a leaf. For counting C_k, for each transaction in the database, all k-subsets of the transaction are generated in lexicographical order. Each subset is searched in the hash tree, and the count of the candidate incremented if it matches the subset. This is the most compute intensive step of the algorithm. The last step forms L_k by selecting itemsets meeting the minimum support criterion. For details on the performance characteristics of *Apriori* we refer the reader to [4].

3 Parallel Association Mining

In this section we will briefly look at some previous parallel algorithms. We will compare our new algorithm against CCPD – *Common Candidate Partitioned Database* algorithm [16]. Though originally designed for shared-memory machines, we ported the CCPD algorithm to run on the DEC cluster. It is essentially the same as *Count Distribution*, but uses some optimization techniques to balance the candidate hash tree, and to short-circuit the candidate search for fast support counting. More details on the optimizations can be found in [16]. Henceforth, we assume that CCPD and *Count Distribution* refer to the same algorithm. All the parallel algorithms assume that the database is partitioned among all the processors in equal-sized blocks, which reside on the local disk of each processor.

3.1 The *Count Distribution* Algorithm

The *Count Distribution* algorithm [3] is a simple parallelization of *Apriori*. All processors generate the entire candidate hash tree from L_{k-1}. Each processor can thus independently get partial supports of the candidates from its local database partition. This is followed by a sum-reduction to obtain the global counts. Note that only the partial counts need to be communicated, rather than merging different hash trees, since each processor has a copy of the entire tree. Once the global L_k has been determined each processor builds C_{k+1} in parallel, and repeats the process until all frequent itemsets are found. This simple algorithm minimizes communication since only the counts are exchanged among the processors. However, since the entire hash tree is replicated on each processor, it doesn't utilize the aggregate memory efficiently. The *Data Distribution* algorithm [3] was designed to utilize the total system memory by generating disjoint candidate sets on each processor. However to generate the global support each processor must scan the entire database (its local partition, and all the remote partitions) in all iterations. It thus suffers from high communication overhead, and performs very poorly when compared to *Count Distribution* [3].

3.2 The *Candidate Distribution* Algorithm

The *Candidate Distribution* algorithm [3] uses a property of frequent itemsets [3, 16] to partition the candidates during iteration l, so that each processor can generate disjoint candidates independent of other processors. At the same time the database is selectively replicated so that a processor can generate global counts independently. The choice of the redistribution pass involves a trade-off between decoupling processor dependence as soon as possible and waiting until sufficient load balance can be achieved. In their experiments the repartitioning was done in the fourth pass. After this the only dependence a processor has on other processors is for pruning the candidates. Each processor asynchronously broadcasts the local frequent set to other processors during each iteration. This pruning information is used if it arrives in time, otherwise it is used in the next iteration. Note that each processor must still scan its local data once per iteration. Even though it uses problem-specific information, it performs worse than *Count Distribution* [3]. *Candidate Distribution* pays the cost of redistributing the database, and it then scans the local database partition repeatedly. The redistributed database will usually be larger than $\mathcal{D}/P$, where $\mathcal{D}$ denotes the number of transactions and P the number of processors. The communication gains in later iterations are thus not sufficient to offset the redistribution cost. In the next section we show how problem-specific information can be used to develop an efficient algorithm that out-performs *Count Distribution* by more than an order of magnitude.

4 Itemset and Transaction Clustering

In this section we present a way to cluster related frequent itemsets together using the equivalence class partitioning scheme. Each equivalence class generates an independent set of candidates. We also present a technique to cluster related transactions together by using the vertical database layout. This facilitates fast support counting using simple intersections, rather than maintaining and searching complex data structures.

4.1 Equivalence Class Partitioning

Let's reconsider the candidate generation step of *Apriori*. Let L_2 = {AB, AC, AD, AE, BC, BD, BE, DE}. Then C_3 = {ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE}. Assuming that L_{k-1} is lexicographically sorted, we can partition the itemsets in L_{k-1} into **equivalence** classes based on their common $k-2$ length prefixes, i.e., the equivalence class $a \in L_{k-2}$, is given as:

$$S_a = [\mathbf{a}] = \{b \in L_{k-1} \mid a[1:k-2] = b[1:k-2]\}$$

Candidate k-itemsets can simply be generated from itemsets within a class by joining all $\binom{|S_i|}{2}$ pairs. For our example L_2 above, we obtain the equivalence classes: S_A = **[A]** = {AB, AC, AD, AE}, S_B = **[B]** = {BC, BD, BE}, and S_D = **[D]** = {DE}. We observe that itemsets produced by the equivalence class **[A]**, namely those in the set {ABC, ABD, ABE, ACD, ACE, ADE}, are independent of those produced by the class **[B]** (the set {BCD, BCE, BDE}. Any class with only 1 member can be eliminated since no candidates can be generated from it. Thus we can discard the class **[D]**. This idea of partitioning L_{k-1} into equivalence classes was independently proposed in [3, 16]. The equivalence partitioning was used in [16] to parallelize the candidate generation step in CCPD. It was also used in *Candidate Distribution* [3] to partition the candidates into disjoint sets.

4.2 Database Layout

Horizontal Data Layout The horizontal database layout, with each TID followed by the items in it, imposes some computation overhead during the support counting step. In particular for each transaction of average length l, during iteration k, we have to test whether all $\binom{l}{k}$ k-subsets of the transaction are contained in C_k. To perform fast subset checking the candidates are stored in a complex hash-tree data structure. Searching for the relevant candidates thus adds additional computation overhead. Furthermore, the horizontal layout forces us to scan the entire database or the local partition once in each iteration. Both *Count* and *Candidate Distribution* must pay the extra overhead entailed by using the horizontal layout.

Vertical Data Layout The vertical (or inverted) layout (also called the *decomposed storage structure* [8]) consists of a list of items, with each item followed by its *tid-list* – the list of all the transaction identifiers containing the item. The vertical layout doesn't suffer from any of the overheads described for the horizontal layout above due to the following three reasons: First, if the tid-list is sorted in increasing order, then the support of a candidate k-itemset can be computed by simply intersecting the tid-lists of any two $(k-1)$-subsets. No complicated data structures need to be maintained. We don't have to generate all the k-subsets of a transaction or perform the search operations on the hash tree. Second, the tid-lists contain all relevant information about an itemset, and enable us to avoid scanning the whole database to compute the support count of an itemset. This layout can therefore take advantage of the principle of locality. All information for an equivalence class is clustered together, so all large itemsets can be generated for it before moving on to the next class. Third, the larger the itemset, the shorter the tid-lists, which is practically always true. This results in faster intersections. For example, let the tid-list of AB, denoted as $\mathcal{T}(AB) = \{1, 5, 7, 10, 50\}$, and let $\mathcal{T}(AC) = \{1, 4, 7, 10, 11\}$. Then the tid-list of ABC is simply, $\mathcal{T}(AC) = \{1, 7, 10\}$. We can immediately determine the support by counting the number of elements in the tid-list. If it meets the minimum support criterion, we insert ABC in L_3.

The inverted layout, however, has a drawback. Examination of small itemsets tends to be costlier than when the horizontal layout is employed. This is because tid-lists of small itemsets provide little information about the association among items. In particular, no such information is present in the tid-lists for 1-itemsets. For example, a database with 1,000,000 (1M) transactions, 1,000 frequent items, and an average of 10 items per transaction has tid-lists of average size 10,000. To find frequent 2-itemsets we have to intersect each pair of items, which requires $\binom{1,000}{2} \cdot (2 \cdot 10{,}000) \approx 10^9$ operations. On the other hand, in the horizontal format we simply need to form all pairs of the items appearing in a transaction and increment their count, requiring only $\binom{10}{2} \cdot 1,000,000 = 4.5 \cdot 10^7$ operations. The *Eclat* algorithm thus uses the horizontal layout for generating L_2 and uses the vertical layout thereafter.

5 The *Eclat* Algorithm

The *Eclat* algorithm was designed to overcome the shortcomings of the *Count* and *Candidate Distribution* algorithms. It utilizes the aggregate memory of the system by partitioning the candidates into disjoint sets using the equivalence class partitioning. It decouples the dependence among the processors right in the beginning so that the redistribution cost can be amortized by the later iterations. Since each processor can proceed independently, there is no costly synchronization at the end of each iteration. Furthermore *Eclat* uses the vertical database layout which clusters all relevant information in an itemset's tid-list. Each processor computes all the frequent itemsets from one equivalence class before proceeding to the next. Thus the local database partition is scanned only once. In contrast *Candidate Distribution* must scan it once in each iteration. *Eclat* doesn't pay the extra computation overhead of building or searching complex data structures, nor does it have to generate all the subsets of each transaction. As the intersection is performed an itemset can immediately be inserted in L_k. Notice that the tid-lists also automatically prune irrelevant transactions. As the itemset size increases, the size of the tid-list decreases, resulting in very fast intersections. The *Eclat* algorithm has four distinct phases. The initialization phase, the transformation phase, the asynchronous phase and the final reduction phase. We will describe each step in detail below. Figures 2 and 3 present the pseudo-code for the *Eclat* algorithm.

5.1 Initialization Phase

The initialization step involves computing all the frequent 2-itemsets from the database. We don't count the support of single

```
Begin Eclat
  /* Initialization Phase*/
  Scan local database partition
  Compute local counts for all 2-itemsets
  Construct global $L_2$ counts

  /* Transformation Phase */
  Partition $L_2$ into equivalence classes
  Schedule $L_2$ over the set of processors $P$
  Transform local database into vertical form
  Transmit relevant tid-lists to other processors
  Local $L_2$ = receive tid-lists from other processors

  /* Asynchronous Phase */
  for each equivalence class $E_2$ in Local $L_2$
     Compute_Frequent($E_2$)

  /* Final Reduction Phase*/
  Aggregate Results and Output Associations
End Eclat
```

Figure 2: The *Eclat* Algorithm

itemsets, since with a very small space overhead the counts of 2-itemsets can be directly obtained in one pass, as opposed to paying the cost of scanning the database twice [1]. For computing 2-itemsets we use an upper triangular array, local to each processor, indexed by the items in the database in both dimensions. Each processor computes local support of each 2-itemset from its local database partition. This is followed by a sum-reduction among all the processors to construct global counts. At the end of the initial phase, all processors have the global counts of the frequent 2-itemsets, L_2, in the database.

5.2 Transformation Phase

The transformation step consists of two sub-steps. First, L_2 is partitioned using the equivalence class partitioning. The partitions are then assigned to the processors so that a suitable level of load-balancing is achieved. Second, the database is transformed from the horizontal to the vertical layout, and repartitioned so that each processor has on its local disk the tid-lists of all 2-itemsets in any equivalence class assigned to it.

5.2.1 Equivalence Class Scheduling

We first partition the L_2 into equivalence classes using the common prefix as described above. We next generate a schedule of the equivalence classes on the different processors in a manner minimizing the load imbalance. For this propose, each equivalence class is assigned a weighting factor based on the number of elements in the class. Since we have to consider all pairs for the next iteration, we assign the weight $\binom{s}{2}$ to a class with s elements. Once the weights are assigned we generate a schedule using a greedy heuristic. We sort the classes on the weights, and assign each class in turn to the least loaded processor, i.e., one having the least total weight at that point. Ties are broken by selecting the processor with the smaller identifier. These two steps are done concurrently on all the processors since all of them have access to the global L_2. Although the size of a class gives a good indication of the amount of work, better heuristics for generating the weights are possible. For example, if we could better estimate the number of frequent itemsets that could be derived from an equivalence class we could use this estimation as our weight. We could also make use of the average support of the itemsets within a class to get better weight factors (see [3] for one such heuristic). We believe that decoupling processor performance right in the beginning holds promise, even though it may cause some load imbalance, since the repartitioning cost can be amortized over later iterations. Deriving better heuristics for scheduling equivalence classes of L_2 is part of ongoing research.

[1] However, if the number of items is very large, it would be better to make two database scans.

5.2.2 Vertical Database Transformation

Once a balanced partitioning of the equivalence classes among the processors is generated, we transform the local database from the horizontal format to the vertical tid-list format. This can be achieved in two steps. First, each processor scans its local database and constructs partial tid-lists for all the frequent 2-itemsets. Second, each processor needs to construct the global tid-lists for itemsets in its equivalence classes. Each processor thus needs to send tid-lists for those itemsets belonging to other processors, while receiving tid-lists for the itemsets it is responsible for. The transformation phase is the most expensive step in our algorithm, since each processor has to exchange information with every other processor to read the non-local tid-lists over the Memory Channel network. More detail on the implementation of this step will be presented below in section 6.

5.3 Asynchronous Phase

```
Begin Compute_Frequent($E_{k-1}$)
  for all itemsets $I_1$ and $I_2$ in $E_{k-1}$
     if (($I_1$.tidlist $\cap$ $I_2$.tidlist) $\geq$ minsup)
        add ($I_1 \cup I_2$) to $L_k$
  Partition $L_k$ into equivalence classes.
  for each equivalence class $E_k$ in $L_k$
     Compute_Frequent($E_k$)
End Compute_Frequent
```

Figure 3: Procedure *Compute_Frequent*

At the end of the transformation phase the database has been redistributed, so that the tid-lists of all 2-itemsets in its local equivalence classes reside on the local disk. Each processor can independently compute all the frequent itemsets, eliminating the need for synchronization with other processors. We read the tid-lists for 2-itemsets within each equivalence class directly from the disk. We then generate all possible frequent itemsets from that class before moving on to the next class. This step involves scanning the inverted local database partition only once. We thus benefit from huge I/O savings and from the locality perspective as well.

Within each equivalence class we look at all pairs of 2-itemsets, and intersect their corresponding tid-lists. If the cardinality of the resulting tid-list exceeds the minimum support, the new itemset is inserted in L_3. Then we split the resulting frequent 3-itemsets, L_3 into equivalence classes based on common prefixes of length 2. All pairs of 3-itemsets within an equivalence are intersected to determine L_4. This process is repeated until there are no more frequent k-itemsets to be found. This recursive procedure is shown in figure 3. Note that once L_k has been determined, we can delete L_{k-1}. We thus need main memory space only for the itemsets in L_{k-1} within one equivalence class. The *Eclat* algorithm is therefore extremely main memory space efficient.

Short-Circuited Intersections The intersections between pairs of itemset tid-lists can be performed faster by utilizing the minimum support value. For example let's assume that the minimum support is 100, and we are intersecting two itemsets – AB with support 119 and AC with support 200. We can stop the intersection the moment we have 20 mismatches in AB, since the support of ABC is bounded above by 119. *Eclat* uses this short-circuit mechanism to optimize the tid-list intersections.

Pruning Candidates Recall that both *Count* and *Candidate Distribution* use a pruning step to eliminate unnecessary candidates. This step is essential in those algorithms to reduce the size of the hash tree. Smaller trees lead to faster support counting, since each subset of a transaction is tested against the tree. However, with the vertical database layout we found the pruning step to be of little or no help. This can be attributed to several factors. First, there is additional space and computation overhead in constructing and searching hash tables. This is also likely to degrade locality. Second, there is extra overhead in generating all the subsets of a candidate. Third, there is extra communication overhead in communicating the frequent itemsets in each iteration, even though it may happen asynchronously. Fourth, because the average size of tid-lists decreases as the itemsets size increases, intersections can be performed very quickly with the short-circuit mechanism.

5.4 Final Reduction Phase

At the end of the asynchronous phase we accumulate all the results from each processor and print them out.

6 Implementation Details

In this section we describe some implementation specific optimizations. We begin by a description of the DEC Memory Channel network, and then present the implementation details of the various communication steps of our algorithm.

6.1 The DEC Memory Channel

Digital's Memory Channel (MC) network [7] provides applications with a global address space using memory mapped regions.

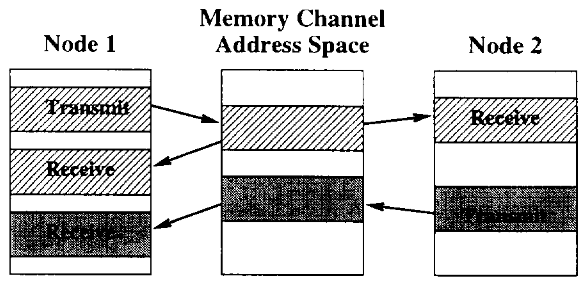

Figure 4: Memory Channel space. The lined region is mapped for both transmit and receive on node 1 and for receive on node 2. The gray region is mapped for receive on node 1 and for transmit on node 2.

A region can be mapped into a process' address space for transmit, receive, or both. Virtual addresses for transmit regions map into physical addresses located in I/O space on the MC's PCI adapter. Virtual addresses for receive regions map into physical RAM. Writes into transmit regions are collected by the source MC adapter, forwarded to destination MC adapters through a hub, and transferred via DMA to receive regions with the same global identifier (see figure 4). Regions within a node can be shared across different processors on that node. Writes originating on a given node will be sent to receive regions on that same node only if *loop-back* has been enabled for the region. We do not use the loop-back feature. We use *write-doubling* instead, where each processor writes to its receive region and then to its transmit region, so that processes on a host can see modification made by other processes on the same host. Though we pay the cost of double writing, we reduce the amount of messages to the hub.

In our system unicast and multicast process-to-process writes have a latency of 5.2 μs, with per-link transfer bandwidths of 30 MB/s. MC peak aggregate bandwidth is also about 32 MB/s. Memory Channel guarantees write ordering and local cache coherence. Two writes issued to the same transmit region (even on different nodes) will appear in the same order in every receive region. When a write appears in a receive region it invalidates any locally cached copies of its line.

6.2 Initialization Phase

This is a straightforward implementation of the pseudo-code presented in figure 2. Once the local counts for all 2-itemsets are obtained, we need to perform a sum-reduction to obtain the global counts. We allocate an array of size $\binom{m}{2}$, (m is the number of items) on the shared Memory Channel region. Each processor then accesses this shared array in a mutually exclusive manner, and increments the current count by its partial counts. It then waits at a barrier for the last processor to update the shared array [2]. After all processors have updated the shared array, each processor sees the global counts for all 2-itemsets. Each processor also broadcasts the local partial counts of the frequent

[2]On P processors, the sum-reduction can be performed more efficiently in $\mathcal{O}(log(P))$ steps. Since it is performed only once in *Eclat*, we opted for the simple $\mathcal{O}(P)$ process described above.

Global and Partial Support for L2

	L2	12	13	15	23	25	34	35
	Global Support	10	13	10	15	16	14	17
Partial Support	P0	3	2	10	4	11	8	5
	P1	3	10	0	7	1	3	5
	P2	4	1	0	4	4	3	7

Equivalence Class Partitioning of L2

P0 - (12, 13, 15)
P1 - (23, 25)
P2 - (34, 35)

Vertical Database Transformation

Legend: Tid-List from Processor P0 ■ P1 ▨ P2 ▩

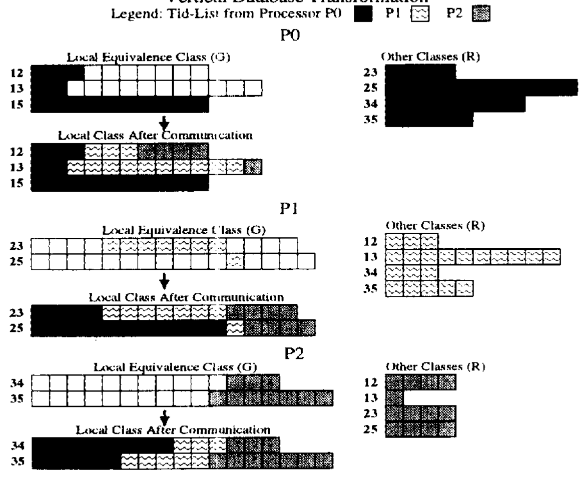
P0
Local Equivalence Class (G)
12
13
15
Local Class After Communication
12
13
15
Other Classes (R)
23
25
34
35
P1
Local Equivalence Class (G)
23
25
Local Class After Communication
23
25
Other Classes (R)
12
13
34
35
P2
Local Equivalence Class (G)
34
35
Local Class After Communication
34
35
Other Classes (R)
12
13
23
25

Figure 5: Vertical Database Transformation

2-itemsets to all the other processors. The partial counts are used to construct the inverted global tid-lists efficiently.

6.3 Transformation Phase

Each processor scans it local database partition a second time and constructs the vertical tid-lists for the frequent 2-itemsets, L_2. Since the original database is initially partitioned in a block fashion, each processor's inverted database consists of disjoint ranges of tids. We make use of this information, along with the knowledge of the partial counts, to place the incoming tid-list from a given processor at an appropriate offset, so that the global tid-list appears lexicographically sorted. This saves us the cost of sorting each tid-list if the transactions were distributed in a random manner. The transformation is accomplished in two steps:

Local Tid-list Transformation To perform the inversion, we break L_2 into two groups. Those itemsets belonging to local equivalence classes assigned to the processor, denoted as G, and those itemsets belonging to other processors, denoted as R. Each processor, P_i, memory maps an anonymous memory region of size $\sum_g$ global_count(g) + $\sum_r$ partial_count(r, P_i), where itemsets $g \in G$, $r \in R$, P_i denotes the processor, and partial_count(r, P_i) is the partial count of itemset r on processor P_i. Each processor then performs the transformation, writing its tid-list for the members of G at the appropriate offset. Members of R are written starting at offset zero. Figure 5 depicts the database transformation step on three processors.

Tid-list Communication Once the transformation of the local database is done, we need to receive the partial tid-lists from other processors for all 2-itemsets in G, and we need to communicate the tid-lists of R to other processors. The incoming tid-lists are again copied at the appropriate offsets. Since the ranges of transaction are distinct and monotonically increasing, the final tid-lists for each 2-itemset appear lexicographically sorted by using the above approach. The tid-lists of itemsets in G are then written out to disk, while those in R are discarded. To communicate the partial tid-lists across the Memory Channel, we take advantage of the fast user-level messages. Each processor allocates a 2MB buffer[3] for a transmit region and a receive region, sharing the same identifier. The communication proceeds in a lock-step manner with alternating write and read phases. In the write phase each processor writes the tid-lists of itemsets in P into its transmit region, until we reach the buffer limit. At this point it enters the read phase, where it scans each processor's receive region in turn, and places the tid-lists belonging to G at the appropriate offsets. Once the read region has been scanned it enters the write phase. This process is repeated until all partial tid-lists are received. At the end of this phase the database is in the vertical tid-list format. Figure 5 shows this process pictorially. Each processor then enters the asynchronous phase, and computes the frequent itemsets, as described in section 5.3. The final reduction is implemented in the same manner as the reduction in the initialization phase.

7 Salient Features of *Eclat*

In this section we will recapitulate the salient features of *Eclat*, contrasting it against *Count* and *Candidate Distribution*. *Eclat* differs from these algorithms in the following respect:

- Unlike *Count Distribution*, *Eclat* utilizes the aggregate memory of the parallel system by partitioning the candidate itemsets among the processors using equivalence class partitioning. It shares this feature with *Candidate Distribution*.
- *Elcat* decouples the processors right in the beginning by repartitioning the database, so that each processor can compute the frequent itemsets independently. It thus eliminates the need for communicating the frequent itemsets at the end of each iteration.
- *Elcat* uses a different database layout which clusters the transactions containing an itemset into tid-lists. Using this layout enables our algorithm to scan the local database partition only three times on each processor. The first scan for building L_2, the second for transforming the database, and the third for obtaining the frequent itemsets. In contrast, both *Count* and *Candidate Distribution* scan the database multiple times – once during each iteration.
- To compute frequent itemsets, *Eclat* performs simple intersections on two tid-lists. There is no extra overhead associated with building and searching complex hash tree data structures. Such complicated hash structures also suffer from poor cache locality [13]. In contrast, all the available memory in *Eclat* is utilized to keep tid-lists in memory which results in good locality. As larger itemsets are generated the size of tid-lists decreases, resulting in very fast intersections. Short-circuiting the join based

[3] A smaller buffer size can be used if there is a constraint on the shared MC space. Since we had approximately 90-100MB of shared MC space with 32 processors, we chose the 2MB buffer size.

on minimum support is also used to speed this step.
• Our algorithm avoids the overhead of generating all the subsets of a transaction and checking them against the candidate hash tree during support counting.
• *Eclat* recursively uses the equivalence class partitioning during each iteration to cluster related itemsets. At any given point only L_{k-1} within one equivalence class needs to be kept in memory. The algorithm thus uses higher levels of the memory hierarchy efficiently.
• The one disadvantage of our algorithm is the virtual memory it requires to perform the transformation. Our current implementation uses memory mapped regions to accomplish this, however, we are currently implementing an external memory transformation, keeping only small buffers in main memory. Our algorithm may need roughly twice the disk space of the other algorithms, since we use the horizontal layout for the initial phase, and the vertical layout thereafter (once we have the new format we can delete the former). As we shall see, the performance gains shown in the next section more than offset this disadvantage.

8 Experimental Evaluation

Database	T	I	$\mathcal{D}$	Total Size
T10.I6.D800K	10	6	800,000	35 MB
T10.I6.D1600K	10	6	1,600,000	68 MB
T10.I6.D3200K	10	6	3,200,000	138 MB
T10.I6.D6400K	10	6	6,400,000	274 MB

Table 1: Database properties

All the experiments were performed on a 32-processor (8 hosts, 4 processors each) DEC Alpha cluster inter-connected via the Memory Channel. Each Alpha processor runs at 233MHz. There's a total of 256MB of main memory per host (shared among the 4 processors on that host). Each host also has a 2GB local disk attached to it, out of which less than 500MB was available to us. All the partitioned databases reside on the local disks of each processor.

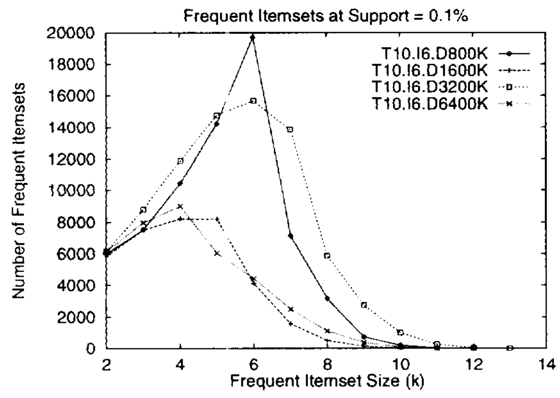

Figure 6: Number of Frequent k-Itemsets

We used different synthetic databases with size ranging form 35MB to 274MB, which were generated using the procedure described in [4]. These have been used as benchmark databases for many association rules algorithms [4, 8, 11, 14, 2], and they mimic the transactions in a retailing environment. Each transaction has a unique ID followed by a list of items bought in that transaction. The data-mining provides information about the set of items generally bought together. Table 1 shows the databases used and their properties. The number of transactions is denoted as $|\mathcal{D}|$, average transaction size as $|T|$, and the average maximal potentially frequent itemset size as $|I|$. The number of maximal potentially frequent itemsets $|L| = 2000$, and the number of items $N = 1000$. We refer the reader to [4] for more detail on the database generation. All the experiments were performed with a minimum support value of 0.1%. The support was kept small so that there were enough frequent itemsets generated. Figure 6 shows the total number of frequent itemsets of different sizes found for the different databases at the above support value.

8.1 *Eclat*: Parallel Performance

Configuration			Database	CD	*Eclat(E)*		CD/E
T	H	P		Total	Total	Setup	Ratio
1	1	1	D800K	1746.0s	98.8s	54.2s	17.7
			D1600K	842.9s	161.4s	113.3s	5.2
			D3200K	3483.0s	540.7s	344.0s	6.4
2	2	1	D800K	1470.0s	53.3s	28.9s	27.6
			D1600K	757.8s	84.1s	54.8s	9.0
			D3200K	2455.9s	215.9s	122.9s	11.4
4	2	2	D800K	1552.7s	34.4s	16.2s	45.1
			D1600K	669.5s	53.8s	38.1s	12.4
			D3200K	2305.1s	144.1s	74.8s	16.0
4	4	1	D800K	1326.5s	32.4s	14.1s	40.9
			D1600K	399.5s	45.7s	29.4s	8.7
			D3200K	1670.2s	113.1s	62.0s	14.8
8	4	2	D800K	1466.5s	23.1s	10.0s	63.5
			D1600K	544.2s	29.7s	16.9s	18.3
			D3200K	1627.8s	77.7s	45.8s	20.9
8	8	1	D800K	1267.3s	24.5s	10.6s	51.7
			D1600K	314.5s	28.3s	15.9s	11.11
			D3200K	1497.2s	67.5s	38.5s	22.2
16	8	2	D800K	1414.1s	27.7s	7.7s	51.1
			D1600K	312.3s	24.9s	14.3s	12.5
			D3200K	1620.5	49.7s	26.9s	32.6
24	8	3	D800K	2112.9s	29.3s	8.7s	72.1
			D1600K	542.5	30.7s	13.3s	17.7
			D3200K	2048.6	51.8s	20.5s	39.5

Table 2: Total Execution Time: *Eclat (E)* vs. *Count Distribution (CD)* (P: #processors/host; H: #Hosts; T: Total #processors)

In this section we will compare the performance of our algorithm with *Count Distribution*, which was shown to be superior to both *Data* and *Candidate Distribution* [3]. In table 2 we give the running times of both algorithms under different processor configurations and on different databases. In all the figures H denotes the number of hosts, P the number of processors per host, and $T = H \cdot P$, the total number of processors used in the experiments. The times shown are the total execution time in seconds. For *Eclat* we also show the break-up for the time spent in the initialization and transformation phase. The last column of the table gives the improvement ratio or speed-up factor obtained by using *Eclat*. Table 2 shows that our algorithm clearly outperforms *Count Distribution* by more than an order of magnitude for most configurations and databases with the improvement ranging between 5 and 18 for the sequential case and between 9 and 70 for the parallel case. This improvement can be attributed to several factors which have been enumerated in section 7. First, *Count Distribution* performs a sum-reduction, and communicates the local counts in each iteration, while *Eclat* eliminates this step entirely. For example T10.I6.D800K has 12 iterations and the synchronization times accumulate over all the iterations. Second, there is no provision for load balancing in *Count Distribution*. The databases are partitioned in equal-sized blocks, while the amount of work may be different for each partition, especially if the transaction sizes are skewed. There is no straightforward way to (re)distribute the work in this algorithm without adding huge communication overhead. While *Eclat* may also suffer from load imbalance, it tries to minimize this in the equivalence class scheduling step (section 5.2.1). Third, *Eclat* utilizes the aggregate memory better and dispenses with maintaining complex hash structures which may suffer from poor cache locality [13]. All available memory is used for the tid-lists and simple intersection operations are performed on these lists, which have good locality. Fourth, *Count Distribution* suffers from high I/O overheads because of multiple scans of the database (12 iterations imply 12 scans).

From table 2 we can also observe that the transformation phase dominates (roughly 55-60%) the total execution of *Eclat*, while the simple intersections of tid-lists facilitate fast frequent itemset determination. In *Count Distribution* the support count phase dominates, where subsets of a transaction are generated and a search is performed on the candidate hash tree. This produces an interesting result. Consider the T10.I6.D800K and T10.I6.D1600K databases. Even though T10.I6.D800K is half the size of T10.I6.D1600K, figure 6 shows that it has more than twice as many frequent itemsets. In *Count Distribution*, T10.I6.D800K generates a much larger hash tree, making it more expensive than T10.I6.D1600K. On the other hand in *Eclat* the larger database, T10.I6.D1600K, takes longer to transform, and hence takes longer time. This fact also explains why we see the best improvement ratio for the T10.I6.D800K database. Since it is small, the transformation is very cheap, and at the same time it generates a lot of frequent itemsets, increasing the time for *Count Distribution*.

Figure 7 shows the speedup obtained for *Eclat* on the different databases on various configuration. The speedup numbers are with respect to a sequential run of the algorithm on the given database. However, the T10.I6.D6400K speedups are with respect to the $P = 1, H = 4, T = 4$ configuration (214.6 sec). Since our current implementation uses memory mapped regions to perform the transformation, we did not have enough space to perform the transformation on a single processor[4].

The figures indicate that with increase in the number of processors per host, there is an improvement only if there is sufficient work. The current implementation of *Eclat* doesn't distinguish between hosts (H) and processors per host (P). It simply partitions the database into T (the total number of processor) chunks. Since all the processors will be accessing the local disk simulta-

[4]If we used uniprocessor time we would get a super-linear speedup with more hosts, since the local database partition size would decrease, and would fit in the memory mapped region.

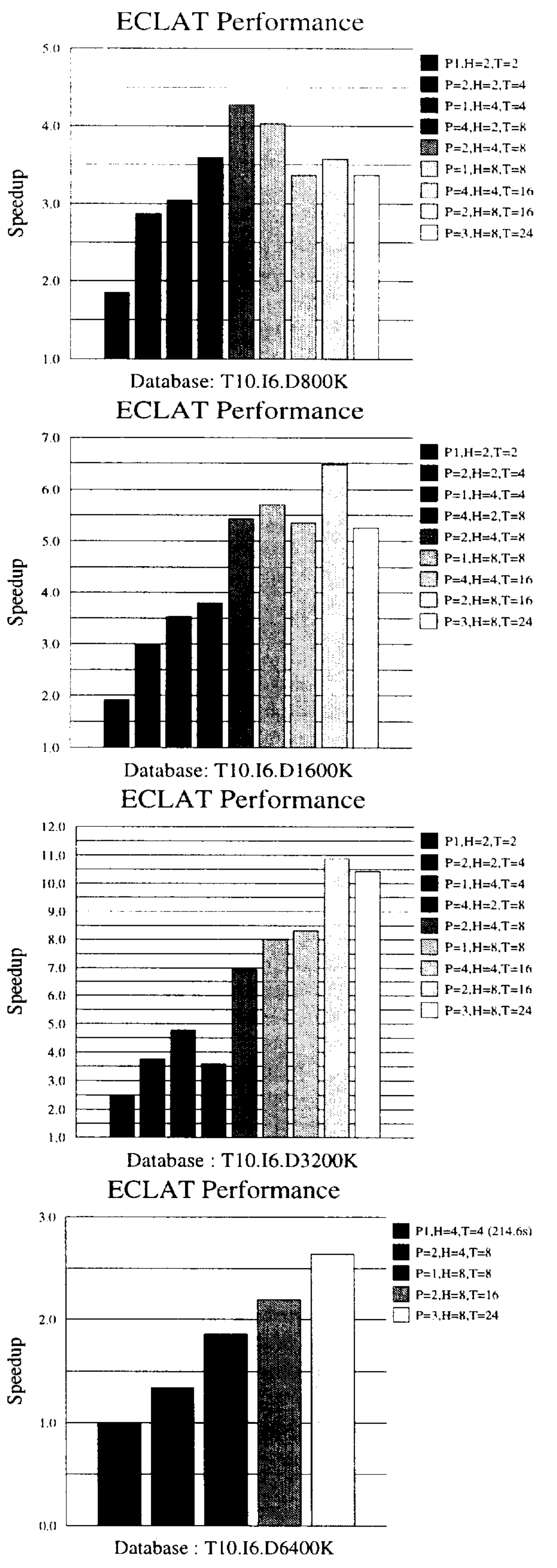

Figure 7: ECLAT: Parallel Performance on Different Databases

neously, we will suffer from a lot of disk contention. This is borne by the results on 8 hosts for T10.I6.D800K. While the relative computation time decreases with increasing number of hosts, the disk contention causes performance degradation with increasing number of processors on each host. The same effect can be observed for *Count Distribution* since it too doesn't use the system configuration information. It also takes an additional hit since the entire hash tree is replicated P times on each host. To solve the local disk contention problem, we plan to modify the current implementations to make use of configuration-specific information. We plan to implement a hybrid parallelization where the database is partitioned only among the hosts. Within each host the processors could share the candidate hash tree in *Count Distribution*, while the *Compute_Frequent* procedure (section 5.3) could be carried out in parallel in *Eclat*.

To further support this fact, for the same number of total processors, *Eclat* does better on configurations that have fewer processors per host. For example, consider the $T = 8$ case. Out of the three configurations — $(H = 2, P = 4)$; $(H = 4, P = 2)$; and $(H = 8, P = 1)$, the last always performs the best (see figure 7). This can also be attributed to the relative cost of computation and disk contention. Speedups with increasing number of hosts for a fixed P are typically very good. The speedups for the larger databases (T10.I6.D3200K and T10.I6.D6400K) are close to linear as we go from $H = 2$ to $H = 8$ for $P = 1$. However, with increasing database sizes, we see performance improvements even with multiple processors on the same host. This is because of the increased computation versus disk I/O cost ratio.

9 Conclusions

In this paper we described *Eclat* — a localized parallel algorithm for association mining. It uses techniques to cluster related groups of itemsets using equivalence class partitioning, and to cluster transactions using the vertical database layout. It then schedules the equivalence classes among the processors, minimizing load imbalance, and repartitions the vertical database so that each processor can compute the frequent itemsets independently. This eliminates the need to communicate in each iteration. *Eclat* scans the local database partitions only three times gaining significantly from the I/O overhead savings. Furthermore, it uses simple intersection operations to determine the frequent itemsets. This feature enables the algorithm to have good cache locality. It also dispenses with keeping complex hash structures in memory, which suffer from poor locality. We implemented *Eclat* on a 32 processor DEC cluster interconnected with the DEC Memory Channel network, and compared it against a well known parallel algorithm *Count Distribution* [3]. Experimental results indicate a substantial improvement of more than an order of magnitude over the previous algorithm.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.

[3] R. Agrawal and J. Shafer. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engg.*, pages 8(6):962–969, 1996.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th VLDB Conf.*, Sept. 1994.

[5] D. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. *4th Intl. Conf. Parallel and Distributed Info. Systems*, Dec. 1996.

[6] D. Cheung, V. Ng, A. Fu, and Y. Fu. Efficient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engg.*, pages 8(6):911–922, 1996.

[7] R. Gillett. Memory channel: An optimized cluster interconnect. In *IEEE Micro, 16(2)*, Feb. 1996.

[8] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *1st Intl. Conf. Knowledge Discovery and Data Mining*, Aug. 1995.

[9] M. Houtsma and A. Swami. Set-oriented mining of association rules in relational databases. In *11th Intl. Conf. Data Engineering*, 1995.

[10] H. Mannila, H. Toivonen, and I. Verkamo. Efficient algorithms for discovering association rules. In *AAAI Wkshp. Knowledge Discovery in Databases*, July 1994.

[11] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[12] J. S. Park, M. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *ACM Intl. Conf. Information and Knowledge Management*, Nov. 1995.

[13] S. Parthasarathy, M. J. Zaki, and W. Li. Application driven memory placement for dynamic data structures. Technical Report URCS TR 653, University of Rochester, Apr. 1997.

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

[15] H. Toivonen. Sampling large databases for association rules. In *22nd VLDB Conf.*, 1996.

[16] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multiprocessors. In *Proc. Supercomputing'96*, Nov. 1996.

[17] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. In *7th Intl. Wkshp. Research Issues in Data Engg*, Apr. 1997.

[18] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. Technical Report URCS TR 651, University of Rochester, Apr. 1997.