A Fast Distributed Algorithm for Mining Association Rules *

David W. Cheung† Jiawei Han‡ Vincent T. Ng†† Ada W. Fu‡‡ Yongjian Fu‡

† Department of Computer Science, The University of Hong Kong, Hong Kong. Email: dcheung@cs.hku.hk.

‡ School of Computing Science, Simon Fraser University, Canada. Email: han@cs.sfu.ca.

†† Department of Computing, Hong Kong Polytechnic University, Hong Kong. Email: cstyng@comp.polyu.edu.hk.

‡‡ Department of Computer Science and Engineering, The Chinese University of Hong Kong,Hong Kong. Email: adafu@cs.cuhk.hk.

Abstract

*With the existence of many large transaction databases, the huge amounts of data, the high scalability of distributed systems, and the easy partition and distribution of a centralized database, it is important to investigate efficient methods for distributed mining of association rules. This study discloses some interesting relationships between locally large and globally large itemsets and proposes an interesting distributed association rule mining algorithm, FDM (*Fast Distributed Mining of association rules*), which generates a small number of candidate sets and substantially reduces the number of messages to be passed at mining association rules. Our performance study shows that FDM has a superior performance over the direct application of a typical sequential algorithm. Further performance enhancement leads to a few variations of the algorithm.*

1 Introduction

An association rule is a rule which implies certain association relationships among a set of objects (such as "occur together" or "one implies the other") in a database. Since finding interesting association rules in databases may disclose some useful patterns for decision support, selective marketing, financial forecast, medical diagnosis, and many other applications, it has attracted a lot of attention in recent data mining research [5]. Mining association rules may require iterative scanning of large transaction or relational databases which is quite costly in processing. Therefore, efficient mining of association rules in transaction and/or relational databases has been studied substantially [1, 2, 4, 8, 10, 11, 12, 14, 15].

Previous studies examined efficient mining of association rules from many different angles. An influential association rule mining algorithm, Apriori [2], has been developed for rule mining in large transaction databases. A DHP algorithm [10] is an extension of Apriori using a hashing technique. The scope of the study has also been extended to efficient mining of sequential patterns [3], generalized association rules [14], multiple-level association rules [8], quantitative association rules [15], etc. Maintenance of discovered association rules by incremental updating has been studied in [4]. Although these studies are on sequential data mining techniques, algorithms for parallel mining of association rules have been proposed recently [11, 1].

We feel that the development of distributed algorithms for efficient mining of association rules has its unique importance, based on the following reasoning. (1) Databases or data warehouses [13] may store a huge amount of data. Mining association rules in such databases may require substantial processing power, and distributed system is a possible solution. (2) Many large databases are distributed in nature. For example, the huge number of transaction records of hundreds of Sears department stores are likely to be stored at different sites. This observation motivates us to study efficient distributed algorithms for mining association rules in databases. This study may also shed new light on parallel data mining. Furthermore, a distributed mining algorithm can also be used to mine association rules in a single large database by partitioning the database among a set of sites and processing the task in a distributed manner. The high flexibility, scalability, low cost performance ratio, and easy connectivity of a distributed system makes it an ideal platform for mining association rules.

In this study, we assume that the database to be studied is a transaction database although the method can be easily extended to relational databases as well. The database consists of a huge number of transaction records, each with a transaction identifier (TID) and a set of data items. Further, we assume that the

*The research of the first author was supported in part by RGC (the Hong Kong Research Grants Council) grant 338/065/0026. The research of the second author was supported in part by the research grant NSERC-A3723 from the Natural Sciences and Engineering Research Council of Canada, the research grant NCE:IRIS/PRECARN-HMI5 from the Networks of Centres of Excellence of Canada, and a research grant from Hughes Research Laboratories.

database is "horizontally" partitioned (i.e., grouped by transactions) and allocated to the sites in a distributed system which communicate by message passing. Based on these assumptions, we examine distributed mining of association rules. It has been well known that the major cost of mining association rules is the computation of the set of *large itemsets* (i.e., *frequently occurring sets of items*, see Section 2.1) in the database [2]. Distributed computing of large itemsets encounters some new problems. One may compute *locally large* itemsets easily, but a locally large itemset may not be *globally large*. Since it is very expensive to broadcast the whole data set to other sites, one option is to broadcast all the counts of all the itemsets, no matter locally large or small, to other sites. However, a database may contain enormous combinations of itemsets, and it will involve passing a huge number of messages.

Based on our observation, there exist some interesting properties between locally large and globally large itemsets. One should maximally take advantages of such properties to reduce the number of messages to be passed and confine the substantial amount of processing to local sites. As mentioned before, two algorithms for parallel mining of association rules have been proposed. The two proposed algorithms PDM and Count Distribution (CD) are designed for share-nothing parallel systems [11, 1]. However, they can also be adapted to distributed environment. We have proposed an efficient distributed data mining algorithm FDM (F*ast* D*istributed* M*ining of association rules*), which has the following distinct feature in comparison with these two proposed parallel mining algorithms.

1. The generation of candidate sets is in the same spirit of Apriori. However, some interesting relationships between locally large sets and globally large ones are explored to generate a smaller set of candidate sets at each iteration and thus reduce the number of messages to be passed.

2. After the candidate sets have been generated, two pruning techniques, *local pruning* and *global pruning*, are developed to prune away some candidate sets at each individual sites.

3. In order to determine whether a candidate set is large, our algorithm requires only $O(n)$ messages for support count exchange, where n is the number of sites in the network. This is much less than a straight adaptation of Apriori, which requires $O(n^2)$ messages.

Notice that several different combinations of the local and global prunings can be adopted in FDM. We studied three versions of FDM: *FDM-LP, FDM-LUP*, and *FDM-LPP* (see Section 4), with similar framework but different combinations of pruning techniques. FDM-LP only explores the *local pruning*; FDM-LUP does both local pruning and the *upper-bound-pruning*; and FDM-LPP does both local pruning and the *polling-site-pruning*.

Extensive experiments have been conducted to study the performance of FDM and compare it against the Count Distribution algorithm. The study demonstrates the efficiency of the distributed mining algorithm.

The remaining of the paper is organized as follows. The tasks of mining association rules in sequential as well as distributed environments are defined in Section 2. In Section 3, techniques for distributed mining of association rules and some important results are discussed. The algorithms for different versions of FDM are presented in Section 4. A performance study is reported in Section 5. Our discussions and conclusions are presented respectively in Sections 6 and 7.

2 Problem Definition

2.1 Sequential Algorithm for Mining Association Rules

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of *items*. Let DB be a database of transactions, where each transaction T consists of a set of items such that $T \subseteq I$. Given an *itemset* $X \subseteq I$, a transaction T *contains* X if and only if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$ and $X \cap Y = \emptyset$. The association rule $X \Rightarrow Y$ holds in DB with *confidence* c if the probability of a transaction in DB which contains X also contains Y is c. The association rule $X \Rightarrow Y$ has *support* s in DB if the probability of a transaction in DB contains both X and Y is s. The task of mining association rules is to find all the association rules whose support is larger than a *minimum support threshold* and whose confidence is larger than a *minimum confidence threshold*.

For an itemset X, its *support* is the percentage of transactions in DB which contains X, and its *support count*, denoted by $X.sup$, is the number of transactions in DB containing X. An itemset X is *large* (or more precisely, *frequently occurring*) if its support is no less than the minimum support threshold. An itemset of size k is called a k*-itemset*. It has been shown that the problem of mining association rules can be reduced to two subproblems [2]: (1) *find all large itemsets for a given minimum support threshold*, and (2) *generate the association rules from the large itemsets found*. Since (1) dominates the overall cost of mining association rules, the research has been focused on how to develop efficient methods to solve the first subproblem [2].

An interesting algorithm, *Apriori* [2], has been proposed for computing large itemsets at mining association rules in a transaction database. There have been many studies on mining association rules using sequential algorithms in centralized databases (e.g., [10, 14, 8, 12, 4, 15]), which can be viewed as variations or extensions to Apriori. For example, as an extension to Apriori, the DHP algorithm [10] uses a direct hashing technique to eliminate some size-2 candidate sets in the Apriori algorithm.

2.2 Distributed Algorithm for Mining Association Rules

We examine the mining of association rules in a distributed environment. Let DB be a database with D transactions. Assume that there are n sites $S_1, S_2, \ldots, S_n$ in a distributed system and the database DB is partitioned over the n sites into $\{DB_1, DB_2, \ldots, DB_n\}$, respectively.

Let the size of the partitions DB_i be D_i, for $i = 1, \ldots, n$. Let $X.sup$ and $X.sup_i$ be the support counts of an itemset X in DB and DB_i, respectively. $X.sup$ is called the *global support count*, and $X.sup_i$ the *local support count* of X at site S_i. For a given minimum support threshold s, X is *globally large* if $X.sup \geq s \times D$; correspondingly, X is *locally large* at site S_i, if $X.sup_i \geq s \times D_i$. In the following, L denotes the globally large itemsets in DB, and $L_{(k)}$ the globally large k-itemsets in L. The essential task of a distributed association rule mining algorithm is to find the globally large itemsets L.

For comparison, we outline the Count Distribution (CD) algorithm as the follows [1]. The algorithm is an adaptation of the Apriori algorithm in the distributed case. At each iteration, CD generates the candidate sets at every site by applying the Apriori_gen function on the set of large itemsets found at the previous iteration. Every site then computes the local support counts of all these candidate sets and broadcasts them to all the other sites. Subsequently, all the sites can find the globally large itemsets for that iteration, and then proceed to the next iteration.

3 Techniques for Distributed Data Mining

3.1 Generation of Candidate Sets

It is important to observe some interesting properties related to large itemsets in distributed environments since such properties may substantially reduce the number of messages to be passed across network at mining association rules.

There is an important relationship between large itemsets and the sites in a distributed database: *every globally large itemsets must be locally large at* some *site(s)*. If an itemset X is *both globally large and locally large* at a site S_i, X is called gl-large at site S_i. The set of gl-large itemsets at a site will form a basis for the site to generate its own candidate sets.

Two monotonic properties can be easily observed from the locally large and gl-large itemsets. First, if an itemset X is locally large at a site S_i, then all of its subsets are also locally large at site S_i. Secondly, if an itemset X is gl-large at a site S_i, then all of its subsets are also gl-large at site S_i. Notice that a similar relationship exists among the large itemsets in the centralized case. Following is an important result based on which an effective technique for candidate sets generation in the distributed case is developed.

Lemma 1 *If an itemset X is globally large, there exists a site S_i, $(1 \leq i \leq n)$, such that X and all its subsets are gl-large at site S_i.*

Proof. If X is not locally large at any site, then $X.sup_i < s \times D_i$ for all $i = 1, \ldots, n$. Therefore, $X.sup < s \times D$, and X cannot be globally large. By contradiction, X must be locally large at some site S_i, and hence X is gl-large at S_i. Consequently, all the subsets of X must also be gl-large at S_i. □

We use GL_i to denote the set of gl-large itemsets at site S_i, and $GL_{i(k)}$ to denote the set of gl-large k-itemsets at site S_i. It follows from Lemma 1 that if $X \in L_{(k)}$, then there exists a site S_i, such that all its size-$(k-1)$ subsets are gl-large at site S_i, i.e., they belong to $GL_{i(k-1)}$.

In a straightforward adaptation of Apriori, the set of candidate sets at the k-th iteration, denoted by $CA_{(k)}$, which stands for size-k candidate sets from Apriori, would be generated by applying the Apriori_gen function on $L_{(k-1)}$. That is,

$$CA_{(k)} = \text{Apriori\_gen}(L_{(k-1)}).$$

At each site S_i, let $CG_{i(k)}$ be the set of candidates sets generated by applying Apriori_gen on $GL_{i(k-1)}$, i.e.,

$$CG_{i(k)} = \text{Apriori\_gen}(GL_{i(k-1)}),$$

where CG stands for candidate sets generated from gl-large itemsets. Hence $CG_{i(k)}$ is generated from $GL_{i(k-1)}$. Since $GL_{i(k-1)} \subseteq L_{(k-1)}$, $CG_{i(k)}$ is a subset of $CA_{(k)}$. In the following, we use $CG_{(k)}$ to denote the set $\cup_{i=1}^{n} CG_{i(k)}$.

Theorem 1 *For every $k > 1$, the set of all large k-itemsets $L_{(k)}$ is a subset of $CG_{(k)} = \bigcup_{i=1}^{n} CG_{i(k)}$, where $CG_{i(k)} = \text{Apriori\_gen}(GL_{i(k-1)})$.*

Proof. Let $X \in L_{(k)}$. It follows from Lemma 1 that there exists a site S_i, $(1 \leq i \leq n)$, such that all the size-$(k-1)$ subsets of X are gl-large at site S_i. Hence $X \in CG_{i(k)}$. Therefore,

$$L_{(k)} \subseteq CG_{(k)} = \bigcup_{i=1}^{n} CG_{i(k)} = \bigcup_{i=1}^{n} \text{Apriori\_gen}(GL_{i(k-1)}).$$

□

Theorem 1 indicates that $CG_{(k)}$, which is a subset of $CA_{(k)}$ and could be much smaller than $CA_{(k)}$, can be taken as the set of candidate sets for the size-k large itemsets. The difference between the two sets, $CA_{(k)}$

and $CG_{(k)}$, depends on the distribution of the itemsets. This theorem forms a basis for the generation of the set of candidate sets in the algorithm FDM. First the set of candidate sets $CG_{i(k)}$ can be generated locally at each site S_i at the k-th iteration. After the exchange of support counts, the gl-large itemsets $GL_{i(k)}$ in $CG_{i(k)}$ can be found at the end of that iteration. Based on $GL_{i(k)}$, the candidate sets at S_i for the $(k+1)$-st iteration can then be generated. According to the performance study in Section 5, by using this approach, the number of candidate sets generated can be substantially reduced to about $10-25\%$ of that generated in CD.

Example 1 illustrates the effectiveness of the reduction of candidate sets using Theorem 1.

Example 1 Assuming there are 3 sites in a system which partitions the DB into DB_1, DB_2 and DB_3. Suppose the set of large 1-itemsets (computed at the first iteration) $L_{(1)} = \{A, B, C, D, E, F, G, H\}$, in which A, B, and C are locally large at site S_1, B, C, and D are locally large at site S_2, and E, F, G, and H are locally large at site S_3. Therefore, $GL_{1(1)} = \{A, B, C\}$, $GL_{2(1)} = \{B, C, D\}$, and $GL_{3(1)} = \{E, F, G, H\}$. Based on Theorem 1, the set of size-2 candidate sets at site S_1 is $CG_{1(2)}$, where $CG_{1(2)} =$ Apriori_gen $(GL_{1(1)}) = \{AB, BC, AC\}$. Similarly, $CG_{2(2)} = \{BC, CD, BD\}$, and $CG_{3(2)} = \{EF, EG, EH, FG, FH, GH\}$. Hence, the set of candidate sets for large 2-itemsets is $CG_{(2)} = CG_{1(2)} \cup CG_{2(2)} \cup CG_{3(2)}$, total 11 candidates. However, if Apriori_gen is applied to $L_{(1)}$, the set of candidate sets $CA_{(2)} =$ Apriori_gen$(L_{(1)})$ would have 28 candidates. This shows that it is very effective to apply Theorem 1 to reduce the candidate sets. □

3.2 Local Pruning of Candidate Sets

The previous subsection shows that based on Theorem 1, one can usually generate in a distributed environment a much smaller set of candidate sets than the direct application of the Apriori algorithm.

When the set of candidate set $CG_{(k)}$ is generated, to find the globally large itemsets, the support counts of the candidate sets must be exchanged among all the sites. Notice that some candidate sets in $CG_{(k)}$ can be pruned by a *local pruning* technique before count exchange starts. The general idea is that at each site S_i, if a candidate set $X \in CG_{i(k)}$ is not locally large at site S_i, there is no need for S_i to find out its global support count to determine whether it is globally large. This is because in this case, either X is small (not globally large), or it will be locally large at some other site, and hence only the site(s) at which X is locally large need to be responsible to find the global support count of X. Therefore, in order to compute all the large k-itemsets, at each site S_i, the candidate sets can be confined to only the sets $X \in CG_{i(k)}$ which are locally large at site S_i. For convenience, we use $LL_{i(k)}$ to denote those candidate sets in $CG_{i(k)}$ which are locally large at site S_i. Based on the above discussion, at every iteration (the k-th iteration), the gl-large k-itemsets can be computed at each site S_i according to the following procedure.

1. Candidate sets generation: Generate the candidate sets $CG_{i(k)}$ based on the gl-large itemsets found at site S_i at the $(k-1)$-st iteration using the formula, $CG_{i(k)} =$ Apriori_gen $(GL_{i(k-1)})$.
2. Local pruning: For each $X \in CG_{i(k)}$, scan the partition DB_i to compute the local support count $X.sup_i$. If X is not locally large at site S_i, it is excluded from the candidate sets $LL_{i(k)}$. (Note: This pruning only removes X from the candidate set at site S_i. X could still be a candidate set at some other site.)
3. Support count exchange: Broadcast the candidate sets in $LL_{i(k)}$ to other sites to collect support counts. Compute their global support counts and find all the gl-large k-itemsets in site S_i.
4. Broadcast mining results: Broadcast the computed gl-large k-itemsets to all the other sites.

For clarity, the notations used so far are listed in Table 1.

D	number of transactions in DB
s	support threshold $minsup$
$L_{(k)}$	globally large k-itemsets
$CA_{(k)}$	candidate sets generated from $L_{(k)}$
$X.sup$	global support count of X
D_i	number of transactions in DB_i
$GL_{i(k)}$	gl-large k-itemsets at S_i
$CG_{i(k)}$	candidate sets generated from $GL_{i(k-1)}$
$LL_{i(k)}$	locally large k-itemsets in $CG_{i(k)}$
$X.sup_i$	local support count of X at S_i

Table 1: Notation Table.

To illustrate the above procedure, we continue working on Example 1 as follows.

Example 2 Assume the database in Example 1 contains 150 transactions and each one of the 3 partitions has 50 transactions. Also assume that the support threshold $s = 10\%$. Moreover, according to Example 1, at the second iteration, the candidate sets generated at site S_1 are $CG_{1(2)} = \{AB, BC, AC\}$; at site S_2, $CG_{2(2)} = \{BC, BD, CD\}$; and at site S_3, $CG_{3(2)} = \{EF, EG, EH, FG, FH, GH\}$. In order to compute the large 2-itemsets, the local support counts

$X.sup_1$		$X.sup_2$		$X.sup_3$	
AB	5	BC	10	EF	8
BC	10	CD	8	EG	3
AC	2	BD	4	EH	4
–	-	–	-	FG	3
–	-	–	-	FH	4
–	-	–	-	GH	6

Table 2: Locally Large Itemsets.

at each site is computed first. The result is recorded in Table 2.

From Table 2, it can be seen that $AC.sup_1 = 2 < s \times D_1 = 5$. AC is not locally large. Hence, the candidate set AC is pruned away at site S_1. On the other hand, both AB and BC have enough local support counts and they survive the local pruning. Hence $LL_{1(2)} = \{AB, BC\}$. Similarly, $LL_{2(2)} = \{BC, CD\}$, and $LL_{3(2)} = \{EF, GH\}$. After the local pruning, the number of size-2 candidate sets has been reduced to five which is less than half of the original size. Once the local pruning is completed, each site broadcasts messages containing all the remaining candidate sets to the other sites to collect their support counts. The result of this count support exchange is recorded in Table 3.

locally large candidates	broadcast request from	$X.sup_1$	$X.sup_2$	$X.sup_3$
AB	S_1	5	4	4
BC	S_1, S_2	10	10	2
CD	S_2	4	8	4
EF	S_3	4	3	8
GH	S_3	4	4	6

Table 3: Globally Large Itemsets.

The request for support count for AB is broadcasted from S_1 to site S_2 and S_3, and the counts sent back are recorded at site S_1 as in the second row of Table 3. The other rows record similar count exchange activities at the other sites. At the end of the iteration, site S_1 finds out that only BC is gl-large, because $BC.sup = 22 > s \times D = 15$, and $AB.sup = 13 < s \times D = 15$. Hence the gl-large 2-itemset at site S_1 is $GL_{1(2)} = \{BC\}$. Similarly, $GL_{2(2)} = \{BC, CD\}$ and $GL_{3(2)} = \{EF\}$. After the broadcast of the gl-large itemsets, all sites return the large 2-itemsets $L_{(2)} = \{BC, CD, EF\}$.

Notice that some candidate set, such as BC in this example, could be locally large at more than one site. In this case, the messages are broadcasted from all the sites at which BC is found to be locally large. This is unnecessary because for each of candidate itemset, only one broadcast is needed. In Section 3.4, an optimization technique to eliminate such redundancy will be discussed. □

There is a subtlety in the implementation of the four steps outlined above for finding globally large itemsets. In order to support both step 2, "local pruning", and step 3, "support count exchange", each site S_i must have two sets of support counts. For local pruning, S_i has to find the local support counts of its candidate sets $CG_{i(k)}$. For support count exchange, S_i has to find the local support counts of some possibly different candidate sets from other sites in order to answer the count requests from these sites. A simple approach would be to scan DB_i twice, once for collection of the counts for the local $CG_{i(k)}$, and once for responding to the count requests from other sites. However, this would substantially degrade the performance.

In fact, there is no need of two scans. At S_i, not only is $CG_{i(k)}$ available at the beginning of the k-th iteration, but also are other sets, i.e., $CG_{j(k)}$ ($j = 1, \ldots, n$, $j \neq i$), because all the $GL_{i(k-1)}$, ($i = 1, \ldots, n$), are broadcasted to every site at the end of the $(k-1)$-st iteration, and the sets of candidate sets $CG_{i(k)}$, ($i = 1, \ldots, n$), are computed from the corresponding $GL_{i(k-1)}$. That is, at the beginning of each iteration, since all the gl-large itemsets found at the previous iteration have been broadcasted to all the sites, every site can compute the candidate sets of every other site. Therefore, the local support counts of all these candidate sets can be found in one scan and stored in a data structure like the hash-tree used in Apriori [2]. Using this technique, the data structure can be built in one scan, and the two different sets of support counts required in the local pruning and support count exchange can be retrieved from this data structure.

3.3 Global Pruning of Candidate Sets

The local pruning at a site S_i uses only the local support counts found in DB_i to prune a candidate set. In fact, the local support counts from other sites can also be used for pruning. A *global pruning* technique is developed to facilitate such pruning and is outlined as follows. At the end of each iteration, all the local support and global support counts of a candidate set X are available. These local support counts can be broadcasted together with the global support counts after a candidate set is found to be globally large. Using this information, some global pruning can be performed on the candidate sets at the subsequent iteration.

Assume that the local support count of every candidate itemset is broadcasted to all the sites after it is found to be globally large at the end of an iteration. Suppose X is a size-k candidate itemset at the k-th iteration. Therefore, the local support counts of all the size-$(k-1)$ subsets of X are available at every site. With respect to a partition DB_i, $(1 \leq i \leq n)$, we use $maxsup_i(X)$ to denote the minimum value of the local support counts of all the size-$(k-1)$ subsets of X, i.e, $maxsup_i(X) = min\{Y.sup_i \mid Y \subset X \; and \; |Y| = k-1\}$. It follows from the subset relationship that $maxsup_i(X)$ is an upper bound of the local support count $X.sup_i$. Hence, the sum of these upper bounds over all partitions, denoted by $maxsup(X)$, is an upper bound of $X.sup$. In other words, $X.sup \leq maxsup(X) = \sum_{i=1}^{n} maxsup_i(X)$. Note that $maxsup(X)$ can be computed at every site at the beginning of the k-th iteration. Since $maxsup(X)$ is an upper bound of its global support count, it can be used for pruning, i.e., if $maxsup(X) < s \times D$, then X cannot be a candidate itemset. This technique is called *global pruning*.

Global pruning can be combined with local pruning to form different pruning strategies. Two particular variations of this strategy will be adopted when we introduce several versions of FDM in Section 4. The first method is called *upper-bound-pruning* and the second one is called *polling-site-pruning*. We will discuss the upper-bound-pruning method here in detail. The polling-site-pruning method will be explained in Subsection 4.3. In the upper-bound-pruning, a site S_i first uses the techniques in Subsections 3.1 and 3.2 to generate and perform local pruning on the candidate sets. Before count exchange starts, the site S_i applies global pruning to the remaining candidate sets. A possible upper bound of the global support count of a candidate set X is the sum

$$X.sup_i + \sum_{j=1, j \neq i}^{n} maxsup_j(X).$$

where $X.sup_i$ is found already in the local pruning. Therefore, this upper bound can be computed to prune the candidate set X at site S_i.

Example 3 We examine the global pruning at S_1 after the local pruning done in Example 2. According to Table 2, the survived candidate sets in the local pruning are AB and BC. Their local support counts at S_1 can be found in Table 2. Furthermore, the local support counts of their subsets from all the sites are also available at S_1 and are listed in Table 4.

From Tables 2 and 4, an upper bound of the support count of AB, (denoted by $AB.\overline{sup}$), is given by

$$AB.\overline{sup} = AB.sup_1 + min(A.sup_2, B.sup_2) + min(A.sup_3, B.sup_3) = 5 + 4 + 4 = 13 < s \times D.$$

Since this upper bound is less than the support threshold, AB is removed from the set of candidate itemsets.

large 1-itemset	local support count at S_1		
	$X.sup_1$	$X.sup_2$	$X.sup_3$
A	6	4	4
B	10	10	5
C	4	12	5

Table 4: Local Support Counts.

On the other hand, an upper bound of the support count of BC, (denoted by $BC.\overline{sup}$), is given by

$$BC.\overline{sup} = BC.sup_1 + min(B.sup_2, B.sup_2) + min(B.sup_3, C.sup_3) = 10 + 10 + 5 = 25 > s \times D.$$

Since it is larger than the threshold, BC is not pruned away and remains as a candidate itemset at S_1. □

Global pruning is a useful technique for reducing the number of candidate sets. Its effectiveness depends on the distribution of the local support counts.

3.4 Count Polling

In the CD algorithm, the local support count of every candidate itemset is broadcasted from every site to every other site. Therefore, the number of messages required for count exchange for each candidate itemset is $O(n^2)$, where n is the number of partitions.

In our method, if a candidate itemset X is locally large at a site S_i, S_i needs $O(n)$ messages to collect all the support counts for X. In general, few candidate itemsets are locally large at all the sites. Therefore, the FDM algorithm will usually require much less than $O(n^2)$ messages for computing each candidate itemset. To ensure that FDM requires only $O(n)$ messages for every candidate itemset in all the cases, a count polling technique is introduced.

For each candidate itemset X, the technique uses an assignment function, which could be a hash function on X, to assign X a *polling site* (assuming that the assignment function is known to every site.) The polling site assigned to X is independent of the sites in which X is founded to be locally large. Therefore, even if X is found to be locally large at more than one site, it will still be sent to the same polling site. For each candidate itemset X, its polling site is responsible to find out whether X is globally large. To achieve that purpose, the polling site of X has to broadcast the polling request for X, collect the local support counts, and compute the global support count. Since there is only one polling site for each candidate itemset X, the number of messages required for count exchange for X is reduced to $O(n)$.

At the k-th iteration, after the pruning phase, (both local and global pruning), has been completed, FDM uses the following procedure at each site S_i to do the count polling.

1. Send candidate sets to polling sites: At site S_i, for every polling site S_j, find all the candidate itemsets in $LL_{i(k)}$ whose polling site is S_j and store them in $LL_{i,j(k)}$ (i.e., candidates are being put into groups by their polling sites). The local support counts of the candidate itemsets are also stored in the corresponding set $LL_{i,j(k)}$. Send each $LL_{i,j(k)}$ to the corresponding polling site S_j.

2. Poll and collect support counts: If S_i is a polling site, S_i receives all $LL_{j,i(k)}$ sent to it from the other sites. For every candidate itemset X received, S_i finds the list of originating sites from which X is being sent. S_i then broadcasts the polling requests to the other sites not on the list to collect the support counts.

3. Compute gl-large itemsets: S_i receives the support counts from the other sites, computes the global support counts for its candidates, and finds the gl-large itemsets. Eventually, S_i broadcasts the gl-large itemsets together with their global support counts to all the sites.

Example 4 In Example 2, assuming that S_1 is assigned as the polling site of AB and BC, S_2 is assigned as the polling site of CD, and S_3 is assigned as the polling site of EF and GH.

Following from the assignment, site S_1 is responsible for the polling of AB and BC. In the simple case of AB, S_1 sends polling requests to S_2 and S_3 to collect the support counts. As for BC, it is locally large at both S_1 and S_2, the pair $\langle BC, BC.sup_2\rangle = \langle BC, 10\rangle$ is sent to S_1 by S_2. After S_1 receives the message, it sends a polling request to the remaining site S_3. Once the support count $BC.sup_3 = 2$ is received from S_3, S_1 finds out that $BC.sup = 10 + 10 + 2 = 22 > 15$. Hence BC is a gl-large itemset at S_1. In this example, with a polling site, the double polling messages for BC has been eliminated. □

4 Algorithm for Distributed Mining of Association Rules

In this section, the basic version of FDM, i.e., the FDM-LP (FDM with Local Pruning) algorithm, is first presented, which adopts two techniques: *candidate set reduction* and *local pruning*, discussed in Section 3. According to our performance study in Section 5, FDM-LP is much more efficient than CD.

4.1 The FDM-LP algorithm

Algorithm 1 FDM-LP: FDM with Local Pruning

Input: DB_i $(i = 1, \ldots, n)$: the database partition at each site S_i.

Output: L: the set of all globally large itemsets.

Method: Iteratively execute the following program fragment (for the k-th iteration) distributively at each site S_i. The algorithm terminates when either $L_{(k)} = \emptyset$, or the set of candidate sets $CG_{(k)} = \emptyset$.

(1) if $k = 1$ then
(2) $\quad T_{i(1)} = get\_local\_count(DB_i, \emptyset, 1)$
(3) else {
(4) $\quad CG_{(k)} = \cup_{i=1}^{n} CG_{i(k)}$
$\qquad\qquad = \cup_{i=1}^{n} Apriori\_gen(GL_{i(k-1)})$;
(5) $\quad T_{i(k)} = get\_local\_count(DB_i, CG_{(k)}, i)$; }
(6) for_all $X \in T_{i(k)}$ do
(7) $\quad$ if $X.sup_i \geq s \times D_i$ then
(8) $\qquad$ for $j = 1$ to n do
(9) $\qquad\quad$ if $polling\_site(X) = S_j$ then
$\qquad\qquad$ insert $\langle X, X.sup_i\rangle$ into $LL_{i,j(k)}$;
(10) for $j = 1, \ldots, n$ do send $LL_{i,j(k)}$ to site S_j;
(11) for $j = 1, \ldots, n$ do {
(12) $\quad$ receive $LL_{j,i(k)}$;
(13) $\quad$ for_all $X \in LL_{j,i(k)}$ do {
(14) $\qquad$ if $X \notin LP_{i(k)}$ then
$\qquad\quad$ insert X into $LP_{i(k)}$;
(15) $\qquad$ update $X.large\_sites$; } }
(16) for_all $X \in LP_{i(k)}$ do
(17) $\qquad send\_polling\_request(X)$;
(18) $reply\_polling\_request(T_{i(k)})$;
(19) for_all $X \in LP_{i(k)}$ do {
(20) $\quad$ receive $X.sup_j$ from the sites S_j,
$\quad$ where $S_j \notin X.large\_sites$;
(21) $\quad X.sup = \sum_{i=1}^{n} X.sup_i$;
(22) $\quad$ if $X.sup \geq s \times D$ then
$\qquad$ insert X into $G_{i(k)}$; }
(23) broadcast $G_{i(k)}$;
(24) receive $G_{j(k)}$ from all other sites S_j, $(j \neq i)$;
(25) $L_{(k)} = \cup_{i=1}^{n} G_{i(k)}$.
(26) divide $L_{(k)}$ into $GL_{i(k)}$, $(i = 1, \ldots, n)$;
(27) return $L_{(k)}$.

Explanation of Algorithm 1

In Algorithm 1, every site S_i is initially a "home site" of a set of candidate sets that it generates. Later, it becomes a polling site to serve the requests from other sites. Subsequently, it changes its status to a remote site to supply local support counts to other polling sites. The corresponding steps in Algorithm 1 for these different roles and activities are grouped and explained as the follows.

1. Home site: generate candidate sets and submit them to polling sites (lines 1 - 10).

   At the first iteration, the site S_i calls *get_local_count* to scan the partition DB_i once and store the local support counts of all the 1-itemsets found in the array $T_{i(1)}$. At the k-th (for

$k > 1$) iteration, S_i first computes the set of candidate set $CG_{(k)}$, and then scans DB_i to build the hash tree $T_{i(k)}$ containing the locally support counts of all the sets in $CG_{(k)}$. By traversing $T_{i(k)}$, S_i finds out all locally large k-itemsets and group them according to their polling sites. Finally, it sends the candidate sets with their local support counts to their polling sites.

2. Polling site: receive candidate sets and send polling requests (lines 11 - 17).

   As a polling site, site S_i receives candidate sets from the other sites and insert them in $LP_{i(k)}$. For each candidate set $X \in LP_{i(k)}$, S_i stores all its "home" sites in $X.large\_sites$, which contains all those sites from which X is sent to S_i for polling. In order to perform count exchange for X, S_i calls $send\_polling\_request$ to send X to those sites not in the list $X.large\_sites$ to collect the remaining support counts.

3. Remote site: return support counts to polling site (line 18).

   When S_i receives polling requests from the other sites, it acts as a remote site. For each candidate set Y it receives from a polling site, it retrieves $Y.sup_i$ from the hash tree $T_{i(k)}$ and returns it to the polling site.

4. Polling site: receive support counts and find large itemsets (lines 19 - 23).

   As a polling site, S_i receives the local support counts for the candidate sets in $LP_{i(k)}$. Following that, it computes the global support counts of all these candidate sets and find out the globally large itemsets among them. These globally large k-itemsets are stored in the set $G_{i(k)}$. Finally, S_i broadcasts the set $G_{i(k)}$ to all the other sites.

5. Home site: receive large itemsets (lines 24 - 27).

   As a "home" site, S_i receives the sets of globally large k-itemsets $G_{i(k)}$ from all the polling sites. By taking the union of $G_{i(k)}$, $(i = 1, \ldots, n)$, S_i finds out the set L_k of all the size-k large itemsets. Further, S_i finds out from L_k the set $GL_{i(k)}$ of gl-large itemsets for each site by using the site list in $X.large\_sites$. The sets $GL_{i(k)}$ will be used for candidate set generation at the next iteration. □

The FDM-LP described above has utilized the techniques described in Subsections 3.1, 3.2, and 3.4. An illustration of FDM-LP by example can be found in Examples 1, 2 and 3 together.

In the following, two refinements of FDM-LP, by adoption of different global pruning techniques, are presented.

4.2 The FDM-LUP algorithm

Algorithm 2 FDM-LUP: FDM with Local and Upper-Bound-Pruning

Method: The program fragment of FDM-LUP is obtained from FDM-LP by inserting the following condition (line 7.1) after line 7 of Algorithm 1.

(7.1) if $g\_upper\_bound(X) \geq s \times D$ then

Explanation of Algorithm 2

The only new step in FDM-LUP is the one for upper-bound-pruning (line 7.1). The function g_upper_bound computes an upper bound for a candidate set X according to the formula suggested in Subsection 3.3. In other words, g_upper_bound returns an upper bound of X as the sum

$$X.sup_i + \sum_{j=1, j \neq i}^{n} maxsup_j(X).$$

As explained in Subsection 3.3, $X.sup_i$ is computed already in the local pruning step, and the values of $maxsup_j(X)$, $(j = 1, \ldots, n,\ j \neq i)$, can be computed from the local support counts from the $(k-1)$-st iteration. If this upper bound is smaller than the global support threshold, it is used to prune away X. FDM-LUP should usually have a smaller number of candidate sets for count exchange in comparison with FDM-LP. □

4.3 The FDM-LPP algorithm

Algorithm 3 FDM-LPP: FDM with Local Pruning and Polling-Site-Pruning

Method: The program fragment of FDM-LPP is obtained from Algorithm 1 by replacing its line 17 with the following two lines.

(16.1) if $p\_upper\_bound(X) \geq s \times D$ then
(17) $send\_polling\_request(X)$;

Explanation of Algorithm 3

The new step in FDM-LPP is the one for polling-site-pruning (line 16.1). At that stage, S_i is a polling site and has received requests from the other sites to perform polling. Each request contains a locally large itemset X and its local support count $X.sup_j$, where S_j is a site from which X is sent to S_i. Note that $X.large\_sites$ is the set of all the originating sites from which the requests for polling X are being sent to the polling site (line 15). For every site $S_j \in X.large\_sites$, the local support count $X.sup_j$ has been sent to S_i already. For a site $S_q \notin X.large\_sites$, since X is not locally large at S_q, its local support count $X.sup_q$ must be smaller than the local threshold $s \times D_q$. Following from the discussion in Subsection 3.3, $X.sup_q$ is bounded by the value $min(maxsup_q(X), s \times D_q - 1)$. Hence an upper bound of $X.sup$ can be computed by the sum

$$\sum_{j \in X.large\_sites} X.sup_j + \sum_{q=1, q \notin X.large\_sites}^{n} min(maxsup_q(X), s \times D_q - 1).$$

In FDM-LPP, S_i calls *p_upper_bound* to compute an upper bound for $X.sup$ according to the above formula. This upper bound can be used to prune away X if it is smaller than the global support threshold. □

As discussed before, both FDM-LUP and FDM-LPP may have less candidate sets than FDM-LP. However, they require more storage and communication messages for the local support counts. Their efficiency comparing with FDM-LP will depend largely on the data distribution.

5 Performance Study of FDM

An in-depth performance study has been performed to compare FDM with CD. We have chosen to implement the representative version of FDM, FDM-LP, and compare it against CD. Both algorithms are implemented on a distributed system by using PVM (Parallel Virtual Machine) [6]. A series of three to six RS/6000 workstations, running the AIX system, are connected by a 10Mb LAN to perform the experiment. The databases in the experiment are composed of synthetic data.

In the experiment result, the number of candidate sets found in FDM at each site is between $10-25\%$ of that in CD. The total message size in FDM is between $10-15\%$ of that in CD. The execution time of FDM is between $65-75\%$ of that in CD. The reduction in the number of candidate sets and message size in FDM is very significant. The reduction in execution time is also substantial. However, it is not directly proportional to the reduction in candidate sets and message size. This is mainly due to the overhead of running FDM and CD on PVM. What we have observed is that the overhead of PVM in FDM is very close to that in CD, even though the amount of message communication is significantly smaller in FDM. From the results of our experiments, it is also clear that the performance gain of FDM over CD will be higher in distributed systems in which the communication bandwidth is an important performance factor. For example, if the mining is being done on a distributed database over wide area or long haul network. The performance of FDM-LP against Apriori in a large database is also compared. In that case, the response time of FDM-LP is only about 20% longer than $1/n$ of the response time of Apriori, where n is the number of sites. This is a very ideal speed-up. In terms of total execution time, FDM-LP is very close to Apriori.

Parameter	Interpretation	Value
| T |	transaction mean size	10
| I |	mean size of maximal potentially large itemsets	4
| L |	number of potentially large itemsets	2000
N	Number of items	1000
S_q	Clustering size	5 - 6
P_s	Pool size	50 - 70
c_r	Correlation level	0.5
M_f	Multiplying factor	1260 - 2400

Table 5: Parameter Table.

The test bed that we use has six workstations. Each one of them has its own local disk, and its partition is loaded on its local disk before the experiment starts. The databases used in our experiment are synthetic data generated using the same techniques introduced in [2, 10]. The parameters used are similar to those in [10]. Table 5 is a list of the parameters and their values used in our synthetic databases. Readers not familiar with these parameters can refer to [2, 10]. In the following, we use the notation Tx.Iy.Dm to denote a database in which $D = m$ (in thousands), $|T| = x$, and $|I| = y$.

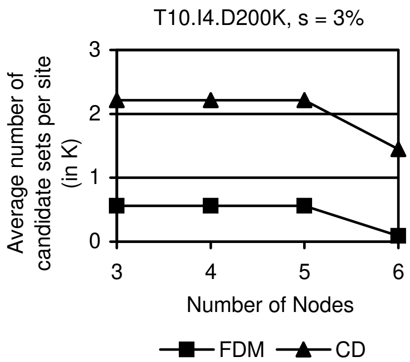

Figure 1: Candidate Sets Reduction (n = 3, 4, 5, 6)

5.1 Candidate Sets and Message Size Reduction

The sizes of the databases in our study range from 200K to 600K transactions, and the minimum support threshold ranges from 3% to 3.75%. Note that the number of candidate sets at each site are the same in CD and different in FDM. In our experiment, we witnessed a reduction of $75-90\%$ of candidate sets on

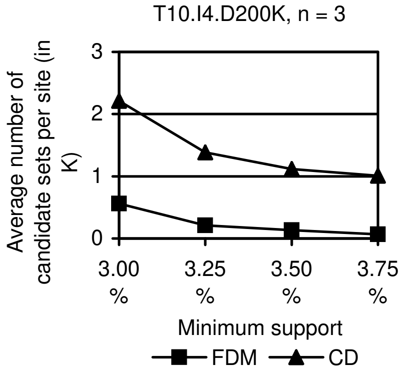

Figure 2: Candidate Sets Reduction

average at each site when FDM-LP is compared with CD. In Figure 1, the average number of candidate sets generated by FDM-LP and CD for a 200K transaction database are plotted against the number of partitions. FDM-LP has a $75 - 90\%$ reduction in the candidate sets. The percentage of reduction increases when the number of partitions increases. This shows that FDM becomes more effective when the system is scaled up. In Figure 2, the same comparison between FDM-LP and CD is presented for the same database with three partitions on different thresholds. In this case, FDM-LP experienced a similar amount of reduction.

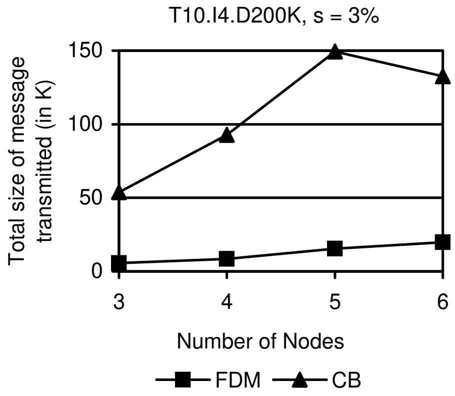

Figure 3: Message Size Reduction (n = 3, 4, 5, 6)

The reduction in candidate sets should have a proportional impact on the reduction of messages in the comparison. Moreover, as discussed before, the polling site technique guarantees that FDM only requires $O(n)$ messages for each candidate set, which is much smaller than the $O(n^2)$ messages required in CD. In our experiment, FDM has about 90% reduction in the total message size in all cases when it is compared with CD. In Figure 3, the total message size in FDM and CD for the same 200K database are plotted against the number of partitions. In Figure 4, the same comparison on the same database of three partitions with different support thresholds are presented. Both results confirm our analysis that FDM-LP is very effective in cutting down the number of messages required.

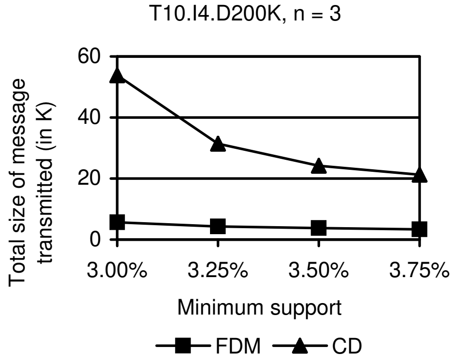

Figure 4: Message Size Reduction

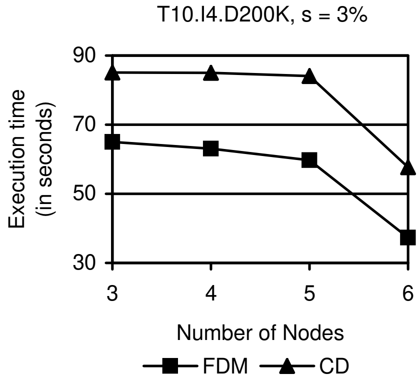

Figure 5: Execution Time (n = 3, 4, 5, 6)

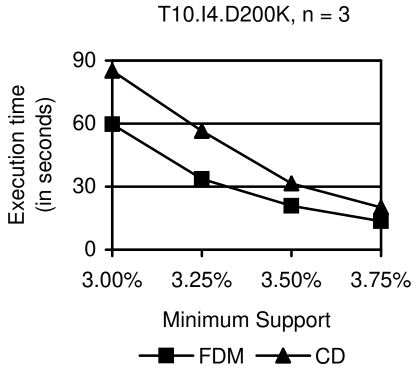

Figure 6: Execution Time

5.2 Execution Time Reduction

We have also compared the execution time between FDM-LP and CD. The execution time of FDM-LP and CD on a 200K database are plotted against the number of partitions in Figure 5. FDM-LP is about

$25-35\%$ faster than CD in all cases. In Figure 6, the comparison is plotted against different thresholds for the same database on three partitions. Again, FDM-LP is shown to have similar amount of speed-up as in Figure 5.

$n = 3$, $D = 600K$, $s = 2\%$	Apriori	FDM-LP
response time (sec)	1474	387
total execution time (sec)	844.7	842.9

Table 6: Efficiency of FDM-LP.

We have also compared FDM-LP on three sites against Apriori with respect to a 600K transactions database in order to find out its efficiency in large database. The result is shown in Table 6. The response time of FDM-LP is only slightly (20%) larger than 1/3 of that of Apriori. In terms of the total execution time, FDM-LP is very close to Apriori. For a large database, FDM-LP may have a bigger portion of the database residing in the distributed memory than Apriori. Therefore, it will be much faster than running Apriori on the same database in a single machine. This shows that FDM-LP on a scalable distributed system is an efficient and effective technique for mining association rules in large databases.

The performance study has demonstrated that FDM generates a much smaller set of candidate sets and requires a significantly smaller amount of messages when comparing with CD. The improvement in execution time is also substantial even though the overhead incurred from PVM prevents FDM from achieving a speed-up proportional to the reduction in candidate sets and message size. Even though, we have only compared CD with FDM-LP, there is enough evidence to show that FDM is more efficient than CD in a distributed environment. In the following sections, we will discuss our future plan of implementing the other versions of FDM.

6 Discussions

In this discussion, we will first discuss the issue of possible extension of FDM for fast parallel mining of association rules. Following that, we will discuss two other related issues: (1) the relationship between the effectiveness of FDM and the distribution of data, and (2) support threshold relaxation for possible reduction of message overhead.

The CD and PDM algorithms are designed for share-nothing parallel environment. In particular, CD has been implemented and tested on the IBM SP2 machine. In designing algorithm for parallel mining of association rules, not only the number and size of messages required should be minimized, but also the number of synchronizations, which is the number of rounds of message communication. CD has a simple synchronization scheme. It requires only one round of message communication in every iteration. Besides the second iteration, PDM also has the same synchronization scheme as CD. If FDM was used in the parallel environment, it has a shortcoming: even though it requires much less message passings then CD, it needs more synchronizations. However, FDM can be modified to overcome this problem. In fact, in each iteration, the candidate set reduction and global pruning techniques can be used to eliminate many candidates and then a broadcast can be used to exchange the local support counts of the remaining candidates. This approach will generate less candidate sets than CD and has the same number of synchronization. Therefore, it will perform better than CD in all cases. Performance studies has been carried out in a 32-nodes IBM SP2 to study several variations of this approach, and the result is very promising.

Another interesting issue is the relationship between the performance of FDM and the distribution of the itemsets among the partitions. From both Theorem 1 and Example 1, it is clear that the number of candidate sets decreases dramatically if the distribution of itemsets is quite skewed among the partitions. If most of the globally large itemsets were locally large at most of the sites, the reduction of candidate sets in FDM would not have been as significant. In the worst case, if every globally large itemset is locally large at all the sites, the candidate sets in FDM and CD will be the same. Therefore, data skewness may improve the performance of FDM in general. Special partitioning technique can be used to increase the data skewness to optimize the performance of FDM. Some further study is required to explore this issue.

The last issue that we want to discuss is the possible usage of the relaxation factor proposed in [11]. In FDM, if a site sends not only those candidate sets which are locally large but also those that are almost locally large to the polling sites, the polling sites may have local support counts from more sites to perform the *global pruning of candidate sets*. For example, if the support threshold is 10%, every site can send the candidate sets whose local support counts exceed 5% to their polling sites. In this case, for some candidate sets, their polling sites may receive local support counts from more sites than the no relaxation case. Hence, the global pruning may be more effective. However, there is a trade-off between sending more candidate sets to the polling sites and the pruning of candidate sets at the polling sites. More study is necessary on the detailed relationship between the relaxation factor and the performance of the pruning.

7 Conclusions

In this paper, we proposed and studied an efficient and effective distributed algorithm FDM for mining association rules. Some interesting properties between locally and globally large itemsets are observed, which leads to an effective technique for the reduction of candidate sets in the discovery of large itemsets. Two powerful pruning techniques, local and global prunings, are proposed. Furthermore, the optimization of the communications among the participating sites is performed in FDM using the polling sites. Several variations of FDM using different combination of pruning techniques are described. A representative version, FDM-LP, is implemented and whose performance is compared with the CD algorithm in a distributed system. The result shows the high performance of FDM at mining association rules.

Several issues related to the extensions of the method are also discussed. The techniques of candidate set reduction and global pruning can be integrated with CD to perform mining in a parallel environment which will be better than CD when considering both message communication and synchronization. Further improvement of the performance of the FDM algorithm using the skewness of data distribution and the relaxation of support thresholds is also discussed.

Recently, there have been interesting studies on the mining of generalized association rules [14], multiple-level association rules [8], quantitative association rules [15], etc. Extension of our method to the mining of these kinds of rules in a distributed or parallel system are interesting issues for future research. Also, parallel and distributed data mining of other kinds of rules, such as characteristic rules [7], classification rules, clustering [9], etc. is an important direction for future studies. For our performance studies, an implementation of the different versions of FDM on an IBM SP2 system with 32 nodes has been carried out and the result is very promising.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. In *IBM Research Report*, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering*, pages 3–14, Taipei, Taiwan, March 1995.

[4] D.W. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 1996 Int'l Conf. on Data Engineering*, New Orleans, Louisiana, Feb. 1996.

[5] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[7] J. Han, Y. Cai, and N. Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. Knowledge and Data Engineering*, 5:29–40, 1993.

[8] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 420–431, Zurich, Switzerland, Sept. 1995.

[9] R. Ng and J. Han. Efficient and effective clustering method for spatial data mining. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 144–155, Santiago, Chile, September 1994.

[10] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *Proc. 1995 ACM-SIGMOD Int. Conf. Management of Data*, pages 175–186, San Jose, CA, May 1995.

[11] J.S. Park, M.S. Chen, and P.S. Yu. Efficient parallel mining for association rules. In *Proc. 4th Int. Conf. on Information and Knowledge Management*, pages 31–36, Baltimore, Maryland, Nov. 1995.

[12] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 432–443, Zurich, Switzerland, Sept. 1995.

[13] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database research: Achievements and opportunities into the 21st century. In *Report of an NSF Workshop on the Future of Database Systems Research*, May 1995.

[14] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 1995 Int. Conf. Very Large Data Bases*, pages 407–419, Zurich, Switzerland, Sept. 1995.

[15] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data*, Montreal, Canada, June 1996.