An Efficient Frequent Pattern Mining Method and its Parallelization in Transactional Databases

S. M. FAHKRAHMAD¹ AND GH. DASTGHAIYFARD²
¹Department of Computer Engineering
Islamic Azad University, Shiraz Branch
Shiraz, Iran
²Department of Computer Science and Engineering
Shiraz University
Shiraz, Iran
E-mail: {mfakhrahmad@cse.; dstghaib@}shirazu.ac.ir

One of the important and well-researched problems in data mining is mining association rules from transactional databases, where each transaction consists of a set of items. The main operation in this discovery process is computing the occurrence frequency of the interesting set of items. i.e., Association Rule mining algorithms search for the set of all subsets of items that frequently occur in many database transactions. In practice, we are usually faced with large data warehouses, which contain a large number of transactions and an exponentially large space of candidate itemsets, which have to be verified. A potential solution to the computation complexity is to parallelize the mining algorithm. In this paper, four parallel versions of a novel sequential mining algorithm for discovery of frequent itemsets are proposed. The parallelized solutions are compared analytically and experimentally, by considering some important factors, such as time complexity, communication rate, load balancing, etc.

Keywords: parallel processing, data mining, frequent itemsets, association rules, load balancing

1. INTRODUCTION

One of the important and attractive problems in data mining [1] is the discovery of Association Rules (ARs) from transactional databases, where each transaction contains a set of items. ARs are represented in the general form of \( X \rightarrow Y \) and imply a co-occurrence relation between \( X \) and \( Y \), where \( X \) and \( Y \) are two sets of items (called itemsets). \( X \) and \( Y \) are called antecedent (left-hand-side or LHS) and consequent (right-hand-side or RHS) of the rule, respectively.

Many evaluation measures are defined to select interesting rules from the set of all possible candidate rules. The mostly used measures for this purpose are minimum thresholds on support and confidence. The Support of an AR, \( X \rightarrow Y \), is the percentage of transactions that contain both \( X \) and \( Y \), simultaneously. This is the probability, \( P(A \cap B) \).

The Confidence of the rule is the percentage of transactions containing \( X \), which also contain \( Y \). This is equal to the conditional probability, \( P(Y | X) \).

Since transactional datasets usually contain a large number of distinct items, the whole number of candidate itemsets is very large. Thus, existing AR mining methods [2-5] try to prune the search space by determining a minimum support threshold for can-
didates under consideration.

A common approach is used by most of the existing algorithms (including the well-known Apriori algorithm) to find ARs. First, finding all frequent itemsets \( i.e., \) the itemsets which occur at least as frequently as a pre-determined minimum support threshold. Then, generating strong ARs (which satisfy both minimum support and minimum confidence thresholds) from the frequent itemsets. In other words, a typical AR mining algorithm finds the set of all subsets of items that frequently occur in many transactions, first, and then using this set, extracts rules on how a subset of items influences the presence of another subset.

The base of such algorithms is the fact that any subset of a frequent itemset must also be frequent, and that both the LHS and the RHS of a frequent rule must also be frequent. Thus, every frequent itemset of length \( n \) can result in \( n \) association rules with a single item on the RHS \([6, 7]\).

There are some approaches, which consider the memory usage as the main factor to be minimized \([8, 9]\). However, for huge datasets, which contain a large number of distinct items and a large number of transactions, an important factor that an AR mining algorithm is expected to have, is scalability, \( i.e., \) the ability to handle massive data stores. Sequential algorithms cannot provide scalability, in terms of the data dimension, size, or runtime performance, for large databases. A solution for improving the performance and providing scalability is parallel and distributed computing. Employing multi-processor systems, mining of frequent itemsets can be accomplished in a reasonable time. There are various metrics to evaluate parallel algorithms, including computational complexity, speedup, communication rate, load balancing, etc.

In this paper, first a sequential algorithm for mining ARs is proposed, which is based on bottom-up approach. The proposed algorithm is very suitable for sparse datasets (where the probability of a specific item in a transaction is low, due to the wide variety of items). It scans the database just once and stores data in a new format within a special data structure, in the main memory. When dealing with sparse datasets, this structure is so compressed that can fit into memory, even when the size of the original dataset is very large. In other words, this sequential algorithm supports scalability for sparse datasets. It is a key feature which is not supported by other sequential algorithms. However, for huge datasets, which have a dense nature, the algorithm may encounter with the lack of memory for holding the data structures. To give a solution for dense datasets, based on this sequential algorithm, four parallel algorithms are proposed, which are compared analytically and experimentally.

The rest of this paper is organized as follows. Section 2 provides an overview of the sequential and parallel algorithms for mining ARs. Section 3 is devoted to presenting the parallel versions of an efficient mining algorithm and an analytical comparison over them. Experimental results are shown in section 4. Finally, section 5 concludes the paper.

2. RELATED WORK

As AR mining is an important issue in the field of data mining, it has been well researched and several sequential algorithms have already been proposed for this purpose. However, there has been relatively less work in parallel mining of ARs. In [10] a number
of distributed data mining algorithms for collective data mining, clustering, and AR mining are introduced. [11] Gives an overview of some of the parallel AR mining methods. However, some newer parallel methods can be found in [12, 18]. Three different parallel versions of the Apriori method are presented in [19]. In all of these methods, the database is supposed to be distributed horizontally among the processors. The first method is called Count Distribution (CD), which is a straight-forward parallelization of Apriori. In CD method, each processor computes the partial support of all candidate itemsets from its local database partition. At the end of each iteration, the processors exchange their partial supports to measure the global supports. The second method is called Data Distribution (DD), which partitions the candidate itemsets into disjoint sets and assigns them to different processors. In DD method, each processor has to scan the entire database (not only its local partition) in all iterations, to measure the global support. Thus, DD method involves a high communication overhead. The Intelligent Data Distribution (IDD), the third method is similar to the second method, it partitions the candidates, but it selectively replicates the database, so that each processor proceeds independently. Among the three parallel versions of Apriori, the CD method is reported to perform the best. There are also some other parallel algorithms in the literature, which outperform the CD method. The Fast Distributed Mining (FDM) [20] and Distributed Mining Of Association rules (DMA) [21] algorithms generate fewer candidate itemsets and involve smaller message sizes compared to the CD method. In [22], Schuster and Wolff propose the Distributed Decision Miner (DDM) algorithm. They report that DDM has a better scalability than CD and FDM with respect to the minimum support threshold.

In [23], Wenbin et al. present two variants of Apriori for frequent itemset mining, namely PBI-GPU and TBI-GPU. These methods employ a bitmap data structure to encode the database. The first implementation (PBI-GPU) stores the itemsets in a bitmap structure. Each part of data is represented by an integer. In order to avoid counting the number of 1’s in similar integers, it constructs a lookup table that stores the mapping of an integer and the number of 1’s in its binary representation. The other one (TBI-GPU) utilizes a trie structure to store the itemsets and adopts a co-processing scheme.

One of the most well-known and efficient algorithms in mining frequent itemsets is FP-Growth [24]. This method acts completely different from others. It discovers frequent itemsets without generating any candidate itemset. In this algorithm, the data is read 3 times and then a structure of hash tree is built in memory. All frequent itemsets can be found by traversing the hash tree. The main problem of FP-Growth is its heavy and database size dependent utilization of main memory. Many researchers have already invested on developing parallelized variants of the method (such as [24, 26]). However, the problem has not been solved. Running the algorithm (even the parallelized versions) for very huge data sets is still impossible due to limitation of main memory. The reason of this issue is that the parallel versions of FP-Growth need to be run over a multi-processor system, in which all the processors use a shared memory. The algorithm can not be used in distributed environments where the data is distributed among a number of loosely coupled sites.

In [27], we had presented a new sequential AR mining algorithm called FastARM, which was shown to be scalable and efficient when dealing with sparse datasets. In order to support the scalability for dense datasets, four parallel versions of that algorithm will be proposed in detail, in section 3.
3. THE PARALLELIZED VERSIONS OF THE MINING ALGORITHM

In the previous section, we introduced a new algorithm for discovery of frequent itemsets in a transactional dataset. The structures used in this method are so that it has a very good performance when the database is sparse. When the dataset is sparse, the hash tables are very small, i.e., we have a high rate of data compression and will rarely run out of memory. Moreover, the small size of hash tables leads to the efficient measurement of the support values. In this case, the algorithm is supposed to be scalable. However, when the dataset is dense, the size of hash tables is not as compressed as it is for sparse datasets. Thus, in some cases we may be faced with lack of memory. In other words, the algorithm may not be scalable for dense datasets. To provide scalability, in this section we introduce four parallel versions of the algorithm. The parallel algorithms will be compared analytically and experimentally, with respect to some important factors, such as time complexity, communication rate, load balancing, etc.

3.1 First Method: Assigning Each Partition to a Processor

The first method has a work-pool approach. In this method, the database is supposed to be distributed horizontally among different processors. If not, we assume that the master processor distributes (using scatter) the database among the other processors. Each processor can be assigned one or more partitions of the data. For ease of illustration, let’s assume that each processor is given just one partition, as shown in an example dataset in Fig. 1. A slave processor is responsible for measuring the occurrence frequency of all items (columns) in its local partition. For each item, a processor generates two numbers. The first is a decimal number, which is the equivalent to the binary number of that item in the assigned partition (as described in section 3). The other number is the local support value of the item. The support values are returned to the master processor.

Fig. 1. Horizontally distributing data among the processors.
After the master processor receives the local support values related to a specific item from all slaves, it is just time to measure the global frequency of the item. However, it does not measure the global support for all items. The following principle is used in order to avoid useless measurements for items which are not likely to be frequent: *In a distributed dataset, an item can be globally frequent only if it is frequent in at least one of the local parts.* Thus, the master processor computes the overall support for an item if at least one of the support values (received from the slave processors) is greater than the MinSupp.

The responsibilities of the master and the slaves will change (as follows) when mining compound frequent itemsets. When the 1-frequent items are detected, the list of frequent items is broadcasted to all slaves, to enable them starting discovery of 2-frequent itemsets. For each pair of 1-frequent items (whose combination is a candidate 2-itemset), a slave processor performs the logical AND operation over the decimal numbers related to the two items (These decimal numbers had been generated during the previous stage). The local support value of the candidate 2-itemset is computed by counting the 1-bit frequency over the resulting number of AND operation. The local supports of each 2-itemset, measured by each slave processor are returned to the master processor. If at least one of the local support values of an itemset is greater than the MinSupp threshold, the master computes the global support value of the itemset.

When all 2-frequent itemsets are detected, the master broadcasts the list of them to all processors. To find 3-frequent itemsets, the slave processors should measure the local support values of all candidate 3-itemsets. Each candidate 3-itemset is the combination of a 1-frequent itemset and a 2-frequent itemset. Hence, the decimal number equivalent to a 3-itemset is resulted by performing AND operation over the decimal numbers of its elements (a singleton and a 2-itemset), which have been generated in the first and second stages, respectively. The global support value for a 3-itemset is similarly computed by the master processor.

In general, for finding \( n \)-frequent itemsets, the master processor broadcasts the list of all \((n-1)\)-frequent itemsets to all processors. The decimal number of an \( n \)-itemset (within a partition) is computed by performing AND operation over the decimal numbers of a singleton and an \((n-1)\)-frequent itemset. So, each slave processor has to hold just the locally computed decimal numbers of 1-frequent and \((n-1)\)-frequent itemsets, which had been generated by itself during the first and the last stages, respectively. The numbers generated during the intermediate stages are not required to be held.

It should be noted that in each iteration, the slave processors must operate synchronously, because the master requires the results from all of them before proceeding.

### 3.1.1 Discussion

The first method presents a load-balanced parallelism. In this method, all slave processors relatively do the same amount of work. The slaves are responsible for measuring the local support values of different itemsets, while the master processor presents the list of itemsets, which are globally frequent.

This method can be used by two approaches. The first approach aims at reducing the number of communications. In this approach, a slave processor first measures the local
support values for all items and then sends the results to the master through a single send command. On the other hand, the goal of the second approach is to minimize the idle times of the processors. In this approach, a slave processor measures the local support value of an itemset and sends the result to the master, immediately. So, the master does not have to remain idle until all the values are measured. However, the second approach involves a large number of communications, compared to the first approach. To present a reasonable solution, we can make a trade-off among the two mentioned approaches. For this purpose, slave processors should measure the local support values of a number of items (instead of one item) and each time send a list of support values to the master.

As mentioned in the previous sections, many sequential algorithms can not be used for dense datasets, since they generally run out of memory. The sequential method proposed in this paper, does not also have a good performance when applying to dense datasets. This challenge was the main reason we decided to present the parallelized versions of the algorithm. It will be shown through experiments that the first parallelized method properly overcomes the lack of memory and has the best performance for dense datasets.

3.2 Second Method: Assigning Each Column to a Processor

In this approach, the data is assumed to be distributed vertically among the processors (one or more columns for each processor). Let’s assume that each processor is responsible for one column. Unlike the previous method, in this case all the processors start their work simultaneously. Each processor constructs the hash table and meanwhile measures the support value (as discussed in section 2) for its assigned column. Having $n$ processors, the processor $P_i$ sends its constructed hash table to all of its subsequent processors (i.e., processors $P_{i+1}$ to $P_n$), if the measured support satisfies the MinSupp threshold. In the next step, each processor combines its own hash table with each of the hash tables received from its prior processors in order to detect 2-frequent itemsets. Similarly, the processors send the hash tables of the discovered 2-frequent-itemsets to their subsequent processors, to enable them detecting 3-frequent itemsets. This process continues until the maximal frequent itemset is found by one of the processors. As an example, consider four processors ($P_0$ to $P_3$) which are used to mine frequent itemsets in a dataset containing four distinct items (A, B, C and D). Fig. 2 monitors the load of each processor through the worst case of this parallel process, i.e., when all itemsets (all combinations of items) are found out as frequent. However, this situation rarely occurs in practice, unless the MinSupp threshold has a very low value (i.e., about zero). Each row in this Fig. 2 belongs to one processor and each cell represents an itemset whose support is measured by the processor.

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$B$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$C$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$D$</td>
</tr>
</tbody>
</table>

Fig. 2. Monitoring loads of the processors when each one is initially responsible for a distinct item.
3.2.1 Discussion

The second method has some advantages and some disadvantages in comparison with the first method. The main advantage of this method is that it needs much fewer communications among the processors. It has a significant effect on the speedup value of the parallelism, as will be shown through the experiments. The main disadvantage of this method is that the workloads of the processors are not balanced. In general, the amount of work a processor has to do is related to the sequence number of the processor. Thus, the processors of higher ranks usually have to do much more work than their counterparts. On the other hand, the processors having smaller numbers have very light workload. Another reason for imbalanced load is a problem called early stopping. This problem occurs when a processor finds out that the item assigned to it (for measuring the support value) is not frequent. Thus, it will not proceed on measuring the support of its combinations. In this situation, the processor stops its cooperation, while there may exist some other processors, which have a large amount of work to do.

3.2.2 Load balancing

As mentioned in the last subsection, the main shortcoming of the second method is that the work-loads of different processors are not balanced. The worst case occurs when the item assigned to the last processor ($P_n$) is the most frequent singleton. As shown in Fig. 2, the processor has to measure the support of this item in combination with all other frequent items. So, in this case, many combinations of this item are likely to be frequent. Thus, a high amount of work has to be performed by $P_n$. On the other hand, if the least frequent item is assigned to $P_n$, most combinations will be infrequent and $P_n$ will have a relatively balanced work-load. For ease of illustration, consider the example shown in Fig. 2. Assume that the attribute D is the most frequent item, which is assigned to the last processor ($P_3$). Since D is a frequent item, many of the combinations containing D (such as BD, ABD, ABCD, etc.) are likely to be frequent, and so, $P_3$ has a large amount of work compared to the other processors.

In order to overcome the discussed problem and to balance the workloads of the processors, we use a technique called index swapping. As the first step, each processor is responsible for measuring the support value of a single item. When the support values for all singletons are measured and before going on measuring the support values of combinations, we swap the indices of processors such that: $P_n$ is the processor which is responsible for the least frequent item, ... and $P_0$ is the processor which is responsible for the most frequent one.

3.3 Third Method: Devoting the $k$th Processor to Mining of $k$-itemsets

The third method introduces a pipeline approach. In this approach the $k$th processor, say $P_k$, is responsible for mining $k$-frequent itemsets. For example, the first processor ($P_1$) just finds frequent singletons, builds hash tables for them and sends the hash tables to the next processor. The hash tables built by each processor are immediately forwarded to the next processor. Each processor (say $P_k$), except the first one, receives hash tables from its
prior processor \((P_{k-1})\) and constructs new hash tables from their combinations (using AND operation). Fig. 3 shows the worst case of this pipeline process (when all itemsets have an acceptable support value) using 4 processors for a dataset containing 4 distinct items (A, B, C and D).

Fig. 3. Monitoring loads of the processors when the \(k\)th processor searches for \(k\)-itemsets.

3.3.1 Discussion

As the experimental results will show, the third method generally requires the least amount of communications among the three parallel methods. This is the main reason of the improved speedup value of this method compared to the second method. It also has another advantage in comparison with the second method. The third method is safe from the problem of early stopping which is a typical challenging problem of the second method, as discussed in the previous section. The main disadvantage of the third method is the imbalanced loads of the processors. Let \(n\) be the number of all distinct items. The number of potential candidate \(k\)-itemsets is obtained from \(C(k, n)\). As we know, the value of \(C(k, n)\) equals the value of \(C(n-k, k)\). In other words, by increasing \(k\) from 1 to \(n/2\), the value of \(C(k, n)\) also increases, while it decreases for values of \(k\) increasing from \(n/2\) to \(n\). That’s why in general, by increasing the rank of processors along the pipeline, the workload first increases and then begins decreasing. This general case can be seen in Fig. 3.

4. EXPERIMENTAL RESULTS

4.1 Evaluation of the Algorithm (FastARM) using Benchmark Data

Our first set of experiments involved real data sets extracted from the Frequent Itemset Mining Dataset Repository, namely BMS-POS, BMS-WebView-1 and BMS-WebView-2.

The BMS-POS dataset contains sale data of several years from a large electronics retailer. Since this retailer has so many different products, product categories are used as items. Each transaction in this dataset is a customer’s purchase transaction consisting of the entire product categories purchased at one time. The goal for this dataset is to find associations between product categories purchased by customers in a single visit to the retailer. This data set contains 515,597 transactions and 1,657 distinct items. The BMS-WebView-1 and BMS-WebView-2 datasets contain several months worth of click stream data from two e-commerce web sites. Each transaction in these data sets is a web session consisting of all the product detail pp. viewed in that session. That is, each product detail

<table>
<thead>
<tr>
<th>(P_1)</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_2)</td>
<td>AB</td>
<td>AC</td>
<td>BC</td>
<td>AD</td>
</tr>
<tr>
<td>(P_3)</td>
<td>ABC</td>
<td>ABD</td>
<td>ACD</td>
<td>BCD</td>
</tr>
<tr>
<td>(P_4)</td>
<td>ABCD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
view is an item. The goal for both of these datasets is to find associations between products viewed by visitors in a single visit to the web site. These two data sets contain 59,602 and 77,512 transactions respectively with 497 and 3,340 distinct items.

We implemented the algorithms in C++ on a 3GHz Intel system with 1 GB RAM and under Windows operating system. In all of the experiments we used $k = 32$ for the size of partitions.

We set the MinConf threshold value to zero and evaluated the performances using the MinSupp value varying within the range of (0.02%-0.1%). The results of these experiments are shown in Figs. 4 (a) and (b). We see in these graphs that for lower values of MinSupp, the performance of FastARM continues to be superior in comparison with the other methods.

![Graphs showing performance of algorithms](image)

(a) Running times of the methods on the BMS-POS data set. (b) Running times of the methods on the BMS-WebView-1 data set.

Fig. 4. Performance of algorithms over some real-life data sets.

### 4.2 Evaluation and Comparison of the Parallel Methods

In the second part, we conducted a set of experiments to evaluate the performances of the proposed parallel methods with respect to different factors, and also compare them with other parallel algorithms. Likewise the first part, the algorithms were implemented in C++ and run concurrently on 5 computers with 3GHz Intel processor and 1 GB RAM within a LAN network with Star topology. For communication, we used the message passing interface (MPI). Data used in different parts of our experiments were generated randomly, such that the probability of an item being present in a transaction is 0.002 (if we need a sparse dataset) or 0.2 (if we need a dense dataset). The size of each generated dataset in our experiment (which were initially saved on the disk) was about 10MB.

#### 4.2.1 Comparison in terms of the communication rate

Communications between the processors in order to transfer a piece of data are performed by some MPI functions such as MPI_Send, MPI_Recv, MPI_Scatter and MPI_
Gather. There is also another MPI function, named MPI_WTime, which can be used for measuring the response time of a set of instructions within the program code. We used this function in the code wherever a communication between the processors was to be performed. The total communication time was then computed by gathering the total communication times from all processors and summing them up by one of the processors.

For message passing, it is desirable to reduce the communication rate because of its time overhead. For networks of workstations (as in our experiment), this challenge is more important rather than in multiprocessor systems, since the communication latency is more significant in such environments.

In the first part of this experiment, we used a synthetic sparse dataset having 100k transactions and 100 distinct items. The dataset was generated randomly, such that the probability of an item being present in a transaction is 0.002. Using this dataset we compared the communication times of the proposed parallel mining methods, proposed in this paper. In the next step, we repeated the experiment using a dense dataset of the same number of items and transactions. This dataset was also generated randomly. Each item was generated by the probability of 0.2 to be present in transactions. Figs. 5 and 6 present the results of the two parts of this experiment, for different values of MinSupp.

Comparing Figs. 5 and 6, we see a high gap between the relative efficiency of the first method in two cases. As shown in section 3, the first method is the only method (among the three) in which the dataset is distributed horizontally. In this method, a processor has to return the decimal numbers computed from its local data, even if they are all zero. In other words, it does not efficiently make use of the hash tables. Thus, it performs approximately similar on dense and sparse data. On the other hand, the other two methods, which distribute the dataset vertically, use hash tables just similar to the sequential algorithm and throw away any zero value resulting in any stage. When the dataset is dense, the probability of emerging zero decimal numbers in a partition decreases. So, the hash tables lose their interesting feature of data compression, since they contain too many entries. When a hash table which is going to be sent is too large, the processor will have to split and send it through more than one transfer. That’s why the second and the third methods are sensitive to the nature of the dataset, whether it is dense or sparse.

![Fig. 5. Communication times of the parallel mining methods, for different values of MinSupp, using a sparse dataset](image)
A PROPOSED MINING METHOD AND ITS PARALLELIZED VERSIONS

4.3 Comparison in Terms of the Speedup Factor

Speedup is a measure of relative performance between a multiprocessor (or multi-computer) system and a single processor system, which is defined in Eq. (1).

\[
\text{Speedup} = \frac{t_s}{t_p}, \tag{1}
\]

where \( t_s \) and \( t_p \) are the execution times on a single processor and a multiprocessor, respectively.

In order to compare the speedup factors of the methods in different cases, in this experiment, we generated some dense datasets, each containing a different number of items. For each case, we first executed the sequential algorithm and measured the response time. Then using 5 processors, each of the parallel algorithms were run and the execution times were measured. The speedup factor of each parallel method was then calculated using Eq. (1). Fig. 7 gives a comparison among the methods in terms of the speedup factor.

It can be seen in Fig. 7 that for datasets having not too many items (attributes), the first parallel method provides the best speedup value. The most likely reason is the opti-
mal load balancing it supports. Another reason is that this method is not as sensitive to dense datasets as the others are. On the other hand, when the dataset has a large no. of items, the load balanced version of the second method outperforms the first method. This is probably due to the increasing of the no. of communications in the first method, as the number of items increases.

4.4 Comparing the Proposed Methods with Other Existing Methods

The goal of this experiment is to assess how efficient the proposed methods are, in comparison with other well-known parallel algorithms. As mentioned in section 2, three different parallel versions of the Apriori method have been proposed, namely Count Distribution, Data Distribution and Intelligent Data Distribution. It was also mentioned that recently, two other methods have been proposed in [23], namely PBI-GPU and TBI-GPU, which improve the process of frequent itemset mining through minimizing the total time of counting items. The metric we selected in this experiment for evaluation of our proposed methods in comparison with the existing algorithms is the response time of the algorithms to find all frequent itemsets. For this purpose, first, we randomly generated a dense dataset, containing 100K transactions and using different values of MinSupp, the response times of the methods were measured. The result showing the relative efficiencies of the methods in terms of the response time are presented in Fig. 8.

![Fig. 8. A comparison between the response times of the proposed methods and three parallel versions of Apriori.](image)

In the second part of this experiment, in order to compare the methods over benchmark data, we employed two dense datasets, namely Kosarak and Chess. Chess is one of the UCI-ML repository datasets and has been prepared by Roberto Bayardo. It consists of 3196 transactions and 75 distinct items. Kosarak is a larger dataset containing 990002 transactions and 41270 items. It has been provided by Ferenc Bodon and contains (anonymized) click-stream data of a Hungarian on-line news portal. The MinSupp value for this experiment was set to 60%. Among the parallel methods of the paper, M1 showed to be the most efficient one. Comparing M1 to the methods proposed in the literature, PBI-GPU and TBI-GPU showed to relatively have the nearest performance. However, for Kosarak dataset, TBI-GPU was more efficient than M1. The results can be seen in Fig. 9.
5. CONCLUSION

In this paper, first, we introduced a sequential mining algorithm for mining of frequent itemsets, which requires just a single scan of the database. Then, we presented four parallel versions of the algorithm. The parallel algorithms were compared analytically and experimentally, with respect to some factors, such as communication rate, response time, computation/communication ratio and load balancing. We showed that each of the proposed methods has some advantages and of course a number of disadvantages. For sparse datasets, the load balanced version of the second method seemed to be more efficient than the others. However, when the database is dense, it was illustrated why the first method is the best to be used as a parallel mining algorithm, especially where the no. of items is not too large.

We also compared the proposed methods with some of the existing parallel mining methods, namely Count Distribution, Data Distribution, Intelligent Data Distribution, PBI-GPU and TBI-GPU. One reason for the proposed methods compared to the existing methods is the high efficiency of the main sequential algorithm, which is the basis of the parallel methods. The second reason is the special techniques and tricks we used in parallel versions, such as avoidance of computing the global support in the first method (in some cases where not necessary), index swapping in the second method to provide load balancing, etc.

REFERENCES

4. A. Savasere, E. Omiecinski, and S. Navathe, “An efficient algorithm for mining as-
21. D. Cheung, V. Ng, A. Fu, and Y. Fu, “Efficient mining of association rules in distributed databases,” IEEE Transactions on Knowledge and Data Engineering, Vol. 8,
A PROPOSED MINING METHOD AND ITS PARALLELIZED VERSIONS


Seyed Mostafa Fakhrahmad received his B.S. in Computer Engineering (Software) from Teacher Training University of Tehran, Iran in Sep. 2003, and his M.S. in Computer Engineering (Artificial Intelligence) from Shiraz University in Nov. 2006. He is currently a Ph.D. student in Computer Engineering (Software) in Department of Computer Science and Engineering, School of Engineering, Shiraz University, Shiraz, Iran. His current research interests include data mining, fuzzy systems and pattern recognition.

Gh. Dastghaibyfard received his M.S. and Ph.D. in Computer Science from Electrical Engineering and Computer Science Department, College of Engineering, University of Oklahoma, Norman Oklahoma U.S.A. in 1979 and 1990, respectively. He is currently an Assistant Professor of Computer Science in Department of Computer Science and Engineering, School of Engineering, Shiraz University, Shiraz, Iran. His current research interests include parallel algorithms, grid computing and information technology.