Parallel Mining of Frequent Patterns in Transactional Databases

S.M. Fakhrahmad, G.H. Dastghaibi Fard

***Abstract*—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. In practice, we are usually faced with large datasets, and an exponentially large space of candidate itemsets. A potential solution to the computation complexity is to parallelize the mining algorithm. In this paper, firstly, we introduce an already proposed sequential mining algorithm for discovery of frequent itemsets, which requires just a single scan of the database. In the next part, we present four parallel versions of the algorithm. The parallel algorithms will be compared analytically and experimentally, regarding some important factors, such as time complexity, communication rate, load balancing, etc.**

***Index Terms*—Parallel Processing, Data Mining, Frequent Itemsets, Association Rules, Load balancing**

I. 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 → 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 → 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, $P(Y|X)$.

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, we first present a sequential algorithm for mining ARs, which is based on bottom-up approach. The 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, we also present four parallel versions of the algorithm and give an illustrating comparison on them.

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 describes the proposed sequential algorithm. Section 4 is devoted to presenting the parallel versions of the proposed algorithm and an analytical comparison over them. Experimental results are shown in Section 5. Finally, Section 6 concludes the paper.

II. 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 [3] a number of distributed data mining algorithms for collective data mining, clustering, and AR mining are introduced. [4] gives an overview of some of the parallel AR mining methods. Three different parallel versions of the Apriori method are presented in [5]. In all of these methods, the database is supposed to be distributed horizontally among the processors.

The first method is named Count Distribution (CD) algorithm, which is a straight- forward parallelization of Apriori. In this method, each processor computes the partial support of all candidate itemsets from its local database partition. At the end of each iteration, the

S.M. Fakhrahmad is with the Department of Computer Engineering, School of Engineering, Islamic Azad University of Shiraz (and PhD student in Shiraz University), Shiraz, Iran
(e-mail: mfakhahmad@cse.shirazu.ac.ir)
G.H. Dastghaibi Fard is with the Department of Computer Science & Engineering, School of Engineering, Shiraz University, Shiraz, Iran (e-mail: dstghaib@shirazu.ac.ir)

processors exchange their partial supports to measure the global supports.

The second is Data Distribution (DD) algorithm. It partitions the candidate itemsets into disjoint sets and assign them to different processors. In this algorithm, each processor has to scan the entire database (not only its local partition) in all iterations, to measure the global support. Thus, the algorithm involves a high communication overhead.

The Intelligent Data Distribution (IDD) algorithm is the third parallel version of Apriori. Similar to the second version, it partitions the candidates, but it selectively replicates the database, so that each processor proceeds independently. Among the three parallel versions of Apriori, the COUNT DISTRIBUTION method is reported to perform the best.

There are also some other parallel algorithms in the literature, which outperform the Count Distribution algorithm. The FDM (FAST DISTRIBUTED MINING) [6] and DMA (DISTRIBUTED MINING OF ASSOCIATION RULES) [7] algorithms generate fewer candidate itemsets and involve smaller message sizes compared to the COUNT DISTRIBUTION algorithm. In [8], Schuster and Wolff propose the DDM (DISTRIBUTED DECISION MINER) algorithm. They report that DDM has a better scalability than COUNT DISTRIBUTION and FDM with respect to the minimum support threshold.

In [9], a new sequential AR mining algorithm called FastARM has been proposed, which is shown to be scalable and efficient when dealing with sparse datasets. In order to support the scalability for dense datasets as well, we developed four parallel versions of the algorithm, which will be discussed in detail, in Section 4. Before introduction of the parallelized methods, the sequential algorithm will be presented in the next section.

III. The Sequential Algorithm

For ease of illustration, we assume the transaction database as a binary-valued dataset having a relational scheme. Each column in this scheme stands for a possible item that can be found in any transaction of the data warehouse and each tuple represents a transaction. Each 0 or 1 value indicates the presence or absence of an itemset in a transaction, respectively. As an example the relation shown in Fig. 1.(b) is the structured form of the dataset of Fig. 1.(a), which contains four transactions.

Cheese, Coke, Egg
Cheese, Egg
Coke, Cheese, Beer
Coke, Beer

a) A transactional dataset

Cheese	*Coke*	*Egg*	*Beer*
1	1	1	0
1	0	1	0
1	1	0	1
0	1	0	1

b) A structured presentation of transactions

Fig. 1. A market basket dataset

As the first step of the algorithm, we divide the relation horizontally into some equi-size partitions, each containing *k* tuples. We comment on choosing the best value for *k* latter in this section. In this relation, each column contains *k* bits in each partition, thus the group of bits in each partition can be vowed as a *k*-bit binary code, which is equivalent to a decimal number between 0 and 2^k-1. These decimal numbers are the major elements of our algorithm.

The partitioned relation is scanned just once and the supports of singletons (1-itemsets) are measured to find 1-frequent itemsets. Meanwhile, for each partition, all nonzero decimal values are extracted. For any column of the dataset, which represents a frequent singleton, we build a hash table in memory. Each value in this hash table, is a non-zero decimal value extracted from a partition and its access key is the number of that partition (an integer number between 1 and *m*, where *m* is the number of partitions). Since we do not insert zero values into the hash tables, then the values recorded in the hash table indicate the regions of the itemset occurrences and limits the search space for the next steps.

The support of a compound itemset such as AB, is easily measured by using the hash tables of its elements (i.e., A and B), instead of scanning the whole database again. In order to calculate the support of a compound itemset, we begin with the smaller hash table (i.e., the one having fewer values). For each key of this hash table, we first verify if it also exists in the other hash table. This verification does not involve any search due to the direct access structure of hash table. If a key exists in both hash tables, then we perform a logical *AND* operation between the the corresponding values related to that key.

The result of the *AND* operation is another integer value, which gives the co-occurrences of A and B in that partition. If the result is zero, it means that there is no simultaneous occurrence of A and B in that partition. We build a similar hash table for the compound itemset, AB, and insert the non-zero integer values resulted from *AND* operations in this table. The size of this hash table is at most equal to the size of the smaller hash table of the two elements. Each number stored in this hash table is equivalent to a binary number, which contains some 1's. The total number of 1's indicates the co-occurrence frequency of A and B. Thus we should just enumerate the total number of 1's for all integer values, instead of scanning the whole database. This measurement can be done using logical Shift Left (*SHL*) or Shift Right (*SHR*) operations over each value and adding up the carry bits until the result is zero (i.e., there is no other 1-bits to be counted).

The efficiency of this structure becomes clearer for measuring the support of higher dimensional itemsets. As we proceed to higher dimensional itemsets, the size of hash tables becomes smaller due to new zeros emerging from *AND* operations. These zeros are not inserted into the result hash table.

To measure the support of an itemset, such as *AB*, the number of 1's in the value field of this hash table (in the binary form) has to be counted. If this value is equal to 1 (i.e., 0001), just one *SHR* operation and thus one comparison is enough to count 1's. However, if we had searched all the data to find the co-occurrences of A and B, the number of required comparisons would have been

48 (for reading the value of A and B in all 24 tuples). In general, this improvement is much more apparent for itemsets of higher dimensions.

In a similar way, the hash tables of 2-frequent itemsets are then used to mine 3-frequent itemsets and in general, n-frequent itemsets are mined using (n-1)-frequent itemsets. However, we do not use all combinations of frequent itemsets to get (n+1)-frequent itemsets. The Apriori principle [10] is used to avoid verifying useless combinations: "An n-dimensional itemset can be frequent if all of its (n-1)-dimensional subsets are frequent". Thus, for example if AB and AC are two frequent itemsets, their combination is ABC, but we do not combine their hash tables unless the itemset BC is also frequent. If all the n-1 subsets of an n-dimensional itemset are frequent, combining two of them is enough to get the hash table of the itemset.

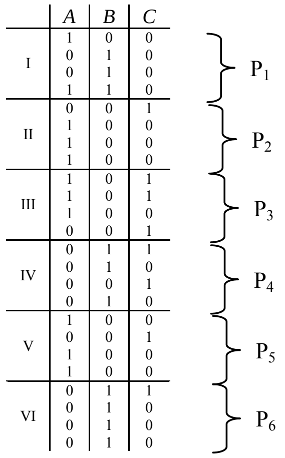
IV. The Parallelized Versions of the Mining Algorithm

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.

A. 1st 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 Fig. 2. 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.

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.



	A	*B*	*C*	
I	1	0	0	P_1
	0	1	0	
	0	1	0	
	1	1	0	
II	0	0	1	P_2
	1	0	0	
	1	0	0	
	1	0	0	
III	1	0	1	P_3
	1	0	1	
	1	0	0	
	0	0	1	
IV	0	1	1	P_4
	0	1	0	
	0	0	1	
	0	1	0	
V	1	0	0	P_5
	0	0	1	
	1	0	0	
	1	0	0	
VI	0	1	1	P_6
	0	1	0	
	0	1	0	
	0	1	0	

Fig. 2. Horizontally distributing data among the processors

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.

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 sends 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.

B. 2nd 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 3) 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. 3 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. 4 belongs to one processor and each cell represents an itemset whose support is measured by the processor.

P_0	*A*							
P_1	*B*	*AB*						
P_2	*C*	*AC*	*BC*	*ABC*				
P_3	*D*	*AD*	*BD*	*CD*	*ABD*	*ACD*	*BCD*	*ABCD*

Fig. 3. Monitoring loads of the processors when each one is initially responsible for a distinct item

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 too 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 work-load. 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 co-operation, while there may exist some other processors, which have a large amount of work to do.

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. 4, 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 Pn, 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. 4. Assume that the attribute D is the most frequent item, which is assigned to the last processor (P3). 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, P3 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.

C. 3rd method: Using processor Pk for 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. 5 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).

P_1	*A*	*B*	*C*	*D*				
P_2			*AB*	*AC*	*BC*	*AD*	*BD*	*CD*
P_3				*ABC*	*ABD*	*ACD*	*BCD*	
P_4					*ABCD*			

Fig. 4. Monitoring loads of the processors when the k-th processor searches for k-itemsets

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. 4.

V. Experimental Results

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. The algorithms were implemented in C++ and run concurrently on 5 systems with 3GHz Intel processor and 1 GB RAM. 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).

A. 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. Figures 5 and 6 present the results of the two parts of this experiment, for different values of MinSupp.

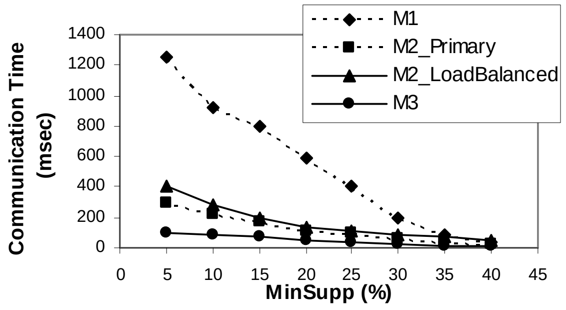


Fig. 5. Communication times of the parallel mining methods, for different values of MinSupp, using a sparse dataset

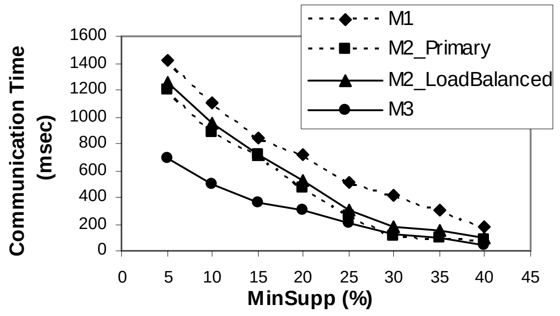


Fig. 6. Communication times of the parallel mining methods, for different values of MinSupp, using a dense dataset

Comparing Figures 6 and 7, we see a high gap between the relative efficiency of the first method in two cases. As shown in Section 4, 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.

B. *Comparison in terms of the Speedup factor*

Speedup is a measure of relative performance between a multiprocessor system and a single processor system, which is defined in equation (1).

$$\text{Speedup} = t_s/t_p\,, \tag{1}$$

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 equation (1). Fig. 7 presents a comparison between the methods in terms of the speedup factor.

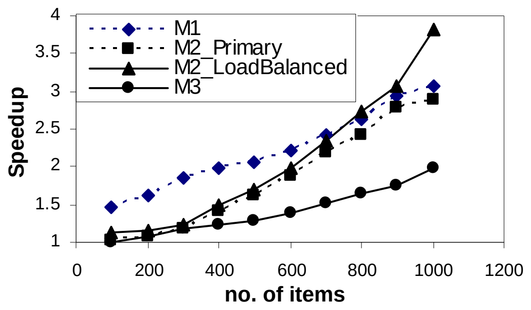


Fig. 7. Speedup factors of the parallel mining methods, for different numbers of items, using a dense dataset

It can be seen in Fig. 8 that for datasets having not too many items (attributes), the first parallel method provides the best speedup value. The most likely reason is the optimal 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.

VI. 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.

References

[1] M. Stonebraker, R. Agrawal, U. Dayal, E. J. Neuhold, and A. Reuter. DBMS research at a crossroads: The vienna update. In *Proc. of the 19th VLDB Conference*, pages 688–692, Dublin, Ireland, 1993.

[2] Xiao, Y. and Dunham, M. H., Considering main memory in mining association rules. In Proc. of Intl. Conf. on Data Warehousing and Knowledge Discovery (DAWAK), 1999.

[3] Byung-Hoon Park and Hillol Kargupta. Distributed data mining: Algorithms, systems, and applications. In Nong Ye, editor, *Data Mining Handbook*, 2002.

[4] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14.25, December 1999.

[5] R. Agrawal and J. Shafer. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engg.*, volume 8, pages 962.969, 1996.

[6] D. Cheung, J. Han, V. Ng, A. Fu, , and Y. Fu. A fast distributed algorithm for mining association rules. In *4... Int'l. Conf. Parallel and Distributed Info. Systems*, 1996.

[7] D. Cheung, V. Ng, A. Fu, , and Y. Fu. Ef_cient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engg.*, volume 8, pages 911.922, 1996.

[8] A. Schuster and R. Wolff. Communication ef_cient distributed mining of association rules. In *ACM SIGMOD Int'l. Conf. on Management of Data*, 2001.

[9] S.M. Fakhrahmad, M.H. Sadreddini and M. Zolghadri Jahromi, Mining Frequent Itemsets in Large Data Warehouses: A Novel Approach Proposed for Sparse Datasets. In Proc. Of: IDEAL 2007, pp. 517-526, 16-19 Dec 2007, Birmingham, UK.

[10] Agrawal, R., Srikant, R., Fast Algorithms for Mining Association Rules in Large Databases, In Proc. of the 20th International Conference on Very Large Data Bases, pp. 487-499, 1994

[11] Webb., G.I., Efficient search for association rules. In Proceedings of the Sixth ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY: ACM, 99-107.

[12] Zaki, M.J., Generating non-redundant association rules. In Proceedings of the Sixth ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY: ACM, 34-43.

[13] Pei, J., Han, J. and Yin, Y., Mining frequent patterns without candidate generation. In Proceedings of ACM-SIGMOD International Conference on Management of Data