Tech. Report TR01-12

Fast Parallel Association Rule Mining Without Candidacy Generation[⋆]

Osmar R. Zaïane Mohammad El-Hajj Paul Lu

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada.
{zaiane, mohammad, paullu}@cs.ualberta.ca

Abstract

Searching for frequent patterns in transactional databases is considered one of the most important data mining problems. Most current association mining algorithms, whether sequential or parallel, adopt an apriori-like algorithm that requires full multiple I/O scans of the data set and expensive computation to generate the potential frequent items. The recent explosive growth in data collection made the current association rule mining algorithms restricted and inadequate to analyze excessively large transaction sets due to the above mentioned limitations. In this paper we introduce a new parallel algorithm MLFPT (Multiple Local Frequent Pattern Tree) for parallel mining of frequent patterns, based on FP-growth mining, that uses only two full I/O scans of the database, eliminating the need for generating the candidate items, and distributing the work fairly among processors to achieve near optimum load balance. We have devised partitioning strategies at different stages of the mining process to achieve near optimal balancing between processors. This algorithm has been experimented on databases made of hundreds of thousands of dimensions and size larger than 10 Giga bytes using 64 processors SGI origin shared memory machine. We have successfully tested our algorithm on datasets larger than 50 million transactions. This paper presents the results of our algorithm on different data sizes experimented on different numbers of processors, and studies the effect of these variations on the overall performance.

[⋆]Research is supported in part by NSERC (the Natural Sciences and Engineering Research Council) of Canada, TeleLearning-NCE (Canadian Networks of Centres of Excellence) and the Canadian Foundation for Innovation.

1 Introduction

Recent days have witnessed an explosive growth in generating data in all fields of science, business, military, etc. The same rate of growth in the processing power of evaluating and analyzing the data did not accompany this massive growth. Due to this phenomenon, a tremendous volume of data is still retained without being studied. Data mining, a research field that tries to ease this problem, proposes some solutions for the extraction of significant and potentially useful patterns from these large collections of data. One of the canonical tasks in data mining is the discovery of association rules. Discovering association rules, considered one of the most important tasks, has been the focus of many studies in the last few years. Many solutions have been proposed using a sequential or parallel paradigm. However, the existing algorithms depend heavily on massive computation and repeated I/O scans for the data sets. Association rule mining algorithms currently proposed in the literature are not sufficient for extremely large datasets and new solutions still have to be found. In particular there is a need for algorithms that do not depend on high computation and repeated I/O scans. Parallelizing existing or new algorithms for finding the association rules seems a viable solution and certainly crucial for large-scale data mining. However, distributing and balancing the mining tasks between the processors without jeopardizing the global solution is not trivial. This is precisely our focus in implementing the MLFPT algorithm presented herein.

1.1 Problem Statement

The problem of mining association rules over market basket analysis was introduced in [1]. Association rules are not limited to market basket analysis, but the analysis of sales or what is known as basket data, is the typical application often used for illustration. The problem consists of finding associations between items or itemsets in transactional data. The data could be retail sales in the form of customer transactions or even medical images [23]. Association rules have been shown to be useful for other applications such as recommender systems, diagnosis, decision support, telecommunication, etc.

Formally, as defined in [3], the problem is stated as follows: Let $\mathcal{I} = \{i_1, i_2, ...i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. A unique identifier TID is given to each transaction. A transaction T is said to contain X, a set of items in $\mathcal{I}$, if $X \subseteq T$. An *association rule* is an implication of the form "$X \Rightarrow Y$", where $X \subseteq \mathcal{I}$, $Y \subseteq \mathcal{I}$, and $X \cap Y = \emptyset$. An itemset X is said to be *large* or *frequent* if its support s is greater or equal than a given minimum support threshold σ. The rule $X \Rightarrow Y$ has a *support* s in the transaction set $\mathcal{D}$ if $s\%$ of the transactions in $\mathcal{D}$ contain $X \cup Y$. In other words, the support of the rule is the probability that X and Y hold together among all the possible presented cases. It is said that the rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. In other words, the confidence of the rule is the conditional probability that the consequent Y is true under the condition of the antecedent X. The problem of discovering all association rules from a set of transactions $\mathcal{D}$ consists of generating the rules that have a *support* and *confidence* greater than given thresholds. These rules are called *strong rules.*

This association-mining task can be broken into two steps:

1. A step for finding all frequent k-itemsets known for its associate extreme I/O scans expense, and massive computational costs.

2. A straightforward step for generating confident rules from the frequent intemsets.

1.2 Related Work

Sequential algorithms: Several algorithms have been proposed in the literature [14, 10, 22, 9, 5, 15, 16, 18] to address the problem of mining association rules. One of the key algorithms, which seems to be the most popular in many applications for enumerating frequent itemsets, is the *apriori* algorithm [3]. This *apriori* algorithm also forms the foundation of most known algorithms whether sequential or parallel. It uses a monotone property stating that for a k-itemset to be frequent, all its k-1-itemsets have to be frequent. The use of this fundamental property reduces the computational cost of candidate frequent itemsets generation. However, in cases of extremely large input sets with outsized frequent 1-items set, the *apriori* algorithm still suffers from the main two problems of repeated I/O scanning and high computational cost. One major hurdle observed with most real datasets is the sheer size of the candidate frequent 2-itemsets and 3-itemsets.

Park et al. have proposed the Dynamic Hashing and Pruning algorithm (DHP) [19]. This algorithm is also based on the monotone *apriori* property, where a hash table is built for the purpose of reducing the candidate space by pre-computing the proximate support for the k+1 item set while counting the k-itemset. DHP has another important advantage, the transaction trimming, which has been applied by removing the transactions that do not contain any frequent items. However, this trimming and the pruning properties caused some problems that made it impractical in many cases [24].

The partitioning algorithm proposed in [6] reduced the I/O cost dramatically. However, this method has problems in cases of high dimensional itemsets, and it also suffers from the high false positives of frequent items. The Dynamic Item set Counting (DIC) reduces the number of I/O passes by counting the candidates of multiple lengths in the same pass. DIC performs well in cases of homogenous data, while in other cases DIC might scan the databases more often than the *apriori* algorithm.

Another innovative approach of discovering frequent patterns in transactional databases, FP-growth, was recently proposed by Han et al. [11]. This algorithm creates a relatively compact tree-structure that alleviates the multi-scan problem and improves the candidate itemset generation. The algorithm requires only two full I/O scans for the dataset. Our approach presented in this paper is based on this idea.

Parallel algorithms: There has been a modest contribution in association rule mining from distributed databases [8] and in a parallel context [2] early on in data mining research. In spite of the significance of the association rule mining and in particular the generation of frequent itemsets, few advances have been done on parallelizing association rule mining algorithms. Most of the work on parallelizing association rules mining on Shared-memory MultiProcessor (SMP) architecture was based on apriori-like algorithms. Zaki et al. proposed the Common Candidate Partitioned Database (CCPD) and the Partition Candidate Common Database (PCCD) algorithms, which both are apriori-like algorithms [25]. Although these were among the first attempts toward parallelizing association rules mining, they suffered from some severe problems like high I/O overhead, disk contentions, and poor data locality. Another apriori-like algorithm is the parallel data mining (PDM) algorithm [20]. This algorithm is a parallel implementation of the sequential DHP algorithm and it inherited its problems, which makes it impractical in some cases.

Another attempt was done on parallizing the association rules finding based on DIC algorithm [7]. This implementation was sensitive to the skewness of the data and assumes that all data should be homogenous in order to get good results.

An excellent recent survey on parallel association rule mining with shared-memory architecture covering most trends, challenges and approaches adopted for parallel data mining can be found in [21]. All approaches spelled out and compared in this extensive survey are apriori-based. These methods not only require repeated scans of the dataset, they also generate extremely large numbers of candidate sets easily approaching 10^{30} candidates in common cases [12].

1.3 Contribution

Apriori-like algorithms suffer from two main severe drawbacks: the extensive I/O scans for the databases, and the high cost of computations required to generate the frequent items. These drawbacks make these algorithms impractical in cases of extremely large databases. Other limitations were observed on the existing parallel association mining algorithms. For instance, most of them work only for few thousands of dimensions, relatively small data sizes, and are sensitive to the data skew, which causes load balancing problems [24].

In this paper we are introducing a new parallel association rules mining algorithm MLFPT based on the FP-growth algorithm [11]. This algorithm scans the database only twice to build a special data structure called Frequent Pattern Tree, which in turn is mined to generate the frequent itemsets. However, a shared tree structure among parallel processors necessitates locking mechanisms at the leaf or node levels or even paths, leading to significant bottlenecks. We have adapted the data structure by dividing the FP-Tree in chunks for each processor while keeping the resulting trees shared among processors to avoid false negatives (i.e. pruning inadvertently frequent itemsets that are locally infrequent). Building this structure reduces significantly the computational costs of generating frequent itemsets. The modified data structure allows for bottom-up traversal of the divided FP-Tree in a fast way during the mining process. Each chunk of the tree forest is assigned locally to one processor. The tree locality reduced the possibility of false sharing where parallel processors are accidentally overwriting other processors' updates, and consequently minimized the ping-pong effect where processors in turn await for resources to be released. The frequent items are also locally cross-linked in these trees, and globally linked by a global header table.

We have implemented this algorithm on a 64 processor 2400 SGI origin machine, where all experiments were tested using high dimensionality data that are of a factor of hundreds of thousands of items, and transactional sizes that range in tens of gigabytes. A special optimization step is added to achieve better load balancing with the goal of distributing the work fairly among processors for the mining process.

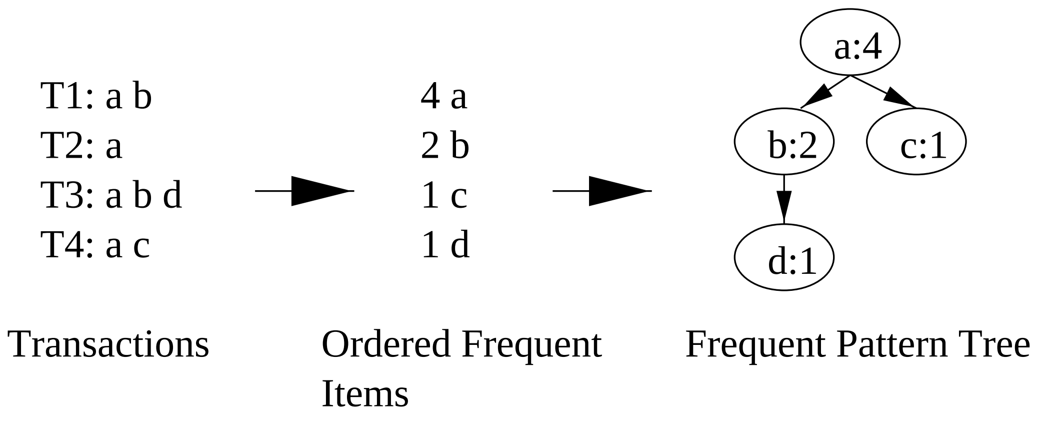


Figure 1: A frequent pattern tree example.

1.4 Preliminaries

The MLFPT algorithm that we are presenting in this paper is the parallel implementation based on the core idea of the FP-Tree algorithm proposed by Han et al. in [11]. This algorithm does not require an iterative generation of candidate frequent itemsets which avoids the computational overheads associated with the apriori-like algorithms. A compacted tree structure is built based on an ordered list of the frequent 1-itemsets present in the transactional database. Ordered frequent items of each transaction are represented by a path in the tree from the tree root to either a leaf node or an internal node. However, given the frequent repetitions of the items, the tree space is considerably smaller than the space needed by the transactions (See Figure 1 for an illustrative example). During the construction of the FP-Tree, the global frequency of the items are taken into account such that each transaction is transformed into a sorted list of frequent items before being mapped into a path in the FP-Tree. When paths overlap due to intersection of the transactions, the corresponding counters associated with the relevent nodes of the tree are incremented, and new items not in the prefix path are added as extension to the path. All item-nodes in the FP-Tree are linked across the tree starting from the item entry in the header table to keep track of each item in all frequent paths to ease the mining process in the subsequent phase.

A symmetric recursive decomposition of the prefix paths of the FP-Tree starting from the least frequent items generates the conditional pattern bases which in turn are used to generate the frequent itemsets. More details about the construction of the the tree for the sequential FP-Tree algorithm and the exploitation of this data structure for mining frequent itemsets can be found in [11]. This algorithm showed to be an order of magnitude faster than any apriori-like algorithm in a sequential programming setting.

The remainder of the paper is organized as follows: Section 2 describes the MLFPT parallel algorithm with an illustrative example. Experimental results are given in Section 3. Finally, Section 4 concludes our work.

2 Multiple Local Parallel Trees

The MLFPT approach we propose consists of two main stages. Stage one is the construction of the parallel frequent pattern trees (one for each processor) and stage two is the actual mining for these data structures, much like the FP-growth algorithm. The major difference between the data structure of the original sequential version and our data structure is the need for parallel access to the nodes of the tree and the shared counters for cumulating the frequencies. Our data structure is distributed among the processors. However, in order to avoid false negatives, where locally infrequent itemsets are pruned inadvertently while they are frequent globally, we need global counters. Though global counters necessitate locking mechanisms for mutual exclusion, that would add significant overhead and waiting time. Our approach with interlinked local counters avoids the need for locking (as schematically illustrated in step 1 of Figure 2. Thus, we evade the famous ping-pong problem in parallel programs.

2.1 Construction of the Multiple Local Parallel Trees

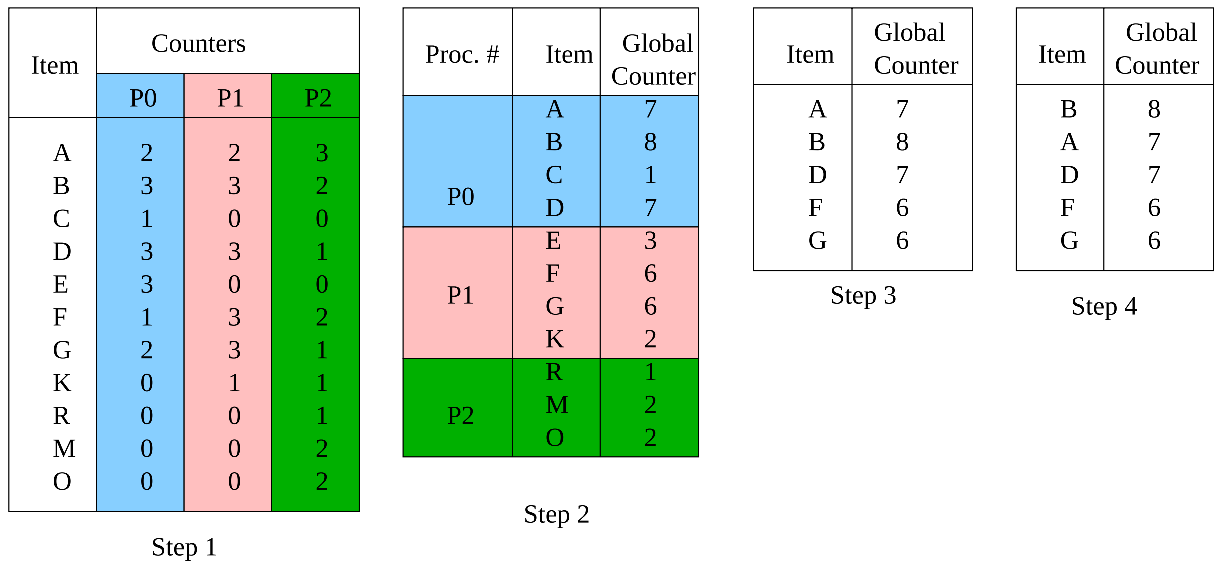
The goal of this stage is to build the compact data structures called Multiple Local Parallel Trees (MLPT) made of chunks of the original FP-Tree. The chunks are interconnected local trees. This construction is done in two phases, where each phase requires a full I/O scan for the dataset.

A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the trees in the second phase. In order to enumerate the frequent items efficiently, we divide the datasets among the available processors. Many strategies can be adopted to split the transactions among the processors such as a split based on the disk pages to avoid excessive movement of the disk reader head, etc. However, we opted for a simple partitioning based on the number of these transactions. Each processor is given an approximately equal number of transactions to read and analyze. As a result, the dataset is split in p equal sizes. For example the first processor would read the first n/p transaction where n is the total number of transactions and p is the number of available processors. The second processor would handle from transaction $\frac{n}{p}+1$ to transaction $\frac{2n}{p}$, etc., or each page accessed from the database is shared equally by the available processors, which is a better strategy. Each processor locally enumerates the items appearing in the transactions at hand. After enumeration of local occurences (i.e. after reading the whole dataset), a global count is necessary to identify the frequent items. This count is done in a parallel execution where each processor is allocated an equal number of items to sum their local supports into global count. This strategy eliminates the need for locking to protect the shared global supports and consequently avoids cache coherency. Finally, in a sequential execution infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called header table, where the items and their respective global support are stored along with pointers to the first occurrence of the item in each frequent pattern tree. Phase 2 would construct a frequent pattern tree for each available processor.

Phase 2 of constructing the MLPT structures is the actual building of the individual

TID	Items Bought	Processor Number
1	A, B, C, D, E	
2	F, B, D, E, G	$\rightarrow P_0$
3	B, D, A, E, G	
4	A, B, F, G, D	
5	B, F, D, G, K	$\rightarrow P_1$
6	A, B, F, G, D	
7	A, R, M, K, O	
8	B, F, G, A, D	$\rightarrow P_2$
9	A, B, F, M, O	

Table 1: Transactional database example.



Item	Counters		
	P0	P1	P2
A	2	2	3
B	3	3	2
C	1	0	0
D	3	3	1
E	3	0	0
F	1	3	2
G	2	3	1
K	0	1	1
R	0	0	1
M	0	0	2
O	0	0	2

Step 1

Proc. #	Item	Global Counter
P0	A	7
	B	8
	C	1
	D	7
P1	E	3
	F	6
	G	6
	K	2
P2	R	1
	M	2
	O	2

Step 2

Item	Global Counter
A	7
B	8
D	7
F	6
G	6

Step 3

Item	Global Counter
B	8
A	7
D	7
F	6
G	6

Step 4

Figure 2: Steps of phase 1.

local trees. This phase requires a second complete I/O scan from the dataset where each processor also reads the same number of transactions as in the first phase. Using these transactions, each processor builds its own frequent pattern tree that starts with a null root. For each transaction read by a processor only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the local FP-Trees as follows: for the first item on the sorted transactional dataset, check if it exists as one of the children of the root. If it exists then increment the support for this node. Otherwise, add a new node for this item as a child for the root node with 1 as support. Then, consider the current item node as the newly temporary root and repeat the same procedure with the next item on the sorted transaction. During the process of adding any new item-node to a given local FP-Tree of a processor p, a link is maintained between this item-node in the tree and its entry in the global header table corresponding to the p processor. The header table holds as many pointers per item as there are available processors.

For illustration, we use an example with the transactions shown in Table 1. Let the number of available processors be 3 and the minimum support threshold set to 4. Every processor reads 3 transactions. Processor P_0 reads transactions 1,2 and 3, processor P_1 reads transactions 4, 5 and 6, and finally processor P_2 reads the last 3 transactions 7, 8 and 9. During the reading process each processor accumulates the local support for the items that occur in its set of transactions. This local summation is illustrated in step 1 of Figure 2. After reading all transactions the item table is divided into 3 segments according to the number of processors, where each processor horizontally sums the global support for its allocated items (step 2 of Figure 2). Step 3 removes all non-frequent items, in our example (C, E, K, R, M, and O), leaving only the frequent items (A, B, D, F, and G). Finally all frequent items are sorted according to their support to generate the sorted frequent 1-itemset. This last step ends phase one of the MLFPT algorithm and starts the second phase, which we will explain by describing the process of building the local tree for processor P_0. The first transaction (A, B, C, D, E) read by processor P_0, is filtered to consider only the frequent items that occur in the header table (i.e. A, B and D) and is sorted according to the items' supports (B, A, D). This ordered transaction generates the first path of the local tree for processor P_0 with all item-node support initially equal to 1. A link is established between each item-node in the tree and its corresponding item entry for processor P_0 in the header table. The same procedure is executed for the second transaction (F, B, D, E, and G), which yields a sorted frequent item list (B, D, F, G) that shares the same prefix (B) with an existing path on its local tree. Item-node (B) support is incremented by 1 making the support of (B) equal to 2 and a new sub path is created with the remaining items on the list (D, F, G) all with support equal to 1. The last transaction read by processor P_0 is (B, D, A, E, G) that yields (B, A, D, G) sorted frequent list, which also shares a prefix (B, A, D) with an existing path on the tree. Items shared in this path are incremented by one (B=3, A=2, and D=2) and a new node (G=1) is added to the tree as a suffix to the existing path.

Figure 3 shows the result of the tree building process. For the sake of simplicity, only links from the items A and B are drawn from the header table.

2.2 Mining Parallel Frequent items using MLPT Trees

Building the trees in the first stage is not a final goal but a means with the purpose of uncovering all frequent patterns without resorting to additional scans of the data. The mining process starts with a bottom up traversal of the nodes on the MLPT structures, where each processor mines fairly equal amounts of nodes. The distribution of this traversal work is predefined by a relatively small sequential step that precedes the mining process. This step sums the global supports for all items and divides them by the number of processors to find the average number of occurrences that ought to be traversed by each processor. If A is this found average, this sequential step goes over the sorted list of items by their respective support and assigns items consecutively for each processors until the cumulated support is equal or greater than the average A. Since it is not realistic to obtain an assignment always equal to the average A, the optimum balance between processors is difficult to obtain. We recognize, however, that other assignment strategies could be used but the strategy we opted for does not add any significant overhead.

At this stage all frequent pattern trees are shared by all processors. The task of the

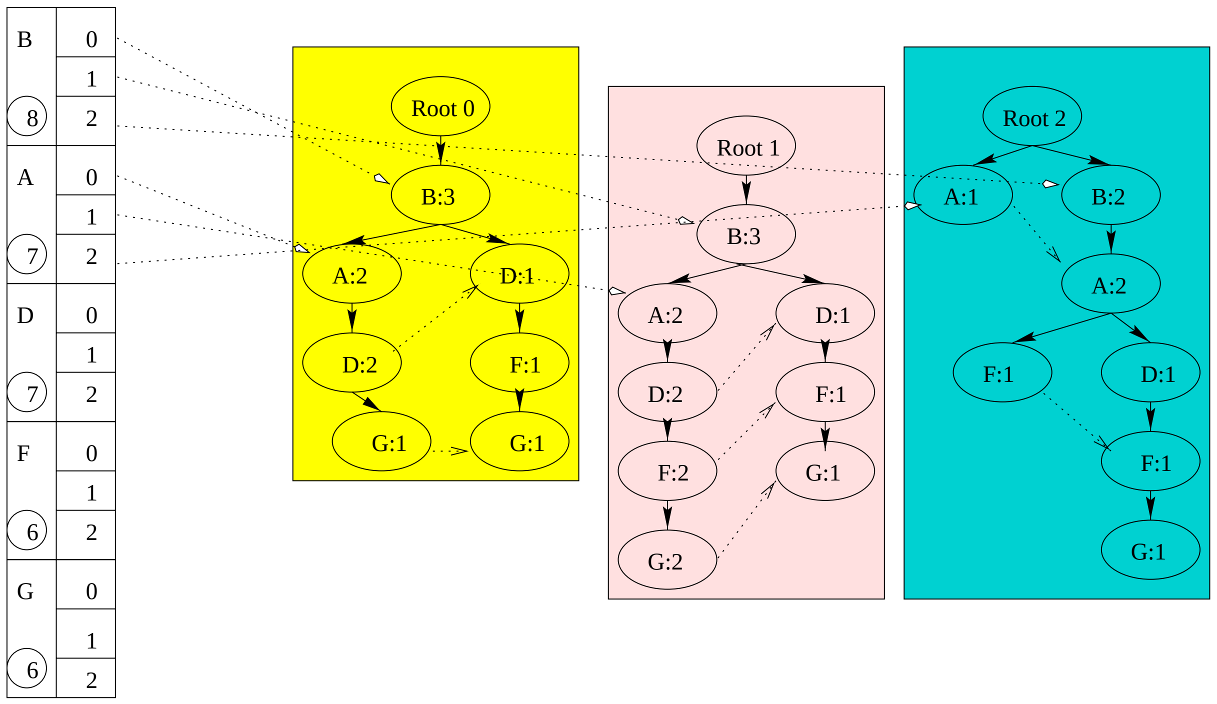


Figure 3: Phase 2 of the construction of the MLPT structure.

processors, once assigned some items, is to generate what is called a conditional pattern base starting from their respective items in the header table. A conditional pattern base is a list of items that occur before a certain item in the frequent pattern tree up to the root of that tree in addition to the minimum support of all the item supports along the list. Since an item can not only occur in many trees but also in many branches of the same tree, many conditional pattern bases could be generated for the same item. Merging all these conditional pattern bases of the same item yields the frequent string, a string also called conditional FP-Tree that contains frequent itemsets and their support in the presence of a given item. The merge is based on the items in the patterns and all the supports of the same items are added up in the same maner as in [11]. If the support of an item is less than the minimum support threshold, it is not added in the frequent string.

In our previous example, the sequential step sums the global support for all frequent items ($8 + 7 + 7 + 6 + 6 = 34$). See Figure 2, Step 4. Since we are using 3 processors in our example, the algorithm allocates for each processor about 11 occurrences, (34/3). This implies that processor P_0 mines items A and B ($8+7 = 15$ occurrences), processor P_1 mines items D and F ($7+6 = 13$ occurrences), and processor P_2 mines item G (6 ccurrences). To trace the frequent patters that A is involved in, One can follow the A entry links from the header table to find that item A appears with (A:2, B:3), (A:2, B:3), (A:2, B:2) and (A:1) in four different paths in the three different trees. The first path indicates that item A appears twice with the string S1=AB:2, twice with the string S2=AB:2 in the second path and finally twice with the string S3=AB:2 in the third path. S1, S2 and S3 are considered to be the conditional pattern bases of item A. By combining the three conditional pattern bases we can conclude the following frequent string {AB: 6 /A} called conditional FP-Tree in [11]. This result indicates that the string {AB} occurs 6 times on the transactional databases, and consequently, we can derive the frequency of all its subsequent frequent strings using the combination of the items in this string.

Items	**Conditional Pattern Base**	**Conditional FP-Tree**
G	(D:1, A:1, B:1)	(B:6, D:6, F:5, A:4)/G
	(F:1, D:1, B:1)	
	(F:2, D:2, A:2, B:2)	
	(F:1, D:1, B:1)	
	(F:1, D:1, A:1, B:1)	
F	(D:1, B:1)	(B:6, D:5)/F
	(D:2, A:2, B:2)	
	(D:1, B:1)	
	(D:1, A:1, B:1)	
	(A:1, B:1)	
D	(A:2, B:2)	(B:7, A:5)/D
	(B:1)	
	(A:2, B:2)	
	(B:1)	
	(A:1, B:1)	
A	(B2)	(B:6)/A
	(B:2)	
	(B:2)	
B	($\emptyset$)	

Table 2: Conditional Pattern Bases and the Conditional FPtrees (mining process).

Table 2 gives all conditional bases and conditional FP-Trees generated from the example in Table 1.

3 Experimental Results

A shared memory SGI Origin 2400 with 64 processors was used to conduct the experiments. We used synthetic transactional databases generated using the IBM Quest synthetic data generator [4]. The sizes of the input databases vary from 1 million transactions to 50 millions using dimensions that are multiples of hundreds of thousands. Each of these transactions has at least 12 items preceded by a unique transactional ID. The largest dataset is in the order 10 Gbytes.

In order to assess our algorithm objectively, we decided to evaluate each phase separately. The SBT [17] library was used to evaluate each phase. In our experiments we studied the MLFPT algorithm with 4, 8, 16, 32, 48 and 64 processors and compared it to its sequential version. The sequential version was of course implemented without the summation phase and with only one tree. Speedup measures the performance of parallel execution compared to the sequential execution: $S_p = T_1/T_p$ where S_p is the speedup achieved with p processors, T_1 is

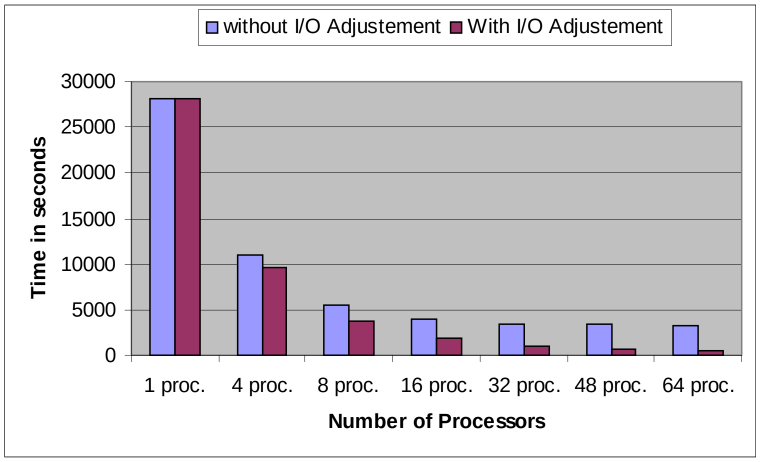


Figure 4: Comparison of execution time for 5 million transactions with and without I/O adjustement.

the sequential execution time and T_p is the execution time using p processors. Experiments were conducted to test the effect of varying the number of processors and varying the input size.

Originally, we had planned to experiment with one billion transactions and a terabyte size. Unfortunately, we were faced with some administrative constraints that prevented us from doing so since the SGI Origin machine is a shared resource. We are arranging exclusive access time to conduct these experiments with such large datasets. Our experimental results, however, suggest that we should be able to scale up to the billion transaction range. Moreover, the disk subsystem used with the parallel machine is not a high-performance disk array. This I/O bottleneck forced us to decide to adjust the I/O from our findings, as will be explained later, to better reflect the potential of the algorithm on a higher performance disk system.

Due to the limited time alloted for execution on the shared resource, we were not able to execute all configurations of our algorithm on all the datasets. For instance, we do not have the execution time for our sequential implementation using more than 5 million transactions, because, beyond that size the process was killed before termination. Nevertheless, we succeeded in running our algorithm with up to 64 processors on 50 million transactions using 100,000 different items and 12 items per transaction on average, which is larger by one order of magnitude than any other reported experiment in the literature.

I/O access is normally of an “embarrassingly parallel” nature. For instance, when data is stored on parallel disks with dedicated channels, twice as many processors should read twice as much data. In other words, with appropriate hardware, if it takes t time for one processor to read some data, it should take t/p for p processors to cover the same data.

Since our parallel machine had a sequential disk with one shared head, to assess the real speedup of MLFPT which does 2 I/O scans of the data regardless of the number of processors, we adjusted the I/O time assuming an “embarrassingly parallel” I/O access.

In our results we decided to adjust the I/O time of our algorithm as follows: The I/O time for parallel execution was estimated using the I/O time for sequential execution divided by the number of processors used. For instance, if using p processors the total execution time is T and the isolated I/O time is t, the execution time with I/O adjusted is calculated $T\prime = T - t + (S/p)$, where S is the isolated I/O time for a sequential execution. In other words, we replaced the 2 scans I/O time recorded with the expected real parallel I/O time.

Table 3 presents the timing in seconds of the MLFPT algorithm after adjusting the I/O time. Figure 4 depicts the significant time reduction with the increase of processors when mining 5 million transactions. It seems that there is a plateau after 16 processors, but when the I/O is adjusted, one can observe a continual slight increase. It happens that due to the disk hardware hold-up, the I/O time increases with the number of processors.

Table 4 and Figure 5 present the speedup achieved in cases of 1 million and 5 million transactions. The speed up with other data sizes were not computed since the sequential time for these sizes was not obtained due to the execution time constraint explained above. The results in Table 4 show that the speedup is normally and uniformly increasing with the number of processors.

	Number of transactions (in millions)				
# of Processors	**1M**	**5M**	**10M**	**25M**	**50M**
1 Proc.	5213.70	28154.00			
4 Proc.	1717.44	9616.20	19352.13		
8 Proc.	731.06	3739.57	7154.61	20644.38	
16 Proc.	346.73	1901.04	3813.90	13809.75	11924.25
32 Proc.	183.29	963.52	2479.93	6753.52	8032.81
48 Proc.	130.17	639.50	1930.00	4407.56	5451.95
64 Proc.	101.73	527.70	1771.39	3326.94	3831.41

Table 3: Timing in seconds of MLFPT algorithm with I/O adjusted.

	Number of transactions (in millions)	
# of Processors	**1M**	**5M**
4 Proc.	3.04	2.93
8 Proc.	7.13	7.53
16 Proc.	15.04	14.81
32 Proc.	28.44	29.22
48 Proc.	40.05	44.03
64 Proc.	51.25	53.35

Table 4: Speed Up for the MLFPT algorithm with I/O adjusted.

3.1 Evaluation of the Results

MLFPT operations are divided into two stages where most of the computation in the MLFPT algorithm is done during building the MLPT trees, and then mining them. Building the

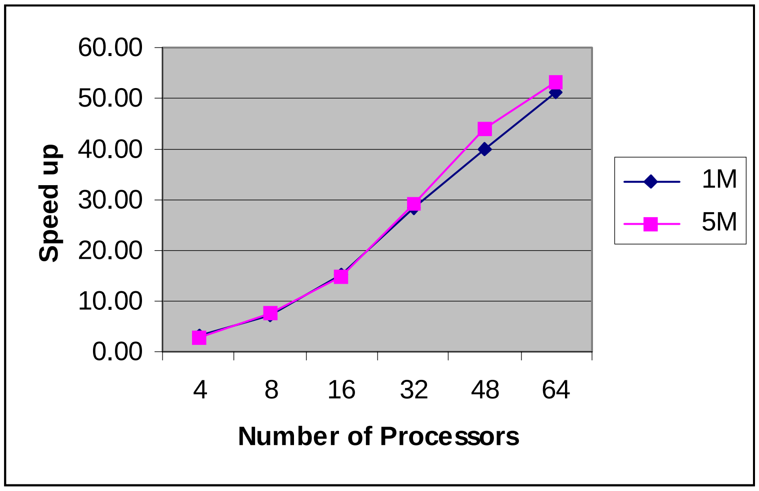


Figure 5: Speedup of the MLFPT algorithm.

frequent pattern trees, which utilize most of the processing time, is shown to be of "embarrassingly parallel" nature and this indeed was the reason for the several-fold improvements achieved as we increased the number of processors in our experiments. This is due to the fact that the work is evenly partitioned among the processors and each unit of work is completely independent of each other where each processor builds a sub-tree representing its partition of transactions. There is no ping-pong effect where processors are waiting for each other.

By comparing our algorithm to the previously published work on this area, we can find that most of the existing work adopts the apriori-based algorithms which need k I/O scans for the datasets to generate a frequent pattern of size k, whereas our algorithm does not need more than two I/O scans. The second problem in the apriori-based algorithms is the massive work needed in generating the candidate sets for each I/O scan where in the MLFPT algorithm this has been replaced by creating small FP-Trees only once, one for each processor, and then mining them for the frequent patterns. Our experiments have shown that this creation and mining is almost linearly proportional to the number of processors and the size of the transactional datasets, where the speedup of the MLFPT algorithm increases as the problem size increases. These results suggest that the MLFPT algorithm would achieve speedups for extremely large datasets as well.

3.2 Load balance and RDFA

RDFA [13] stands for the Relative Deviation of the size of the largest partition From the Average partition size of the p partitions. It is used to measure how a load is evenly balanced between the processors by measuring the deviation between the most loaded processor and the average load of all processors, and is defined as $\frac{m \times p}{n}$ where m is the size of the largest partition, p is the number of processors and n is the sum of all partitions. Ideally, which is never achieved in practice, RDFA measure should be equal to 1. Based on our experience, tt is common to see measures higher than 5 in parallel programs that split work horizontally. In the parallel data mining literature, to the best of our knowledge, no study on measuring and evaluating load balancing between processors has be done or published.

For our algorithm, partitioning the data is done several times during the mining process.

While building the MLPT structure the algorithm has to scan the dataset twice where each processor has to read the same number of transactions either to build the frequent 1-itemsets or to build the chunks of trees.

Load balancing is an issue during the mining process. Traditional horizontal division is used to divide the itemsets into equal size of itemsets to mine. However this approach yields a poor load balance due to the skewness of the transactional databases. In our experiments we assigned the same number of node-items for each processor to mine. Load balancing has been within 36% of optimal in cases of transactional datasets of size equal or less than 25 million transactions which is trully exceptional. RDFA measures do go higher for dataset sizes that reache 50 million transactions. However, the results even for 50 million transactions is very encouraging and illustrates small deviations from the average load, thus indicating a good load balance between processors.

We recognize that load balancing may deteriorate beyond the 50 million transaction level. However, we are investigating a new solution for partitioning the traversal of the tree nodes among processors during the mining stage of our algorithm. Table 5 presents the RDFA results of the MLFPT algorithm.

	Number of transactions (in millions)				
# of Processors	**1M**	**5M**	**10M**	**25M**	**50M**
4 Proc.	1.30	1.34	1.33	1.36	2.51
8 Proc.	1.29	1.34	1.35	1.36	2.49
16 Proc.	1.28	1.34	1.34	1.36	2.48
32 Proc.	1.27	1.33	1.36	1.35	2.30
48 Proc.	1.23	1.33	1.36	1.35	2.28
64 Proc.	1.21	1.33	1.36	1.34	2.21

Table 5: RDFA measure for our experiments.

4 Conclusion and Future Work

In this paper, we have introduced an efficient parallel implementation of an FP-Tree-based association rule mining algorithm and have proposed a solution for load balancing among processors and resource sharing with minimum mutual-exclusion locking. We have discussed our experiments with this new parallel algorithm, MLFPT, for mining frequent patterns without candidate generation. The MLFPT algorithm overcomes the major drawbacks of parallel association rule mining algorithms derived from *apriori*, in particular the need for k I/O passes over the data.

Our experiments showed that with I/O adjusted, the MLFPT algorithm could achieve an encouraging many-fold speedup improvement, which is almost linearly proportional to the number of processors. Although our current empirical datapoints for load balancing show close to optimum balancing, an important element of our future work is in improving load balancing even further by devising new strategies for partitioning tree traversal responsibilities among processors.

The implementation of our algorithm and the experiments conducted were on a shared memory and shared hard drive architecture. We have recently acquired a cluster with 8 dual processor nodes and we plan to investigate the same approach with shared nothing architecture and devise a new protocol for sharing global resources while minimizing the message passing overhead. We are in the process of experimenting our algorithms with up to 1 billion transactions.

Given the constraints of using shared resources (i.e. the SGI Origin machine) we have managed to experiment our algorithm on databases with 50 million transactions, while experiments reported in the literature barely reach 3.2 million transactions [24, 21].

Further development and experiments with Multiple Local Parallel Tree-based association rule mining using shared memory and shared-nothing architectures will be reported in the future.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

[2] R. Agrawal and J. C. Shafer. Parallel mining of association rules: Design, implementation, and experience. *IEEE Trans. Knowledge and Data Engineering*, 8:962–969, 1996.

[3] 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.

[4] IBM Almaden.

[5] E. Baralis and G. Psaila. Designing templates for mining association rules. *Journal of Intelligent Information Systems*, 9:7–32, 1997.

[6] S. Brin, R. Motwani, J. D. Ullman, and S.Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data*, pages 255–264, Tucson, Arizona, May 1997.

[7] D. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on shared-memory multi-processors. In *10th ACM Symp. Parallel Algorithms and Architectures*, pages 279–228, New York, 1998.

[8] D.W. Cheung, J. Han, V. Ng, A. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *Proc. 1996 Int. Conf. Parallel and Distributed Information Systems*, pages 31–44, Miami Beach, Florida, Dec. 1996.

[9] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. 1996 ACM-SIGMOD Int. Conf. Management of Data*, pages 13–23, Montreal, Canada, June 1996.

[10] 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.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.

[12] Jiawei Han and Micheline Kamber. *Data Mining, Concepts and Techniques.* Morgan Kaufmann, 2001.

[13] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19:1079–1103, 1993.

[14] H. Mannila, H. Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94)*, pages 181–192, Seattle, WA, July 1994.

[15] R.J. Miller and Y. Yang. Association rules over interval data. In *Proc. 1997 ACM-SIGMOD Int. Conf. Management of Data*, pages 452–461, Tucson, Arizona, May 1997.

[16] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. 1998 ACM-SIGMOD Int. Conf. Management of Data*, pages 13–24, Seattle, Washington, June 1998.

[17] E. Novillo and P. Lu. On-line debugging and performance monitoring with barriers. In *15th International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, California, USA, April 2001.

[18] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. 1998 Int. Conf. Data Engineering (ICDE'98)*, pages 412–421, Orlando, FL, Feb. 1998.

[19] 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.

[20] J.S. Park, M.S. Chen, and P.S. Yu. Efficient parallel data mining for association rules. In *ACM Int'l Conf. Information and Knowledge Management*, pages 31–36, New York, 1995.

[21] S. Parthasarathy, M. J. Zaki, and M. Ogihara. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems: An International Journal*, 3(1):1–29, February 2001.

[22] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules on large databases. In *Proc. 21st Int. Conf. on Very Large Data Bases (VLDB)*, pages 432–443, Zurich, Switzerland, 1995.

[23] Osmar R. Zaïane, Jiawei Han, and Hua Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Int. Conf. on Data Engineering (ICDE'2000)*, pages 461–470, San Diego, CA, February 2000.

[24] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency, Special Issue on Parallel Mechanisms for Data Mining*, 7(4):14–25, December 1999.

[25] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Supercomputing'96*, Pittsburg, PA, November 1996.