Fast Parallel Mining of Frequent Itemsets

H. D. K. Moonesinghe, Moon-Jung Chung, Pang-Ning Tan
Department of Computer Science & Engineering
Michigan State University
East Lansing, MI 48824
(moonesin, chung, ptan@cse.msu.edu)

Abstract

Association rule mining has become an essential data mining technique in various fields and the massive growth of the available data demands more and more computational power. To address this issue, it is necessary to study parallel implementations of such algorithms. In this paper, we propose a parallel approach to the Frequent Pattern Tree (FP-Tree) algorithm, which is a fast and popular tree projection based mining algorithm. In our approach we build several local frequent pattern trees and carry out the mining task parallely until all the frequent patterns are generated. We have devised a dynamic task scheduling strategy at different stages of the algorithm to achieve good workload balancing among processors at runtime. According to experimental results with data sets generated by the IBM synthetic data generator on a 32 processor distributed memory environment (*Terascale* Computing System), our parallel algorithm resulted in higher speedups in almost all the cases compared to the sequential algorithm. Also, our parallel algorithm showed scalable performance for larger data sets.

Keywords: Parallel association rule mining, Parallel FP-Tree algorithm.

1. Introduction

Knowledge Discovery & Data mining (KDD) has become an important inference process to discover previously unknown patterns in vast amount of data. Such knowledge discovery process provides a lot of useful information in all the fields of business, science, medicine, and etc. For example, in the field of business, identifying customer buying patterns and customer groups provide wealth of information to improve the organization. When considering the availability of massive volume of data, analyzing and decision making is still a major issue.

One popular and commonly used data mining task is the mining for associations, which is the process of finding associations between items in transactional data. There are several association rule mining algorithms available [1, 3, 13, 19]. One interesting algorithm is the *FP-Tree* algorithm recently proposed by Han et al [3]. Although the *FP-Tree* algorithm is very efficient, when compared with previously available association rule mining algorithms, it still takes a lot of time to mine massive volume of data measured in millions of transactions. To address this issue, it is necessary to study parallel implementation of such algorithms. Parallel implementation of tree projection based algorithms, such as FP-Tree, has received relatively little attention. Existing parallel implementations such as [2] have been targeted to shared-memory environments only.

In this paper, we propose a parallel formulation of the FP-Tree algorithm on a distributed memory environment. Our parallel algorithm consists of two main stages: parallel construction of FP-trees for each available processor, and parallel formulation of the FP-Growth sequential mining method to mine each FP-tree. We construct conditional pattern bases [3] and build Conditional FP-Trees (CFPT) [3] recursively in parallel until all the frequent itemsets are generated. We have identified two major bottlenecks of existing parallel approaches; one is the time to combine the conditional pattern bases of every processor to generate CFPTs and the other is the time to mine CFPTs recursively. Here we attack them by proposing a master-worker based dynamic task scheduling technique to balance the workload at run time. Due to the skewness of transactional databases, dynamic task scheduling technique we propose here will give good performance than existing approaches with static task scheduling. We also give a time complexity analysis of the parallel algorithm to justify our results.

We have implemented and tested our algorithm using large data sets on a *Terascale* Computer System (TCS) at the *Pittsburgh Supercomputing Center*. Our machine environment consists of *Compaq Alpha* server ES45 nodes connected by a high-speed network and we used MPI [15] to achieve parallel communication. We have used up to 32 processors of this system and evaluated our algorithms with transactional database size of up to 5 millions. In almost all the cases, we achieved high speedups and scalable performance.

The rest of the paper is organized as follows: Section 2 describes the problem statement and related research in this area. Section 3 describes the parallel execution model and the proposed parallel FP-tree algorithm in detail. Section 4 presents the environment used to test our algorithm and the results obtained by evaluating it on the target architecture. Finally, Section 5 concludes the paper.

2. Frequent Itemset Mining

The problem of mining for frequent patterns can be stated as follows: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct *items*. An *itemset* S is a non-empty subset of items; i.e. $S \subseteq I$. Itemset with k items is called a *k-itemset*. Also, an item can occur only once in an itemset.

A transaction database D consists of set of transactions where each transaction is of the form: (transaction_id, itemset). The *support* of an itemset S is defined as the fraction of total transactions that contain S. An itemset is called *frequent* if its support is above a user specified minimum support threshold t. Given a database D of transactions and support threshold t, the problem of mining for frequent patterns is to find all frequent itemsets in the database.

2.1 Related Work

The problem of mining association rules was first addressed by Agrawal et al [8]. Since then a number of algorithms for association rule mining has been proposed [1, 3, 4, 7, 13, 18, 19]. A popular algorithm is the *Apriori* algorithm [1], which forms the foundation for many sequential and parallel algorithms. However candidate set generation of this algorithm seems to be a major cost. Recently tree projection algorithms [4, 7], where transactions in the database are projected to a lexicographic tree, were proposed. *FP-Tree* [3] is another such algorithm, which creates a compact tree structure and applies partitioned-based, divide & conquer method of mining. This approach has shown faster execution time than other recent techniques in the literature.

There have been several parallel association rule mining algorithms in the literature [2, 5, 6, 14, 15, 16, 17, 21, 22, 23]. Most such parallel algorithms are *Apriori* based. Agrawal et al [6]

presents three different parallel versions of the *Apriori* algorithm on distributed memory environment. The count distribution algorithm replicates the generation of the candidate set and is a straightforward parallelization of *Apriori*. The other two algorithms (data distribution and hybrid) partition the candidate set among processors. Park et al [14] uses a similar approach by replicating the candidate set on all processors. Several parallel mining algorithms were presented in [5] for generalized association rules and they addressed the problem of skewed transactional data. Cheung et al [22] also addressed the effect of data skewness in their *FPM* (Fast Parallel Mining) algorithm, which is based on the count distribution approach. A parallel tree projection based algorithm, called *MLFPT*, based on *FP-Tree* algorithm is presented in [2] for a shared memory environment. In our approach we construct several local FP-trees similar to that in [2] on a distributed memory environment. Also, [2] used static task partitioning strategy to balance the workload. But in our approach we used master-worker based dynamic task-scheduling strategy during the merging phase of conditional pattern bases and the mining phase to balance the workload dynamically at runtime, and obtained fast performance than in [2] on a distributed memory environment.

3. Parallel Itemset Mining

Our proposed parallel frequent pattern mining algorithm consists of parallel construction of frequent pattern trees and parallel mining of the tree structure on a distributed memory environment. A detailed description of each major step is described in subsequent sections.

3.1 Parallel Frequent Pattern Tree Construction

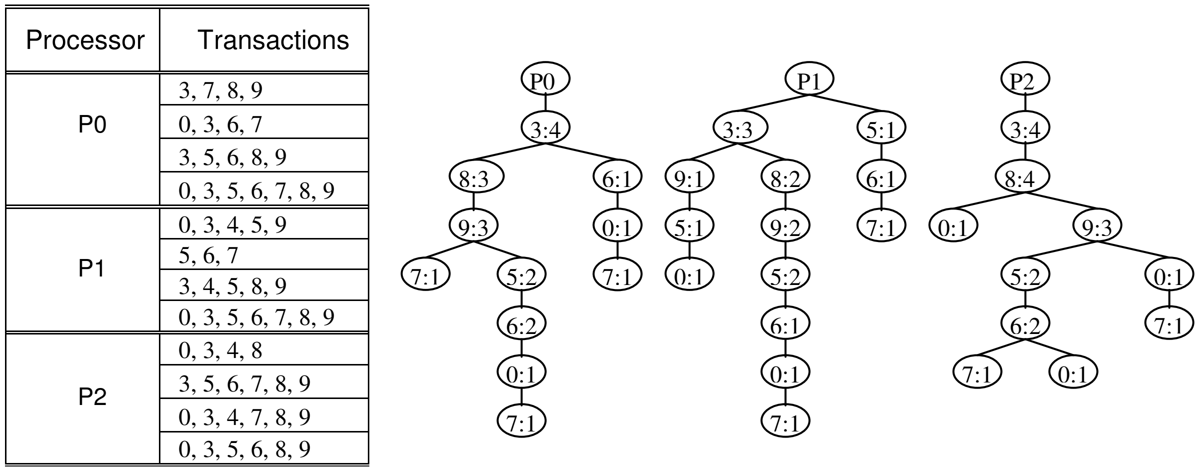
The first stage of the parallel mining algorithm is the construction of FP-trees parallely on each processor. For this purpose, we divide the transactional database (D) equally among available processors (P). This ensures that each processor gets N/P transactions ($D_{N/P}$), where N and P are the total number of transactions in the database and the number of processors available respectively. Partitioning of the database among the P processors is done randomly. After partitioning of data, it is necessary to identify the frequent 1-itemset ($F_{1\text{-}itemset}$) before building a local FP-tree. Once each processor counts the frequency ($f_{local}(i)$) of each item i using its local data partition $D_{N/P}$ all worker processors send the local count $f_{local}(i)$ to master processor. The master processor collects all such items and combines them to generate a global count $f_{global}(i)$. After that it removes the items with support count less than the minimum support threshold t. Once a complete frequent 1-itemset is constructed, it will be broadcasted to all the processors in the group.

The next step is the building of FP-trees. Each processor scans its local database $D_{N/P}$ and inserts frequent items into its local FP-tree. Construction of a FP-tree by each processor for its local database is as same as that described in the serial algorithm [3]. Further more, we have used the same data structure illustrated in [3] to implement the tree.

We have shown a sample database, with 12 transactions and 10 items (0, 1, 2,…, 9) in Figure 1. The database is partitioned among 3 processors (*P0*, *P1*, *P2*) where each processor has equal number of transactions. With a support threshold of 6 we can determine the frequent 1-itemset with the support count as {3:11, 8:9, 9:9, 5:8, 6:7, 0:7, 7:7}. Figure 1 also shows the initial FP-trees constructed by each processor using its local database.

3.2 Parallel Mining & Frequent Itemset Generation

Our mining approach consists of several stages. In the first stage we traverse the local FP-tree and form the conditional pattern bases. In the next stage, we combine conditional pattern bases of every processor to build the first conditional FP-tree (CFPT) for each frequent item. Final stage is to perform further mining by building conditional pattern bases and CFPTs recursively until it generates all the frequent itemsets. A detail description is given in subsequent sections.



Processor	Transactions
P0	3, 7, 8, 9
	0, 3, 6, 7
	3, 5, 6, 8, 9
	0, 3, 5, 6, 7, 8, 9
P1	0, 3, 4, 5, 9
	5, 6, 7
	3, 4, 5, 8, 9
	0, 3, 5, 6, 7, 8, 9
P2	0, 3, 4, 8
	3, 5, 6, 7, 8, 9
	0, 3, 4, 7, 8, 9
	0, 3, 5, 6, 8, 9

Figure 1: Partitioned transaction database & initial FP-trees

3.2.1 Construction of Conditional Pattern Bases

Each processor visits its header table (local frequent 1-itemset) in a top-down manner and forms the conditional pattern bases for each frequent item. Formation of the conditional pattern bases is done by a bottom-up traversal of the nodes in the local FP-Tree as in the serial algorithm [3]. This process is illustrated in Table 1 for our sample database.

3.2.2 Building of First Conditional FP-Tree

When all the conditional pattern bases are available, conditional FP-Trees are built by merging the conditional pattern bases. Our method of merging is similar to that mentioned in [2] and [3]. For each frequent item, the conditional pattern bases are merged such that the support counts of the same items are added to calculate the total support count. Also, if this total support count of an item is less than the minimum support threshold, the item will be removed from the conditional FP-tree. This process is illustrated in Table 1 for the minimum support threshold of six.

We found out that the merging phase is one bottleneck of our parallel approach, which takes a lot of time for a large database. So, here we propose a parallel model to generate conditional FP-trees, such that the workload is fairly balanced among available processors. Our parallel model is a master-worker model. The master processor submits the items to be mined and the worker processors generate the conditional FP-Trees for those items. Once a worker processor completes the generation of conditional FP-tree for a given item, it sends a token to the master processor requesting the next item. The task of the master processor is to listen to incoming requests from

any worker processors. It responds to them by sending the next item. Once the worker processor receives the next item, it will start generating the CFPT for that item. The communication overhead is minimal because each processor sends a token only. Also, the workload is balanced fairly among the processes in the group. Because once a processor finished its task, it gets another one immediately.

Items	Conditional Pattern Bases			Conditional FP-Trees	
	P0	P1	P2	Before Threshold	After Threshold
7	9 8 3: 1 0 6 5 9 8 3: 1 0 6 3: 1	0 6 5 9 8 3: 1 6 5: 1	6 5 9 8 3: 1 0 9 8 3: 1	(3:6, 8:5, 9:5, 5:4, 6:5, 0:4)	(3:6)
0	6 5 9 8 3: 1 6 3: 1	5 9 3: 1 6 5 9 8 3: 1	8 3: 1 6 5 9 8 3: 1 9 8 3: 1	(3:7, 8:5, 9:5, 5:4, 6:4)	(3:7)
6	5 9 8 3: 2 3: 1	5 9 8 3: 1 5: 1	5 9 8 3: 2	(3:6, 8:5, 9:5, 5:6)	(3:6, 5:6)
5	9 8 3: 2	9 3: 1 9 8 3: 2	9 8 3: 2	(3:7, 8:6, 9:7)	(3:7, 8:6, 9:7)
9	8 3: 3	3: 1 8 3: 2	8 3: 3	(3:9, 8:8)	(3:9, 8:8)
8	3: 3	3: 2	3:4	(3:9)	(3:9)
3	∅	∅	∅	∅	∅

Table 1: Conditional pattern bases & first conditional FP-Trees after the merging

3.2.3 Generation of Frequent Itemsets

Generation of frequent itemsets is the final stage of our parallel mining process. Here we have proposed a parallel model to find frequent itemsets by building conditional pattern bases and conditional FP-Trees recursively by each processor. Once a conditional FP-tree with a one branch is constructed, we obtain all possible combinations of the items in that tree as frequent itemsets similar to that in the serial *FP-Growth* algorithm [3].

Our parallel model is a master-worker model similar to that used in the merging phase. The master processor submits the base items to be mined and the worker processors carry out the mining task for those items, and generate frequent itemsets. In this model, once a worker processor finished its task it gets another one immediately, which makes all the processors busy until the end of mining process. Here the work load balancing happens at runtime. Our master-worker model continues until all the frequent itemsets are generated for each frequent item in the $F_{1\text{-}itemset}$. After that all worker processors send their frequent itemsets to the master processor and the mining process stops.

For each item, frequent item sets are generated by constructing conditional pattern bases and conditional FP-Trees recursively as shown by Figure 2 (only first recursive level conditional FP-Trees are shown for clarity). Here we have three processors and therefore two worker processors

generate frequent itemsets as shown in Figure 2. Also, Figure 2 shows all the frequent itemsets for our example database.

Processor 1			Processor 2		
Item	Level-1 Conditional FP-Trees	Frequent Itemsets	Item	Level-1 Conditional FP-Trees	Frequent Itemsets
7	P1 – 3:6	(3 7: 6)	0	P2 – 3:7	(3 0: 7)
6	P1 – (3:6 – 5:5), 5:1 First recursive level	(3 6: 6) (5 6: 6)	5	P2 – 3:7 – (9:1), (8:6 – 9:6)	(9 5: 7) (3 5: 7) (8 5: 6) (3 9 5: 7) (3 8 5: 6) (8 9 5: 6) (8 3 9 5: 6)
8	P1 – 3:9	(3 8: 9)	9	P2 – 3:9 – 8:8	(3 9: 9) (8 9: 8) (3 8 9: 8)

Figure 2: Frequent itemsets generated by subsequent construction of pattern bases & CFPTs

3.2 Parallel FP-Tree Algorithm

In this section we give our parallel FP-Tree itemset mining algorithm, based on dynamic task partitioning technique.

Algorithm: Parallel frequent itemset mining.
Input: Transaction database $D_{N/P}$ for each processor and minimum support threshold t.
Output: The complete set of frequent itemsets.
Method:

(1) read local transactional database $D_{N/P}$;
(2) count local frequency $f_{local}(i)$ for each item i;
(3) if *Master* processor then
(4) for each *Worker* processor do
(5) receive $f_{local}(i)$;
(6) let $F_{1\text{-}itemset} = \{i \mid \sum f_{local}(i) \geq t$ for each item $i\}$ & broadcast $F_{1\text{-}itemset}$ to all;
(7) else send $f_{local}(i)$ for each item i to *Master* and receive frequent 1-itemset $F_{1\text{-}itemset}$;
(8) build local FP-Tree FPT_{local} by scanning local $D_{N/P}$ for items in $F_{1\text{-}itemset}$;
(9) traverse FPT_{local} and generate conditional pattern bases and broadcast to all;
(10) if *Master* processor then
(11) for each frequent item i $\in F_{1\text{-}itemset}$ do // Task scheduling to form CFPTs
(12) get *Worker* processor request and send item i;
(13) for each frequent item i $\in F_{1\text{-}itemset}$ do // Task scheduling for mining
(14) get *Worker* processor request and send item i to be mined;

(15) for each *Worker* processor do
(16) collect frequent itemsets and output all frequent itemsets;
(17) else do // Merging of conditional pattern bases
(18) request next item i and generate Conditional FP-Tree $CFPT_i$;
(19) until end of frequent items;
(20) broadcast *CFPT*s to every processor except *Master* and receive all *CFPT*s;
(21) do
(22) request next item i and call *FP-Growth-OneItem (CFPTs, null, i)*;
(23) until end of frequent items;
(24) send frequent itemsets to *Master*;

Subroutine FP-Growth-OneItem (*Tree*, α, i)
Method:

(1) if *Tree* contains a single path and $i \neq$ null then
(2) generate itemset with support $\geq t$ for each combination of the nodes in the path
(3) else if $i \neq$ null then
(4) generate itemset $\beta = i \cup \alpha$ and construct β's conditional pattern bases and $CFPT_\beta$
(5) else for each i in the header table of *Tree*
(6) generate itemset $\beta = i \cup \alpha$ and construct β's conditional pattern bases and $CFPT_\beta$
(7) if $CFPT_\beta \neq \emptyset$ call *FP-Growth-OneItem*($CFPT_\beta$, β, *null*)

4. Performance Evaluation

4.1. Evaluating Environment

Our machine environment consists of 750 *Compaq Alpha* server ES45 nodes and two separate front-end nodes (*Lemieux* at *Pittsburgh Supercomputing Center*). Each such node contains four 1-GHz processors and runs the Tru64 Unix operating system. A Quadrics interconnection network connects the nodes. Each node is a 4 processor SMP, with 4Gb of memory [13].

Our programs are compiled using C compilers (gcc) with default code optimization. To run a job, we use batch mode (dedicated) available on *Lemieux*. The Portable Batch System (PBS) scheduler controls all access to *Lemieux's* compute nodes.

We used transactional databases generated by the IBM Quest synthetic data generator [12]. The sizes of our databases vary from 1 to 5 millions transactions. There are 10,000 different items in all of our transactional databases and the average length of a transaction is 10 to 20 items. Various parameters of the databases are shown in Table 2. Also, each transaction contains set of items preceded by the length of the transaction. We used minimum support threshold of 0.1% in all our experiments.

Parameters	Data Sets				
	T10I4D1000k	T20I4D1000k	T10I4D1500k	T10I4D2000k	T10I4D5000k
No. of Transactions in Database	1,000,000	1,000,000	1,500,000	2,000,000	5,000,000
No. of different Items in Database	10,000	10,000	10,000	10,000	10,000
Avg. No. of Items per Transaction	10	20	10	10	10
Avg. length of Maximal Pattern	4	4	4	4	4

Table 2: Parameters of the generated data sets

We used Message Passing Interface (MPI) library [15] to achieve parallel communication of our programs. MPI message passing library is available on the machine environment we used.

4.2. Experimental Results

We have run the proposed parallel algorithm with 2, 4, 8, 16 and 32 processors, and compared it with the FP-tree sequential algorithm. Sequential version of the FP-Tree algorithm is obtained from the author [3]. Both parallel and sequential algorithms were executed and the frequent itemsets were generated for a minimum support threshold of 0.1%. Itemsets generated by both algorithms were compared and parallel algorithm generated the same result as that of the serial algorithm. We have measured FP-tree building time and mining time in seconds. Table 3 shows our measurements for all the data sets for both parallel and sequential (P = 1) FP-Tree algorithms. Figure 3 shows execution time variation for the parallel algorithm. Two figures were shown because of the difference in time scale.

Data Set	Number of Processors					
	1	2	4	8	16	32
T10I4D1000k						
Build-Tree	1466.94	428.37	137.04	49.40	21.47	10.41
Mining	443.82	248.37	85.92	56.76	38.96	35.12
Total	1910.76	676.75	222.95	106.16	60.44	45.53
T20I4D1000k						
Build-Tree	9539.23	2372.88	647.34	186.57	64.57	20.30
Mining	3152.22	1554.34	773.78	431.67	267.94	239.92
Total	12691.45	3927.22	1421.12	618.24	332.51	260.23
T10I4D1500k						
Build-Tree	3281.21	889.67	274.67	91.26	36.74	18.22
Mining	659.37	367.02	126.48	77.55	55.06	47.25
Total	3940.58	1256.69	401.15	168.81	91.80	65.47
T10I4D2000k						
Build-Tree	5945.10	1548.94	443.55	142.83	54.49	25.77
Mining	880.45	484.29	166.62	90.12	77.22	64.62
Total	6825.55	2033.23	610.17	232.95	131.71	90.39
T10I4D5000k						
Build-Tree	33560.51	9299.69	2308.47	678.49	218.43	89.41
Mining	2269.07	1301.71	778.89	395.00	252.67	185.79
Total	35829.58	10601.40	3087.35	1073.49	471.10	275.20

Table 3: Execution time in seconds for FP-Tree algorithm

When analyzing the results of the parallel algorithm, total execution time always became smaller when the number of processors is increased. When we consider the time to build the FP-trees, it shows very impressive results. FP-tree building time becomes lower and lower when the number of processors increases and shows *superlinear* speedups for all the data sets. The reason behind this *superlinear* speedup is the dramatic reduction of the tree building cost (tree searching cost for insertion of new transaction and the variation of the number of tree nodes to be constructed on the existing tree) with the number of processors increasing. We will show later

how the time complexity of this stage justifies this behavior. Similar behavior has been observed in the past [22], when using such divide-and-conquer technique in mining with data set partitioning

The *build-tree* stage results suggest that we can improve the serial algorithm by applying divide-and-conquer technique for tree construction. Here, instead of building a single tree, we can partition the database into, say k partitions, and build k trees. The pattern bases of each tree can be combined using a similar approach as in the merging stage of the parallel algorithm. Preliminary experiments and results, which are not reported here, shows good improvements to the serial FP-Tree construction algorithm.

Figure 4 shows the speedup achieved by our algorithm according to the P = 2 processor case. It is clear that in almost all the cases, we have achieved good speedups. This is significant for higher database sizes and the parallel algorithm tends to be run faster for larger data sets. Total execution time always became lower when number of processors is increased. That is our algorithm shows scalable performance in this machine environment.

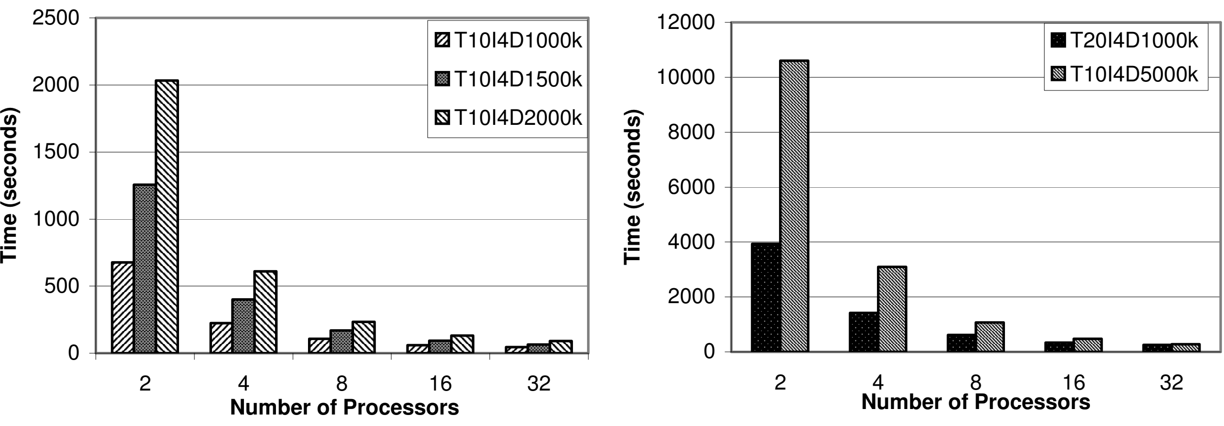


Figure 3: Execution time variations for small & large data sets

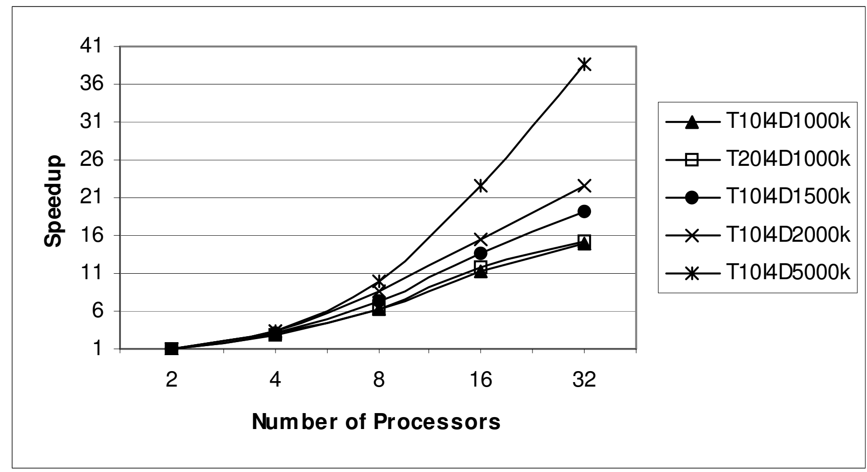
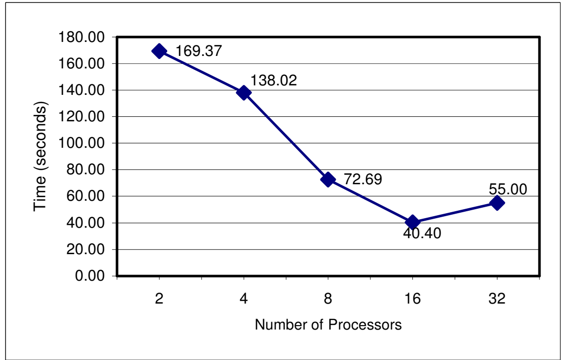


Figure 4: Scalability of the parallel FP-Tree algorithm

When we consider the time taken for the mining stage, Table 3 also shows that, for smaller data sets, the reduction of time from the 16-processor case to the 32-processor case is not sharp. For example, in T10I4D1000k data set, mining stage takes 38.96 seconds in the 16-processor case whereas it is 35.12 seconds for the 32-processor case. The reason is that there are several communication costs that we cannot avoid when the number of processors is increased. One such communication is the broadcasting of conditional FP-Trees by worker processors to all other worker processors. The size of the conditional FP-Tree is constant irrespective of the number of processors because it is an $n \times n$ table (n = number of frequent items) consisting of support counts. Time complexity analysis of this phase shows that its time complexity is $O(n^2P)$, where P is the number of processors. Another communication cost is the broadcast of conditional pattern bases. In our algorithm, each processor sends pattern bases to every processor. Therefore, the time complexity analysis of this communication phase is $O(nP)$. So, when the number of processors increases, communication time also increases heavily. Therefore these costs tend to be significant for higher number of processors in smaller data sets, but for larger data sets our algorithm performs well as shown by the evaluation.

Balancing the work load is a major issue in any parallel algorithm with task partitioning strategy. There are several static task partitioning techniques, such as random partitioning and partitioning based on support count, used in past approaches [2]. In random partitioning itemset is divided into equal size of itemsets randomly. Here the performance is poor because of the skewness of the transactions. Although partitioning based on support count gives better performance than random partitioning, work load is not accurately determined by the support count. Support count of an item is the frequency of that item in the database. It does not include other parameters of the transactional database, such as average number of items in a transaction, average length of the maximal pattern etc. Therefore, performance obtain from static task partitioning is sub optimal. Figure 5 shows how static partitioning works when merging conditional pattern bases to form the CFPTs. Here merging time increases after 16 processors, because of the load imbalance. Figure 5 (r.h.s. figure) shows the variation of merging time spent by each processor in 8-processor case.



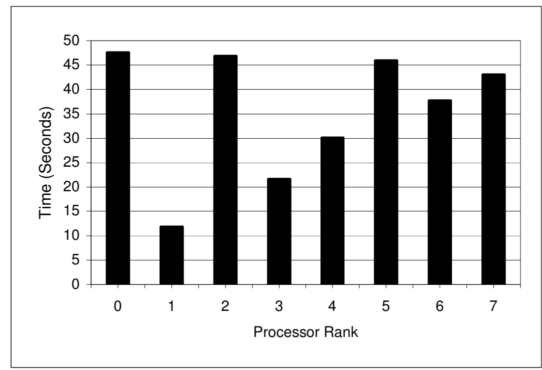
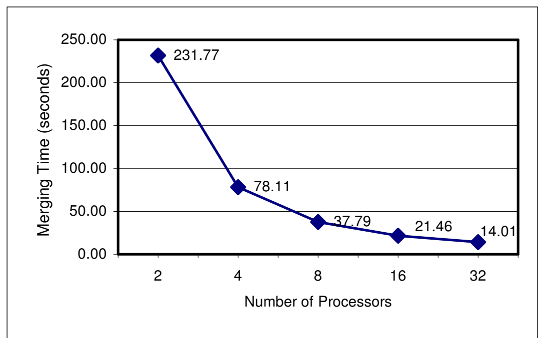


Figure 5: Parallel merging time variation for T10I4D1000k data set



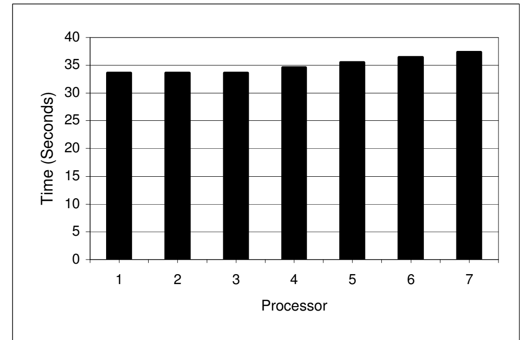
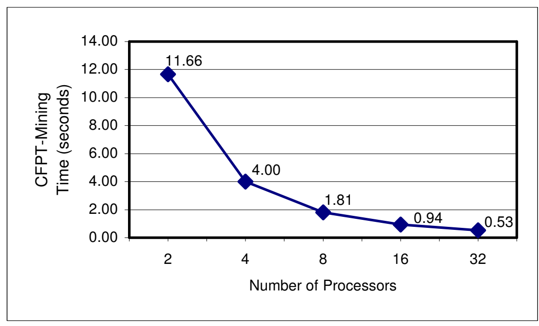


Figure 6: Parallel merging time variation for T10I4D1000k data set with load balancing



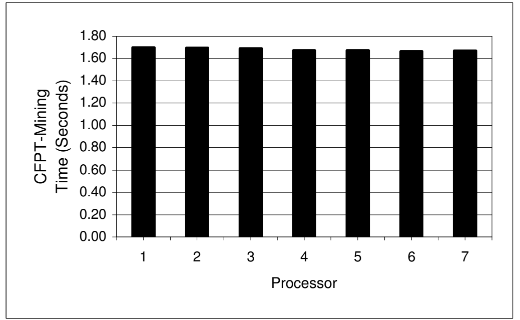


Figure 7: Time variation of Conditional FP-Tree mining stage for T10I4D1000k data set

Our proposed task partitioning strategy is a dynamic task partitioning technique, which balance the work load at runtime. When we consider the effectiveness of our master-worker based task scheduling technique used in the parallel merging of conditional pattern bases and mining of conditional FP-trees, it is clear that we always balance the work load well. We have measured the merging time taken by each processor for our experimental data sets and found out that the workload is completely balanced. Each processor spends almost the same time spend by another processor with a variation of ±10%. Figure 6 shows this scenario for T10I4D1000k data set in merging stage and Figure 7 shows this for the mining phase of CFPTs. Both figures show scalable performance and well balanced workloads.

4.3 Time Complexity Analysis of the Parallel Algorithm

In this section, we analyze the time complexity of our algorithm. Our parallel FP-tree algorithm consists of two stages: tree building stage & mining stage. When we consider the tree building stage of our parallel FP-Tree algorithm, we partition N transaction equally among P processors. So each processor works with total of N/P transactions and therefore in the worst case we have at most N/P branches in a single tree. Also, to insert i^{th} transaction into the tree, we have to search i-1 branches in the growing tree in the worst case. Now we can derive the time taken by the build tree stage as $\sum_{i=1}^{N/P}(i-1)+C$, which is $O(N^2/P^2)$. Here C is a constant. So, the speedup of the tree

building stage approach to P^2. This explains why we achieve *superlinear* speedups such as 375.3 in T10I4D5000k data set for 32 processors.

Now we analyze the time taken by the mining stage of the algorithm. Major computational components of the algorithm are: generation of pattern bases, merging of pattern bases to form CFPTs and use of FP-growth mining to generate frequent items. When time to generate pattern bases is considered, we have *n* frequent items and, each processor traverse the local FP-tree to find *N/P* number of pattern bases in the worst case for each frequent item. So, the time to generate pattern bases is O(*nN*/*P*). In the merging part, CFPTs need to be generated for *n* frequent items. In our master-worker model worker processors perform the actual merging of pattern bases. So, each worker processor shares the merging time. So, the cost of a single worker processor is: $\frac{1}{(P-1)}\sum_{K=1}^{n} T_{Merging_K}$ + C, where C is a constant. In mining stage we use FP-Growth mining method for each frequent item to generate frequent itemsets. Since we have parallelized this stage using a master-worker model, each of the (*P-1*) worker processors shares this cost evenly. So the cost of a single worker processor is: $\frac{1}{(P-1)}\sum_{K=1}^{n} T_{FP-Growth_K}$ + C.

The communication components of the mining stage are: communication of pattern bases to every processor, communication of CFPTs to every processor, collection of all frequent itemsets from each worker processor by the master processor. In the algorithm, each processor communicates pattern bases, of each frequent item, to every processor. So, the time to communicate pattern bases is O(*nP*). In our algorithm worker processors communicate CFPTs to every processor. Also, CFPT is a $n \times n$ table consisting of support counts. So, time to communicate CFPTs is O(n^2P). Also, after the generation of frequent item sets, master processor needs to collect them. So the communication cost for this phase is O(*P*). So the total communication cost is of O(*nP*) + O(n^2P) + O(*P*).

When analyzing the speedup of the mining stage the maximum we will achieve is O(*P*), when considering only the computational components of this stage. But, because of the communication cost we will never achieve this linear speedup. Figure 8 shows the actual speedup achieved by our algorithm for mining stage. Further more, because of the task scheduling based parallelism, it is possible that when number of processors is increased, the speedup achieved by the mining stage never increases, as one processor might be busy with a longest mining task.

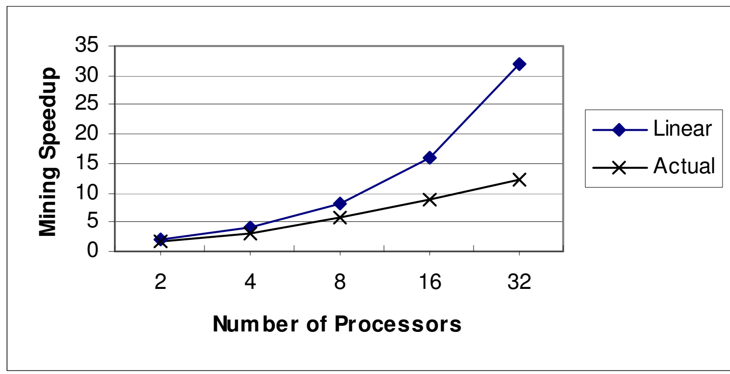


Figure 8: Speedup of the mining phase for T10I4D5000k data set

5. Conclusions

In this paper, we have proposed a parallel implementation of the sequential *FP-Tree* (and *FP-Growth*) mining algorithm. The main stages of our algorithm are to build the trees and carry out the mining task in parallel until all the frequent itemsets are generated. We have tested our algorithm using several data sets of size up to 5 millions on a 32-processor distributed memory environment. Our experiments showed good speedups for almost all the cases. Furthermore, it showed scalable performance on the 32-processor environment tested. This was significant for larger data sets, particularly with more than 2 millions transactions.

Experiments to measure execution time of various components of the algorithm revealed that, the master-worker based dynamic task scheduling model, we proposed for merging of conditional pattern bases and for mining of CFPTs has balanced the work load efficiently and showed scalable performance results. Therefore, we can say that our proposed dynamic task scheduling technique is an effective method for load balancing in distributed memory environments. Furthermore, this shows the effectiveness of dynamic task scheduling techniques when compared with the static task scheduling approaches.

Analysis of the experimental results and the time complexity of the parallel algorithm revealed that the sequential pattern tree construction approach can be further improved by constructing multiple trees sequentially for large databases, instead of constructing single large tree as in [3]. The number of trees to be built depends on several parameters of the database, such as average length of the transaction, number of transactions in the database, number of frequent items, and support threshold. We intend to improve the serial algorithm based on our findings and run our parallel algorithm for even higher number of processors with larger data sets, based on the resource availability on the machine environment.

References

[1] R. Agrawal and R. Srikant. Fast Algorithms for mining Association rules. *Proceedings of VLDB*, Sept 1994.

[2] O. R. Zaïane, Mohammad El-Hajj, and Paul Lu. Fast Parallel Association Rule Mining Without Candidacy Generation. *Proc. of the IEEE 2001 International Conference on Data Mining (ICDM'2001)*, San Jose, CA, USA, November 29-December 2, 2001.

[3] J. Han, J. Pei, and Y.Yin. Mining Frequent Patterns without Candidate Generation. *In ACM SIGMOD*, 2000.

[4] R. Agarwal, C. Aggarwal, and V. Prasad. A Tree Projection algorithm for generation of frequent itemsets. *In Journal of Parallel and Distributed Computing*, 2000.

[5] T. Shintani and M. Kitsuregawa. Parallel mining algorithms for generalized association rules with classification hierarchy. *In Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, pages 25--36, 1998.

[6] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. On Knowledge And Data Engineering*, 8:962-969, 1996.

[7] R. Agarwal, C. Aggarwal, V. Prasad, and V. Crestana. A tree projection algorithm for generation of large itemsets for association rules. *IBM Research Report*, RC21341, November 1998.

[8] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *In Proc. of the ACM SIGMOD conference on management of data*. pages 207-216, 1993.

[9] IBM Almaden, *Synthetic* Data Generation Code for Associations and Sequential Patterns. http://www.almaden.ibm.com/cs/quest/syndata.html

[10] Pittsburgh Supercomputing Center. Lemieux. *http://www.psc.edu/machines/tcs/lemieux.html*.

[11] W. Gropp and Ewing Lusk. User's Guide for mpich, a Portable Implementation of MPI. *Argonne National Laboratory*: University of Chicago (ANL/MCS-TM-ANL-96/6).

[12] MPI home page. *http://www.mcs.anl.gov/mpi*

[13] J. S. Park, M. S. Chen, and P. S. Yu. Using a hash-based method with transaction trimming for mining association rules. *IEEE TKDE*, 9(5), pp 813--825, Sept./Oct. 1997.

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

[15] D. Cheung, J. Han, V. T. Ng, A. W. Fuan, Y. Fu. A Fast Distributed Algorithm for Mining Association Rules. *Proc. of 1996 Int'l Conf. on Parallel and Distributed Information Systems (PDIS'96),* Miami Beach, Florida, USA, Dec. 1996.

[16] E. H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, No. 3, May/June 2000.

[17] D. Cheung, Hu K. and Xia S. Asynchronous Parallel Algorithm for Mining Association Rules on a Shared-memory Multi-processors. *Proc. The Tenth Annual ACM Symposium on Parallel Algorithms And Architectures (SPAA98)*, Puerto Vallarta, Mexico, June, 1998.

[18] K. Wang, Y. He, and J. Han. Pushing Support Constraints Into Association Rules Mining. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 3, May/June 2003.

[19] J. Han, and Y. Fu. Discovery of Multiple-Level Association Rules from Large Databases. *IEEE Transactions on Knowledge and Data Engineering*, 11(5), 1999.

[20] J. Han, M. Kamber. Data Mining Concepts and Techniques. *Morgan Kaufmann*, 2001.

[21] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, pages 343--373, 1998.

[22] D. Cheung and Y. Xiao. Effect of Data Skewness in Parallel Mining of Association Rules. *Proc. Pacific-Asia Conf. Knowledge Discovery and Data Mining*, Lecture Notes in Computer Science, Vol. 1394, SpringerVerlag, New York, 1998, pp. 48--60.

[23] T. Shintani, M. Kitsuregawa. Hash based parallel algorithms for mining association rules. *In Proc. of 4th Int. Conf. on Parallel and Distributed Information Systems*, 1996.