International Journal of Engineering Research and Development
e-ISSN: 2278-067X, **p**-ISSN : 2278-800X, www.ijerd.com
Volume 5, Issue 9 (January 2013), PP. 30-35

Data Mining By Parallelization of Fp-Growth Algorithm

Prof Vikram singh Tamra[1], Robil varshney[2]
[1,2]Computer Science, Swit Ajmer, Rajasthan, India

Abstract:- In this paper we present idea to make one main tree on master node and slave do processing with database rather than have multiple FP-trees, one for each processor Firstly, the dataset is divided equally among all participating processors P_i. Before the tree construction initiates, every processor must sort the items according to their support count. The master processor gathers all local counts to be summed together and ultimately form a global support count to be broadcasted to all processors in the group. Items having support count less than the minimum threshold are removed from the process in slave node as well as master node. The next step is the FP-tree construction in which each node scans and sends the transaction according to threshold. At the same time master node, make its tree and also updated the tree as items come from slave node.

I. INTRODUCTION

One of the currently fastest and most popular algorithms for frequent item set mining is the FP-growth algorithm. It is based on a prefix tree representation of the given database of transactions (called an FP-tree), which can save considerable amounts of memory for storing the transactions.
Frequent pattern mining plays an essential role in mining associations, correlations, causality, sequential patterns , episodes , multidimensional patterns and many other important data mining tasks. Most of the previous studies adopt an Apriori-algorithm may still suffer from the following two nontrivial costs:

- It is costly to handle a huge number of candidate sets. For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 length-2 candidates and accumulate and test their occurrence frequencies.
- It is tedious to repeatedly scan the database and check a large set of candidates by pattern matching, which is especially true for mining long patterns.

II. DEFINITION OF TERMS

Let $I = \{ i_1, i_2, i_3, \ldots i_n\}$ be a set of items. An item set X is a non-empty subset of I . An item set with m items is called an m-itemset. Tuple $< t_{id}, X >$ is called a transaction where t_{id} is a transaction identifier and X is an item set. A transaction database TDB is a set of transactions. Given a transaction database TDB , the support of an item set X , denoted as sup(X) , is the number of transactions including the item set X . A frequent pattern is defined as the item set whose support is higher than the minimum support min_sup.

III. CONCEPT OF FREQUENT PATTERN MINING ALGORITHM

In 2000, Han *et al.* proposed the FP-growth algorithm—the first pattern-growth concept algorithm. FP-growth constructs an FP-tree structure and mines frequent patterns by traversing the constructed FP tree. The FP-tree structure is an extended prefix-tree structure involving crucial condensed information of frequent patterns

FP-growth algorithm is an efficient method of mining all frequent item sets without candidate's generation. The algorithm mine the frequent item sets by using a divide-and-conquer strategy as follows: FP-growth first compresses the database representing frequent item set into a frequent-pattern tree. The next step is to divide a compressed database into set of conditional databases (a special kind of projected database), each associated with one frequent item. Finally, mine each such database separately.

a) FP-tree structure

The FP-tree structure has sufficient information to mine complete frequent patterns. It consists of a prefix tree of frequent 1-itemset and a frequent-item header table. Each node in the prefix-tree has three fields: *item-name*, *count,* and *node-link*.

- *item-name* is the name of the item.
- *count* is the number of transactions that consist of the frequent 1-items on the path from root to this node.
- *node-link* is the link to the next same item name node in the FP-tree.

Each entry in the frequent-item header table has two fields: *item-name* and *head of node-link*.

- *item-name* is the name of the item.
- *head of node-link* is the link to the first same item-name node in the prefix-tree.

b) Construction of FP-tree

FP-growth has to scan the *TDB* twice to construct an FP-tree. The first scan of *TDB* retrieves a set of frequent items from the *TDB* . Then, the retrieved frequent items are ordered by descending order of their supports. The ordered list is called an F-list. In the second scan, a tree *T* whose root node *R* labeled with “null” is created. Then, the following steps are applied to every transaction in the *TDB* . Here, let a transaction represent [*p* | *P*] where p is the first item of the transaction and *P* is the remaining items. In each transaction, infrequent items are discarded.

Then, only the frequent items are sorted by the same order of F-list.Call insert_tree (*p* | *P*, *R*) to construct an FP-tree. The function insert_tree (*p* | *P*, *R*) appends a transaction [*p* | *P*] to the root node *R* of the tree *T* .Pseudo code of the function insert_tree (*p* | *P*, *R*) is shown in Figure 1.An example of an FP-tree is shown in Figure 2. This FP-tree is constructed from the *TDB* shown in Table 1 with *min_sup* = 3. In Figure 2, every node is represented by (*item* □ □*name* : *count*) . Links to next same item name node are represented by dotted arrows.

TID	Item	Frequent Item
1	s, z, x, w, r, p, l, i	s, x, z, l, i
2	z, y, x, s, p, l, j	s, x, z, y, l
3	y, s, k, o, j	s, y
4	y, x, n, e, i	x, y, i
5	z, s, x, t, m, i, l, k	s, x, z, l, i

Table1. Sample TDB

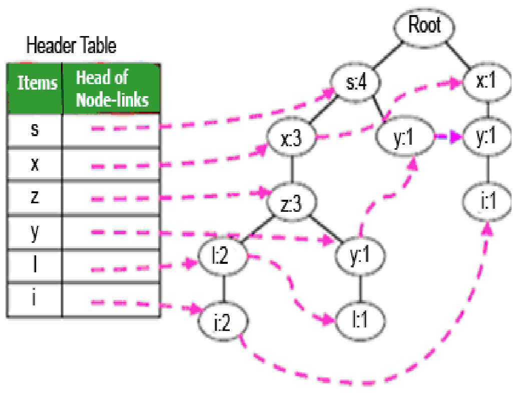

Figure 2. Example of an FP-tree

{(s:2, x:2, z:2, l:2),(x:1, y:1)}
i’s conditional pattern base

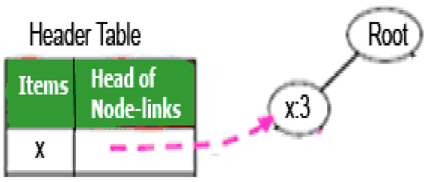

Figure 3.i ’s conditional FP-tree

c) FP-growth

FP-growth mines frequent patterns from an FP-tree. To generate complete frequent patterns, FP-growth traverses all the node-links from “head of node-links” in the FP-tree’s header table. For any frequent item a_i , all possible frequent patterns including a_i can be mined by following

a_i ’s node-link starting from a_i ’s head in the FP-tree header table. In detail, a_i’s prefix path from a_i’s node to root node is extracted at first. Then, the prefix path is transformed into a_i’s conditional pattern base, which is a list of items that occur before a_i with the support values of all the items along the list. Then, FP-growth constructs a_i’s conditional FP-tree containing only the paths in a_i’s conditional pattern base. It then mines all the frequent patterns including item a_i from a_i's conditional FP-tree. For example, we describe how to mine all the frequent patterns including item *p* from the FP-tree shown in Figure 2. For node i , FP-growth mines a frequent pattern (i:3) by traversing i 's node-links through node (i:2) to node (i:1). Then, it extracts i 's prefix paths; < s:4, x:3, z:3, l:2 > and <x:1,y:1>. To study which items appear together with *i* , the transformed path < s:2, x:2, z:2, l:2 > is extracted from < s:4, x:3, z:3, l:2 > because the support value of *i* is 2. Similarly, we have <x:1,y:1>. The set of these paths {(s:2,x:2,z:2,l:2),(x:1,y:1)}is called *i*'s conditional pattern base. FP-growth then constructs *i* 's conditional FP-tree containing only the paths in *i* 's conditional pattern base as shown in Figure 3. As only *x* is an item occurring more than *min_sup* appearing in *i* 's conditional pattern base, *i* 's conditional FP-tree leads to only one branch (x:3). Hence, only one frequent pattern (xi:3) is mined. The final frequent patterns including item i are (i:3) and (xi:3).

IV. EXISTED AND PROPOSED APPROACH

The proposed algorithm is based on one of the best known sequential techniques referred to as Frequent Pattern (FP) Growth algorithm. Unlike most of the earlier parallel approaches based on different variants of the Apriori Algorithm, the algorithm presented here does not explicitly result in having entire counting data structure duplicated on each processor. Furthermore, the proposed algorithm introduces minimum communication (and hence synchronization) overheads by efficiently partitioning the list of frequent elements list over processors.

4.1.1. Existing Parallel Fp-Tree Constructing Algorithm

It is obvious that Constructing FP-tree is the key step in FP-growth algorithm, FP-tree contains compressed message of frequent item sets, it is easy to mining frequent item sets from FP-tree. To construct FP-tree, we must scan data base twice, one for creating 1-itemset and one for build FP-tree. To mine useful information from database effectively, we needs to solve efficiency problem of mining, when data quantity very large, efficiency of mining algorithm become the key of mining problem, to increase efficiency of mining algorithm, so we developed parallel algorithm in a effective manner.

First we scan the database to identified the frequent 1-itemsets. In order to enumerate the frequent items efficiently, then we divide the datasets among the available processors. Each processor is given an approximately equal number of transactions to read and analyze. As a result, the dataset is split in n equal sizes. Each processor enumerates the items appearing in the transactions at hand. After enumeration of sub occurrences, a global count is necessary to identify the frequent items. This count is done in parallel where each processor is allocated an equal number of items to sum their sub supports into global count. Finally, in a sequential phase infrequent items with a support less than the support threshold are not consider in next steps and the remaining frequent items are sorted by their frequency.

On master node, These sorted transaction items are used in constructing the 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 .As transaction comes from slave nodes, master add them in it's already build tree and make one global tree with header table link and count. After that master applied FP Growth on this tree to find the conditional pattern base and make recursive call.

Existing Algorithm (PFPTC: Parallel FP-tree Constructing Algorithm)
Input: A transaction database DB and a minimum support threshold ξ.
Output: Its frequent pattern tree, FP-Tree
Method: The FP-tree is constructed in the following steps.

```
Suppose there are n processor p1, p2… pn;
Divide database DB into n part DB1… DBn with same size;
Processor $p_i$ scan DBj once. And create total set of frequent items F and their supports. Sort F in support descending order as L.
for ( j = 1; j≤n; j++) do ( concurrently)
{
Processor $p_j$ parallel scan DBj, construct FP-$tree_j$, do not create head table and node link ;
}
While (n>1) do
for ( j = 1; j≤n; j=j+2)
{
Processor pj parallel call FP-merge (FP-treej, FP-$tree_j$+1, int (j/2)+1 );
n=n/2 ;
}
Create head table and same nodes link for FP-tree1;
Return FP-tree1;
End;
```

4.1.2. Proposed Algorithm (PFPTC : Parallel FP-Tree Constructing Algorithm)

Input: A transaction database DB and a minimum support threshold ξ.
Output: Its frequent pattern tree, FP-Tree
Method: The FP-tree is constructed in the following steps.

```
Suppose there are n processor $p_1, p_2,\ldots,p_n$
(one master node $p_1$, remaining slave nodes $p_2,p_3\ldots\ldots p_n$);
Divide database DB into n part $DB_1,DB_2\ldots DB_n$ with same size;
Processor $p_j$ scans $DB_j$ once. And create total set of items and their supports. Send it to master node which merges them to generate the cumulative support.
The master node sends the generated results back to slaves individually.
for ( j = 2; j≤n; j++) do ( concurrently)
{
Processor pj parallel scan DBj,
For each transaction, find the frequent items of the transaction based on the global threshold.
Send the results to the master node for each transaction.
}
If (j=1)
{
For each transaction, find the frequent items of the transaction based on the global threshold and
   if ( tree is created)
      Add this data
Else
      Create tree with this data.
Receive the results from the slave nodes for each transaction for each slave node one by one.
Add this data to tree.
Call FP Growth on root.
Calculate execution time for FP Tree on each node (including master node)
Free memory
End;
```

V. PERFORMANCE EVALUATION

Number of experiments have been done to evaluate the scalability of parallel algorithm. Performance evaluation would be incomplete if we exclude from discussion issues that do not occur in sequential programming such as sources of parallel overhead and scalability.
The performance analysis results will indicate comparisons as follows:
Serial execution Vs. Parallel execution of given application. This results show how well cluster is performing for given application.
Show the scalability of parallel algorithm by means of execution, speed up and efficiency.

5.1 Serial Execution Vs Parallel Execution:

5.1.1 Consideration Of Database (Size=5577)

here we are taken a database of size 5577 transaction and perform mining of item serial as well as parallel. The corresponding result is shown in table. We analyze it through graph.

Threshold	Serial Execution (seconds)	Parallel Execution (seconds)
.10	.104	.048
.15	.060	.040
.20	.044	.038

Table 2. Execution time in serial and parallel algorithm

5.1.1.2 Graph Corresponding to Table:

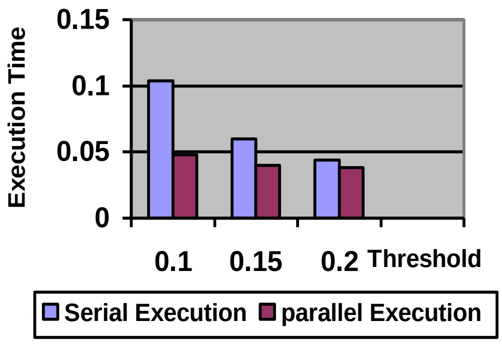

Figure 4. Graph between execution time and threshold

5.2 Scalability Evaluation:
Serial run-time is indicated using Ts
Parallel run-time using Tp.

5.2.1 Speed up:

Speed-up, S, is defined as "the ratio of the serial run-time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors:
S = Ts / Tp

5.2.2 Efficiency:

Efficiency E, is "a measure of the fraction of time for which a processor usefully employed"; it is defined as "the ratio of speed-up to the number of processors".
E = S / P
Result corresponding to Speed up and Efficiency

Threshold	Speed up	Efficiency
.10	2.16	.72
.15	1.5	.50
.20	1.15	.38

Table 3. Table for speed up and efficiency

VI. CONCLUSION

Throughout the last decade, a lot of people have implemented and compared several algorithms that try to solve the frequent item set mining problem as efficiently as possible. Moreover, we experienced that different implementations of the same algorithms could still result in significantly different performance results. As a consequence, several claims that were presented in some articles were later contradicted in other articles.
In this paper, we performed mining frequent item in transactional database for making difference between sequential and parallel algorithm, we noted the time of making FP Tree for both algorithm on different threshold. After compare the result of different metric with idle parallel system metric, we conclude that our algorithm works good in parallel environment.

VII. FUTURE SCOPE

"Change is the only constant thing.**"**

Rightly said. There is always a scope for amelioration. The future extension to this project may include some updations such as using optimization to make the database efficient. The database here comprised of numeric values. Instead, alphabetical letters could also be taken and then the scanning would be word by word. Also, most significantly, the processing time can be further compressed. Apart from that, we can also introduce the stream data analysis, in which data comes regularly and we have to perform mining on time bases. Efficiency of processors also matters in this approach, if some processors are more efficient than others than we can perform the load balancing concept by mean of divide the task according to processor efficiency.

The optimization method introduced still operates on static work generation. It puts down to the appropriate level and each time, produces work generation only in the middle 50 percent of the tree. This was applied in order to reduce the communication costs caused by the generation of jobs that are too small and thus not worth sending them to a different unit for processing. It would be much better if a dynamic work generation scheme is introduced. Such a scheme must be able produce a desirable number of nodes for processing according to the number of nodes at each level.

REFERENCES

[1]. T. Imielinski R. Agrawal and A.N. Swami. A tree projection algorithm for generation of frequent item sets. Parallel and Distributed Computing, pages 350–371, March 2001

[2]. J. Han J. Pei and Y. Yin. Minning frequent pattern without candidate generation. In Proc. 2000 ACMSIGMOD Int. Conf. on Management of Data. ACM, 2000.

[3]. Agrawal, R., Imielinski, T. and Swami, A. (1993) Mining association rules between sets of items in large databases. In Proceedings of the 1993 International Conference on Management of Data (SIGMOD 93), pages 207-216.

[4]. Bayardo R. J. (1998). Efficiently mining long patterns from databases. Proceedings of the 1998 ACM SIGMOD international conference on Management of data, 85-93.

[5]. Christian Borgelt. An implementation of fp-growth algorithm.

[6]. Li J., Liu Y., and Choudhary A. (2006). Parallel Data Mining Algorithms for Association Rules and Clustering. Handbook of Parallel Computing: Models, Algorithms and Applications. Sanguthevar Rajasekaran and John Reif, ed., CRC Press.

[7]. Han, E. H., Karypis, G., and Kumar, V. (2000) Scalable parallel data mining for association rules. IEEE Transactions on Knowledge and Data Engineering, 12, 337- 352. 76

[8]. Agrawal, R., Shafer, J. C., Center, I., and San Jose, C. A. (1996) Parallel mining of association rules. Knowledge and Data Engineering, IEEE Transactions on, 8, 962-969.

[9]. Chung, S. M., and Luo, C. (2003) Parallel mining of maximal frequent itemsets from databases. Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on, 134-139.