

# Parallel FP-growth on PC cluster

Iko Pramudiono and Masaru Kitsuregawa

Institute of Industrial Science, The University of Tokyo  
4-6-1 Komaba, Meguro-ku, Tokyo 153-8505, Japan  
{iko,kitsure}@tkl.iis.u-tokyo.ac.jp

**Abstract.** FP-growth has become a popular algorithm to mine frequent patterns. Its metadata FP-tree has allowed significant performance improvement over previously reported algorithms. However that special data structure also restrict the ability for further extensions. There is also potential problem when FP-tree can not fit into the memory. In this paper, we report parallel execution of FP-growth. We examine the bottlenecks of the parallelization and also method to balance the execution efficiently on shared-nothing environment.

## 1 Introduction

Frequent pattern mining has become one popular data mining technique. It also becomes the fundamental technique for other important data mining tasks such as association rule, correlation and sequential pattern.

FP-growth has set new standard for frequent pattern mining [4]. The compression of transaction database into on-memory data structure called FP-tree benefits FP-growth with performance better than previously reported algorithms such as Apriori [2].

Further performance improvement can be expected from parallel execution. Parallel engine is essential for large scale data warehouse. Particularly, development of parallel algorithms on large scale shared nothing environment such as PC cluster has attracted a lot of attention since it is a promising platform for high performance data mining. However parallel algorithm for complex data structure like FP-tree is much harder to implement compared to sequential program or shared-memory parallel system.

Section 2 lists related works on frequent pattern mining and its parallel executions. Section 3 briefly describes the underlying sequential FP-growth algorithm. In section 4 we explain our approaches for parallel execution of FP-growth on shared-nothing environment and we give the evaluation in section 5. Section 6 concludes the paper.

## 2 Related Works

Apriori is the first algorithm which addresses mining frequent pattern in 1995, particularly to generate association rules [2]. Many variants of Apriori based algorithms are developed since then.

Pioneering works on parallel algorithm for frequent pattern mining were done in [3, 5]. A better memory utilization schema called Hash Partitioned Apriori (HPA) was proposed in [6].

Some alternatives to Apriori-like "generate-and-test" paradigm were proposed such as TreeProjection[1]. However it was FP-growth that brought the momentum for the new generation of frequent pattern mining algorithms[4].

### 3 FP-growth

The FP-growth algorithm can be divided into two phases : the construction of FP-tree and mining frequent patterns from FP-tree [4].

#### 3.1 Construction of FP-tree

The construction of FP-tree requires two scans on transaction database. The first scan accumulates the support of each item and then selects items that satisfy minimum support, i.e. frequent 1-itemsets. Those items are sorted in frequency descending order to form F-list. The second scan constructs FP-tree.

First, the transactions are reordered according to F-list, while non-frequent items are stripped off. Then reordered transactions are inserted into FP-tree. The order of items is important since in FP-tree itemset with same prefix shares same nodes. If the node corresponds to the items in transaction exists the count of the node is increased, otherwise a new node is generated and the count is set to 1.

FP-tree also has a frequent-item header table that holds head of node-links, that connect nodes of same item in FP-tree. The node-links facilitate item traversal during mining of frequent pattern.

#### 3.2 FP-growth

Input of FP-growth algorithm is FP-tree and the minimum support. To find all frequent patterns whose support are higher than minimum support, FP-growth traverses nodes in the FP-tree starting from the least frequent item in F-list. The node-link originating from each item in the frequent-item header table connects the same item in FP-tree.

While visiting each node, FP-growth also collects the prefix-path of the node, that is the set of items on the path from the node to the root of the tree. FP-growth also stores the count on the node as the count of the prefix path. The prefix paths form the so called *conditional pattern base* of that item.

The conditional pattern base is a small database of patterns which co-occur with the item. Then FP-growth create small FP-tree from the conditional pattern base called *conditional FP-tree*. The process is recursively iterated until no conditional pattern base can be generated and all frequent patterns that consist the item are discovered.

The same iterative process are repeated for other frequent items in the F-list.

## 4 Parallel execution of FP-growth

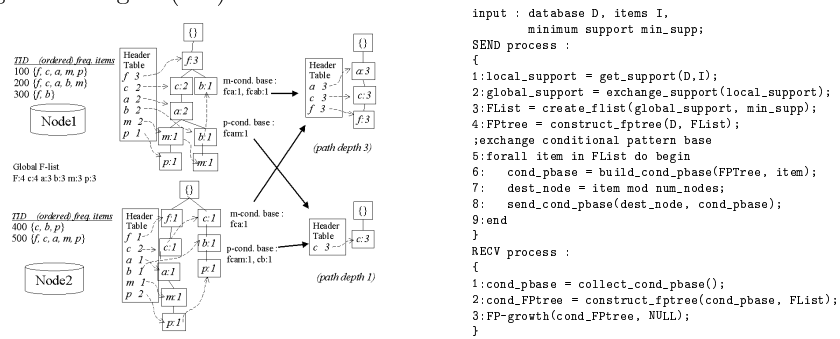
Since the processing of a conditional pattern base is independent of the processing of other conditional pattern base, it is natural to consider it as the execution unit for the parallel processing.

Here we describe a simple parallel version of FP-growth. We assume that the transaction database is distributed evenly among nodes.

### 4.1 Trivial parallelization

The basic idea is each node accumulates a complete conditional pattern base and processes it independently until the completion before receiving other conditional pattern base.

Pseudocode for this algorithm is depicted in Fig. 1 (right) and the illustration is given in Fig. 1 (left).



**Fig.1.** Trivial parallel execution : illustration (left) pseudo code (right)

After the first scan of transaction database the support count of all items are exchanged to determine globally frequent items. Then each node builds F-list since it also have global support count. Notice that each node will have the identical F-list. At the second database scan, each node builds local FP-tree from local transaction database with respect to the global F-list.

To find all frequent patterns, the From the local FP-tree, local conditional pattern bases are generated. But instead processing conditional pattern base locally, we use hash function to determine which node should process it. We can do this because of the accumulation of local conditional pattern base results in the global conditional FP-tree.

### 4.2 Path depth

It is obvious to achieve good parallelization, we have to consider the granularity of the execution unit or parallel task. Granularity is the amount of computation done in parallel relative to the size of the whole program.

When the execution unit is the processing of a conditional pattern base, the granularity is determined by number of iterations to generate subsequent conditional pattern bases. The number of iteration is exponentially proportional

with the depth of the longest frequent path in the conditional pattern base. Thus here we define *path depth* as the measure of the granularity.

**Definition 1.** *Path depth is the longest path in the conditional pattern base whose count satisfies minimum support count.*

Notice that path depth is similar with the term “longest pass” in Apriori based algorithms. The path depth also can be calculated during the creation of FP-tree.

Since the granularity differs greatly, many nodes with smaller granularity will have to wait busy nodes with large granularity. This wastes CPU time and reduces scalability. It is confirmed by Fig. 2 (left) that shows the execution of the trivial parallel scheme given in the previous subsection. The line represents the CPU utilization ratio in percentage. Here other nodes have to wait node 1 (pc031) completes its task.

To achieve better parallel performance, we have to split parallel tasks with large granularity. Since the path depth can be calculated when creating FP-tree, we can predict in advance how to split the parallel tasks.

Here we use the iterative property of FP-growth that a conditional pattern base can create conditional FP-tree, which in turn can generate smaller conditional pattern bases. At each iteration, the path depth of subsequent conditional pattern bases is decremented by one.

So we can control the granularity by specifying a *minimum path depth*. Any conditional pattern base whose path depth is smaller than the threshold will be immediately executed until completion, otherwise it is executed only until the generation of subsequent conditional patterns bases. Then the generated conditional pattern bases are stored, some of them might be executed at the same node or sent to other idle nodes. After employing the path depth adjustment we get a more balanced execution as shown in Fig. 2 (right).

## 5 Implementation and Performance Evaluation

### 5.1 Implementation

As the shared nothing environment for this experiment we use PC cluster of 32 nodes that interconnected by 100Base-TX Ethernet Switch. Each PC node runs the Solaris 8 operating system on Pentium III 800Mhz with 128 MB of main memory.

Three processes are running on each node :

1. SEND process create FP-tree, send conditional pattern base
2. RECV process receive conditional pattern base, process conditional pattern base after exchanging finish
3. EXEC process process conditional pattern base in background when exchanging

There are also small COORD processes that receive requests for conditional pattern base from idle nodes and coordinate how to distribute them.

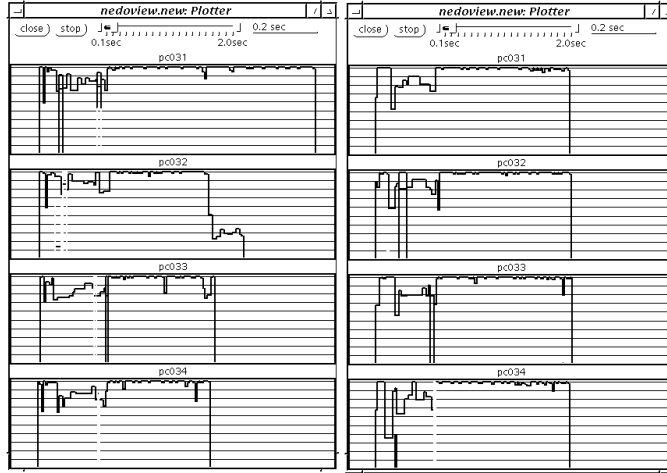


Fig. 2. Trivial execution (left) with path depth (right) (T25.I10.D100K 0.11%)

## 5.2 Performance Evaluation

For the performance evaluation, we use synthetically generated dataset as described in Apriori paper [2]. In this dataset, the average transaction size and average maximal potentially frequent itemset size are set to 25 and 20 respectively. While the number of transactions in the dataset is set to 100K with 10K items.

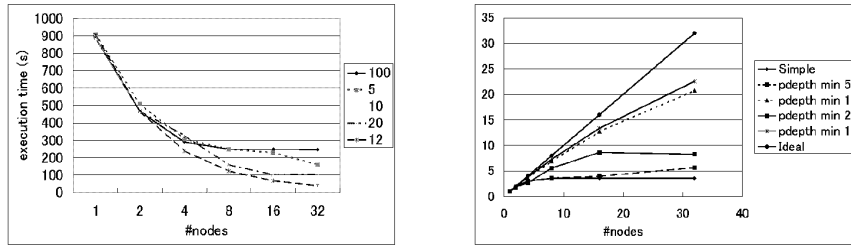
We have varied the minimum path depth to see how it affects performance. The experiments are conducted on 1, 2, 4, 8, 16 and 32 nodes. The execution time for minimum support of 0.1% is shown in Fig. 3 (left) The best time of 40 seconds is achieved when minimum path depth is set to 12 using 32 nodes. On single node, the experiment requires 904 seconds in average.

Fig. 3 (right) shows that path depth greatly affects the speedup achieved by the parallel execution. The trivial parallelization, denoted by “Simple”, performs worst since almost no speedup achieved after 4 nodes. This is obvious since the execution time is bounded by the busiest node, that is node that has to process conditional pattern base with highest path depth.

When the minimum path depth is too low such as as “pdepth min = 5”, the speedup ratio is not improved because there are too many small conditional pattern bases that have to be stored thus the overhead is too large. On the other hand, when the minimum path depth is too high, as represented by “pdepth min = 20”, the granularity is too large so that the load is not balanced sufficiently.

When the minimum path depth is optimum, sufficiently good speedup ratio can be achieved. For “pdepth min = 12”, parallel execution on 8 nodes can gain speedup ratio of 7.3. Even on 16 nodes and 32 nodes, we still can get 13.4 and 22.6 times faster performance respectively.

However finding the optimal value of minimum path depth is not a trivial task yet, and it is becoming one of our future work.



**Fig.3.** Execution time(left) Speedup ratio(right) for T25.I20.D100K 0.1%

## 6 Conclusion

We have reported the development of parallel algorithm or FP-growth that designed to run on shared-nothing environment. The algorithm has been implemented on top of PC cluster system with 32 nodes. We have also introduced a novel notion of *path depth* to break down the granularity of parallel processing of conditional pattern bases.

Although the data structure of FP-tree is complex and naturally not suitable for parallel processing on shared-nothing environment, the experiments show our algorithm can achieve reasonably good speedup ratio.

We are going to make direct comparison with other parallel algorithms using various datasets to explore the suitability of our algorithm.

## References

1. R. Agarwal, C. Aggarwal and V.V.V. Prasad "A Tree Projection Algorithm for Generation of Frequent Itemsets". In *J. Parallel and Distributed Computing*, 2000
2. R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In *Proceedings of the 20th International Conference on VLDB*, pp. 487-499, September 1994.
3. R. Agrawal and J. C. Shafer. "Parallel Mining of Association Rules". In *IEEE Transaction on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 962-969, December, 1996.
4. J. Han, J. Pei and Y. Yin "Mining Frequent Pattern without Candidate Generation" In *Proc. of the ACM SIGMOD Conference on Management of Data*, 2000
5. J.S.Park, M.-S.Chen, P.S.Yu "Efficient Parallel Algorithms for Mining Association Rules" In *Proc. of 4th International Conference on Information and Knowledge Management (CIKM'95)*, pp. 31-36, November, 1995
6. T. Shintani and M. Kitsuregawa "Hash Based Parallel Algorithms for Mining Association Rules". In *IEEE Fourth International Conference on Parallel and Distributed Information Systems*, pp. 19-30, December 1996.