Tree Structure based Parallel Frequent Pattern Mining 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. Frequent pattern mining has become a fundamental technique for many data mining tasks. Many modern frequent pattern mining algorithms such as FP-growth adopt tree structure to compress database into on-memory compact data structure. Recent studies show that the tree structure can be efficiently mined using frequent pattern growth methodology. Higher level of performance improvement can be expected from parallel execution. In particular, PC cluster is gaining popularity as the high cost-performance parallel platform for data extensive task like data mining. However, we have to address many issues such as space distribution on each node and skew handling to efficiently mine frequent patterns from tree structure on a shared-nothing environment. We develop a framework to address those issues using novel granularity control mechanism and tree remerging. The common framework can be enhanced with temporal constrain to mine web access patterns. We invent improved support counting procedure to reduce the additional communication overhead. Real implementation using up to 32 nodes confirms that good speedup ratio can be achieved even on skewed environment.

1 Introduction

Frequent pattern is defined as the pattern, which can be a set of items or a sequence, that occurs together in a database frequent enough to satisfy a certain minimum threshold [1, 3]. The frequent pattern is important since it indicates a certain regularity in the database. The frequent pattern mining is also the key step in many data mining tasks such as association rule mining, sequence pattern mining etc.

Recently a paradigm has become a new trend in the field of frequent pattern mining research. The paradigm is often called pattern growth or divide-and-conquer. The main idea of the paradigm is the projection of database into a compact on-memory data structure and then use divide-and-conquer method to extract frequent patterns from the data structure. One of the most successful pioneering algorithms is FP-growth [5]. FP-growth devises a data structure called FP-tree that collects all information required to mine frequent patterns.

The need for faster frequent pattern mining has motivated the development of parallel algorithms [8, 2, 10, 7]. In addition, parallel platforms have become more affordable. In particular, PC cluster that composed of commodity components is a promising high cost-performance platform. However the development of parallel algorithms for PC cluster has to face many hurdles such as the relatively

poor network latency and bandwidth, and memory constraint. Recently some works also address the heterogeneity of PC cluster configuration [6]. Since the life cycle of PCs has been becoming very short, upgrading a PC cluster means adding newer generation of PCs to existing configuration. A parallel execution on such mixed environment needs adaptive parallel algorithm to balance the workload.

Here we report the development of tree structure based parallel frequent pattern mining algorithm for parallel execution on heterogenous PC cluster. We also address some methods for load balancing and better memory space utilization since complex data structures like tree tend to increase processing skew. Other contribution is that our framework of parallelization can be easily adapted for other tree structure based algorithms. We show that by the implementation of parallel web access pattern mining [4, 9]. Optimization to reduce the intercommunication is also proposed.

2 FP-growth

The FP-growth algorithm can be divided into two phases : the construction of the FP-tree and mining frequent patterns from the FP-tree [5].

2.1 Construction of FP-tree

The construction of the FP-tree requires two scans of the 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 the FP-tree.

First, the transactions are reordered according to the F-list, while non-frequent items are stripped off. Then reordered transactions are inserted into the FP-tree. If the node corresponding 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 one.

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

2.2 FP-growth

Input to the FP-growth algorithm are the 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.

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 that co-occur with the item. Then FP-growth creates 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 contain the item are discovered.

3 Parallel execution of FP-growth

From Lemma 1, it is natural to consider the processing of conditional pattern base as the execution unit for the parallel processing.

Lemma 1. *The processing of a conditional pattern base is independent from other conditional pattern base.*

Proof. From the definition of the conditional pattern base, item a's conditional pattern base generation is only determined by existence of the item a in the database. The conditional pattern base is generated by traversing the node-links. The resulting conditional pattern base is unaffected by node-links of other items since node-link connects nodes of same item only and there is no node deletion during the processing of the FP-tree. □

3.1 Trivial parallelization

The basic idea is that each node accumulates a complete conditional pattern base and processes it independently to the completion before receiving other conditional pattern bases. Pseudo code for this algorithm is depicted in Fig. 2 (right) and the illustration is given in Fig. 1.

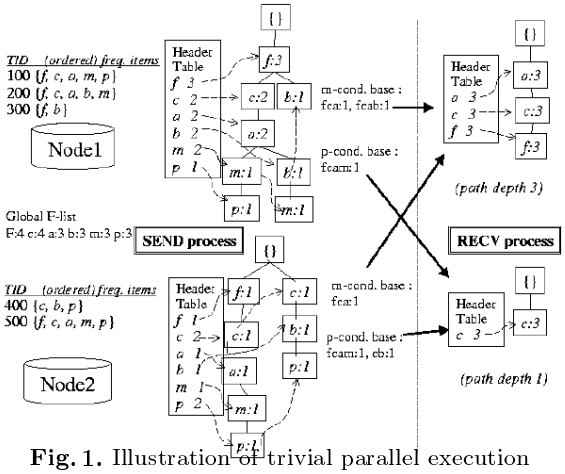


Fig. 1. Illustration of trivial parallel execution

Basically, we need two kind of processes : SEND process and RECV process. After the first scan of transaction database, SEND process exchanges the support count of all items to determine globally frequent items. Then each node builds F-list since it also has global support count. Notice that each node will have the identical F-list. At the second database scan, SEND process builds a local FP-tree from the local transaction database with respect to the global F-list.

From the local FP-tree, local conditional pattern bases are generated. SEND process use hash function to determine which node should process it, instead of processing conditional pattern base locally. The RECV process at the destination node will collect the conditional pattern bases from all SEND processes and then executes the FP-growth. We can do this because of the following lemma :

Lemma 2. *Accumulation of local conditional pattern bases results in the global conditional FP-tree.*

Proof. When a prefix-path in the global conditional FP-tree already exists, the counts of prefix-paths in the other conditional pattern bases are simply added to the path in the tree. Thus the final global conditional FP-tree is not affected by how the prefix-paths are contained in the conditional pattern bases. □

The lemma also automatically leads to the following lemma that will be useful when we are discussing how to handle the memory constraint.

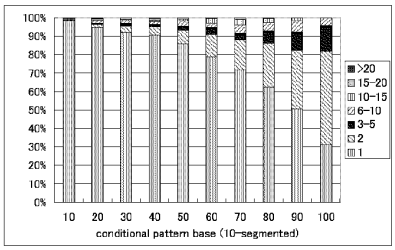
Lemma 3. *The way to split the branches of FP-tree does not affect the resulting conditional pattern bases.*

3.2 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.

Lemma 4. *Path depth can be calculated when creating FP-tree.*

Proof. FP-tree contains all the frequent patterns, thus the depth, the distance of a node from the root, of an item is also preserved in the tree structure. Thus, it can be collected during or after the generation of FP-tree. □




```
input : database D, items I,
        minimum support min_supp;
SEND process :
{
1:local_support = get_support(D,I);
2:global_support = exchange_support(local_support);
3:FList = create_flist(global_support, min_supp);
4:FPtree = construct_fptree(D, FList);
;exchange conditional pattern base
5:forall item in FList do begin
6:    cond_pbase = build_cond_pbase(FPTree, item);
7:    dest_node = item mod num_nodes;
8:    send_cond_pbase(dest_node, cond_pbase);
9:end
}
RECV process :
{
1:cond_pbase = collect_cond_pbase();
2:cond_FPtree = construct_fptree(cond_pbase, FList);
3:FP-growth(cond_FPtree, NULL);
}
```

Fig.2. Path depth distribution (left) Trivial parallel pseudo code (right)

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.*

Example In Fig. 1, the longest pattern that satisfies the minimum support count when processing m's conditional pattern base is <acf> then the path depth of m's conditional pattern base is three.

Typical path depth distribution of conditional pattern base is given in Fig. 2 (left). When the items following F-list order are divided into ten equal segments, the vertical axis represents the distribution of path depth for each segment. For example if F-list contains 100 items from 1 to 100, the third segment contains items range from 71 to 80. Most of conditional pattern base have small path

depth, but some have very large path depth. Since the granularity differs greatly, many nodes with smaller granularity will have to wait busy nodes with large granularity.

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. Note that at each iteration, the path depths of subsequent conditional pattern bases are decremented by one.

Therefore, 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 pattern 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. Since node with heavy processing load can split the load and disperses it to other nodes, path depth approach can also absorb the processing skew among nodes to some extent. Complete pseudo code of this mechanism is depicted in Fig. 3 (left).

```
SEND process :
{
1:cond_pbase = get_stored_cond_pbase();
2:if(cond_pbase is not NULL) then
3:  send_cond_pbase(cond_pbase);
4:end if
}
RECV process :
{
1:cond_pbase = get_stored_cond_pbase();
2:if(cond_pbase is NULL) then
3:  cond_pbase = receive_cond_pbase();
4:end if
5:cond_FPtree = construct_fptree(cond_pbase, FList);
6:FP-growth(cond_FPtree, cond_pbase.itemset);
}
procedure FP-growth(FPtree, X);
input : FP-tree Tree, itemset X;
{
1:for each item y in the header of Tree do {
2:  generate pattern Y = y U X with
    support = y.support;
3:  cond_pbase = construct_cond_pbase(Tree, y);
4:  if (cond_pbase.path_depth < min_path_depth) then
5:    Y-Tree = construct_fptree(cond_pbase,Y-FList);
6:    if (Y-Tree is not NULL) then
7:      FP-growth(Y-Tree, Y);
8:    end if
9:  else
10:    store_cond_pbase(cond_pbase, Y);
11:  end if
12:end for
}
```

```
construct_fptree(database D, flist FList)
input : database D, F-list FList;
output : FP-tree FPtree;
{
1:while not eof(D) do
2: line = read_trans(D);
3: o_trans = get_ordered_trans(line);
4: if available_memory() <  mem_lim then
5:    mig_item_lim = next_branch_migration(FPtree);
6:    dest_node = hash(mig_item_lim);
7:    migrate_tree_branch(dest_node, FPtree);
8: end if
9: if head_of(o_trans) > mig_item_lim then
10:    dest_node = hash(head_of(o_trans));
11:    migrate_trans(dest_node, o_trans);
12: else
13:    insert_fptree(FPtree, o_trans);
14: end if
15: receive_trans(o_trans);
16: if (o_trans is not NULL)
17:    insert_fptree(FPtree, o_trans);
18:end while
19:mig_plan = balancing_tree_size(FPtree);
20:reallocate_branch(FPtree, mig_plan);
}
```

Fig.3. Pseudo code : path depth (left) remerging tree (right)

3.3 Memory constraint

As the data is getting bigger, one of the first resources to get exhausted is local memory. Distributed memory helps alleviate. However, the distribution of FP-tree over nodes is also accompanied by space overhead since some identical prefix-paths are redundantly created at different nodes.

We can eliminate the space redundancy if the branches of FP-tree are remerged, i.e. each branch of FP-tree is allocated exactly to only one node. Lemma 3 guarantees that we can safely modify the way to split FP-tree.

As depicted in Fig. 3 (right), when we detect that the available memory drops below *mem_lim* threshold, we pick next larger branch in FP-tree for migration. Here the notion branch means all prefix paths with the same root item. The variable *mig_item_lim* now holds the last migrated item. The destination node is

decided with hash function. Then we migrate the whole branch to the destination node. The destination node receives all prefix paths with the same root item from all nodes. In other word, the branches of the item are *remerged* from all local FP-trees to a particular destination node.

Then while reading the transaction database, if the head of the ordered transaction *o_trans* is one of the migrated items, the ordered transaction is sent to appropriate node, otherwise it is inserted to the local FP-tree.

At the end, we have to examine whether the size of local FP-trees is fairly balanced among the nodes. If there is significant skew of FP-tree size distribution, a migration plan of FP-tree branches is derived to balance the skew.

4 Mining Web Access Patterns

Essentially, a web access pattern is a sequential pattern in a large set of parts of a web log, which is pursued frequently by users. The problem of mining sequential patterns was first addressed by AprioriAll algorithm [4].

Here our parallel algorithm is based on WAP-mine. WAP-mine also uses a tree data structure called WAP-tree to register access sequences and corresponding counts compactly [9].

With little modification described here, our common framework for tree data structure can be applied for parallel execution of WAP-mine.

4.1 WAP-mine

Since the order of the sequence has to be preserved, WAP-mine does not need to sort the items, thus it does not create F-list. WAP-mine also differs from FP-growth in the way of support counting.

In a web access sequence, there are many occasions where the same page is visited more than once. For example in the sequence <abacad>, the page *a* is visited three times. However the support of the item in the sequence is only one.

To avoid double counting, WAP-mine devises an unsubsumed count property. For each prefix sequence of item *a* with count c, inserted into *a*-conditional pattern base, all of its sub-prefix sequences are inserted also but with count -c.

4.2 Intercommunication Reduction

The unsubsumed count property correctly counts the support of items in the sequences, the size of conditional pattern base increases linearly with the number of the item duplications in the sequences. Since our parallelization scheme exchanges the conditional pattern bases, a lot of data has to be sent through the network.

To reduce the size of conditional pattern bases, we use a *direct count decrement* method. While traversing the path from the node to the root in order to collect the prefix-sequence of item *a* with count c, we simply decrement c from the count of the duplicate nodes directly. The duplicate node here is the node on the path with the same item label *a*. Thus we do not have to generate the sub-prefix sequences. In addition, if the count of a duplicate node becomes zero, we do not have to generate the conditional pattern base starting from the node.

Our method significantly saves both time to generate the conditional pattern base and the required network bandwidth. We omit the proof of the correctness and efficiency analysis here due to the space limitation.

5 Implementation and Performance Evaluation

Here we give the detail of implementation and the results of experiments.

5.1 Implementation

As the shared nothing environment for this experiment we use a PC cluster of 47 nodes that are interconnected by a 100Base-TX Ethernet Switch. There are two types of configuration : (A) older 32 nodes have 800MHz Pentium III and 128 MB of main memory, and (B) newer 15 nodes have a 1.5 GHz Pentium4 and 384 MB of main memory. The operating system is Solaris 8. The programs are developed in C using native socket library.

Three processes are running on each node :

(1) SEND process

creates FP-tree, and sends conditional pattern bases.

(2) RECV process

receives the conditional pattern bases, and processes conditional pattern bases after exchanging phase finishes.

(3) EXEC process

processes the conditional pattern bases in background during the exchanging phase.

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

5.2 Performance Evaluation

For performance evaluation, we use several types of datasets. However due to the space constraint, we only pick some representative results.

We use synthetically generated datasets as described in Apriori and Apriori-All papers[3, 4]. The web sequences are treated as customer sessions. The term transaction here is used also to represent a sequence. The following notions are used for data generation : the number of transactions (D), the number of items (i), the average transaction size (T) and the average of maximal potentially frequent itemset (I). For example T25.I20.D100K.i10K means that in the dataset, number of transactions in the dataset is set to 100K with 10K items. And the average transaction size and average maximal potentially frequent itemset size are set to 25 and 20 respectively.

Comparison with HPA First we compare the performance of our parallel FP-growth with that of Hash Partitioned Apriori(HPA), a variant of parallel Apriori with better memory utilization schema[10]. HPA improves the Apriori by partitioning the candidate patterns among the member nodes in the cluster using hash function. To count the support of all candidate itemsets, each node reads its local transactions data and applies the same hash function to determine where to send the item combinations.

The dataset is T15.I4.D1M.i5k. he experiments are conducted on 2, 4, 8, and 15 nodes of configuration B. The execution time for minimum supports of 0.03% and 0.05% are shown in Fig. 4 (left). Notice that the execution time (y-axis) is log scale. The results show that the parallel FP-growth outperforms HPA at least an order of magnitude for this dataset.

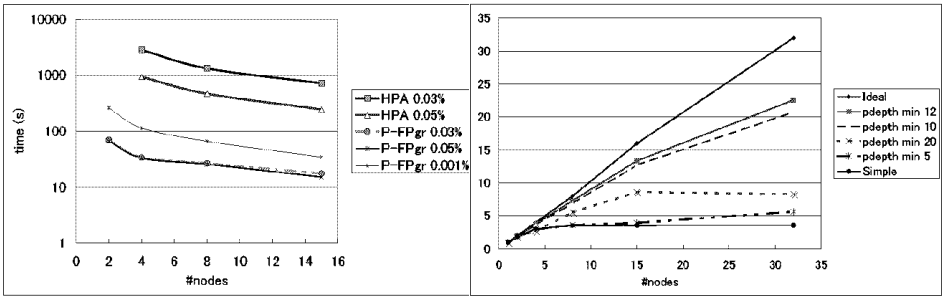


Fig.4. Execution time T15.I4.D1M.i5k(left) Speedup ratio T25.I20.D100K.i5K 0.1%(right)

Speedup ratio We also measure the speedup ratio that is the ratio of performance gain when more processing nodes are used. Here we show the results with up to 32 nodes of configuration A. The dataset is T25.I20.D100K.i5K.

Fig. 4 (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 four 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 "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 eight 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 works.

Heterogeneous Environment We set up a heterogeneous environment with three nodes of the configuration B and one node of the slower configuration A. We also allocate more data on the slowest node to mimic data skew. 70.000 transactions of T25.I10.D100K.i10K dataset are stored in that node while the rest 30.000 transactions are distributed evenly among other nodes.

The execution trace in the left part of Fig. 5 (left) shows that without path depth, the faster nodes are left idle before the node pc025 (the bottom graph) finishes. After employing the path depth adjustment, we get a more balanced execution as shown in right part of Fig. 5 (left). The overall time is improved from 400s to 317s. The minimum path depth here is 12.

Remerging FP-tree Here we show that remerging FP-tree such as depicted in subsection 3.3 can help relaxing the memory constraint of parallel FP-growth. It is important since if on memory FP-tree consumes too much memory space, further processing will be inflicted by memory thrashing. Figure 5 (right) shows the total number of nodes in the all FP-trees relative to those in single node.

The horizontal axis represents the number of processing nodes while the vertical axis represents the memory redundancy. For this experiment, we remerge all branches of FP-tree except the largest one whose root is the most frequent item. The dataset is T25.I20.D100K.i10K with minimum support 0.1%.

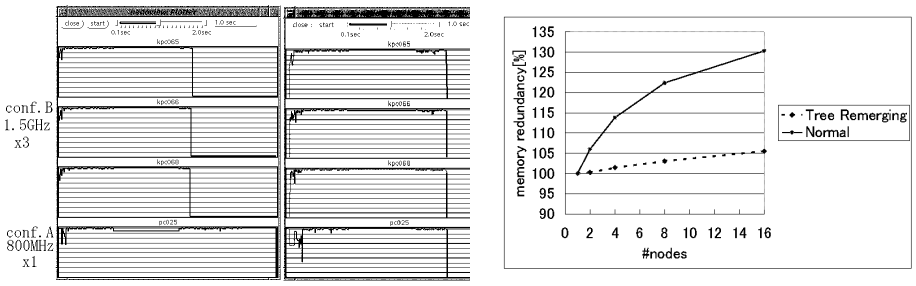


Fig.5. Execution trace, heterogeneous (left) Global memory consumption(right)

One can infer that without remerging FP-tree, 30% more memory is needed by all nodes for parallel execution with 16 nodes. When we employ the remerging FP-tree only 5% of memory is wasted globally. One can also observe that the space saving is becoming more significant when the number of processing nodes is increasing. Thus, the remerging FP-tree potentially can leverage the execution of very large scale data on massive array of processing nodes.

Mining Web Access Patterns We have varied the minimum support to see how it affects performance. The experiments are conducted on 2, 4, 8 and 15 nodes of configuration B with dataset T20.I10.D1M.i10K. The execution time for minimum support of 0.0003%, 0.00005%, 0.001% and 0.003% is shown in Fig. 6 (left).

Figure 6 (right) shows that good speedup ratio is achieved for all minimum support. When the minimum support is set to 0.0003%, 15 processing nodes gain 13.6 times faster performance. Our algorithm suitably balances the processing load and keeps the intercommunication overhead low. When the minimum support is lower, the overhead is relatively smaller, thus the speedup is improved.

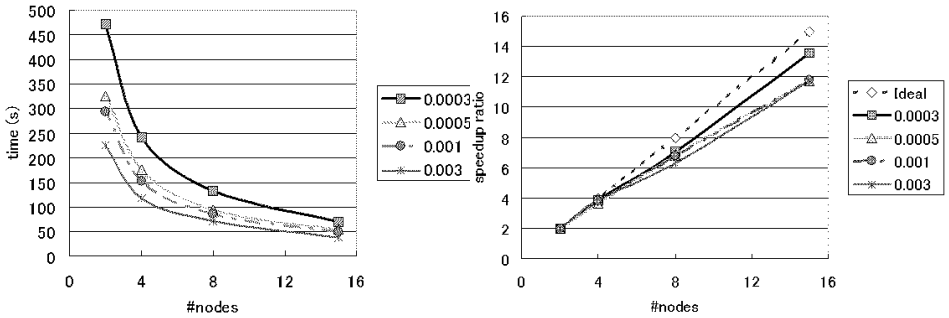


Fig.6. Execution time (left) Speedup ratio (right) for T20.I10.D1M.i10K

6 Conclusion

The development of tree structure based parallel algorithm has to address several issues. We have used the FP-growth as the underlying algorithm and proposed *path depth* based mechanism to break down the granularity of parallel processing. The parallel execution on a PC cluster proves that it is much faster than other parallel algorithm called HPA and it also achieves sufficient speedup ratio up to 32 nodes.

Skew handling is also important on heterogenous environment where the configuration of each node is different. We have shown that the path depth can handle such mixed configuration.

We also showed that the way to partition the FP-tree among nodes does not affect the final result of FP-growth. Based on the observation we proposed a method to remerge FP-tree among nodes to save global memory consumption. In particular, parallel execution with many processing nodes will be benefited from remerging FP-tree.

Our parallel framework has the potential to enhance the performance of other pattern-growth paradigm based modern algorithms. We have shown that mining web access patterns can be implemented efficiently using the framework. We are planning to implement other algorithms on our framework.

References

[1] R. Agrawal, T. Imielinski, A. Swami. "Mining Association Rules between Sets of Items in Large Databases". In *Proc. of the ACM SIGMOD Conference on Management of Data*, 1993.

[2] R. Agrawal and J. C. Shafer. "Parallel Mining of Associaton Rules". In *IEEE Transaction on Knowledge and Data Engineering*, Vol. 8, No. 6, pp. 962–969, December, 1996.

[3] R. Agrawal and R. Srikant. "Fast Algorithms for Mining Association Rules". In *Proceedings of the 20th Int. Conf. on VLDB*, pp. 487–499, September 1994.

[4] R. Agrawal and R. Srikant. "Mining Sequential Patterns". In *Proc. of International Conference of Data Engineering*, pp. 3–14, March 1995.

[5] J. Han, J. Pei and Y. Yin "Mining Frequent Pattern without Candidate Generation" In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2000

[6] K. Goda, T. Tamura, M. Oguci, and M. Kitsuregawa "Run-time Load Balancing System on SAN-connected PC Cluster for Dynamic Injection of CPU and Disk Resource" In *Proc of 13th Int. Conf. on Database and Expert Systems Applications (DEXA'02)*, 2002

[7] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri "Adaptive and Resource-Aware Mining of Frequent Sets" In *Proc. of the Int. Conf. on Data Mining*, 2002

[8] J.S.Park, M.-S.Chen, P.S.Yu "Efficient Parallel Algorithms for Mining Association Rules" In *Proc. of 4th Int. Conf. on Information and Knowledge Management (CIKM'95)*, pp. 31–36, November, 1995

[9] J. Pei, J. Han, B. Mortazavi-asl and H. Zhu "Mining Access Patterns Efficiently from Web Logs" In *Proc. of fourth Pacific-Asia Conference in Knowledge Discovery and Data Mining(PAKDD'00)*, 2000.

[10] T. Shintani and M. Kitsuregawa "Hash Based Parallel Algorithms for Mining Association Rules". In *IEEE Fourth Int. Conf. on Parallel and Distributed Information Systems*, pp. 19–30, December 1996.