Shared Nothing Parallel Execution of FP-growth

Pramudiono IKO† and Masaru KITSUREGAWA†

† Institute of Industrial Science, The University of Tokyo
Komaba 4–6–1, Meguro-ku, Tokyo, 153–8505 Japan

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 the parallelization of such special data structure is difficult, in particular for shared nothing parallel machines such as PC cluster. Here we report how we utilize a characteristic called "path depth" to balance the workload among processing nodes to obtain sufficient parallel speedup ratio.

Key words frequent pattern mining, parallel processing, PC cluster

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. Here we examine the bottlenecks of the parallelization and also method to balance the execution efficiently on shared-nothing environment.

There is also a potential problem when FP-tree can not fit into the memory. As far as we know, there is no report about the scalability of FP-growth when handling very large transaction data. We will examine how to save memory consumption of FP-tree during execution with many nodes based on our elaboration of the characteristics of FP-growth. Our method reveals the potential of parallel execution of FP-growth not only to leverage the performance but also the scalability.

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 that 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] [7]. A better memory utilization schema called Hash Partitioned Apriori (HPA) was proposed in [8]. It is also extended to handle generalized association rule mining [9].

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].

H-mine has been proposed to address certain type of transaction database called *sparse data* where the performance of FP-growth deteriorates since FP-tree becomes too bushy [6]. H-mine can be switched with FP-growth during the execution.

Recently some algorithms that improve FP-growth by eliminating the generation of conditional pattern bases are proposed such as OpportuneProject and PP-mine [5] [10].

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 the same nodes. 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 1.

FP-tree also has a frequent-item header table that holds head of node-links, which 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 that 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 is 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 as shown in Lemma 1, it is natural to consider it 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 0 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.

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.

Pseudo code for this algorithm is depicted in Fig. 1 and the illustration is given in Fig. 2.

```
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);
}
```

Figure 1 Pseudo code 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 local FP-tree from local transaction database with respect to the global F-list.

From the local FP-tree, local conditional pattern bases are generated. Instead of processing conditional pattern base locally, SEND process use hash function to determine which node should process it. 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 base 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.

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

The lemma will be useful when we are discussing how to handle the memory constraint. Although generally the size of conditional pattern bases decreases rapidly, we have to make sure that the processing of conditional pattern base will not overflow the memory of the node.

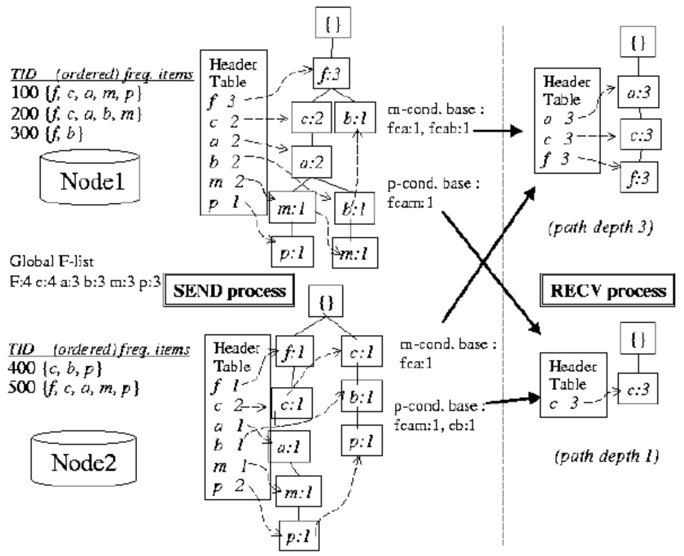

Figure 2 Illustration of trivial parallel execution

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.

In particular, although the trivial approach uses random distribution of conditional pattern bases, the time to process each conditional pattern base could vary.

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. 2 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.

[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.

Notice that path depth is similar with the term "longest pass" in Apriori based algorithms.

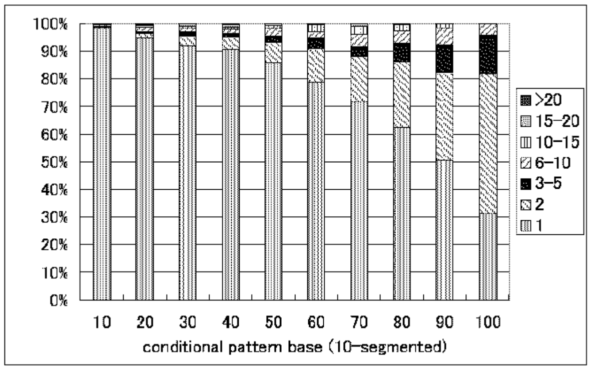

Figure 3 Typical distribution of path depth (T25.I20.D100K 0.1%)

Typical path depth distribution of conditional pattern base is given in Fig. 3. 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. The ratio of small path depth also decreases along the items in F-list.

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. 5 (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. 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 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. 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. 4

```
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
}
```

Figure 4 Pseudo code to balance granularity with path depth

After employing the path depth adjustment we get a more balanced execution as shown in Fig. 5 (right).

4.3 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.

After having the global F-list, distributing reordered transactions to the nodes according to their header item results no duplication of prefix path. However, some branches, particularly whose root items have large count, tend to be very

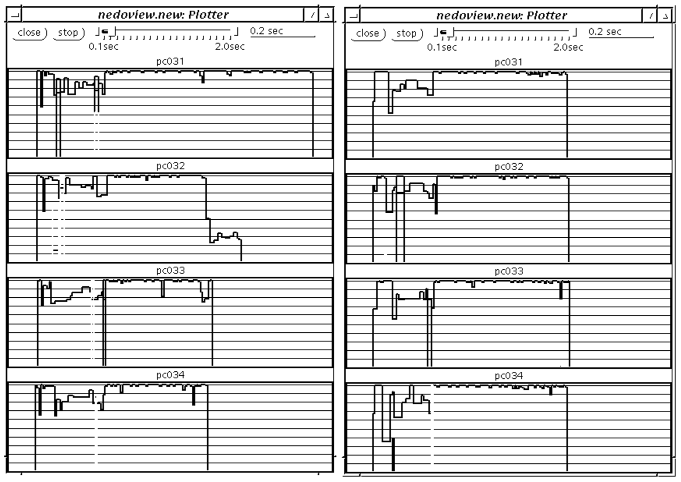

Figure 5 Trivial execution (left) with path depth (right) (T25.I10.D100K 0.11%)

large and may not fit to memory of any single node. Other consideration is the overhead of communication to exchange the transactions. So one has to select branches for remerging to get the optimal trade off between space saving and the execution time.

Here is our approach to make better memory utilization when creating local FP-trees.

As depicted in Fig. 6, 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.

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. Based on the plan, some branches of local FP-tree are relocated to other nodes.

4.4 Optimization of conditional base exchange phase

Our observation shows that the bottleneck of parallel FP-growth execution is the phase of conditional base exchanging. When the number of nodes is small, the CPU utilization ratio is limited by the network bandwidth. However, if many

```
construct_fptree(database D, flist FList)
input : database D, F-list FList;
output : FP-tree FPtree;
{
while not eof(D) do
 line = read_trans(D);
 o_trans = get_ordered_trans(line);
 if available_memory() $<$  mem_lim then
    mig_item_lim = next_branch_migration(FPtree);
    dest_node = hash(mig_item_lim);
    migrate_tree_branch(dest_node, FPtree);
 end if
 if head_of(o_trans) > mig_item_lim then
    dest_node = hash(head_of(o_trans));
    migrate_trans(dest_node, o_trans);
 else
    insert_fptree(FPtree, o_trans);
 end if
 receive_trans(o_trans);
 if (o_trans is not NULL)
    insert_fptree(FPtree, o_trans);
end while
mig_plan = balancing_tree_size(FPtree);
reallocate_branch(FPtree, mig_plan);
}
```

Figure 6 Pseudo code to reduce memory consumption of local FP-trees

nodes are employed, much CPU time is wasted because of the blocking.

Original FP-growth algorithm states that the conditional pattern bases of FP-tree are processed following the item ordering in F-list. The round robin fashion of conditional pattern base exchanging has potential for blocking since all nodes follow the same order to decide the destination node for sending the conditional pattern base. When a receiving node is busy, nodes that are sending to the node are prevented from doing any other work before the sending process completes.

We can make several optimizations since Lemma 1 assures that the processing order has no effect on the result. Our observation on path depth distribution as shown in Fig. 3 also indicates that conditional pattern bases with small path depth are more likely found with the least frequent item in the F-list. Therefore, if following conventional processing order, many nodes stay idle after quickly finishing conditional pattern bases with small path depth.

- ordering of conditional base processing

Based on our observation, we employ the reversed order to process the conditional pattern base. Thus, we process the most frequent item in the F-list first.

- destination node on each round

When processing the conditional pattern bases we make sure that every nodes will not receive other conditional pattern base before all other nodes receive their own. We call the cycle as a round. Instead of using round robin scheme to decide the destination node of a conditional pattern base for each round, we employ a random selection to reduce the possibility of blocking.

- buffer

Needless to say, bigger receive buffer will reduce the possibility of blocking. However, if the size of the buffer is too large, little space left for further processing of FP-tree. Optimal buffer size is determined by some heuristics based on available memory.

- background process

Since the network bandwidth can limit the entire processing; we make a separate background process to handle conditional pattern bases during exchange phase.

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.

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. The number of transactions in the dataset is set to 100K with 10K items.

5.2.1 Execution time

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. 7 The best time of 40 seconds is achieved when minimum path depth is set to 12 using 32 nodes. On single node, all experiments require 904 seconds in average.

5.2.2 Speedup ratio

Fig. 8 shows that path depth greatly affects the speedup achieved by the parallel execution. The trivial parallelization, denoted by “Simple”, performs worst since almost no

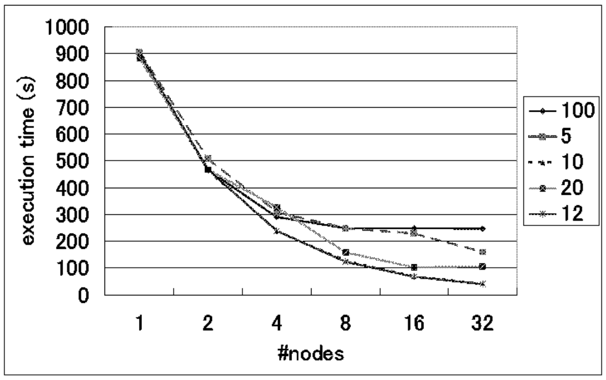

Figure 7 Execution time (T25.I20.D100K 0.1%)

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.

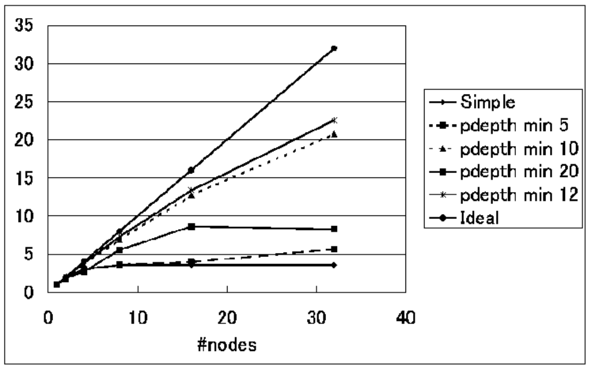

Figure 8 Speedup ratio (T25.I20.D100K 0.1%)

5.2.3 Execution trace

We have developed a tool to monitor the parallel execution on the PC cluster. The execution trace of parallel FP-growth is shown in Fig. 9. The figure shows CPU resource usage, path-depth, interconnection network (send/receive) and the number of conditional pattern bases. Horizontal axis is elapsed time. The vertical axis for the top graph denotes CPU utilization ratio for overall process and EXEC process. The second graph denotes the path depth of the conditional pattern base and currently processing pattern length. The third graph denotes data transfer throughput in MB/s for interconnection network. The network throughput is divided into two parts, send throughput and receive throughput. The fourth graph shows the number of conditional pattern bases. There are three kind of such information : those currently stored in the node, total sent to other nodes, total received from other nodes.

One can observe that the biggest lost of CPU utilization ratio when the conditional pattern bases are exchanged among nodes in the beginning. Thus, the optimization of this phase is important to improve the speedup ratio. The effect of the optimizations, which are discussed in subsection 4.4., can be confirmed in Fig. 10

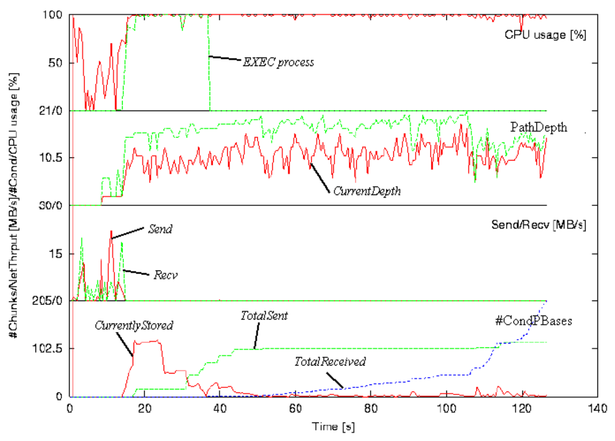

Figure 9 Execution trace (T25.I20.D100K 0.1% 16nodes)

5.3 Remerging FP-tree

Here we show that remerging FP-tree such as depicted in subsection 4.3 can help relaxing the memory constraint of parallel FP-tree. It is important since if on memory FP-tree consumes too much memory space, further processing will be inflicted by memory thrashing. Fig. 11 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.

One can infer that without remerging FP-tree, 30% more

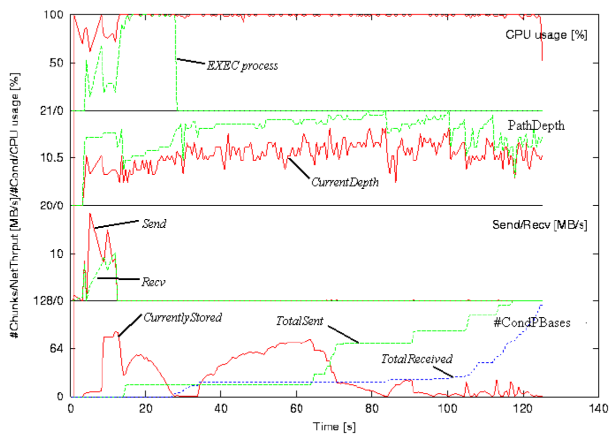

Figure 10 Optimized conditional pattern base exchanging (T25.I20.D100K 0.1% 16nodes)

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.

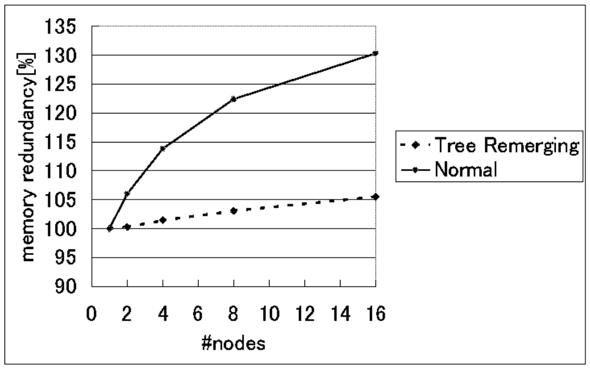

Figure 11 Global memory consumption of FPtree after remerging

6. Conclusion

We have reported the development of parallel algorithm of 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 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 algorithm. We are planning to implement algorithms such as H-mine and OppotuneProject on our framework.

Another issue is the load balancing when the extreme skew exists in the data. We would like to examine how good path depth approach absorbs such skew. Skew handling is also important on heterogenous environment where the configuration of each node is different.

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 Int. Conf. on VLDB*, pp. 487–499, September 1994.

[3] 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.

[4] 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

[5] J. Liu, Y. Pan, K. Wang, J. Han “Mining Frequent Item Sets by Opportunistic Projection“ In *Proc. of the ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2002

[6] J. Pei, J. Han, H. Lu S. Nishio, S. Tang and D. Yang ”H-Mine : Hyper-Structure Mining of Frequent Patterns in Large Databases” In *Proc. of Int. Conf. on Data Mining*, 2001

[7] 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

[8] 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.

[9] T. Shintani, M. Kitsuregawa “Parallel Mining Algorithms for Generalized Association Rules with Classification Hierarchy.” In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pp. 25–36, 1998.

[10] Y. Xu, J.X. Yu, G. Liu, H. Lu ”From Path Tree To Frequent Patterns: A Framework for Mining Frequent Patterns” In *Proc. of Int. Conf. on Data Mining*, 2002.