Tree Partition based Parallel Frequent Pattern mining on Shared Memory Systems

Dehao Chen[1], Chunrong Lai[2], Wei Hu[2],WenGuang Chen[1],
Yimin Zhang[2], and Weimin Zheng[1]

[1]Tsinghua University
Dept. of Computer Science
Beijing, China 100084
chendh05@mails.tsinghua.edu.cn
{cwg, zwm-dcs}@tsinghua.edu.cn

[2]Intel Corporation
Intel China Research Center
No.2 Kexueyuan South Road
Beijing, China 100080
{chunrong.lai,wei.hu,yimin.zhang}
@intel.com

Abstract

In this paper, we present a tree-partition algorithm for parallel mining of frequent patterns. Our work is based on FP-Growth algorithm, which is constituted of tree-building stage and mining stage. The main idea is to build only one FP-Tree in the memory, partition it into several independent parts and distribute them to different threads. A heuristic algorithm is devised to balance the workload. Our algorithm can not only alleviate the impact of locks during the tree-building stage, but also avoid the overhead that do great harm to the mining stage. We present the experiments on different kinds of datasets and compare the results with other parallel approaches. The results suggest that our approach has great advantage in efficiency, especially on certain kinds of datasets. As the number of processors increases, our parallel algorithm shows good scalability.

1. Introduction

Association rule mining searches for interesting relationships among items in a given data set. One of the most famous examples of association rule mining is the market basket problem, which was introduced in [1]. Since the invention of Apriori algorithm [1, 2], a lot of algorithms have been devised to cope with the problem of frequent pattern mining, which is the most time-consuming part of association rule mining process. However, as for extremely large datasets, the currently proposed frequent pattern mining algorithms still consume too much time. One solution is to design more efficient mining algorithms to reduce the repeated I/O scans as well as to minimize the memory requirement and calculating time. So algorithms like kDCI [3], FP-Growth [4], etc, are proposed. Another alternative solution is to parallelize the algorithm.

As the recent development of CMP and SMP architecture, the shared-memory parallel program is becoming more and more popular. The shared memory structure can provide an economic parallel solution with good efficiency and high scalability. The present frequent pattern mining algorithms can be divided into two categories: apriori-like algorithms, which are the implementations of the classical apriori algorithm, and the other ones, which are completely different in structure with the classical apriori algorithm. Among both kinds of algorithms, FP-Growth [4] can achieve good efficiency. It's faster than any of the apriori-like algorithms and it only has to scan the whole database twice.

This paper proposed an approach to parallelize the FP-Growth algorithm on shared-memory structures. Section 2 gives a brief introduction to FP-Growth Algorithm. Section 3 presents related parallelization work of FP-Growth. Section 4 and Section 5 introduced our parallelization algorithm. Section 6 showed the experiment results as well as comparisons with other parallel algorithms.

2. FP-Growth Algorithm

FP-Growth algorithm is based on tree structures. The algorithm can be divided into two steps.

1-4244-0054-6/06/$20.00

2.1. Building FP-Tree

Scan the database twice. The first scan gathers necessary information for the second scan to build the FP-Tree from each line of transaction. The algorithm is shown below:

Algorithm 1 FP-tree construction.

Input: A transaction database DB and a minimum support threshold ξ.

Output: FP-tree, the frequent-pattern tree of DB.

Method: The FP-tree is constructed as follows.

1. Scan the transaction database DB once. Collect F, the set of frequent items, and the support of each frequent item. Sort F in support-descending order as FList, the list of frequent items.

2. Create the root of an FP-tree, T , and label it as "null". For each transaction Trans in DB, do the following.

Select the frequent items in Trans and sort them according to the order of FList. Let the sorted frequent-item list in Trans be [p|P], where p is the first element and P is the remaining list. Call $insert\_tree([p|P], T)$.

The function $insert\_tree([p|P], T)$ is performed as follows. If T has a child N such that N.item_name = p.item_name, then increment N's count by 1; else create a new node N, with its count initialized to 1, its parent link linked to T , and its node_link linked to the nodes with the same item_name via the node_link structure. If P is nonempty, call $insert\_tree(P, N)$ recursively.

2.2. Mining from the FP-Tree

It's an iterative procedure: each step produces a set of conditional pattern base and then calculated together. The algorithm is shown below:

Algorithm 2 FP-growth: Mining frequent patterns with FP-tree by pattern fragment growth.

Input: A database DB, represented by FP-tree constructed according to Algorithm 1 , and a minimum support threshold ξ.

Output: The complete set of frequent patterns.

Method: Call FP_Growth(FP_tree, null), which is shown in Figure 1.

3. Related Work

Among those algorithms proposed to address the problem of mining frequent patterns, one of the key algorithms, which seemed to be the most popular in many applications for enumerating frequent itemsets, is the apriori algorithm [1, 2]. It's the foundation of most known algorithms whether sequential or parallel. Salvatore et al. have proposed Direct Count & Intersect (kDCI) [3] as well as its parallel version (parDCI [5]). However, it requires at least 3 full database scans. FP-Growth [4], which was proposed by Han et al., creates a relatively compact tree-structure that alleviates the multi-scan problem and improved the candidate itemset generation. The algorithm requires only 2 full database scans. Our approach presented in this paper was based on this idea. In spite of the significance of the frequent pattern mining, particularly the generation of frequent itemsets, few advances have been made on parallelizing frequent pattern mining algorithms. Most of the work on parallelizing association rule mining on Shared-memory Multi Processor architecture was based on apriori-like algorithms [5, 6, 7].

Osmar et al. proposed a parallel algorithm [8], which build extra trees for each process. Each thread builds its own FP-Tree from certain part of the database, calculates out the candidate pattern base from its own FP-Tree and then merges the candidate pattern bases together. It can achieve good acceleration, but as the number of threads increases, the total memory required by all FP-Trees increases too, which may cause great overhead and do harm to scalability.

Iko et al. also tested the multi-tree parallel FP-Growth algorithm on PC Clusters, and came across the same problem of extra nodes introduced by multi-tree. Although they proposed a remerging algorithm [9], which can minimize the number of extra nodes needed. However, remerging operation itself consumes much time and new overhead is introduced.

R.Jin, et al. did an experiment [10], building multi-trees with full replication as well as building one tree. They compare the multi-tree solution with one-tree solution and found that one-tree solution has poor scalability. However, their one-tree solution uses locks to solve the consistency problems whenever the tree needs to be expanded. Although this method can avoid the overhead of the extra nodes, the locks make the scalability very poor. In their second approach of parallelization, which uses full replication to reduce locks, they get a speed-up ratio of 4.72 on an 8 threads parallel environment.

```
Procedure FP_Growth(Tree, α)
begin
    /* Mining single prefix−path FP−tree */
    if Tree contains a single prefix path then
        begin
            let P be the single prefix−path part of Tree;
            let Q be the multipath part with the top branching node replaced by a null root;
            for each combination (denoted as β) of the nodes in the path P do
            generate pattern β ⋃ α with support = minimum support of nodes in β;
            let freq pattern set(P) be the set of patterns so generated;
        end
    else let Q be Tree;
    /* Mining multipath FP−tree */
    for each item ai in Q do
        begin
            generate pattern β = a_i ⋃ α with support = a_i.support;
            construct β′s conditional pattern−base and then β′s conditional FP−tree Tree β;
            if Tree β = φ(13) then call FP−growth(Tree β, β);
            let freq pattern set(Q) be the set of patterns so generated;
        end
    return (freq pattern set(P) ⋃ freq pattern set(Q) ⋃ (freq pattern set(P)× freq pattern set(Q)))
end
```

Figure 1. Pseudocode of FP_Growth Procedure

4. Tree Partition Algorithm

4.1. Master/Slave Model

According to [10], if we let each thread to deal with a mount of transactions dynamically, we have to add lock to each node of the tree to make sure of the consistency. This will do great harm to the scalability. In the experiment of [10], the speed-up ratio in 8 threads circumstances will be less than 1. So we introduced our Master/Slave Model.

In the multi-thread environments, we make one thread as the master thread, and the remaining threads as slave threads. The master thread's task is to load each line of transaction from the database and distribute it to each slave threads. Each slave thread has its own transaction queue. It gets a transaction from the queue each time the master thread put one transaction into it, and disposes the transaction to build the tree. Thus the master thread is a producer, which produces transactions for each slave thread to consume.

This model makes it possible for master thread to do some preliminary measures of the transaction before delivering it to the slave thread. According to the results of the preliminary measures, the master thread can also decide to which slave thread the transaction should be delivered.

However, this model cannot eliminate the lock of each node in the tree. The further work is to partition the tree into several independent parts, and distribute them to slave threads.

4.2. Content based tree partition

Now we propose the idea of the content based tree partition algorithm. The most important point is to partition the tree equally so that each thread can get approximately equal workload, i.e. get approximately equal amount of transactions to process. The idea can be illustrated as following. A database DATA contains 8 items: "A", "B", "C", $\cdots$, "G", "H", where "A", "B" and "C" are the first three most frequent items. DATA has totally 80 transactions. Assume we want to partition all transactions according to the content of the first N most frequent items (N can be heuristically determined according to the number of threads available, usually $N = 10$ works well for most databases). Here we set $N = 3$ for simplicity and the transactions are partitioned into $2^3 = 8$ chunks according to items "A", "B" and "C" (a content base chunk in database is equal to a sub-branch in FP-tree, as explained above). See Table 1 for an example of the distribution chunks (e.g. 101 means the chunk contain transactions all have items "A" and "C", but not "B"). If there are two threads available, i.e. need to group the chunks into two groups, a heuristic search algorithm can be employed to group the 8 chunks into two groups, and make each group contain approximate equal number

of transactions. The result can be $\{111, 110, 101, 011\}$ and $\{100, 101, 001, 000\}$, each group contains 40 transactions. Figure 2 shows the partitioned sub-branches and grouping result. Two sub-branches circled with red dashed line are the group assigned to thread 1, and two other sub-branches circled with blue solid line are the group assigned to thread 2.

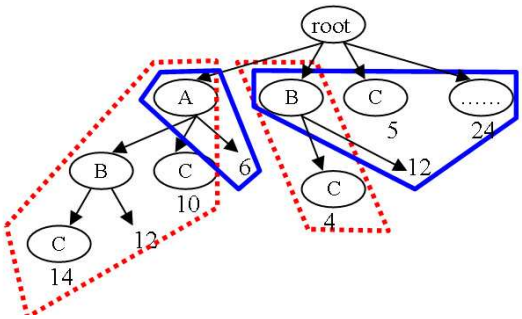

Figure 2. Content based tree partition and grouping

ABC	count
111	14
110	12
101	10
100	6
011	4
010	5
001	5
000	24

Table 1. Transaction Distribution by Contents

In the real world, we have to set N (the number of most frequent items we count) large enough to make the load balanced among each thread. Generally, if we set N between 10 and 12, we will achieve good load-balance. However, we have to divide all transactions into 2^N chunks, and form $T = (Numberofthreads) - 1$ groups to distribute each group to a thread. It will be an NP-Complete problem to get the most optimized result; hence a heuristic method is introduced in the next section.

4.3. Heuristic Algorithm

As described in the previous part, there are $M = 2^N$ chunks of transactions. Assuming V_i is a vector which contains all chunks that belong to group i; C_i is the number of transactions of group i (the size of V_i). Our goal is to distribute them into $T - 1$ groups, making the total number of transactions in every group almost equivalent. It can be formalized as follows:

$$V = V_1, V_2, \cdots, V_{T-1}$$

$$f = \sum_{i=1}^{T-1} \left(C_i - \frac{\sum_{j=1}^{T-1} C_j}{T-1} \right)^2$$

The goal is to find V to make f minimized.

Obviously, it's possible to find the best optimized solution to make the load perfectly balanced. However, it's an NP-Complete problem. So we have to use heuristic algorithms to achieve partially optimized solution.

Algorithm 3 Heuristic Algorithm

Input: $P = \{P_1, P_2, \cdots, P_m\}$

Output: $V = \{V_1, V_2, \cdots, V_{T-1}\}$

Method: Call Find_Distribution(P), which is shown in Figure 3.

```
Procedure Find_Distribution(P)
begin
    Null->V_i, i = 1, 2, ..., T-1
    0- > C_i, i = 1, 2, ..., T-1
    For each P_i do
        Find j, C_j = min(C_k, k = 1, 2, ..., T-1)
        V_j = V_j ⋃ chunk i
        C_j = C_j + sizeof(chunk i)
    Return V
end
```

Figure 3. Pseudocode of Find_Distribution Procedure

The time complexity of the heuristic algorithm is $O(M(T-1))$. In most cases, it will achieve satisfying results and balances the workloads very well.

5. Parallelized FP-Growth

After the first FP-Tree is built, the next step is called FP-Growth, which is to mine the frequent patterns from the tree. The serial version of the mining algorithm is a recursive procedure. So we derived the

first iteration of the recursion from the other iterations and parallelize it.

The loop size of the first iteration depends on the number of frequent items whose count is no less than the minimum support. As for most cases, the loop size is not so large. In order to avoid load-imbalance problems, we use dynamic scheduling. The experiments in Section6 demonstrate that the load is perfectly balanced among threads in the FP-Growth stage.

6. Experiment Results

The experiment is based on 4 datasets. The first two datasets are "Kosarak.dat" and "Accidents.dat", and the other two datasets are selected from "Webdoc.dat". All datasets are available from [11].

6.1. Execution Time

We compared our parallel algorithm with the multi-tree version of parallel FP-Growth [8, 9], which is the most efficient parallel FP-Growth algorithms available.

Tree Building Stage

This stage includes two phases of database scans. After the stage, a complete frequent pattern tree was built in the memory.

Figure 4 shows that the tree-building stage is IO-Intensive for Webdoc dataset. As the number of processors increases, the consumed time doesn't change much. This is because this kind of data in datasets like Webdocs requires only a little processing but much I/O intensive reading from the database. However, Tree-Partition version has a better scalability than the Multi-Tree version. We can see from the figure, as the number of processors increases, the consumed time of Tree-Partition version decreases a little bit while the consumed time of Multi-Tree version increases. This is because as the number of processors increases, the Multi-Tree version has to build more trees, which contains more nodes than Tree-Partition version.

From Figure 5, the scalability of Tree-Partition version is not good for both datasets. This is because both datasets require more data processing than I/O reading. This kind of datasets is mostly made up of large amount of small transactions. The Tree-Partition version has to maintain a task queue to deal with each line of transaction, which may apparently introduce locks to maintain the queue. As the number of processors increases, the locks used to maintain the queue begin to take effect. However, for this kind of datasets, the most time-consuming stage is the mining stage. The next part indicates that the performance

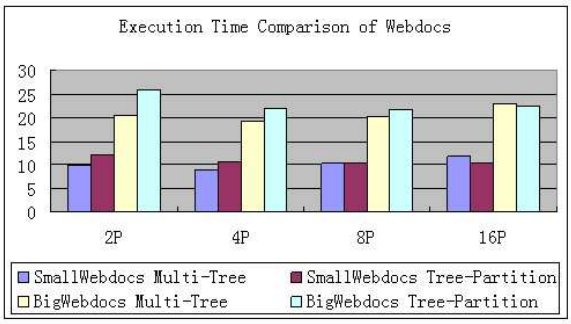

Figure 4. Execution Time Comparison of Tree Building Stage (Webdoc)

of Tree-Partition algorithm over this kind of datasets is much better in mining stage.

Mining Stage

The time consumed by this stage depends on the scale of the tree built in the previous stage. All the processing is done in the main memory.

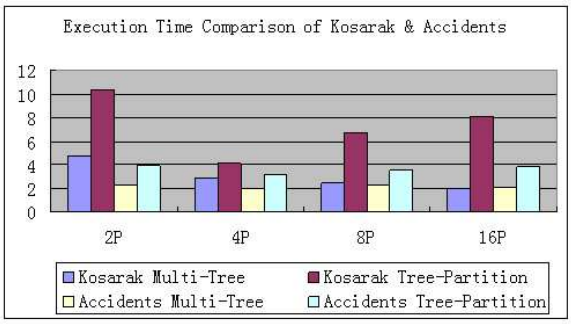

Figure 5. Execution Time Comparison of Tree Building Stage (Kosarak & Accidents)

From Figure 6 and Figure 7, there're great advantages of the Tree-Partition algorithm over the Multi-Tree algorithm. Although both algorithms have good scalability, the Tree-Partition version is 5% to 40% faster than the Multi-Tree version. The main reason is that the Tree-Partition algorithm only maintains one tree in the main memory, while the Multi-Tree version maintains as many trees as the number of processors. The extra trees may introduce extra nodes, which will finally add workload to the algorithm. However, the number of extra nodes is not linear with the number of processors. Figure 8 shows the total number of nodes of multi trees, as a function of number of processors.

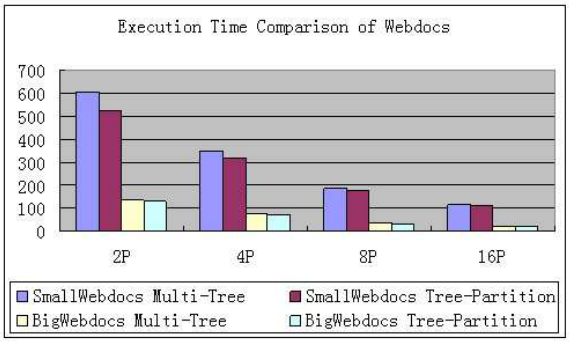

Figure 6. Execution Time Comparison of Mining Stage (Webdoc)

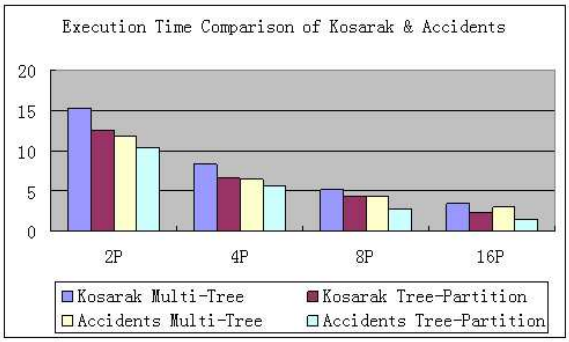

Figure 7. Execution Time Comparison of Mining Stage (Kosarak & Accidents)

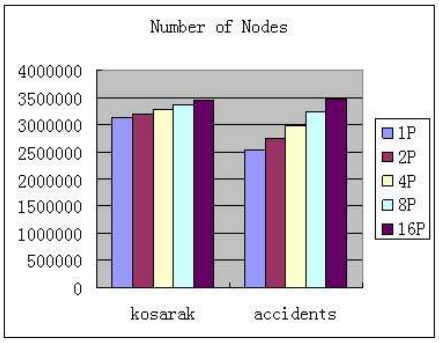

Figure 8. Total Number of Nodes for Multi-Tree version on Kosarak & Accidents

According to our analysis, the number of extra nodes introduced by multi-tree algorithm depends on the datasets. If the datasets are constituted with large amount of similar transactions, that is, transactions with fewer items and more duplicated data, will produce much more extra nodes as the number of processors increases. As a result, this kind of datasets will show bad scalability in multi-tree version. However, this kind of datasets is very commonly used in regular applications like web data analysis and commercial rule mining. In tree-partition version of the algorithm, the total number of tree nodes will not change when the number of threads increases. As a result, the mining stage of tree-partition algorithm shows good scalability in this kind of datasets.

6.2. Speed-Up Ratio

Table 2 shows the comparison of the speed-up ratio of the mining stage between multi-tree version and tree-partition version of the parallel FP-Growth. We can see that tree-partition version gets up to 45% advantage over multi-tree version.

		2P	4P	8P	16P
Tree Partition	smallwebdocs	1.99	3.28	6.12	9.98
	bigwebdocs	1.93	3.54	7.93	14.01
	kosarak	1.86	3.53	5.40	10.38
	accidents	1.72	3.20	6.77	12.48
Multi Tree	smallwebdocs	1.80	3.10	5.81	9.61
	bigwebdocs	1.92	3.51	7.89	12.97
	kosarak	1.88	3.44	5.49	8.29
	accidents	1.72	3.15	4.66	6.88

Table 2. Speed-Up Ratio of the Mining Stage of the two Parallel Algorithms

Figure 9 shows the speed-up ratio of the mining stage of tree-partition version of the parallel FP-Growth. As most of the processing in this stage is done in main memory, the scalability is good and we achieved nearly linear speed-up ratio.

The total speed-up ratio, which is shown in Table 3 and Figure 10, is lower than that of the mining stage. This is because there are still two phases of database scan, which turn out to be I/O intensive and could hardly be parallelized. However, when we face the big datasets like webdocs, the scalability turns out to be good because the mining stage takes up much more time than the I/O scan phases. And our algorithm can efficiently reduce the time spent on mining stage.

Among these 4 datasets, kosarak is a special case. As shown in Table 3, the total speed-up ratio is not

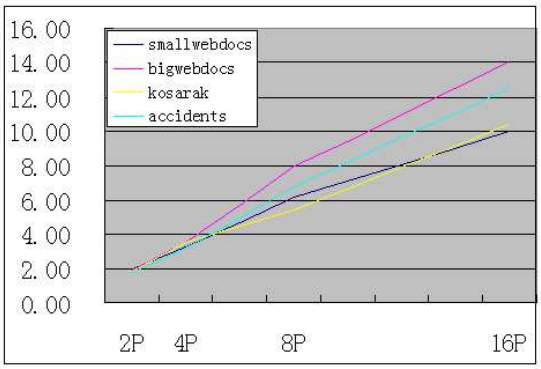

Figure 9. Speed-Up Ratio of the Mining Stage of Tree Partition Parallel Algorithm

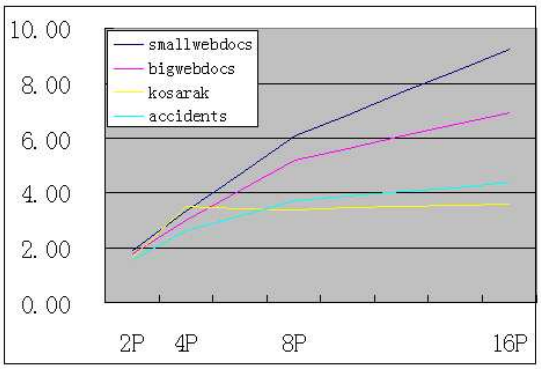

Figure 10. Total Speed-Up Ratio of Tree Partition Parallel Algorithm

good. This is because of the bad performance at the tree-building stage, which had been described in 6.1.1. We consider this kind of datasets as an extreme condition. This is because kosarak consists of large amount of items and short transactions, and most transactions do little contribution to the final association rules mined from the whole database. In most cases, especially when dataset is very large, this kind of extreme condition cannot be satisfied, and our parallel algorithm can achieve good speed-up ratio.

		2P	4P	8P	16P
Tree Partition	smallwebdocs	1.93	3.35	6.07	9.24
	bigwebdocs	1.79	3.01	5.19	6.93
	kosarak	1.64	3.50	3.39	3.63
	accidents	1.62	2.64	3.74	4.38
Multi Tree	smallwebdocs	1.80	3.07	5.58	8.83
	bigwebdocs	1.79	3.00	5.27	6.49
	kosarak	1.88	3.36	4.92	6.97
	accidents	1.64	2.74	3.50	4.68

Table 3. Total Speed-Up Ratio of the two Parallel Algorithms

7. Conclusion

In this paper, we have focused on shared memory parallelization of FP-Growth algorithm. By building one global FP-Tree, a parallel algorithm with good scalability is designed. Our algorithm successfully avoided the overhead introduced when building multi-trees. And the tree partition algorithm can minimized the lock overhead and make the workload perfectly balanced.

The experiment results include consuming time comparison with multi-tree parallel algorithm as well as the speed-up ratio. These experiments established the following:

1. Good efficiency is achieved for the tree-partition based parallel algorithm.

2. The tree-partition based parallel algorithm produces no overhead of extra nodes while multi-tree parallel algorithm produces many redundant extra nodes.

3. Different datasets show different characters. Some datasets requires much more I/O operations than pure processing, so the speed-up ratio will not appear to be so good. However, as the datasets become larger, the performance of the tree-partition based parallel algorithm becomes better.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *19 ACM SIGMOD Conf. on the Management of Data, Washington,DC*, pages 207C–216.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[3] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri. kDCI: a multi-strategy algorithm for mining frequent sets. In *FIMI*, 2003.

[4] Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMODREC: ACM SIGMOD Record*, 29, 2000.

[5] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. An efficient parallel and distributed algorithm for counting frequent sets. In *VECPAR*, pages 421–435, 2002.

[6] P. Becuzzi, M. Coppola, and M. Vanneschi. Mining of association rules in very large databases: A structured parallel approach. In *Euro-Par*, pages 1441–1450, 1999.

[7] T. Shintani and M. Kitsuregawa. Hash based parallel algorithms for mining association rules. In *PDIS*, pages 19–30, 1996.

[8] O. R. Zaïane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *ICDM*, pages 665–668, 2001.

[9] I. Pramudiono and M. Kitsuregawa. Tree structure based parallel frequent pattern mining on PC cluster. In *DEXA*, pages 537–547, 2003.

[10] R. Jin, G. Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *IEEE Trans. Knowl. Data Eng*, 17(1):71–89, 2005.

[11] Frequent itemset mining dataset repository. URL:http://fimi.cs.helsinki.fi/data/