Parallel Association Rule Mining based on FI-Growth Algorithm

Bundit Manaskasemsak 1  Nunnapus Benjamas 2  Arnon Rungsawang 1,4
Athasit Surarerks 3  Putchong Uthayopas 2,4

1 MIKE Lab., Dept. of Computer Engineering, Kasetsart Univ., Bangkok 10900, Thailand. {un, arnon}@mikelab.net
2 HPCNC Lab., Dept. of Computer Engineering, Kasetsart Univ., Bangkok 10900, Thailand. g4885043@ku.ac.th
3 ELITE Lab., Dept. of Computer Engineering, Chulalongkorn Univ., Bangkok 10330, Thailand. athasit@cp.eng.chula.ac.th
4 Thai National Grid Center, Software Industry Promotion Agency, Ministry of Information and Communication Technology, Thailand. putchong_ut@thaigrid.or.th

Abstract
Association rule mining is one of the most important techniques in data mining. It extracts significant patterns from transaction databases and generates rules used in many decision support applications. Many organizations such as industrial, commercial, or even scientific sites may produce large amount of transactions and attributes. Mining effective rules from such large volumes of data requires much time and computing resources. In this paper, we propose a parallel FI-growth association rule mining algorithm for rapid extraction of frequent itemsets from large dense databases. We also show that this algorithm can efficiently be parallelized in a cluster computing environment. The preliminary experiments provide quite promising results, with nearly ideal scaling on small clusters and about half of ideal (15 fold speedup) on a thirty-two processor cluster.

1. Introduction
Association rule mining [1,3], a popular data mining technique, has been deployed in many decision support systems for years. It extracts significant patterns and then generates some interesting rules from historical data, called transaction-oriented databases. Industrial and commercial organizations employ this technique to extract some high co-occurrence items for increasing their sales [7,16]. In medical applications, physicians use it to discover important patterns and predict effects of patients’ diseases [9,14].

Basically, association rule mining consists of two main steps: frequent itemsets generation tries to extract the most frequent patterns, and rule generation uses these frequent patterns to generate interesting rules. Practically, the first step is usually more complex and requires greater time to perform than the second one. Two fundamental algorithms proposed for finding the frequent itemsets from large databases are the Apriori [3] and Closed [12] algorithms. The former was proposed to first find the smallest frequent itemsets (1-itemsets) and then expand to larger itemsets, while the latter performed in the opposite way. However, both algorithms require many database passes. The FP-growth [6] and FI-growth [4] algorithms were proposed to reduce this cost. These algorithms need at most two database scans; but need to fit all data structures in main memory.

Transaction-oriented databases are usually very large. In a large department store, for example, the entire sales transactions of their customers are continually being added. Mining useful rules from such large and volatile databases is a challenging problem. Fast association rule mining inevitably requires large computing resources. We concur with other researchers that cluster computing technology offers a potential solution: e.g., parallel Apriori approach [2,5], parallel FP-growth approach [17,8], etc.
The objective of this paper is to utilize parallelization on a computing cluster environment for fast extraction of frequent itemsets from large dense databases. We propose an alternative approach to parallel association rule mining based on the FI-growth algorithm [4]. We perform data parallelization by first partitioning the large database into several portions and then distributing them to processors to compute the frequent itemsets in parallel. The intrinsic character of the “FI-tree” representation enables each processor to run almost independently, so that we can efficiently parallelize the algorithm. Our experimental results show that the speedup is almost 4 times using four processors, and up to 15 times when using thirty-two processors.

The rest of this paper is organized as follows. Section 2 surveys some related works concerning both sequential and parallel association rule mining. Section 3 provides details of the FI-growth algorithm; while section 4 presents its design and parallel implementation. Section 5 describes the experiments and discusses the results. Finally, section 6 concludes the paper.

2. Related works

The Apriori algorithm [3] has emerged as the best classical algorithm for association rule mining since 1994. Apriori uses a bottom-up approach to generate candidate itemsets of length $k$ from itemsets of length $k-1$. Then it prunes the candidates if they are infrequent enough, i.e., their frequency or support is less than a user-specified minimum support ($\text{min}_s\text{up}$) value. We call these processes the “generate-and-test” paradigm.

Subsequently, Park et al. [10] proposed the direct hashing and pruning (DHP) algorithm, an extension of Apriori, which uses an additional memory for a hash table to hold the approximate supports of candidate 2-itemsets in advance during the first iteration. This hash-based technique can reduce the number of generated candidate itemsets so that it requires less computation in later iterations.

The FP-growth algorithm [6] is a new generation of frequent pattern mining that uses a compressed FP-tree structure for mining a complete set of frequent itemsets without candidate itemset generation. The algorithm is divided into two phases. (1) Construct an FP-tree that encodes the data set by reading the database and mapping each transaction onto a path in the FP-tree, while simultaneously counting the support of each item. (2) Extract frequent itemsets directly from the FP-tree using a bottom-up strategy to find all possible frequent itemsets ending with a particular item; equivalent to the suffix-based approach. This mining phase can run significantly faster if the FP-tree is small enough to fit into main memory. In practice, the more prefix paths are shared, the greater the compression of the FP-tree is. The size of the FP-tree is typically smaller if all items are ordered from highest to lowest support values. However, for very large databases, a lot of time is required to first sort the supports of 1-itemsets. To avoid this overhead, Amphawan and Surarerks proposed the frequent item tree (FI-growth) algorithm [4]. This algorithm constructs an FI-tree representation by ordering the items by sequence in transactions. They also proposed an efficient operation to manipulate the FI-tree that is described in the next section.

To overcome the limitations of scalability (w.r.t. size of databases) and run-time performance of sequential algorithms, most researchers rely on high-performance technologies such as parallel and distributed computing. Most such parallel data mining algorithms were developed from the Apriori-based method. For example, Agrawal and Shafer [2] presented three parallelization approaches for mining association rules: count distribution, data distribution, and candidate distribution. The first approach is simple parallelization. All processors generate the entire list of candidates; thus, each processor independently counts partial supports and then exchanges them with the others. In the second one, each processor generates the disjoint candidate sets and broadcasts them to all others. In the last one, this approach determines a heuristic based on support for partitioning both data and candidates. Thus, each processor can generate disjoint candidates and global counts independently. Zaki [18] also discussed several parallel association rule mining algorithms based on various parallelization schemes.

A few other parallel algorithms should be mentioned here. The PDM algorithm [11] is a parallel version of DHP; each processor determines the global supports of 1-itemsets through an all-to-all broadcast and then approximates counts for 2-itemsets with a hash table. Pramudiono and Kitsuregawa [13] reported results for a parallel FP-growth algorithm on a shared-nothing cluster environment. Recently, efficient database partitioning methods for optimal load-balance and minimum communication overhead have been reported in [17] and [8], respectively.

3. FI-Growth algorithm

Similar to the FP-growth algorithm [6], FI-growth [4] represents the data set as a prefix sharing tree, called an “FI-tree”. The FI-growth algorithm does not
only avoid time consuming operation of sorting the 1-itemsets as in FP-growth, but also provides an efficient operation for FI-tree manipulation. It commonly consists of two phases: FI-tree construction and mining. In the following subsections, we briefly review an FI-tree representation and how it is constructed. We then present the three steps of mining frequent itemsets with an FI-tree; additional details and pseudo-codes are also provided in [4].

3.1. FI-Tree representation

An FI-tree is a memory representation of the data set encoding, where each transaction is mapped onto the FI-tree with prefix sharing paths. In practice, the FI-tree structure consists of two structures in memory: a “header table” that has linkages to each item in an “items-tree”. Constructing an FI-tree requires scanning the database only twice: the first scan creates the header table, and the second scan creates the items-tree.

In order to show how an FI-tree is constructed, we use an example transaction dataset shown in Table 1. The first column in the table is the ordered transaction IDs, while the second one shows the list of items corresponding to the transaction. Note that the items in all lists must be in the same relative order.

Table 1. Example of a transaction database.

<table>
<thead>
<tr>
<th>TID</th>
<th>List of items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a, c, d, e, f</td>
</tr>
<tr>
<td>2</td>
<td>a, b, e</td>
</tr>
<tr>
<td>3</td>
<td>c, e, f</td>
</tr>
<tr>
<td>4</td>
<td>a, c, e, f</td>
</tr>
<tr>
<td>5</td>
<td>c, e, f</td>
</tr>
</tbody>
</table>

FI-tree construction first creates a header table having size corresponding to the number of unique items. During the first scan, each transaction is read to update the support values. Then, we prune items with supports that are less than a pre-defined min_sup, which is taken to be 2 (40%) in our example. During the second scan, FI-tree construction only considers the items that appear in the pruned header table (labeled below as item:count). Each transaction is read and additional paths in the FI-tree are created. The same sequences of items are grouped into the same sup-paths. The FI-tree constructed from the example transaction database is depicted in Figure 1.

Figure 1. The initial FI-Tree structure: the header table is on the left side and the items-tree is on the right side.

3.2. Frequent itemsets generation with FI-Tree

An advantage of the FI-tree is that we can directly generate and discover frequent itemsets without any further scanning of the database. This mining phase is composed of three steps. The first one is a branching step: the FI-tree is branched using an item frequency combination. The second step is subset finding: the algorithm attempts to explore all possible remaining paths in the FI-tree. The last is a pruning step used to eliminate uninteresting results.

3.2.1. Combining operation. In this sub-section, we present a useful operation, called combining operation, to merge two sub-trees, that is, the same sub-paths are grouped and their counts summed. For a new created sub-tree, it will either be added as a new child of root, or replace the old one if the root already has that child item.

Let tree(a) be a tree (or sub-tree) that has a as a root node, and © refer to a combining operation. The combining operation has the following properties.

1) Self-reflective property: tree(a) © tree(a) is equal to tree(a) itself.
2) Commutative property: tree(a1) © tree(a2) is equal to tree(a2) © tree(a1).
3) Associative property: (tree(a1) © tree(a2)) © tree(a3) is equal to tree(a1) © (tree(a2) © tree(a3)).

For example, consider the item “c” of the header table in Figure 1. It has linkages to two sub-trees. The combining operation applied on these two sub-trees is depicted in Figure 2. The result (grey nodes) replaces the old one that is linked from root.
3.2.2. Mining process. As mentioned before, there are three steps in this phase.

Branching step: The first step is to sum the count of each item. From the initial FI-tree in Figure 1, we iteratively apply the combining operation to all sub-trees that are linked from the same item contained in the header table.

Subset finding step: The main goal of this step is to find all possible remaining paths. Consider each sub-tree of the root node; we first create linkages from the header table to all nodes, and then recursively apply the combining operation to that sub-tree. After that, we discard all linkages, and iterate the process for finding subsets of the remaining sub-trees.

Pruning step: The last step is to remove nodes (and also paths) for itemsets that fall below the minimum support. For the example FI-tree in Figure 1, the final items-tree is depicted in Figure 3. We have pruned the items using min_sup equal to 40%. In this tree we can easily spot the frequent itemsets, such as \{a : 3\}, \{a,c : 2\}, \{a,e,d : 2\}, \{a,c,d,f : 2\}.

4. Parallel FI-Growth

In this section, we present implementation details of a parallel version of the FI-growth algorithm [4] that parallelizes the association rule mining process. We employ a data parallelism technique on a PC cluster (see the workflow in Figure 4). We first partition the transaction database into several portions, and distribute them to different processors for computation. Each processor independently constructs its own local FI-tree structure and discovers corresponding frequent itemsets. However, all processors need to perform a one-time synchronization to exchange their sub-trees before the last two steps in the mining phase.

4.1. Hierarchical minimum support

In the pre-processing phase, transaction database is partitioned and portions are distributed to compute nodes. The partitioning has an effect on the local counts, namely, a decrease of support values of items residing in each partition. Consequently, some items may be pruned out if local pruning is performed, although their global support counts would exceed the required min_sup value. The more partitions created, the more items may be pruned out locally. This pruning directly affects the correctness of final results. However, there are two solutions to avoid such a problem:
i) After creating each local header table, all processors synchronize their lists of item counts, and then merge those lists to obtain global support values. Only the items whose global support is less than the min_sup are pruned out.

ii) Incorporate a hierarchical minimum support technique, utilizing two values of minimum support: min_sup_L1 is defined and used to prune the local header table while min_sup_L2 is defined to prune the local items-tree. Consequently, min_sup_L1 is set to be less than min_sup_L2.

In practice, the first approach provides the most correct result; however, it requires a costly all-to-all broadcast process. The second one gives an approximate pruning that avoids the significant communication overhead of the first one, at the expense of some errors depending on the value of min_sup_L1 and dataset partitioning. In the implementation reported in this paper, we use the second approach.

4.2. Parallelization

In the pre-processing phase, transaction database is equally partitioned and partitions are assigned to all processors. For example, to process the example database in Figure 1 using two processors, we could assign TIDs 1, 2, 3 to the first processor p0, and assign TIDs 4, 5 to the second processor p1.

In the distributed FI-tree construction phase, suppose we assign min_sup_L1 a value of 1 (or 20%) and min_sup_L2 a value of 2 (or 40%). The resulting initial local FI-trees are depicted in Figure 5.

In the mining phase, the combining operations during the branching step can be independently performed by each processor. Figure 6 depicts the evolution of the FI-trees after the branching step. The synchronization of sub-trees' exchange occurs between the processors. Then, the subset finding and pruning steps can be independently run on each processor. More details of these last two steps can be found in [4]. In the next section, we explain how sub-trees are exchanged during the synchronizing step.

4.3. FI-Tree synchronization

The synchronizing step begins with all processors exchanging their local header tables. The combined header table is used to determine which local sub-trees should be sent to which target processors.

- Exchanging of local header table: This process is used to create a global header table on each processor. To reduce the communication overhead, only the list of items is broadcast to other processors. The left-most column in Figure 7 shows the resulting global header tables of p0 and p1.
- Sending of local sub-tree: After each processor has its own global header table, it will determine which local sub-tree(s) should be kept, and which should be sent to the target processors and deleted afterward. Let tree(x) be a sub-tree having item x as root, and |tree(x)| be the number of nodes within that sub-tree. Consider any two sub-trees tree(x) and tree(y); if item x comes before item y in the sequence of items, then the former will have more possible
paths to expand than the latter so that \(|\text{tree}(x)|\) is in general greater than \(|\text{tree}(y)|\). For the example shown in Figure 6, the sequence of items is \(<a, b, c, d, e, f>\); then |tree\((a)\)| is probably greater than |tree\((b)\)|, or |tree\((c)\)| is probably greater than |tree\((d)\)|, etc.

Since we attempt to balance the computing load between processors, we use a simple round robin heuristic to choose which sub-trees should be mined at which processor. We take the remainder of the item’s order in the sequence of items modulo the number of processors to assign a sub-tree to a corresponding processor. For the example in Figure 7, since we begin counting the sequence of item \(a\) from 0, sub-tree \(\text{tree}(a), \text{tree}(b), \text{tree}(c), \text{tree}(d), \text{tree}(e), \text{and tree}(f)\) will be assigned to processor \(p_0, p_1, p_0, p_1, p_0,\) and \(p_1\), respectively.

5. Experiments and results

5.1. Experimental setup

Hardware and environment configuration: We have tested the algorithm on a cluster of x86-64 based SMP machines named “Bedrocks”. Each machine consists of dual 3.2GHz Intel quad-core processors, 4GB of main memory, and an 80GB SATA disk. All machines are equipped with the Linux-based operating system and inter-connected via a 1000Base-TX Ethernet switch. In our implementation, the parallel algorithm is written in the C language and uses the MPICH message passing library version 1.2.7. All experiments were run under no-load conditions, i.e.,
without any competition from other computing tasks in the cluster.

**Data set:** For the test data set, we utilized the standard “IBM synthetic data generator” [15] to synthesize a transaction database. We used 1000 unique items to create 16 million records; each has average transaction length of 10.

### 5.2. Evaluation results and discussion

We ran the test data set on the Bedrocks cluster, using 1 to 32 processors, and measured the total wall-clock time needed for FI-tree construction and mining processes. We repeated each experiment at least 5 times and averaged the run-time results.

Figure 8 illustrates the average running times for the test data set using 1, 2, 4, 8, 16, and 32 processors. We ran the algorithm with two selected minimum support values, 30% and 15%. The results show that run-time drops rapidly as the number of processors increase from 1 to 4, and decreases more slowly as additional processors are used. This may be due to the additional communication overhead incurred. Figure 9 shows that the measured speedup is nearly ideal for 1 to 4 processors; for more than 4 processors, performance is less than ideal but still nearly linear in the number of processors.

As previously mentioned, dividing the test data set into smaller partitions and processing them independently have an effect on local support counts, the result being that items may be locally pruned out even though their global support counts exceed the required minimum value. To investigate this, we study this effect on the test data set using the hierarchical minimum support technique described in section 4.1. We set the \( \min_{supL2} \) to 15%, and simply define the value of \( \min_{supL1} \) to be \( \min_{supL2} \) divided by number of processors. Thus, for an experiment using two processors, \( \min_{supL1} \) will be set to \( \min_{supL2}/2 \).

Figure 10 shows the rule generation accuracy in terms of the number of rules generated using the hierarchical minimum support technique compared with the rules obtained by using single processor configuration. In the experiments, we also set the minimum confidence (\( \min_{conf} \)) to be 75% to prune some unimportant rules. The result in Figure 10 shows that the hierarchical technique can produce up to 0.17% error for 32 processors.
6. Conclusion

Rapid frequent itemsets extraction and significant rules generation from large databases present challenges to data mining research in many areas, including run-time and memory requirements. In this paper we propose a parallel FI-growth algorithm to accelerate association rule mining. We present results of an implementation on an x86-based computing cluster which shows good scalability between 1 and 32 processors, with nearly ideal scaling using 2 to 4 processors, and within 50% of ideal for up to 32 processors. This demonstrates that the FI-tree structure is suitable for independent parallel processing, though some synchronization overhead is incurred.

To improve parallelization and reduce synchronization overhead, we employ a hierarchical minimum support heuristic for distributed pruning of discovered items. The experiments show that this hierarchical approach yields results very close to pruning based on min_sup setting on single processor configuration. In future work, we plan to further investigate the scalability and effects of partitioning and memory requirements using larger size databases and varying partitions. We also plan to explore other techniques to reduce the communication overhead during the data exchange between processors, and study the problem of load balancing.

Acknowledgement

We would like to thank James Edward Brucker for his reading and valuable comments.

7. References


