Parallel Bifold: Large-Scale Parallel Pattern Mining with Constraints

Mohammad El-Hajj, Osmar R. Zaïane
Department of Computing Science, University of Alberta Edmonton, AB, Canada
{mohammad, zaiane}@cs.ualberta.ca

Abstract

When computationally feasible, mining extremely large databases produces tremendously large numbers of frequent patterns. In many cases, it is impractical to mine those datasets due to their sheer size; not only the extent of the existing patterns, but mainly the magnitude of the search space. Many approaches have suggested the use of constraints to apply to the patterns or searching for frequent patterns in parallel. So far, those approaches are still not genuinely effective to mine extremely large datasets.

In this work we propose a method that combines both strategies efficiently, i.e. mining in parallel for the set of patterns while pushing constraints. Using this approach we could mine significantly large datasets; with sizes never reported in the literature before. We are able to effectively discover frequent patterns in a database made of billion transactions using a 32 processors cluster in less than 2 hours.

keywords: Frequent mining, parallel data mining, parallel constraint mining.

paper size: 20 pages.

1 Introduction

Frequent Itemset Mining (FIM) is a key component of many algorithms which extract patterns from transactional databases. For example, FIM can be leveraged to produce association rules, clusters, classifiers or contrast sets. This capability provides a strategic resource for decision support, and is most commonly used for market basket analysis.

One challenge for frequent itemset mining is the potentially huge number of extracted patterns, which can eclipse the original database in size. In addition to increasing the cost of mining, this makes it more difficult for users to find the valuable patterns. Introducing constraints to the mining process helps mitigate both issues. Decision makers can

restrict discovered patterns according to specified rules. By applying these restrictions as early as possible, the cost of mining can be constrained. For example, users may be interested in purchases whose total price exceeds $100, or whose items cost between $50 and $100.

While discovering hidden knowledge in the available repositories of data is an important goal for decision makers, discovering this knowledge in a "reasonable" time is capital. Despite the increase in data collection, the rapidity of the pattern discovery remains vital and will always be essential. Speeding up the process of knowledge discovery has become a critical problem, and parallelism is shown to be a potential solution for such a scalability predicament. Naturally, parallelization is not the only and should not be the first solution to speedup the data mining process. Indeed, other approaches might help in achieving this goal, such as sampling, attribute selection, restriction of search space, and algorithm or code optimization [15]. Some of these approaches might be used in conjunction with parallelism to achieve the desired speedup. A legitimate issue is whether parallelism is needed in data mining. Efficiency is crucial in knowledge discovery systems, and with the explosive growth of data collection, sequential data mining algorithms have become an unacceptable solution to most real size problems even after clever optimizations. To illustrate the complexity of the problem of frequent itemset enumeration in today's real data, assume a small token case with only 5 possible items (i.e. a store that sells only 5 distinct products), the lattice that represents all possible candidate frequent patterns has $2^5 - 1 = 31$ itemsets. Applications that generate transactions with sizes greater than 100 items per transaction are common. In those cases, to find a frequent itemset with size 100, it would take a search space of $2^{100} - 1 = 1.27 * 10^{30}$ itemsets. Adding the fact that most real transactional databases are in the order of millions, if not billions, of transactions and the problem becomes intractable with current sequential solutions. With hundreds of gigabytes, and often terabytes and thousands of distinct items, it is unrealistic for one processor to mine the data sequentially, especially when multiple passes over these enormous databases are required.

Dividing the mining task among different processors represents a potential solution for the above-mentioned problem especially if this parallelism provides answers for decision makers in a reasonable time period and time is of the essence. There are different design issues that affect building parallel frequent mining algorithms [35, 34]. These design issues are significantly affected by the specification of the problem that the system is trying to solve.

Constraint based mining is an ongoing area of research where two important categories of constraints *monotone* and *anti-monotone* [20] are studied in this work. *Anti-monotone* constraints are constraints that when valid for a pattern, they are consequentially valid for any subset subsumed by the pattern. *Monotone* constraints when valid for a pattern are inevitably valid for any superset subsuming that pattern. The straightforward way to deal with constraints is to use them as a filter post-mining. However it is more efficient to consider the constraints during the

mining process. This is what is refereed to as "*pushing the constraints*" [24]. Most existing algorithms leverage (or push) one of these types during mining and postpone the other to a post-processing phase.

1.1 Problem Statement

The problem of mining association rules over market basket analysis was introduced in [1, 2]. The problem consists of finding associations between items or itemsets in transactional data. The data is typically retail sales in the form of customer transactions, but can be any data that can be modeled into transactions. For example medical images where each image is modeled by a transaction of visual features from the image [33], or text data where each document is modeled by a transaction representing a bag of words [14], or web access data where click-stream visitation is modeled by sets of transactions [19], all are well suited applications for association rules or frequent itemsets. Association rules have been shown to be useful for other applications such as recommender systems [29], diagnosis [10], decision support [11], telecommunication [21], and even supervised classification [4, 3]. The main and most expensive component in mining association rules is the mining of frequent itemsets. Formally, the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items. Each item is an object with some predefined attributes such as price, weight, etc. and m is considered the dimensionality of the problem. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A transaction T is said to contain X, a set of items in I, if $X \subseteq T$. A constraint ζ is a predicate on itemset X that yields either *true* or *false*. An itemset X satisfies a constraint ζ if and only if $\zeta(X)$ is *true*. An itemset X has a *support s* in the transaction set $\mathcal{D}$ if *s%* of the transactions in $\mathcal{D}$ contain X. Two particular constraints pertain to the support of an itemset, namely the *minimum support* constraint and the *maximum support* constraint. An itemset X is said to be *infrequent* if its *support* s is smaller than a given minimum support threshold σ; X is said to be *too frequent* if its *support* s is greater than a given maximum support Σ; and X is said to be *large* or *frequent* if its *support* s is greater or equal than σ and less or equal than Σ.

1.2 Contributions in this paper

In this paper we present a new parallel frequent mining algorithm that is based on our previous work of BifoldLeap [13] that generates the set of frequent patterns satisfying the user input constraints. We show that pushing the constraints while parallelizing the mining task allows us to mine databases of sizes never reported before, and in a reasonable time using a cluster made of 32 processors.

The rest of this paper is organized as follows: In section 2, we discuss the types of constraints used in this work.

Leap-traversal approach is illustrated in section 3. Pushing constraints in this approach is explained in section 4. Section 5 describes our proposed parallel approach "Parallel Bifold Leap" where we evaluate some strategies for load sharing. Section 6 presents performance results on experiments assessing scalability and speed-up. Finally, we highlight some related work in Section 7 and conclude the paper in section 8.

2 Constraints

It is known that algorithms for discovering frequent patterns generate an overwhelming number of those patterns. While many new efficient algorithms were recently proposed to allow the mining of extremely large datasets, the problem due to the sheer number of patterns discovered still remains. The set of discovered patterns is often so large that it becomes useless. Different measures of interestingness and filters have been proposed to reduce the discovered patterns, but one of the most realistic ways to find only those interesting patterns is to express constraints on the patterns we want to discover. However, filtering the patterns post-mining adds a significant overhead and misses the opportunity to reduce the search space using the constraints. Ideally, dealing with the constraints should be done as early as possible during the mining process.

2.1 Categories of Constraints

A number of types of constraints have been identified in the literature [20]. In this work, we discuss two important categories of constraints – *monotone* and *anti-monotone*.

Definition 1 (*Anti-monotone constraints*)
A constraint ζ is *anti-monotone* if and only if an itemset X violates ζ, so does any superset of X. That is, if ζ holds for an itemset S then it holds for any subset of S. Many constraints fall within the *anti-monotone* category. The minimum support threshold is a typical anti-monotone constraint. As an example, $sum(S) \leq v(\forall a \in S, a \geq 0)$ is an *anti-monotone* constraint. Assume that items A, B, and C have prices \$100, \$150, and \$200 respectively. Given the constraint ζ = ($sum(S) \leq \$200$), then since itemset AB, with a total price of \$250, violates the ζ constraint, there is no need to test any of its supersets (e.g. ABC) as they also violate the ζ constraint.

Definition 2 (*Monotone constraints*)
A constraint ζ is *monotone* if and only if an itemset X holds for ζ, so does any superset of X. That is, if ζ is violated for an itemset S then it is violated for any subset of S. An example of a *monotone* constraint is $sum(S) \geq v(\forall a \in S, a \geq 0)$. Using the same items A, B, and C as before, and with constraint ζ = ($sum(S) \geq 500$), then knowing that ABC violates the constraint ζ is sufficient to know that all subsets of ABC will violate ζ as well. Table 1

MONOTONE	ANTI-MONOTONE
$min(S) \leq v$	$min(S) \geq v$
$max(S) \geq v$	$max(S) \leq v$
$count(S) \geq v$	$count(S) \leq v$
$sum(S) \geq v (\forall a \in S, a \geq 0)$	$sum(S) \leq v (\forall a \in S, a \geq 0)$
$range(S) \geq v$	$range(S) \leq v$
$support(S) \leq v$	$support(S) \geq v$

Table 1. Commonly used *monotone* and *anti-monotone* constraints

presents commonly used constraints that are either *anti-monotone* or *monotone*. From the definition of both types of constraints we can conclude that *anti-monotone* constraints can be pushed when the mining-algorithm uses the bottom-up approach, as we can prune any candidate superset if its subset violates the constraint. Conversely, the *monotone* constraints can be pushed efficiently when we are using algorithms that follow the top-down approach as we can prune any subset of patterns from the answer set once we find that its superset violates the *monotone* constraint.

3 The Leap Traversal Approach

Contrary to most existing parallel algorithms for mining frequent patterns, our algorithm is not apriori-based. To mine for frequent patterns in parallel while pushing constraints, we rely on a completely new and different approach and use special structures that fit well a distributed or cluster environment. Before elaborating on our parallel algorithm, we first present the data structures and explain the general concepts. Our algorithm is based on our recent new lattice traversal strategy HFP-Leap [31] and the approach for pushing constraints presented in [13]. In our parallel approach, HFP-Leap still performs the actual leap-traversal to find maximal patterns. Then filter those patterns based on constraints that will be pushed during the mining process. We first present the idea behind HFP-Leap then show how this idea can be used to push constraints using the BifoldLeap algorithm. Finally the parallelization process of this algorithm is explained highlighting our load sharing strategies. The Leap-Traversal approach we discuss consists of two main stages: the construction of a Frequent Pattern tree (HFP-tree); and the actual mining for this data structure by building the tree of intersected patterns.

3.1 Construction of the Frequent Pattern Tree: The sequential way

The goal of this stage is to build a compact data structure, which is a prefix tree representing sub-transactions pertaining to a given minimum support threshold. This data structure, compressing the transactional data, is based the FP-tree by Han et al. [17]. The tree structure we use, called HFP-tree is a variation of the original FP-tree. We

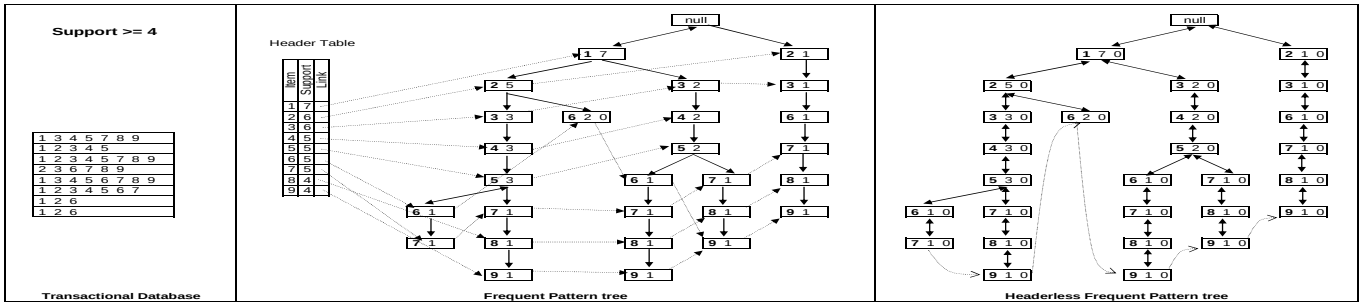


Figure 1. Example of transactional database with its original FP-tree and Headerless FP-tree.

start introducing the original FP-tree before discussing the differences with our data structure. The construction of the FP-tree is done in two phases, where each phase requires a full I/O scan of the database. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase.

After the enumeration of the items appearing in the transactions, infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called header table, where the items and their respective supports are stored along with pointers to the first occurrence of the item in the frequent pattern tree. The actual frequent pattern tree is built in the second phase. This phase requires a second complete I/O scan of the database. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-Tree.

Each ordered sub-transaction is compared to the prefix tree starting from the root. If there is a match between the prefix of the sub-transaction and any path in the tree starting from the root, the support in the matched nodes is simply incremented, otherwise new nodes are added for the items in the suffix of the transaction to continue a new path, each new node having a support of one. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

Our tree structure is similar in principle to the FP-tree But has the following differences. We call this tree Headerless-Frequent-Pattern-Tree or HFP-tree.

1. We do not maintain a header table, as a header table is used to facilitate the generation of the conditional trees in the FP-growth model [17]. It is not needed in our leap traversal approach;

2. We do not need to maintain the links between the same itemset across the different branches (horizontal links);

3. The links between nodes are bi-directional to allow top-down and bottom-up traversals of the tree;

4. All leaf nodes are linked together as the leaf nodes are the start of any pattern base and linking them helps the discovery of frequent pattern bases;

5. In addition to *support*, each node in the HFP-tree has a second variable called *participation*.

Basically, the support represents the support of a node, while participation represents, at a given time in the mining process, the number of times the node has participated in already counted patterns. Based on the difference between the two variables, *participation* and *support*, the special patterns called *frequent-path-bases* are generated. These are simply the paths from a given node x, with participation smaller than the support, up to the root, (i.e. nodes that did not fully participate yet in frequent patterns). Figure 1 presents the Headerless FP-tree and the original FP-tree for the same transactional data. Algorithm 1 shows the main steps in our approach while pushing the two types of con-

Algorithm 1 HFP-Bifold: BifoldLeap Headerless FP-tree

Input: D (transactional database); $P()$; $Q()$; and σ (Support threshold).
Output: Maximal patterns with their respective supports.
Scan D to find the set of frequent 1-itemsets $F1$
Scan D to build the Headerless FP-tree HFP
$FPB \leftarrow$ FindFrequentPatternBases(HFP)
$PQ - Patterns \leftarrow$ Find-PQ-Patterns(FPB, σ)
Output $PQ - Patterns$

straints which are: The conjunction of all *anti-monotone* constraints that comprises a predicate which we call $P()$. A second predicate $Q()$ contains the conjunction of the *monotone* constraints. After building the Headerless FP-tree with 2 scans of the database, we mark some specific nodes in the pattern lattice using *FindFrequentPatternBases*. Using the FPBs, Bifold idea in *Find-PQ-Patterns* discovers the frequent patterns that satisfy both types of constraints $P()$ and $Q()$. Algorithm 2 shows how patterns in the lattice are marked. The linked list of leaf nodes in the HFP-tree is traversed to find upward the unique paths representing sub-transactions. If frequent maximals exist, they have to be among these complete sub-transactions. The participation counter helps reusing nodes exactly as needed to determine the frequent path bases. Algorithm 3 presents the BifoldLeap idea which is Leap with constraints. This idea with its pruning algorithms are explained in the details in next section.

3.2 Construction of the Frequent Pattern Tree: The parallel way

Constructing the parallel HFP-tree is done by allocating each processor to an equal size-partition of the original dataset. The parallel way has a similar idea of the two-phase sequential method described in the previous section, where each processor finds its local candidate items and broadcasts them to generate the global frequent 1-itemset in phase one. Second phase of building the HFP-tree builds one HFP-tree for each processor based on the global Frequent 1-itemset collected in the previous phase. Building those trees is an embarrassingly parallel task, where

Algorithm 2 FindFrequentPatternBases: Marking nodes in the lattice

Input: HFP (Headerless FP-Tree).
Output: FPB (Frequent pattern bases with counts)

$ListNodesFlagged \leftarrow \emptyset$
Follow the linked list of leaf nodes in HFP
for each leaf node N **do**
  Add N to $ListNodesFlagged$
end for
while $ListNodesFlagged \neq \emptyset$ **do**
  $N \leftarrow$ Pop($ListNodesFlagged$) {from top of the list}
  $fpb \leftarrow$ Path from N to $root$
  fpb.branchSupport $\leftarrow N$.support - N.participation
  for each node P in fpb **do**
    P.participation $\leftarrow P$.participation + fpb.branchSupport
    if P.participation $< P$.support AND $\forall c$ child of P, c.participation $= c$.support **then**
      add P in $ListNodesFlagged$
    end if
  end for
  add fpb in FPB
end while
RETURN FPB

each tree built is completely independent from others built by different processors. Detail explanations on this process is explained in Section 5.

4 BifoldLeap: Leap with constraints

The algorithm *COFILeap* [12] offers a number of opportunities to push the monotone and anti-monotone predicates, $P()$ and $Q()$ respectively. We start this process by defining two terms which are head (H) and tail (T) where H is a frequent path base or any subset generated from the intersection of frequent path bases, and T is the itemset generated from intersecting all remaining frequent path bases not used in the intersection of H. The intersection of H and T, $H \cap T$, is the smallest subset of H that may yet be considered. Thus Leap focuses on finding frequent H that can be declared as local maximals and candidate global maximals. *BifoldLeap* extends this idea to find local maximals that satisfy $P()$. We call these P-maximals. Although we further constrain the P-maximals to itemsets which satisfy $Q()$, not all subsets of these P-maximals are guaranteed to satisfy $Q()$. To find the itemsets which satisfy both constraints, the subsets of each P-maximal are generated in order from long patterns to short. When a subset is found to fail $Q()$, further subsets do not need to be generated for that itemset, as they are guaranteed to fail $Q()$ also. constraints can be pushed while intersecting the frequent path bases, which is the main phase where both types of constraints are pushed at the same time (Algorithm 3).There are two high-level strategies for pushing constraints during the intersection phase. First, $P()$ and $Q()$ can be used to eliminate an itemset or remove the need to evaluate its intersections with additional frequent path bases. Second, $P()$ and $Q()$ can be applied to the

Algorithm 3 Find-PQ-Patterns: Pushing $P()$ and $Q()$

Input: FPB (Frequent Pattern Bases); $P()$; $Q()$; and σ (Support threshold).
Output: Frequent patterns satisfying P(), Q()
$PLM(PLMaximals) \leftarrow \{P(FPB)$ and frequent$\}$
$InFrequentFPB \leftarrow notFrequent(FPB)$
for each pair $(A, B) \in InFrequentFPB$ **do**
  $header \leftarrow A \cap B$
  Add $header$ in PLM and Break IF ($P(header)$ AND is frequent and not $\emptyset$)
  Delete $header$ and break IF (Not $Q(header)$)
  $tail \leftarrow$ Intersection(FPBs not in $header$)
  delete $header$ and break IF (Not $P(header \cap tail)$)
  Do not check for $Q()$ in any subset of $header$ IF ($Q(header \cap tail)$)
end for
for each pattern P in PLM **do**
  Add P in PGM IF ((P not subset of any $M \in PGM$)
end for
PQ-Patterns $\leftarrow$ GPatternsQ(FPB, PGM) (All subsets that satisfy both types of constraints)
Output PQ-Patterns

"head intersect tail" ($H \cap T$), which is the smallest subset of the current itemset that can be produced by further intersections. These strategies are detailed in the following four theorems.

Theorem 1: If an intersection of frequent path bases (H) fails $Q()$, it can be discarded, and there is no need to evaluate further intersections with H.

Proof: If an itemset fails $Q()$, all of its subsets are guaranteed to fail $Q()$ based on the definition of monotone constraints. Further intersecting H will produce subsets, all of which are guaranteed to violate $Q()$.

Theorem 2: If an intersection of frequent path bases (H) passes $P()$, it is a candidate P-maximal, and there is no need to evaluate further intersections with H.**Proof**: Further intersecting H will produce subsets of H. By definition, no P-maximal is subsumed by another itemset which also satisfies $P()$. Therefore, none of these subsets of H are potential new P-maximals.

Theorem 3: If a node's $H \cap T$ fails $P()$, the H node can be discarded, and there is no need to evaluate further intersections with H. **Proof**: If an itemset fails $P()$, then all of its supersets will also violate $P()$ from the definition of anti-monotone constraints. Since a node's $H \cap T$ represents the subset of H that results from intersecting H with all remaining frequent path bases, H and all combinations of intersections between H and remaining frequent path bases are supersets of $H \cap T$ and therefore guaranteed to fail $P()$ also.

Theorem 4:If a node's $H \cap T$ passes $Q()$, $Q()$ is guaranteed to pass for any itemset resulting from the intersection of a subset of the frequent path bases used to generate H plus the remaining frequent path bases yet to be intersected with H. $Q()$ does not need to be checked in these cases. **Proof**: $Q()$ is guaranteed to pass for all of these itemsets

because they are generated from a subset of the intersections used to produce the $H \cap T$ and are therefore supersets of the $H \cap T$.

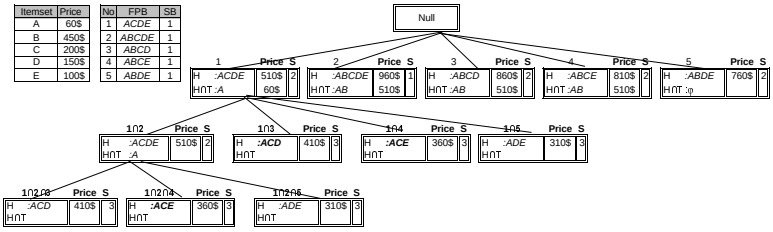


Figure 2. Pushing $P()$ and $Q()$.

The following example, shown in Figure 2, illustrates how *BifoldLeap* works. It this example a transactional database is made from five items, A, B, C, D, and E, with prices \$60, \$450, \$200, \$150, and \$100 respectively. Assume this transactional database generates 5 frequent path bases, $ACDE$, $ABCDE$, $ABCD$, $ABCE$, and $ABDE$, each with branch support one. The anti-monotone predicate, $P()$, is $Sum(Prices) \leq \$500$, and the monotone predicate, $Q()$, is $Sum(prices) \geq \$100$. Intersecting the first FPB with the second produces $ACDE$ which has a price of \$510, and therefore violates $P()$ and passes $Q()$. Next, we examine the $H \cap T$, the intersection of this node with the remaining three FPBs, which yields A with price \$60, passing $P()$ and failing $Q()$. None of these constraint checks provide an opportunity for pruning, so we continue intersecting this itemset with the remaining frequent path bases. The first intersection is with the third FPB, producing ACD with price \$410, which satisfies both the anti-monotone and monotone constraints. The second intersection produces ACE, which also satisfies both constraints. The same thing occurs with the last intersection, which produces ADE. Going back to the second frequent path base, ABCDE, we find that the $H \cap T$, AB, violates the anti-monotone constraint with price \$510. Therefore, we do not need to consider $ABCDE$ or any further intersections with it. The remaining nodes are eliminated in the same manner. In total, three candidate P-maximals were discovered. We can generate all of their subsets while testing only against $Q()$. Finally, the support for these generated subsets can be computed from the existing frequent path bases.

5 Parallel BifoldLeap: Building the structures in parallel and mining them in parallel

The parallel BifoldLeap starts by partitioning the data among the parallel nodes, where each node receives almost the same number of transactions. Each processor scans its partition to find the frequency of candidate items. The list of all supports is reduced to the master node to get the global list of frequent 1-itemsets. The second scan of each partition starts with the goal of building a local headerless frequent patterns tree. From each tree, the local set

of frequent path bases is generated. Those sets are broadcasted to all processors. Identical frequent path bases are merged and sorted lexicographically, the same as with the sequential process. At this stage the pattern bases are split among the processors. Each processor is allocated a carefully selected set of frequent pattern bases to build their respective intersection trees, with the goal of creating similar depth trees among the processors. This distribution is discussed further below. Pruning algorithms are applied at each processor to reduce the size of the intersection trees as it is done in the sequential version [31]. Maximal patterns that satisfy the $P()$ constraints are generated at each node. Each processor then sends its P-maximal patterns to one master node, which filters them to generate the set of global P-maximal patterns and then find all their subsets that satisfy $Q()$. Algorithm 4 presents the steps needed to generate the set of patterns satisfying both $P()$ and $Q()$ in parallel.

5.1 Load sharing among processors

While the trees of intersections are not physically built, they are virtually traversed to complete the relevant intersections of pattern bases. Since each processor can handle independently some of these trees and the sizes of these trees of intersections are monotonically decreasing, it is important to cleverly distribute these among the processors to avoid significant load imbalance. A naïve and direct approach would be to divide the trees sequentially. Given p processors we would give the first $\frac{1}{p}^{th}$ trees to the first processor, the next fraction to the second processor, and so on. This strategy unfortunately leads to eventual imbalance between processors since the last processor getting all small trees would undoubtedly terminate before other nodes in the cluster. A more elegant and effective approach would be a round robin approach considering the sizes of the trees: when ordered by size, the first p trees are distributed one to each processor and so on for each set of p trees. This avoids having a processor dealing with only large trees while another processor is intersecting with only small ones. Again this strategy may still create imbalance between processors, however, less acute than the naïve direct approach. The strategy that we propose, and call First-Last, distributes two trees per processor at a time. The largest tree and the smallest tree are assigned to the first processor, then the second largest tree and penultimate small tree to the second processor, the third largest tree and third smallest tree to the third processor and so on in a loop. This approach seems to advocate a better load balance as is demonstrated by our experiments.

5.2 Parallel Leap Traversal Approach : An Example

The following example illustrates how the BifoldLeap approach is applied in parallel. Figure 3.A presents 7 transactions made of 8 distinct items which are: A, B, C, D, E, F, G, and H with prices 10$, 20$, 30$, 40$, 50$,

Algorithm 4 Parallel-HFP-BifoldLeap: Parallel-BifoldLeap with Headerless FP-tree

Input: D (transactional database); $P()$; $Q()$; and σ (Support threshold).
Output:
Patterns satisfying both $P()$ and $Q()$ with their respective supports.
- D is already distributed otherwise partition D between the available p processors;
- Each processor p scans its local partition D_p to find the set of local candidate 1-itemsets L_pC1 with their respective local support;
- The supports of all L_iC1 are transmitted to the master processor;
- Global Support is counted by master and $F1$ is generated;
- $F1$ is broadcasted to all nodes;
- Each processor p scans its local partition D_p to build the local Headerless FP-tree L_pHFP based on $F1$;
- $L_pFPB \leftarrow$ FindFrequentPatternBases(L_pHFP);
- All L_pFPB are sent to the master node ;
- Master node generates the global FPB from all L_pFPB;
- The global FPB are broadcasted to all nodes;
- Each Processor p is assigned a set of local header nodes LHD from the global FPB; {this is the distribution of trees of intersections}
for each i in LHD **do**
  $LOCAL - P - Maximals \leftarrow$ Find-P-Maximals($FPB, \sigma, P(), Q()$);
end for
- Send all $LOCAL - P - Maximals$ to the master node;
- The master node prunes all $LOCAL - P - Maximals$ that have supersets itemsets in $LOCAL - P - Maximals$ to produce $GLOBAL - P - Maximals$;
- The master node generates frequent patterns satisfying both P() and Q() from $GLOBAL - P - Maximals$.

60$, and 70$ respectively. Assuming we want to mine those transactions with a support threshold equals to at least 3 and generates patterns that their total prices are between 30$ and 100$ (i.e $P() : SumofPrices < 100$, and $Q()SumofPrices > 30$, using two processors. Figures 3.A and 3.B illustrate all the needed steps to accomplish this task. The database is partitioned among the two processors where the first three transactions are assigned to the first processor, $P1$, and the remaining ones are assigned to the second processor, $P2$ (Figure 3.A).

In the first scan of the database, each processor finds the local support for each item: $P1$ finds the support of A, B, C, D, E, F and G which are 3, 2, 2, 2, 2, 1 and 2 respectively, and $P2$ the supports of A, B, C, D, E, F, and H which are 2, 3, 3, 3, 3, 3, 2. A reduced operation is executed to find that the global support of A, B, C, D, E, F, G, and H items is 5, 5, 5, 5, 5, 4, 2, and 2. The last two items are pruned as they do not meet the threshold criteria (support > 2), and the remaining ones are declared frequent items of size 1. The set of Global frequent 1-itemset is broadcasted to all processors using the first round of messages.

The second scan of the database starts by building the local headerless tree for each processor. From each tree the local frequent path bases are generated. In $P1$ the frequent-path-bases $ABCDE$, ABE, and $ACDF$ with branch support equal to 1 are generated. $P2$ generates $ACDEF$, $BCDF$, BEF, and $ABCDE$ with branch supports equal to 1 for all of them (Figure 3.B). The second set of messages is executed to send the locally generated frequent path bases to $P1$. Here, identical ones are merged and the final global set of frequent path bases are broadcasted to all processors with their branch support. (3.C)

Each processor is assigned a set of header nodes to build their intersection tree as in Figure 3.D. In our example,

the first, third, and sixth frequent path bases are assigned to $P1$ as header nodes for its intersection trees. $P2$ is assigned to the second, fourth, and fifth frequent path bases. The first tree of intersection in $P1$ produces 3 P-maximals (i.e. with total prices is less than 100$) $BCD : 90\$$, $ABE : 80\$$, and $ACD : 80\$$ with support equal to 3, 3, and 4 respectively. The second assigned tree does no produce any P-maximals. $P1$ produces 3 local P-maximals which are $BCD : 90\$$, $ABE : 80\$$, and $ACD : 80\$$. $P2$ produced $BE : 70\$$, and $AE : 60\$$ with support equal to 4, and 4 respectively. All local P-maximals are sent to $P1$ in which any local P-maximal that has any other superset of local P-maximals from other processors are removed. The remaining patterns are declared as global P-maximals (Figure 3.E). Subsets of the Global P-maximals that satisfy $Q()$ which is prices > 30 are kept and others are pruned. The final results set produces $D : 40\$$, $E : 50\$$, $AC : 40\$$, $AD : 50\$$, $BC : 50\$$, $BD : 60\$$, $CD : 70\$$, $BE : 70\$$, $AE : 60\$$, $BCD : 90\$$, $ABE : 80\$$, and $ACD : 80\$$

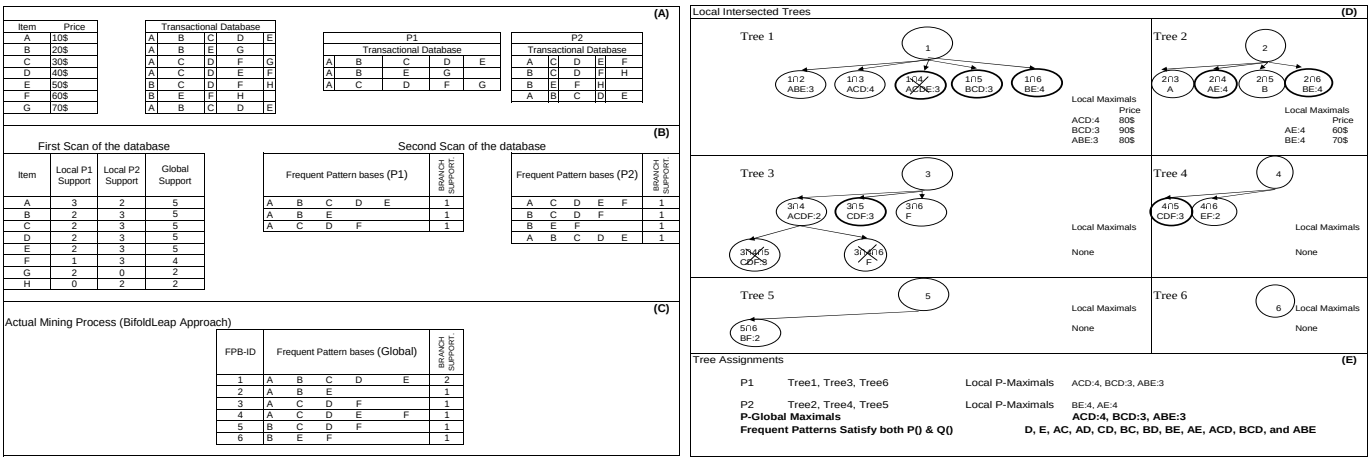


Figure 3. Example of Parallel BifoldLeap: Finding the pattern bases, Intersecting pattern bases

6 Performance Evaluations

To evaluate our parallel BifoldLeap approach, we conducted a set of different experiments to test the effect of pushing *monotone* $Q()$ and *anti-monotone* $P()$ constraints separately, and then both in combination for the same datasets. These experiments were conducted using a cluster made of twenty boxes. Each box has Linux 2.4.18, dual processor 1.533 GHz AMD Athlon MP 1800+, 1.5 GB of RAM. Nodes are connected by Fast Ethernet and Myrinet 2000 networks. In this set of experiments, we generated synthetic datasets using [18]. All transactions are made of 100,000 distinct items with an average transaction length of 12 items per transaction. The size of the transactional databases used varies from 100 million transactions to 1 billion transactions.

With our best efforts and literature searches, we were unable to find a parallel frequent mining algorithm that

could mine more than 10 million transactions, which is far less than our target size environment. Due to this large discrepancy in transaction capacity, we could not compare our algorithm against any other existing algorithms, as none of them could mine and reach our target data size.

We conducted a battery of tests to evaluate the processing load distribution strategy, the scalability vis-à-vis the size of the data to mine, and the speed-up gained from adding more parallel processing power. Some of the results are portrayed hereafter.

6.1 Effect of load distribution strategy

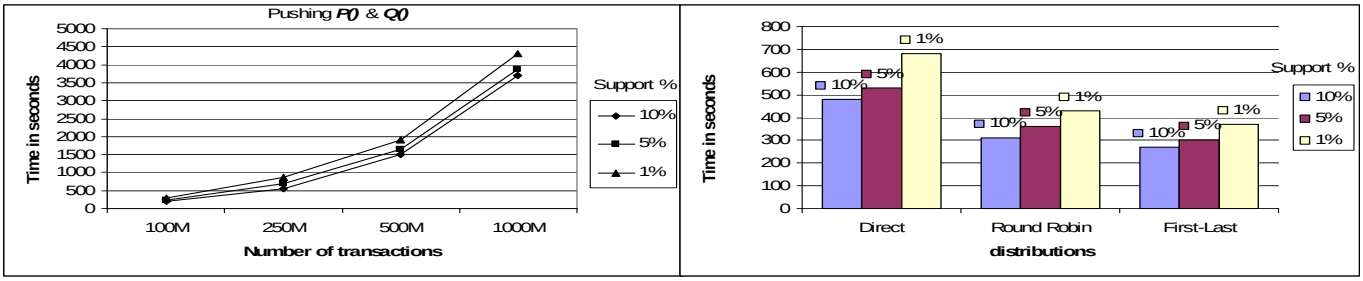
We enumerated above three possible strategies for tree of intersection distribution among the processors. As explained, the trees are in decreasing order of size and they can either be distributed arbitrarily using the naïve approach, or more evenly using a round robin approach, or finally with the First-Last approach.

The naïve and simple strategy uses a direct and straightforward distribution. For example if we have 6 trees to assign to 3 processors, the first two trees are assigned to the first processor, the third and fourth trees are assigned to the second processor, and the last two trees are assigned to the last processor. Knowing that the last trees are smaller in size than the first trees, the thirds processor will inevitably finish before the first processor. In the round robin distribution, the first, second and third tree are allocated respectively to the first, second and third processor and then the remaining forth, fifth and sixth trees are assigned respectively to processor one, two and three. With the last strategy of distribution, First-Last, the trees are assigned in pairs: processor one works on the first and last tree, processor two receives the second and fifth tree, while the third processor obtains the third and fourth trees.

From our experiments in Figure 4.B we can see that the First-Last distribution gave the best results. This can be justified by the fact that since trees are lexicographically ordered then in general trees on the left are larger than those on the right. By applying the First-Last distributions we always try to assign largest and smallest tree to the same node. All our remaining experiments use the First-Last distribution methods among intersected trees.

6.2 Scalability with respect to database size

One of the main goals in this work is to mine extremely large datasets. In this set of experiments we tested the effect of mining different databases made of different transactional databases varying from 100 million transactions up to one billion transactions while pushing both type of constraints $P()$ and $Q()$. To the best of our knowledge, experiments with such big sizes have never been reported in the literature. We mined those datasets using 32 processors, with three different support thresholds: 10%, 5% and 1%. We were able to mine one billion transactions



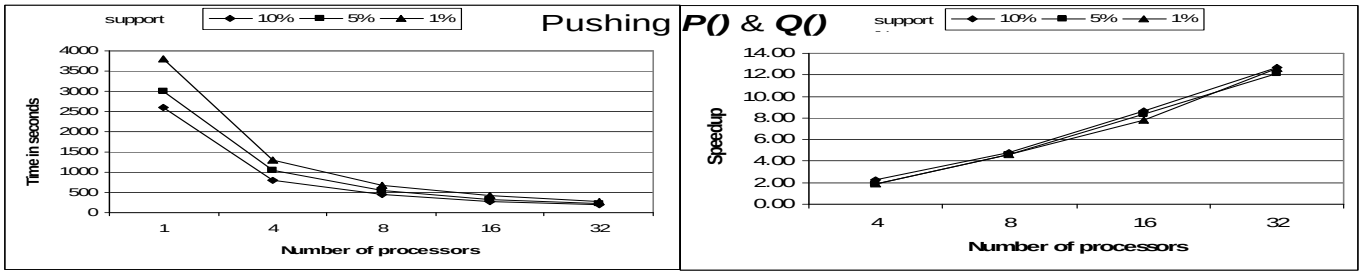
(A) (B)

Figure 4. (A) Scalability with respect to the transaction size, (B) Effect of node distributions, (number of processors = 32)

in 3700 seconds for a support of 0.1 up to 4300 seconds for a support of 0.01. Figure 4.A shows the results of this set of experiments. While the curve does not illustrate a perfect linearity in the scalability, the execution time for the colossal one billion transaction dataset was a very reasonable one hour and forty minutes with a 0.01 support and 32 relatively inexpensive processors.

6.3 Scalability with respect to number of processors

To test the speed-up of our algorithm with the increase of processors we fixed the size of the database to 100 million transactions and examined the execution time on this dataset with one processor up to 32 processors. The execution time is reduced sharply when two to four parallel processors are added then continues to decrease significantly afterward with additional processors (Figure 5.A). The speedup was fairly acceptable as almost two folds were achieved with 4 processors, 4 folds while using 8 processors, and almost 13 folds while using 32 processors. This results are depicted in Figure 5.B.



(A) (B)

Figure 5. (A) Scalability with respect to the number of processors. (B) Speedup with different support values (transaction size = 100M)

7 Related work

Recent days have witnessed an explosive growth in generating data in all fields of science, business, military, etc. The same rate of growth in the processing power of evaluation and analyzing the data has not followed this massive growth to the same extent.

It has therefore become critical to design efficient parallel algorithms to achieve the data mining tasks. Another reason that necessitates the parallel solution is that most extremely large databases reside in different locations, and the cost of bringing them into one site for sequential discovery can be prohibitively expensive. In this section, we are mentioning the parallel environments used in the literature and most of the existing parallel association rule mining algorithms.

Parallel environment has been described either as a single computer with multiple processors sharing the same address space (i.e. Shared Memory) or as multiple interconnected computers where each one has its own independent local memory (i.e. Shared Nothing), or Distributed Memory [30]. The first platform allows any processor to access the memory space directly, where synchronization occurs via locks and barriers. Applications running on such a platform try to achieve the following goals: reducing the synchronization points, achieving high data locality which means maximizing access to local cache, and avoiding false sharing. In the later approach, Shared Nothing, if a processor requires data contained in another processor's memory space, messages must be passed between them using some function library routines like MPI, or PVM. The main objective of this method is to reduce and optimize communication costs, and to have good decomposition of data, because it is highly affected by the distribution of work among nodes. It is true that shared memory programming offers simplicity over the distributed one, but using the common memory with a finite bus in shared memory affects scalability. On the other hand the distributed architecture solves the scalability problem at the expense of programming simplicity.

In the realm of association rules, existing parallel frequent itemset mining algorithms are divided among the two parallel environments mentioned above where distributed algorithms are grouped into two main categories based on how candidate sets are handled. Some algorithms rely on replications of candidate sets while others partition the candidate set.

Replication is the simplest approach. In this approach the candidate generation process is replicated and the counting step is performed in parallel where each processor is assigned part of the database to mine. This method suffers mainly from three problems. First, not all local frequent items are global frequent items, the "false positive phenomenon." Second, not all non-local frequent items are non-global frequent items, the "false negative phenomenon." Finally, it depends heavily on the memory size. The main algorithms on this class are: Count Distribution algorithm

[26], Parallel Partition algorithm [27], Fast Distributed Mining algorithm [8], Fast Parallel Mining algorithm [8], Parallel Data Mining algorithm [22]. These algorithms differs in the they compute the candidate patterns and the number of round of messages sent for each phase.

Partitioning Algorithms are the second type of parallel algorithms that rely on the concept of partitioning the candidate set among processors. Here, each processor handles only a predefined set of candidate items and scans the entire database, leading to prohibitive I/O costs. In cases of extremely large databases these algorithms collapse due to excessive I/O scans required of them. In general they are used to mine relatively small databases with limited memory bandwidth. Some of these algorithms are Data Distribution algorithm, Candidate Distribution algorithm [26], Intelligent Data Distribution algorithm [16]. the first three algorithms suffer from the expensive communication due to the ring based All-To-All broadcasting for the local portions of the data sets. The later algorithm solve this issue by applying Point-To-Point communication between neighbors, thus eliminating any communication contentions. Most of the above mentioned algorithms are based on the apriori algorithm[1], which requires multi-scan of the database and a massive candidate generation phase. That is why most of them are not fully scalable for extremely large datasets.

A parallelization of the MaxMiner [5] is presented in [9]. The algorithm inherits the effective pruning of MaxMiner but also its drawbacks. It is efficient for long maximal patterns but not as capable when most patterns are short. It also requires multiple scans of the data making it inefficient for extremely large datasets.

A PC-cluster based algorithm proposed in [25], derived from the sequential FP-growth algorithm [17], exhibits good load balancing. Being a non-apriori based approach, the candidacy generation is significantly reduced. However, node-to-node communication is considerable especially for sending conditional patterns. The algorithm displays good speedup, but on the other hand it does not scale to extremely large datasets as the larger the dataset, the more conditional patterns are found, and the more node-to-node communication is required.

Myriad shared memory-based parallel frequent mining algorithms are described in the literature such as Asynchronous Parallel Mining [7], Parallel Eclat, MaxEclat, Clique, MaxClique, TopDown, and AprClique algorithms all reported in [23]. These algorithms are mainly apriori-based and suffer from expensive candidacy generation and communication costs. Multiple Local Frequent Pattern tree Algorithm [32], which was among the first non apriori-based parallel mining algorithm, was our first attempt parallelizing FP-growth. Such algorithms show good performance while mining for frequent patterns, but due to the nature of shared memory environments with limited bus and common disks, they are not suitable to be scaled for extremely large dataset.

In a recent study [28] of parallelizing Dualminer which is the first frequent mining algorithm that supports pushing

the two types of constraints, the authors showed that by mining relatively small sparse datasets consisting of 10K transactions and 100K items, the sequential version of Dualminer took an excessive amount of time. Unfortunately, the original authors of Dualminer did not show any single experiment to depict the execution time of their algorithm but only the reduction in predicate executions [6].

What distinguishes our approach from the afore mentioned algorithms is the strategy for traversing the lattice of candidate patterns. Candidate checking was significantly reduced by using pattern intersections and communication costs are condensed thanks to the self-reliant and independent processing modules, and finally, the data structure we use and the approach of sharing tasks support a quasi de facto load balance.

8 Conclusion

Parallelizing the search for frequent patterns plays an important role in opening the doors to the mining of extremely large datasets. Not all good sequential algorithms can be effectively parallelized and parallelization alone is not enough. An algorithm has to be well suited for parallelization, and in the case of frequent pattern mining, clever methods for searching are certainly an advantage. The algorithm we propose for parallel mining of frequent patterns while pushing constraints is based on a new technique for astutely jumping within the search space, and more importantly, is composed of autonomous task segments that can be performed separately and thus minimize communication between processors.

Our proposal is based on the finding of particular patterns, called pattern bases, from which selective jumps in the search space can be performed in parallel and independently from each other pattern base in the pursuit of frequent patterns that satisfy user's constraints. The success of this approach is attributed to the fact that pattern base intersection is independent and each intersection tree can be assigned to a given processor. The decrease in the size of intersection trees allows a fair strategy for distributing work among processors and in the course reducing most of the load balancing issues. While other published works claim results with millions of transactions, our approach allows the mining in reasonable time of databases in the order of billion transactions using relatively inexpensive clusters; 16 dual-processor boxes in our case. This is mainly credited to the low communication cost of our approach.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[3] M.-L. Antonie and O. R. Zaïane. Text document categorization by term association. In *In Proc. of the IEEE 2002 International Conference on Data Mining*, pages 19–26, Maebashi City, Japan, 2002.

[4] W. H. B. Liu and Y. Ma. Integrating classification and association rule mining. In *4th Intl. Conf. on Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York City, NY, August 1998.

[5] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD*, 1998.

[6] C. Bucila, J. Gehrke, D. Kifer, and W. White. Dualminer: A dual-pruning algorithm for itemsets with constraints. In *Eight ACM SIGKDD Internationa Conf. on Knowledge Discovery and Data Mining*, pages 42–51, Edmonton, Alberta, August 2002.

[7] D. Cheung, K. Hu, and S. Xia. Asynchronous parallel algorithm for mining association rules on a shared-memory multi-processors. In *Proc. 10th ACM Symp. Parallel Algorithms and Architectures, ACM Press, NY*, pages 279 – 288, 1998.

[8] D. W.-L. Cheung, J. Han, V. Ng, A. W.-C. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *PDIS*, pages 31–42, 1996.

[9] S. M. Chung and C. Luo. Parallel mining of maximal frequent itemsets from databases. In *15th IEEE International Conference on Tools with Artificial Intelligence*, 2003.

[10] N. L. D. Gamberger and V. Jovanoski. High confidence association rules for medical diagnosis. In *Intelligent Data Analysis in Medicine and Pharmacology, (IDAMAP'99)*, Washington, DC, November 1999.

[11] S. Downs and M. Wallace. Mining association rules from a pediatric primary care decision support system. In *Proc American Medical Informatics Association Annual Symposium*, 2000.

[12] M. El-Hajj and O. R. Zaïane. Mining with constraints by pruning and avoiding ineffectual processing. In *The 18th Australian Joint Conference on Artificial Intelligence, Springer Verlag LNCS 3809*, pages 1001–1004, Sydney, Australia, 2005.

[13] M. El-Hajj, O. R. Zaïane, and P. Nalos. Bifold constraint-based mining by simultaneous monotone and anti-monotone checking. In *The fifth IEEE International Conference on Data Mining (ICDM'05)*, Houston, TX, 2005.

[14] R. Feldman and H. Hirsh. Mining associations in text in the presence of background knowledge. In *Proc. 2st Int. Conf. Knowledge Discovery and Data Mining*, pages 343–346, Portland, Oregon, Aug. 1996.

[15] A. Freitas. Survey of parallel data mining. In *Proc. 2nd Int. Conf. on the Practical Applications of Knowledge Discovery and Data Mining*, pages 287–300, January 1996.

[16] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *In ACM SIGMOD Conf. Management of Data*, 1997.

[17] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.

[18] IBM_Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html.

[19] M. D. J. Srivastava, R. Cooley and P.-N. Tan. Web usage mining: Discovery and applications of usage patterns form web data. In *SIGKDD Explorations, 1(2)*, January 2000.

[20] L. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *In proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD'99)*, pages 157–168, 1999.

[21] H. M. M. Klemettinen and H. Toivonen. Rule discovery in telecommunication alarm data. In *Journal of Network and Systems Management, 7(4)*, 1999.

[22] S. Park, M. Chen, and P. S. Yu. Efficient parallel data mining for association rules. In *In ACM Intl. Conf. Infomration and Knowledge Management*, 1995.

[23] S. Parthasarathy, M. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems, in knowledge and information systems. In *Volume 3, Number 1*, pages 1–29, 2001.

[24] J. Pie and J. Han. Can we push more constraints into frequent pattern mining? In *In ACM SIGKDD Conference*, pages 350–354, 2000.

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

[26] A. R. and S. J. Parallel mining of association rules. In *IEEE Transactions in Knowledge and Data Eng.*, pages 962–969, 1996.

[27] A. Savasere, E. Omiecinski, , and S. Navathe. An efficient algorithm for mining association rules in large databases. In *In Proceedings of the 21st International Conference on Very Large Data Bases*, pages 432–444, 1995.

[28] R. M. Ting, J. Bailey, and K. Ramamohanarao. Paradualminer: An efficient parallel implementation of the dualminer algorithm. In *Eight Pacific-Asia Conference, PAKDD 2004*, pages 96–105, Sydney, Australia, May 2004.

[29] S. A. A. W. Lin and C. Ruiz. Efficient adaptive-support association rule mining for recommender systems. In *Data Mining and Knowledge Discovery*, pages 6(1):83 – 105, January 2002.

[30] B. Wilkinson and M. Allen. Parallel programming techniques and applications using networked workstations and parallel computers. In *Alan Apt, New Jersy, USA*, 1999.

[31] O. R. Zaïane and M. El-Hajj. Pattern lattice traversal by selective jumps. In *In Proc. 2005 Int'l Conf. on Data Mining and Knowledge Discovery (ACM-SIGKDD)*, August 2005.

[32] O. R. Zaïane, M. El-Hajj, and P. Lu. Fast parallel association rule mining without candidacy generation. In *Proc. of the IEEE 2001 International Conference on Data Mining*, 2001.

[33] O. R. Zaïane, J. Han, and H. Zhu. Mining recurrent items in multimedia with progressive resolution refinment. In *Int. Conf. on Data Engineering (ICDE'2000)*, pages 461–470, San Diego, CA, February 2000.

[34] M. J. Zaki and C.-T. Ho. Parallel and distributed association mining: A survey, in ieee concurrency, special issue on parallel mechanisms for data mining. In *Vol. 7, No. 4*, 1999.

[35] M. J. Zaki and C.-T. Ho. Large-scale parallel data mining, lecture notes in artificial intelligence, state-of-the-art-survey. In *Volume 1759, Springer-Verlag*, 2000.