Parallel Leap: Large-Scale Maximal Pattern Mining in a Distributed Environment

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 been suggested such as sequential mining for maximal patterns or searching for all 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 maximal patterns which, to the best of our knowledge, has never been proposed efficiently before. 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.

1. Introduction

The last decades have witnessed a massive growth in data collection techniques from different sources like satellite images, surveillance cameras, commercial domain transactions, etc.; this has led to huge archiving of data often without the ability to extract useful actionable information. The need to discover actionable knowledge from these massive data collections, for security, scientific or competitive reasons is obvious today. In the commercial domain alone, considering all the daily commercial transactions, or the goods movements and management with Radio Frequency Identification, RFID is phenomenal. Market competition motivates the timely discovery of useful patterns in the collected transactional data to gain competitive edge and help decision support. Data mining is the process in which hidden, implicit knowledge can be extracted from a store of databases or facts. The techniques have been proven very effective in many applications. However, while computers are getting faster and more powerful, they cannot sustain the tremendous increase in data collection we are able to amass today. New strategies are needed to scale with the amplified data gathering.

One of the major data mining techniques for pattern discovery and consequently one of the most studied in the data mining community, is association rule analysis in which strong relationships between co-occurring items in transactional data are discovered. Association rules are based on frequent itemset mining which is, simply put, the enumeration of sets of items frequently occurring together. The search (i.e. enumeration) is bound by count thresholds, known as support, or some other imposed constraints. Although mining for frequent itemsets is indeed necessary for association rule mining that is useful for customer behavior analysis or many other applications, frequent itemsets are valuable in many other knowledge discovery tasks, from the pre-processing of data to the characterization of discovered patterns. Frequent itemsets are constructive in building classification models, clustering data, discovering contrast sets, etc. In specific applications such as bio-informatics, frequent itemsets are an asset in micro-array analysis, protein structure prediction, etc. Hence, discovering frequent itemsets forms an essential canonical task in data mining.

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 [7]. 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.

Finding the set of frequent patterns is the first step in finding association rules. Once frequent itemsets are known, generating the association rules is trivial. Discovering the frequent patterns is essentially pinpointing some itemsets with high support in this massive lattice of candidates. In the literature, there are different approaches for efficient and effective counting and enumeration of frequent itemsets. Primarily, these approaches differ in the way they traverse the lattice, or search space. Most algorithms apply bottom up traversal of the lattice in order to enumerate the frequent itemsets. In other words, they search for short frequent patterns and build up on those that are frequent. Others might use top down search in cases of long frequent itemsets. They discover the long patterns before focusing on shorter ones. Some have also proposed hybrid strategies that merge top-down with bottom-up approaches.

Other fundamental differences between approaches are in the type of frequent patterns they aim at discovering. Rather than discovering all the frequent itemsets, one could discover a representative subset of these itemsets and then generate all the needed patterns. The set of frequent itemsets contains indeed many redundancies and could be represented by a smaller set called the frequent closed itemsets, or an even smaller set called the maximal frequent itemsets from which all the frequent itemsets can be generated. A detailed definition of frequent and maximal patterns is explained in the next section. The strategies aiming at these smaller subsets are typically faster and more scalable.

1.1. Problem Statement

The problem of mining frequent itemsets stems from the problem of mining association rules over market basket analysis as introduced in [2]. The problem consists of finding sets of items (i.e. itemsets) that are sufficiently frequent in a transactional database.

Formally, as defined in [2], the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items 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 $X \subseteq T$. An itemset X is said to be *frequent* if its *support* s (i.e. ratio of transactions in $\mathcal{D}$ that contain X) is greater than or equal to a given minimum support threshold σ. A frequent itemset $\mathcal{M}$ is considered maximal if there is no other frequent set that is a superset of $\mathcal{M}$. Consequently, any subset of a maximal pattern is a frequent pattern. Discovering all Maximal patterns effortlessly yields the complete set of frequent patterns. Therefore, we solely contemplate the discovery of maximals in this paper.

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 leap-traversal [17] that generates the set of maximal patterns. We show that using the traversal approach while parallelizing the mining approach 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 our leap-traversal approach and describe our proposed parallel approach in Section 3. We evaluate some strategies for load sharing and present performance results on experiments assessing scalability and speed-up in Section 4. Finally, we highlight some related work in Section 5 and conclude the paper.

2. The Leap Traversal Approach

Contrary to most existing parallel algorithms for mining frequent patterns, our algorithm is not apriori-based [1]. To mine for maximal patterns in parallel, 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 lattice traversal strategy HFP-Leap [17]. In our parallel approach, HFP-Leap still

performs the actual leap-traversal to find maximal patterns. We first present the idea behind HFP-Leap then show how this idea can be parallelized.

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.

Algorithm 1 HFP-Leap: Leap-Traversal with Headerless FP-tree

Input: D (transactional database); 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)
$Maximals \leftarrow$ FindMaximals(FPB, σ)
Output $Maximals$

2.1. Frequent Pattern Tree Construction

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. [9]. The tree structure we use, called HFP-tree is a variation of the original FP-tree. We 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 the same as the FP-tree except for 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 [9]. 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 tree 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* FPB 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).

Algorithm 1 shows the main steps in our approach. After building the Headerless FP-tree with 2 scans of the database, we mark some specific nodes in the pattern lattice using *FindFrequentPatternBases* where 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. Using the FPBs, the leap-traversal in *FindMaximals* discovers the maximal patterns at the frequent pattern border in the lattice.

2.2. The Leap-Traversal approach

Algorithm 2 is the actual leap traversal to find maximals using FP-trees generated all at one time using the Headerless FP-tree. It starts by listing some candidate maximals stored in *PotentialMaximals*, which is initialized with the frequent pattern bases that are frequent. All the non-frequent FPBs are used for the jumps of the lattice leap traversal. These FPBs are stored in the list *List* and intermediary lists *NList* and *NList2* will store the nodes in the

lattice that the intersection of FPBs would point to; in other words, the nodes that may lead to maximals. The nodes in the lists have two attributes: *flag* and *startpoint*. For a node n, *flag* indicates that a subtree in the intersection tree should not be considered starting from the node n. For example, if node ($A \cap B$) has a flag C, then the subtree under the node ($A \cap B \cap C$) should not be considered. For a given node n, *startpoint* indicates which subtrees in the intersection tree, descendants of n, should be considered. For example, if a node ($A \cap B$) has the startpoint D, then only the descendants ($A \cap B \cap D$) and so on are considered, but ($A \cap B \cap C$) is omitted. Note that $ABCD$ are ordered lexicographically. At each level in the intersection tree, when *NList2* is updated with new nodes, the theorems in [17] are used to prune the intersection tree. In other words, the theorems help avoid useless intersections (i.e. useless maximal candidates). The same process is repeated for all levels of the intersection tree until there is no other intersections to do (i.e. *NList2* is empty). At the end, the set of potential maximals is cleaned by removing subsets of any sets in *PotentialMaximals*.

It is obvious in the Leap-traversal approach that superset checking and intersections play an important role. We found that the best way to work with this is by using the bit-vector approach where each frequent item is represented by one bit in a vector. In this approach, intersection is nothing but applying the AND operation between two vectors, and subset checking is nothing but applying the AND operation followed by equality checking between two vectors. If $A \cap B = A$ then A is a subset of B.

3. Parallel Leap Traversal Approach

The parallel leap traversal approach starts by partitioning the data among the parallel nodes. 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. This distribution is discussed further below. Pruning algorithms are applied at each processor to reduce the size of the intersection trees [17]. Maximal patterns are generated at each node. Each processor then sends its maximal patterns to one master node, which filters them to generate the set of global maximal patterns. Algorithm 3 presents the steps needed to generate the set of maximal patterns in parallel.

Algorithm 2 FindMaximals: The actual leap-traversal

```
Input: FPB (Frequent Pattern Bases); σ (Support threshold).
Output: Maximals (Frequent Maximal patterns)

{which FPBs are maximals?}
List ← FPB; PotentialMaximals ← ∅
for each i in List do
    Find support of i {using branch supports}
    if support(i) > σ then
        Add i to PotentialMaximals
        Remove i from List
    end if
end for

Sort List based on support
NList ← List; NList2 ← ∅
∀i ∈ NList initialize i.flag ← NULL AND i.startpoint ← index of
i in NList

while NList ≠ ∅ do
    {Intersections of FPBs to select nodes to jump to}
    for each i in NList do
        g ← Intersect(i, j) {where j ∈ List AND i ≪ j (in lexico-
        graphic order) AND not j.flag}
        g.startpoint ← j; Add g to NList2
    end for
    for each i in NList2 do
        Find support of i {using branch supports}
        if support(i) > σ then
            Add i to PotentialMaximals
            Remove all duplicates or subsets of i in NList2; Remove i
            from NList2
        else
            if duplicates of i exist in NList2 then remove them except
            the most right one then remove i from NList2
            Remove all non frequent subsets of i from NList2
            if ∃j ∈ NList2 AND j ⊇ i then
                i.flag ← j
            end if
            for all j in List do
                if j ≫ i.startpoint (in lexicographic order) then
                    n ← Intersect(i, j)
                    Find support of n {using branch supports}
                    if support(n) < σ then
                        Remove i from NList2
                    end if
                end if
            end for
        end if
    end for
    NList ← NList2; NList2 ← ∅
end while

Remove any x from PotentialMaximals if (∃M ∈
PotentialMaximals AND x ⊂ M)
Maximals ← PotentialMaximals
RETURN Maximals
```

3.1. 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.

Algorithm 3 Parallel-HFP-Leap: Parallel-Leap-Traversal with Headerless FP-tree

Input: D (transactional database); σ (Support threshold).
Output: Maximal patterns 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**
  $LOCALMaximals \leftarrow$ FindMaximals(FPB, σ);
end for

- Send all $LOCALMaximals$ to the master node;
- The master node prunes all $LOCALMaximals$ that have supersets itemsets in $LOCALMaximals$ to produce $GLOBALMaximals$;
- The master node outputs $GLOBALMaximals$.

3.2. Example: Parallel Leap Traversal

The following example illustrates how the leap traversal approach is applied in parallel. Figure 1.A presents 7 transactions made of 8 distinct items which are: A, B, C, D, E, F, G, and H. Assuming we want to mine those transactions with a support threshold equal to at least 3, using two processors, Figure 1 illustrates 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 1.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 1.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 (Figure 1.C).

Each processor is assigned a set of header nodes to build their intersection tree as in Figure 1.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 $ACDE$, BCD, and ABE, with support equal to 3, 3, and 3 respectively. The second assigned tree produces CDF with support equal to 3. $P1$ produces 4 local maximals which are BE, AE, BE, CDF with support equal to 4, 4, 4, 3 respectively. $P2$ produced CDF, BE, and AE with support equal to 3, 4, and 4 respectively. All local maximals are sent to $P1$ in which any local maximal that has any other superset of local maximals from other processors are removed. The remaining patterns are declared as global maximals (Figure 1.E).

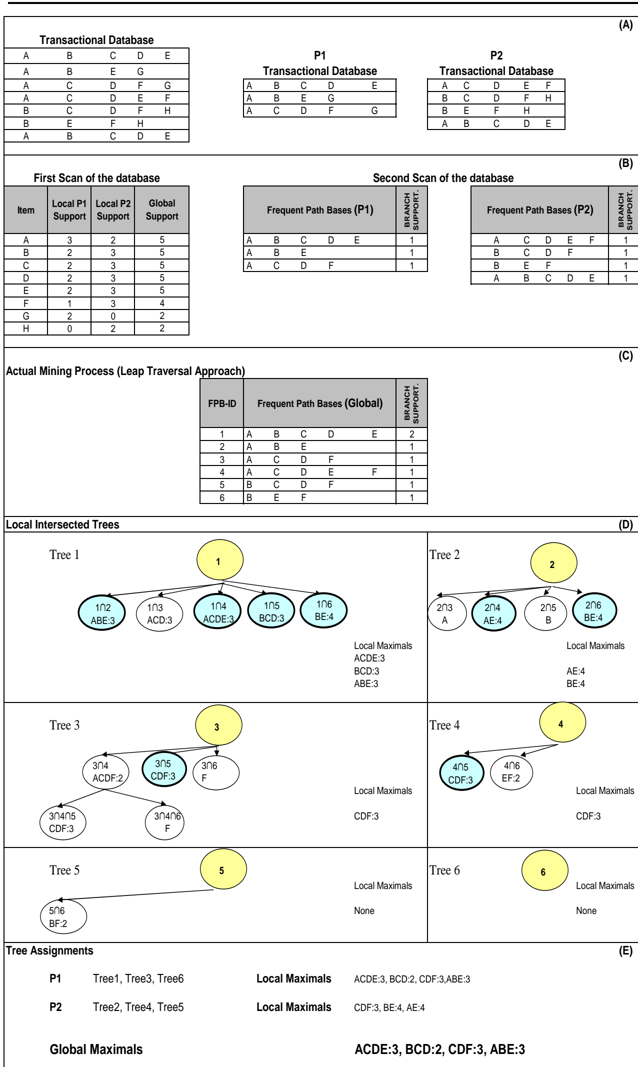

Figure 1. Example of Parallel Leap Traversal: Finding and intersecting the path-bases

4. Performance Evaluations

To evaluate our parallel leap-traversal approach, we conducted a set of different experiments using a cluster made of sixteen 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 [10]. 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 did not compare our algorithm against any other existing algorithm.

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.

4.1. 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 our suggested First-Last approach.

From our experiments in Figure 2 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.

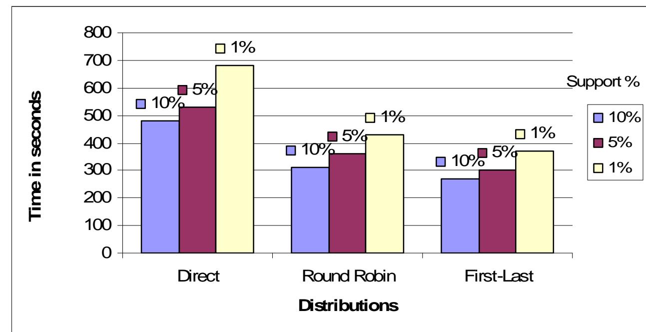

Figure 2. Effect of Leap node distributions

4.2. Scalability: 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. 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%. Figure 3.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. We were able to mine one billion transactions in 5020 seconds for a support of 0.1% up to 6100 seconds for a support of 0.01%.

4.3. Scalability: 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 3.B). 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 3.C.

5. Related work

In the realm of association rules, existing parallel frequent itemset mining algorithms are divided among two parallel environments which have 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 [16]. In any of these environments 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 [14], Parallel Partition algorithm [15], Fast Distributed Mining algorithm [5], Fast Parallel Mining algorithm [5], Parallel Data Mining algorithm [11].

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 [14], Intelligent Data Distribution algorithm [8].

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 [3] is presented in [6]. 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 [13], derived from the sequential FP-growth algorithm [9], 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 [4], Parallel Eclat, MaxEclat, Clique, MaxClique, TopDown, and AprClique algorithms all reported in [12]. These algorithms are mainly apriori-based and suffer from expensive candidacy generation and communication costs. Multiple Local Frequent Pattern tree Algorithm [18], which was among the first non apriori-based parallel mining algorithm, was our 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.

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.

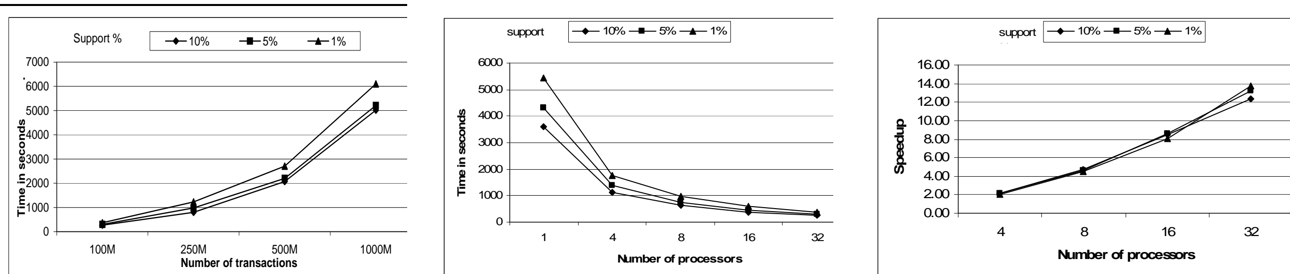

Figure 3. A. Scalability: Different transaction Size TS (*p*= 32)), B. Scalability: Different number of processors, C. Speedup: Different support values (TS = 100M For Figures B and C)

6. 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 maximal patterns 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 maximal patterns. 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.

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

[4] 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, NY*, pages 279 – 288, 1998.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.