Parallel Binary Approach for Frequent Itemsets Mining

Boutheina Missaoui, Khedija Arour, and Yahya Slimani

Abstract—The technique of association rules discovering is one of the most known and the most explored techniques of data mining. This technique has two main phases: the first is to extract all the frequent itemsets and the second is to generate association rules from these frequent itemsets. The first phase is the most expensive given the large number of accesses to transactions database and the large number of candidate itemsets. As databases are generally very large, a solution to avoid the repetitive and costly access is to represent them by compact structures. In this paper, we propose a parallel binary approach for frequent itemsets extracting, to deal with the great number of candidates and to take advantage of multicore architectures. This approach is implemented using a compact data structure based on signatures tree for the representation of the database to access it only once.

Index Terms—Data structure, frequent itemsets, open MP, parallelism.

I. INTRODUCTION

Association rule mining is very used in different domains [1]. Sequential algorithms of frequent itemsets mining (like Apriori [2], FP-Growth [3]) are not enough efficient with the growth of the physical storage capacity and the need to saving data, hence the need to introduce parallel algorithms. In fact, parallelism becomes very used to increase applications speed and to resolve larger problems. It allows exploiting modern architectures as well.

We propose a parallel solution for frequent itemsets mining. Currently, the processors know a rapid evolution in the number of cores (within a single processor of a machine) that maybe will reach many ten and we should benefit from this growth. Our solution is dedicated to multicore machines with shared memory and implemented using the OpenMP tool (see http://www.openmp.org).

This paper is organized as follows: In Section II, we survey a state of art. Section III presents our proposed approach we call P-STM (Parallel - Signatures Tree Mine), the construction process of the data structure STree used by it, the search process of a signature in STree, and the process of generating frequent itemsets. In Section IV, we present and discuss some experimental results of our proposition. Finally, Section V concludes and presents some future works.

II. STATE OF THE ART

In order to improve the efficiency of the frequent itemsets mining algorithm, parallelism has been used on many algorithms but with different degrees of success. Most of these parallel algorithms are based on Apriori because it is simple and easy to implement. However, they still access the database many times. There are other algorithms which could be parallelized. PD-CLUB is a parallel version of the algorithm D-CLUB which is based on bitmaps. The basic idea of this parallelization is to dynamically regroup itemsets together with their bit vectors associated to clusters and divide the task of frequent itemsets extraction to smaller sub-tasks performed independently in parallel: each one operates a cluster of itemsets. Eclat is another algorithm which was also parallelized (ParEclat, ParMaxEclat...). Parallelism is employed here by dividing the extraction tasks for different equivalence classes of itemsets independently assigned to available processors and distributing the tid-lists associated. These algorithms based on Eclat do more than one access to the database to extract frequent itemsets and suffer from the communication cost that comes from the exchange of local tid-lists. FP-Growth was also parallelized by assigning the conditional FP-Trees to different processors. The major overhead of communication and synchronization is the exchange of local conditional pattern bases. The parallel algorithms of frequent itemsets generating based on FP-Growth still access the database twice like it. The benefit is very limited for sparse databases and its advantage for dense ones does not increase accordingly with the number of processors [4]–[8].

Several attempts to parallelize many algorithms of frequent itemsets extracting have been proposed using task parallelism or data parallelism on machines with shared memory or distributed memory. The application of parallelism was either on candidates’ calculation or transactions database usage. Each one of these parallel algorithms has its advantages and disadvantages. There is no efficient algorithm for all cases.

III. P-STM: PARALLEL APPROACH FOR GENERATING FREQUENT ITEMSETS

There are many ways to optimize the frequent itemsets mining algorithm [4]. We explore two ways of optimization: using a compact structure to access databases only once and the parallelization to deal with a large number of candidates. This section presents a structure called STree and our parallel algorithm P-STM that uses it to extract frequent itemsets.

A. The Construction of STree Structure Used by P-STM

Our approach uses a compact structure that we called...
**STree** (Signatures Tree) to represent the transactions database. **STree** is based on the tree of signatures used primarily in the field of information research and proposed by Chen [9]. Each transaction is represented by a signature which is a sequence of bits of fixed size obtained by using one (or many) hash function(s). In **STree**, all signatures are different and the number of leaves is equal to the number of different signatures. It is a binary tree that contains two types of nodes:

1) An internal node has two children (left and right). The left edge has the value 0 and the right edge corresponds to the value 1. It contains a positive number called position that denotes the bit position of the signature to be controlled.

2) A leaf node points to a signature and contains the number of transactions that generate this signature. It is a special purpose that we add to support the presence of identical signatures and resolve the problem of collisions (states in which two different data have the same signature).

So we use the algorithms of construction, research and update detailed in [9] but with some modifications.

To construct **STree**, first we construct a root node containing the first signature \( S_1 \) and the number of transactions generating \( S_1 \) (initially equal to 1). Then we insert each new signature \( S \) (of next transaction) in **STree**. We traverse the tree from the root. Let \( v \) be the node encountered. If \( v \) is an internal node (first case), we control the bit in \( S \) corresponding to the number \( j = \text{position}(v) \) (found in \( v \)). If \( S[j] = 1 \), we go right else we go left. If \( v \) is a leaf node (second case), we compare \( S \) with the signature \( S' \) in \( v \). If \( S = S' \), we increment the number of transactions \( nt \) in the leaf \( v \). Else \( S \) is a new signature that does not already exist in **STree**. In this case, we assume that the first \( k \) bits of \( S \) and \( S' \) are identical. Therefore \( S \) differs from \( S' \) in the position \( k+1 \). A new internal node \( u \) is built with position (\( u \)) \( = k+1 \) and \( v \) will be one of its children. The node \( u \) takes the place of \( v \). If \( S[k+1] = 1 \), \( v \) becomes the left child of \( u \) and a new leaf node containing \( S \) (with \( nt = 1 \)) is its right child. Otherwise \( v \) is the right child and the node of \( S \) is the left one [10].

**STree** uses hash function to transform transactions to binary signatures. The problem is the presence of false positives (selected signatures which are not really relevant, called also false drops or false hits). Two main factors influence the rate of false positives: the signature size and the choice of hash function. Our goal is to find a function that gives reasonable size signature and avoids as possible the collisions to reduce false positives number. There are many hash functions. They can be classified into two classes: cryptographic (SHA-MD family: MD5, SHA1... is the most known.) and non cryptographic (Modulo...).

Table I shows an example of signature generation of the transaction \( Tr \) (composed of followed items \( \{1, 3, 4\} \) using Modulo function \( H \)). Table II gives signature counting of \( Tr \) using MD5 function. Signatures items are superimposed to give the signature of \( Tr \).

The example given in Table II shows that MD5 (or any cryptographic function) when transformed to binary format gives a very high score of bits equal to 1. This is because of the superimposition between items signatures to obtain the transaction signature. That is why we eliminated the usage of cryptographic function (alone) from our choices.

### Table I: Example of Signature Generation: \( H = \text{mod} 8, \text{SIZE} = 8 \)

<table>
<thead>
<tr>
<th>Item</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01000000</td>
</tr>
<tr>
<td>3</td>
<td>∨</td>
</tr>
<tr>
<td>4</td>
<td>∨</td>
</tr>
</tbody>
</table>

- Signature of \( Tr \) = 01011000

### Table II: Size MD5 Signature \( l = 128 \); Signature (\( \text{Tr} = \text{Signature (ITEM1=}1 \text{, ITEM2=}3 \text{, ITEM3=}4 \text{)} \))

<table>
<thead>
<tr>
<th>Item</th>
<th>MD5 Signature (binary format)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1100 0100 1100 1010 0100 0010 0011 1000 1010 0000 1011 0001 1010 0100 1011 1010 0111 1110 1000 1000 0000 1000 1111 1011 1001 0000 1010 0101 0001 0010</td>
</tr>
<tr>
<td>2</td>
<td>1110 1100 1000 1100 1011 1000 0111 1110 1010 0100 1011 0110 1111 1100 0110 1111 1110 1111 1111 1000 1110 1111 1111 1111 1111 1001</td>
</tr>
<tr>
<td>3</td>
<td>0110 0100 0111 1111 1110 0110 0111 0111 1001 1010 0100 1011 0110 1011 1111 1111 1111 1111 1111 1111 1111 1111 1001 1010 0100 1000 0001 1010 1010</td>
</tr>
<tr>
<td>4</td>
<td>1111 1110 0101 1111 1111 1111 1111 1111 1001 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111</td>
</tr>
<tr>
<td>Tr</td>
<td>1110 1100 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111</td>
</tr>
</tbody>
</table>

We tested both functions: cryptographic (MD5) and non cryptographic (Modulo). We found that the choice of the right function depends on the characteristics of the database used. We choose Modulo (greater than or equal to the number of items in the databases). For databases that contain a large number of items like Retail we choose Modulo 1024 (chosen maximum signature size =1024).

Table III shows an example of transaction database. The relevant stages of construction of its structure **STree** are shown in Fig. 1. Transactions \( T_1 \) and \( T_4 \) generate the same signature (collision). It will be denoted \( S_1 \) in **STree**.

### Table III: Example of Transactions Database and the Corresponding Signatures; Hash Function: [MOD 8]; Signature Size=8 Bits

<table>
<thead>
<tr>
<th>TID</th>
<th>Transactions</th>
<th>Signatures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,4,7,8</td>
<td>S_1</td>
</tr>
<tr>
<td>2</td>
<td>1,3,4,5,13</td>
<td>S_2</td>
</tr>
<tr>
<td>3</td>
<td>1,2,4,5,7,11</td>
<td>S_3</td>
</tr>
<tr>
<td>4</td>
<td>1,3,4,7</td>
<td>S_4</td>
</tr>
</tbody>
</table>

(1) Insert \( S_1 \); (2) Insert \( S_2 \); (3) Insert \( S_3 \); (4) Insert \( S_4 \);

Fig. 1. Steps of corresponding **STree** construction for Table III.

B. The Research of an Itemset Signature in **STree**

The support of an itemset is estimated by the search of its signature in **STree**. Overestimating the support value is probable. This is due to the fact that the signatures tree is an inexact representation of the database. But it can never be underestimated and we call it sup_max (maximum support). It is equal to the sum of transactions numbers \( nt \) found during
the research process in \textit{STree}. To search a signature \( S_I \) of an itemset \( I \), we traverse the tree. Let \( v \) be the node encountered and \( j=\text{position} (v) \) the position to control. If \( S_I[i] = 1 \), the corresponding bit in a signature of the tree should be 1 and we visit only the right child; else it can be 1 or 0 and we traverse right and left children. This reflects that only signatures verifying the condition \( S_I \land S = S_I \) will be selected [10].

For example, let \( I \) be the itemset composed of items \{2, 8\}. We apply the hash function \( H= [\text{mod} 8] \) to obtain the signature \( S_I = 10100000 \). Fig. 2 represents in bold the path of \( \text{STree} \) visited when looking for \( S_I \), \( S_I[1] = 0 \) (root=1) then we visit the right child (\( S_I \)) and the left child (3). \( S_I[3] = 1 \) so we visit only the right child (\( S_I \)). Two signatures \( S_I \) and \( S_3 \) are visited and both of them does not satisfy the condition \( S_I \land S = S_I \) thus no one is selected and \( \text{supmax}(I) = 0 \).

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{signature_search_path}
\caption{Example of a signature search path in \textit{STree}.}
\end{figure}

### C. Extracting Frequent Itemsets with \textit{P-STM}

\textit{P-STM} begins with a sequential region which is the construction of \textit{STree} and the identification of the frequent items by one access to the database. Then the algorithm move to generate and test candidate \((k+1)\)-itemsets (generated from the set of frequent \( k \)-itemsets).

With \textit{P-STM}, we take advantage of modern multicore machines which are accessible for public. We use data parallelism with \textit{OpenMP} (http://www.openmp.org/) which is an effort to define a standard for shared memory parallelization with fewer details to manage even to maintain and ease of implementation compared to others. It offers optimal scalability and efficient usage of memory. It is a library giving instructions to the compiler to allow execution of \textit{C++} or \textit{Fortan} program on multithreading. Table IV lists the directives that we used.

| **TABLE IV: LIST OF OPENMP DIRECTIVES USED BY P-STM** |
|-------------------|------------------|-----------------|
| **Name**          | **Type**         | **Utility**     |
| #include <omp.h>  | directive        | include \textit{OpenMP} file |
| omp_nested        | environment      | activate nested |
| #pragma omp       | directive        | define parallel region |
| parallel           | directive        | parallelize a loop |
| #pragma omp for    | function         | share a variable |
| shared             | function         | announce a lock |
| omp_lock_t         | function         | initialize a lock |
| omp_init_lock      | function         | lock by a thread |
| omp_unset_lock     | function         | unlock |
| omp_destroy_lock   | function         | destroy the lock |

We realize the parallelization at two levels, at the level of research in \textit{STree} and the level of candidates pruning:

### D. The Parallel Search in \textit{STree}

A binary tree (like \textit{STree}) had features which allow the parallelism. Indeed, we notice symmetry between its branches that are independent. So research in \textit{STree} can be done in parallel where each subtree can be assigned to a thread. For example if we use two threads, the left subtree is assigned to a thread and the right one is given to another. Each thread that reaches a leaf, increments the variable \textit{supmax} if the leaf satisfies the condition required as seen. The threads can access \textit{supmax} at same time. So this variable is shared between them. This is done using the directive "shared" with \textit{OpenMP} and we manage the synchronization using a variable lock with type \textit{omp_lock_t} (see Table IV).

#### E. The Parallel Pruning of Candidates

In sequential approach, we need considerable computational resources for candidates pruning. It realizes \( n \) independent treatments that can be parallelized, where \( n \) is the number of candidates. One treatment is to generate the signature of a candidate itemset, to search it in \textit{STree} to compute its support and then to check if it is frequent (to keep) or not (to prune it). If we distribute the work to do on multiple processors, where each thread handles a certain number of candidates, the extraction time will be significantly reduced. In our algorithm \textit{P-STM}, the pruning is done in parallel by parallelizing the loop "for" with the directive \textit{omp for} of \textit{OpenMP}. If we have e.g., 4000 candidates and 4 threads to use, each thread is responsible of 1000 candidates.

The following is the formal description of \textit{P-STM}.

**Algorithm P-STM Algorithm**

**Input:** transactions database \( D \)

**Output:** all frequent itemsets set \( FI \)

**Begin**

Initially, \( FI \) is empty

\( GEN\_STree (D) /*STree construction */ \)

4: \( FI = FI \cup 1\text{-FI} /* frequent 1\text{-itemsets }*/ \)

\( k = 1; \)

while \( |k-FI| > 1 \) do /*\( k-FI \): frequent \( k \)-itemsets */

\#pragma omp parallel shared(FI) num_threads(4)

\{#pragma omp for nowait schedule(dynamic)

\( i = 1, n \rightarrow 1 \) do

10: for \( j = i+1, n \)

/*generate \((k+1)\)-candidate \( Cij \) (join2 elements)*/

\( \text{Scij} \leftarrow GEN\_SIG (Cij) \)

ParallelResearch(Scij, supmax)

14: if \text{supmax} \geq \text{minsup} then

\( FI = FI \cup \{Cij\} /*\text{using lock}*/ \)

end if

end for

18: end for

\( k++; \)

end while

return \( FI \)

22: end

IV. EVALUATION OF \textit{P-STM}

The set of experiments has been done on a laptop with an Intel Core i3 370M 2.4 GHZ with 3 GB of RAM. Our
algorithm is implemented with C++ using the standard library STL and Boost library version 1.47.0 (boost::dynamic_bitset to generate signatures). The code is cross-platform (running on both platforms OpenSuSE 11.3 and Windows 7) and was compiled with gcc version 4.6.1. We tested our algorithm on different transactions databases from FIMI workshop (http://fimi.cs.helsinki.fi/data): dense databases (Mushroom, Chess, and Accidents) and sparse databases (T10I4D100K, T40I10D100K, and Retail).

We compared our results with those obtained by the reference Apriori (http://adrem.ua.ac.be/~goethals/software). The frequent itemsets extracted by our approach are the same as obtained by Apriori for all used databases except Retail (with few false drops in addition).

In first experiment, we compare our approach P-STM (using STree) and FP-Growth (using FP-Tree, http://adrem.ua.ac.be/~goethals/software/) and the comparison criterion is the memory used. Then we evaluate P-STM by using measures of quality for parallelization which are the speedup and the efficiency.

A. Memory Consumption

The structure used by FP-Growth is FP-Tree (n-airy tree). Its size, when representing a database, depends on minsup and explodes for low supports. In worst cases it is equal to \( T^*M \); where \( T \) is the number of transactions and \( M \) is the average size of transactions [11]. Or, STree used by P-STM, has a constant nodes number and constant memory for all values of minsup. The number of nodes in worst cases is equal to \( 2^*T-1 \) (leaf nodes number = \( T \); internal nodes number =\( T-1\)).

We compare P-STM (using 1 thread) and FP-Growth using the memory consumption criterion. Values are in kilo bytes.

![Fig. 3. P-STM vs FP-Growth for Mushroom database.](image)

![Fig. 4. P-STM vs FP-Growth for T10I4D100K database.](image)

Fig. 3 gives a comparison between the two algorithms for an example of dense databases: Mushroom. The two curves have almost the same shape but FP-Growth is better. That proves the effectiveness of FP-Growth for such databases (dense ones) [3], [11].

Fig. 4 concerns an example of sparse databases: T40I10D100K. For FP-Growth, we notice a rapid growth of memory consumption when reducing minsup as shown in the grey curve (because of frequent projections done by the algorithm and the number of auxiliary structures which increases consequently). P-STM is better in this case. It consumes less memory than FP-Growth.

B. Speed Up

The first measure we use to evaluate our parallel algorithm P-STM is the speedup. It is the sequential time (on one core or thread) by the parallel time (on \( C \) core/thread).

Speed up: \( S(C) = \frac{\text{Sequential Time}}{\text{Parallel Time}} \)

In theory we suppose that time will be divided into \( C \). This vision ignores two essential points: the scalability on parallel architecture and the overhead linked to parallelism. Speed up is called linear (ideal) when \( S(C) = C \). In practice, it exists cases where \( S(C)>C \) and it is called superlinear [12].

The first histogram (Fig. 5) shows the speed up of P-STM for an example of dense databases (Mushroom) using 2, 4, and 8 threads with different values of minsup. Values are between 1 and 2. We notice that the speed up is improving by decreasing the minimum support (minsup). P-STM performs better when number of candidates and data quantity are more important.

![Fig. 5. Speed up of P-STM for Mushroom database.](image)

The second histogram (Fig. 6) shows speed up values for a sparse base T10I4D100K with different values of minsup and threads. It varies between 1 and 4.5 (superlinear speedup). Values improve with decreasing minsup.

For high supports, STree construction part is largely more important than treatment (pruning) candidates. Or for low supports, candidates pruning part increases then parallelism contribution is more significant. The machine contains 2
cores with hyper-threading (4 threads or 4 logical cores) technology which is able to execute two threads on a core at a time. Therefore, it increases probability for threads to take advantage of memory locality; thereby reaching to the point of the superlinear efficiency [12]. Logically we should use a threads number equal or superior to the number of processors existed in the machine, to profit of all power. Best values of speed up are obtained with 4 threads.

C. Efficiency

The efficiency is another measure of quality for parallelism. It is the speed up on the number of threads used (or cores).

Efficiency: \( E(C) = \frac{S(C)}{C \times T} \) (C threads).

The efficiency is called linear when \( E(C) = 1 \). When \( S(C) > C \) or \( E(C) > 1 \) the efficiency is called super-linear. It is due to larger effective cache size on shared memory multiprocessor and multicore systems [12].

Table V shows the efficiency values of \( PSTM \) for an example of dense databases \( \text{(Mushroom)} \) with different values of \text{minsup} \. Values are improved when reducing \text{minsup}. The efficiency varies between 0.5 and 0.7 using two threads. These values are the best ones in this table. Table VI shows the efficiency values of \( PSTM \) for an example of sparse databases: \( T104D100K \). Values are also improved by decreasing the values of \text{minsup} \ and they are better than the previous example. There are some cases of superlinear parallelism \((E(C) > 1)\) proving its performance.

<table>
<thead>
<tr>
<th>TABLE V: EFFICIENCY OF PSTM FOR MUSHROOM DATABASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{minsup}</td>
</tr>
<tr>
<td>90%</td>
</tr>
<tr>
<td>80%</td>
</tr>
<tr>
<td>70%</td>
</tr>
<tr>
<td>60%</td>
</tr>
<tr>
<td>50%</td>
</tr>
<tr>
<td>40%</td>
</tr>
<tr>
<td>30%</td>
</tr>
<tr>
<td>20%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE VI: EFFICIENCY OF PSTM FOR T104D100K DATABASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{minsup}</td>
</tr>
<tr>
<td>7%</td>
</tr>
<tr>
<td>6%</td>
</tr>
<tr>
<td>5%</td>
</tr>
<tr>
<td>4%</td>
</tr>
<tr>
<td>3%</td>
</tr>
<tr>
<td>2%</td>
</tr>
<tr>
<td>1%</td>
</tr>
</tbody>
</table>

V. CONCLUSION

Data structures used by the algorithms of frequent itemsets mining have high costs. Our parallel algorithm \( PSTM \) uses \( STree \) structure to represent transactions database. It gives all necessary information for the process of frequent itemsets mining. It has a constant size with no necessity to reconstruction or auxiliary structures (like \( FP-Tree \)) and does not depend on \text{minsup}\ prefixed. The constant memory consumption of \( STree \) influences positively on the performances of \( PSTM \) which exploit \( STree \) to extract the frequent itemsets. It consumes less memory than \( FP-Growth \) for sparse databases and for small supports. Generally the problem of multiple accesses of the databases remains for other algorithms. \( PSTM \) accesses the database only once.

\( PSTM \) has good values of speed up and efficiency especially with low supports. The usage of parallelism improves the execution time and allows the treatment of more voluminous databases. We envisage to experiment \( PSTM \) on machines more larges (with more cores) and to develop hybrid distributed version with the library MPI (Message Passing Interface) [13]. Another perspective is to test another way to proceed for generating candidates: to generate only signature candidates (without generating candidate itemsets).

ACKNOWLEDGMENT

B. Missaoui thanks her family especially her father Amor and everyone supported her.

REFERENCES


Boutheina Missaoui got her bachelor degree in Computer Science applied in management from ESCT, Tunis, Tunisia in 2008. She got her master degree from the High Institute of Management of Tunis (ISG-Tunis) recently in 2012. Her research interests include Ontology, Parallel Algorithms and Data mining.
Khedija Arour received her Engineering diploma and Ph.D. degrees from the department of Computer Science of the Science Faculty of Tunis, Tunisia in 1992 and 1996 respectively. She is currently an assistant professor in the department of Computer Science and Mathematics at National Institute of Science and Applied Technology of Tunis, Tunisia, Carthage University. Dr. AROUR’s research interests are mainly on haute performance data mining and large scale information retrieval systems.

Yahya Slimani studied at the Computer Science Institute of Alger's (Algeria) from 1968 to 1973. He received the B.Sc. (Eng.), Dr Eng and PhD degrees from the Computer Science Institute of Alger's (Algeria), University of Lille (France) and University of Oran (Algeria), in 1973, 1986 and 1993, respectively. He was Full Professor at the Department of Computer Science of Faculty of Sciences of Tunis and he is currently at ISAMM of Manouba (Tunisia). These research activities concern Data mining, Parallelism, Distributed systems, Grid and Cloud Computing. Prof. Yahya Slimani has published more than 210 papers from 1986 to 2012. He contributed to Parallel and Distributed Computing Handbook, Mc Graw-Hill, 1996. He is member of Editorial Boards of several international journals and chair of international conferences.