2010 IEEE International Conference on Data Mining

PGLCM: Efficient Parallel Mining of Closed Frequent Gradual Itemsets

Trong Dinh Thac Do*†, Anne Laurent† and Alexandre Termier*
*LIG, CNRS UMR 5217
Grenoble University, Grenoble, France
Email: dot@imag.fr, Alexandre.Termier@imag.fr
†LIRMM, CNRS UMR 5506
University Montpellier 2, Montpellier, France
Email: laurent@lirmm.fr

***Abstract*—Numerical data (e.g., DNA micro-array data, sensor data) pose a challenging problem to existing frequent pattern mining methods which hardly handle them. In this framework, gradual patterns have been recently proposed to extract covariations of attributes, such as: *"When X increases, Y decreases"*. There exist some algorithms for mining frequent gradual patterns, but they cannot scale to real-world databases. We present in this paper GLCM, the first algorithm for mining closed frequent gradual patterns, which proposes strong complexity guarantees: the mining time is linear with the number of closed frequent gradual itemsets. Our experimental study shows that GLCM is two orders of magnitude faster than the state of the art, with a constant low memory usage. We also present PGLCM, a parallelization of GLCM capable of exploiting multicore processors, with good scale-up properties on complex datasets. These algorithms are the first algorithms capable of mining large real world datasets to discover gradual patterns.**

***Keywords*-Data mining; frequent pattern mining; gradual itemsets; parallelism;**

I. Introduction

Frequent pattern mining is the component of data mining focused on extracting *patterns* that occur frequently in data. These patterns can be seen as abstractions of the contents of large datasets, potentially providing insightful information. Most of the works on frequent pattern mining have focused on categorical data, either mere sets (frequent itemset mining) [1], or more complex data having a structure of sequence [2], tree [3] or graph [4]. Few of these works have focused on numerical data. There have been some works on quantitative itemset mining, where items can have numerical values [5]. These works focused on discretizing numerical data in order to handle them the same way as categorical data. Thus, only a small part of the information present in the numerical data was exploited. Despite some improvements [6], [7] over original works on quantitative association rule mining, analyzing numerical data remained marginal in the field of frequent pattern mining.

However, most corporate data and many scientific datasets have many attributes which are numerical: sales numbers, prices, ages, expression level of a gene, quantity of light received by a sensor, etc.

Recently, a new pattern mining field has emerged for analyzing such data: mining of *gradual patterns*. Gradual patterns can be expressed as covariations of several attributes, for example: *"the higher the age, the higher the salary, the lower the free time"*. An algorithm based on Apriori has been proposed [8] for mining gradual patterns. This algorithm, as the original Apriori, can mine simple datasets, but it does not scale on large real-world datasets.

Apriori is the pioneer of all algorithms for mining frequent itemsets, however state-of-the-art algorithms considerably outperform it. One of the major steps was the works of Pasquier et al. [9], which showed that it was sufficient to mine *closed* frequent itemsets, with run time improvements over an order of magnitude in many cases. Later, the FIMI'04 workshop [10] made a competition between all closed frequent itemset mining algorithms. The winner was the LCM algorithm [11].

LCM is based on a theoretical improvement: its authors showed that a closed frequent itemset could be computed by extension of a unique other closed frequent patterns. The closed frequent itemsets are thus the nodes of a covering tree, over which efficient depth first search strategies can be applied, with a very low memory usage.

Our goal is to exploit the principles giving the good performances of LCM in order to compute efficiently gradual itemsets over large real-world databases. The contribution of this paper is threefold:

- We show that likewise itemsets, it is possible to build a covering tree over the search space of closed frequent gradual itemsets.
- We present an algorithm and an implementation for mining efficiently closed frequent gradual itemsets, based on the principle of the LCM algorithm. This algorithm, GLCM, has the same strong complexity guarantees as LCM: its time complexity is linear with the number of closed frequent gradual itemsets, and its memory complexity does not depend on this number. We experimentally show that our algorithm significantly improves the state of the art.
- Last, we show a simple parallelization of our algorithm, using the Melinda library [12]. We experimentally

1550-4786/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDM.2010.101

tid	age	salary	loans	cars
t_1	22	2,500	0	3
t_2	35	3,000	2	1
t_3	33	4,700	2	1
t_4	47	3,900	1	2
t_5	53	3,800	2	2

Table I
EXAMPLE DATASET

show that the parallel version can take advantage of recent multicore processors and handle large real-world datasets.

It is especially interesting to note that despite LCM proven efficiency, algorithms based on its principle have mostly remained theoretical (for example [13]). By showing how to design and implement an efficient algorithm with this principle, we hope to help the diffusion of the ideas found in LCM, which are still the key to efficient pattern mining algorithms.

The paper is organized as follows: in Section II, we present in detail the concept of gradual patterns and related works. In Section III, we recall the base principles of LCM, show how they can be applied to gradual patterns, and present our algorithm and its parallelization. Section IV presents detailed experiments both about the sequential and the parallel version of our algorithm. We conclude and give directions for future research in Section V.

II. GRADUAL ITEMSETS AND RELATED WORKS

As described above, gradual itemsets (also known as gradual patterns) refer to patterns like *"the higher the age, the higher the salary"*. They can be compared to fuzzy gradual rules that have first been used for command systems, for instance for braking systems: *"the closer the wall, the stronger the brake force"*. However, in this framework, rules are provided by human experts. More recently, it has been shown that such rules can be mined [14], [15], [8], and can even benefit from the new multicore architectures [16]. In order to tackle the problem of the number of patterns generated by the approach, [17] introduces closed gradual itemsets.

In this framework, the authors deal with gradual items, gradual itemsets and closed gradual itemsets as defined below.

A. Preliminary Definitions

A dataset is a set of tuples $\mathcal{R}$ defined over the schema $\mathcal{S} = \{I_1, \ldots, I_n\}$.

Table I shows an example dataset where $S = \{age, salary, loans, cars\}$.

A gradual item is a pair (i, v) of an item (attribute) $i \in S$ and a variation $v \in \{\uparrow, \downarrow\}$ where $\uparrow$ stands for a positive (ascending) variation and $\downarrow$ for a negative (descending) variation.

A gradual itemset is defined as a non-empty set of gradual items. For instance, the gradual itemset $P_1 = \{(age, \uparrow), (salary, \uparrow)\}$ means *"the higher the age, the higher the salary"*.

The evaluation of the support of such gradual itemsets has been defined in different manners depending on the authors. [15] is based on regression, while [14], [18] and [19] consider the number of tuples that are concordant and discordant, in the idea of exploiting the Kendall's tau ranking correlation coefficient [20]. In [8], the authors consider another definition of the support based on the maximum proportion of tuples that can be ordered according to the gradual itemset.

In this framework, let us consider a gradual itemset $P = \{(i_{k_1}, v_{k_1}), \ldots, (i_{k_j}, v_{k_j})\}$ where $\{k_1, \ldots, k_j\} \subseteq \{1, \ldots, n\}$ and the $k_1, ..., k_j$ are all distinct. Two tuples t and t' can be ordered with respect to P if all the values of the corresponding items from the gradual itemset can be ordered to respect the respective variation: for every $l \in [k_1, k_j]$, $t.i_l \leq t'.i_l$ if $v_l = \uparrow$ and $t.i_l \geq t'.i_l$ if $v_l = \downarrow$. The fact that t precedes t' in the order induced by P is denoted $t \lhd_P t'$.

For instance, from Table I, it can be seen that t_1 and t_2 can be ordered with respect to P_1 as $t_1.age \leq t_2.age\ AND\ t_1.salary \leq t_2.salary$: we have $t_1 \lhd_P t_2$.

Let $L = \{t_1, ..., t_m\}$ be a list of tuples from $\mathcal{R}$ and P be a gradual itemset. L respects P if $\forall i \in [1, m-1]$ we have $t_i \lhd_P t_{i+1}$. Let $\mathcal{L}_P$ be the set of lists of tuples that respect P.

The formal definition of the support of P is $support(P) = \frac{max_{L \in \mathcal{L}}(|L|)}{|\mathcal{R}|}$, i.e. it is the size of the longest list of tuples that respects P. Note that the support of an itemset containing a single item is always 100% as it is always possible to order all the tuples by one column.

The rest of this paper is based on this definition of support.

From the implementation point of view, it seems that [8] is the most efficient one so far. It deals with binary matrices to represent how tuples are ordered (adjacency matrix) with regard to a gradual itemset. As shown below, this representation is very efficient, especially as it allows to apply AND binary masks for joining gradual itemsets represented by their matrices. However, as the problem is similar to determining the longest path in a graph, it can be time consuming. Moreover, managing all the matrices can lead to performance degradation due to memory consumption.

For instance, Table II reports the way the information is stored in main memory and then processed for computing the support and joining itemsets. In this matrix, it can be seen that t_2 precedes t_2, t_4 and t_5 (value 1 in the matrix) but not t_1 and t_3. We have here: $support(\{(age, \uparrow), (salary, \uparrow)\}) = 3/5$ as P_1 is supported by the maximum ordered lists of tuples $< t_1, t_2, t_4 >$ and $< t_1, t_2, t_5 >$.

	t_1	t_2	t_3	t_4	t_5
t_1	1	1	1	1	1
t_2	0	1	0	1	1
t_3	0	0	1	0	0
t_4	0	0	0	1	0
t_5	0	0	0	0	1

Table II
BINARY MATRIX CORRESPONDING TO $\{(age, \uparrow), (salary, \uparrow)\}$

B. Closed Gradual Itemsets

Closed itemsets have been studied for many years as they represent one of the keys to manage huge databases and to reduce the number of patterns without loss of information. Generally speaking, p is said to be closed if there does not exist any p' such that $p \subset p'$ and $support(p) = support(p')$

Two closure operators have been defined in [17] for closed gradual itemsets: g and f. Given an ordered list of tuples S, f returns the gradual itemset (all the items associated with their respective variations) respecting all sequences in S. Given a gradual itemset P, g returns the set of the maximal lists of tuples which respect the variations of all gradual items in P.

For instance, $f(< t_1, t_5, t_3 >) = \{(loans, \uparrow), (cars, \downarrow)\}$ and $g(\{(age, \uparrow), (salary, \uparrow)\}) = \{< t_1, t_2, t_4 >, < t_1, t_2, t_5 >\}$

Provided these definitions, a gradual itemset p is said to be closed if $f(g(p)) = p$. The closure of p is denoted $Clo(p)$.

Compared to the context of classical items, the main issue here is to manage the fact that g does not return a set of tuples but it returns a *set of lists of tuples* that can be ordered.

As far as we know, these definitions have not been included by the authors within the mining process, but rather as a post-processing step which is not efficient. Indeed, it does not allow to benefit from the runtime and memory reduction and thus does not provide any added value for running the algorithms on huge databases. We thus propose below a novel approach to cope with this.

III. EFFICIENTLY MINING GRADUAL ITEMSETS

In this section, we present our algorithms for mining efficiently closed frequent gradual itemsets. We first explain the principle of the LCM algorithm for mining closed frequent itemsets. We show how we could adapt this algorithm to gradual itemsets, and we give complexity results on the new algorithm. We then present a parallelization of this algorithm.

A. LCM principle

LCM is the most efficient algorithm for computing closed frequent itemsets, as shown by the results of the FIMI'04 competition [10]. It is the only such algorithm to exhibit a complexity which is linear with the number of closed frequent itemsets to find, hence its name: ***L**inear time **C**losed itemset **M**iner*. This result comes from an important theoretical advance: the authors of LCM could prove that there existed a covering tree over all the closed frequent itemsets, and the edges of this tree could be computed efficiently at runtime. The closed frequent itemsets can thus be discovered with a depth first algorithm, without maintaining a special storage space for the previously obtained patterns: a closed frequent itemset can be outputted as soon as it is discovered. This is not the case for most other closed frequent itemset mining algorithms, which have to keep in memory all the previously found frequent itemsets, to avoid duplications and for some algorithms such as [21] check closure. The closed frequent itemsets can thus only be outputted at the end of computation, and the memory usage can be very important.

LCM is a depth first search algorithm. Each node of the search tree either corresponds to a closed frequent itemset or to an empty leaf. The pseudo-code of LCM is given in Algorithm 1 (coming from [22]).

Algorithm 1 Algorithm LCM

```
1: Input: 𝒯:transaction database, ε:minimum support
2: Output: Enum_ClosedPatterns(⊥) ;

3: Function Enum_ClosedPatterns(P:closed frequent pattern)
4: if P is not frequent then
5:    return ;
6: end if
7: output P ;
8: for i = core_i(P) + 1 to |ℐ| do
9:    Q = Clo(P ∪ i) ;
10:   if P(i − 1) = Q(i − 1) then
11:      Enum_ClosedPatterns(Q) ;
12:   end if
13: end for
```

Each recursive iteration represents a node of the search tree. Its input is a closed frequent pattern. If this pattern is not frequent (line 5), then we are at the end of a branch. Else, the pattern if frequent and the main goal of the iteration is to compute all its direct descendants in the search tree.

This is done with an operation called *prefix preserving extension (ppc-extension)*. Let P be a closed frequent itemset. For an item $i \in P$, we define $P(i) = \{j \mid j \in P \text{ and } j \leq i\}$. Let $core\_i(P)$ be the minimum index i such that $\mathcal{T}(P(i)) = \mathcal{T}(P)$ (with $core\_i(\perp) = 0$). Then an itemset Q is called a ppc-extension of P if

(i) $Q = Clo(P \cup \{i\})$ for some $i \in \mathcal{I}$
(ii) $i \notin P$ and $i > core\_i(P)$
(iii) $P(i-1) = Q(i-1)$, i.e. P and Q share the same $(i-1)$-prefix.

The authors of LCM have shown that for any closed frequent itemset $Q \neq \perp$, there exist only one closed frequent itemset P such that Q is a ppc-extension of P (Theorem 2 in [22]). This is the interest of ppc-extension: it is the very

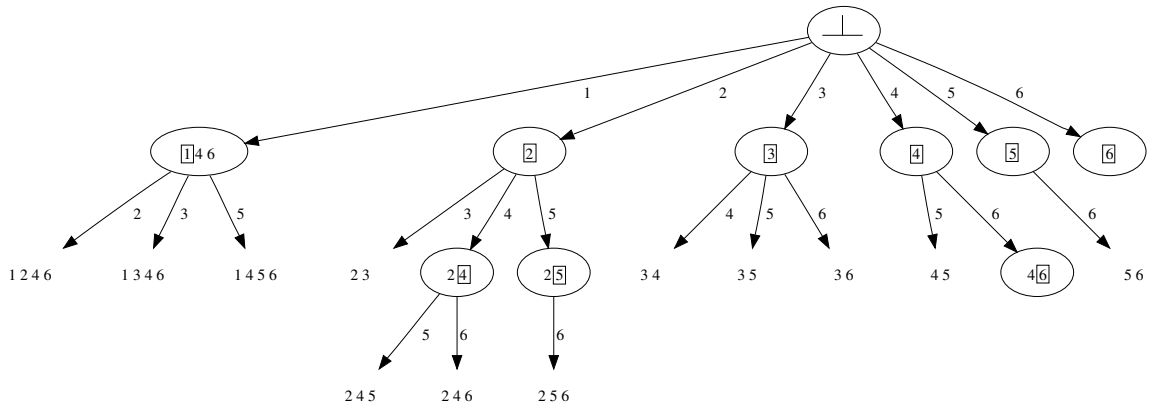

Figure 1. Example of LCM execution, circled itemsets are the closed frequent itemsets, boxed items are $core\_i$ values.

operation that allows the building of a covering tree of closed frequent itemset. Here P will be the unique father of Q, and $\perp$ is the root of the covering tree.

In Algorithm 1, ppc-extension is performed in lines 8-10. For a closed frequent itemset P, all its possible ppc-extensions Q are searched. First, all the i satisfying condition (ii) are iterated over in line 8. Q is computed according to condition (i) in line 9. And condition (iii) is checked in line 10. In line 11, Q is a ppc-extension of P, so a new recursive iteration (i.e. a new node of the search tree) is built with Q.

Example: Let us consider the following transaction database:

Transaction id	Transaction items
t_1	1,2,3,4,5,6
t_2	2,3,5
t_3	2,5
t_4	1,2,4,5,6
t_5	2,4
t_6	1,4,6
t_7	3,4,6

With $\varepsilon = 3$, the depth first search performed by LCM is shown in Figure 1[1].

By definition $core\_i(\perp) = 0$, so any item can be used to extend $\perp$. This is shown by the arcs outgoing from $\perp$, each labelled with the item used for extension. We explain here the branch for item 1. In the case of item 1, $Clo(\{1\}) = \{1, 4, 6\}$, which is our leftmost node for depth 1. It is frequent in the database, so we circle it in the figure as a solution, and the iteration continues. We have to compute $core\_i(\{1, 4, 6\})$. Removing 6 from $\{1, 4, 6\}$ gives the itemset $\{1, 4\}$, and $Clo(\{1, 4\}) = \{1, 4, 6\}$, hence $core\_i(P) < 4$. Removing 4 and 6 from $\{1, 4, 6\}$ gives the itemset $\{1\}$, and we already know that $Clo(\{1\}) = \{1, 4, 6\}$. 1 is the smallest item i such that $Clo(P(i)) = Clo(P)$, thus $core\_i(\{1, 4, 6\}) = 1$. The item 1 is boxed in the figure.

[1]The interested reader will have noticed that it is the same example database as in [22]. However the support value is 3 here instead of 2 in [22], hence the difference in output.

Possible extensions for $\{1, 4, 6\}$ are 2, 3 and 5. The corresponding itemsets are represented below $\{1, 4, 6\}$ in the figure, however they are not frequent so the function immediately returns. The itemsets are not circled in this case, to show that they are not closed frequent itemsets.

B. Adapting LCM principle to gradual itemsets

In order to mine closed frequent gradual itemsets with an algorithm similar to LCM, we need to be able to build a covering tree over all the closed frequent gradual itemsets. As seen before, we thus need to redefine ppc-extension for closed frequent gradual itemsets.

We first give the definition of $core\_i$ for gradual itemsets. The definition is almost identical to the itemset case: let P be a closed gradual itemset , for an item x we have $P(x) = \{(y, v) \mid (y, v) \in P \text{ and } y \leq x\}$. $core\_i(P)$ is the minimum item i such that $(i, v) \in P$ and $g(P(i)) = g(P)$, with $core\_i(\perp) = 0$.

We then give the definition of ppc-extension: the closed gradual itemset Q is a ppc-extension of P if

(i) $Q = Clo(P \cup \{(i, v)\})$ for some (i, v), with $i \in \mathcal{I}$ and $v \in \{\uparrow, \downarrow\}$,
(ii) (i, v) satisfies $(i, v) \notin P$ and $(i, \neg v) \notin P$ and $i > core\_i(P)$ (with $\neg \uparrow = \downarrow$ and $\neg \downarrow = \uparrow$),
(iii) $P(i-1) = Q(i-1)$

Our ppc-extension for gradual itemsets verifies the same theorem than ppc-extension for itemsets:

Theorem 1: Let $Q \neq \perp$ be a closed gradual itemset.Then, there is just one closed gradual itemset P such as Q is a ppc-extension of P.

Skech of proof The proof of this theorem follows exactly the same steps as the proof in [22]. This proof mainly resorts to the properties of the closure operators f and g, which are the same for any closure operators. For gradual itemsets, the properties of monotonicity, extensivity and idempotency have been proven in [17].

A property that is often used in the proof of [22] is, when adapted to gradual itemsets: let P be a gradual itemset and $(i, v) \notin P$ be a gradual item. Then $g(P \cup \{(i, v)\}) = g(P) \cap g(\{(i, v)\})$. This property is not demonstrated in [17], however it comes from Theorem 1 in [8], where the set of sequences output by g is encoded into a binary matrix. ∎

Thanks to the ppc-extension for gradual itemsets, we can write a mining algorithm similar to LCM. We present our GLCM algorithm in Algorithm 2.

For simplicity (and performance) reasons, we have decided to represent gradual items as integers, instead of pairs. A gradual item (i, v) will be encoded by the integer $enc(i, v)$ such as:

$$enc(i, v) = \begin{cases} 2i & \text{if } (i, v) = (i, \uparrow) \\ 2i + 1 & \text{if } (i, v) = (i, \downarrow) \end{cases}$$

The even integers representing positive (ascending) variations and the odd integers representing negative (descending) variations. With this encoding, we force the items of $\mathcal{I}$ to be $[0, n - 1]$, with a renaming if necessary.

With such a coding of gradual items, ppc-extension for gradual itemsets is even closer of ppc-extension for itemsets: point (i) and (iii) are strictly identical, where i stands for the encoding of a gradual item and $core\_i$ is applied on encoded gradual itemsets and returns itself an integer code instead of an item. To avoid confusion with item 0, we fix $core\_i(\perp) = -1$. For point (ii), we have to check the (encoded) value of $core\_i$. If it is an even value of the form $enc(i, v) = 2i$, it means that the gradual itemset P contains $(i, \uparrow)$. It would not make sense to try to extend P with $enc(i, v) + 1 = 2i + 1$, corresponding to the gradual item $(i, \downarrow)$. In this case we directly skip to $enc(i, v) + 2$, i.e. $(i + 1, \uparrow)$. This verification is handled in lines 18-22 of Algorithm 2. Lines 23-27 show the ppc-extension itself, and are very similar to LCM.

The differences between LCM and GLCM lie first in a small optimization specific to gradual itemset in lines 14-15. Any gradual itemset, representing co-variations items, has a symmetric gradual itemset where the items are the same and the variations are all reversed. For example, the symmetric of $\{(1, \uparrow), (2, \downarrow), (3, \uparrow)\}$ is $\{(1, \downarrow), (2, \uparrow), (3, \downarrow)\}$, supported by the same tid sequences but in the reverse order. It is redundant to compute a gradual itemset and its opposite, so we arbitrarily decided to compute only gradual itemsets whose first variation is ascending, they represent themselves and their opposite as well.

The computation of support in line 11 is standard for gradual itemsets: we compute the longest transaction sequence supporting P. More interesting is the computation of closure, shown in Algorithm 3. As a pre-processing, at the initialization of the algorithm we compute for each gradual item the sequence of tuples corresponding to the order for that item (line 3 in Algorithm 2). We then encode this sequence in a binary matrix in the same way as [8]: columns and lines represent tids. If a transaction t_i is before a transaction t_j in the sequence, then there is a 1 in the corresponding binary matrix. This is a matrix encoding for graphs, where the graph is simply the sequence of tids.

Algorithm 2 Algorithm GLCM

1: **Input:** $\mathcal{T}$:transaction database, ε:minimum support
2: **for all** gradual item $(i, v) \in \mathcal{I} \times \{\uparrow, \downarrow\}$ **do**
3: $L_{enc(i,v)} \leftarrow$ tid sorted in v order of item i value
4: $B_{enc(i,v)} \leftarrow$ bitmap matrix associated to $L_{enc(i,v)}$
5: **end for**
6: $\mathcal{B} \leftarrow \{B_1, ..., B_{2\times|\mathcal{I}|}\}$
7: **for all** gradual item encoding $e \in [0, 2 \times |\mathcal{I}| - 1]$ **do**
8: $GlcmLoop(\{e\}, \mathcal{T}, \mathcal{B}, \varepsilon)$
9: **end for**

10: **Function GlcmLoop(P:closed gradual itemset, $\mathcal{T}, \mathcal{B}, \varepsilon$)**
11: **if** computeLongestPath($P, \mathcal{T}, \mathcal{B}$) $< \varepsilon$ **then**
12: **return** ;
13: **end if**
14: **if** $P[0]$ is odd **then**
15: **return** ; *// Symmetrical itemset has already been tested*
16: **end if**
17: output P ;
18: **if** ($core\_i(P)$ is odd) **or** ($core\_i(P) = -1$) **then**
19: $k = core\_i(P)$
20: **else**
21: $k = core\_i(P) + 1$
22: **end if**
23: **for** $e = k + 1$ to $2 \times |\mathcal{I}| - 1$ **do**
24: $Q = Clo(P \cup \{e\}, \mathcal{B})$;
25: **if** $P(e - 1) = Q(e - 1)$ **then**
26: GlcmLoop($Q, \mathcal{T}, \mathcal{B}, \varepsilon$) ;
27: **end if**
28: **end for**

Example: Let us consider a gradual itemset P supported by two lists of tuples tids: $< t_1, t_3, t_2 >$ and $< t_5, t_3, t_4 >$. Then the binary matrix B_P encoding these sequences is shown below:

Transaction id	t_1	t_2	t_3	t_4	t_5
t_1	1	1	1	0	0
t_2	0	1	0	0	0
t_3	0	1	1	1	0
t_4	0	0	0	1	0
t_5	0	0	1	1	1

The binary matrices contain all the necessary information to make the computations of the f and g closure functions, as shown in Algorithm 3.

Di Jorio et al. [8] have shown that given two gradual patterns P and Q and their respective binary matrix representations B_P and B_Q, the binary matrix of $P \cup Q$ was

Algorithm 3 Functions Clo and G

1: **Function Clo**($P, \mathcal{B}$)
2: **return** $F(G(P, \mathcal{B}), \mathcal{B})$;

3: **Function F**($BM, \mathcal{B}$)
4: $P \leftarrow \emptyset$
5: **for all** gradual item encoding $e \in [1, 2 \times |\mathcal{I}| - 1]$ **do**
6: $tmp \leftarrow BM \& B_e$
7: **if** $tmp = BM$ **then**
8: $P \cup = \{e\}$
9: **end if**
10: **end for**
11: **return** P ;

12: **Function G**($P, \mathcal{B}$)
13: $BM \leftarrow B_{P[0]}$
14: **for all** gradual item encoding $e \in P$ **do**
15: $BM \ \& = B_e$
16: **end for**
17: **return** BM ;

$B_{P \cup Q} = B_P$ AND B_Q. This is exactly what we need for closure function g: we give it a gradual pattern P as input, and want all the transaction sequences supporting P. By ANDing all the binary matrices of the items in P (lines 13-16), we obtain the binary matrix of P, B_P. This matrix encodes the graph representing the order between tuples supporting P. To get the actual sequences, we would have to find all the longest pathes in this graph. However as the result of g is directly passed to f, and that f can directly work with the matrix B_P, we can avoid this step.

f itself takes a set of sequences of tid, here encoded in a binary matrix BM. It has to check for each item if the variations of this item are compatible with each of the input sequences. This comes to check if all the 1 in the input matrix BM can be found in the matrix of item i, B_i (remember that if $B[x, y] = 1$, it means that $t_x < t_y$ for the gradual itemset associated to B). We thus AND BM and B_i for each item i, and keep only the items i such that BM AND $B_i = BM$.

The use of binary matrices has the advantage to avoid costly computations with sequence and graph structures, and lead to compact structures thanks to bitmap representation.

Complexity: The GLCM algorithm makes a depth first exploration of a covering tree over all the closed frequent gradual itemsets. Like LCM, it's complexity is thus linear in the number of closed frequent gradual itemsets to find.

For each closed frequent gradual itemset, the complexity of support computation comes to find the longest path in a graph, which as been proven to be linear in the number of nodes of the graph for directed acyclic graphs, i.e. $O(||\mathcal{T}||)$ in our case. The complexity of closed frequent itemset computation is thus, like LCM, dominated by closure computation. Analyzing Algorithm 3 shows that f and g both loop on the items, and make binary matrix computations inside the loops. The time complexity of Clo is thus $O(|\mathcal{I}| \times ||\mathcal{T}||^2)$. Closure operation is embedded in a loop on items in GlcmLoop, so the overall time complexity per closed frequent gradual itemset is $O(|\mathcal{I}|^2 \times ||\mathcal{T}||^2)$. The space complexity mainly depends on the storage of the initial database and of the binary matrices for items, this gives a space complexity of $O(||\mathcal{T}|| + |\mathcal{I}| \times ||\mathcal{T}||^2)$.

GLCM inherits from LCM its good complexity properties. Especially, its space complexity do not depend on the number of closed frequent gradual itemsets to find. This allow in practice to run with a very low and near constant memory usage, whereas other algorithms can use exponentially more memory.

C. Parallelization

The existing work for discovering frequent gradual itemsets, Grite [8], has been parallelized as the Grite-MT algorithm in order to exploit multicore processors, with good results [16].

We also give a simple parallelization for the GLCM algorithm in this paper, which is based on the works for parallelizing LCM [12]. The authors have defined a parallelism environment, Melinda, that simplifies the parallelization of existing sequential algorithms by relieving the algorithm designer of the burden of manual thread synchronization while being efficient for recursive algorithm.

Melinda is based on the Linda approach [23]. It consists of a shared memory space, called *TupleSpace*. All the threads can access the TupleSpace, and either deposit or retrieve a data unit called *Tuple*, via the two primitives $get(Tuple)$ and $put(Tuple)$. All the synchronizations for accessing the TupleSpace are handled by Melinda.

PGLCM is the parallel implementation of GLCM, using Melinda, shown in Algorithm 4. In PGLCM, the tuples correspond to recursive calls. A tuple will thus simply be a closed gradual itemset P that would normally have been passed in parameter to GlcmLoop. The other arguments of GlcmLoop are constants after initialization, so for sake of efficiency they are treated as global variables accessible by all threads and do not need to be passed in the tuples.

Instead of recursive calls (lines 8 and 26 of GLCM), PGLCM only creates a new tuple in the TupleSpace: $put(P)$ (lines 8 and 27 in PGLCM). This means that there is new node of the covering tree to explore.

The threads themselves execute the code of Algorithm 5.

As soon as a thread is idle, it asks for a tuple in order to work on a new node of the search tree (line 2 of Algorithm 5). Melinda enforce locality properties, so a thread of a given processor is most likely to receive tuples that it has put before, and whose corresponding data will already be in the cache of the processor.

Algorithm 4 Algorithm PGLCM

Input: $\mathcal{T}$:transaction database, ε:minimum support, N : number of threads
for all gradual item $(i, v) \in \mathcal{I} \times \{\uparrow, \downarrow\}$ **do**
  $L_{enc(i,v)} \leftarrow$ tid sorted in v order of item i value
  $B_{enc(i,v)} \leftarrow$ bitmap matrix associated to $L_{enc(i,v)}$
end for
$\mathcal{B} \leftarrow \{B_1, ..., B_{2\times|\mathcal{I}|}\}$
for all gradual item encoding $e \in [0, 2 \times |\mathcal{I}| - 1]$ **do**
  $put(\{e\})$
end for
wait for all threads to complete

Function PGlcmLoop(P:closed gradual itemset)
if computeLongestPath$(P, \mathcal{T}, \mathcal{B}) < \varepsilon$ **then**
  return ;
end if
if $P[0]$ is odd **then**
  return ; *// Symmetrical itemset has already been tested*
end if
output P ;
if ($core\_i(P)$ is odd) **or** ($core\_i(P) = -1$) **then**
  $k = core\_i(P)$
else
  $k = core\_i(P) + 1$
end if
for $e = k + 1$ to $2 \times |\mathcal{I}| - 1$ **do**
  $Q = Clo(P \cup \{e\}, \mathcal{B})$;
  if $P(e-1) = Q(e-1)$ **then**
    $put(Q)$;
  end if
end for

Algorithm 5 Function threadFunction()

while $get(tuple)$ **do**
  $P \leftarrow tuple.pattern$;
  $PGlcmLoop(P)$;
end while

When there are no more tuples in the TupleSpace and no thread working, Melinda sends a termination signal, and the program stops.

IV. Experiments

We present in this section an experimental study on the execution time and memory consumption of GLCM and PGLCM. We first present comparative experiments between our new algorithms and the current state of the art, Grite (sequential) [8] and Grite-MT (parallel) [16]. The comparison is "unfair": GLCM/PGLCM compute only the closed frequent gradual itemsets, whereas Grite/Grite-MT compute all the frequent gradual itemsets. However, there exist no algorithm (before GLCM) for mining closed frequent gradual itemsets. The experiments in the paper defining the notion of closure for gradual itemsets [17] rely on a post-processing of the results of Grite, which takes even more time than running Grite alone.

Thus, our experiments reflect the fact that up to now the only way to get gradual itemsets was to use Grite/Grite-MT, and we show the advantage of using our approach instead.

The comparative experiments are based on synthetic datasets produced with the same modified version of IBM Synthetic Data Generator for Association and Sequential Patterns as the one used in [8], [16].

All the experiments are conduced on a 4-socket server with 4 Intel Xeon 7460 with 6 cores each, for a total of 24 cores. The server has 64 GB of RAM. We compare our C++ implementation of GLCM/PGLCM with the original C++ implementation of Grite/Grite-MT.

A. Comparative experiments: sequential

The first experiment compares the run time and memory usage for GLCM and Grite. The dataset used, C1000A20, has 1000 transactions and 20 items. Figure 2 shows the execution time for both programs when varying the support, with a logarithmic scale for time. Figure 3 shows the memory usage.

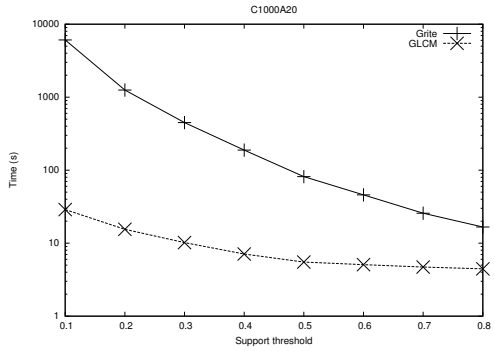

Figure 2. Time vs support, sequential

The execution time results show that GLCM is two orders of magnitude faster that Grite for handling this small dataset: for the lowest value it answers in 29s, while Grite needs 1 hour and 40 minutes. Both programs have a low memory usage on this small dataset. As expected GLCM memory usage is constant whatever the support value, while for lower support values Grite increases its memory usage, because it depends on the number of frequent gradual itemsets to find.

B. Comparative experiments: parallel

The next experiment compares the scaling capacities of PGLCM and Grite-MT on several cores, for the dataset

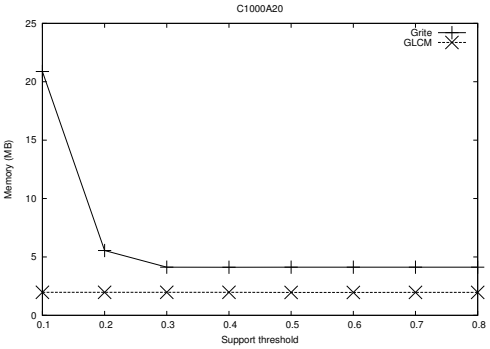

Figure 3. Memory vs support, sequential

C500A50 with 500 transactions and 50 items. This dataset is more difficult than the previous one, as the complexity lies in the number of items which determines the number of (closed) frequent gradual itemsets.

Figure 4 shows the execution time for both algorithms w.r.t. the number of threads, with a logarithmic scale for time. Figure 5 shows the speedup w.r.t. sequential execution for both algorithms. Last, Figure 6 shows the memory consumption.

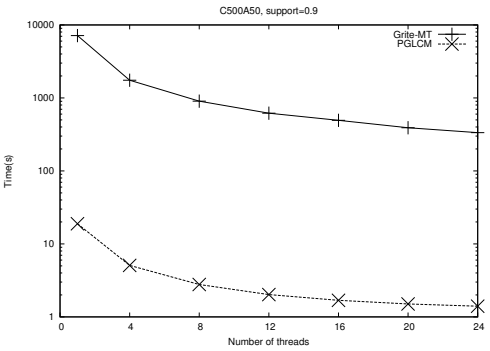

Figure 4. Time vs #threads

In this experiment with a more complex dataset, PGLCM is again two orders of magnitude faster than Grite-MT. With all 24 threads, PGLCM completes execution near instantly in 1.4s, while Grite-MT needs 335s. The memory usage does not change much whatever the number of threads for both programs. Grite-MT exhibits a better speedup than PGLCM on this experiment. However, the run times for PGLCM get very low with more than 8 threads: they are between 1 and 2 seconds. There may not be enough work for PGLCM to exploit all 24 threads.

We thus did the same experiment with C800A100, a more

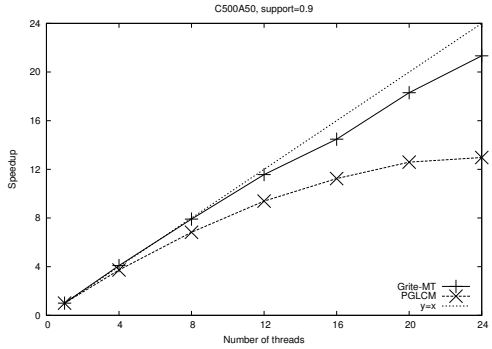

Figure 5. Speedup

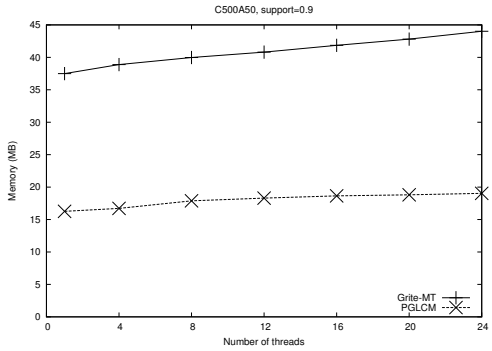

Figure 6. Memory vs #threads

complex dataset with 800 transactions and 100 items. Grite-MT could not run for this dataset: it filled up the 64 GB of RAM of our machine and could not complete. This excessive memory consumption was already mentioned in [16], and comes from the fact that memory complexity in Grite/Grite-MT, like in Apriori, depends on the number of frequent gradual itemsets to find. PGLCM does not have this problem, we thus report its run time in Figure 7, the speedup in Figure 8 and the memory consumption in Figure 9.

For this more complex problem the run time with 24 cores is 48s, so there is enough computation to keep the program busy. The speedups are far better in this case, with an excellent speedup of 22.15 for 24 threads. The granularity of our parallelization is well adapted to complex datasets. For further works, it could be interesting to be able to decompose the computations in lower granularity tasks when faced with simpler datasets such as C500A50.

C. Using PGLCM to mine real-world data

We have also run experiments on a real dataset of DNA micro-arrays describing gene expressions in the framework

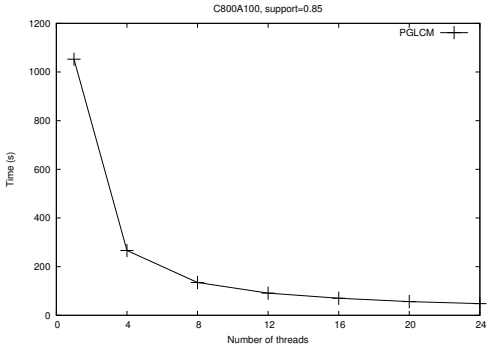

Figure 7. Time vs #threads

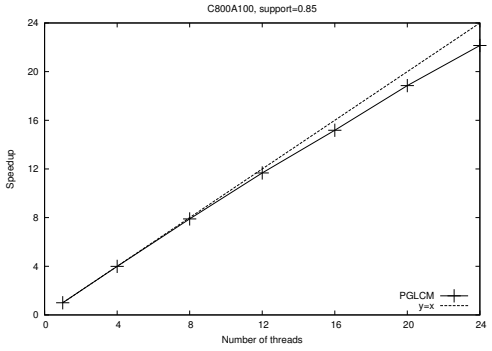

Figure 8. Speedup

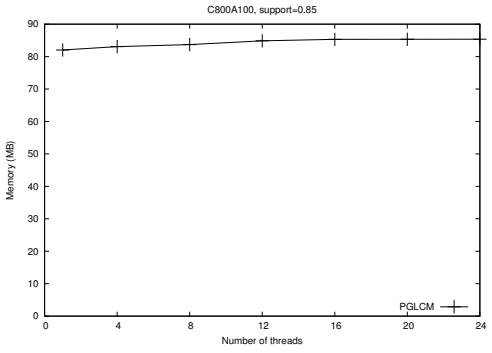

Figure 9. Memory vs #threads

of breast cancer. The data are described over $4,408$ genes which correspond to the items, there are 108 transactions. This dataset is very complex, and up to now no algorithm was able to process it in order to extract gradual patterns. We could run PGLCM with 24 threads on this dataset. The run times for different support value are presented in Figure 10. Memory consumption remained constant at 17 GB.

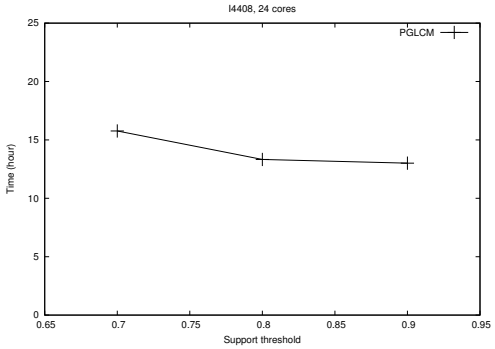

Figure 10. Time vs support, 24 threads

This dataset is much more complex than the synthetic datasets presented before: this time PGLCM needed nearly 16 hours to give results for the lowest support value tested. However, the important point is that for the first time gradual itemsets could be mined from this gene expression dataset. Correlation between expression values of genes is very important to determine gene regulations, our algorithms thus opens the way for new methods for analyzing gene expression data, and many other numerical datasets.

V. Conclusion and perspectives

We have presented in this paper GLCM, the first algorithm for directly mining closed frequent gradual itemsets. Such gradual itemsets allow to find covarations between numerical attributes, with many applications to real data.

Our algorithm is based on the ppc-extension idea developed in the LCM algorithm, and which is currently the most efficient way to mine closed patterns, with a time complexity linear in the number of results to find and a memory complexity constant w.r.t. the number of results to find.

We also parallelized our algorithms in order to exploit the computing power of recent multicore processors.

Our experimental study have shown that our approach, either sequential or parallel, is two orders of magnitudes faster than the state of the art. Our parallel algorithm scales well with the number of available cores for complex datasets, where such computing power is really needed. The low memory requirements of our algorithm allow it to handle large real world datasets, which could not be

handled by existing algorithms due to memory saturation. Our algorithms thus removed the lock that prevented the use of gradual patterns analysis in realistic applications.

Our work opens several perspectives. An immediate perspective is to cooperate with practitioners having large numerical datasets, in order to help them extracting and analyzing gradual datasets. Reporting the results of such experiments will allow to show the practical interest of gradual pattern and hopefully lead to further research in the field of gradual pattern mining.

Other perspectives lie in the improvement of our algorithms. Currently, our algorithm always uses the same binary matrices whatever the gradual itemset under consideration. However [8] has shown that the binary matrices could be reduced: some transactions never appear in the support of a pattern, so the corresponding lines and columns can be suppressed from the corresponding binary matrix. This optimization would not reduce the theoretical complexity, however the FIMI workshop results [10], [22] showed that reducing databases was one of the keys for reducing practical run time.

ACKNOWLEDGMENT

The authors would like to thank Benjamin Négrevergne for providing them the Melinda library.

REFERENCES

[1] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th VLDB Conference*, 1994, pp. 487–499.

[2] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Prefixspan: Mining sequential patterns by prefix-projected growth," in *ICDE*, 2001, pp. 215–224.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," in *In Proc. of the Second SIAM International Conference on Data Mining (SDM2002), Arlington, VA*, Avril 2002, pp. 158–174.

[4] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *PKDD*, 2000, pp. 13–23.

[5] R. Srikant and R. Agrawal, "Mining quantitative association rules in large relational tables," in *SIGMOD Conference*, 1996, pp. 1–12.

[6] Y. Aumann and Y. Lindell, "A statistical theory for quantitative association rules," *J. Intell. Inf. Syst.*, vol. 20, no. 3, pp. 255–283, 2003.

[7] T. Washio, Y. Mitsunaga, and H. Motoda, "Mining quantitative frequent itemsets using adaptive density-based subspace clustering," in *ICDM*, 2005, pp. 793–796.

[8] L. Di Jorio, A. Laurent, and M. Teisseire, "Mining frequent gradual itemsets from large databases," in *Int. Conf. on Intelligent Data Analysis, IDA'09*, 2009.

[9] N. Pasquier, Yves, Y. Bastide, R. Taouil, and L. Lakhal, "Efficient mining of association rules using closed itemset lattices," *Information Systems*, vol. 24, pp. 25–46, 1999.

[10] B. Goethals, "Fimi repository website," http://fimi.cs.helsinki.fi/, 2003-2004.

[11] T. Uno, M. Kiyomi, and H. Arimura, "Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *FIMI*, 2004.

[12] B. Negrevergne, A. Termier, J.-F. Mehaut, and T. Uno, "Discovering closed frequent itemsets on multicore: Parallelizing computations and optimizing memory accesses," in *The 2010 International Conference on High Performance Computing & Simulation (HPCS 2010)*, 2010, pp. 521–528.

[13] H. Arimura and T. Uno, "An output-polynomial time algorithm for mining frequent closed attribute trees," in *15th International Conference on Inductive Logic Programming (ILP'05)*, 2005.

[14] F. Berzal, J.-C. Cubero, D. Sanchez, M.-A. Vila, and J. M. Serrano, "An alternative approach to discover gradual dependencies," *Int. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems (IJUFKS)*, vol. 15, no. 5, pp. 559–570, 2007.

[15] E. Hüllermeier, "Association rules for expressing gradual dependencies," in *Proc. of the 6th European Conf. on Principles of Data Mining and Knowledge Discovery, PKDD'02*. Springer-Verlag, 2002, pp. 200–211.

[16] A. Laurent, B. Négrevergne, N. Sicard, and A. Termier, "Pgp-mc: Towards a multicore parallel approach for mining gradual patterns," in *DASFAA (1)*, 2010, pp. 78–84.

[17] S. Ayouni, A. Laurent, S. B. Yahia, and P. Poncelet, "Mining closed gradual patterns," in *10th International Conference on Artificial Intelligence and Soft Computing, ICAISC 2010*, ser. LNCS, vol. 6113, 2010, pp. 267–274.

[18] T. Calders, B. Goethals, and S. Jaroszewicz, "Mining rank-correlated sets of numerical attributes," in *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge Discovery and Data mining*, 2006, pp. 96–105.

[19] A. Laurent, M.-J. Lesot, and M. Rifqi, "Graank: Exploiting rank correlations for extracting gradual dependencies," in *Proc. of FQAS'09*, 2009.

[20] M. Kendall and B. Babington Smith, "The problem of m rankings," *The annals of mathematical statistics*, vol. 10, no. 3, pp. 275–287, 1939.

[21] C. Lucchese, S. Orlando, and R. Perego, "Parallel mining of frequent closed patterns: Harnessing modern computer architectures," in *ICDM*, 2007, pp. 242–251.

[22] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "An efficient algorithm for enumerating closed patterns in transaction databases," in *Discovery Science*, 2004, pp. 16–31.

[23] D. Gelernter, "Multiple tuple spaces in linda," in *PARLE (2)*, 1989, pp. 20–27.