HAL
open science

Pattern mining rock: more, faster, better

Alexandre Termier

To cite this version:

Alexandre Termier. Pattern mining rock: more, faster, better. Databases [cs.DB]. Université de Grenoble, 2013. tel-01006195

HAL Id: tel-01006195

https://theses.hal.science/tel-01006195v1

Submitted on 14 Jun 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Pattern mining rock: more, faster, better

Habilitation Thesis

(Computer Science)

University of Grenoble

public defense scheduled on July the 8th, 2013

by

Alexandre Termier

Jury composition

President :	Claudia Roncancio	Pr, Grenoble INP, France
Reviewers :	Hiroki Arimura	Pr, Hokkaido University, Japan
	Jean-François Boulicaut	Pr, INSA Lyon, France
	Mohammed Javeed Zaki	Pr, Rensselaer Polytechnic Institute, NY, USA
Examiners :	Marie-Odile Cordier	Pr, IRISA, France
	Jean-François Méhaut	Pr, University of Grenoble, France
	Marie-Christine Rousset	Pr, University of Grenoble, France
	Takashi Washio	Pr, Osaka University, Japan

Laboratoire d'Informatique de Grenoble – CNRS UMR 5217
Université Joseph Fourier,
Maison Jean Kuntzmann - 110 av. de la Chimie - Domaine Universitaire de Saint-Martin-d'Hères - BP 53 - 38041 Grenoble cedex 9 France
Tél. : +33 (0)4 76 51 43 61 - Fax : +33 (0)4 76 51 49 85
http://www.liglab.fr

Mis en page avec la classe thloria.

OUTLINE

Introduction **1**

1 Mine more: *One algorithm to rule them all* **5**

A Preliminaries: LCM algorithm principles . . . 6

1 Definitions . . . 6

2 LCM algorithm . . . 7

B Gradual pattern mining . . . 9

C Periodic pattern mining . . . 11

D ParaMiner: towards (pattern mining) world domination . . . 19

E Perspectives for "Mine more" . . . 24

2 Mine faster: *Multicore processors to the rescue* **27**

A Multicore processors . . . 28

B Melinda: a simple API for parallel data-driven programs . . . 31

C Breaking the memory wall for ParaMiner . . . 35

D ParaMiner vs specialized parallel algorithms . . . 40

E Perspectives for "Mine faster" . . . 41

3 Mine better: *Less is more* **45**

A Mining execution traces from Multi-Processor System-on-Chip . . . 46

1 Mining contention patterns in extremely large execution traces . . . 46

2 The SoCTrace project . . . 48

B Use domain knowledge to enable scalable exploration of large user datasets . . . 53

C Reducing the number of patterns . . . 54

Conclusion and Future Work **57**

Bibliography **59**

INTRODUCTION

This beginning of the XXIst century marks the long standing revenge of **data**. For centuries, paper data has been shelved in libraries. For decades, numerical data has been sleeping on hard drives of individuals and corporations. Nowadays, large Internet companies such as Google, Facebook or Yahoo! have realized that large quantities of data could turn into a gold mine for their business. Driven by these companies and the tools they proposed, more and more organizations are turning to the analysis of large quantity of data, for financial, societal or scientific profit. New "buzz words" have emerged, such as *Big Data* and *Medium Data*. While such words can be seen as a fresh coating of flashy paint on well established concepts, their very existence shows that data analysis is going mainstream. Its ideas, concepts and techniques are reaching more company managers, more scientists and many, many more students.

It is thus an exciting time to be a researcher in **data mining**, the young branch of Computer Science dedicated to analyzing data in order to turn it into *actionable knowledge*.

This habilitation thesis presents my main research results and perspectives in the field of data mining called **pattern mining**. The inception of many of these works lies in my two post-doctoral periods in Japan, in Osaka University from 2004 to 2006 and in the Institute of Statistical Mathematics in Tokyo from 2006 to 2007. These postdocs helped me to mature the main research topics that I wanted to develop in my career. I started actively working on these topics with my PhD students and colleagues since I became Associate Professor at Grenoble University in 2007. All the results presented in this habilitation thesis are post-2007.

Intuitively, the goal of pattern mining is to discover interesting *regularities* in data, such regularities being called **patterns**. The field is often called **frequent pattern mining**, for the measure of interestingness is in most existing works the **frequency** of a pattern, i.e. how many times it is repeated in the data. In this habilitation thesis, I will present both works where pattern interestigness is based on frequency, and works where other interestingness measures can be exploited.

The most iconic application of pattern mining is the analysis of supermarket data [AIS93], in order to answer questions such as *which are the products that are often sold together ?* Such information allows to improve product placement, decide better promotions, and product cross-selling. Pattern mining has also been applied successfully to various domains such as bioinformatics [ZS06], telecommunications networks [GAA10], software execution analysis [LYY+05] among many others. The domain is alive and vibrant, with novel kind of useful patterns presented every year at major data mining conferences, algorithms several orders of magnitude faster than those existing ten years before, and challenging new perspectives for the years to come.

Everything seems bright for pattern mining. In our new *data age*, where data analysis tools are king, and with the usefulness of the insights it can provide in data, it should end up as one of the most beloved tools of data analysts.

Sadly, the pretty story stops here, and the real motivations behind the works presented in this habilitation thesis will soon appear. In practice, most existing data analysis tools only provide the most basic pattern mining tools for discovering *frequent itemsets* and *association rules*. Very few tools offer the possibility to mine more complex patterns, such as sequential, tree or graph patterns. Worse, the implementations of frequent itemset mining in these tools often rely on the antiquated Apriori [AIS93] algorithm, which pioneered the domain but is three to four orders of magnitude slower than modern approaches such as LCM [UKA04]. This can be illustrated by looking at the following well known tools:

- Weka [atUoW13], an open source data mining tool used a lot in data mining classes, allows to mine frequent itemsets and association rules with Apriori and FPGrowth (optional package). Another package allows to mine generalized sequential patterns.
- Knime [Kni13], a more advanced open source data mining tool, is limited as-is to frequent itemsets. It can however use any tool of Weka, and C. Borgelt wrote a plugin allowing to use Apriori, FPGrowth, Relim, Sam algorithms. These algorithms allow to mine closed and maximal frequent itemsets. However, they have been proven less efficient than the state of the art.
- Microsoft LINQ includes an association mining algorithms, described as "a straightforward implementation of the well known Apriori algorithm" [Mic12].
- Mahout [Apa12], the Apache Machine Learning toolbox for clusters on MapReduce, contains 11 clustering algorithms and 10 classification algorithms. But it contains only one frequent itemset mining algorithm [LWZ+08], based on FPGrowth [HPY00].

As a pattern mining researcher, I find this situation deeply frustrating: all data analysis tools are be joining the great data bandwagon, and pattern mining is late for the party !

I thus focused my works on what I consider as the three main areas where pattern mining has to be improved in order to become a **mainstream tool for data analysts**. Not in the limited setting of frequent itemsets and association rules, but with all the richness of pattern mining, which is able to tackle very diverse types of data and discover a range of patterns as broad as the data analyst can imagine. I present my contribution in these areas in three chapters:

1. A first critical area is to **extend the reach** of pattern mining by discovering more kinds of patterns likely to interest practitioners for analyzing their data. Such novel patterns can be patterns adapted to complex data types not addressed by existing pattern mining approaches, or complex patterns uncovering non trivial relations in well known types of data. My contributions to this area, presented in Chapter 1, entitled **Mine more**, are twofold:
    - first, I contributed to improve the state of the art on two novel kinds of patterns: **gradual patterns** (work of M2 student Thac Do) and **periodic patterns with unrestricted gaps** (work of PhD student Patricia Lopez Cueva)
    - second, in the PhD of Benjamin Negrevergne we proposed **ParaMiner**, a **generic pattern mining** algorithm capable of efficiently tackling a broad class of pattern mining problems. With ParaMiner, a practitioner can directly specify the kind of pattern he is interested in and benefit from an efficient algorithm, allowing tailor-made pattern definitions for more focused data analysis.

2. The second area is to **reduce the processing time** of pattern mining algorithms. This is especially critical when one want to mine complex patterns as those presented above. One of the solutions to reduce processing time is to exploit efficiently the parallel processing capacities of today's **multicore processors**. My results in this area are presented in Chapter 2, entitled **Mine faster**. Most of these works were conduced in the context of the PhD of Benjamin Negrevergne. I first explain why designing parallel pattern mining algorithms is especially delicate. I then present the approach that we used in ParaMiner to ensure it a good parallel scalability. These results show that the critical **dataset reduction** step of modern pattern mining algorithms can, when conduced naively, completely ruin parallel scalability. Our approach shows how to detect such situations, in which kind of pattern mining problems it is most likely to happen, and proposes solutions to get back parallel scalability without sacrificing execution time.

3. The third and last area is to **provide more understandable results** to the data analysts. Most of my works presented in the previous chapters, which are now mature enough, show how to improve and expand the "main engine" of pattern mining. However as is, the output of this "engine" is difficult to exploit: in most cases, it is a list having millions of patterns. Examining such list is time consuming and error prone. Thus in Chapter 3, entitled **Mine better**, I will present several different approaches that we are working on with PhD students and colleagues, in order to improve results or results presentation. These approaches are based on the idea that as output of a data mining process, the patterns should explain as well as possible the data, without overwhelming the user. I work on this topic in real application contexts, especially the analysis of large application execution traces in embedded systems. The works presented in this chapter are less mature than the two other chapters, and the whole chapter can be considered as a chapter or short and medium term perspectives.

Contents

A Preliminaries: LCM algorithm principles
1 Definitions
2 LCM algorithm
B Gradual pattern mining
C Periodic pattern mining
D ParaMiner: towards (pattern mining) world domination
E Perspectives for "Mine more"

CHAPTER

1

MINE MORE: *One algorithm to rule them all*

Since its inception in 1993 [AIS93], researchers in pattern mining have strived to extend the reach of this domain to either extract more complex patterns from the data (such as [SA95], [BKWH11]) or to extract patterns from more and more kinds of data (e.g. sequences [TC12], trees [AAA+02] or graphs [IWM00]). This work is the most fundamental of all in pattern mining: it allows to exploit pattern mining techniques on an increasing number of domains, and pushes researchers and data owners to ask increasingly complex questions to data thanks to advanced pattern mining techniques. It *opens up the imagination of data analysts*, by providing, as imperfect as they can be, new tools to analyze their data. Making these tools faster and easier to use only comes afterwards.

From a pattern mining researcher point of view, such task requires first to come up with a new type of pattern to discover. This is often the result of a cooperation with practitioners, that have a real problem and actual data to analyze. Once the pattern mining problem is posed, the pattern mining researcher proposes an algorithm efficient enough to process the real data it was designed for.

In this regard, my major contribution to the field of pattern mining is to have proposed several algorithms for solving novel pattern mining problems, which are all based on the best known techniques for **closed** pattern enumeration and testing. The best synthesis of these techniques can be found in the LCM algorithm [UKA04] from Uno et al., which won the FIMI workshop of frequent itemset miners in 2004, and is to date the best sequential algorithm for mining frequent itemsets. Despite its efficiency, this algorithm is non-trivial to understand, and there are only few work that build upon LCM, compared with the myriad of (even recent) work based on the antiquated Apriori algorithm [ALS12, DJLT09], and the many other based on the far from optimal FP-Growth algorithm [TWSY10, LL07]. My works can all be seen as examples of how to build efficient pattern mining algorithms based on the techniques presented in LCM, with an opinionated goal of shifting pattern mining research on the higher grounds that these techniques allow. Note that the enumeration used in LCM is close from former FCA results

[Gan84, Kuz96]: working in such kind of enumeration will help in bridging the gap between pattern mining and FCA community. The advantages being:

- for pattern mining researchers, to access the wealth of strong enumeration work of FCA researchers, with potential for better algorithms ;
- for FCA researchers, to access the culture of pattern mining researchers on high performance algorithms applied to large real data, and gain an improved visibility.

In this chapter,

- I first present a synthetic view of the LCM algorithm and especially the enumeration technique used in this algorithm. This introductory section is designed to help understanding how LCM can be extended, and is the basis for the rest of the chapter.
- I present my work on **gradual patterns**. It was conduced during two M2R internship periods of Trong Dinh Thac Do, in collaboration with Anne Laurent (University of Montpellier). This work was presented at ICDM'10 conference [DLT10] and has been submitted in second revision in the KAIS journal [DTLN13].
- Next, I present my work on **periodic patterns**, done with the PhD of Patricia Lopez Cueva, in collaboration with Jean-François Méhaut (University of Grenoble) and Miguel Santana (STMicroelectronics). This work was presented in the EMSoft'12 conference [CBT+12].
- I then present ParaMiner, a first work on **efficient generic closed pattern mining**, allowing to handle a broad range of pattern mining problems with a single algorithm. This is the PhD work of Benjamin Négrevergne, in collaboration with Jean-François Méhaut and Marie-Christine Rousset (both at University of Grenoble). This work is published in the DMKD journal [NTRM13].
- I finish the chapter by giving some future research directions, focus on extensions of the ParaMiner algorithm.

A Preliminaries: LCM algorithm principles

The rest of this chapter is heavily based on the principles proposed by Uno and Arimura in the LCM algorithm [UAUA04]. We thus briefly give some definitions used throughout this chapter, and explain the most important principles of the LCM algorithm.

1 Definitions

The data is constituted of **items** drawn from a set $\mathcal{I} = \{i_1, ..., i_m\}$, also called **ground set** in the following. A **dataset** is a set of **transactions** $\mathcal{D} = \{t_1, ..., t_n\}$ where each transaction is a set of items: $\forall i \in [1..n] \;\; t_i \subseteq \mathcal{I}$. Each transaction $t_k \in \mathcal{D}$ has a unique identifier $tid(t_k)$. We assume for simplicity $tid(t_k) = k$.

An **itemset** I is an arbitrary set of items: $I \subseteq \mathcal{I}$. An itemset I **occurs** in a transaction t_k, with $k \in [1..n]$, if $I \subseteq t_k$. Given a dataset $\mathcal{D}$, the **conditional dataset** for an itemset I is the dataset restricted to the transactions of $\mathcal{D}$ where I occurs: $\mathcal{D}[I] = \{t \mid t \in \mathcal{D} \text{ and } I \subseteq t\}$.

The **tidlist** of an itemset I in a dataset $\mathcal{D}$ is the set of identifiers of the transactions where the itemset occurs: $tidlist_{\mathcal{D}}(I) = \{tid(t_k) \mid t_k \in \mathcal{D} \text{ and } I \subseteq t_k\}$. The **support** of an itemset I in a dataset $\mathcal{D}$ is the number of transactions of $\mathcal{D}$ in which I occurs: $support_{\mathcal{D}}(I) = |tidlist_{\mathcal{D}}(I)|$. Given a frequency

threshold $\varepsilon \in [0, n]$, an itemset I is said to be **frequent** in dataset $\mathcal{D}$ if $support_{\mathcal{D}}(I) \geq \varepsilon$. When the dataset is clear from context, $\mathcal{D}$ will be omitted from the notations of $tidlist$ and $support$ in the rest of the chapter.

If an itemset I is frequent, all its sub-itemsets $I' \subset I$ are also frequent. The sub-itemsets that have exactly the same support as I do not bring any new information and it is thus not necessary to compute them nor to output them, which can lead in practice to one order of magnitude faster computation and reduced output size [PBTL99]. The remaining itemsets are called **closed frequent itemsets**. More formally, a frequent itemset I is said to be closed if there exist no itemset I' with $I' \supset I$ such that $support(I) = support(I')$.

Problem statement: (closed frequent itemset mining) The goal of LCM is, given a dataset $\mathcal{D}$ and a frequency threshold ε, to return all the closed frequent itemsets having at least frequency ε.

2 LCM algorithm

A simplified pseudo-code of the LCM algorithm is presented in Algorithm 1.

Algorithm 1: LCM

Data: dataset D, minimum support threshold ε
Result: Output all closed itemsets in D

```
begin
    ⊥_clo ← Clo(⊥) = ⋂_{t∈D} t
    output ⊥_clo
    D_{⊥_clo} ← reduce(D, ⊥_clo)
    foreach e ∈ I s.t. e ∉ ⊥_clo do
        expand(⊥_clo, e, D_{⊥_clo}, ε)
Function expand(P, e, D, ε)

    Data: Closed frequent itemset P, item e, reduced dataset D_P, minimum support threshold ε
    Result: Output all closed itemsets descending from (P, e) in the enumeration tree
    begin
        if support_{D_P}(P ∪ {e}) ≥ ε then                    /* Frequency test */
            Q ← Clo(P ∪ {e}) = ⋂_{t∈D_P[{e}]} t              /* Closure computation */
            if firstParentTest(Q, (P, e)) then                /* First parent test */
                output Q
                D_Q = reduce(D_P, Q)
                foreach i ∈ augmentations(Q, (P, e), D_Q) do  /* Itemset enumeration */
                    expand(Q, i, D_Q, ε);
```

The main program has to compute the closure of $\perp$, for the case where an itemset would be present in all transactions of the dataset. Once this closure is computed, $expand$ is called with all possible items of $\mathcal{I}$ as augmentations of $\perp_{clo}$ (excepts the ones in $\perp_{clo}$).

On getting intimate with LCM

Since 2007, the LCM algorithm is at the heart of many of my works. In order to build my understanding of this algorithm, it is worth mentioning that in addition to theoretical work and work with students, I also implemented myself twice the LCM algorithm. The most interesting of these implementations is an implementation in Haskell, a popular functional language, with a strong focus on parallelism on multicores. To the best of my knowledge, it is the only implementation of LCM in functional programming, and it is available publicly on Hackage, the public Haskell package repository, as the `hlcm` package [Hac10]. This implementation has two interests:

- It is, to date, the most concise (code-line wise) implementation of LCM having all the optimizations of the original algorithm
- It was one of our earliest parallel implementation of LCM, which allowed us to explore both how to parallelize tasks in LCM, and how adequate a functional programming language was for pattern mining.

The conclusion of this experiment was mixed [TNMS11]: although I could get decent results (one order of magnitude slower than original C implementation) and the algorithm was very easy to write in Haskell, optimizing it for performance was on the other hand a difficult process, and required low level knowledge of the garbage collector of Haskell. The constant evolution of the language runtime, especially for parallel applications, also means that these results are not definitive, and should be reconsidered in a few years time once the language has further matured (for example, at the time of the tests, garbage collector stopped all threads when collecting only for one thread, an important source of inefficiency).
It was nevertheless an excellent exercise to get a good working knowledge of LCM, and Haskell remained my language of choice for quickly prototyping new algorithms.

LCM is a *backtracking* algorithm, so most of the work is performed by the recursive function $expand$. This function takes as input an already found closed frequent itemset $P \subseteq \mathcal{I}$, an item to augment it $e \in \mathcal{I}$, and D_P the *reduced dataset* of P, which is a dataset smaller than the conditional dataset of P, but that is guaranteed to be equivalent w.r.t. the soundness and completeness of computations performed in $expand$. The support of $P \cup \{e\}$ is first computed in line 10. Then the closure of $P \cup \{e\}$ is computed in line 11. This is done by taking the intersection of all the transactions of the conditional dataset of $\{e\}$ in D_P: these are all the transactions of the input dataset D where $P \cup \{e\}$ occurs. At this point, a problem arises in the enumeration: several calls to $expand$ with different parameters (P, e) could lead by closure to the same Q. This means that several independant branches of enumerations would land on the same pattern, needing a costly duplicate detection mechanism (usually of exponential memory complexity). The way LCM avoids this problem is to impose an arbitrary enumeration order that allows to perform a tree-shaped enumeration of the search space of closed frequent patterns.

This means that for a closed pattern Q, there is only one pair (P, e) which is allowed to generate it. Such (P, e) is called the **first parent** of Q. Hence in line 12, the algorithm tests if for the found Q, the couple (P, e) received in parameters is its first parent. Uno and Arimura [UAUA04] have shown that the first parent test in the case of frequent itemsets could be done in polynomial time. If the test passes Q is output in line 13. The reduced dataset of Q is computed in line 14 in order to feed next recursive calls to $expand$. Then, all possible further augmentations of Q are enumerated in line 15 and given as parameters to recursive calls together with Q.

One can note that we have not instanciated the critical functions of LCM, which are $support$, $firstParentTest$, $augmentations$ and $reduce$. Several variants of each of these functions exist, and explaining them in detail is not relevant here. The important point to note is that the structure of the LCM algorithm is quite simple, and that it leaves a lot of room for modification and improvement by replacing the above mentioned functions.

The original LCM algorithm provides strong complexity guarantees: it is of **polynomial space** complexity and of **output linear** time complexity, hence its name: Linear Closed itemset Miner.

B Gradual pattern mining

Most pattern mining methods consider only symbolic data, and cannot be applied directly on numerical data. Gradual pattern mining is one of the rare pattern mining method considering directly numerical data, without any discretization step. It's goal is to discover **co-variations of attributes** in the data. For example, when the data consists of records about individuals, a typical gradual pattern could be: *the older a person is, the higher its salary is*.

More precisely, the input data considered is a matrix where the lines are *transactions* and the columns are *attributes*. A *gradual pattern* is a set of *gradual items*, each gradual item being a direction of variation of an attribute: either increasing (↑) or decreasing (↓). The *support* of a gradual pattern is the size of the longest sequence of transactions that respects the variations indicated by its gradual items.

Exemple: Consider the following table:

tid	age	salary	loans	cars
t_1	22	2500	0	3
t_2	35	3000	2	1
t_3	33	4700	1	1
t_4	47	3900	1	2
t_5	53	3800	3	2

A gradual pattern is $\{age^{\uparrow}, salary^{\uparrow}\}$. The DAG of transactions ordered according to this pattern is shown in Figure 1.1. The longest sequences of transactions exhibiting simultaneous increase of age and $salary$ are: $\langle t_1, t_2, t_4\rangle$ and $\langle t_1, t_2, t_5\rangle$. Both have length 3, so the support of $\{age^{\uparrow}, salary^{\uparrow}\}$ is 3.

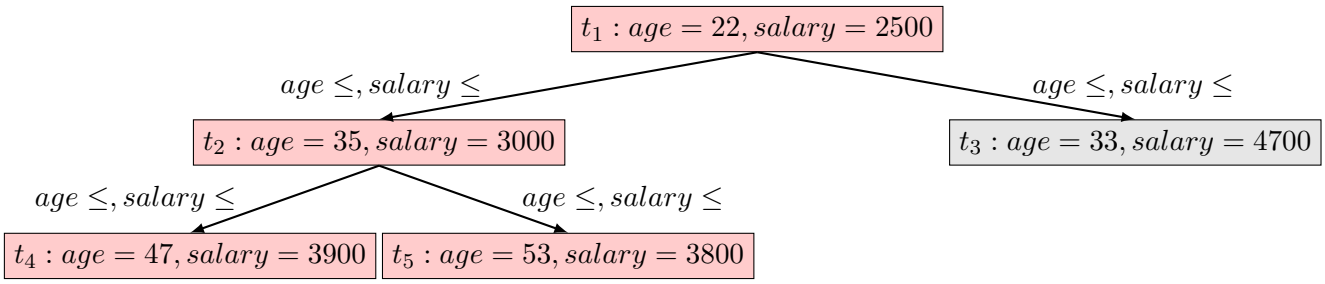


FIGURE 1.1: Transactions ordered according to gradual pattern $\{age^{\uparrow}, salary^{\uparrow}\}$. Nodes of longest paths are colored in red.

The gradual patterns are particularly useful when the records have no particular order, such as different independant observations with changing parameters in a biological experiment. The gradual patterns can discover many complex co-variations that would have been difficult to detect with classical techniques.

However, computing gradual patterns is computationally expensive, so it is important to restrict the algorithm to finding *closed* gradual patterns, in order to avoid any redundancy in the search space. The theoretical definition of closed gradual patterns had been proposed in [ALYP10], but without a mining algorithm.

This is the challenge that we tackled in the M2R internship of Thac Do, co-supervised with Anne Laurent (LIRMM, University of Montpellier), with two objectives:

- propose an efficient algorithm to mine gradual patterns, able to scale up to real datasets
- base this algorithm on the principles of LCM presented above.

In this work, detailed in [DLT10, DTLN13], we could propose an encoding of gradual itemsets as simple itemsets. Using this encoding and modified versions of the $support$, $firstParentTest$ and $augmentations$ functions used in LCM, we could propose GLCM, a variant of LCM for mining closed frequent gradual itemsets. Due to this similarity, we do not present the pseudo code here, the interested reader is referred to the aforementioned publications.

Despite the absence of dataset reduction in GLCM, it has the same complexity guarantees as LCM, and our experiments have confirmed its efficiency. We also made a parallel version of this algorithm, PGLCM. It will be briefly evoked in Chapter 2.

We show on Figure 1.2 a comparative experiment of GLCM with the state of the art that existed before. This state of the art is Grite [DJLT09], an algorithm for mining frequent gradual itemsets based on Apriori. As Grite does not mine closed patterns, the experiment is not fair. However at that time it was the only algorithm available to practitioners willing to compute gradual itemsets. The experiment is conduced on an Intel Xeon 7460 @ 2.66 GHz with 64 GB of RAM. The dataset is a synthetic dataset produced by a modified version of the IBM synthetic data generator.

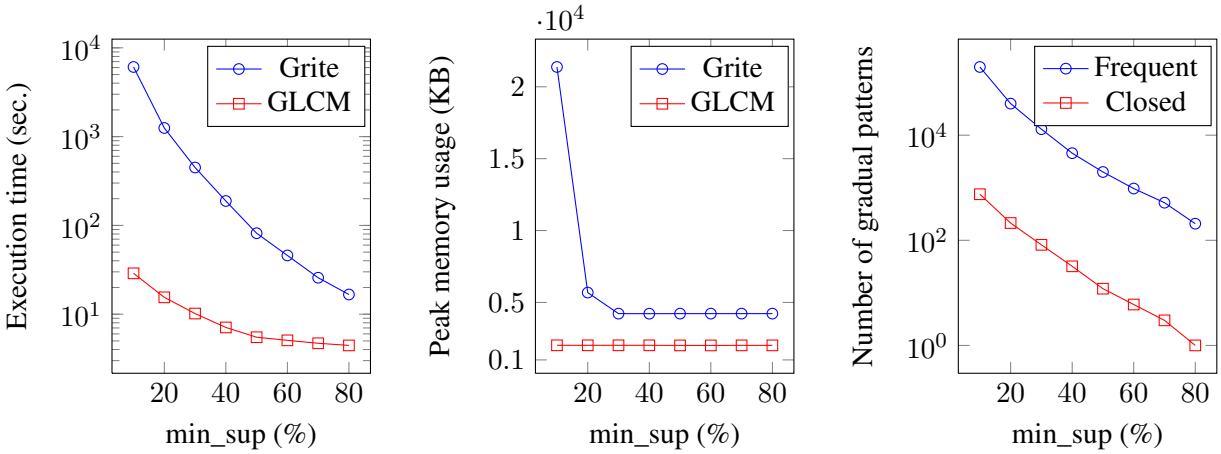


FIGURE 1.2: Minimum support threshold comparative plot

As the results show, GLCM is two to three orders of magnitudes faster than Grite, confirming the interest of an approach mining closed patterns and based on LCM.

Thanks to these excellent results, GLCM was the first algorithm able to mine a real biological dataset of microarray data, with 20 lines and 4408 attributes. The analysis of these results is still ongoing at University of Montpellier.

To conclude this section, I present the analysis of a simple dataset from the American stock market that we did with Thac Do. This dataset collects weekly NYSE quotations for 300 different stocks. We

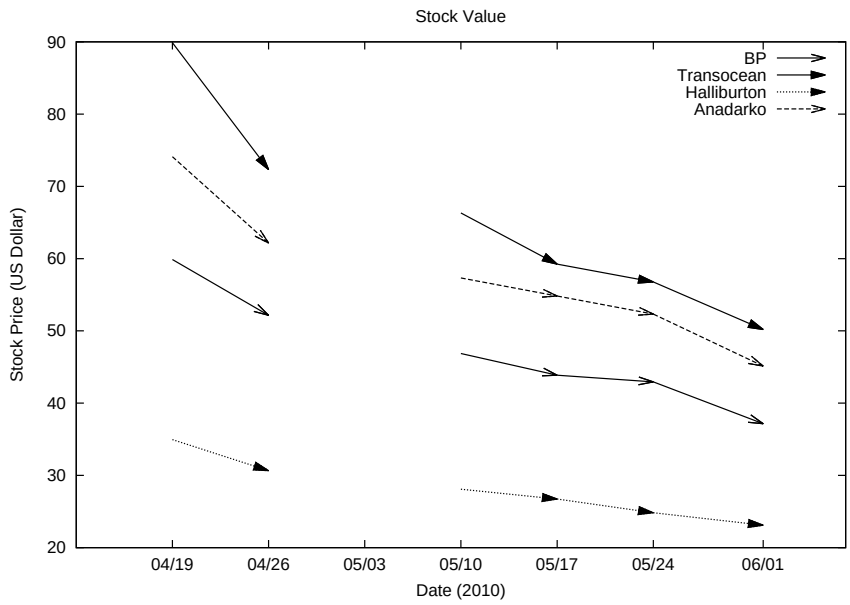


FIGURE 1.3: BP, Halliburton, Transocean and Anadarko's stock price decrease in the same time

present in Figure 1.3 a closed gradual pattern found for the year 2010 at the dates corresponding to the Deepwater Horizon oil spill accident in the Gulf of Mexico.

Unsurprisingly, at this time BP stock price decreases. More interesting is the fact that three other companies see their stock decrease with BP: Transocean, Halliburton, Anadarko. As an European, I didn't know these companies, nor that they had links with BP. However, some googling reveals that:

- Halliburton was responsible for cementing the ocean floor beneath the platform
- Transocean owned the rig where the accident happened, BP was only leasing it
- Anadarco has a 25% share in BP's Macondo Prospect

The gradual pattern quickly showed that there existed a relation between these four compagnies, allowing non specialists to conduct informed searches and possibly take better decision.

GLCM/PGLCM are available as OpenSource: http://membres-liglab.imag.fr/termier/PGLCM.html

Basing the work of Thac Do on LCM was in line with a longer term objective for me: see how far it was possible to go with the LCM principle, and how difficult it was to adapt this principle to mining problems more complex that itemsets. As such, this work is one of the early steps that led to ParaMiner, described later in this chapter.

C Periodic pattern mining

Even in purely symbolic data, there are many numeric relations to be found. One of those is the **periodicity** at which a set of items repeat itself in the data. Consider a transactional dataset, which is a

sequence of transactions $\{t_1, ..., t_n\}$. Each transaction is a set of items, and has a transaction identifier t_i. For example, each transaction could be all the events occuring during a certain time window. If a given itemset I occurs in the transactions $\{t_1, t_3, t_5, t_7\}$, then it occurs every 2 transactions. Such itemset is **periodic**, with a period of 2. Many earlier works on periodic pattern mining [HYD99],[YWY03] proposed definitions based on that intuition. Such definitions are very strict, as they don't allow any disruption in the periodicity. Later works thus proposed more relaxed definitions, allowing "gaps" in the periodicity [MH01]. Although this is better adapted to real world data where periodic behavior can be altered for some transactions and come back after, this lead to an even greater number of output patterns, being computationnally expensive to mine and difficult to exploit by an analyst.

In the CIFRE (joint funding between academy and industry) PhD of Patricia Lopez Cueva, co-supervised with Jean-François Méhaut (University of Grenoble) and Miguel Santana (STMicroelectronics), we tackled this problem in the context of execution trace analysis of multimedia applications [CBT+12]. In this context, the transactions can either correspond to the set of trace events occurring in a fixed time window, or to the set of trace events occurring during the decoding of an audio/video frame.

To explain our definition of periodic patterns, we first present the definition of the basic component of a periodic pattern, that we call a *cycle*:

Definition 1. *(Cycle)* Given a dataset $\mathcal{D}$, an itemset X and a period p, a cycle of X, denoted $cycle(X, p, o, l)$, is a maximal set of l transactions in $\mathcal{D}$ containing X, starting at transaction t_o and separated by equal distance p:

$$cycle(X, p, o, l) = \{t_k \in \mathcal{D} \mid 0 \leq o < |\mathcal{D}|, X \subseteq t_k, k = o + p * i, 0 \leq i < l, X \nsubseteq t_{o-p}, X \nsubseteq t_{o+p*l}\}$$

TABLE 1.1: Example dataset

t_k	Itemset	t_k	Itemset	t_k	Itemset
t_1	a, b	t_5	a, b	t_9	k, l
t_2	c, d	t_6	g	t_{10}	a, b
t_3	a, b	t_7	h, i		
t_4	e, f	t_8	a, b, j		

Example: In the example dataset in Table 1.1, the itemset $X = \{a, b\}$ is found at a period $p = 2$ on transactions from t_1 ($o = 1$) to t_5, therefore it forms a cycle of length $l = 3$, denoted $cycle(\{a, b\}, \mathbf{2}, 1, 3) = \{t_1, t_3, t_5\}$ *(Note: periods are presented in bold for clarity)*. The other possible cycles, shown in red in the figure, are not actual cycles as they do not satisfy the maximality constraint.

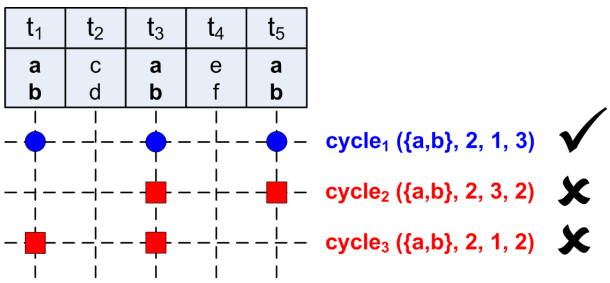


FIGURE 1.4: Example of cycles and non-cycles

Definition 2. *(Periodic pattern)* An itemset X together with a set of cycles C and a period p form a **periodic pattern**, denoted $P(X, p, s, C)$, if the set of cycles $C = \{(o_1, l_1), ..., (o_k, l_k)\}$, with $1 \leq k \leq m$ and m being the maximum number of cycles of period p in the dataset $\mathcal{D}$, is a set of cycles of X such that:

- All cycles have the same period p
- All cycles are consecutive:
  $\forall (o_i, l_i), (o_j, l_j) \in C$ such that $1 \leq i < j \leq k$, we have $o_i < o_j$
- Cycles do not overlap:
  $\forall (o_i, l_i), (o_j, l_j) \in C$ such that $i < j$, we have $o_i + (p * (l_i - 1)) < o_j$

With s being the **support** of the periodic pattern, i.e. the sum of all cycle lengths in $C = \{(o_1, l_1), ..., (o_k, l_k)\}$, calculated with the formula

$$s = \sum_{i=1}^{k} l_i$$

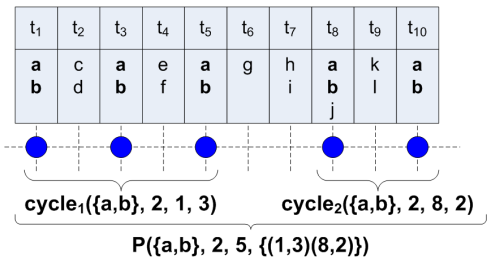


FIGURE 1.5: Periodic pattern formation

Definition 3. *(Frequent Periodic Pattern)* Given a minimum support threshold min_sup, a periodic pattern P is **frequent** if its support is greater than min_sup, i.e. $P(X, p, s, C)$ is frequent if and only if $s \geq min_sup$.

Example: Given the dataset shown in Table 1.1 and a minimum support of two transactions ($min_sup = 2$), the set of frequent periodic patterns is presented in Table 1.2 *(note: only periods not greater that the length of the dataset divided by min_sup are considered).*

TABLE 1.2: Set of frequent periodic patterns

Frequent Periodic Patterns	
$P_1(\{a\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$	$P_9(\{a,b\}, \mathbf{4}, 2, \{(1,2)\})$
$P_2(\{b\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$	$P_{10}(\{a\}, \mathbf{5}, 2, \{(3,2)\})$
$P_3(\{a,b\}, \mathbf{2}, 5, \{(1,3)(8,2)\})$	$P_{11}(\{b\}, \mathbf{5}, 2, \{(3,2)\})$
$P_4(\{a\}, \mathbf{3}, 2, \{(5,2)\})$	$P_{12}(\{a,b\}, \mathbf{5}, 2, \{(3,2)\})$
$P_5(\{b\}, \mathbf{3}, 2, \{(5,2)\})$	$P_{13}(\{a\}, \mathbf{5}, 2, \{(5,2)\})$
$P_6(\{a,b\}, \mathbf{3}, 2, \{(5,2)\})$	$P_{14}(\{b\}, \mathbf{5}, 2, \{(5,2)\})$
$P_7(\{a\}, \mathbf{4}, 2, \{(1,2)\})$	$P_{15}(\{a,b\}, \mathbf{5}, 2, \{(5,2)\})$
$P_8(\{b\}, \mathbf{4}, 2, \{(1,2)\})$	

This very relaxed definition of periodic patterns allows to discover many interesting periodic behaviors in the data, as well as their disruptions. Its disadvantage is that it outputs a huge number of patterns,

many of which are just redundant subparts of other patterns. Our objective was thus to considerably reduce the number of periodic patterns output to the analyst, without losing information, i.e. we were looking for a **condensed representation** of periodic patterns. From [CRB04], a set of patterns $\mathcal{C}$ is a condensed representation of a set of patterns $\mathcal{F}$ if from the elements of $\mathcal{C}$ it is possible to recompute all the elements of $\mathcal{F}$ without re-scanning the original data. In the case of frequent itemsets (and of frequent gradual patterns seen before), the set of **closed** frequent itemsets / gradual patterns is a condensed representation of the set of frequent itemsets / gradual patterns. This works because the frequent itemsets can be expressed as an *intent* (the itemset) and an *extent* (the transactions supporting the itemset), thus fitting well into a Galois lattice from which closure definition is derived. On the other hand, periodic patterns have a third component, which is the period. We have shown that indeed periodic patterns could be represented as **triadic concepts**, for which it is known that no closure relation can be defined in the traditional sense of Galois lattices [CBRB09].

We present in the following definitions how periodic patterns can be represented in a triadic setting.

Definition 4. *(Periodic Triadic Context)* A **periodic triadic context** is defined as a quadruple $(\mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{Y})$ where $\mathcal{I}$ is the set of items, $\mathcal{P}$ is the set of periods, $\mathcal{D}$ is the set of transactions, and $\mathcal{Y}$ is a ternary relation between $\mathcal{I}$, $\mathcal{P}$ and $\mathcal{D}$, i.e. $\mathcal{Y} \subseteq \mathcal{I} \times \mathcal{P} \times \mathcal{D}$.

TABLE 1.3: Representation of the relation $\mathcal{Y}$.

$\mathcal{I}/\mathcal{P} - \mathcal{T}$	2										3									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
a	×		×		×			×		×					×			×		
b	×		×		×			×		×					×			×		
...																				

$\mathcal{I}/\mathcal{P} - \mathcal{T}$	4										5									
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
a	×				×								×		×			×		×
b	×				×								×		×			×		×
...																				

Example: Given the dataset shown in Table 1.1, the corresponding periodic triadic context is shown in Table 1.3. Each cross in the table represents an element of the ternary relation $\mathcal{Y}$. For example, $P_1(\{a\}$, **2** ,5, $\{(1,3)(8,2)\})$ in Table 1.2 is transformed into the triples $(a, 2, t_1), (a, 2, t_3), (a, 2, t_5), (a, 2, t_8), (a, 2, t_{10})$ shown by the corresponding crosses in Table 1.3. For simplicity, items not forming any cycle of any possible period are not shown on the table.

Definition 5. *(Frequent Triple)* Given a minimum support threshold min_sup, a triple (I, P, T), with $I \subseteq \mathcal{I}$, $P \subseteq \mathcal{P}$, $T \subseteq \mathcal{D}$ and $I \times P \times T \subseteq \mathcal{Y}$, is **frequent** if and only if $I \neq \emptyset$, $P \neq \emptyset$ and $|T| \geq min_sup$.

Example: In Table 1.3, given a minimum support threshold of two transactions ($min_sup = 2$), we can observe several frequent triples such as $(\{a\}, \{2, 4\}, \{t_1, t_5\})$ or $(\{a, b\}$, $\{2\}, \{t_1, t_3, t_5\})$, since the number of transactions forming those triples is greater or equal to 2.

Definition 6. *(Periodic Concept)* A **periodic concept** of a periodic triadic context $(\mathcal{I}, \mathcal{P}, \mathcal{D}, \mathcal{Y})$ is a frequent triple (I, P, T) with $I \subseteq \mathcal{I}$, $P \subseteq \mathcal{P}$ and $T \subseteq \mathcal{D}$, such that none of its three components can be enlarged without violating the condition $I \times P \times T \subseteq \mathcal{Y}$.

Example: In Table 1.4, we can observe the set of periodic concepts extracted from the set of frequent triples obtained from the dataset shown in Table 1.3. The triples forming this set are periodic concepts since it is not possible to extend any of the attributes of the triple without violating the relation $\mathcal{Y}$.

TABLE 1.4: Set of periodic concepts

Periodic Concepts
$T_1(\{a,b\},\{2\},\{t_1,t_3,t_5,t_8,t_{10}\})$
$T_2(\{a,b\},\{2,4\},\{t_1,t_5\})$
$T_3(\{a,b\},\{2,5\},\{t_3,t_5,t_8,t_{10}\})$
$T_4(\{a,b\},\{2,3,5\},\{t_5,t_8\})$
$T_5(\{a,b\},\{2,3,4,5\},\{t_5\})$

One of our major contributions, presented in [CBT+12], was to define the notion of **Core Periodic Concepts (CPC)**, which is a condensed representation for the triadic concepts of periodic patterns, based on properties of the period.

Definition 7. *(Core Periodic Concept)* A periodic concept (I,P,T) is a **Core Periodic Concept** if there does not exist any other periodic concept (I',P',T') such that $I=I'$, $P'\subset P$ and $T'\supset T$.

Example: In Table 1.5, we can observe the set of CPC extracted from the set of periodic concepts shown in Table 1.4. For instance, $T_2(\{a,b\},\{2,4\},\{t_1,t_5\})$ is not a CPC since there exists $T_1(\{a,b\},\{2\},\{t_1,t_3,t_5,t_8,t_{10}\})$ with the same itemset $\{a,b\}$, a smaller set of periods $\{2\}\subset\{2,4\}$ and a bigger set of transactions $\{t_1,t_3,t_5,t_8,t_{10}\}\supset\{t_1,t_5\}$.

TABLE 1.5: Set of CPC

Core Periodic Concepts
$M_1(\{a,b\},\{2\},\{t_1,t_3,t_5,t_8,t_{10}\})$

In [CBT+12], we had no efficient way to compute the core periodic patterns, and had to use a two step process computing all the triadic concepts first with the DPeeler algorithm [CBRB09] and post-processing them to extract the CPC.

The difficulty of the proposed approach for efficient mining is that determining if a triadic concept is a CPC requires to compare it with other triadic concepts. Such approach is symptomatic of DAG-shaped enumeration strategies, which are hard to predict and thus require an exponential space complexity to store all the triadic concepts found so far in order to determine if a given triadic concept is a CPC or not.

However, in recent works of Patricia Lopez Cueva that haven't been published yet but that will soon appear in her PhD document, we deepened our analysis of CPC and proposed a property that we called **connectivity**. This property allows to determine, for any triadic concept, if it is a CPC or not, *without relying on any other triadic concept*. It is based on necessary relations that exist between periods and transactions of a CPC. This property is needed to build a first parent test for CPC.

From an enumeration point of view, in the general case of n-ary patterns [CBRB09], the components of the tuples making the patterns are independant and have thus to be augmented independently. This complicates the enumeration strategy. However with CPC, thanks to the strong connection between periods and transactions that we isolated, we have shown that it is possible to make an enumeration strategy based on the itemset part of the CPC, thus exploiting LCM principles. The periods and transactions require additional computations in the recursive loop, however these computations only require polynomial time and space, and can benefit from dataset reduction mechanisms.

The complete algorithm for mining directly CPC is called PerMiner. The pseudo-code of PerMiner is shown in Algorithm 2. The algorithm starts by computing the CPC for $\perp$, in case an itemset appears

Algorithm 2: The PerMiner algorithm

```
1  procedure PerMiner(D, min_sup);
   Data: dataset D, minimum support threshold min_sup
   Result: Output all Core Periodic Concepts that occur in D
2  begin
3      if |D| ≥ min_sup then
4          ⊥_clo ← ⋂_{t∈D} t
5          output (⊥_clo, {1..|D|/2}, D)
6          D_⊥clo = {t \ ⊥_clo | t ∈ D}
7          foreach e ∈ I with e ∉ ⊥_clo do
8              perIter(⊥_clo, D_⊥clo, e, ∅, min_sup)
9  Function perIter(X, D_X, e, min_sup, min_red)
10
       Data: Itemset of a discovered CPC X, reduced dataset D_X, item e, minimum support
             threshold min_sup
       Result: Output all Core Periodic Concepts whose itemset is prefixed by X and whose
               transactions are in D_X, with minimal support min_sup.
11     begin
12         B := getPeriods(tidlist(X ∪ {e}), min_sup)      /* Period computation */
13         G := group(B)
14         S ← ∅                                           /* Closure computation */
15         foreach g ∈ G do
16             X' := ⋂_{t∈g.occs} t
17             S := S ∪ (X', g.periods, g.occs)
18         S := filter(S);                                 /* First parent test */
19         enum ← ∅                                        /* Itemset enumeration */
20         foreach (X', P, T) ∈ S do
21             if firstParentTest(X', (X, e)) then
22                 Q = X ∪ X'
23                 output (Q, P, T)
24                 if Q ∉ enum then
25                     D_Q = reduce(D_X, Q, e, min_sup)
26                     foreach i ∈ augmentations(Q, (X, e), D_Q) do
27                         perIter(Q, D_Q, i, min_sup)
28                     enum := enum ∪ Q
```

in all transactions. If such case, we have shown that the correct set of periods to return is $\{1..|D|/2\}$ [Cue13].

Then for all items that can augment $\perp_{clo}$, the recursive function $perIter$ is called. This function can be explained as follows:

- as LCM, it takes as parameter *the itemset part* of an already found CPC and an augmentation, together with a reduced dataset and a support threshold
- the algorithm first computes all the periods that are admissible for the augmented itemset (line 12). This steps exploits the properties of CPC to return only periods likely to appear in CPC. For more details, the interested reader is referred to the PhD of Patricia Lopez Cueva [Cue13].
- A CPC has a set of periods. All the periods found at previous step that have the same transaction list are regrouped in a single set of periods (line 13), that is guaranteed to participate in a CPC. These sets of periods make the group G.
- In lines 14-17, for each group of periods the itemset is maximally extended by intersection of the transactions supporting the group. After this step, S is guaranteed to contain a set of CPCs.
- All the CPCs of S do not necessarily have (X, e) as first parent. CPCs of S that include other CPCs of S (itemset-wise) are suppressed in line 18. Then a first parent test inspired by LCM but also from works on polynomial-delay enumeration by Boley et al. [BHPW10] is performed in line 21.
- The found CPC are outputted (line 23) and the dataset is reduced on the transactions containing Q (line 25).
- Last, augmentations are determined in a way similar to LCM (line 26) and recursive calls are performed.

We have proved in [Cue13] that this algorithm was sound and complete. We also proved that its complexity was **polynomial space** and **polynomial delay**.

We compared experimentally a C++ implementation of PerMiner to our previous three-step approach, which chains a dataset preprocessing to convert it to triples, the original C++ implementation of DPeeler [CBRB09] and a C++ postprocessor we wrote that filters CPCs from the periodic concepts. The experiment is conduced on an Intel Xeon 7460 @ 2.66 GHz with 64 GB of RAM. The dataset is a synthetic dataset produced by a modified version of the IBM synthetic data generator (details in [CBT+12] and [Cue13]).

Figure 1.6 shows a comparison of PerMiner and 3-STEP performance on a simple synthetic dataset with 1000 transactions and 200 distinct items. The curve on the left shows the wall clock time w.r.t. the minimum support value, whereas the curve on the left shows the number of CPCs found. Programs were stopped after 10^5 seconds. It can be seen that for minimum support values lower than 30%, 3-STEP run time exceeds 10^5 seconds of execution, whereas PerMiner is one to two orders of magnitude faster, demonstrating the interest of the approach.

We then compared both algorithms running time on a real execution trace of a multimedia application used for board testing at STMicroelectronics. The raw trace has 528,360 events in total. It is transformed into a matrix of 15,000 transactions (1 transaction representing 10ms of trace) and 72 distinct items, having in average 35 items per transactions.

Figure 1.7 shows the execution time w.r.t. minimum support for computing CPCs in this trace, as well as the number of CPCs extracted. PerMiner is one to two orders faster than 3-STEP in this experiment.

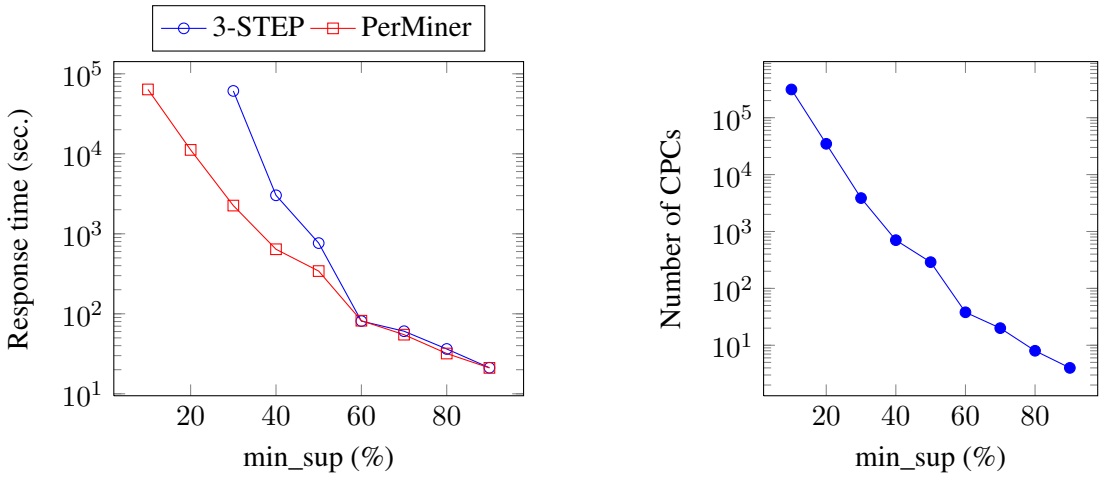


FIGURE 1.6: Minimum support threshold comparative plot

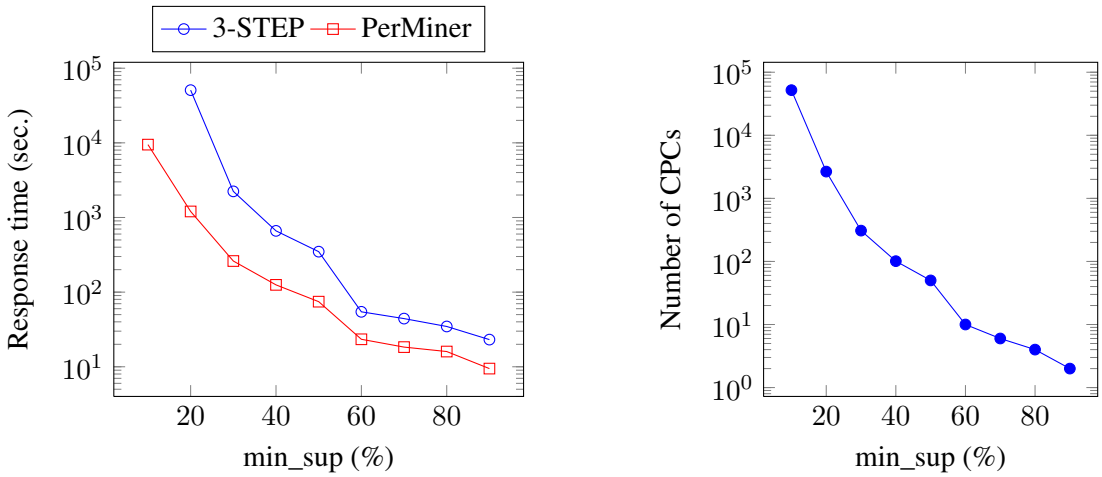
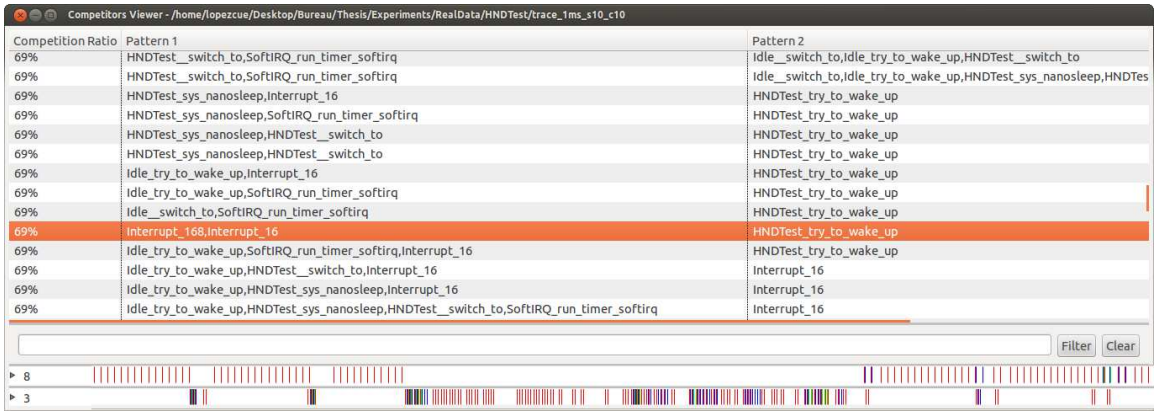


FIGURE 1.7: Minimum support threshold comparative plot

This experiment demonstrates that PerMiner good scalability allow it to compute CPCs from large real world traces in a reasonable amount of time, and that it is the only approach able to do so for low support value, that are need for fine grained analysis of the trace.

Regarding the interest of the periodic patterns found, we show in Figure 1.8 two periodic patterns extracted from a test application of STMicroelectronics, called *HNDTest*.



Competition Ratio	Pattern 1	Pattern 2
69%	HNDTest__switch_to,SoftIRQ_run_timer_softirq	Idle__switch_to,Idle_try_to_wake_up,HNDTest__switch_to
69%	HNDTest__switch_to,SoftIRQ_run_timer_softirq	Idle__switch_to,Idle_try_to_wake_up,HNDTest_sys_nanosleep,HNDTes
69%	HNDTest_sys_nanosleep,Interrupt_16	HNDTest_try_to_wake_up
69%	HNDTest_sys_nanosleep,SoftIRQ_run_timer_softirq	HNDTest_try_to_wake_up
69%	HNDTest_sys_nanosleep,HNDTest__switch_to	HNDTest_try_to_wake_up
69%	HNDTest_sys_nanosleep,HNDTest__switch_to	HNDTest_try_to_wake_up
69%	Idle_try_to_wake_up,Interrupt_16	HNDTest_try_to_wake_up
69%	Idle_try_to_wake_up,SoftIRQ_run_timer_softirq	HNDTest_try_to_wake_up
69%	Idle__switch_to,SoftIRQ_run_timer_softirq	HNDTest_try_to_wake_up
69%	Interrupt_168,Interrupt_16	HNDTest_try_to_wake_up
69%	Idle_try_to_wake_up,SoftIRQ_run_timer_softirq,Interrupt_16	HNDTest_try_to_wake_up
69%	Idle_try_to_wake_up,HNDTest__switch_to,Interrupt_16	Interrupt_16
69%	Idle_try_to_wake_up,HNDTest_sys_nanosleep,Interrupt_16	Interrupt_16
69%	Idle_try_to_wake_up,HNDTest_sys_nanosleep,HNDTest__switch_to,SoftIRQ_run_timer_softirq	Interrupt_16

FIGURE 1.8: Two periodic patterns from HNDTest application

They come from a small tool that we wrote to find *competitor* periodic patterns, i.e. patterns where one is active in the "gap" of periods of the other. Such patterns are likely to show disturbances in the system.

The timeline of execution is shown on the bottom of the figure, with the first pattern on top and the other under. Color marks signal occurrences of the corresponding pattern (different colors indicate different instances of the pattern with different offsets). The first pattern, $\{Interrupt_168, Interrupt_16\}$, is a periodic poll of the USB bus: it comes from the system. The second pattern, $\{HNDTest_try_to_wake_up\}$, is increasing its activity when the test application wants to get out of idle state and work. Here we can note that intense activity of $\{HNDTest_try_to_wake_up\}$ has temporarily disabled USB checks: it is likely that the application has masked too aggressively other interruptions for a too long time. This behavior is dangerous, as if during this time data is received from (or sent to) an USB source, there is a risk of data loss.

This work on PerMiner shows that even for complex patterns that do not fit intuitively in the classical setting of LCM, it is possible to design an efficient enumeration strategy based on the same principles, with excellent experimental results.

D ParaMiner: towards (pattern mining) world domination

At the beginning of this chapter, I have deliberately presented LCM in a very generic way: an algorithm structure where one has to instanciate several key functions:

- $support$
- Clo
- $firstParentTest$
- $augmentations$

- $reduce$

We have seen, for GLCM and PerMiner, that instanciating these functions was not trivial and required involved work from a pattern mining researcher. This is unfortunate, as it means that for practitioners and data owners, adapting the LCM algorithm to their pattern mining tasks is mostly out of reach.

The main difficulty is handling the complex dependencies between all these functions. Especially, designing $firstParentTest$ and $augmentations$ that suits the patterns at hand is tough.

In pioneering works about **generic pattern mining**, Arimura and Uno [AU09] and Boley et al. [BHPW07, BHPW10] proposed novel enumeration methods of closed patterns. They have found a large class of pattern mining problems for which $firstParentTest$ and $augmentations$ can be fixed, Furthermore, the test $support_{D_P}(P \cup \{e\}) \geq \varepsilon$ can be replaced by a more generic test $Select(P \cup \{e\})$, allowing to have interest function for patterns that go beyond the classic frequency test. They have shown that as long as the $Select$ and Clo functions provided were of polynomial time and space complexity, the complete algorithm exhibited polynomial space complexity and polynomial delay time complexity.

However, despite their interest and these strong complexity guarantees, no implementation was provided for these works, which remained purely theoretical. The reason is simple: even if $Select$ and Clo are polynomial, in most existing pattern mining problems these functions will have to scan a dataset that can have millions of transactions. This will have to be done at least once per pattern found, leading to extremely long running times if done naively.

Our goal in the PhD of Benjamin Négrevergne was to enrich the state of the art in generic pattern mining with an approach making it easy for practitioners to add new pattern mining problems, while having performances close to the best specialized algorithms. We designed ParaMiner, a generic pattern mining algorithm based on the principles proposed by Arimura and Uno and Boley et al., having the same complexity guarantees, but exhibiting actual performance close to those of the best specialized algorithms.

ParaMiner differs from existing generic pattern mining approach in the following ways:

- DMTL [AHCS+05] captures a wider range of pattern mining problems but does not provide recent enumeration techniques based on efficient handling of closure, and thus is not competitive with specialized algorithms. Its great flexibility, translated in an implementation based on C++ templates, can also make it difficult to use for non-data mining specialists.

- iZi [FMP09], like ParaMiner, focuses on different patterns based on set-encoding, but does not integrate closure and is based on older enumeration techniques.

Our solution to provide an efficient generic pattern mining algorithm is to tackle the last function that I presented in the pseudo-code of LCM (Algorithm 1) and that has been left untouched in the previous theoretical works, the $reduce$ function. The difficulty is that $reduce$ is closely related to $Select$ and Clo, and thus before us no related works tackled the problem of proposing a *generic* $reduce$ function.

By introducing some constraints (satisfied by most existing pattern mining problems) on the form of $Select$ and Clo, we could propose an efficient and generic $reduce$ function.

In the following, I will briefly explain the theoretical foundations on which ParaMiner is based. I will then introduce the constraints on $Select$ that we impose, and define the range of pattern mining problems that ParaMiner can solve. I will then explain intuitively how the algorithm operates, with a special focus on our generic $reduce$ function.

In order to explain the theoretical foundations of ParaMiner, it is first necessary to define formally the notion of pattern w.r.t. the predicate $Select$:

Definition 8. Given a dataset $\mathcal{D}_\mathcal{I}$ built over a ground set $\mathcal{I}$ and a selection predicate $Select$, $X \subseteq \mathcal{I}$ is a **pattern** in $\mathcal{D}_\mathcal{I}$ if and only if:

1. X occurs in $\mathcal{D}_\mathcal{I}$
2. $Select(X, \mathcal{D}_\mathcal{I}) = true$

Note that differing from Arimura and Uno [AU09], we integrate the dataset from the ground up in our definitions. Similarly, we define formally the notions of **augmentation** and **first parent**, that have only been presented intuitively at the beginning of this chapter:

Definition 9. A pattern Q is an *augmentation* of a pattern P if there exists $e \in Q \setminus P$ such that $Q = P \cup \{e\}$.

Definition 10. Let P be a closed pattern, and Q the closure of an augmentation $P \cup \{e\}$ of P such that $P < P \cup \{e\}$. P is the *first parent* of Q if there does not exist a closed pattern $P' < P$ and an element e' such that $P' < P' \cup \{e'\}$ and Q is the closure of $P' \cup \{e'\}$.

Now, in order to get a deeper understanding on pattern enumeration with first parent, we first need to present some concepts from *set theory*.

The first one is the concept of **set system**, which allows to rewrite neatly the goal output of any pattern mining algorithm.

Definition 11. A *set system* is an ordered pair $(E, \mathcal{S})$ where E is a set of elements and $\mathcal{S} \subseteq 2^E$ is a family of subsets of E.

With $\mathcal{I}$ the ground set and $\mathcal{F}$ the set of patterns defined above, it can be immediately seen than $(\mathcal{I}, \mathcal{F})$ is indeed a set system. The major interest of a set system is when there exists an underlying structure linking together all the elements of the set sytem. Such structure can be characterized by **accessibility** properties, which we list below, from the weakest to the strongest.

Definition 12. A set system $(\mathcal{I}, \mathcal{F})$ is **accessible** if for every non-empty $X \in \mathcal{F}$, there exists some $e \in X$ such that $X \setminus \{e\} \in \mathcal{F}$.

Definition 13. A set system $(\mathcal{I}, \mathcal{F})$ is **strongly accessible** if it is accessible and if for every $X, Y \in \mathcal{F}$ with $X \subset Y$, there exist some $e \in Y \setminus X$ such that $X \cup \{e\} \in \mathcal{F}$.

Definition 14. A set system $(\mathcal{I}, \mathcal{F})$ is an **independence** set system if $Y \in \mathcal{F}$ and $X \subseteq Y$ together imply $X \in \mathcal{F}$.

As one can see, these properties are based on extending patterns with one element, i.e. are based on the augmentation operation. They are thus closely linked with the first parent operation. The works of Arimura and Uno [AU09] and Boley et al. [BHPW10] provide generic first parent operations and their complexities for the three accessibility cases:

- when the set system is an independence set system, the first parent test is very simple: one has to check if $Q > P$ or not
- when the set system is strongly accessible, Boley et al. have shown in [BHPW10] that the first parent test could be done with an intersection between Q and a small list of size $O(log(|\mathcal{I}|))$.
- when the set system is accessible, Arimura and Uno have shown in [AU09] that the first parent test was polynomial. However it is much more expensive than the previous cases, as testing for first parent may lead to the reverse enumeration of a whole branch of the enumeration tree.

Our first constraint was thus to restrict the $Select$ function to those leading to a *strongly accessible* set system. We are guaranteed to have a very efficient first parent test, while covering many existing existing pattern mining problems [NTRM13].

We can now give in Algorithm 3 the pseudo code of the ParaMiner algorithm, where $firstParentTest$ and $augmentations$ have been fixed to those proposed in [BHPW10] for the strongly accessible case. Note that to ensure the soundness of this first parent test that relies on an *extension list*, the structure of the algorithm is slightly modified from what has been shown in LCM and PerMiner. $expand$ no longer takes the augmentation e as an argument, and starts by looping over all possible enumerations (line 8). The rest of the algorithm is very similar in structure to LCM and PerMiner.

Algorithm 3: ParaMiner

Data: dataset D, selection criterion $Select$, closure operator Clo
Result: Output all closed patterns in D

```
1  begin
2      ⊥_clo ← Clo(⊥, D)
3      output ⊥_clo
4      expand(⊥_clo, D, ∅)

5  Function expand(P, D_P, EL)
6
       Data: Closed pattern P, reduced dataset D_P, extension list EL
       Result: Output all closed patterns descending from P in the enumeration tree
7      begin
8          foreach e ∈ I s.t. e ∉ P and e ∈ D_P do      /* Augmentation enumeration */
9              if Select(P ∪ {e}, D_P) = true then                  /* Pattern test */
10                 Q ← Clo(P ∪ {e}, D_P)                  /* Closure computation */
11                 if Q ∩ EL = ∅ then                       /* First parent test */
12                     output Q
13                     D_Q = reduce(D_P, Q, EL)
14                     expand(Q, i, D_Q, EL)
15                     EL ← EL ∪ {e}
```

In order to propose a generic $reduce$ function that doesn't compromise the soundness and completeness of ParaMiner, it has to:

- avoid compromising the results of $Select$ and Clo (lines 9 and 10)
- keep all the necessary augmentations in the dataset for the loop of line 8

In usual pattern mining algorithms, dataset reductions have two main operations:

1. they suppress the lines where the pattern does not occur
2. from the remaining lines, they suppress the elements that are unlikely to belong to a descendant of the pattern (via first parent extension)

The generic $reduce$ function that we proposed in ParaMiner follows the same structure. In order to guarantee that $reduce$ does not compromise $Select$ results, we introduced a supplementary constraint on $Select$, that we called **decomposability**:

Definition 15. A selection criterion $Select(P, D)$ is **decomposable** if it can be expressed as the conjunction of two predicates, one on P and the other on the tidlist of P in D:

$$Select(P, D) = p_1(P) \wedge p_2(tidlist(P, D))$$

The writing of predicate p_2 guarantees that the transactions that do not contain the pattern have no influence on the result of $Select$. Thus operation 1) of dataset reduction can be conduced without harmful effects. Moreover, predicate p_2 does not have as input the precise elements of the transactions containing P, so it is not impacted by operation 2) of dataset reduction. Predicate p_1 guarantees that only the elements of a pattern have an impact on $Select$. Operation 2) of dataset reduction is thus only required to preserve, in the transactions where P occurs, the elements of P and all the elements leading to augmentations and closures of P by first parent enumeration. All the other elements can be safely removed.

The difficulty is thus to determine, in the $expand$ function for a given P augmented with a given e, what are the elements that are guaranteed not to be augmentations and closures of descendants of P by first parent extension.

According to line 11 of Algorithm 3, there cannot be any element of EL in the patterns output in line 12, thus such elements are good candidates for removal from the dataset. Some of these elements participate in closures computed in line 10 and removing them would make these closures incorrect in a way that would systematically defeat the first parent test of line 11, leading to many duplicates in the enumeration. Hence, only the elements of EL not participating in any further closure can be removed.

Lets consider a simple example. Suppose that the ground set is $\mathcal{I} = \{A, B, C, D\}$, that $P = \{C\}$ and that $EL = \{A, B\}$. Consider the following (reduced) dataset for pattern $\{C\}$:

$$D_{\{C\}} = \begin{array}{lcccc} t_1: & A & B & C & D \\ t_2: & A & & C & D \\ t_3: & & & C & \end{array}$$

The only augmentation of $\{C\}$ than can lead to a closure is by element D, appearing in transactions t_1 and t_2. When performing a closure on t_1 and t_2, element A of EL will be kept in the closure as it appears in both transactions. However element B of EL will be suppressed by the closure: such element can be suppressed in advance from the dataset by the reduction operation.

Such elements can be computed with an algorithm detailed in [NTRM13]. Intuitively, this algorithm groups transactions in the dataset according to their elements not in EL and containing elements different from P. In the above example, there would be only one group, containing transactions with $\{D\}$: $\{t_1, t_2\}$. For such groups, all elements of EL that do not appear in all transactions are removed, as they cannot be part of a closure. We have also proved in [NTRM13] that this operation has no impact on the soundness and completeness of ParaMiner.

This completes the presentation of our generic dataset reduction, that we called $ELReduction$. In order to show its impact on ParaMiner's execution time, we present in Figure 1.9 an experiment comparing ParaMiner with ParaMiner$_{\text{nodsr}}$, which is a stripped down version of ParaMiner where the dataset reduction has been removed. $Select$ and Clo are set up for frequent itemset mining, the dataset is the well known Mushroom dataset from FIMI repository [Goe04], and the machine is an Intel Core i7 7560 with 64 GB of RAM.

Without the dataset reduction, despite the small size of the dataset, the ParaMiner$_{\text{nodsr}}$ algorithm can take more than 1000 seconds of execution for lowest support values: being that long on a toy dataset, the

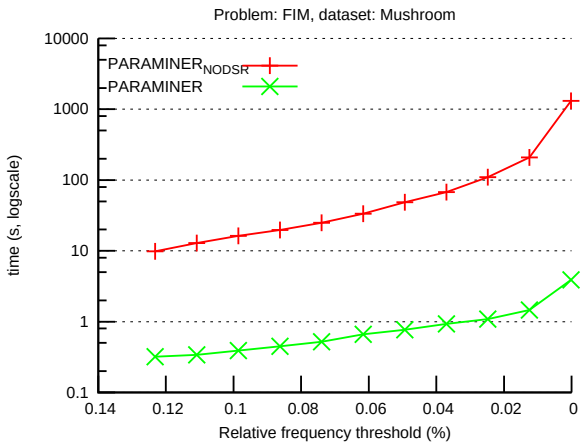


FIGURE 1.9: PARAMINER vs PARAMINER$_{no_dsr}$, sequential execution times on the dataset Mushroom (a).

algorithm can't be exploited on more realistically sized datasets. On the other hand, $ParaMiner$ with ELreduction activated is two orders of magnitude faster than ParaMiner$_{\text{nodsr}}$ for lowest support values, mining Mushroom in less than 10 seconds, which is in line with state of the art algorithms (LCM requires 1s).

This confirms the interest of integrating a generic dataset reduction in a generic pattern mining algorithm. In the next chapter, detailing the parallel processing capacities of ParaMiner, more comparative experiments with state of the art pattern mining algorithms will be presented.

To conclude, we have presented in this last section our approach for efficient and generic pattern mining. Currently, we have instantiated three pattern mining problems in the ParaMiner's framework:

- closed frequent itemset mining
- closed frequent gradual itemset mining
- closed frequent relational graph mining

The detail of $Select$ and Clo for these problems, as well as proofs of strong accessibility and decomposability, can be found in [NTRM13].

ParaMiner is an OpenSource program, available at http://membres-liglab.imag.fr/negrevergne/paraminer/.

E Perspectives for "Mine more"

In the topic of this chapter, my perspectives are centered around ParaMiner. The results published in [NTRM13] and presented in this document (theoretical in current chapter and practical in next chapter) show that we succeeded in designing an efficient, generic and parallel pattern mining algorithms. However this is only a first step, far from my self-assigned goal: make ParaMiner and its successors reference data mining algorithms, exploited by many practitioners for analyzing real data, and used as a starting point or baseline by pattern mining researchers producing new and better algorithms.

To reach this goal many perspectives have to be explored:

- From a practical point of view, making new ParaMiner instances must be as simple as possible. For the moment we are working on the engineering side, by producing a better and easier to understand

implementation, in which making $Select$ and Clo will be simpler than in our current prototype. However, a practitioner will still have to verify manually strong accessibility and decomposability properties of its $Select$ criterion, which is not straightforward. An exciting perspective would be to propose basic $Select$ operators (strongly accessible and decomposable), and an algebra for composing them while preserving these properties. This would allow to propose on top of this algebra a small DSL (Domain Specific Language) for writing $Select$ criterion, similarly to SQL which is built on top of relational algebra. The difficulty is that the strong accessibility property is ill-studied and not trivial to understand. Thus designing the proposed algebra could be easier for researchers with a good background in set theory, which are rare in the data mining area.

- Due to the lack of work on generic pattern mining, it is difficult to have an intuitive idea of the class of pattern mining problems which can be handled by ParaMiner. A theoretical perspective, connected with the previous one, would be to have a better characterization of the pattern mining problems that can be handled by ParaMiner. Such problems first have to be *representable as sets*. Here we can benefit from recent works from Lhouari & Petit [NP12], which present the notions of problems *(efficient) weakly representable as sets*. Such notions can be used as starting points to better understand the theoretical characterization of pattern mining problems that can be handled by ParaMiner. The difficulty, here also, is to take into account the notion of strong accessibility.

- Beside understanding what problems ParaMiner can handle, it is also important to expand the range of ParaMiner so that is can handle more problems and thus be more useful:

  - In [NTRM13], we presented the notion of **partial strong accessibility**, which is a way to handle problems where the strong accessibility property is only verified above a given border in the search space of patterns. Such case arises in constraint pattern mining, especially with *monotonous constraints*. An straightforward perspective would be to design an efficient improvement of ParaMiner using this notion. The difficulty being to propose and efficient and generic way to reach the border, from which it is possible to switch to the classical ParaMiner enumeration. Another perspective could be to extend this work to a more generic case where strong accessibility is only verified in parts of the search space, with "jumps" between these parts.

  - A difficult case for generic pattern mining is handling **sequences**, as many sequence mining problems cannot be encoded efficiently in a set framework. A solution is to restrict oneself to sequence mining problems where known set representations exists, for example *frequent rigid sequences with wildcards* [NP12], or by adapting work done on *conjuctive sequences* [RCP08]. Even in such cases, ParaMiner may require some adaptations to be used. A more ambitious perspective which I am very interested in is to get rid of sets altogether, and to adapt ParaMiner to "natively" handle sequences. Such radical solution requires to adapt the strongly accessible set systems framework to a framework based on sequences. A difference with sets is that there are several ways to define inclusion of sequences, hence this will lead to several accessibility definitions. Another point is that many practical sequence mining problems deal with a dataset which is a single long sequence, i.e. where there are no transactions. With the most relaxed $Select$ criterion for sequences (for example when the patterns are *episodes* [MTV97]), such problems leave very little room for a dataset reduction to operate, making it difficult to design. To get a better understanding on these problems, I am starting a collaboration with Thomas Guyet from IRISA lab, on the topic of generic sequence mining.

Contents

A Multicore processors
B Melinda: a simple API for parallel data-driven programs
C Breaking the memory wall for ParaMiner
D ParaMiner vs specialized parallel algorithms
E Perspectives for "Mine faster"

CHAPTER

2

MINE FASTER: *Multicore processors to the rescue*

Most of the processors exploited in today's computing platforms, from smartphones to desktops to high-end clusters, are **multicore** processors. These processors combine several independent processing units into a single chip package. In order to exploit their power to have faster and more scalable algorithms, it is necessary to write **parallel algorithms** adapted to the specificity of these processors.

There are two major difficulties with this task:

- it is inherently difficult to program parallel algorithms
- performance behavior of parallel algorithms is very different from traditional sequential algorithms.

I addressed both issues in the works that I conduced with my PhD and Master students since 2007. Most of these works were done in collaboration with Jean-François Méhaut, Professor at University Joseph Fourier and specialist in parallelism and performance evaluation. After a presentation of multicore processors and of parallel programming difficulties, I will briefly present our solutions in this chapter.

- To alleviate the programmining difficulty, we choose in the Master and PhD of Benjamin Négrevergne to propose a custom parallelism library, called **Melinda**. Melinda is both easy to use and customize, well adapted to pattern mining algorithms, and has low overheads.
- With regard to parallel performance, a previous study [TP09] have shown on a particular tree mining problem that the parallel architecture of a multicore processor was particularly stressed by a classical pattern mining algorithm, especially the bus between the memory and the cores. Thanks to the genericity of ParaMiner, we could extend this experiment to several different pattern mining problems. We also contributed a generic solution that improves the parallel performance of all pattern mining problems that can be handled by ParaMiner.

- Thanks to the two contributions presented above, we show on comparative experiments that ParaMiner can have performance on par with state of the art specialized algorithms.
- At the end of this chapter, I will give some perspectives in order to further improve parallel pattern mining algorithms on multicores processors, and also briefly discuss perspectives on clusters driven by MapReduce.

A Multicore processors

The goal of a computer processor is to execute as many operations per second as possible. Making more operations allows to run more complex algorithms and to process larger volumes of data in a reasonable amount of time.

Processor designers have thus strived to increase the performance of their processors. They are helped by the empirical **Moore's law**: *the number of transistors on a die doubles every two years*.

Until 2005 they exploited a simple corollary of this law: the number of transistors doubles because they can be engraved with a finer resolution, and this finer resolution allows to increase the frequency of operation of the processor. This is known as **frequency scaling**, and it as been the dominant paradigm in computer architecture for years. For algorithms, it means that a "magic" doubling of performance can be expected by waiting two years and buying a new processor: a very convenient feature.

In 2005, physical limits due to the difficulty of heat dissipation were reached, preventing further frequency scaling, and processors have been locked since then to a 2-3 GHz frequency. However, Moore's law remained valid. The solution processor designers found to continue increasing the number of operations per second their processors could do was thus to pack several processors on a single die, with independent execution units but possibly shared caches. Each of these processors is called a **core**, and the complete package is called a **multicore processor**. First solutions proposed multicore processors with two cores, but nowadays four to eight cores are common.

We show a recent laptop dual-core processor (Intel Core i5-2520M) in Figure 2.1, and a server processor with eight cores (Intel Xeon X7560) in Figure 2.2.

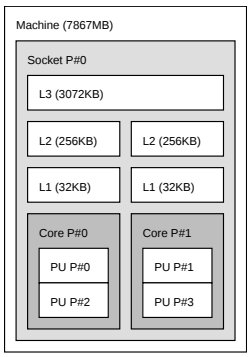


FIGURE 2.1: Topology of an Intel Core i5-2520M

These figures were produced with the `lstopo` tool [Ope13b], and allow to see for each processor (termed *Socket* in the figure) its cores and its cache hierarchy.

There are two ways to exploit the multiple cores of these processors in order to get more work done. The simple one is to execute independently one sequential application per core. This allows to use legacy applications unmodified, and to get the results of both applications in a time as long as the execution lime of the longest application. However, this approach does not allow to accelerate computation intensive applications such as pattern mining.

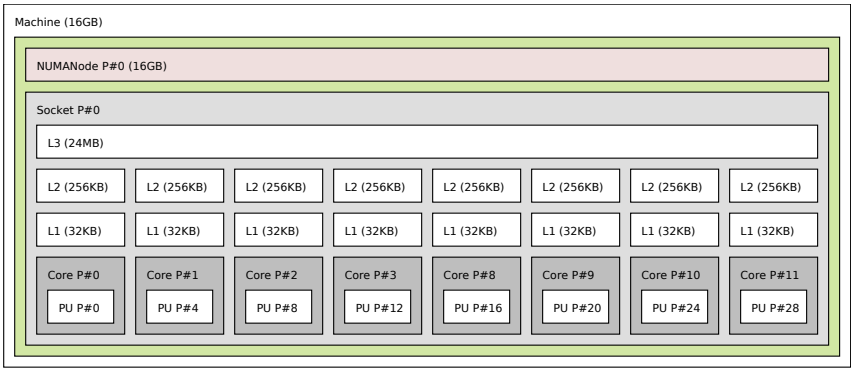


FIGURE 2.2: Topology of an Intel Xeon X7560

This is achieved with the other approach, which is to rewrite the application as a parallel program, capable of exploiting several *threads* of execution which will be dispatched over the cores and benefit from all the power of the multicore processor. The catch is that it is notoriously difficult to write parallel programs. One of the biggest difficulties is that in most applications, threads have to communicate together, for example to exchange results of dependent computations. This communication is done by writing values in memory locations known by both communicating threads. A precise order of access on these memory locations have to be enforced in order to guarantee correct results. This is done through **synchronization** primitives such as **locks** or **semaphores**, which are not trivial to use especially when many of them are involved. Failure in synchronization can lead in the worst case errors such as *deadlocks*, which block the program, or to undeterministic bugs difficult to hunt.

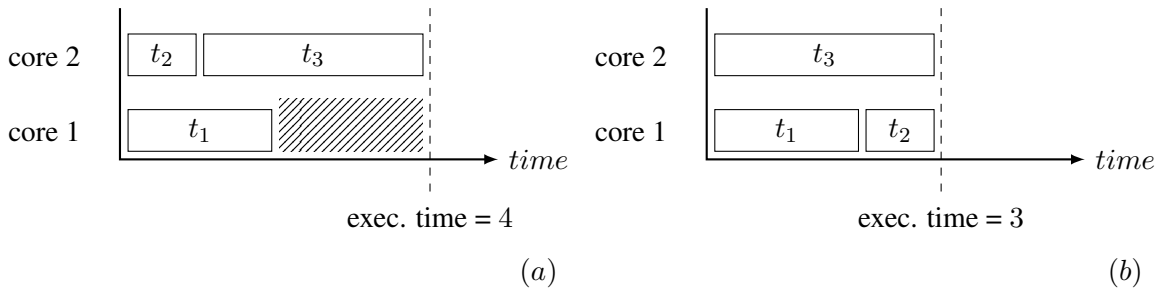


FIGURE 2.3: Unbalanced execution (a) vs. balanced execution (b).

Performance-wise, the main issues faced by programmers are:

- **synchronization issues**: in practice, what a synchronization mechanism such as a lock do is to make sequential a portion of the parallel program, in order for a thread to have exclusive access to a resource. During this exclusive access, other threads will not be working if they are waiting for the same resource. This alters parallel performance, and having too much synchronization in a parallel program can ruin the gain of performance from parallelism.

- **load balance issues**: another important issue is to guarantee that all threads have enough work to do until the end of execution. If not, some threads will be idle for some portion of the execution while others have too much work to do, a situation called **load unbalance** (shown in Figure 2.3 (a)). Having a good load balance (Figure 2.3 (b)) is especially difficult in applications with irregular and unpredictable workloads, which is often the case in pattern mining.

For both of these issues, researchers in parallelism have proposed many solutions, such as lock-free algorithms [CB04], work stealing techniques [BL99] or transactional memory [MMTW10].

There is a third issue that arises in multicore architectures, called the **memory wall**, which is more difficult to fix for the programmer and that results from *physical* limitations of the architecture. The typical communication architecture in a UMA (Unified Memory Access) multicore computer is shown in Figure 2.4.

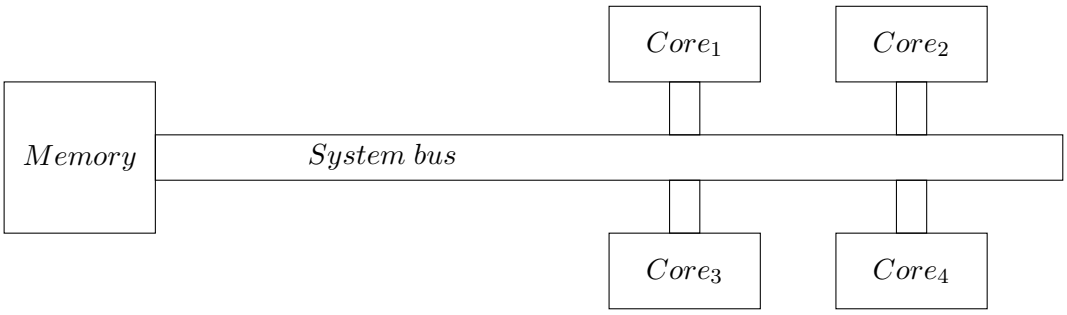


FIGURE 2.4: System bus, memory and cores

There is a system bus, on which are connected both the main memory, and all the cores of the multicore processor. All cores requests for data in memory thus transit by the same system bus. If there are too many of such requests, the bus can easily become saturated, and the memory response time is also limited, which can form waiting queues.

For technical and cost reasons, the bus capacity cannot be increased easily, neither can the memory response time. However, the number of cores is always increasing, leading to congestion situations when all of them are simultaneously making requests to the memory.

A partial solution, especially used in large system with several multicore processors is to use NUMA (Non-Uniform Memory Access) architectures, where each processor has a dedicated physical memory bank. However there still is one system bus, and when the application has bad locality properties there will be many "distant" accesses between cores and memories far from them, that can again saturate the system bus.

A more fundamental solution used by processor designers since many years is to use large *caches*, which are small and fast memories integrated on the processor die. Most applications are likely to reuse a data they accessed (*temporal locality*), or to access data close to that data (*spacial locality*). Cache memories can hold data access shortly before, and prefetch data that is likely to be used in a near future, accelerating memory accesses and limiting traffic on the bus. They are usually organized in a hierarchy, with the smallest, fastest and most expensive caches closest to each core, and larger, slower and cheaper caches shared by several cores. For example in the two processors shown in Figures 2.1 and 2.2, each core has a fast Level 1 (L1) cache of 32 KB, an intermediate L2 cache of 256 KB, and a large L3 cache shared by all the cores of the processor (3 MB for the Core i5, 24 MB for the Xeon).

Many papers have shown that for frequent itemset mining it was very important to make algorithms exploiting well the caches, such as [ZPL97], [GBP+05b].

However most pattern mining algorithms have been designed with single-core processors in mind, and thus are biased to sacrifice memory space in order to save CPU cycles. This is especially true for structured pattern mining algorithms (trees, graphs), which maintain data structures in memory that can become to large to fit any cache, thus requesting frequent accesses to the main memory. Tatikonda et al. [TP09] were the first to show the devastating effect of such bias on multicore processors. Due to intense communications between the cores and the memory, the bandwidth limit is quickly reached, and the algorithm can't scale to more than a few cores.

The solution that they proposed was simple: change the bias to take into account the characteristics of multicore architectures, and try to reduce as much as possible bandwidth usage, at the expense of extra computations by the cores. *Extra computations is the cheapest resource of recent CPUs*.

As a former tree mining specialist that designed tree [TRS04, TRS+08] and DAG mining [TTN+07] algorithms, I was especially receptive to the arguments of this paper, and it strongly influenced my thinking on how to do parallel pattern mining on multicores. Most of the experiments that we did to understand and improve the parallel scalability of our pattern mining algorithms originate from the ideas exposed by Tatikonda et al., that we tried to improve and generalize.

B Melinda: a simple API for parallel data-driven programs

In order to ease the burden of parallel programming, several APIs providing higher level interfaces have been proposed, such as OpenMP [Ope13a]. However, the tradeoff for easier programming is a lack of control on the parallel runtime, that can both hinder experiments reproducibility and performance optimization. For instance, with Jean-François Méhaut, we conduced initial experiments with OpenMP in C++ in 2007. These experiments showed that three major compilers (GNU `g++`, Intel `icc`, Portland `pgc++`) had a very different implementation of the way threads were handled, leading to large runtimes differences.

With Benjamin Négrevergne, our student (M2R then PhD), we thus decided to make a high level API that would be specialized for parallel pattern mining algorithms, and with a behavior that we could control and study. We called this API **Melinda**, as it derives from the Linda approach proposed by Gelernter in [Gel89].

The majority of pattern mining algorithms that we are interested in are depth first search algorithms that produce a vast amount of intermediary data. Melinda has thus been designed from the ground up to be *data driven*. It borrows to the Linda approach the idea of having an approach based on *tuples*:

- all threads can access a shared memory space called a **TupleSpace**
- the TupleSpace handle a set of tuples, each of which represent an unit of work for the threads
- the threads can either insert new tuples from the TupleSpace, or retrieve one.

The original Linda approach allows to have many different tuple formats simultaneously in the TupleSpace and complex retrieval operations on tuples to get only tuples verifying certain characteristics. Although this is convenient for the programmer, this can lead to high overheads in the Linda runtime, which are incompatible with data mining performance requirements. Thus our Melinda approach is a stripped-down version of Linda, where the tuples have a fixed format, and where tuple retrieval does not allow complex querying. This allows to have an easy to use framework, with very low runtime overheads.

The API of Melinda has two main primitives:

- **put(tuple):** inserts a tuple into the TupleSpace
- **get():** retrieves a tuple from the TupleSpace (the tuple is deleted from the TupleSpace)

Accessing the TupleSpace with these primitives is represented in the Figure 2.5.

First, at least one thread has to put one or more tuples in the TupleSpace to give work to the other threads. Then, during execution idle threads retrieve work (tuples) with get, while threads producing work (tuples) insert it in the TupleSpace with put.

With these settings, it is easy to write the depth first algorithms that we described in Chapter 1:

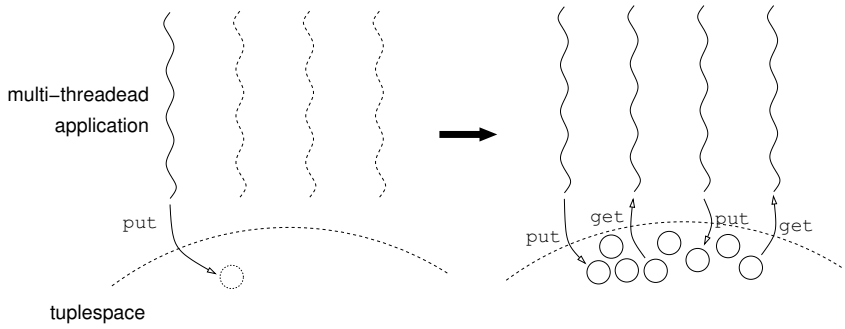


FIGURE 2.5: Accessing the TupleSpace with `put` and `get`

- a tuple contains the parameters of a recursive call
- instead of performing recursive calls, a thread creates a tuple with the necessary parameters, and put it into the TupleSpace
- idle threads call get in order to receive new tuples to process

We show in Algorithm 4 the ParaMiner algorithm exploiting the Melinda API. Note that for sake of readability, the TupleSpace itself is not shown: it is supposed to be a global variable used inside calls to functions $initTupleSpace$, $isTupleSpaceEmpty$, put and get.

As can be seen in line 7 for patterns descendants of $\perp$, and in line 18, calls to $expand$ have been replaced by calls to put. The threads themselves are initialized in lines 3-4, and perform the function $runThread$. This function, as long as there are tuples in the TupleSpace, retrieves them with get, and makes the necessary calls to $expand$ in order to process the pattern given in the tuple. For sake of simplification, the precise termination condition is not shown in the above pseudo-code. In practice, when all threads are blocked and waiting for tuples, it means that the TupleSpace is empty and that no thread is processing a tuple (which may generate new tuples), so the program is stopped.

The $expand$ function now keeps track of the current depth in the enumeration tree with parameter d. Thanks to the efficiency of dataset reduction, for high values of d usually $expand$ has only little work to do. This happens especially in simple pattern mining problems such as frequent itemset mining. For such deep nodes of the enumeration tree, the processing time of $expand$ can become close to the overheads of the parallel runtime overheads, defeating the performance advantage of parallel execution. The classical solution used by the parallelism community is to set up a *depth cutoff* (line 17), which above a certain depth switches back to sequential execution (line 20). The threshold value for switching, max_depth, is a parameter of the algorithm and depends on the pattern mining problem at hand and even sometimes of the dataset.

The advantages of our approach for writing parallel pattern mining using Melinda are the following:

- **Ease of use:** The synchronizations are handled by the TupleSpace implementation, the programmer doesn't have to care about them.
- **Data driven:** The tuples representing recursive calls, in pattern mining algorithms it means that they represent patterns, which are the intermediate data that can suffer from combinatorial explosion. This guarantees that the main complexity factor is the one that benefits the more from parallelism.

Algorithm 4: ParaMiner

```
Data: dataset D, selection criterion Select, closure operator Clo, depth cutoff threshold
      max_depth
Result: Output all closed patterns in D
1  begin
2      initTupleSpace()
3      foreach i ∈ 1..NUM_THREADS do
4          createThread(runThread())
5      ⊥_clo ← Clo(⊥, D)
6      output ⊥_clo
7      put((⊥_clo, D_⊥_clo, ∅, 0))

8  Function expand(P, D_P, EL, d)
9
       Data: Closed pattern P, reduced dataset D_P, extension list EL, current enumeration depth d
       Result: Output all closed patterns descending from P in the enumeration tree
10     begin
11         foreach e ∈ I s.t. e ∉ P and e ∈ D_P do      /* Augmentation enumeration */
12             if Select(P ∪ {e}, D_P) = true then                  /* Pattern test */
13                 Q ← Clo(P ∪ {e}, D_P)                  /* Closure computation */
14                 if Q ∩ EL = ∅ then                       /* First parent test */
15                     output Q
16                     D_Q = reduce(D_P, Q, EL)
17                     if d < max_depth then                      /* Depth cutoff */
18                         put((Q, D_Q, EL, d + 1))
19                     else
20                         expand(Q, D_Q, EL, d + 1)
21                     EL ← EL ∪ {e}

22 Function runThread()
23
24     while !isTupleSpaceEmpty() do
25         (P, D_P, EL, d) ← get()
26         expand(P, D_P, EL, d)
```

- **Efficiency:** It is easy to implement the TupleSpace in a way such that simultaneous put and get require little to no synchronization. The structure is simple enough to have low overheads, and can scale to large problems with many tuples involved.

- **Customizability:** The behavior of a parallel program using Melinda boils down to a simple scheduling problem: when a thread on a core executes get, what tuple will it receive ? The choice of some tuples can favor locality, while the choice of other tuples can help with load balance. This behavior can be tuned at will.

In order to understand better how Melinda's behavior can be customized, it is necessary to get the intuition of how we designed the TupleSpace. The TupleSpace is a set of **deques**, which are data structures well adapted for simultaneous insertion and extraction with little to no synchronization. We call each of these deques an *internal*.

- when put is called with a new tuple, it first has to choose an internal for insertion, or create a new one if necessary. This choice is delegated to a function called $distribute$, which takes as input the tuple and the id of the thread requesting insertion, and returns an internal id. Then put performs the actual insertion.

- when get is called, it has to choose a non empty internal for retrieval. This choice is delegated to a function called $retrieve$, which takes as input the id the thread requesting a tuple, and returns a non-empty internal id. Then get retrieves the tuple from that internal.

$distribute$ and $retrieve$ are relatively simple functions of Melinda which are exposed to the programmer willing to alter its behavior. Our default implementation of these functions is to have one internal per core, in order to favor locality as much as possible: new tuples produced on a core are likely to have associated reduced datasets residing in the cache of this core, so giving back these tuples to the thread working on this core will allow to exploit immediately the contents of this cache. Such locality problems have been shown to be of great importance for the performance of parallel pattern mining algorithms on multicores [ZPL97, GBP+05a, BPC06]. We thus focused on these problems, which is explained with more details in Chapters 3 (subsection 3.3.3) and 4 (subsection 4.3.3) of the PhD of Benjamin Négrevergne [Neg11].

Other implementations for $distribute$ and $retrieve$ are possible, for example by creating one internal per enumeration depth level in $distribute$, and by exploiting in $retrieve$ the heuristic that $expand$ is usually computed faster for deeper patterns to improve the scheduling.

We advocate that for pattern mining researchers, Melinda is a nice tool to explore scheduling strategies for pattern mining.

Thanks to the ease of use of Melinda, since 2007 we could propose 4 new parallel pattern mining algorithms, and could focus on algorithm and work-sharing concerns instead of low level synchronization issues. These algorithms are:

- PLCM [NTMU10]

- PGLCM [DLT10], [DTLN13]

- ParaMiner [NTRM13], [Neg11]

- PerMiner [Cue13]

C Breaking the memory wall for ParaMiner

Experiments of Tatikonda et al. [TP09] on *bandwidth pressure* from a pattern mining algorithm on a multicore architecture have been done in the extreme case of tree mining. In this case, CPU-costly *tree inclusion* tests have to be performed, and can be accelerated later by keeping large data structures called *occurrences lists* in memory. Thanks to ParaMiner's genericity, several different pattern mining problems that exhibit different CPU and memory usage behaviors can be evaluated, where the algorithm is fixed and only $Select$ and Clo change.

We conduced preliminary experiments with our C++ implementation of a preliminary parallel implementation of ParaMiner corresponding to the pseudo-code of Algorithm 4. The pattern mining problems tested were closed frequent itemset mining (FIM), closed frequent gradual itemset mining (GRI), and closed frequent relational graph mining according to the definitions in [YZH05] (CRG). The detail of the $Select$ and Clo functions for these three problems can be found in [NTRM13]. The depth cutoff threshold is set to a depth of 2, which gave the best results for these experiments.

The characteristics of the datasets used in the all the experiments of this chapter are given in the table below. Note that we report the size of the dataset as given in input to ParaMiner after encoding the problem as a set.

Problem: dataset name	$\|\| D_E \|\|$	$\mid E \mid$	Size (MB)
FIM : BMS-2 (sparse)	320,601	3,340	2.2
FIM : Accidents (dense)	11,500,870	468	34
GRI : I4408	50,985,072	4,408	194.5
CRG : Hughes-40	270,985	794	6.0

About these datasets:

- *BMS-2* and *Accident* are well known datasets from the FIMI repository [Goe04]. They are representatives of sparse and dense datasets respectively. We got similar results for each density class on other datasets of the FIMI repository, and only report results on these ones.
- *I4408* is a real dataset from bioinformatics containing gene expression values in the context of breast cancer [DLT10]. There are 4,408 genes tracked (4,408 attributes), and 100 experiments (100 lines) in the original dataset. Due to encoding as sets expressing variation directions of attributes between lines, the numbers of lines is squared in the input to ParaMiner [NTRM13].
- *Hughes-40* is also a dataset coming from microarray experiments. It represent a set of 1,000 potential gene interaction networks computed from real microarray data with a statistical algorithm presented in [IGM01].

All the experiments have been conduced on a large server with 4 Intel Xeon X7560 processors clocked at 2.27 GHz. These processors have each 24 MB of L3 cache, and 8 cores, for a total of 32 cores. The total RAM of the server is 64 GB, and the bandwidth of the bus is 9.7 GB/s. In our experiments the time is always wall-clock time measured with the Unix `time` command. Time thus include dataset loading and preprocessing (single threaded), thread creation and result output. Although it lowers our parallel performance due to Amdahl's law, we consider that it is fairest experimental setting, as it reflects the time that a practitioner will have to wait to get result, and the real speedup that parallelism will bring.

We present the **speedup** (execution time on n cores divided by execution time on 1 core) measure for the three problems FIS, GRI and CRG on our datasets in Figure 2.6.

This figure is especially interesting. It is the first time that with a fixed pattern mining algorithm, the parallel behavior of several pattern mining problems can be evaluated. It shows that:

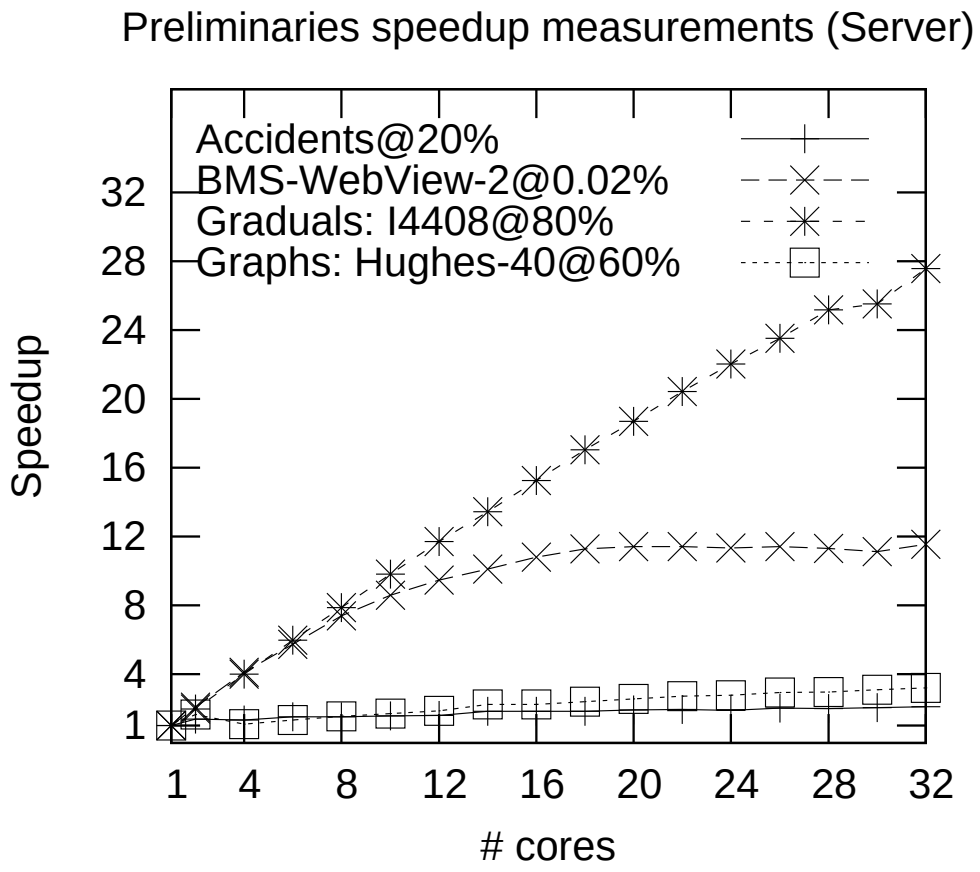


FIGURE 2.6: Speedup for first version of ParaMiner

- closed frequent gradual itemset mining (GRI) scales very well with the number of cores, reaching a final speedup of 28 out of 32 cores
- closed frequent itemset mining (FIM) has a mixed behavior depending on the dataset:
  - on the sparse dataset, it scales well up to 8 cores, then the scaling reaches a plateau with a final speedup of 12 out of 32 cores
  - on the dense dataset there is close to no parallel scaling, with a final speedup of 2 out of 32 cores
- closed frequent relational graph mining (CRG) also has very modest parallel scaling, with a final speedup of 3 out of 32 cores

In order to understand the wide discrepancy of parallel scaling between FIM on dense datasets and GRI, we studied bus bandwidth in both cases. For this, we made measurements with the *performance monitoring unit* (PMU) embedded in recent processors to understand lower level behavior of the processor. It is difficult to have a direct measurement of used bandwidth as there are no performance counters in the bus on our machine, but it can be easily deduced by indirect observation of some CPU counters. When a core makes a memory access, there are performance counters that measure how long this access takes to be completed. If the bus is saturated, all memory operations will be delayed, and thus the percentage of memory accesses performed with unusually large delays will increase.

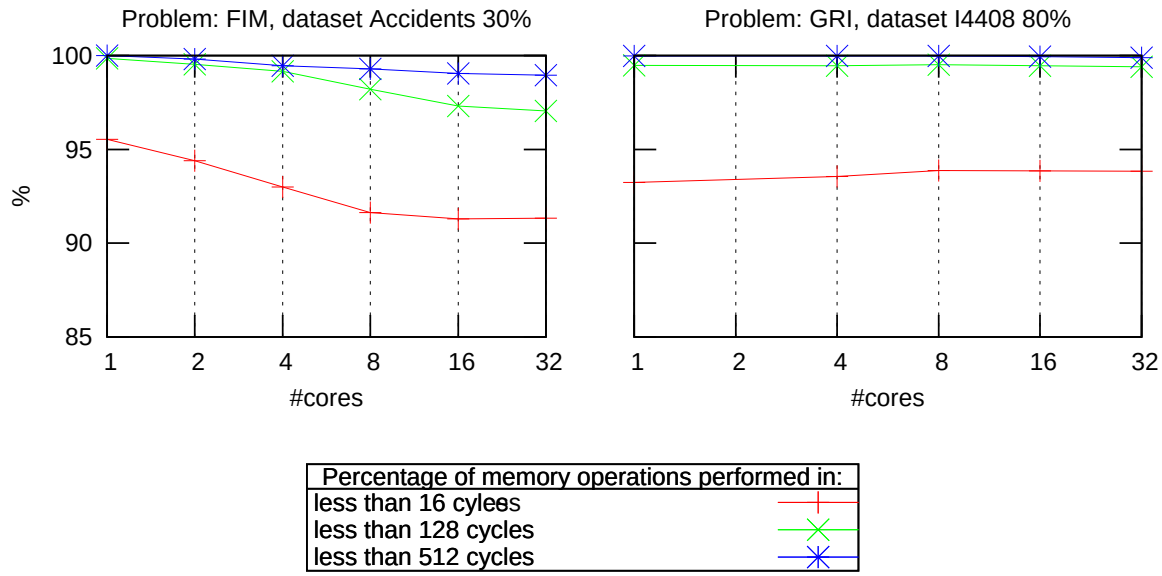


FIGURE 2.7: Memory transfer latencies

This is represented in Figure 2.7.

Usually on the computer used for experiments, a memory access is performed in 128 cycles if the data is not in the cache, less if the data is already in the cache. For the GRI problem, the proportions of different delays for memory accesses stays constant with the number of cores: the bus is always equally loaded. Around 1% of the accesses are out of the cache and take more than 128 cycles, which can be easily understood from the large size of the input dataset: most of the times the dataset reduction works well and allows the data to stay in cache, but a few times when the algorithm backtracks in the enumeration it has to fall back on larger reduced datasets or even the original dataset, which are likely to have been evicted from cache. This situation is normal and confirms the good speedup exhibited by GRI in Figure 2.6. On the other hand, for FIM on *Accidents* the latency of memory accesses quickly increases when increasing the number of cores. Even with as few as 2 cores, extremely long memory accesses with latency higher than 512 cycles (4 times normal access latency) appear, and make more than 1% of all accesses for 32 cores. This is symptomatic of heavy bus congestion, and explains the lack of speedup shown in Figure 2.6.

Dataset reduction by itself is an expression of the "memory over CPU" tradeoff: all the transactions containing a pattern are stored together as a reduced dataset, in order to avoid costly rescans of the complete dataset by $Select$ and Clo. Often, the reduction in following computations is so tremendous that dataset reduction is unavoidable in modern pattern mining algorithms. However, it is also very costly to perform it, both for CPU, memory usage (RAM or cache), and bandwidth usage. We are thus confronted with an **amortized analysis** problem, which can be expressed intuitively by: *Is the gain in performances of a dataset reduction worth the resources paid doing it ?*

Answering this question theoretically is difficult, as the gain are to be computed on a subtree of computation which is unknown a priori, and there are complex interactions between the different resources (CPU, cache/RAM, bus bandwidth). Although I am interested in such theoretical approach, I consider it as an open perspective that requires teaming up with a specialist of theoretical amortized analysis.

However, it is possible to understand the amortizing behavior of dataset reductions in ParaMiner by using an experimental approach. Our approach follows the following steps:

1. For each dataset reduction performed by ParaMiner in its enumeration tree, identified by its input

pattern, a *reduction factor* is computed, i.e. how much the dataset reduction shrinks its input dataset.

2. The dataset reductions are sorted in decreasing order of reduction factor. The hypothesis being that the dataset reductions having the highest reduction factors are the most important for gaining performance.

3. ParaMiner is executed by activating only the top $x\%$ of dataset reductions having the highest reduction factors. During this execution the number of memory accesses per 1000 instructions (excluding cache hits) is measured, it corresponds to the requested bus bandwidth.

Figure 2.8 show the results of this experiment for ParaMiner on FIM problem and *Accidents* dataset.

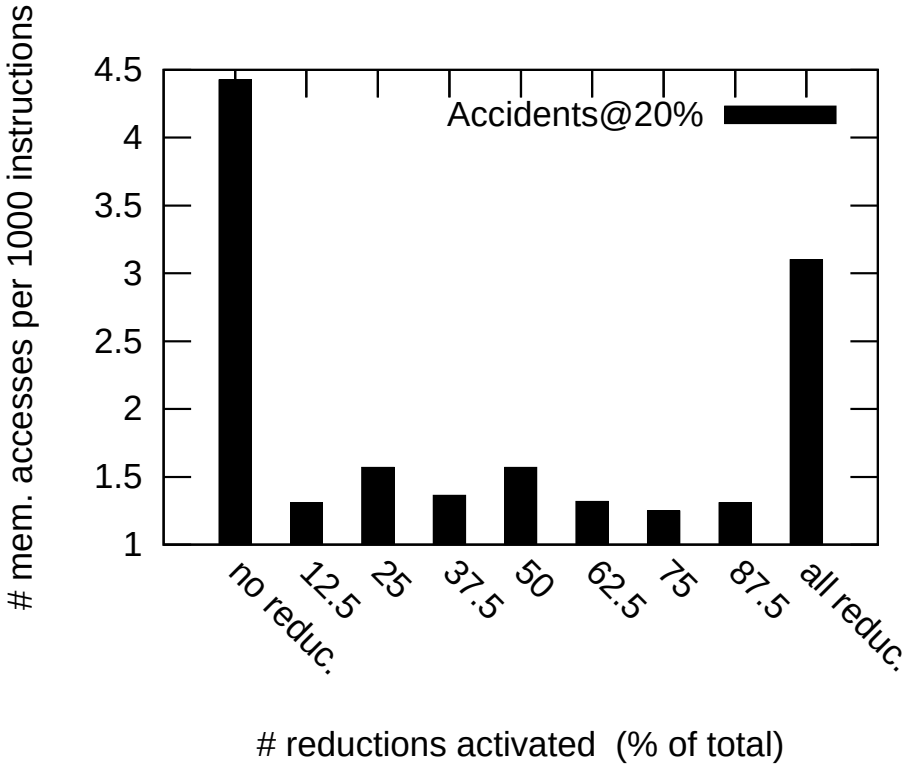


FIGURE 2.8: Bandwidth impact of top-$x\%$ datasets reductions

There are three parts of this histograms:

- When 100 % of dataset reductions are performed (usual case of all pattern mining algorithms), ParaMiner does too many memory accesses, indicating congestion and ultimately bad parallel speedup. This corresponds to the results of Figures 2.6 and 2.7.

- When too few dataset reductions are done (less than 12.5 %), then the datasets are not reduced enough and the algorithm has to scan through gigantic datasets that mostly don't fit in cache. This leads to bad usage of the machine resources, and ultimately long execution times.

- In all the other cases, the number of memory accesses is lower, meaning that the system bus is less stressed, which ultimately leads to better parallel speedups and possibly to better execution times.

We confirmed that the same behavior can be observed on other problems and other datasets, with *Accidents* reported here as the most extreme case. These results clearly show that **systematically doing dataset reduction is harmful for parallel pattern mining performance.**

However, doing too few dataset reductions is also detrimental for performance. Dataset reduction must be kept, but driven by a well designed criterion that decides when it is necessary and when it can be skipped.

We proposed such criterion in [NTRM13]. Intuitively, this criterion estimates a maximal reduction factor that can be achieved by the dataset reduction of ParaMiner, and doesn't perform dataset reductions that can't reach an high enough reduction factor (given by a threshold). The interested reader is invited to read the paper for more details on this criterion.

Using the modified dataset reduction proposed, with a criterion set to only keep reductions that divide at least by 2 the size of their input dataset, we repeated the speedup experiment presented at the beginning of this section. The new results are presented in Figure 2.9.

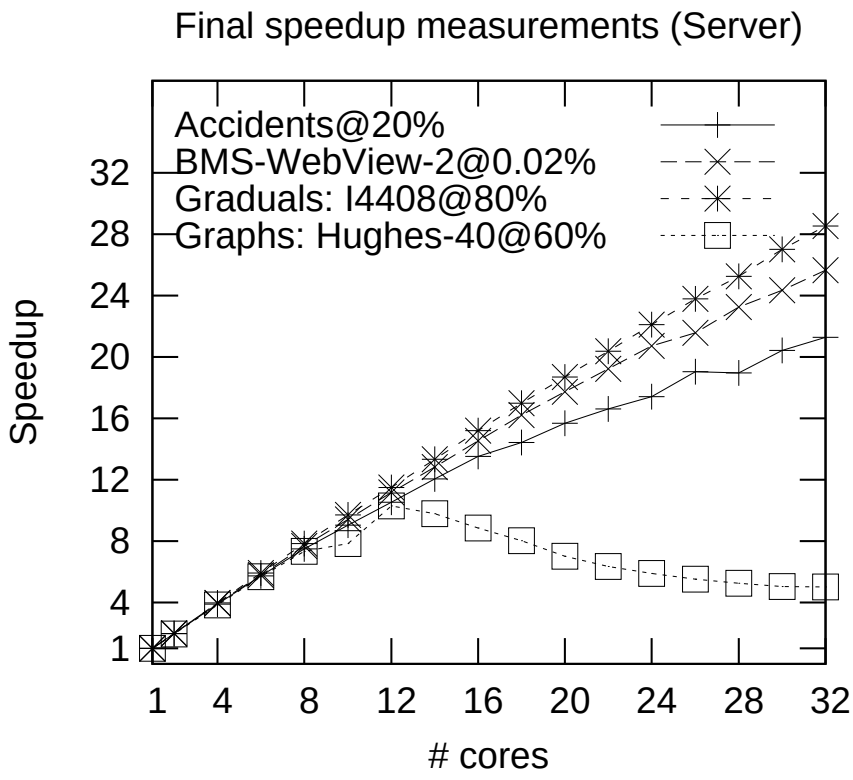


FIGURE 2.9: Speedup for final version of ParaMiner

The results change drastically for the FIM problem:

- for the sparse dataset *BMS-WebView-2*, there is no longer a plateau, and a final speedup of 25 out of 32 cores is reached.
- for the dense dataset *Accidents*, there is a ten times increase in speedup, with a final speedup of 21 out of 32 cores.

For the CRG problem, there is an excellent scalability up to 12 cores, but it degrades after. The reason is that unfortunately on the dataset used for CRG the dataset reduction has difficulties to get good

reduction factors, and can't reduce enough the data. Usually when a dataset is well reduced, datasets close to the leaves of the enumeration tree are very small and hold in L2 or even L1 cache, leading to very quick $Select$ and Clo operations. Such case cannot be reached with the CRG problem and *Hughes-40* dataset, so at 12 cores the bandwidth gets saturated and adding more cores worsens the situation.

Last, the performances stay very good for the GRI problem.

These results confirm the interest of our approach, and the efficiency of our generic criterion for selective dataset reduction activation. This criterion, like the rest of ParaMiner, is generic, so any pattern mining problem that can be handled by ParaMiner immediately benefit from it.

D ParaMiner vs specialized parallel algorithms

To conclude on ParaMiner, we compare its mining performance with state of the art algorithm for the FIM, GRI and CRG problems.

For the FIM problem, we compared our C++ implementation of ParaMiner with our C++ implementation of PLCM [NTMU10], which is specialized for frequent itemset mining and can thus take advantage of a better first parent test and a more aggressive dataset reduction. We also compared it with the C++ implementation of MT-Closed, a parallel closed frequent itemset miner proposed by Lucchese et al. [LOP07]. MT-Closed is better adapted to dense datasets, thanks to the use of bitmap representations and SIMD instructions. All algorithms were executed using 32 cores, the results are presented in Figure 2.10.

For the sparse dataset, ParaMiner's run time is worse than PLCM but slightly better than MT-Closed, which is one order of magnitude slower than PLCM. For the dense dataset, ParaMiner is between one and two orders of magnitude slower than MT-Closed and PLCM.

These results show that despite its genericity, ParaMiner can have reasonable run times on the very competitive problem of closed frequent itemset mining.

For the CRG problem, first defined in [YZH05], there are no available implementation of algorithms solving this problem. We thus compared ParaMiner with gSpan [YH02] and Gaston [NK05], two state of the art general graph mining algorithms. The comparison is clearly unfair because the CRG problem is simpler than general graph mining. However it reflects the choice of algorithms that a practitioner interested in the CRG problem would have. ParaMiner is run on only 1 core, because gSpan and Gaston are not parallel algorithms. The results are shown on Figure 2.11 (a).

It clearly appears that ParaMiner is much more efficient for the CRG problem than gSpan and Gaston, being two orders of magnitude faster for the lower support values. It demonstrates the interest of having a generic algorithm that can be easily adapted to non-standard problems.

Last, for the GRI problem we compared ParaMiner with our C++ implementation of PGLCM presented in Chapter 1. Both algorithms are executed with 32 cores, the results are presented in Figure 2.11 (b).

Here, ParaMiner is two orders of magnitude faster than PGLCM, that was the previous state of the art for the GRI problem. The problem is that PGLCM does not have a dataset reduction, and such a dataset reduction is difficult to design when the dataset is a matrix of numerical values. On the other hand, through set encoding ParaMiner requires a much bigger initial input matrix, but reduces it effectively, leading to vastly improved execution time.

This is one of the key advantages of ParaMiner: any recent or unusual problem, such as CRG or GRI, can immediately benefit from a state of the art enumeration and a state of the art dataset reduction, allowing to start with efficient implementations that can process real world data. This is convenient for practitioners, which need such performance. It also allows pattern mining researchers to start from a strong baseline: works for improving this baseline are likely to be significant for the community, as were

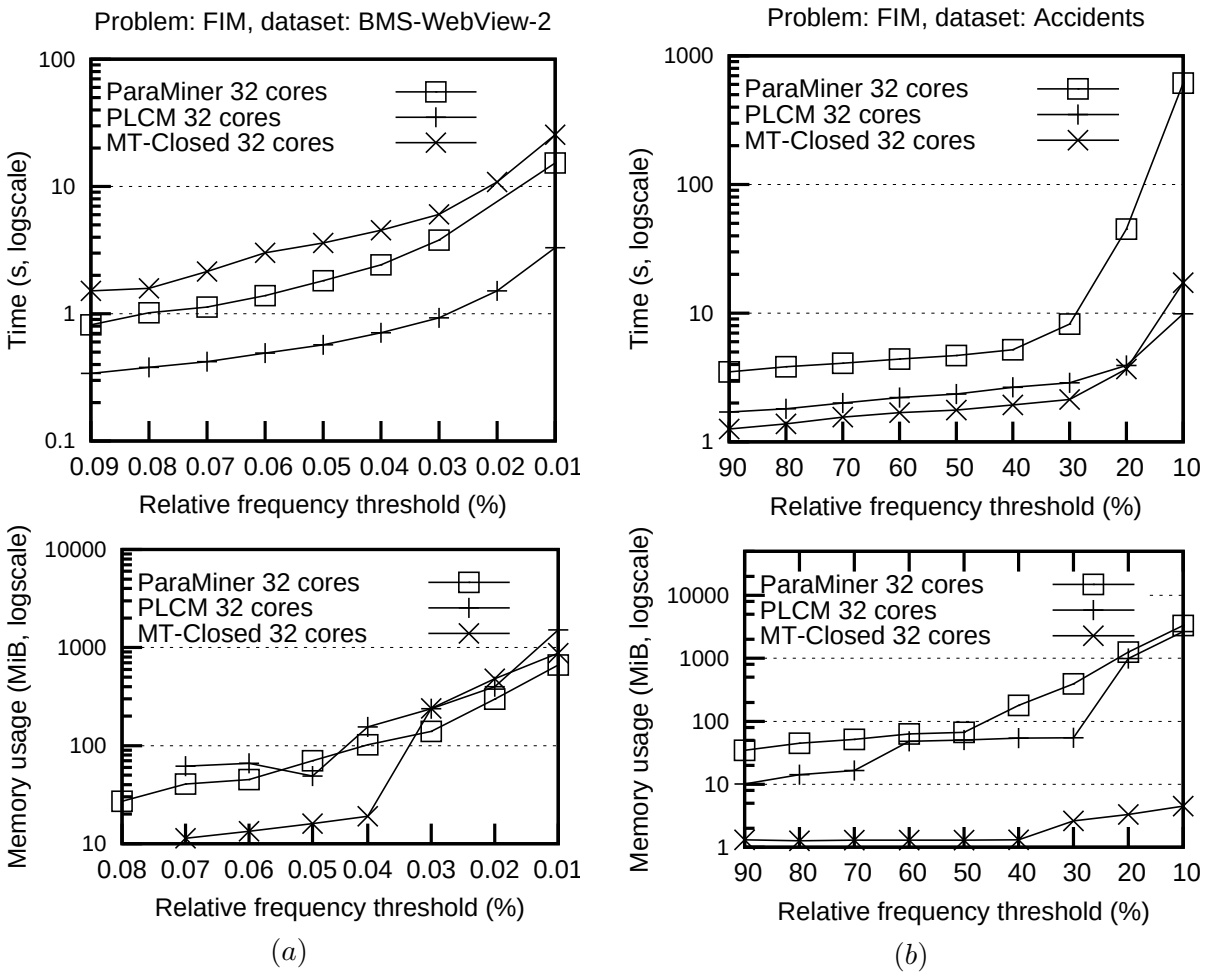


FIGURE 2.10: Comparative experiments for FIM, on BMS-WebView-2: (a) and Accidents: (b). Runtimes (top) and memory usage (bottom).

first works to improve Apriori in the late 90's.

E Perspectives for "Mine faster"

Up to now in the pattern mining community, parallelism has too often been seen as an implementation trick, and getting one or two orders of magnitude through better theory and algorithms on a single core has been the most sought after contributions in the domain. However future processors are evolving from multicore to manycore (see for example recent products from Tilera [Til13], Kalray [Kal13] and Intel Xeon Phi), with hundreds of cores. Exhibiting good parallel scaling on such processors means two orders of magnitude faster algorithms than classical sequential algorithms. Sooner or later, the majority of pattern mining (and more generally data mining) researchers will realize that however nice the algorithms and theory they produce, it will have to fit the parallel computing model. This is what the community once did in the monocore era, when people realized that efficient handling of the memory hierarchy was key for good performances.

For my part, I find this research topic very exciting. The works described in this chapter are a first step, and there are many perspectives that I am interested in:

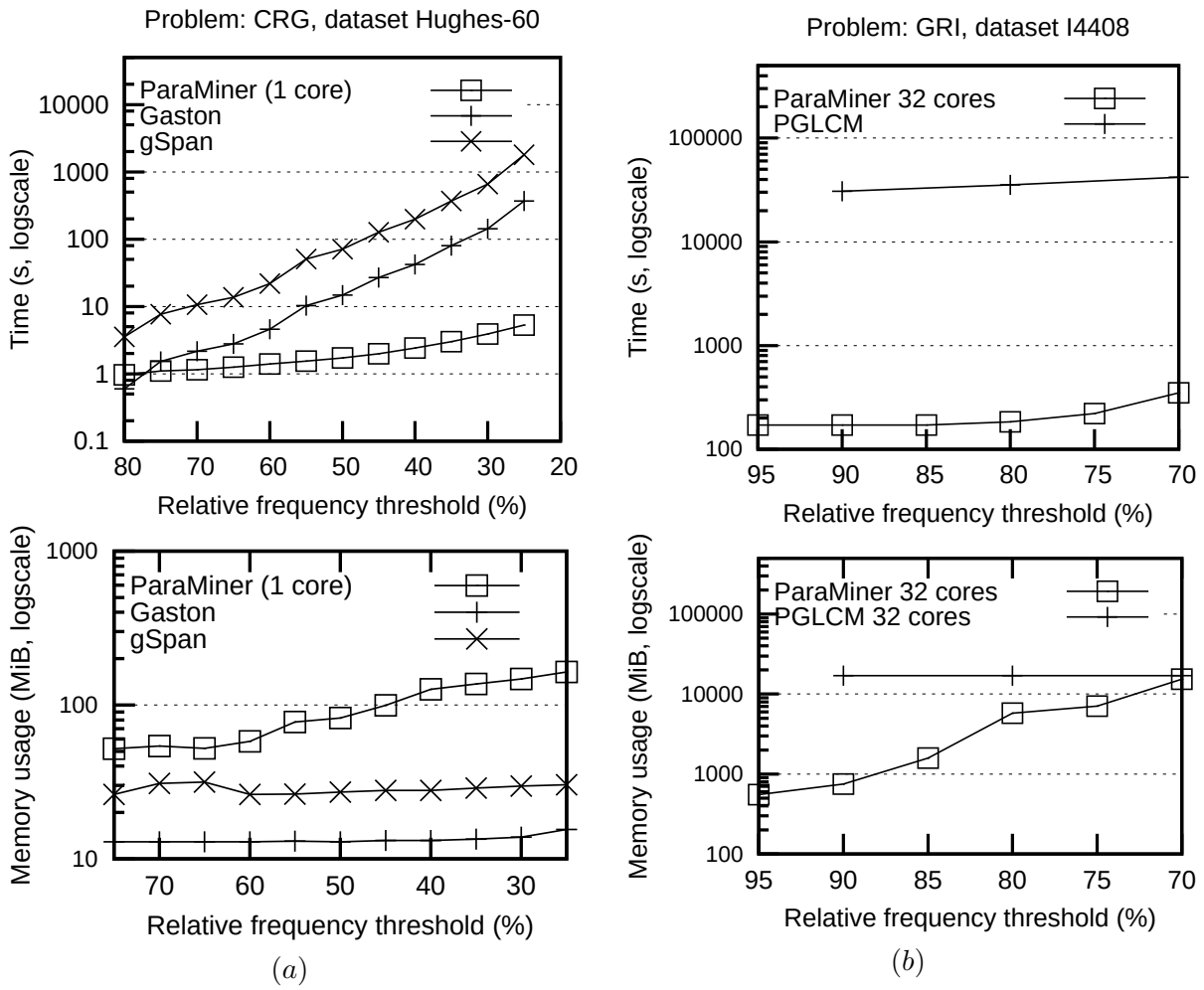


FIGURE 2.11: Comparative experiments for CRG on Hughes-40 and GRI on I4408. Runtimes (top) and memory usage (bottom).

- A perspective linked with of pattern mining DSL of the previous chapter is to provide some control over parallelism in the DSL. This would allow to have extremely fine granularity parallel tasks in the $Select$. Such fine grained decomposition would help for complex $Select$ predicates and cases where the datasets can't be reduced enough, such as what happened in our experiments with CRG.
- Another interesting perspective previously evoked in this chapter would be to make a theoretical amortized analysis of dataset reduction in ParaMiner. This would help to find a better criterion for determining if a dataset reduction has to be performed or not, with strong complexity guarantees.
- This work on dataset reduction is strongly correlated with cache usage. There is a fascinating area of research about **cache aware** [SCD02] and **cache oblivious** [FLPR99, BGS10] algorithms.
  - cache aware algorithms take the cache hierarchy as a parameter, and provide theoretical upper bounds on number of cache misses, effectively increasing performances
  - cache oblivious algorithms don't need to know the cache size, and guarantee that (asymptotically) the data for the computation will fit into the cache.

ParaMiner is tending towards cache obliviousness: as the dataset is progressively reduced, it is

likely to fit into the cache at some point. However there are no strong theoretical guarantees of it, and many counter examples of datasets that can't be reduced well can be found. It would be interesting to turn ParaMiner into a real cache oblivious algorithm. Two difficulties arise: the first one is that the $Select$ predicate is not known, and may use extra memory. This could be controlled by adding some reasonable hypothesis on memory usage by both parts of $Select$ (which, we recall, has to be decomposable). The other difficulty is that currently only relatively simple and regular problems are known to have a cache oblivious algorithm, mostly on basic matrix operations. A problem such as (generic) pattern mining is much more complex and would represent a significant improvement in the study of cache oblivious algorithms.

- There are more and more data which are produced today as high throughput streams, with needs for real-time analysis. Such real-time needs require highly efficient parallel algorithms, that are possibly simpler in their results that classical pattern mining algorithms, but that are able to take advantage of manycores or GPU architectures. In the starting PhD of Serge Emteu, co-advised with Jean-François Méhaut (University of Grenoble) and Miguel Santana (STMicroelectronics), we are working on such algorithms on a GPU target. The goal being to mine trace output of applications running on prototype embedded systems developed by STMicroelectronics, in order to quickly detect defects and accelerate the debugging process.

- A longer term perspective is to work on the recent trend of **co-design** between architectures and algorithms [KVBH11]. The HPC (High Performance Computing) community, currently working on exascale clusters (i.e. delivering 1 ExaFlops = 10^{18} Flops) that should become available around 2020, is facing several problems. Due to the massive parallelism (billions of cores) and comparatively modest bandwidth of these machines, it will be difficult to have good performances on all types of applications. However, these clusters will be extremely costly to buy, and also to power: for the scientific applications they will run, they must deliver performance significant enough to justify that cost. The idea is thus to design simultaneously the algorithms and the architectures (hence the term *co-design*) in order to make architectures choices best adapted to the applications of interest, and to take this architecture into account efficiently in the applications code. Pattern mining being an especially demanding application, it would benefit from such effort, even at a scale more modest than exascale clusters.

Last, this chapter on parallelism would not be complete without a mention to Big Data. There are nowadays many works on analyzing large scale datasets on clusters using the MapReduce paradigm [DG04]. This is interesting in several ways for me:

- there exists datasets and pattern mining problems that are too large to be handled by a single machine. Finding efficient solutions to handle them and produce patterns is a real challenge.

- Even on multicores/manycores, the tendency is to limit as much as possible shared memory and cache coherency, which are complex to implement physically, lead to inefficiencies and also to fatal bugs for programmers. Models coming from the cluster word (message passing) and functional programming (Map Reduce) are seen as the way to make better manycore chips and program them more easily. Thus working on MapReduce today is also a good way to gain experience for working on future manycore processors.

In the internship and soon to start PhD of Martin Kirchgessner, co-advised with Vincent Leroy and Sihem Amer-Yahia (both at University of Grenoble), we are working on adapting LCM for MapReduce. This problem is more difficult than previous work such as PFP [LWZ+08] which adapted FPGrowth to MapReduce, because of the closure which prevents some known efficient decompositions of the dataset on several machines.

Contents

A Mining execution traces from Multi-Processor System-on-Chip
1 Mining contention patterns in extremely large execution traces
2 The SoCTrace project
B Use domain knowledge to enable scalable exploration of large user datasets
C Reducing the number of patterns

CHAPTER 3

MINE BETTER: *Less is more*

The two previous chapters have shown my work on how to extend the reach of pattern mining, and how to make it faster thanks to multicore architectures. However once a pattern mining algorithm finishes, its user is left with a list of patterns, possibly with millions of entries. Finding the most interesting patterns in this list is extremely time consuming, and can be seen as a *second order* data mining problem.

There is increasingly more research on how to either get only the most interesting patterns, or make the output of pattern mining algorithms more manageable. This corresponds to the following areas:

- control the part of the search space of pattern mining algorithms (constraint pattern mining [SVA97, SC05, BJ10, BL07]).
- reduce in an understandable way the output of pattern mining algorithms (top-k pattern mining [HWLT02], k-pattern set mining [GNR13])
- discover few patterns that answer to very specific questions (emerging and discriminative pattern mining [DB13])
- provide visualizations over the set of patterns output [KSS05, CL10]

In the rich industrial and academic environment of Grenoble, I am currently confronted with challenging applications, with collaborators or practitioners that have high expectations from data mining techniques. This chapter presents my latest works, focused on how to make pattern mining more useful for these applications. Contrarily to the previous chapters, it presents research which is ongoing or even just starting, giving a quick outlook at my short and mid-term perspectives. It is organized in three sections:

- Mining execution traces from **Multi-Processor System-on-Chip (MPSoC)**
- Mining patterns social communities in **user data**
- Mining fewer patterns from **matrix factorisations**

A Mining execution traces from Multi-Processor System-on-Chip

Since the 60's, Grenoble is one of the strategic places in France for the development of its semi-conductors industry. Today, it hosts an important part of the STMicroelectronics company (including 2 Fabs as well as R&D offices), which is world's 5th provider of semi-conductors.

Semi-conductor companies such as STMicroelectronics are heavily investing on **Multi-Processor System-on-Chip (MPSoC)**. These are highly integrated electronic packages integrating several computation cores, memories, specialized processing units and I/O components. They are both cheap and powerful, and thus drive most of existing embedded systems, from our smartphones and tablets to car and plane computers. However, due to their heavy parallelism, they are extremely difficult to debug with today's tools. The solution is to collect large execution traces of applications running on MPSoC, and analyze them *post-mortem* in order to uncover bugs or bad performance causes. This is a complex data analysis problem, and thus an exciting playground for data miners.

I have active collaborations both with the TIMA lab (Grenoble public research unit on the future of MPSoC), and with STMicroelectronics. In both cases our goal is to provide tools that can be of actual use to developers of applications on MPSoC, which exclude the traditional "skim-this-million-pattern-list" approach. We have started working on different approaches on how to make pattern mining more useful for these application developers. I present these approaches and some preliminary results below.

1 Mining contention patterns in extremely large execution traces

The first approach that I want to present is also the simplest, and consists in outputting as meaningful and understandable patterns as possible **while using legacy pattern mining algorithms**, in a precise application context.

This is the goal of the PhD of Sofiane Lagraa, that started in december 2010 and that will end in early 2014, co-supervised with Frédéric Pétrot (TIMA lab, University of Grenoble). The team of F. Pétrot (SLS team, TIMA lab) is working on the future generations of Multi-Processor System-on-Chip (MPSoC), which are the chips that will power set-top boxes and smartphones in a few years. Such chips have multiple processing cores and memories (16 - 32 cores is common), and performances issues of applications come as well from bad parallel programming that from complex interactions between the application, the system and the hardware, making such issues very difficult to detect and reproduce. The solution is to use **MPSoC simulators**, that simulate the complete execution of the application on a MPSoC and output a trace of this execution, and to analyze this trace.

Such simulations output large amount of traces (ex: 10 seconds of video decoding leads to 24 GB of trace *per core*), making the analysis of such traces an application area of choice for data mining methods. Despite such needs in data mining, there are still few works in this area, making the PhD of Sofiane Lagraa interesting for TIMA and LIG labs to see how useful data mining, and especially pattern mining methods, can be for analyzing large execution traces.

The most usual reason for sub-optimal performance in a MPSoC is **contention**, i.e. when too many cores are requesting the same resource simultaneously, typically a memory. The memory being only able to serve a given number of accesses per cycles, the other accesses are put in a waiting queue, delaying further operations by requesting CPUs. This problem is similar to what we saw in Chapter 2, except that in MPSoC there can be several memories and the bus topology is more complex.

In the traces, contention can be observed through an increase of the *latency*, which is a metric indicating for each memory access how many cycles it took to complete. For example consider Figure 3.1.

It corresponds to a simulated execution of a Motion-JPEG (MJPEG) decoder on a MPSoC with 4 cores. For each core, all memory accesses are plotted, with the time stamp of the access in X-axis and

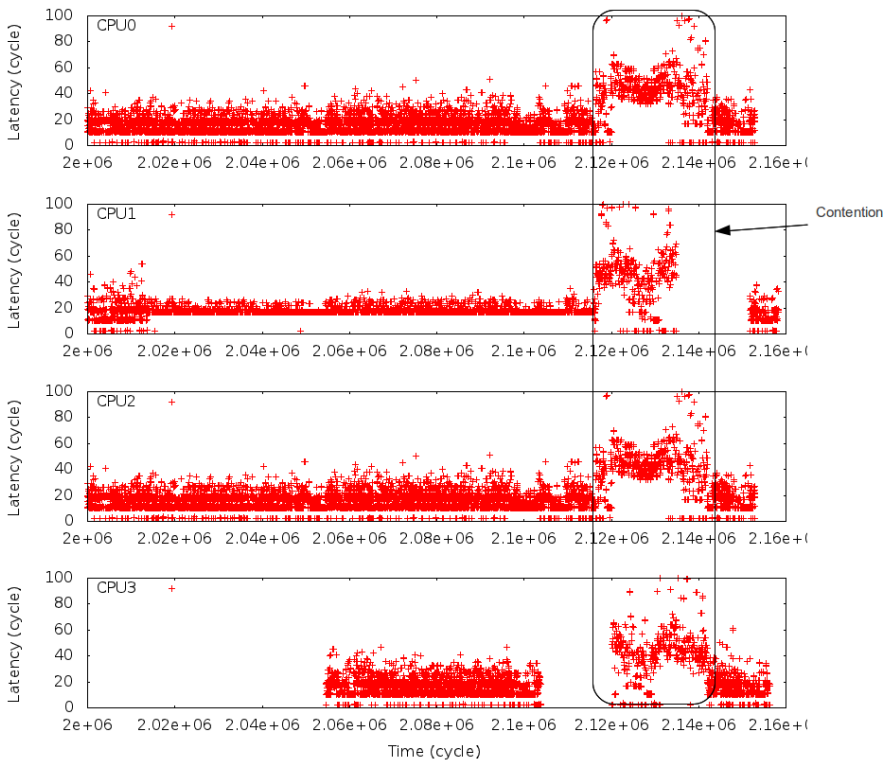


FIGURE 3.1: An occurrence of contention in an execution trace

its latency in Y-axis. Towards the end of the execution all cores exhibit a simultaneous peak of latency: it is symptomatic of contention.

Contention being a co-occurrence of several accesses to the same physical memory location, it can be easily detected through frequent itemset mining, whose goal is precisely to detect co-occurrences. Existing algorithms can be exploited, as long as a necessary **preprocessing** of data is performed.

The method we proposed in [LTP13] relies on the preprocessing steps explained below:

- All latency values are analyzed in order to determine latency values significantly above than average. This is performed with a simple statistical method: high latencies values are the upper quartile of the latency values distribution.
- For each event having a high latency, a *transaction* is constructed. This transaction contains all events in a window of ω cycles around the high latency event, accross all CPUs (ω being a parameter).
- The *items* in the transactions are the events:
  - when the event is a data access the precise address becomes an item
  - when the even is an instruction access the precise address as well as the function containing the instruction are turned into items, in order to have both fine and coarse granularity information in the transaction.

  For each event, its discretized latency as well as CPU are also added to the items in order to have patterns as informative as possible.

A standard closed frequent itemset mining algorithm is then run on these transactions in order to find sets of events co-occurring often while contention arises.

We present in the table below two patterns found according to our method for a MJPEG video decoding application running on simulated MPSoC with 4 and 8 cores:

Platform	Contention pattern	Support
4 cores	CPU[0,3] [0x10009ee4, 0x10009f78] idct [0x10016b50, 0x10016f2c] memcpy lat_10_20 lat_20_30	72 %
8 cores	CPU[0,7] [0x10009b10, 0x1000a224] idct [0x10016ab0, 0x10016e8c] memcpy lat_10_20 lat_20_30	88 %

These patterns show that there is a contention problem between the idct and memcpy functions in both platforms. The addresses colored respectively in red and blue give the precise addresses of the loops of idct and memcpy for which the contention occurs. The patterns tell us that this contention affects all CPUs of the considered platform (set of all CPUs represented by items `CPU[0,3]` and `CPU[0,7]`). Given that the average latency is around 10 cycles in both platforms (represented by item `lat_10_20`, that still appears in the pattern due to some normal accesses), the patterns also inform us that the contention doubles or triples the latency of impacted accesses (presence of item `lat_20_30`).

The use of such pattern for a developer of the MJPEG decoder is to indicate him that the main function of the MJPEG application, `idct`, which performs inverse discrete cosine transformation, is negatively impacted by too many memory accesses, that the architecture can't handle well. This can help him to rethink the way the application is written, in order to either reduce or schedule better the memory accesses. A more elaborate discussion can be found at the end of our DATE paper [LTP13].

Admittedly the method we proposed is simple from a pattern mining point of view. It has the advantage to quickly give results which are understandable by an expert, and which can help to improve the application.

It's cons are that there are still many patterns returned, and a pattern miner must be heavily involved to design the preprocessing in order to guarantee having interesting and interpretable results.

We are thus working on a more elaborate method which crosses several executions of the same program on platforms having different number of cores. Our goal is to identify the sets of instructions whose performance behavior changes the most when the number of cores increases, in the sense of loss of scalability. We submitted our preliminary results in this direction to the EMSoft'13 conference.

2 The SoCTrace project

The approach above presents solutions for one of the problems of MPSoC execution trace analysis. However, there are many other kinds of problems, and for debugging or optimization application developers may be interested in very peculiar characteristics of their traces.

This is the goal of the SoCTrace project to improve the set of tools available for trace analysis. This large project sponsored by French Ministry of Industry (FUI project) is lead by STMicroelectronics, its other members being University of Grenoble (LIG and TIMA labs), INRIA (French National Institute for Informatics and Automatics), and ProbaYes and Magilem companies. The project aims to provide new tools for storing, querying, analyzing and visualizing large execution traces coming from STMicroelectronics MPSoCs. I am the responsible of the "University of Grenoble" partner in this project, which involves 5 permanent researchers from the university, and will fund 3 PhDs and 2 engineers during the 4 years of the project.

In this project, pattern mining has an important role to play to simplify how developers interact with the traces. Today, when developpers use traces to understand the behavior of their application,

they usually rely on visualization techniques that represent all the events of the trace *one by one*. Such visualization is often based on Gantt charts, as shown in Figure 3.2

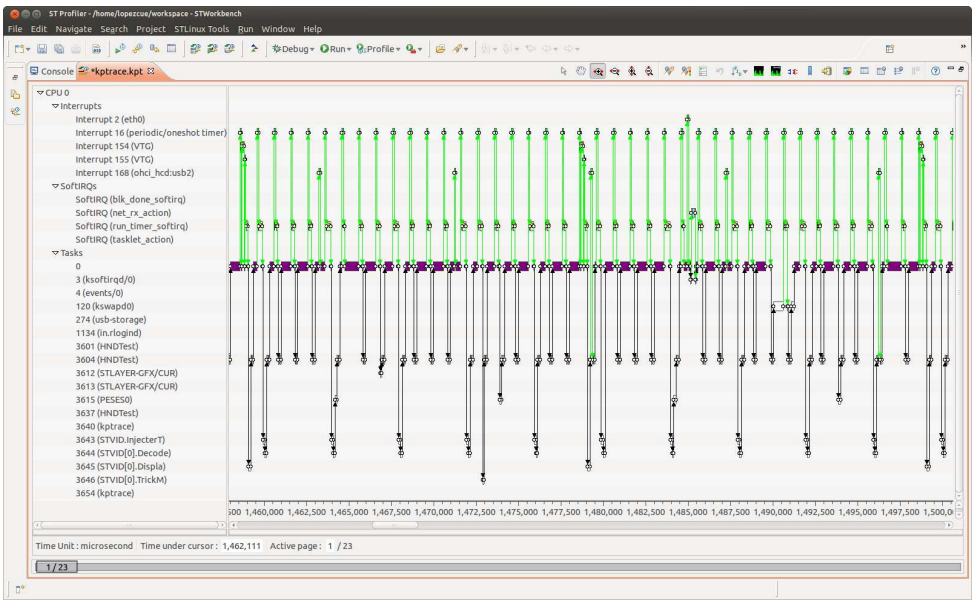


FIGURE 3.2: Classic Gantt chart visualization of a trace portion (KPTrace tool [STM])

On the figure, an extremely short portion of a trace of video decoding is shown. One can immediately note the complexity of this visualization, which makes very difficult to have a quick trace overview.

Simplifying trace representation: One approach that we propose is to exploit pattern mining in order to extract regularities in the trace in order to simplify the trace representation. Especially, many of the applications that STMicroelectronics is interested in are multimedia applications, which revolve around the processing of audio/video *frames*. From frame to frame the computations performed are roughly the same: frequent sequential patterns inside frames can be detected and exploited to abstract the events they cover in a simplified rewriting of the trace. An example of such rewriting is shown in Figure 3.3: on the left part, a fictional trace is shown, with frames materialized and three sequential patterns occurring in both frames. On the right part, a rewriting of the trace with these patterns is shown: it contains much less elements, reducing the number of information a developer has to handle to understand the trace.

Our goal is not to find all sequential patterns existing in frames, but a given number of patterns (user parameter k) that allow an optimal coverage of the complete trace. Choosing a low number of patterns allow the developer to quickly grasp the main repeating steps of the applications, and also to see where these steps are not respected.

For example consider Figure 3.4 (a), which is a rewriting of an actual audio/video decoding trace produced with GStreamer open source framework [GSt13], with the patterns we found ($k = 10$).

Each line corresponds to a frame, and each pattern is represented by a colored block whose length is its number of events. Black blocks corresponds to areas of the frame not matched by any of the 10 patterns we found. The developer can quickly identify the different steps of frame decoding captured by patterns: the gray or yellow patterns at the beginning of most frames correspond to initialization, the orange pattern at the middle corresponds to actual video decoding, and the pink pattern at the end correspond to end of decoding and finalization. The developer can also notice that irregularities come mostly at beginnings of frames. Zooming on the fourth frame in Figure 3.4 (b), one can see to the precise events inside the patterns and can spot in the unmatched region (in black) a rare call to

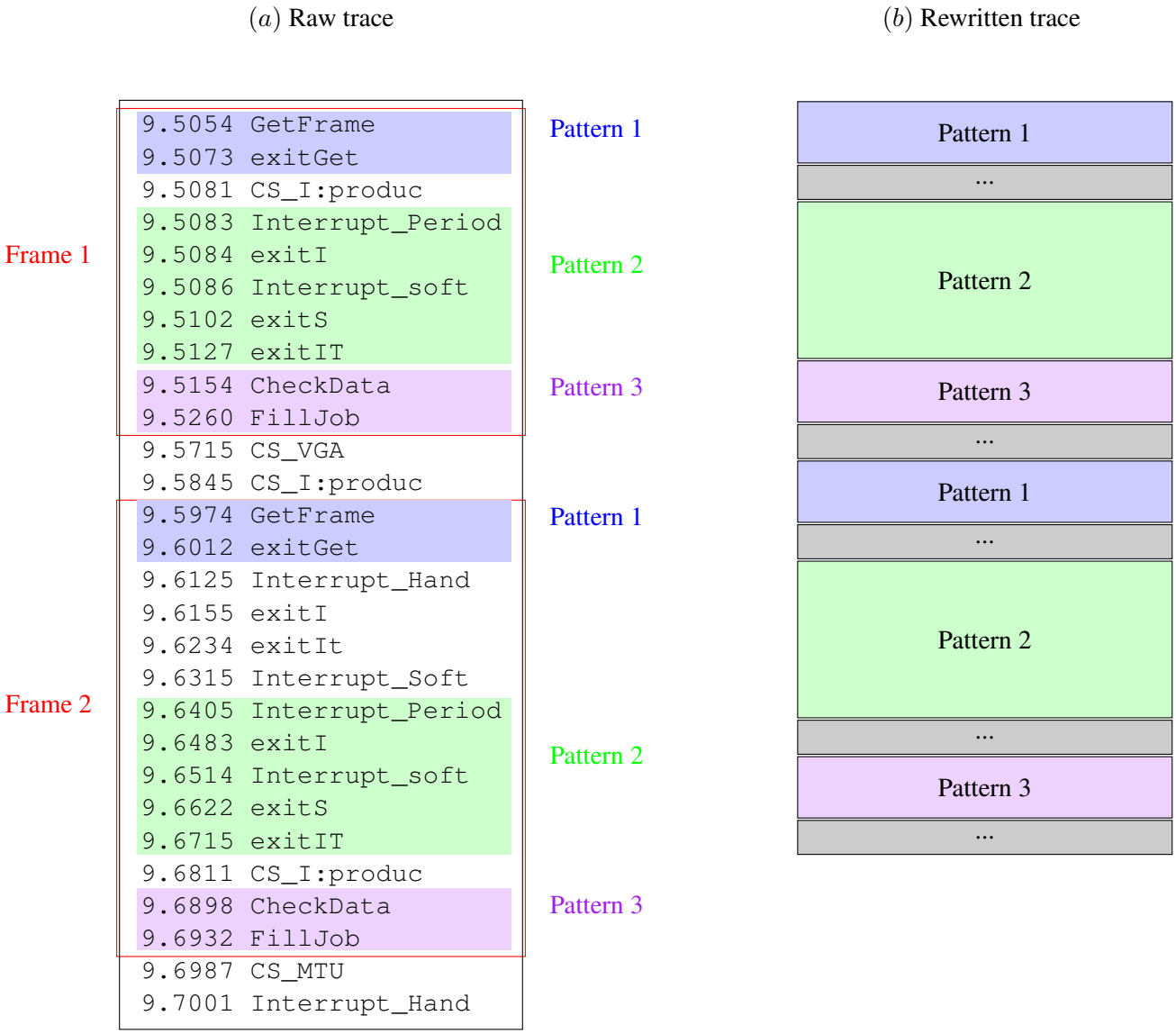


FIGURE 3.3: Example of frame rewriting with patterns

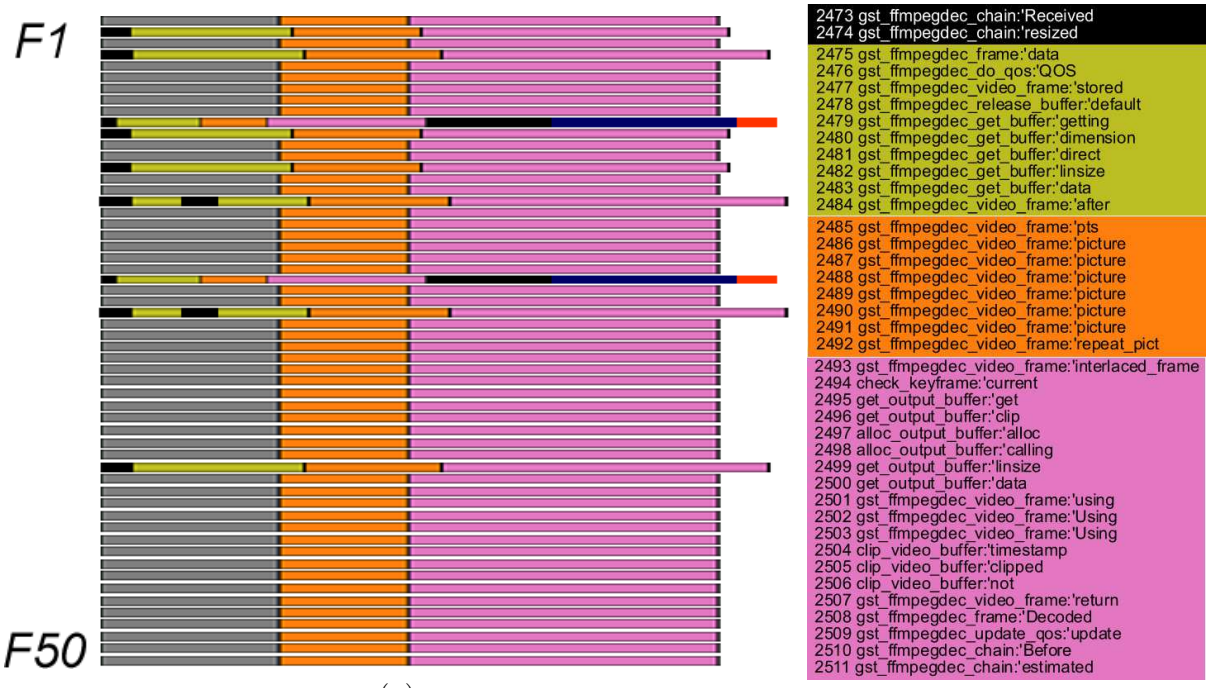


FIGURE 3.4: Trace rewritings: (a) set of 50 frames, (b) zoom on fourth frame

`gst_ffmpegdec_chain:'resized`, which asks for buffer resizing. Usually buffers are properly dimensioned but here a buffer wasn't, requiring this resizing call. Memory operations being critical for performance, the developer is likely to investigate further on the reason of this buffer resizing.

From a more theoretical point of view, we have shown in a recent submission to the KDD'13 conference that this problem was an instance of the *packing problem* [Ege08], and NP-hard problem for which no general solution is known. In a previous paper [KFI+12], we presented a simple solution to this problem based on a basic greedy algorithm. In a recent submission to the KDD conference we present several more elaborated solutions based on algorithms mixing pattern mining steps and greedy steps that improve processing time by up to two orders of magnitude.

In this work I collaborate with:

- the PhD student Christiane Kamdem-Kengne, who is supervised by Noha Ibrahim and Marie-Christine Rousset (both at the University of Grenoble) and financed by the SoCTrace project
- Takashi Washio from Osaka University, who I know since my postdoc in 2004-2005, and who brought us his expertise in greedy algorithms.

Exploiting domain knowledge for trace analysis: A problem of the previous approach is that the patterns found to rewrite the trace are not associated any semantics. The user has to manually inspect the events constituting the patterns in order to determine their semantics.

In the PhD of Léon-Constantin Fopa, financed by the SoCTrace project and that I co-supervise with Fabrice Jouanot and Jean-François Méhaut (both at the University of Grenoble), we are studying a method to automatically associate semantics to frequent patterns found in the trace, and to use this information to provide higher quality rewritings of the trace. To this end, we exploit a flexible formalization of the domain knowledge as a lightweight **ontology**, which is a graph-based representation of knowledge as *concepts* and *relations* between these concepts. This ontology represents base concepts of the

underlying hardware and application. We show an example of such ontology in Figure 3.5. It represents a preliminary ontology that we designed for audio/video decoding with the GStreamer framework.

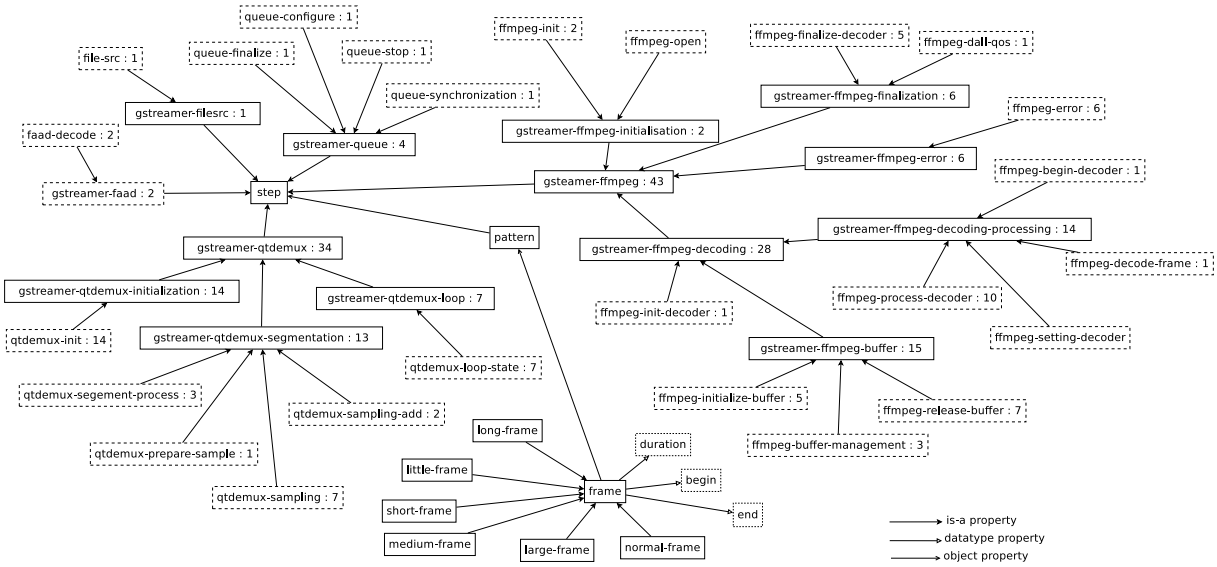


FIGURE 3.5: Ontology for audio/video decoding with GStreamer

With the help of such ontology, the challenge is to associate semantics to the frequent patterns found in the trace, such as those presented above for simplified rewritings. This semantics come by establishing *mappings* between the frequent patterns and concepts of the ontology. We have proposed a first simple method for automatically mapping frequent sequences to concepts of the ontology: in our case each concept of the ontology corresponds to a well identified step of the GStreamer audio/video decoding pipeline, so many individual events of the trace can be mapped to a concept. A pattern is thus mapped to the concept of the majority of the events constituting it.

Figure 3.6 shows a simple rewriting of a GStreamer trace based on the patterns found by the method presented above,where each pattern has been mapped to a class of an ontology representing video decoding with GStreamer. The x-axis is time. A rectangle corresponds to a pattern occurrence, where the length of the pattern is the duration of this occurrence. Each pattern is given a color corresponding to the ontology concept to which it is associated. The top row (`Summary`) shows a rewriting of the trace with all patterns occurrences. The rows below give partial rewritings of the trace, with one row per ontology concept. For each of these rows, only the occurrences of the patterns mapped to the ontology concept of the row are represented. For example the ontology concept `queue-stop` is represented in the 6th row of the figure, its color is pink. It has a long occurrence at the beginning of the trace, and three short occurrences at the end. The last row (`not_matched`) corresponds to parts of the trace not covered by any pattern.

As simple as our mapping method is, the figure shows its potential usefulness: here in the middle of the decoding, the computer running the experiment has been purposefully stressed with high CPU, memory and disk activity in order to disturb the decoding process. One can notice in the middle part of the figure that pattern occurrences become extremely long, while they are short before and after the perturbation. By looking at the classes on the left, one can immediately know which parts of the GStreamer pipeline are the most impacted the induced heavy machine load. Here the `queue` and `qtdemux` elements, which are responsible of sending raw data to the audio and video decoding algorithms, are delayed by the heavy load, possibly due to high number of concurrent disk accesses. Then the CPU-bound el-

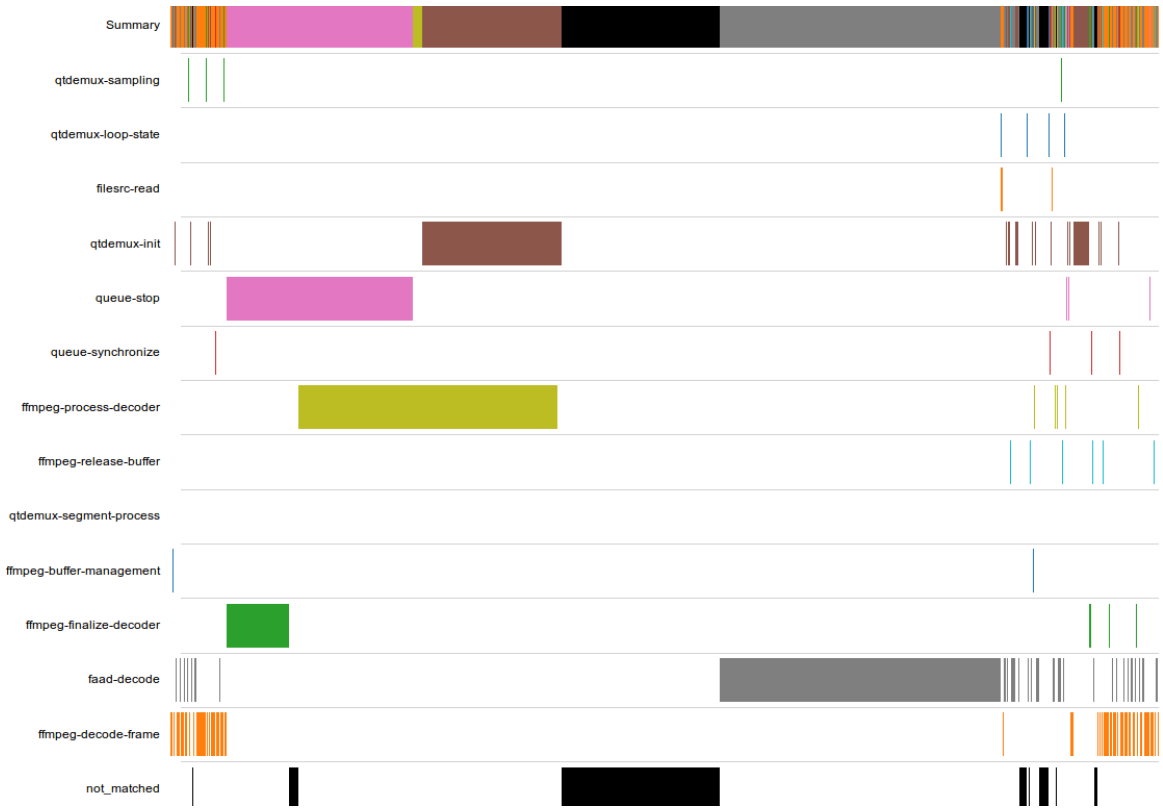


FIGURE 3.6: Trace rewritten with semantic labels on patterns

ements `faad-decode` (audio decoding) and `ffmpeg-process-decoder` (video decoding) take a large performance hit, possibly due this time to high load on CPUs.

We submitted these first results, as a proof of concept, to the EMSoft'13 conference. Our goal for extending our preliminary results, are first to provide a complete hierarchical view of the trace through the ontology to the developpers, using the hierarchy of concepts existing in the ontology, in order to provide several levels of abstraction for analysis of the trace. Next, this work allows perform complex queries on the trace using the high level concepts of the ontology, with query languages such as SPARQL [W3C08]: we want to demonstrate the interest of such advanced querying for developers in order to uncover complex problems in the trace.

B Use domain knowledge to enable scalable exploration of large user datasets

The approach presented above has shown how to use domain knowledge in order to avoid showing the patterns to the data analyst. In that approach, patterns are just representatives of concepts in the ontology, and data analysts are mostly shown how concepts of the ontology appear in the trace.

In another approach set up in a different application context, we consider an orthogonal direction: show the patterns to the data analyst, but use a taxonomy to simplify the elements constituting patterns, through a controlled generalization process.

The application concerned is the analysis of **user data**, that we briefly describe below. Numerous

social networking or social interactions website produce large amounts of such user data, which are data where a transaction corresponds to an user, and the contents of the transaction correponds to actions from that user. For example, movie rating datasets such as MovieLens [Gro11] associate to each user the set of ratings he/she gaves to movies. Flickr [Fli13] associates to each user a set of pictures, with for each pictures informations such as tags, geolocalisation, exposure information. Moreover, such datasets also maintain for each user a set of attributes (age, sex,. . .) which can help to better characterize the users.

Analyzing these data can allow to discover communities of users having common characteristics, and exhibiting similar behaviors. This information can serve several purposes: it can be exploited to improve and personalize the website, for sociological studies, and of course for online advertisement.

There has been a large range of work for detecting communities, for example in social networks [CNM04, HAHH12]. But few of these approaches focus on giving an understandable definition of the communities founds. Hence in starting PhD of Behrooz Omidvar Tehrani, co-advised with Sihem Amer-Yahia (University of Grenoble), we are studying how to discover user communities and to label them a relevant descriptions based on user characteristics, and with a short, understandable behavior based on frequent patterns found in the data. This means that only a few patterns can be used to characterize a community.

The domain knowledge that we are interested in are taxonomies of actions that can do users (for example taxonomy over the movie domain in the case of MovieLens). This way generalization operations can be carried on the elements of the patterns describing user's actions. Many different patterns will by generalization of their elements become identical, they can thus be aggregated and lead to fewer patterns for the final description of communities.

Combined with demographic information, this approach will lead to community descriptions of the form: *This community represents 35-40 years old men who watched Drama movies from the nineties and The Matrix*. Preliminary testings have shown us that doing such pattern aggregation does not give good results when based on a naive taxonomy generalization operator: it quickly gets to over-general patterns which are not useful. We are now working on more elaborate generalization operators that are based on statistical characteristics of the dataset to allow or not the generalization. In our fictional example above, many different movies have been generalized to a higher level category *Drama movies from the nineties*. However *The Matrix* could not be generalized to a higher level category (possibly "Science Fiction"), giving an important information about the community: its members do not like science fiction movies in general, but all found some special interest in *The Matrix* movie.

C Reducing the number of patterns

When there is no domain knowledge available to reduce the number of potential patterns, there still exists solution to summarize the contents of the dataset with a few patterns. Most of these solutions boil down to an optimization problem: among the complete set of N patterns, choose k patterns (with $k \ll N$) optimizing some objective function that represents how well the dataset is summarized. Some of these approaches exploit an optimization criteria specified through constraints, and then proceed to an exhaustive search of the best set of k patterns [GNR13], with a considerable computational cost. Other approaches, while based on a similar principle, use heuristics to reduce the search space . There are also approaches that exploit criterion based on information theory, and try to determine the set of patterns that compress best either the dataset through rewriting [VvLS11] or the set of frequent patterns [AGM04].

The solution that we explore in the starting PhD of Hamid Mirisae, co-advised with Eric Gaussier (University of Grenoble), is different. It is based on the fact that patterns "summarizing well" the data should be related to the "important parts" of a matrix. One example of such "important parts" are **latent factors** found during matrix decomposition such as Non-Negative Matrix Factorization. The number of

such latent factors can be controlled through an input parameter of the matrix decomposition algorithm, and they are guarantee to give together a good approximation of the initial matrix. Extracting frequent itemsets from these latent factors can thus severely reduce the output size while having some guarantees on how representative these itemsets are over the set of all frequent itemsets.

These are not the first studies on mining frequent itemsets over latent factors from matrix decompositions: the topic has already been explored by Miettinen et al. [MMG+08] and Lucchese et al. [LOP10]. Our difference is that these works are based on binary arithmetic which can produce counter-intuitive results, whereas we are focusing on an approach based on classical integer arithmetic, which guarantees simpler to interpret results.

Conclusion and Future Work

The work presented in this habilitation thesis shows the evolution of my research interests. Since my PhD and postdocs, I have constantly made contribution to the "classical" area of pattern mining, which is to propose new patterns or faster ways to mine existing patterns. My focus has always been *closed patterns*, for tree mining [TRS04, TRS+08], directed acyclic graph mining [TTN+07], gradual pattern mining [DLT10, DTLN13] and periodic pattern mining [CBT+12, Cue13]. Since 2007, I have adventured myself in the more unusual areas of generic pattern mining and parallel pattern mining on multicores, that I think are of large interest for the future of pattern mining. Their combination to make parallel and generic algorithms is especially powerful, as demonstrated by the ParaMiner algorithm [NTRM13]. My most recent research interest is to study how to make pattern mining results more understandable for an analyst, with a strong applicative focus on trace analysis.

Throughout this document I gave perspectives in each of the areas of research I am interested in (Chapter 1 section 5, Chapter 2 section 5, most of Chapter 3). In this final chapter, I will not repeat them but instead I will give my priorities among them, in order to show the future research direction that I intend to follow.

The first point is that I see the ParaMiner algorithm as a strong asset, that I want to build upon and extend for the coming years. Ideally, in the near future most of my work should thus be based on ParaMiner. This is not possible today, and requires deep theoretical work to extend foundations of ParaMiner. For example, currently it is difficult to integrate ParaMiner with the recent works of the "Mine better" chapter, mainly because ParaMiner focus on pattern mining problems *representable as sets* make it impractical for trace mining tasks that require general sequential patterns. One of my highest priorities is thus to turn ParaMiner into an algorithm capable of handling a broad range of sequence mining problems. Another point is that ParaMiner is an *exhaustive* algorithm, mining all patterns of the data satisfying the pattern definition it was given, with the risk of overwhelming users with too many results. An interesting perspective to reduce the output of ParaMiner without sacrificing its complexity guarantees would be to exploit for the enumeration the notion of *abstract lattice* presented by Soldano and Ventos [SV11]. Intuitively, this work from Formal Concept Analysis adds an operator that merges together the equivalence classes of several closed patterns, and keeps one representative for each bigger equivalence class obtained. The result is a smaller set of "closed" pattern, which are still organized in a lattice, and where the difference between these patterns and the complete set of actual closed patterns is formally defined by the operator given as input. It is a high priority for me to study if strong accessibility is preserved in this setting, and if yes on how to modify ParaMiner to operate on abstract lattices.

These two perspectives would allow ParaMiner to cover most existing pattern mining problems (both

representable as sets and representable as sequences), and at the same time to reduce its output in a principled way. When adding the other perspective presented in Chapter 1, section 5 about establishing an algebra and Domain Specific Language (DSL) for specifying pattern mining problems in ParaMiner, this would make ParaMiner an invaluable tool for data analysts.

Working on such ambitious perspectives for ParaMiner implies to also keep working on improving parallel performance on multicore processors. In the favorable case where the perspectives stated above lead to a successful outcome and that ParaMiner becomes a well used data analysis tool in the years to come, it will soon, like most pattern mining algorithms, be confronted with manycore processors. These processors will present in an amplified way the same difficulties as current multicore processors, and pose new challenges due for example to more complex bus architectures, or lack of cache coherency in some architectures. In order to get ready for the multicore era, my highest priority is to better formalize the use of caches and bus that is done in our pattern mining algorithms and especially ParaMiner. Finding the conditions to turn these algorithms into cache oblivious or at least cache aware algorithms would guarantee a more consistent parallel scalability across architectures and pattern mining problems, which will be necessary for a large diffusion of our approaches. This requires a mix of theoretical, algorithmic and experimental work in order to guarantee the actual scalability of the solutions proposed. I also consider our recent work on MapReduce evoked at the end of Chapter 2 as well as some of my work on the Haskell language as an important preparation for the manycore era: functional programming paradigms are increasingly seen as a possible way to program efficiently multicore and manycore processors in the years to come. In order to get a glimpse at the challenges offered by manycore processors, an interesting perspective is to study the parallel scalability of pattern mining algorithms such as ParaMiner on the Intel Xeon Phi, which is a recently available manycore with 60 cores.

Among all the research perspectives presented in this document, I think that the ones presented above are those who fit best how I see myself as a pattern mining researcher: first and foremost, I consider myself as a **toolsmith**. For nearly two decades, we pattern miners have provided elaborate, hand crafted data analysis tool, requiring lots of skills to be used. Striving to make the best possible pattern mining algorithms will remain the core of my research activity for a foreseeable future. However through collaboration with data owners in applications such as trace analysis, I want to make these algorithms of practical use for as many people as possible.

With better pattern mining tools, an exciting future application area should also arise in the forthcoming years. Soon, with the advent of 3D printers, FabLabs and software for easily designing complex objects (for example VehicleForge of the DARPA FANG project [DAR13]), even physical objects of our everyday lives could be produced from numerical designs coming from individuals, and not large companies. In such setting, pattern mining could have a large social impact, allowing to automatically come with a wide range of templates of various objects, or new components useful for many different objects. Fabrication of such key components could then be optimized in order to reduce natural resource consumption at a very large scale, with tremendous benefits for both people and environment.

Could that be the birth of a new research field, **eco-mining** ?

BIBLIOGRAPHY

[AAA+02] Tatsuya ASAI, Hiroki ARIMURA, Kenji ABE, Shinji KAWASOE, et Setsuo ARIKAWA. « Online Algorithms for Mining Semi-structured Data Stream ». Dans *ICDM*, pages 27–34, 2002. 5

[AGM04] F. AFRATI, A. GIONIS, et H. MANNILA. « Approximating a collection of frequent sets ». Dans *KDD 04*, pages 12–19, Seattle, Washington, USA, 2004. 54

[AHCS+05] M. AL HASAN, V. CHAOJI, S. SALEM, N. PARIMI, et M.J. ZAKI. « DMTL: A generic data mining template library ». *Library-Centric Software Design (LCSD)*, 53, 2005. . 20

[AIS93] Rakesh AGRAWAL, Tomasz IMIELINSKI, et Arun N. SWAMI. « Mining Association Rules between Sets of Items in Large Databases ». Dans *SIGMOD Conference*, pages 207–216, 1993. . . . . 1, 2, 5

[ALS12] Avinash ACHAR, Srivatsan LAXMAN, et P. S. SASTRY. « A unified view of the apriori-based algorithms for frequent episode discovery ». *Knowl. Inf. Syst.*, 31(2):223–250, 2012. 5

[ALYP10] S. AYOUNI, A. LAURENT, S. Ben YAHIA, et P. PONCELET. « Mining Closed Gradual Patterns ». Dans *International Conference on Artificial Intelligence and Soft Computing (ICAISC)*, pages 267–274, 2010. 10

[Apa12] APACHE. « Mahout ». `http://mahout.apache.org/`, 2012. 2

[atUoW13] Machine Learning Group at the University of WAIKATO. « Weka ». `http://www.cs.waikato.ac.nz/ml/weka/`, 2013. 2

[AU09] H. ARIMURA et T. UNO. « Polynomial-Delay and Polynomial-Space Algorithms for Mining Closed Sequences, Graphs, and Pictures in Accessible Set Systems ». Dans *SIAM Internal Conference on Data Mining (SDM)*, pages 1087–1098, 2009. . . . . 20, 21

[BGS10] Guy E. BLELLOCH, Phillip B. GIBBONS, et Harsha Vardhan SIMHADRI. « Low depth cache-oblivious algorithms ». Dans *SPAA*, pages 189–199, 2010. 42

[BHPW07] M. BOLEY, T. HORVÁTH, A. POIGNÉ, et S. WROBEL. « Efficient Closed Pattern Mining in Strongly Accessible Set Systems ». Dans *Mining and Learning with Graphs (MLG)*, 2007. 20

[BHPW10] M. BOLEY, T. HORVÁTH, A. POIGNÉ, et S. WROBEL. « Listing closed sets of strongly accessible set systems with applications to data mining ». *Theoretical Computer Science*, 411(3):691–700, 2010. . . . . 17, 20, 21, 22

[BJ10] Jean-François BOULICAUT et Baptiste JEUDY. Constraint-based Data Mining. Dans *Data Mining and Knowledge Discovery Handbook*, pages 339–354. 2010. 45

[BKWH11] Marina BARSKY, Sangkyum KIM, Tim WENINGER, et Jiawei HAN. « Mining Flipping Correlations from Large Datasets with Taxonomies ». *PVLDB*, 5(4):370–381, 2011. . . . 5

[BL99] Robert D. BLUMOFE et Charles E. LEISERSON. « Scheduling Multithreaded Computations by Work Stealing ». *J. ACM*, 46(5):720–748, 1999. 30

[BL07] Francesco BONCHI et Claudio LUCCHESE. « Extending the state-of-the-art of constraint-based pattern discovery ». *Data Knowl. Eng.*, 60(2):377–399, 2007. 45

[BPC06] G. BUEHRER, S. PARTHASARATHY, et Y.-K. CHEN. « Adaptive Parallel Graph Mining for CMP Architectures ». Dans *International Conference on Data Mining (ICDM)*, pages 97–106, 2006. 34

[CB04] Guojing CONG et David A. BADER. « Lock-Free Parallel Algorithms: An Experimental Study ». Dans *HiPC*, pages 516–528, 2004. 30

[CBRB09] Loïc CERF, Jérémy BESSON, Céline ROBARDET, et Jean-François BOULICAUT. « Closed patterns meet n-ary relations ». *ACM Transactions on Knowledge Discovery Data*, 3(1):3:1–3:36, mars 2009. . . . . 14, 15, 17

[CBT+12] Patricia López CUEVA, Aurélie BERTAUX, Alexandre TERMIER, Jean-François MÉHAUT, et Miguel SANTANA. « Debugging embedded multimedia application traces through periodic pattern mining ». Dans *EMSOFT*, pages 13–22, 2012.. 6, 12, 15, 17, 57

[CL10] Christopher L. CARMICHAEL et Carson Kai-Sang LEUNG. « CloseViz: visualizing useful patterns ». Dans *Proceedings of the ACM SIGKDD Workshop on Useful Patterns*, UP'10, pages 17–26, New York, NY, USA, 2010. ACM. 45

[CNM04] Aaron CLAUSET, M. E. J. NEWMAN, et Cristopher MOORE. « Finding community structure in very large networks ». *Phys. Rev. E*, 70:066111, Dec 2004. 54

[CRB04] Toon CALDERS, Christophe RIGOTTI, et Jean-François BOULICAUT. « A Survey on Condensed Representations for Frequent Sets ». Dans *Constraint-Based Mining and Inductive Databases*, pages 64–80, 2004. 14

[Cue13] Patricia Lopez CUEVA. « *Analysis and Visualization of Execution Traces on MPSoC* ». PhD thesis, University of Grenoble, 2013. . . . . 17, 34, 57

[DAR13] DARPA. « VehicleForge ». `http://vehicleforge.org/`, 2013. 58

[DB13] Guozhu DONG et James BAILEY, éditeurs. *Contrast Data Mining: Concepts, Algorithms, and Applications*. CRC Press, 2013. 45

[DG04] Jeffrey DEAN et Sanjay GHEMAWAT. « MapReduce: Simplified Data Processing on Large Clusters ». Dans *OSDI*, pages 137–150, 2004. 43

[DJLT09] L. DI-JORIO, A. LAURENT, et M. TEISSEIRE. « Mining frequent gradual itemsets from large databases ». *Advances in Intelligent Data Analysis VIII*, pages 297–308, 2009. 5, 10

[DLT10] T. D. T. DO, A. LAURENT, et A. TERMIER. « PGLCM: Efficient Parallel Mining of Closed Frequent Gradual Itemsets ». Dans *International Conference on Data Mining (ICDM)*, pages 138–147, 2010. . . . . 6, 10, 34, 35, 57

[DTLN13] Trong Dinh Thac DO, Alexandre TERMIER, Anne LAURENT, et Benjamin NEGREVERGNE. « PGLCM: Efficient Parallel Mining of Closed Frequent Gradual Itemsets ». *Knowledgne and Information Systems*, 2013. in second submission. 6, 10, 34, 57

[Ege08] Jens EGEBLAD. « *Heuristics for Multidimensional Packing Problems* ». PhD thesis, University of Copenhagen, Department of Computer Science, 2008. 51

[Fli13] FLICKR. « Home page ». `http://www.flickr.com/`, 2013. 54

[FLPR99] Matteo FRIGO, Charles E. LEISERSON, Harald PROKOP, et Sridhar RAMACHANDRAN. « Cache-Oblivious Algorithms ». Dans *FOCS*, pages 285–298, 1999. 42

[FMP09] F. FLOUVAT, F. De MARCHI, et J.-M. PETIT. « The iZi Project: Easy Prototyping of Interesting Pattern Mining Algorithms ». Dans *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 1–15, 2009. 20

[GAA10] Bodhisattwa GANGOPADHYAY, Artur ARSENIO, et Cláudia ANTUNES. « Comparative Study of Pattern Mining Techniques for Network Management System Logs for Convergent Network ». Dans *ICDEM*, pages 101–108, 2010. 1

[Gan84] Bernhard GANTER. « Two basic algorithms in concept analysis ». preprint 831, Technische Hochshule Darmstadt, 1984. 6

[GBP+05a] A. GHOTING, G. BUEHRER, S. PARTHASARATHY, D. KIM, A. NGUYEN, Y.-K. CHEN, et P. DUBEY. « Cache-conscious frequent pattern mining on a modern processor ». Dans *Very Large Data Bases (VLDB)*, pages 577–588. VLDB Endowment, 2005. 34

[GBP+05b] Amol GHOTING, Gregory BUEHRER, Srinivasan PARTHASARATHY, Daehyun KIM, Anthony D. NGUYEN, Yen-Kuang CHEN, et Pradeep DUBEY. « Cache-conscious Frequent Pattern Mining on a Modern Processor ». Dans *VLDB*, pages 577–588, 2005. 30

[Gel89] David GELERNTER. « Multiple Tuple Spaces in Linda ». Dans *PARLE (2)*, pages 20–27, 1989. 31

[GNR13] Tias GUNS, Siegfried NIJSSEN, et Luc De RAEDT. « k-Pattern Set Mining under Constraints ». *IEEE Trans. Knowl. Data Eng.*, 25(2):402–418, 2013. . . . . 45, 54

[Goe04] Bart GOETHALS. « FIMI repository website ». `http://fimi.cs.helsinki.fi/`, 2004. . . . . 23, 35

[Gro11] GroupLens Research GROUP. « MovieLens Datasets ». `http://www.grouplens.org/node/73`, 2011. 54

[GSt13] GSTREAMER. « open source multimedia framework ». http://gstreamer.freedesktop.org/, 2013. 49

[Hac10] HACKAGEDB. « HLCM package page ». http://hackage.haskell.org/package/hlcm, 2010. 8

[HAHH12] Alireza HAJIBAGHERI, Hamidreza ALVARI, Ali HAMZEH, et Sattar HASHEMI. « Community Detection in Social Networks Using Information Diffusion ». Dans *ASONAM*, pages 702–703, 2012. 54

[HPY00] J. HAN, J. PEI, et Y. YIN. « Mining frequent patterns without candidate generation ». *Special Interest Group on Managment of Data (SIGMOD)*, 29(2):1–12, 2000. 2

[HWLT02] Jiawei HAN, Jianyong WANG, Ying LU, et Petre TZVETKOV. « Mining Top-K Frequent Closed Patterns without Minimum Support ». Dans *ICDM*, pages 211–218, 2002. 45

[HYD99] J. HAN, Y. YIN, et G. DONG. « Efficient mining of partial periodic patterns in time series database ». Dans *ICDE*, 106. Published by the IEEE Computer Society, 1999. 12

[IGM01] S. IMOTO, T. GOTO, et S. MIYANO. « Estimation of genetic networks and functional structures between genes by using Bayesian networks and nonparametric regression ». Dans *Pacific Symposium on Biocomputing 2002: Kauai, Hawaii, 3-7 January 2002*, 175. World Scientific Pub Co Inc, 2001. 35

[IWM00] A. INOKUCHI, T. WASHIO, et H. MOTODA. « An apriori-based algorithm for mining frequent substructures from graph data ». *Principles of Data Mining and Knowledge Discovery (PKDD)*, pages 13–23, 2000. 5

[Kal13] KALRAY. « Home Page ». http://www.kalray.eu, 2013. 41

[KFI+12] C. Kamdem KENGNE, L. C. FOPA, N. IBRAHIM, Alexandre TERMIER, Marie-Christine ROUSSET, et Takashi WASHIO. « Enhancing the Analysis of Large Multimedia Applications Execution Traces with FrameMiner ». Dans *ICDM Workshops*, pages 595–602, 2012. 51

[Kni13] KNIME. « Home page ». http://www.knime.org/, 2013. 2

[KSS05] Daniel A. KEIM, Jörn SCHNEIDEWIND, et Mike SIPS. « FP-Viz: Visual Frequent Pattern Mining ». Dans *Proceedings of IEEE Symposium on Information Visualization (InfoVis '05), Poster Paper*, Minneapolis, USA, 2005. 45

[Kuz96] Sergei KUZNETSOV. « Mathematical aspects of concept analysis ». *Journal of Mathematical Science*, 80(2):1654–1698, 1996. 6

[KVBH11] Darren J. KERBYSON, Abhinav VISHNU, Kevin J. BARKER, et Adolfy HOISIE. « Codesign Challenges for Exascale Systems: Performance, Power, and Reliability ». *IEEE Computer*, 44(11):37–43, 2011. 43

[LL07] Eric LI et Li LIU. « Optimization of Frequent Itemset Mining on Multiple-Core Processor ». Dans *VLDB*, pages 1275–1285, 2007. 5

[LOP07] C. LUCCHESE, S. ORLANDO, et R. PEREGO. « Parallel Mining of Frequent Closed Patterns: Harnessing Modern Computer Architectures ». Dans *International Conference on Data Mining (ICDM)*, pages 242–251, 2007. 40

[LOP10] Claudio LUCCHESE, Salvatore ORLANDO, et Raffaele PEREGO. « A generative pattern model for mining binary datasets ». Dans *SAC*, pages 1109–1110, 2010. 55

[LTP13] Sofiane LAGRAA, Alexandre TERMIER, et Frédéric PÉTROT. « Data Mining MPSoC Simulation Traces to Identify Concurrent Memory Access Patterns ». Dans *DATE (Design, Automation and Test in Europe)*, 2013. . . . . 47, 48

[LWZ+08] Haoyuan LI, Yi WANG, Dong ZHANG, Ming ZHANG, et Edward Y. CHANG. « Pfp: parallel fp-growth for query recommendation ». Dans *RecSys*, pages 107–114, 2008. . 2, 43

[LYY+05] Chao LIU, Xifeng YAN, Hwanjo YU, Jiawei HAN, et Philip S. YU. « Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs ». Dans *SDM*, 2005. 1

[MH01] Sheng MA et Joseph L. HELLERSTEIN. « Mining Partially Periodic Event Patterns with Unknown Periods ». Dans *Proceedings of the 17th International Conference on Data Engineering*, pages 205–214, Washington, DC, USA, 2001. IEEE Computer Society. . 12

[Mic12] MICROSOFT. « Microsoft Association Algorithm Technical Reference ». `http://msdn.microsoft.com/en-us/library/cc280428.aspx`, 2012. 2

[MMG+08] Pauli MIETTINEN, Taneli MIELIKÄINEN, Aristides GIONIS, Gautam DAS, et Heikki MANNILA. « The Discrete Basis Problem ». *IEEE Trans. Knowl. Data Eng.*, 20(10):1348–1362, 2008. 55

[MMTW10] Paul E. MCKENNEY, Maged M. MICHAEL, Josh TRIPLETT, et Jonathan WALPOLE. « Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory ». *Operating Systems Review*, 44(3):93–101, 2010. 30

[MTV97] Heikki MANNILA, Hannu TOIVONEN, et A. Inkeri VERKAMO. « Discovery of Frequent Episodes in Event Sequences ». *Data Min. Knowl. Discov.*, 1(3):259–289, 1997. 25

[Neg11] Benjamin NEGREVERGNE. « *A Generic and Parallel Pattern Mining Algorithm for Multi-Core Architectures* ». PhD thesis, University of Grenoble, 2011. 34

[NK05] Siegfried NIJSSEN et Joost N. KOK. « The Gaston Tool for Frequent Subgraph Mining ». *Electr. Notes Theor. Comput. Sci.*, 127(1):77–87, 2005. 40

[NP12] Lhouari NOURINE et Jean-Marc PETIT. « Extending Set-based Dualization: Application to Pattern Mining ». Dans *ECAI*, pages 630–635, 2012. 25

[NTMU10] B. NEGREVERGNE, A. TERMIER, J.-F. MEHAUT, et T. UNO. « Discovering Closed Frequent Itemsets on Multicore: Parallelizing Computations and Optimizing Memory Accesses ». Dans *International Conference on High Performance Computing and Simulation (HPCS)*, pages 521–528, 2010. . . . . 34, 40

[NTRM13] Benjamin NÉGREVERGNE, Alexandre TERMIER, Marie-Christine ROUSSET, et Jean-François MÉHAUT. « ParaMiner: a generic pattern mining algorithm for multi-core architectures ». *Data Min. Knowl. Discov.*, 2013. . . . . 6, 22, 23, 24, 25, 34, 35, 39, 57

[Ope13a] OPENMP. « Home Page ». `http://openmp.org/`, 2013. 31

[Ope13b] OPENMPI. « Portable Hardware Locality (hwloc/lstopo) ». http://www.open-mpi.org/projects/hwloc/, 2013. 28

[PBTL99] N. PASQUIER, Y. BASTIDE, R. TAOUIL, et L. LAKHAL. « Discovering Frequent Closed Itemsets for Association Rules ». Dans *International Conference on Database Theory (ICDT)*, pages 398–416, 1999. 7

[RCP08] Chedy RAÏSSI, Toon CALDERS, et Pascal PONCELET. « Mining conjunctive sequential patterns ». *Data Min. Knowl. Discov.*, 17(1):77–93, 2008. 25

[SA95] Ramakrishnan SRIKANT et Rakesh AGRAWAL. « Mining Generalized Association Rules ». Dans *VLDB*, pages 407–419, 1995. 5

[SC05] A. SOULET et B. CRÉMILLEUX. « An Efficient Framework for Mining Flexible Constraints ». Dans *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 661–671, 2005. 45

[SCD02] Sandeep SEN, Siddhartha CHATTERJEE, et Neeraj DUMIR. « Towards a theory of cache-efficient algorithms ». *J. ACM*, 49(6):828–858, 2002. 42

[STM] STMICROELECTRONICS. « KPTrace ». http://www.stlinux.com/devel/traceprofile/kptrace. 49

[SV11] Henry SOLDANO et Véronique VENTOS. « Abstract Concept Lattices ». Dans *ICFCA*, pages 235–250, 2011. 57

[SVA97] Ramakrishnan SRIKANT, Quoc VU, et Rakesh AGRAWAL. « Mining Association Rules with Item Constraints ». Dans *KDD*, pages 67–73, 1997. 45

[TC12] Nikolaj TATTI et Boris CULE. « Mining closed strict episodes ». *Data Min. Knowl. Discov.*, 25(1):34–66, 2012. 5

[Til13] TILERA. « Home Page ». http://www.tilera.com/, 2013. 41

[TNMS11] Alexandre TERMIER, Benjamin NEGREVERGNE, Simon MARLOW, et Satnam SINGH. « HLCM: a first experiment on parallel data mining with Haskell ». Research Report RR-LIG-009, LIG, Grenoble, France, 2011. 8

[TP09] S. TATIKONDA et S. PARTHASARATHY. « Mining tree-structured data on multicore systems ». *Very Large Data Base (VLDB)*, 2(1):694–705, 2009. . . . . 27, 30, 35

[TRS04] Alexandre TERMIER, Marie-Christine ROUSSET, et Michèle SEBAG. « DRYADE: A New Approach for Discovering Closed Frequent Trees in Heterogeneous Tree Databases ». Dans *ICDM (International Conference on Data Mining)*, pages 543–546, 2004. . . . . 31, 57

[TRS+08] Alexandre TERMIER, Marie-Christine ROUSSET, Michèle SEBAG, Kouzou OHARA, Takashi WASHIO, et Hiroshi MOTODA. « DryadeParent, An Efficient and Robust Closed Attribute Tree Mining Algorithm ». *TKDE (IEEE Transactions on Knowledge and Data Engineering)*, 20(3):300–320, 2008. . . . . 31, 57

[TTN+07] Alexandre TERMIER, Yoshinori TAMADA, Kazuyuki NUMATA, Seiya IMOTO, Takashi WASHIO, et Tomoyuki HIGUCHI. « DigDag, a first algorithm to mine closed frequent embedded sub-DAGs ». Dans *MLG (International Workshop on Mining and Learning with Graphs)*, pages 41–45, 2007. . . . . 31, 57

[TWSY10] Vincent S. TSENG, Cheng-Wei WU, Bai-En SHIE, et Philip S. YU. « UP-Growth: an efficient algorithm for high utility itemset mining ». Dans *KDD*, pages 253–262, 2010. 5

[UAUA04] T. UNO, T. ASAI, Y. UCHIDA, et H. ARIMURA. « An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases ». Dans *Discovery Science*, pages 16–31, 2004. . . . . 6, 8

[UKA04] T. UNO, M. KIYOMI, et H. ARIMURA. « LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets ». Dans *Frequent Itemset Mining Implementatino Workshop (FIMI)*, 2004. . . . . 2, 5

[VvLS11] Jilles VREEKEN, Matthijs van LEEUWEN, et Arno SIEBES. « Krimp: mining itemsets that compress ». *Data Min. Knowl. Discov.*, 23(1):169–214, 2011. 54

[W3C08] W3C. « SPARQL Query Language for RDF ». `http://www.w3.org/TR/rdf-sparql-query/`, 2008. 53

[YH02] X. YAN et J. HAN. « gSpan: Graph-based substructure pattern mining ». *International Conference on Data Mining (ICDM)*, 2002. 40

[YWY03] Jiong YANG, Wei WANG, et Philip S. YU. « Mining Asynchronous Periodic Patterns in Time Series Data ». *IEEE Transactions on Knowledge and Data Engineering*, 15(3):613–628, mars 2003. 12

[YZH05] X. YAN, X. J. ZHOU, et J. HAN. « Mining Closed Relational Graphs with Connectivity Constraints ». Dans *International Conference on Data Engeneering (ICDE)*, pages 357–358, 2005. . . . . 35, 40

[ZPL97] Mohammed Javeed ZAKI, Srinivasan PARTHASARATHY, et Wei LI. « A Localized Algorithm for Parallel Association Mining ». Dans *SPAA*, pages 321–330, 1997. . . . . 30, 34

[ZS06] Mohammed J. ZAKI et Karlton SEQUEIRA. Data Mining in Computational Biology. Dans Srinivas ALURU, éditeur, *Handbook of Computational Molecular Biology*, Computer and Information Science Series, Chapter 38, pages 38/1–38/26. Chapman & Hall/CRC Press, 2006. 1