Efficient parallel mining of gradual patterns on multicore processors

Anne Laurent, Benjamin Négrevergne, Nicolas Sicard, and Alexandre Termier

Abstract Mining *gradual patterns* plays a crucial role in many real world applications where huge volumes of complex numerical data must be handled, *e.g.,* biological databases, survey databases, data streams or sensor readings. Gradual patterns highlight complex order correlations of the form *"The more/less X, the more/less Y"*. Only recently algorithms have appeared to mine efficiently gradual rules. However, due to the complexity of mining gradual rules, these algorithms cannot yet scale on huge real world datasets. In this paper, we thus propose to exploit parallelism in order to enhance the performances of the fastest existing one (GRITE) on multicore processors. Through a detailed experimental study, we show that our parallel algorithm scales very well with the number of cores available.

1 Introduction

Frequent pattern mining is a major domain of data mining. Its goal is to efficiently discover in data patterns having more occurrences than a pre-defined threshold. This domain started with the analysis of transactional data (frequent itemsets), and quickly expanded to the analysis of data having more complex structures such as sequences, trees or graphs [Han and Kamber, 2006]. All the frequent pattern mining algorithms must explore a huge search space and are very expensive computationaly, the cost increasing with the complexity of the patterns to be mined. A large part

Anne Laurent
Univ. Montpellier 2, LIRMM, CNRS UMR 5506, 162 rue Ada, 34095 Montpellier cedex 5 e-mail: laurent@lirmm.fr

Benjamin Négrevergne, Alexandre Termier
LIG, UJF, CNRS UMR 5217, 681 rue de la Passerelle, BP 72, 38402 Saint Martin d'Hères e-mail: Benjamin.Negrevergne@imag.fr, Alexandre.Termier@imag.fr

Nicolas Sicard
LRIE, EFREI, 30-32 avenue de la République, 94800 Villejuif e-mail: nicolas.sicard@efrei.fr

of the research works in pattern mining thus consisted in designing more and more efficient algorithms, capable of scaling up on huge and/or complex databases.

Very recently, a new pattern mining problem appeared: mining frequent *gradual itemsets* (also known as *gradual patterns*). This problem considers transactional databases where attributes can have a numerical value. The goal is then to discover frequent co-variations between attributes, such as: "The higher the age, the higher the salary". This problem has numerous applications, as well for analyzing client databases for marketing purposes as for analyzing patient databases in medical studies. For instance, it has recently received a lot of attention for applications on breast cancer where a large number of DNA micro-arrays are generated by biologists containing numeric data describing the levels of expression of genes. Di Jorio et al. [Di Jorio et al., 2009] recently proposed GRITE, a first efficient algorithm for mining gradual itemsets and gradual rules. This algorithm is based on Apriori [Agrawal and Srikant, 1994], and can be applied on synthetic databases having up to several hundreds of attributes or up to thousands of lines, whereas previous algorithms were limited to databases having six attributes [Berzal et al., 2007]. However, because the problem at hand is by far more complex than that of classical frequent itemset mining problem, the GRITE algorithm can be very long or impossible to execute, even on databases that could appear *small* regarding current frequent pattern mining tasks and current available scientific databases. Thus GRITE cannot scale on large real databases (*e.g.,* having both a large number of lines and columns, or having millions of lines/hundreds or thousands of attributes).

There are two (complementary) options to improve the scaling up capabilities of GRITE and to allow experts using this algorithm to handle large databases. The first one lies in algorithmic improvements, for example using pattern growth techniques [Han et al., 2000] and by defining the notion of *closure* on frequent gradual itemsets [Pasquier et al., 1999, Uno, 2005]. This first option needs an important theoretical and algorithmic work, and is thus an indispensable but mid-term solution. In order to provide more quickly experts with performant algorithms capable of exploiting their real-world databases, a second solution is to exploit parallelism on recent multi-core processors.

Since 2005, physical limits have prevented further frequency increases in processors, cancelling the possible related performance improvements. However the number of transistors on a die continues to double every 18 months according to Moore's Law, which leads to a new generation of processors having multiple computation *cores*. Exploiting optimally these processors is achieved through the writing of parallel programs. The multi-core processors have different specifities from either clusters of commodity computers or SMPs (Symetric Multi-processors): the memory is not distributed like in clusters but shared. This is similar to SMPs, but a lot of SMPs have a NUMA (Non Uniform Memory Access) architecture: there are several memory blocks, with different access speeds from each processor depending on how far the memory block is from the processor. Multi-core processors are usually on a UMA (Uniform Memory Access) architecture: there is one memory block, so all cores have an equal access time to this memory. However there is also only one bus between the multi-core processor and its memory, which means that

programs can be more limited by bandwith than raw computing power. The usage of memory must be done in a very carefull way, unlike previous sequential programs.

Pattern mining researchers, always in need of more computing power, have started investigating new algorithms dedicated for multi-core processors, as presented for example in [Buehrer et al., 2006, Lucchese et al., 2007], [Liu et al., 2007], [Tatikonda and Parthasarathy, 2009]. Analyzing their first results shows that the more complex the patterns to mine (trees, graphs), the better the scale-up results on multiple cores could be. This is because memory accesses are a very limiting factor, and in case of complex patterns there are a lot of computations to perform on the data loaded into the processor's cache, which execution time far outstrips the time to recover the data from memory. This suggests that using multicore processors for mining gradual itemsets using the GRITE algorithm could give interesting results, as the computations to perform for a simple frequency count are very complex and include computing the longest path of a graph. We show in our experiments that indeed, there is a quasi-linear scale up with the number of cores for our multi-threaded algorithm.

The outline of this paper is as follows: In Section 2, we explain the notion of gradual itemsets. In Section 3, we present with more details the related works on gradual patterns and parallel pattern mining. In Section 4, we present our parallel algorithm for mining frequent gradual itemsets, and in Section 5, we show the results of our experimental evaluation. Last, we conclude and give some perspectives in Section 6.

2 Gradual Patterns

Gradual patterns refer to itemsets of the form *"The more/less X_1, ..., the more/less X_n"*. We assume here that we are given a database *DB* that consists of a single table whose tuples are defined on the attribute set $\mathscr{I}$. In this context, gradual patterns are defined to be subsets of $\mathscr{I}$ whose elements are associated with an ordering, meant to take into account increasing or decreasing variations. Note that $t[I]$ hereafter denotes the value of t over attribute I.

For instance, we consider the database given in Table 1 describing fruits and their characteristics.

Definition 1 *(Gradual Itemset) Given a table DB over the attribute set $\mathscr{I}$, a gradual item is a pair (I,θ) where I is an attribute in $\mathscr{I}$ and θ a comparison operator in $\{\geq,\leq\}$.*

A gradual itemset $g = \{(I_1,\theta_1),...,(I_k,\theta_k)\}$ is a set of gradual items of cardinality greater than or equal to 2*.*

For example, $(Size,\geq)$ is a gradual item, while $\{(Size,\geq),(Weight,\leq)\}$ is a gradual itemset.

The support of a gradual itemset in a database *DB* amounts to the extent to which a gradual pattern is present in a given database. Several support definitions have

Id	Size (S)	Weight (W)	Sugar Rate (SR)
t_1	6	6	5.3
t_2	10	12	5.1
t_3	14	4	4.9
t_4	23	10	4.9
t_5	6	8	5.0
t_6	14	9	4.9
t_7	18	9	5.2
t_8	23	10	5.3
t_9	28	13	5.5

Table 1 Fruit Characteristics

been proposed in the literature (see Section 3 below). In this paper, we consider the support as being defined as the number of tuples that can be ordered to support all item comparison operators:

Definition 2 *(Support of a Gradual Itemset) Let DB be a database and* $g = \{(I_1, \theta_1), ..., (I_k, \theta_k)\}$ *be a gradual itemset. The cardinality of g in DB, denoted by* $\lambda(g, DB)$*, is the length of the longest list* $l = \langle t_1, \ldots, t_n \rangle$ *of tuples in DB such that, for every* $p = 1, \ldots, n-1$ *and every* $j = 1, \ldots, k$*, the comparison* $t_p[I_j]\ \theta_j\ t_{p+1}[I_j]$ *holds.*

The support of g in DB, denoted by $supp(g, DB)$*, is the ratio of* $\lambda(g, DB)$ *over the cardinality of DB, which we denote by* $|DB|$*. That is,* $supp(q, DB) = \frac{\lambda(g, DB)}{|DB|}$.

In order to compute $\lambda(g, DB)$, [Di Jorio et al., 2009] proposes to consider the graph where nodes are the tuples from *DB* and where there exists a vertex between two nodes if the corresponding tuples can be ordered according to g.

For example, Figure 1 shows the ordering of the tuples of *DB*, according to the gradual itemset $g = \{(S, \geq), (SR, \leq)\}$, whose intuitive meaning is *the bigger the size, the lower the sugar rate*. As in this graph, the length of the longest totally ordered list of tuples is 5, and as *DB* contains 9 tuples, we have $supp(g, DB) = \frac{5}{9}$.

The algorithm proposed by Di Jorio et al. to mine gradual itemsets is an APriori-based algorithm. We point out that gradual itemsets are assumed to be sets of cardinality greater than or equal to 2, because sorting tuples according to one attribute is always possible, which is not the case when considering more than one attribute.

The algorithm thus starts with the computation of the support of all gradual patterns constituted by a pair of gradual items (attribute, operator), and then operates at every level k by combining the frequent patterns containing $k-1$ gradual items to build up candidates containing k attributes that are then validated or not after computing their support before processing to step $k+1$.

It is worth noting that the storage of all orderings at level k, even in a binary format, can be very memory-consuming. However, the main bottleneck is often due to the fact that the computation of the support is very time-consuming. The tuples must indeed be ordered depending on the gradual itemset being considered. This ordering is stored in a binary matrix associated with the graph. Then the length of the longest path of this graph is computed in order to get the support.

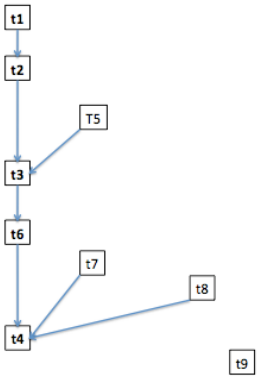

Fig. 1 Graph of $g = \{(S, \geq), (SR, \leq)\}$ as computed from Table 1

3 Related Work

In this section, we discuss the related works on mining gradual patterns as well as on parallel frequent pattern mining.

3.1 Gradual Pattern Mining

Gradual patterns and gradual rules have been studied for many years in the framework of control, command and recommendation.

More recently, data mining algorithms have been studied in order to automatically mine such patterns [Di Jorio et al., 2009, Berzal et al., 2007, Di Jorio et al., 2008, Fiot et al., 2008, Hüllermeier, 2002, Laurent et al., 2009].

The approach in [Hüllermeier, 2002] uses statistical analysis and linear regression in order to extract gradual rules. In [Berzal et al., 2007], the authors formalize four kinds of gradual rules in the form *The more/less X is in A, then the more/less Y is in B*, and propose an Apriori-based algorithm to extract such rules. However, frequency is computed from pairs of objects, increasing the complexity of the algorithm. Despite a good theoretical study, the algorithm is limited to the extraction of gradual rules of length 3.

The approach in [Fiot et al., 2008] is the first attempt to formalize gradual sequential patterns. This extension of itemsets allows for the combination of gradual temporality ("*the more quickly*") and gradual list of itemsets. The extraction is done by the algorithm GRaSP, based on generalized sequential patterns [Masseglia et al., 2004] to extract gradual temporal correlations.

In [Di Jorio et al., 2009] and [Di Jorio et al., 2008], two methods to mine gradual patterns are proposed. The difference between these approaches lies in the compu-

tation of the support: whereas, in [Di Jorio et al., 2008], a heuristic is used and an approximate support value is computed, in [Di Jorio et al., 2009], the correct support value is computed (see above).

In [Laurent et al., 2009], the authors propose another way to compute the support, by using ranking such as the Kendall τ ranking correlation coefficient, which basically computes, instead of the length of the longest path, the number of pairs of lines that are correctly ordered (concordant and discordant pairs).

It is important to note in this respect that, in the current paper, the method of [Di Jorio et al., 2009] is used for the computation of frequent gradual patterns, as it is the most efficient exhaustive method to mine such patterns.

3.2 Parallel Frequent Pattern Mining

Since 1996, pattern mining researchers have worked on parallel algorithms. There were numerous works for mining frequent patterns on SMPs [Zaki, 1999] or on clusters [Agrawal and Shafer, 1996, Zaki et al., 1997]. At that time, the main memory of commodity computers was much smaller than the size of most databases (hundreds of Megabytes versus Gigabytes), so the first interest of parallel computing was to handle efficiently bigger databases through distribution. However, with the advent of bigger memories and the discovery of more efficient ways of exploring the search space (*e.g.,* closed frequent patterns), publications about parallel pattern mining became scarce until 2005. Since the apparition of multicore processors, also called *Chip MultiProcessors (CMP)*, a new trend of research has emerged on how to have better performances by using these CMP. [Buehrer et al., 2006] pioneered this trend, presenting a parallel graph mining algorithm with excellent scale-up capacities. Their algorithm is based on gSpan [Yan and Han, 2002], and their contribution consists in an efficient depth-first way to share the work between the cores, and a technique to exploit cache temporal locality when deciding to either immediatly mine recursive calls or enqueue them. In 2007 Lucchese et al. [Lucchese et al., 2007] presented the first algorithm to mine closed frequent itemsets on CMP. Their contribution is focused on how to partition the work, and they show the interest of using SIMD instructions for further increasing performance. The same year, Liu et al. [Liu et al., 2007] presented a parallelisation of the well known FP-growth [Han et al., 2000] algorithm. More recently, Tatikonda et al. [Tatikonda and Parthasarathy, 2009] presented an algorithm for mining frequent trees with near-linear speedup. They show that the main limiting factor for performance of parallel pattern mining algorithm on CMP is that the memory is shared among all the cores. So if each core requests a lot of data, the bus will be saturated and the performance will drop: there is a too important *bandwidth pressure*. This is opposite to what had always worked well with sequential algorithms, where to avoid redundant computations a large quantity of intermediary data was stored in memory. Here Tatikonda et al. show that the *working set* size must be reduced as much as possible, at the expense of redundant computations if needed. They also

show that traditional pointer-based data structures are ill-adapted for CMP parallel pattern mining, because of their bad locality in the cache.

In this work, we tackle the complex problem of mining gradual patterns. We are in the favorable case where there are a lot of computations to do for each chunk of data transfered from memory, so the bandwidth pressure should not be a problem as long as the memory usage is kept low. This work is the first work on parallel extraction of gradual patterns.

4 PGP-mc: Parallel Gradual Pattern Extraction

4.1 Gradual Pattern Characteristics

The gradual itemset extraction problem relies on the following two costly operations: (i) database lines scheduling and associated binary matrix construction and (ii) longest path exploration (see [Di Jorio et al., 2009] for more details).

This problem is different from the classical itemsets problem in which we can tell if a given line of the database does - or does not - support the searched itemset independently of other lines. Even the sequential pattern extraction is an intermediary problem because operations can be distributed on different blocks of database lines (all lines from the same block belong to the same client).

In the gradual pattern case, all lines are required for each itemset construction, making the distribution of data based on line blocks very difficult. Instead, our proposal is based on the fact that these operations are repeated a significant number of times during frequent itemset searches.

4.2 Proposed Solution: GRITE-MT

The sequential GRITE algorithm relies on a tree-based exploration, where every level $N+1$ is built upon the previous level N. The first level of the tree is initialized with all attributes, which all become itemset siblings. Then, *itemsets* from the second level are computed by combining *frequent itemsets* siblings from the first level through what we call the $Join()$ procedure. Candidates whose support is greater than a pre-defined threshold - they are considered as *frequent* - are retained in level $N+1$. Algorithm 1 shows a simplified view of the level $N+1$ construction.

In this solution, every level cannot be processed until the previous one has been completed, at least partially. So, we focused our efforts on the parallelization of each level construction where individual combinations of itemsets (through the $Join()$ procedure) are mostly independant tasks. The main problem is that the number of operations inside each inner *foreach* loop of the algorithm 1 cannot be easily anticipated, at least for levels higher than 2. Moreover, the number of siblings may vary

Algorithme 1 Simplified GRITE level processing.

```
1 foreach itemset i in level N do
2     foreach itemset j in Siblings_{j>i}(i) do
3         itemset k ← Join(i, j)
4         if k is frequent then
5             k becomes child node of i (gets index j)
6             f k is stored in level N + 1 g
7         endif
8     endforeach
9 endforeach
```

by a large margin depending of itemsets i. A simple parallel loop would lead to an irregular load distribution on several processing units.

In order to offset this irregularity, our approach dynamically attributes new tasks to a pool of threads on a "first come, first served" basis. At first, all frequent itemsets from the given level are marked unprocessed and queued in Q_i. A new frequent itemset i is dequeued and all its siblings are stored in a temporary queue Q_{si}. Each available thread then extracts the next unprocessed sibling j from Q_{si} and builds a new candidate k from i and j. The candidate is stored in level $N+1$ if it is considered frequent. When Q_{si} is empty, the next frequent itemset i is dequeued and Q_{si} is filled with its own siblings. The process is repeated until all itemsets i are processed (*e.g.*, Q_i is empty). Algorithm 2 is a simplified description of this multithreaded approach.

4.3 Implementation and Preliminary Optimizations

As mentioned earlier, memory bandwidth is one of the main factors which can limit speed-up progression in multithreaded programs on *CMP*s. Another problem comes from the very unpredictable amount of memory which will be dynamically allocated to store all frequent items. Dynamic memory allocations are usually system-level tasks that cannot be performed simultaneously by the operating system. This may cause another penalty for parallel executions because threads may have to wait longer before obtaining a requested memory zone. As a matter of fact, some preliminary experiments have shown that it can be very problematic when the number of concurrent threads grows.

In order to simplify memory management and eliminate all unnecessary temporary memory transactions, we profiled and optimized the initial C++ implementation from [Di Jorio et al., 2009]. The underlying algorithm was not modified during the process. The resulting sequential program now needs, on average, less than half of

Algorithme 2 Simplified GRITE-MT (multithreaded) level processing.

```
1  i, j : itemsets
2  P_t : pool of threads
3  Q_i : queue ← itemsets from level N
4  Q_si : queue ← ∅ {unprocessed siblings}
5  foreach thread in P_t (in parallel) do
6      while Q_i ≠ ∅ OR Q_si ≠ ∅ do
7          if Q_si = ∅ then
8              i ← dequeue(Q_i)
9              Q_si ← Siblings_{j>i}(i)
10         endif
11         j ← dequeue(Q_si)
12         local itemset k ← Join(i, j)
13         if k is frequent then
14             k becomes child node of i (gets index j)
15             {k is stored in level N+1}
16         endif
17     endwhile
18 endforeach
```

the memory usage and almost a third of the initial execution time. We conducted our experiments with this optimized version.

Note that threads have been implemented in our program using the POSIX threads library (threads are automatically scheduled on hardware processing units by the operating system).

5 Experimental Results and Discussion

In this section we report experimental results from the execution of our program on two different workstations with up to 32 processing cores.

- COYOTE is a workstation containing 8 AMD Opteron 852 processors (each with 4 cores), 64GB of RAM with Linux Centos 5.1 and g++ 3.4.6.
- IDKONN is a workstation containing 4 Intel Xeon 7460 processors (each with 6 cores), 64GB of RAM with Linux Debian 5.0.2 and g++ 4.3.2.

The experiments are led on synthetic databases automatically generated by a tool based on an adapted version of IBM Synthetic Data Generation Code for Associa-

tions and Sequential Patterns[1]. This tool generates numeric databases depending on the following parameters: the number of lines, the number of attributes/columns and the average number of distinct values per attribute.

5.1 Scalability

The following figures illustrate how the proposed solution scales with both the increasing number of threads and the growing complexity of the problem. The complexity comes either from the number of lines or from the number of attributes in the database as the number of individual tasks is related to the number of attributes while the complexity of each individual task - itemsets joining - depends on the number of lines. In this article, we report results for two sets of experiments.

The first set of experiments involves databases with relatively few attributes but a significant number of lines. This kind of databases usually produces few frequent items with moderate to high thresholds. As a consequence the first two level computations represent the main part of the global execution time. Figure 2 shows the evolution of execution times for 10000-line databases - ranging from 10 to 50 attributes - on COYOTE. Figure 3 gives the corresponding speed-ups.

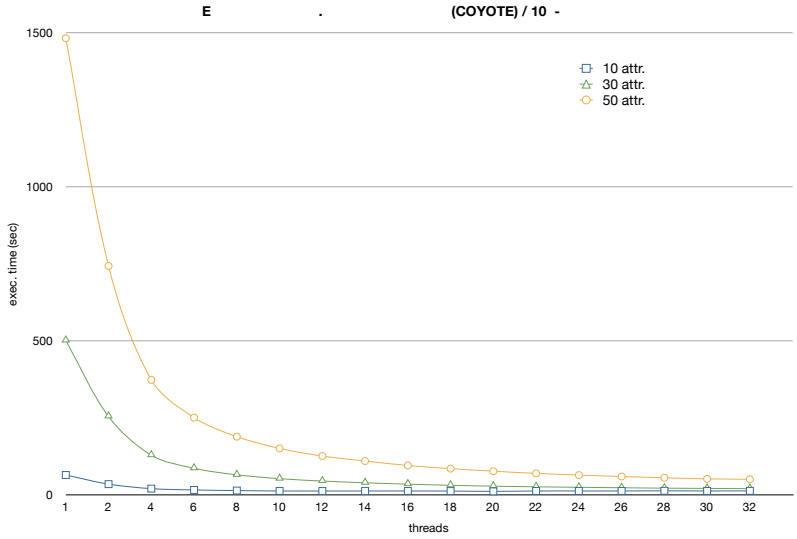

Fig. 2 Execution time related to the number of threads. Test databases ranging from 10 to 50 attributes with 10k lines, on COYOTE.

[1] www.almaden.ibm.com/software/projects/hdb/resources.shtml

Fig. 3 Speed-up related to the number of threads. Test databases ranging from 10 to 50 attributes with 10k lines, on COYOTE.

As shown by Figure 3, speed-ups can reach very satisfying values in sufficiently complex situations. For example, speed-up is around 30 with 50 attributes where the theoretical maximum is 32. The upper limit for 10 and 20 attributes is not really surprising and can be explained by the lower number of individual tasks. As the number of tasks decreases and the complexity of each task increases, it becomes more and more difficult to reach an acceptable load balance. This phenomenon is especially tangible during the initial database loading phase (construction of the first level of the tree) where the number of tasks is exactly the number of attributes. For example, the sequential execution on the 10-attribute database takes around 64 seconds from which the database loading process takes 9 seconds. With 32 threads, the global execution time goes down to 13 seconds but more than 5.5 seconds are still used for the loading phase.

We report in Figure 4 the results of the same experiments on IDKONN. We just give the speedups, as the results are very similar to the results of COYOTE. Excellent speedups, up to 22.3 out of 24 threads, are obtained with a 50-attribute database. For a small database with only 10 attributes, speedups are limited to 4.8, for the same reasons as above.

The second set of experiments reported in this article is about databases with growing complexity in terms of attributes. Figure 5 shows the evolution of execution times for 500-line databases with different number of attributes - ranging from 50 to 350 - on IDKONN. Figure 6 reports the corresponding speed-ups.

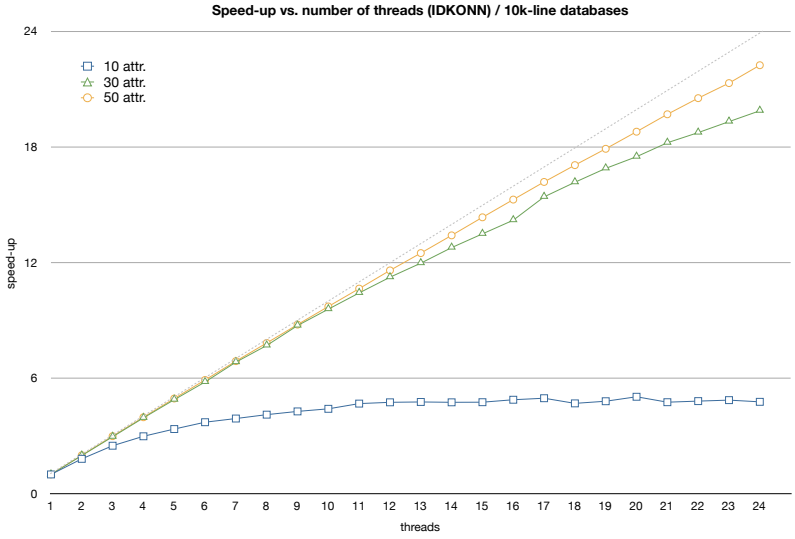

Fig. 4 Speed-up related to the number of threads. Test databases ranging from 10 to 50 attributes with 10k lines, on IDKONN.

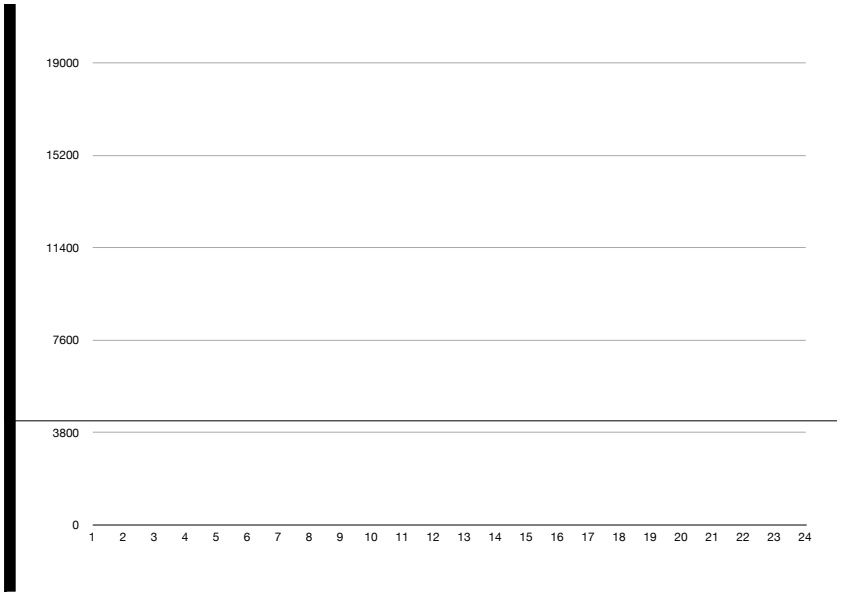

Fig. 5 Execution time related to the number of threads. Test databases ranging from 50 to 350 attributes with 500 lines, on IDKONN.

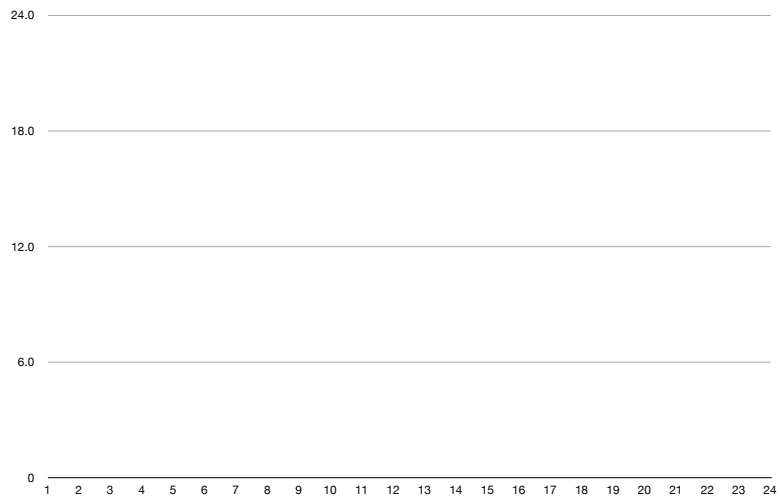

Fig. 6 Speed-up related to the number of threads. Test databases ranging from 50 to 350 attributes with 500 lines, on IDKONN.

As we can see, our solution is extremely efficient and scales very well for many attributes: we almost reach the theoretical maximum linear speed-up progression for 150 attributes or more. For example, the sequential processing of the 350 attributes database took more than five hours while it spend approximatively 13 minutes using 24 threads on IDKONN. Furthermore, speed-up results are particularly stable from one architecture to another, meaning that performances do not rely on very specific architectural features (caches, memory systems...). Figure 7 shows very similar results on COYOTE (with 32 threads)[2].

The 50-attribute database experiment may not seem relevant in a massive parallelization problem as its sequential execution time peaks only at 3.3 seconds on COYOTE. But, with an execution time of less than 0.2 second with 16 threads, this example illustrates how our approach can still achieve a very tangible acceleration on this particular case, which appears as crucial for real time or near real time data mining and applications (*e.g.,* intrusion/fraud detection).

[2] Note that due to process running time limitations, we could'nt run tests for databases with more than 300 attributes on COYOTE.

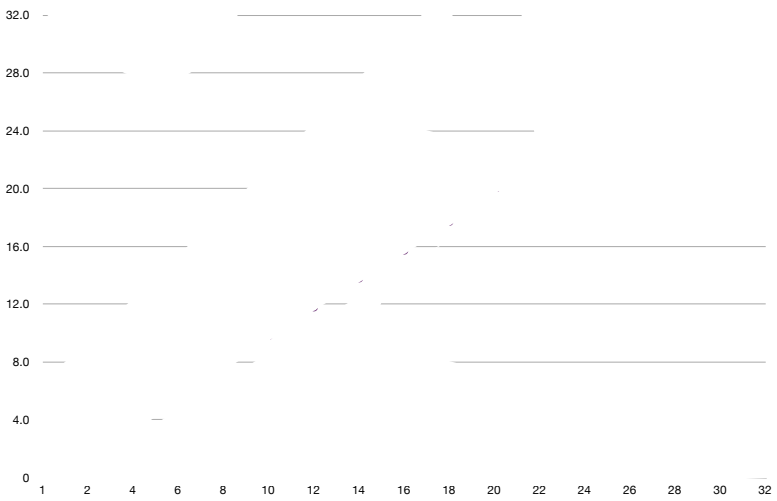

Fig. 7 Speed-up related to the number of threads. Test databases ranging from 50 to 300 attributes with 500 lines, on COYOTE.

5.2 Memory Limits

The major difficulty for this kind of problems is memory consumption, in particular because a very large number of candidates - equal to the number of frequent item pairs - have to be processed at each level. In order to illustrate this situation, we applied our program on a 30-line and 1500-attribute database. We found 1.6M frequent items at level 2 but at the next stage, 5M more new frequent itemsets were found having used just 10,000 frequent items of level 2. According to memory consumption pace at this stage, almost 150GB of RAM would have been necessary to store all level 3 frequent items. As we will explain in the conclusion and perspective section, these limitations lead us to explore other approaches like parallelization on distributed memory systems (*clusters*) that scale better on a memory level.

6 Conclusion and Perspectives

In this paper, we propose an original parallel approach to mine large numeric databases for gradual patterns like *the oldest a people, the higher his/her salary*. Mining these rules is indeed very difficult as the algorithms must perform many time-consuming operations to get the frequent gradual patterns from the databases.

In order to tackle this problem, our method intensively uses the multiple processors and cores that are now available on computers. Parallelism has recently gained much attention and is one of the most promising solution to manage huge real world problems. The experiments performed show the interest of our approach, by leading to quasi-linear speed-ups on problems that were previously very time-consuming or even impossible to manage, especially in the case of databases containing a lot of attributes.

This work opens many perspectives, not only based on technical improvements depending on ad-hoc architectures of the machines, but also based on other data mining paradigms. First, we will consider closed gradual patterns in order to cut down the computation runtimes. Second, we will consider pattern growth techniques [Han et al., 2000] in order to speed up both the sequential and parallel runtimes, and to avoid consuming too much memory. Finally, we will study the use of another parallel framework: clusters (including clusters of multi-core machines in order to benefit from both architectures).

Acknowledgements

The authors would like to acknowledge Lisa Di Jorio for providing the source code of the implementation of the GRITE algorithm [Di Jorio et al., 2009].

References

[Agrawal and Shafer, 1996] Agrawal, R. and Shafer, J. C. (1996). Parallel mining of association rules. *IEEE Trans. Knowl. Data Eng.*, 8(6):962–969.

[Agrawal and Srikant, 1994] Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, pages 487–499.

[Berzal et al., 2007] Berzal, F., Cubero, J.-C., Sanchez, D., Vila, M.-A., and Serrano, J. M. (2007). An alternative approach to discover gradual dependencies. *Int. Journal of Uncertainty, Fuzziness and Knowledge-Based Systems (IJUFKS)*, 15(5):559–570.

[Buehrer et al., 2006] Buehrer, G., Parthasarathy, S., and Chen, Y.-K. (2006). Adaptive parallel graph mining for cmp architectures. In *ICDM*, pages 97–106.

[Di Jorio et al., 2008] Di Jorio, L., Laurent, A., and Teisseire, M. (2008). Fast extraction of gradual association rules: A heuristic based method. In

Di Jorio et al., 9008 Di 7253(Jorio, 725 (L., 7253(Lau

[Han et al., 2000] Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. In *SIGMOD'00 : Proceedings of the International Conference on Management of Data*, pages 1–12, Dallas, USA.

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

[Laurent et al., 2009] Laurent, A., Lesot, M.-J., and Rifqi, M. (2009). Graank: Exploiting rank correlations for extracting gradual dependencies. In *Proc. of FQAS'09*.

[Liu et al., 2007] Liu, L., Li, E., Zhang, Y., and Tang, Z. (2007). Optimization of frequent itemset mining on multiple-core processor. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1275–1285. VLDB Endowment.

[Lucchese et al., 2007] Lucchese, C., Orlando, S., and Perego, R. (2007). Parallel mining of frequent closed patterns: Harnessing modern computer architectures. In *ICDM*, pages 242–251, Omaha, USA.

[Masseglia et al., 2004] Masseglia, F., Poncelet, P., and Teisseire, M. (2004). Pre-processing time constraints for efficiently mining generalized sequential patterns. In *International Syposium on Temporal Representation and Reasoning*, pages 87–95. IEEE Computer Society.

[Pasquier et al., 1999] Pasquier, N., Yves, Bastide, Y., Taouil, R., and Lakhal, L. (1999). Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24:25–46.

[Tatikonda and Parthasarathy, 2009] Tatikonda, S. and Parthasarathy, S. (2009). Mining tree-structured data on multicore systems. In *VLDB '09: Proceedings of the 35th international conference on Very large data bases*, pages 694–705, Lyon, France.

[Uno, 2005] Uno, T. (2005). Lcm ver. 3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *In Proc. of the ACM SIGKDD Open Source Data Mining Workshop on Frequent Pattern Mining Implementations*, pages 77–86, Chicago, USA.

[Yan and Han, 2002] Yan, X. and Han, J. (2002). gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining*, page 721, Washington, DC, USA. IEEE Computer Society.

[Zaki, 1999] Zaki, M. J. (1999). Parallel sequence mining on shared-memory machines. In *Large-Scale Parallel KDD Systems Workshop, KDD Conference*, pages 161–189, San-Diego, USA.

[Zaki et al., 1997] Zaki, M. J., Parthasarathy, S., Ogihara, M., and Li, W. (1997). Parallel algorithms for discovery of association rules. *Data Min. Knowl. Discov.*, 1(4):343–373.