HAL
open science

Multicore Mining of Correlated Patterns

Christian Ernst, Alain Casali

To cite this version:

Christian Ernst, Alain Casali. Multicore Mining of Correlated Patterns. IMMM 2013, Nov 2013, Lisboa, Portugal. pp 18-23. emse-00921628

HAL Id: emse-00921628

https://hal-emse.ccsd.cnrs.fr/emse-00921628v1

Submitted on 20 Dec 2013

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.

Multicore Mining of Correlated Patterns

Christian Ernst
Ecole des Mines de St Etienne
CMP - Site Georges Charpak
Gardanne, France
ernst@emse.fr

Alain Casali
Aix-Marseille Universit,
CNRS, LIF UMR 7279,
Marseille, France
alain.casali@lif.univ-mrs.fr

***Abstract*—We present a new approach related to the discovery of correlated patterns based on the use of multicore architectures. Our work rests on a full Knowledge Discovery in Databases system allowing one to extract Decision Correlation Rules based on the Chi-squared *criterion* from any database that includes a target column. We use a levelwise algorithm as well as contingency vectors, an alternate and more powerful representation of contingency tables. The goal is to parallelize the extraction of relevant rules by invoking the Parallel Patterns Library which allows a simultaneous access to the whole available cores on modern computers. We finally present first results and performance gains.**

***Keywords*—Data Mining, Decision Correlation Rule, Multicore Architecture, Parallel Pattern Library.**

I. Introduction and Motivation

Innovations in multicore architectures have begun to allow parallelization on inexpensive desktop computers. Many standard software products will soon be based on recent parallel computing concepts implemented on such hardware. Consequently, there is a growing interest in the field of parallel data mining algorithms, especially in Association Rules Mining (ARM). By exploiting multicore architectures, parallel algorithms may improve both execution time and memory requirement issues, two main objectives of ARM.

Independently of this framework, we developed a Knowledge Discovery in Databases (KDD) system based on the discovery of Decision Correlation Rules (DCRs) with large and specialized databases [1]. The rules are functional in semiconductor fabs: the goal is to discover the parameters that have the most impact on a specific parameter, the yield of a given product. DCRs are close to Association Rules, but present huge technical differences. After implementing DCRs using "conventional sequential algorithms", we adapted our approach to multicore implementation issues.

This paper is organized as follows: In Section II, we expose current aspects of multicore programming. Section III is dedicated to related work: we present (i) an overview of ARM over a multicore architecture and (ii) what DCRs are. Section IV describes the concepts used for multicore decision rules mining and our algorithms. In Section V, we show first results of experiments. Last Section summarizes our contribution and outlines some research perspectives.

II. New features in multicore programmation

In the last two decades, parallelization on personal computers has consisted to develop multithreaded code layers. What required complex co-ordination of threads, due to the interweaving of shared data processing. Although threaded applications added limited performance on single-processor machines, the extra overhead of development has been difficult to justify. But with Intel and AMD introducing commercially multicore chips in 2005, non exploiting the resources provided by multiple cores will now quickly reach performance ceilings. At that last date, no simple software environment able to take advantage of the different processors have been simultaneously proposed. New opportunities appeared in 2010, presented in the C++ language used in our developments.

Multicore processing influenced actual computational software development. Many modern languages do not support multicore functionality. Therefore, different conceptual models deal with the problem, such as using a coordination language (programming libraries and/or higher order functions). Users program using these abstractions, and an "intelligent" compiler then chooses the best implementation based on the context [2]. Cilk++, OpenMP, OpenHMPP, TBB, *etc.*, are examples of such models having been recently proposed for use on multicore platforms. A comparison of some OpenX approaches can be found in [3]. Nevertheless, the majority of these models rests on an intelligent transformation of general code into multithreaded code.

A novel albeit simple idea was proposed by the Open Multiprocessing consortium in 1997, based on the fact that looping functions are the key area where splitting parts of a loop across all available hardware resources may increase application performance. The OpenMP Architecture Review Board became an API that supports shared memory multiprocessing programming now also in C++. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. In order to schedule a loop across multiple threads, the OpenMP `pragma` directives were introduced in 2005 to explicitly relay to the compiler more about the transformations and optimizations that should take place. To illustrate our purpose, we compute in parallel an approximation of the value of π using a Riemann Zeta function ($\pi^2/6 = 1/1^2 + 1/2^2 + 1/3^2 + ...$, see Listing 1):

```
double pi2 = 0.0;
#pragma omp for
for (int i = 1; i < 1000000; i++) {
 #pragma omp atomic
 pi2 += 6.0 / (i * i);
}
```

Listing 1: Computing π using basic multithreaded parallelization

The first directive requests that the *for* loop should be executed on multiple threads, while the second is used to prevent multiple simultaneous writes to the *pi2* variable.

The example also shows the limits of parallelism. It is widely agreed that applications that may benefit from using more than one processor necessitate (i) operations that require a substantial amount of processor time, measured in seconds rather than milliseconds and (ii) one or more loops of some kind, or operations that can be divided into discrete but significant units of calculation that can be executed independently of one another. So the chosen example with a single instruction at each iteration does not fit parallelization, but is used nevertheless to illustrate in a simple way the new features introduced by actual multicore programming techniques.

These were first developed by Microsoft through an own "parallel" approach in the 2000s. Since 2010, and the relevant versions of the .NET framework and Visual Studio, Microsoft enhanced support for parallel programming by providing a runtime tool and a class library among other utilities. The library is composed of two parts: Parallel LINQ (PLINQ), a concurrent query execution engine, and Task Parallel Library (TPL), a task parallelism component of the .NET framework. What is particularly advanced is that this component entirely hides the multithreading activity on the cores: the job of spawning and terminating threads, as well as scaling the number of threads according to the number of available cores, is done by the library itself. The main concept is here a Task, which can be executed independently.

The Parallel Patterns Library (PPL) is the corresponding available tool in the Visual C++ environment, and is defined within the Concurrency namespace. The PPL operates on small units of work (Tasks), each of them being defined by what is called a λ expression (see below). The PPL defines almost three kinds of facilities for parallel processing: (i) algorithm templates for parallel operations, (ii) class templates for managing shared resources, and (iii) class templates for managing and grouping parallel tasks.

Listing 2 rewrites our example using the *parallel_for* algorithm, equivalent to a *for* loop that executes loop iteration in parallel on multiple cores:

```
float pi2 = 0;
parallel_for(1, 1000000, [&pi2](long n)
 {
   // share <pi2> between the cores
   pi2 += 6.0 / (n * n);
 }
);
```

Listing 2: Computing π using multi-core parallelization

The PPL also proposes the *parallel_for_each* algorithm (for repeated operations on a STL container), and the *parallel_invoke* algorithm (which executes a set of two or more independent Tasks in parallel).

As mentioned when discussing the OpenMP `pragma` directives, if the computation on each iteration in the *parallel_for* is very short and it is the case here, there will be important overhead in allocating the task to a core on each iteration, which may severely erode any reduction in execution times. This will also be the case if the overall loop integrates important shared resources management, as will be shown in Section IV.

The second main novelty introduced by the PPL is the use of λ expressions: A computational model invented by Alonzo Church in the 1930s, which directly inspired both the syntax and the semantics of most functional programming languages [4]. The λ calculus in its most basic form has two operations: (i) Abstractions, which correspond to anonymous functions, and (ii) Applications, which exist to apply the function. Anonymous functions are often called "lambdas", "lambda functions" or "lambda expressions": They remove all need for scaffolding code, allowing a predicate function to be defined in-line in another statement.

The syntax of a λ function is reasonably straight-forward, of the form:

```
[lambda-capture] (parameter-list) {->} return-type
    {statement-list}
```

In our example (Listing 2), the element of the lambda in the square brackets is called the capture specification: It relays to the compiler that a lambda function is being created and that the local variable *pi2* is being captured by reference. The final part is the function body.

Lambdas behave like function objects (as did previously functors), except for that we cannot access the class that is generated to implement a lambda in any way other than using the lambda. Consequently, any function that accepts functors as arguments will accept lambdas, but any function only accepting function pointers will not.

These and many other features of λ functions have been included in the C++11 language norm, allowing a more declarative programming style, taking for example advantage of STL algorithms in a much streamlined and cleaner form. λ functions allow the inline definition of a function body in the code section in which it is to be logically used. As well as providing strong hints to the compiler about potential real time optimizations, λ functions make discerning the intent about what a section of code is doing much easier.

III. Related Work

Due to the variety of the algorithms (and their specific internal data structures) no general model allowing parallel ARM computation exists. Main techniques based on A-Priori algorithms [5] are described in Section 3.A. Other multicore ARM approaches are based either on vertical mining [6] or on FP-Growth [7]. Each model consists in an multicore optimized architecture built upon specific thread managers [4]. Finally, Section 3.B presents the main results about what Decision Correlation Rules are and how can we compute them using a single processor.

A. A-Priori based algorithms

Most of the parallel ARM algorithms are based on parallelization of A-Priori that iteratively generates and tests

candidate itemsets from length 1 to k until no more frequent itemsets are found. These algorithms can be categorized into *Count Distribution*, *Data Distribution* and *Candidate Distribution* methods [8]. The *Count Distribution* method divides the database into horizontal partitions, that are independently scanned, in order to obtain the local counts of all candidates on each process. At the end of each iteration, the local counts are summed up a into the global counts so that frequent itemsets can be found. The *Data Distribution* method partitions both the database and the candidate itemsets in the main memory of parallel machines. Since each candidate is counted by only one process, all processes have to exchange database partitions during each iteration in order, for each process, to obtain the global counts of the assigned candidate itemsets. The *Candidate Distribution* method also partitions candidate itemsets but replicates, instead of partitioning and exchanging, the database transactions. Thus, each process can proceed independently.

B. Decision Correlation Rules

Brin et al. [9] have proposed the extraction of correlation rules using the Chi-Squared (χ^2) statistic instead of the support and the confidence measures. The χ^2 (i) is a more significant measure in a statistical way than an association rule, (ii) takes into account the presence but also the absence of the items and (iii) is non-directional, highlighting thus more complex existing links than implications. A correlation rule is represented by an itemset.

Let r be a binary relation over a set of items $\mathcal{R} = \mathcal{I} \cup T$. $\mathcal{I}$ represents the items of the binary relation used as analysis *criteria* and $\mathcal{T}$ is a target attribute which may not necessarily have a value. The computation of the value for the χ^2 function for an item $X \subseteq \mathcal{R}$ is based on its contingency table. In order to simplify the notation, we introduce, in a first step, the lattice of the literalsets associated with $X \subseteq \mathcal{R}$. This set of cardinality $|X|$ contains all the literalsets that can be built up given X.

Definition 1 (Literalset Lattice): Let $X \subseteq \mathcal{R}$ be a pattern, we denote by $\mathbb{P}(X)$ the literalset lattice associated with X. This set is defined as follows: $\mathbb{P}(X) = \{Y\overline{Z}$ such that $X = Y \cup Z$ and $Y \cap Z = \emptyset\} = \{Y\overline{Z}$ such that $Y \subseteq X$ and $Z = X \backslash Y\}$.

Definition 2 (Contingency Table): For a given pattern X, its contingency table, noted $CT(X)$, is a $2^{|X|}$ matrix. Each cell yields the support of a literalset $Y\overline{Z} \in \mathbb{P}(X)$: the number of transactions including Y and containing no 1-item of Z.

In order to compute the value of the χ^2 function for a pattern X, we apply the following formula:

$$\chi^2(X) = \sum_{Y\overline{Z} \in \mathbb{P}(X)} \frac{(Supp(Y\overline{Z}) - E(Y\overline{Z}))^2}{E(Y\overline{Z})} \qquad (1)$$

Brin et al. [9] have shown that there is a single degree of freedom between the items. A table giving the centile values in function of the χ^2 value for X can be used in order to obtain the correlation rate for X.

Definition 3 (Correlation Rule): Let $MinCor$ (≥ 0) be a given threshold and $X \subseteq \mathcal{R}$ a pattern. If $\chi^2(X) \geq MinCor$, then X is a valid correlation rule. If X contains an item of $\mathcal{T}$, then the obtained rule is called a Decision Correlation Rule (DCR).

Moreover, in addition to the previous constraint, the Cochran *criteria* [10] are used to evaluate whether a correlation rule is semantically valid: all literalsets of a contingency table must have an expectation value different to zero and 80% of them must have a support larger than 5%. This last *criterium* has been generalized as follows: $MinPerc$ of the literalsets of a contingency table must have a support larger than $MinSup$, where $MinPerc$ and $MinSup$ are given thresholds.

Definition 4 (Equivalence Class): We denote by $[Y\overline{Z}]$ the equivalence class associated with the literal $Y\overline{Z}$: it contains the set of transaction identifiers including Y and containing no value of Z ($i.e.$, $[Y\overline{Z}] = \{i \in Tid(r)$ such that $Y \subseteq Tid(i)$ and $Z \cap Tid(i) = \emptyset\}$).

Definition 5 (Contingency Vector): Let $X \subseteq \mathcal{R}$ be a pattern. The contingency vector of X, denoted $CV(X)$, groups the set of the literalset equivalence classes belonging to $\mathbb{P}(X)$ ordered according to the lectic order.

Since the union of the equivalence classes $[Y\overline{Z}]$ of the literalset lattice associated with X is a partition of the Tids, we ensure that a single transaction identifier belongs only to one single equivalence class. Consequently, for a given pattern X, its contingency vector is an exact representation of its contingency table. To derive the contingency table from a contingency vector, it is sufficient to compute the cardinality of each of its equivalence classes. The following proposition shows how to compute the CV of the $X \cup A$ pattern given the CV of X and the set of Tids containing pattern A.

Proposition 1: Let $X \subseteq \mathcal{R}$ be a pattern and $A \in \mathcal{R} \backslash X$ a 1-item. The contingency vector of the $X \cup A$ pattern can be computed given the CVs of X and A as follows:

$$CV(X \cup A) = (CV(X) \cap [\overline{A}]) \cup (CV(X) \cap [A]) \qquad (2)$$

In order to mine DCRs, we have proposed [1] the LHS-CHI2 algorithm (see Algorithm 1) based both (i) on a double recursion in order to browse the search space according to the lectic order and (ii) on CVs.

The `CREATE_CV` function is an implementation of formula 2, while the `CtPerc` predicate checks the relaxed Cochran *criteria*.

IV. Parallel Extraction of Correlated Patterns

The development of multicore applications raises two difficulties in terms of (i) application design and (ii) shared resource management. The second aspect is rather "normal" when coping with parallelism. And will constitute the aim of this section, illustrated through specific mechanisms provided by the PPL. But application design must not be underestimated because, amongst other points, of its impact on resource management. Parallelizing existing algorithms is an important consideration as well [2]. The use of recursive

Algorithm 1: LHS-CHI2 Algorithm.

input : X and Y two patterns
output: $\{ Z \subseteq X$ such that $\chi^2(Z) \geq MinCorr\}$

1 **if** $Y = \emptyset$ ***and*** $\exists t \in \mathcal{T} : t \in X$ ***and*** $|X| \geq 2$ ***and*** $\chi^2(X) \geq MinCorr$ **then**
2 | **Output** X, $\chi^2(X)$
3 **end**
4 $A := max(Y)$;
5 $Y := Y \backslash \{A\}$;
6 LHS-CHI2 (X,Y) ;
7 $Z := X \cup \{A\}$;
8 CV(Z) := CREATE_CV (CV(X),Tid(A)) ;
9 **if** $CtPerc\ (CV(Z), MinPerc, MinSup)$ ***and*** $|Z| \leq MaxCard$ **then**
10 | LHS-CHI2 (Z,Y) ;
11 **end**

algorithms in a multicore environment is here a sufficient challenge. This because recursion cannot be measured in terms of number of loops to perform: We first tried to replace the recursive calls by calls to appropriate threads, which quickly appeared "impossible'. Another approach was based on the well known fact that each recursive algorithm can be rewritten in a iterative way. However, the *while* loop used to run over the used stack may not be evaluated in terms of a *for* loop due to the absence of explicit boundaries.

In order to solve the problem, we recalled that we first compared our LHS-CHI2 algorithm to a LEVELWISE one, based on the same monotone and anti-monotone constraints but which did not include Contingency Vectors management. The main reason of the obtained performance gains is that pruning the search space using the lectic order is much more "elegant" than using the LEVELWISE order but has no impact nor on the results nor on the performances. On the other hand, generating the candidates at a given level is a bounded task, limited by the number of existing 1-items. So we decided to (i) use the LEVELWISE order to prune in a parallel way the search space and (ii) to keep the CVs in order to manage the constraints.

The corresponding result is presented hereafter through different functions. The overall algorithm (see Listing 3), called *PLW_Chi2*, where PLW stands for Parallel LEVELWISE, demonstrates first parallel features of our method in order to generate (and then to test) the candidates.

In order to simplify the notations, the following Listings use uc, ui, ul, us to substitute to, respectively, $unsigned\ char$, $unsigned\ int$, $unsigned\ long$ and $unsigned\ short$ standard declarations.

```
void PLW_Chi2 (us X[], us sX, us sI)
// X[] : set of computed 1-items
// sX : number of valid 1-items within X[]
// sI : total number of 1-items in X[]
{
 // number of candidates at level cl and (cl+1)
 ul cit, nit;
 cit = sX;
 for (uc cl = 2; cl <= MaxLv && cit > 0; cl++)
 {
   nit = 0L;
   T_Res aRes;
   combinable<ul> lnit;
   parallel_for(0u,(ui) cit,[cl, X, sX, &aRes, &lnit](int i)
    {
     dowork_level (cl, X, sX, i, sI, &aRes);
     lnit.local() += aRes.nit;  // ...
    });
    nit = lnit.combine(plus<ul>());
    // ...
    cit = nit;
    update_shared_resources ();
 }
}
```

Listing 3: The simplified PLW_CHI2 method

Parallelization takes place at each *cl* level of the LEVELWISE search algorithm. The number of launched Tasks at level *cl* directly depends of the number of existing candidates at level *(cl - 1)*, *e.g.*, *cit*. Each Task corresponds to a call to the *dowork_level ()* function, which performs the work it is intended to do (see below), and collects some statistics during the call through the *aRes* object. Let us mention here that database access is performed through global objects.

In this paragraph, we only focus on the signification of a particular statistic, the *lnit* member of the *aRes* object: It sums the number of discovered candidates to be examined at the next level. Because each Task computes its own candidates for the next level, the method has to pay attention to the possible interference which could take place during the overall parallel computation on such a "shared" variable, which can be seen as an aggregation pattern. A two-phase approach is therefore used: First, partial results are locally computed on a per-Task basis. Then, once all of the per-Task partial results are at disposal, the results are sequentially merged into one final accumulated value. The PPL provides the *combinable* class data structure that creates per-Task local results in parallel, and merge them as a final sequential step. In the above code, the final accumulated object is the *lnit* object, which decomposes into local to each Task *lnit.local()* sub-objects. After the *parallel_for* loop achieves, the final sum is produced by invoking the *combine()* method on the global object.

Listing 4 partially shows the implementaton of the *dowork_level ()* function:

```
void dowork_level (uc nc, us pX[], us X, ul nel,
    us sIX, T_Res& pRes)
{
 us vmin, tCand[MaxLv + 1]; // a candidate
 ul j, k;
 uc *theCV;  // a CV
 // other declarations and initializations
 // get current itemset
 vmin = get_pattern (nc, tCand, pX, cX, nel, sIX);
 //j is the index of the first 1-item to add
 j = 0;
 while (j < cX && pX[j] <= vmin)  j++;
 for (k = j; k < cX; k++)
 {
  // add a 1-item to current itemset to produce a candidate
  tCand[0] = pX[k];
  // compute its CV if the constraints are valid
  theCV = compute_CV (tCand, nc,  ...);
  // memorize the candidate and add it to results
  // if applies
  store_CV (tCand, nc, theCV, pRes, ...);
  // update statistics
  pRes.nit++;  //...
 }
}
```

Listing 4: Code of main method called by PLW_CHI2

We shall not enter into the implementation details of this function. First because the code is most C likely and is easy to understand. And second because it does not include any specific parallel or shared memory features. So, we shall only explain its overall functionalities. The *for* loop is used to produce all the candidates at the current stage (the *tCand* variable). This is done by "adding" the possible existing 1-items to the base itemset managed by the function, and identified by the *nel* "number" (we shall discuss this aspect later). Once having generated such a candidate, we verify first if the different constraints underlying to our method are verified or not by the candidate. If it is the case, we compute its Contingency Vector using the *compute_CV ()* function. We second (try to) memorize the candidate in order to reuse it at the next level, and we add the candidate to the results if it contains one item of the target attribute.

Finally, the *store_CV ()* function (see Listing 5) describes, in a very simplified way, a specific section of code dedicated to the storage of results:

```
bool store_CV (us X[], us cardX, ...)
{
 //add X to the result file if X contains the target
 if (...)
 {
  critical_section cs;
  cs.lock();
  if (...)
    write_llhsp_to_file (X, cardX, ...);
  else
    write_pattern_to_file (X, cardX, ...);
  cs.unlock();
  }
// ...
}
```

Listing 5: Sharing ressources with the PPL using a critical section

Let us focus on the functionality involved in the first *if* statement: k-itemsets verifying the whole defined constraints and including an item belonging to the target column must be included into the results. This is done through their insertion into data files (one is associated to each value of k). During the parallellization process, each Task may write to one of these files each time it discovers a new valid itemset. What raises another shared resource problem, addressed by the PPL by the use of critical sections (a well-known concept in multithreading developments), as shown in the above code. When encountering such an instruction at run-time, the OS will not authorize any other Task to execute before the "lock" has been released.

To finish this Section, we explain the way we manage the memorization of candidates and associated information such as CVs. The main shared data structure in our developments is a tree storing the k-itemsets of "interest" (see Listing 6). The corresponding node structure, given in the C language, is:

```
typedef struct pattern_node
{
 unsigned short *Pattern; // the pattern
 unsigned char *pCV; // pointer to the CV
 T_NM *brother; // pointer to next node at same level
 T_NM *son; // pointer to next node at lower level
 ...
} T_NM;
```

Listing 6: Main index structure used by the PLW_CHI2 method

Each time a Task discovers a candidate verifying the whole constraints, the candidate is inserted into the tree. The insertion by itself uses the critical section concept we just introduced. Because the stored itemsets (patterns) are lexicographically organized within the tree, each of them can be referred to by a node number (what explains the *nel* "number" introduced above). Finally, after evaluating the candidates, the exploring process will retain them or not. In the latter case, the tree structure may be garbaged, which is done by the *update_shared_resources ()* function called at the end of our global *PLW_Chi2* method.

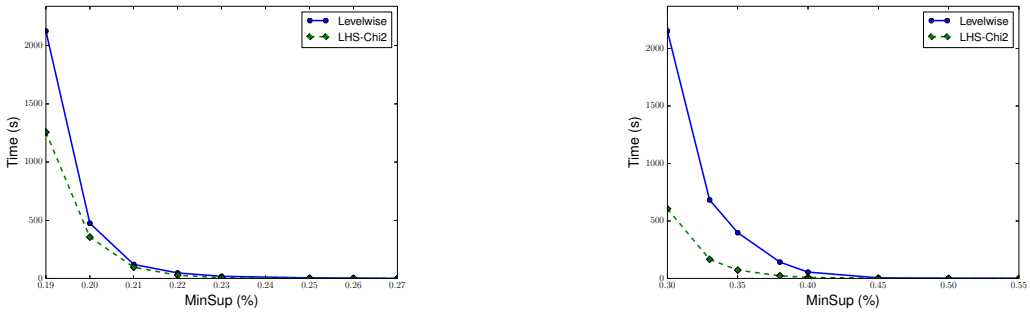
V. Experimental Analysis

As briefly mentioned in the Introduction Section, this work has been initially applied on raw data measurement files provided by two industrial manufacturing partners in the area of Microelectronics: STMicroelectronics (STM) and ATMEL (ATM). The results of the realized experimental series are presented on 2 plans to be followed. They are associated with an analysis of 2 files among those supplied by both manufacturers. The first one (STM) contains 1241 columns and 296 lines. The second (ATM) consists of 749 columns and 213 lines. We chose a target attribute among a few possible columns. In both cases, the presented diagrams show the execution times of two methods when $MinSup$ varies while $MinPerc$ (0.34 for the STM file and 0.24 for the ATM one) and $MinCor$ (1.6 resp. 2.8) are fixed (see section III-B for more details).

Figures 1(a) and 1(b), extracted from [1], show the execution times of a standard LEVELWISE algorithm and the LHS-CHI2 algorithm on a non core computer (a HP Workstation with a 1.8 GHz processor and 4 Gb RAM, working under a Windows XP 32 bits OS). The difference between the two methods is that the LEVELWISE method uses no contingency vectors but standard computation of contingency tables. As the graphs point it out, the response times of the LHS-CHI2 method are between 30% and 70% better than LEVELWISE.

Figures 2(a) and 2(b) show the same execution times using the LHS-CHI2 algorithm and the presented PLW-CHI2 algorithm on a 4 core computer (a DELL Workstation with a 2.8 GHz processor and 12 Gb RAM working under the Windows 7 64 bits OS).

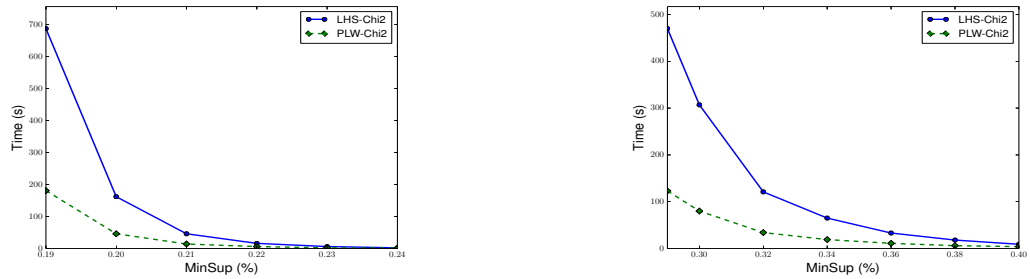
As it is easy to understand, the LHS-CHI2 method works here about two times faster on the multicore architecture, this is not because of the number of cores (which are not used) but because of the computer basic enhanced capabilities. When regarding to the performances of the PLW-CHI2 method, there is a gain factor of about 3.5, which is to compare to the number of available cores, which is 4. In other words, the parallelization of the LHS-CHI2 algorithm raises performance gains practically equals to the number of cores, the (little) loss being due to the shared memory management issues. Let us underline here that we do not integrate in these amounts that one core remains dedicated to system management ...



(a) Results for STM File

(b) Results for ATM File

Fig. 1: Execution times with a single processor.



(a) Results for STM File

(b) Results for ATM File

Fig. 2: Execution times with 4 cores.

VI. Conclusion and Future Work

In this paper, we present a new approach to discover correlated patterns, based on the usage of multicore architectures. Our approach is based on two concepts: Contingency Vectors, an alternate representation of contingency tables, and the Parallel Patterns Library. One advantage of Contingency Vectors is that they allow the Chi-squared computation of a k-itemset directly from one of its subsets. However, the usage of this library has a disadvantage: The parallelization of recursive algorithms is hard (we do not control neither the number of cores, nor the depth of the tree), even if we derecursify the algorithm. That is why we have chosen to implement a LEVELWISE algorithm which implements these two concepts. Experiments are convincing because our *PLW_Chi2* algorithm gains a time factor of about 3.5 (when using 4 cores) in comparison with the recursive version. For future works, we intend to develop a new version of the recursive algorithm using Contingency Vectors and to build our own thread/core manager.

References

[1] A. Casali and C. Ernst, "Discovering correlated parameters in semiconductor manufacturing processes: A data mining approach," *Semiconductor Manufacturing, IEEE Transactions on*, vol. 25, no. 1, pp. 118–127, 2012. I, III-B, V

[2] J. Darlington, M. Ghanem, Y. ke Guo, and H. W. To, "Guided resource organisation in heterogeneous parallel computing," 1996. II, IV

[3] G. Krawezik and F. Cappello, "Performance comparison of mpi and openmp on shared memory multiprocessors," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 1, pp. 29–61, 2006. II

[4] H. P. Barendregt, "Functional programming and lambda calculus," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 321–363. II, III

[5] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 1994, pp. 487–499. III

[6] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "Parallel algorithms for discovery of association rules," *Data Min. Knowl. Discov.*, vol. 1, no. 4, pp. 343–373, 1997. III

[7] E. Li and L. Liu, "Optimization of frequent itemset mining on multiple-core processor," in *VLDB*, C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C.-C. Kanne, W. Klas, and E. J. Neuhold, Eds. ACM, 2007, pp. 1275–1285. III

[8] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. Knowl. Data Eng.*, vol. 8, no. 6, pp. 962–969, 1996. III-A

[9] S. Brin, R. Motwani, and C. Silverstein, "Beyond market baskets: Generalizing association rules to correlations," in *SIGMOD Conference*, 1997, pp. 265–276. III-B, III-B

[10] D. Moore, "Measures of lack of fit from tests of chi-squared type," *Journal of statistical planning and inference*, vol. 10 (2), no. 2, pp. 151–166, 1984. III-B