HAL
open science

Extracting correlated parameters on multicore architectures

Christian Ernst, Alain Casali

To cite this version:

Christian Ernst, Alain Casali. Extracting correlated parameters on multicore architectures. CD-ARES 2013, Sep 2013, Regensburg, Italy. pp 118-133. emse-00921635

HAL Id: emse-00921635

https://hal-emse.ccsd.cnrs.fr/emse-00921635v1

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.

Extracting Correlated Patterns on Multicore Architectures

Alain Casali[1] and Christian Ernst[2]

[1] Laboratoire d'Informatique Fondamentale de Marseille (LIF),
CNRS UMR 6166, Aix Marseille Université
IUT d'Aix en Provence, Avenue Gaston Berger,
13625 Aix en Provence Cedex, France
alain.casali@lif.univ-mrs.fr
[2] Ecole des Mines de St Etienne, CMP - Georges Charpak
880 avenue de Mimet, 13541 Gardanne
ernst@emse.fr

Abstract. In this paper, we present a new approach relevant to the discovery of correlated patterns, based on the use of multicore architectures. Our work rests on a full KDD system and allows one to extract Decision Correlation Rules based on the Chi-squared *criterion* that include a target column from any database. To achieve this objective, we use a levelwise algorithm as well as contingency vectors, an alternate and more powerful representation of contingency tables, in order to prune the search space. The goal is to parallelize the processing associated with the extraction of relevant rules. The parallelization invokes the PPL (Parallel Patterns Library), which allows a simultaneous access to the whole available cores / processors on modern computers. We finally present first results on the reached performance gains.

1 Introduction and Motivation

In the past couple of years, innovations in hardware architecture, like hyperthreading capabilities or multicore processors, have begun to allow parallel computing on inexpensive desktop computers. This is significant in that standard software products will soon be based on concepts of parallel programming implemented on such hardware, and the range of applications will be much broader than that of scientific computing (the main area for parallel computing). Consequently, there is a growing interest in the research 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, the main objectives of the data mining field.

Independently of this framework, we soon developed a KDD system based on the discovery of Decision Correlation Rules with large and specialized databases (Casali and Ernst, 1). These rules are functional in semiconductor fabrication capabilities. The goal is to discover the parameters that have the most impact on a specific parameter, the yield of a given product ... Decision Correlation

Rules (DCRs) are similar to Association Rules. But, as it will be shown, there are huge technical differences, and the overall computation times of DCRs are in the end much better. Furthermore, after implementing DCRs using "conventional sequential algorithms", we decided to adapt this approach to multicore implementation possibilities.

This paper is organized as follows. In Section 2, we recall current features of multicore programming. Section 3 is dedicated to related work: we present (*i*) an overview of Association Rules Mining over a multicore architecture and (*ii*) what Decisional Correlation Rules are. Section 4 describes the concepts used for multicore decision rules mining and our algorithm. In Section 5 we show the results of first experiments. Finally, in the final Section, we summarize our contribution and outline some research perspectives.

2 Recent advances in multicore programmation

Multicore processing is not a new concept, however only recently has the technology has become mainstream with Intel or AMD introducing commercially available multicore chips in 2008. At that date, no software environment able to take advantage simultaneously of the different existing processors had been proposed, let alone produced for the commercial market.

The situation radically changed in 2010. We present in this section these new opportunities, showing the covered aspects in the C++ language used in our developments. We first introduce what the approaches towards parallelization were until only a few years ago. We then present quickly the Lambda Calculus, a new programming paradigm integrated into the last C++ norm and essential to understand multicore prorammation. And we finally expose the PPL environment, which allows effective multicore programmation in a simple way.

2.1 Parallelization issues on modern desktop computers

For about twenty years, parallelizing tasks on personal computers consisted essentially in the development of multithreaded code, by adding or not higher abstraction levels on the base threaded layers. This often required complex coordination of threads, and introduced difficulties in finding bugs due to the interweaving of processing of data shared between the threads. Although threaded applications added limited amounts of performance on single-processor machines, the extra overhead of development was difficult to justify on the same machines.

But given the increasing emphasis on multicore chip architectures, developers unable to design software to fully exploit the resources provided by multiple cores will now quickly reach performance ceilings.

Multicore processing has thus affected the ability of actual computational software development. Many modern languages do not support multicore functionality. This requires the use of specialized libraries to access code written

in languages such as C. There are different conceptual models to deal with the problem, for example using a coordination language and program building blocks (programming libraries and/or higher order functions). Each block may have a different native implementation for each processor type. Users simply program using these abstractions, and an intelligent compiler then chooses the best implementation based on the context (Darlington, 2).

Many parallel programming models have been recently proposed (Cilk++, OpenMP, OpenHMPP, ...) for use on multicore platforms. Intel introduced a new abstraction for C++ parallelism called TBB. Former and more organization oriented research efforts include the Codeplay Sieve System, Cray's Chapel, Sun's Fortress, and IBM's X10. A comparison of some OpenXXX approaches can be found in (Hamidouche and al., 3).

But the majority of these models roughly bases on the intelligent transformation of general code into multithreaded code. A new, even if simple idea, has been proposed by the OpenMP consortium in FORTRAN 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 Open Multiprocessing Architecture Review Board (or OpenMP ARB) became an API that supports shared memory multiprocessing programming 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. The example chosen to illustrate our purposes is the computation of the sum of an integer vector:

```
vector<int> v = ...;
int sum = 0;
#pragma omp for
for (int i = 0; i < v.size(); i++) {
   #pragma omp atomic
   sum += v[i];
}
```

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 *sum* variable ...

This approach has then be adapted to multicore programmation, as illustrated hereafter. Let us previously underline that parallelism also has its limits. It is widely admitted that applications that may benefit from using more than one processor necessitate (*i*) operations that require substantial amount of processor time, measured in seconds (or larger denominations) 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.

2.2 A new programming paradigm: the Lambda Calculus

The example above without parallelization issues can be re-written as follows:

```
vector<int> v = ...;
int sum = 0;
for (int i = 0; i < v.size(); i++){
   sum += v[i];
}
```

A declarative looping technique generalized the syntax for STL containers: the *for_each* function has been introduced in a first C++ 0x draft, so the code can be re-written in:

```
vector<int> v = ...;
int sum = for_each(v.begin(), v.end(), Adder());
```

The third argument is a functor: a class overloading the () operator. In our example, the functor has to perform the addition. This approach did not result in much success for many reasons, the two main of which include: (*i*) developers must implement the functor, but defining a whole class in order to finally execute a single instruction is a bit verbose; (*ii*) using a named function is sub-optimal because confusion may arise with the "C" pointer function syntax.

Recognizing these shortcomings, the C++ draft provided a simplified syntax based on Lambda functions. The associated formalism is a computational model invented by Alonzo Church in the 1930s, which directly inspired both the syntax and the semantics of most functional programming languages (Mitchell, 4). The basic concept of the λ calculus is the "expression", recursively defined as follows:

<expression>:= <name>— <function>— <application>
<function>:= λ <name>.<expression>
<application>:= <expression><expression>

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. Abstraction is performed using the λ operator, giving the lambda calculus its name. Anonymous functions are often called "lambdas", "lambda functions" or "lambda expressions": these remove all the need for the scaffolding code, and allow a predicate function to be defined in-line in another statement. The current example can thus be re-written as follows:

```
vector<int> v = ...;
int sum = 0;
for_each(v.begin(), v.end(),[&sum](int x) {sum += x;}
);
```

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

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

In the example, the first 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 *sum* is being captured by reference. The final part is the function body.

Therefore, lambdas behave like function objects, except for 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 much more features of lambda functions have been included in the C++11 language norm, allowing a more declarative programming style, taking advantage of (STL) algorithms in a much simpler and cleaner form. Lambda 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, lambda functions make discerning the intent about what a section of code is doing much easier.

2.3 Multicore programmation using the PPL

In a different way than the OpenMP consortium, Microsoft developed its own "parallel" approach in the 2000s within an internal work group called Parallel Extensions. 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. This component hides entirely the multi-threading 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 a λ calculus expression. The PPL defines almost three kinds of facilities for parallel processing: (*i*) templates for algorithms for parallel operations, (*ii*) class templates for managing shared resources, and (*iii*) class templates for managing and grouping parallel tasks.

The library provides essentially three algorithms defined as templates for initiating parallel execution on multiple cores:

- The *parallel_for* algorithm is the equivalent of a *for* loop that executes loop iteration in parallel,
- The *parallel_for_each* algorithm executes repeated operations on a STL container in parallel,

– The *parallel_invoke* algorithm executes a set of two or more independent Tasks in parallel, in the sense of different *programs* within the same runtime environment.

Our current example re-written using parallel capabilities:

```
vector<int> v = ...;
int sum = 0;
parallel_for_each(v.begin(), v.end(), [&sum](int x)
 {
   // make sure <sum> is shared between the "cores"...
   sum += x;
 }
);
```

Let us underline again that if the computation on each iteration is very short, there will be inevitably 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 4. This means that upgrading from sequential to parallel computing must be done very carefully.

3 Related Work

Due to the variety of the algorithms (and their specific internal data structures) there does not exist any general model allowing parallel ARM computation. Main techniques are described in Section 3.1. They just are actual models optimized for the usage of multicore architecture. Developers have to write their own thread managers (Herlihy and Shavit, 5). Section 3.2 presents the main results about what Decision Correlation Rules are and how can we compute them using a single processor.

3.1 Association Rules Mining using Multicore Support

Current research can be divided into three main categories: (*i*) adaptation of the A-Priori algorithm, (*ii*) vertical mining, and (*iii*) pattern-growth method.

A-Priori based algorithms: Most of the parallel ARM algorithms are based on parallelization of A-Priori (Agrawal and Srikant, 6) 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 (Agrawal and Shafer, 7), (Han and Kamber, 8). The *Count Distribution* method partitions the database into horizontal partitions, that are independently scanned, in order to obtain the local counts of all candidate 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 utilizes the main memory of parallel machines by partitioning both the database and the candidate itemsets. 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 partition and exchanging, the database transactions. Thus, each process can proceed independently.

Vertical mining: To better utilize the aggregate computing resources of parallel machines, a localized algorithm (Zaki *et. al*, 9) based on parallelization of *Eclat* was proposed and exhibited excellent scalability. It makes use of a vertical data layout by transforming the horizontal database transactions into vertical tid-lists of itemsets. By 1-item, the tid-list of an itemset is a sorted list of IDs for all transactions which contain the 1-itemset. Frequent k-itemsets are organized into disjoint equivalence classes by common $(k-1)$-prefixes, so that candidate $(k+1)$-itemsets can be generated by joining pairs of frequent k-itemsets from the same classes. The support of a candidate itemset can then be computed simply by intersecting the tid-lists of the two component subsets. Task parallelism is employed by dividing the mining tasks for different classes of itemsets among the available processes. The equivalence classes of all frequent 2-itemsets are assigned to processes and the associated tid-lists are distributed accordingly. Each process then mines frequent itemsets generated from its assigned equivalence classes independently, by scanning and intersecting the local tid-lists.

Pattern-Growth Method: The pattern-growth method derives frequent itemsets directly from the database without the costly generation and test of a large number of candidate itemsets. The detailed design is explained in the FP-growth algorithm (Han *et al.*, 10). Basically, it makes use of a frequent-pattern tree structure (FP-tree) where the repetitive transactions are compacted. Transaction itemsets are organized in that frequency-ordered prefix tree such that they share common prefix part as much as possible, and re-occurrences of items/itemsets are automatically counted. Then the FP-tree is pruned to mine all frequent patterns (itemsets). A partitioning-based, divide and conquer strategy is used to decompose the mining task into a set of smaller sub-tasks for mining confined patterns in the so-called conditional pattern bases. The conditional pattern base for each item is simply a small database of counted patterns that co-occur with the item. That small database is transformed into a conditional FP-tree that can be processed recursively (Zaiane *et al.*, 11, Pramudiono and Kitsuregawa 12, Li and Liu 13).

3.2 Decision Correlation Rules

Brin et al. (14) have proposed the extraction of correlation rules. The platform is no longer based on the support nor the confidence of the rules, but on the

Chi-Squared statistical measure, written χ^2. The use of χ^2 is well-suited for several reasons: (i) it is a more significant measure in a statistical way than an association rule, (ii) the measure takes into account not only the presence but also the absence of the items and (iii) the measure is non-directional, and can thus highlight more complex existing links than a "*simple*" implication. Unlike association rules, a correlation rule is not represented by an implication but by the patterns for which the value of the χ^2 function is larger than or equal to a given threshold.

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. For a given transaction, the target attribute does 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 contains all the literalsets that can be built up given X, and having $|X|$.

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 $CT(X)$, contains exactly $2^{|X|}$ cells. Each cell yields the support of a literalset $Y\overline{Z}$ belonging to the literalset lattice associated with 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, for each item $Y\overline{Z}$ belonging to its literalset lattice, we measure the difference between the square of the support of $Y\overline{Z}$ and its expectation value ($E(Y\overline{Z})$), and divide by the average of $Y\overline{Z}$ ($E(Y\overline{Z})$). Finally, all these values are summed.

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

Brin et al. (14) 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 threshold given by the end-user 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* (Moore, 15) is used to evaluate whether a correlation rule is semantically valid: all literalsets of a contingency table must have an expectation value not equal to zero and 80% of them must have a support larger than 5% of the whole population.

This last *criterium* has been generalized by Brin et al. (14) as follows: $MinPerc$ of the literalsets of a contingency table must have a support larger than $MinSup$, where $MinPerc$ and $MinSup$ are thresholds specified by the user.

Definition 4 (Equivalence Class associated with a literal). *Let $Y\overline{Z}$ be a literal. Let us denote by $[Y\overline{Z}]$ the equivalence class associated with the literal $Y\overline{Z}$. This class contains the set of transaction identifiers of the relation 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. If the literalsets, related to the equivalence classes of a CV, are ordered according to the lectic order, it is possible to know, because of the binary coding used, the literal relative to a position i of a contingency vector ($i \in [0; |X| - 1]$). This is because the literal and the integer i have the same binary coding. The following proposition shows how to compute the CV of the $X \cup A$ pattern given the CV of X and the set of identifiers of the relation 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 contingency vectors of X and A as follows:*

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

In order to mine DCRs, we have proposed in (Casali and Ernst, 1) the LHS-Chi2 algorithm (see Alg. 1 for a simplified version). This algorithm is 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 used, given the contingency vector of a pattern X and the set of the transaction identifiers containing a 1-item A, to build the contingency vector of the pattern X ∪ A. The `CtPerc` predicate checks the relaxed Cochran *criteria*.

4 Extracting Correlated Patterns in Parallel

As it is easy to apprehend when regarding last section, the use of a recursive algorithm in a multicore programming environment is an effective challenge. This

Alg. 1 LHS-Chi2 Algorithm.

Input: X and Y two patterns
Output: $\{itemset\ Z \subseteq X$ such that $\chi^2(Z) \geq MinCor\}$

1: **if** $Y = \emptyset$ **and** $|X| \geq 2$ **and** $\exists t \in \mathcal{T} : t \in X$ **and** $\chi^2(X) \geq MinCor$ **then**
2: **Output** X, $\chi^2(X)$
3: **end if**
4: $A := max(Y)$
5: $Y := Y \backslash \{A\}$
6: LHS-Chi2(X,Y)
7: $Z := X \cup \{A\}$
8: VC(Z) := CREATE_CV(CV(X),Tid(A))
9: **if** $CtPerc(CV(Z), MinPerc, MinSup)$ **then**
10: LHS-Chi2(Z,Y)
11: **end if**

because recursion cannot be measured in terms of number of loops to perform. However, parallelizing existing algorithms is an important consideration as well. We first tried to replace the recursive calls by calls to appropriate threads, which quickly appeared as an impossible solution. Another possible approach was based on the well known fact that each recursive algorithm can be rewritten in a iterative format. 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.

Finally, and in order to solution 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. This is why we decided to go back to the Levelwise order to prune in a parallel way the search space, and to keep the Contingency Vectors in order to manage the constraints.

The corresponding result is presented hereafter in the form of three functions. The overall algorithm, called *PLW_Chi2* (where PLW stands for Parallel Levelwise), demonstrates the parallel features of our method. The second function *dowork_level* constitutes each single Task executed in parallel. Finally, a particular aspect of such a Task is detailed in a third function, associated to specific shared resource management issues. We first present these functions through simplified code and comment them in a second stage.

```
void PLW_Chi2 (unsigned short X[], unsigned short sX, unsigned short sI)
// X[] : set of computed 1-items
// sX : number of valid 1-items within X[]
// sI : total number of 1-items in X[]
```

```
{
   unsigned long cit, nit; // number of candidates at level l and (l+1)
   cit = sX;
   for (unsigned char curlev = 2; curlev <= MaxLv && cit > 0; curlev++)
   {
       nit = 0L;
       T_Res aRes;
       combinable<unsigned long> lnit;
       parallel_for(0u, (unsigned) cit, [curlev, X, sX, &aRes, &lnit] (int i)
       {
           dowork_level (curlev, X, sX, i, sI, &aRes);
           lnit.local() += aRes.nit;  // ...
       });
       nit = lnit.combine(plus<unsigned long>());  // ...
       cit = nit;
       update_shared_resources ();
   }
}
```

Parallelization takes place at each level (*curlev* variable) of the LEVELWISE search algorithm. The number of launched Tasks at level l directly depends of the number of existing candidates at level *(l - 1)*, e.g. *cit*. Each Task corresponds principally to a call to the *dowork_level* function, which performs the work it is intended to do, and collects some statistics during the call through the *aRes* object. We interest us here only to a particular statistic, the *lnit* member of the *aRes* object, which 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 here as an aggregation pattern.

We use therefore a two-phase approach: First, we calculate partial results locally on a per-Task basis. Then, once all of the per-Task partial results are at disposal, we sequentially merge the results into one final accumulated value. The PPL provides a special data structure that makes it easy to create per-Task local results in parallel, and merge them as a final sequential step. This data structure is the *combinable* class. 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.

The *dowork_level* function is for its part roughly implemented as follows:

```
void dowork_level (
    unsigned char nc, unsigned short pX[], unsigned short cX,
    unsigned long nel, unsigned short sIX, T_Res& pRes)
{
```

```
    unsigned short vmin, tCand[MaxLv + 1]; // a candidate
    unsigned long j, k;
    unsigned char *theVC;  // a CV
    // other declarations and initializations ...

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

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 here to produce all the candidates of 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 (the whole is done by 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.

The last function we present is *store_CV*. We focus moreover on a very specific section of code dedicated to the storage of results:

```
bool store_CV (unsigned short X[], unsigned short 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();
      }
    // ...
}
```

Let us first explain the functionality involved in the last *if* statement. k-itemsets verifying the whole defined constraints and including one item belonging to the target column have to be included into the results. This is managed 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 multi-threading 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 presentation, some explanations concerning the way we manage the memorization of candidates (and associated information such as Contingency Vectors). The main shared data structure in our developments is a tree storing the k-itemsets of "interest". The corresponding node structure, given in C:

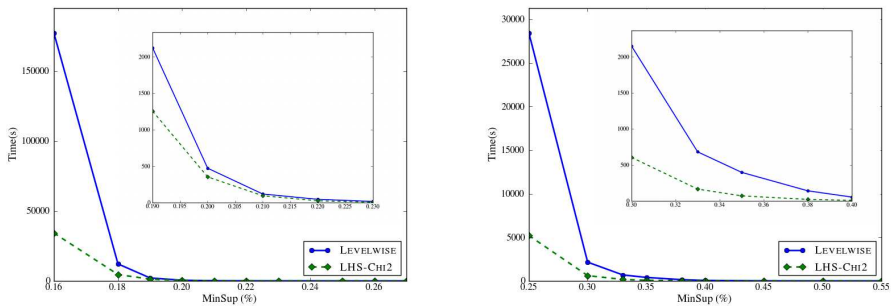
```
typedef struct pattern_node
{
   unsigned short *Mot; /* the pattern */
   unsigned char *pVC; /* pointer to the Contingency Vector */
   T_NM *frere; /* pointer to next node at same level */
   T_NM *fils; /* pointer to next node at lower level */
   ...
} T_NM;
```

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.

5 Experimental Analysis

As briefly mentioned in the Introduction Section, this work has been initially applied on concrete data measurement files provided by two industrial manufacturing partners in the area of Microelectronics : STMicroelectronics (STM)

and ATMEL (ATM, which became LFoundry). 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. The signification of these parameters were given in section 3.2.



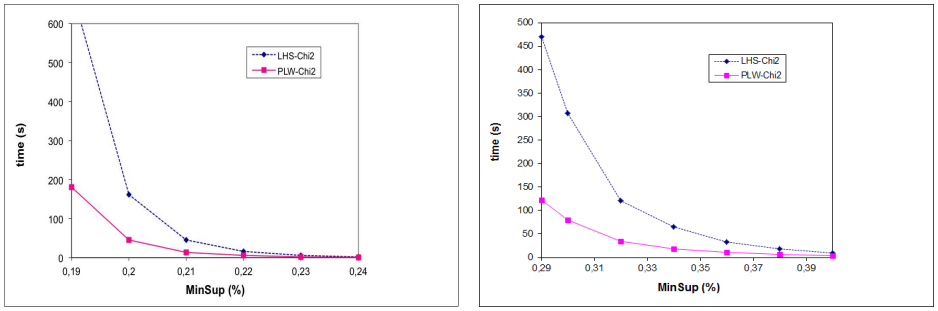
(a) Results for STM File (b) Results for ATM File

Fig. 1. Execution times with a single processor.

Figures 1(a) and 1(b), extracted from (Casali and Ernst, 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. An increasing windowing of the results is provided for subsequent sub-intervals of $MinSup$.

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 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



(a) Results for STM File (b) Results for ATM File

Fig. 2. Execution times with 4 cores.

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.

6 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 (PPL). One of the advantages 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 the PPL has two disadvantages: on one hand we need to use lambda calculus, and on the other hand, 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 new algorithm obtains a time gain 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 a bitmap representation for a Contingency Vector, thus we can minimize disk I/O, and, for finer control processors, build our own thread manager.

References

Casali, A., Ernst, C.: Discovering correlated parameters in semiconductor manufacturing processes: A data mining approach. Semiconductor Manufacturing, IEEE Transactions on **25**(1) (2012) 118–127

Darlington, J., Ghanem, M., ke Guo, Y., To, H.W.: Guided resource organisation in heterogeneous parallel computing (1996)

Tatikonda, S., Parthasarathy, S.: Mining tree-structured data on multicore systems. PVLDB **2**(1) (2009) 694–705

Mitchell, J.C.: Foundations for programming languages. Foundation of computing series. MIT Press (1996)

Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)

Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In Bocca, J.B., Jarke, M., Zaniolo, C., eds.: VLDB, Morgan Kaufmann (1994) 487–499

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

Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann (2000)

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

Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In Chen, W., Naughton, J.F., Bernstein, P.A., eds.: SIGMOD Conference, ACM (2000) 1–12

Zaïane, O.R., El-Hajj, M., Lu, P.: Fast parallel association rule mining without candidacy generation. In Cercone, N., Lin, T.Y., Wu, X., eds.: ICDM, IEEE Computer Society (2001) 665–668

Pramudiono, I., Kitsuregawa, M.: Tree structure based parallel frequent pattern mining on pc cluster. In Marík, V., Retschitzegger, W., Stepánková, O., eds.: DEXA. Volume 2736 of Lecture Notes in Computer Science., Springer (2003) 537–547

Li, E., Liu, L.: Optimization of frequent itemset mining on multiple-core processor. In: VLDB. (2007) 1275–1285

Brin, S., Motwani, R., Silverstein, C.: Beyond market baskets: Generalizing association rules to correlations. In: SIGMOD Conference. (1997) 265–276

Moore, D.: Measures of lack of fit from tests of chi-squared type. Journal of statistical planning and inference **10 (2)**(2) (1984) 151–166