Mining for Functional Dependencies Using Shared Radix Trees in Many-Core Multi-Threaded Systems

Joel Fuentes, Claudio Parra, David Carrillo and Isaac D. Scherson

Abstract We consider the problem of mining for functional dependencies in relational databases. Intermediate data structures, although simple, explode in size and a solution is proposed using radix trees to reduce memory utilization. Parallelism is further applied in a Multi-Core computer to further speedup the process. Because bit-permutations are the basis of the construction of a binary intermediate matrix, radix trees reduce the memory usage 10 times. Multi-Threading the construction and processing of the intermediate data leads to a concurrent computing average-over-time of 63 % on an equivalent speedup of 6.3 on a system with 12 cores, 256 GB of memory and 1 TB SSD.

1 Introduction

With the advent of computing systems that use silicon devices with many CPUs per chip, also known as Many-Core or Multi-Core systems, new challenges are posed to programming applications that attempt to use parallelism to achieve a significant computational improvement. Typical Many-Core computers use chips that contain two, four or more cores each. These Multi-Core chips also include an on-chip hierarchical shared cache system that provides local and shared caches. An interface to a large common main DRAM storage completes the solid state memory hierarchy. These systems are normally programmed using threads that are managed by

J. Fuentes (✉)
Universidad del Bío-Bío, Chillán, Chile
e-mail: jfuentes@ubiobio.cl

C. Parra
Universidad Católica del Maule, Talca, Chile
e-mail: parra.claudio.alejandro@gmail.com

D. Carrillo · I.D. Scherson
University of California, Irvine, USA
e-mail: dcarril@ics.uci.edu

I.D. Scherson
e-mail: isaac@ics.uci.edu

the operating system to execute in the available cores attempting to use as much parallelism as possible. The main problem that arises is the management of shared data structures in the shared hierarchical memory to guarantee synchronized access to the data structures, avoiding deadlock and providing a correct access sequence as required by the program.

Herlihy and Shavit [7] wrote a book that discusses the methodologies used to properly program Multi-Core systems. Exploiting parallelism depends very much on the synchronization mechanisms available to avoid shared data conflicts. Their book has become a classic and has been adopted to teach MultiProgramming courses.

Experience with practical programs shows that in addition to deadlock avoidance, programmers need to be very careful about write through latencies that are bound to create incorrect value reads if threads access variables before the write through mechanism updates values throughout the memory hierarchy. It seems that even when sequentializing shared data accesses, the write through mechanism may get in the way of correct execution, and might lead to a slowdown in program execution.

In this paper we consider an actual practical problem encountered when trying to discover functional dependencies (FDs) in relational data bases using a recently introduced technique based on the generation of refutations [6]. Given a relational database with n records of k attributes each, the method starts by generating a refutation matrix that represents exhaustively all attribute groups where no dependencies can be found. It is shown that the size of this matrix can explode beyond the storage capabilities of the computer and a need is identified to represent it using radix trees. The generation of this refutation matrix is almost embarrassingly parallel and a discussion is presented on how to gain performance by exploiting concurrency both with a straight forward matrix data structure as well as with a radix tree representation.

It will be shown that a big bottleneck is an intermediate data structure whose size may overcome the available storage in the computer. Radix trees are suggested to reduce the demand on memory and are shown to yield a reduction of 10 times on average for the worst case. Parallelization of the intensive phase of the procedure is done on a Multi-Core engine using Multi-threaded programming. Recognizing the inherent sequentialization present in shared memory/data programs, a figure of merit is used to determine what percentage of the execution time threads are allowed to run in parallel. A companion speedup is also calculated. For a 12-core computer, with 256 GB of DRAM and 1 TB disk, experimentation shows that on average a speedup of 6.3 is achieved with parallelism observable 63 % of the time.

2 Algorithm for Mining Functional Dependencies

Consider a relational database where R is a relation with a set of attributes $A = \{a_i | i = 1 \dots k\}$ and r is an instance of R where each attribute in each tuple can assume a value in some domain. We denote by $t(a_i)$ the value of attribute a_i

in tuple t. A functional dependency (FD) is an expression of the form $X \rightarrow a_i$, where $X \subseteq \{A - a_i\}$. X is called the *determinant set* and a_i is called the *dependent attribute*. The FD $X \rightarrow a_i$ is *valid* in the instance r if and only if for every pair of rows (tuples) $t, t' \in r$, whenever $t[a_j] = t'[a_j]$ for all $a_j \in X$, it is also the case that $t[a_i] = u[a_i]$.

Many direct algorithms have been proposed to find FDs in relational databases [2, 4, 5, 8, 10]. This work is based on a novel approach that first prunes the search space by determining which attributes cannot depend on others. The idea is to generate first "refutations" by exhaustively searching all tuple pairs in the relational database to identify which subsets of attributes cannot determine another attribute.

To facilitate the description of the operations in Refutation-Based FD mining (RB-FD) algorithm [6], let us give the following definitions:

1. Let $A = \{a_i \mid i = 1 \dots k\}$ be a set of attributes where each attribute can assume a value in some domain.
2. A relation R over A describes all possible tuples of values of attributes in A.
3. An instance r of R is a subset of tuples of R. We denote by $t(a_i)$ the value of attribute a_i in tuple t.
4. A refutation $[A - \{a_i\}] \nrightarrow a_i$ holds if and only if for $t, t' \in r,\ t(a_i) \neq t'(a_i) \wedge \exists a_j,\ j \neq i,\ t(a_j) = t'(a_j)$.

A refutation is found when two different values for $t(a_i)$ and $t'(a_i)$ correspond to tuples where some subset of the remainder $[A - \{a_i\}]$ attributes are equal.

The result of finding refutations can be kept on a binary matrix H where a row corresponds to the comparison of a pair of tuples t, t' in r. If $t(a_i) \neq t'(a_i)$, the row is generated with 1s at attribute positions where $t(a_j) = t'(a_j)$, $i \neq j$, and 0s elsewhere. If $t(a_i) = t'(a_i)$, no row is generated.

Note that if a_i does not depend on some set of other attributes it does not depend on any subset of those attributes. We conclude that when looking for refutations, we only need to retain those with the maximum number of attributes other than a_i.

If two refutations are generated such that the one with more 1s, say $\underline{h}$, has 1s in all the positions where other $\underline{h'}$ has 1s, then only the one with more 1s ($\underline{h}$) needs to be retained. We say that the former ($\underline{h}$) contains the latter ($\underline{h'}$). The test for containment can be summarized as follows:

$$\text{If } (\underline{h} \wedge \underline{h'}) \oplus \underline{h} = \underline{0} \Rightarrow \text{ retain } \underline{h'}$$
$$\text{If } (\underline{h} \wedge \underline{h'}) \oplus \underline{h'} = \underline{0} \Rightarrow \text{ retain } \underline{h}$$

The procedure for generating the matrix H for an attribute a_i in r is shown in pseudocode in Algorithm 2 below.

```
Input  : relation $r$, set of attributes $A$
Output: set of refutations $H$
for $t \in r$ do
    for $t' \in r$ do
        if $t(a_i)! = t'(a_i)$ then
            $h = \emptyset$;
            for $a_j \in \{A - a_i\}$ do
                if $t(a_j) == t'(a_j)$ then
                    $h = h \cup a_j$;
                end
            end
            for $h' \in H$ do
                if $(h \wedge h') \oplus h = 0$ then
                    $continue\ next\ t'$;
                end
                if $(h \wedge h') \oplus h = 0$ then
                    $H = H \setminus \{h'\}$;
                end
            end
            $H = H \cup h$;
        end
    end
end
```

Algorithm 2: Generation of refutations H for attribute a_i.

Example. Given a 6-attribute relation with $A = \{a, b, c, d, e, f\}$ such that the domain for $a = \{a_1, a_2, a_3, a_4, a_5\}$, for $b = \{b_1, b_2, b_3, b_4\}$ and so on for c, d, e and f (see Fig. 1), and the instance r shown in Fig. 1 we produce the matrix H for attribute 6 as its refuted attribute. We observe that the tuples 1 and 2 produce the refutation $\{a, b\} \nrightarrow f$ (represented by the vector $\underline{h} = 11000$) since the values of attribute f in tuples 1 and 2 are distinct and for attributes a and b are equal; tuples 1 and 3 produce the refutations $\{b, d\} \nrightarrow f$ ($\underline{h} = 01010$); tuples 1 and 7 produce the refutations $\{a\} \nrightarrow f$ ($\underline{h} = 10000$) and so on. Keeping only the maximal refutations we discard the refutation $\{a\} \nrightarrow f$ and finally obtain the refutations $\{a, b\}$ and $\{b, d\}$ represented by the matrix shown below:

Fig. 1 Relation instance with $A = \{a, b, c, d, e, f\}$

Tuple ID	a	b	c	d	e	f
1	a_1	b_3	c_2	d_1	e_4	f_1
2	a_1	b_3	c_3	d_3	e_1	f_2
3	a_2	b_3	c_5	d_1	e_5	f_4
4	a_3	b_3	c_2	d_3	e_3	f_1
5	a_4	b_2	c_2	d_8	e_2	f_1
6	a_5	b_4	c_4	d_1	e_3	f_1
7	a_1	b_1	c_3	d_7	e_6	f_2

$$H = \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Phase 1 of the RB-FD algorithm is the generation of the refutation matrix H just described.

As seen in [6], Phase 2 consists in finding the minimal transversals of a hypergraph represented by a binary matrix H, the bit-wise complement of H. A minimal transversal τ_m of the hypergraph H' is a $(k-1)$-bit binary vector that contains the minimal number of 1s such that $\tau_m \wedge h_i$ contains at least one 1 for all rows i of H'.

The procedure is supposed to generate all possible distinct minimal transversals for each matrix H (for each attribute). The detail of the minimal transversals generation is not within the main scope of this paper and are left to the interested to read in [6].

Example. From the refutation matrix found from r in Fig. 1 in the previous example, we continue now by obtaining the hypergraph of the complements of the hyperedges in H, which is $H' = \{\{c, d, e\}, \{a, c, e\}\}$, represented by the binary vectors $H' = \{00111, 10101\}$. Finally the minimal transversals correspond to the functional dependencies in minimal form:

$$H = \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

That is, we have discovered the functional dependencies $\{a, d\} \rightarrow f$, $\{c\} \rightarrow f$ and $\{e\} \rightarrow f$.

Even though the generation of the refutation matrices H for each attribute has polynomial complexity, it can become dominant due to the large size of the database r (large n and k) and also because the size of H may explode as it will be seen in the next section.

3 The Size of Refutation Matrices

Consider as above a n-tuple instance r of a relation R. It was shown that to generate a refutation matrix H, all pairs of tuples in r need to be compared to obtain refutation vectors that are inserted in a running matrix H following the containment rule.

To find the number of rows in H, that is the number of "non-contained" refutations, out of the $O(n^2)$ generated by comparing all possible pairs of tuples in r, assume that some pair of tuples generated a refutation with a maximum number of 1s. It is obvious that the only non-contained other refutations that could be added to H are those binary vectors that are some permutation of this maximum 1s refutation. All other refutations will be contained in the maximum one, or in one that is permutation of the maximum one.

If q is the number of 1s in the refutation vector with the maximum number of ones, the maximum number of non-contained refutations (rows) in H is:

$$max\ rows\ in\ H = \binom{k}{q} = \frac{(k-1)!}{q!(k-1-q)!} \qquad k, q \in \mathbb{N}$$

which has a maximum when $q = \frac{k-1}{2}$.

The proof of this maximum can be obtained by induction and we omit it as it is simple and does not add substantial knowledge to this work.

Observation: If $n^2 \geq \frac{k-1!}{q!(k-1-q)!}$ and all permutations of a binary vector with $q = \frac{k-1}{2}$ 1s are generated, the maximum size of H is:

$$|H| = \frac{k-1!}{\frac{k-1}{2}!(k-1-\frac{k-1}{2})!} \tag{1}$$

Example. Table 1 illustrates the growth of memory utilization needed to store refutation matrix H when increasing the number of attributes in the worst case.

It is clear that there is a need for a compact representation of refutations in main memory. In [9] Knuth shows different forms to represents all the permutations and introduces the concept of path for these sequences. Table 2 illustrates all the combinations that corresponds to a worst case when finding $\binom{6}{3} = 20$ maximal refutations from a relation with 6 attributes. From the second to the fourth column corresponds to different forms of representing these binary strings that can also be seen as compact representations. The second column corresponds to the dual combination $b_p \dots b_1$ that lists the position of zeros. The third column represents the primal combination $c_p \dots c_1$ that lists the positions of the ones. The fourth column corresponds to the multicombination $d_p \dots d_1$ that lists the number of 0s to the right of each 1.

Furthermore, Table 2 presents the path of each binary string. Each binary string is equivalent to a path of length $k-1$ from the corner to corner of an $q \times (k-1-q)$ grid, because such a path contains q vertical steps and $k-1-q$ horizontal steps.

Table 1 Memory utilization for refutation matrix H

# Attributes	Memory
16	25.7 KB
24	8.1 MB
32	2.4 GB
40	689 GB
48	195 TB

Table 2 Representations of maximal refutations for $k = 6$ and $q = 3$

$a_5a_4a_3a_2a_1a_0$	$b_3b_2b_1$	$c_3c_2c_1$	$d_3d_2d_1$	Path
000111	543	210	000	
001011	543	310	100	
001101	541	320	110	
001110	540	321	111	
010011	532	410	200	
010101	531	420	210	
010110	530	421	211	
011001	521	430	220	
011010	520	431	221	
011100	510	432	222	
100011	432	510	300	
100101	431	520	310	
100110	430	521	311	
101001	421	530	320	
101010	420	531	321	
101100	410	532	322	
110001	321	540	330	
110010	320	541	331	
110100	310	542	332	
111000	210	543	333	

The concept of path in this sense is useful to define a new data structure to store refutations. It can be seen that the worst case presents common prefix sequences in a half of the total of refutations.

4 Reducing the Size of Intermediate Data Structures

The refutation matrix H represents a simple structure to store refutations but it produces high costs to keep the maximal refutations in it. For example, when a refutation is found, a review step over the entire matrix H must be done by removing all its subsets (contained refutations). The other important cost is the memory usage, where this structure can explode if the worst case is presented.

To deal with these issues the radix tree data structure is introduced. A radix tree is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. In a regular radix tree each edge is assigned with some symbol. Thus, any route from a tree root to one of its leaves defines precisely only one string. As the refutations are represented as a binary strings, for the radix tree only the symbols 0 and 1 are considered.

When inserting these refutation in the radix tree and based on the property that all the refutations have the same length, the refutations are represented as paths from the root to every leaf. Figure 2 on the right illustrates the refutations from H inserted in the radix tree. To make the radix tree smaller, it is possible compress it. Such form means that a number of bits B per node is defined allowing that nodes with the same bits in the same level are packed.

Unlike balanced trees, radix trees permit lookup, insertion, and deletion in $O(k)$ time rather than logarithmic. However for our radix tree implementation, a fixed number of bits B is defined as the strict number of bits per node. This value will also represent the maximum common number of bits for different refutations that can be compacted in a node.

Formally, let B be the number of bits in each node. The worst case for the radix tree when looking up for a refutation is similar to the matrix H, $O(|H|)$. It occurs when having defined the value B there are no common prefixes of size B for the set of refutations. For instance, if there exist $|H|$ different refutations and they are different from each other in the first B bits, $\{X_1, \ldots, X_B\}$, and the rest of bits $\{X_{B+1}, \ldots, X_k\}$ for every refutation are similar, then there is no possible compression. However, from the previous section it was shown that if the worst case scenario for the number of refutations occurs, then common prefixes exist (common initial paths exist from one corner to the opposite on the diagonal).

$$H = \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

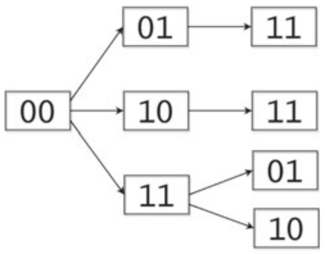

Fig. 2 Maximal refutations stored in the matrix H (*left*) and the radix tree (*right*) with $B = 2$

The average case can be calculated as follows. Let n be the number of refutations with k bits. Let $M = 2^B$ the number of possible combinations given B bits, and m_i the i-th combination of bits. Let r_a and r_b be refutations. Finally, let $bits(B, r_a)$ be the first B bits of the refutation r_a.

The probability of having two refutations with the same first B bits is:

$$\begin{aligned} Pr(bits(B, r_a) == bits(B, r_b)) &= \sum_{i=1}^{M} Pr(bits(B, r_a) = m_i \ AND \ bits(B, r_b) = m_i) \\ &= M * Pr(bits(B, r_a) = m_i) * Pr(bits(B, r_b) = m_i) \\ &= M * (\frac{1}{M})^2 \\ &= \frac{1}{M} \end{aligned}$$

From above, $Pr(bits(B, r_a) \neq bits(B, r_b)) = 1 - \frac{1}{M}$. Then the probability that $n - 1$ refutations are different to r_a is $(1 - \frac{1}{M})^{(n-1)}$. This means r_a is unique in its first B bits. If all the refutations are unique in their first B bits, then the expected number of different refutations is:

$$E[D] = \sum_{i=1}^{n} Pr(r_i \ is \ unique) = n * (1 - \frac{1}{M})^{(n-1)}$$

The number of refutations that share the first B bits is $n(1 - (1 - \frac{1}{M})^{(n-1)})$

For the first level of the tree there are in average $n * (1 - \frac{1}{M})^{(n-1)} * B$ bits, and if it is compared with the matrix H the memory usage is reduced by $(1 - \frac{1}{2^B})^{(n-1)}$. On the following levels the considerations are the same.

Experimental results show the important reduction in the memory usage by using the radix tree. For instance, Fig. 4 illustrates a set of experiments comparing the memory usage by the matrix H and the radix tree when increasing the number of refutations. This experiment corresponds to a sequential execution of the algorithm using both data structures.

5 Parallel Generation of Refutation Sets

The goal of the parallelization of the RB-FD on multicore systems is to minimize the processing time with the computing power being efficiently utilized. Thus, the parallelization plus the new data structure proposed would allow mining big datasets in shorter time.

From results presented in [6] it can be seen that first step of RB-FD represents the most time-consuming task. This part of the method involves two tasks:

1. Obtain the set of maximal refutations H with the form $[A - a_i] \nrightarrow a_i$ by comparing every pairs of tuples for every attribute. That is, $\forall a_i \in A$, the set of pairs of tuples $(t, t') \in r \times r$ such that $(\forall a_j \in [A - a_i])\ t[a_j] = t'[a_j] \wedge t[a_i] \neq t'[a_i]$. This task is exactly $\Theta(k^2 \frac{n(n-1)}{2})$ with n the number of tuples and k the number of attributes.
2. Keep only the maximal refutations in H. For every new refutation, check its maximality with the existing refutations in H.

The complexity of the first task does not seem to involve a problematic issue even when having a big number of tuples. However the verification of maximal refutations when inserting a new one in H would produce an upper bound of $O(|H|)$ whose complexity becomes the hardest with the presence of the worst case scenario. Thus, we can have $O(|H|) = \frac{k!}{p!(k-q)!}$ as a verification for every new refutation found.

The *Dynamic Multithreading* model described in [3] allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming. The model represents a multithreaded computation as a directed acyclic graph $G = (V, E)$ whose vertices are instructions and $(u, v) \in E$ if u must be executed before v. The time T_p needed to execute the computation on p cores depends on two parameters of the computation: its work T_1 and its span T_∞. The work is the running time on a single core, that is, the number of nodes (i.e., instructions) in G, assuming each instruction takes constant time. Since p cores can execute only p instructions at a time, we have $T_p = \Omega(T_1/p)$. The span is the length of the longest path in G. Since the instructions on this path need to be executed in order, we also have $T_p = \Omega(T_\infty)$. Together, these two lower bounds give $T_p = \Omega(T_\infty + T_1/p)$. The degree to which an algorithm can take advantage of the presence of $p > 1$ cores is captured by its speed-up T_1/T_p and its parallelism T_1/T_∞. In the absence of cache effects, the best possible speed-up is p, known as linear speed-up. Parallelism provides an upper bound on the achievable speed-up.

The proposed parallel solutions adopt the *Dynamic Multithreading* model. Following this model, two important features are defined to reflect the parallel behavior: nested parallelism and parallel loops. Nested parallelism allows a subroutine to be spawned, allowing the caller to proceed while the spawned subroutine is computing its result. A parallel loop (*parallel for* in Algorithms 3 and 4) is like an ordinary *for* loop, except that the iterations of the loop can execute concurrently.

Two parallel alternatives are presented in this section using radix trees.

5.1 Parallelism Through Attributes

It is easy to see that the first step of the studied algorithm is embarrassing parallel. The objective of this step is to obtain the set of maximal refutations for every attribute in A, it means that for every attribute there exists a set of refutations that is independent from the sets generated for other attributes.

```
Input  : relation $r$, set of attributes $A$
Output: sets of refutations $H_1, \dots, H_k$
parallel for $a \in A$ do
    $H_a$ = findRefutations(a);
end
function findRefutations(a)
    for $t \in r$ do
        for $t' \in r$ do
            if $t(a)! = t'(a)$ then
                $h = \emptyset$;
                for $a_j \in \{A - \{a\}\}$ do
                    if $t(a_j) == t'(a_j)$ then
                        $h = h \cup a_j$;
                    end
                end
                addToRadixTree($H_a$, $h$);
            end
        end
    end
    return $H_a$;
end
```

Algorithm 3: Parallel RB-FD through the attributes usign radix trees (RB-FD-rtree-att).

Given the set of attributes $A = \{a_1, a_2, \dots, a_k\}$ and an instance of relation r, this alternative associates independent threads to the search of refutations for each attribute A_i. It results in a set of concurrent threads accessing (read operations) the relation r to find refutations in it. The refutations are kept in radix trees, meaning that each thread has its own radix tree which is independent from another thread's radix tree.

Each process computes exactly $\Theta(k\frac{n(n-1)}{2})$ operations in finding refutations. The Algorithm 3 presents the parallel solution through the set of attributes. The set of refutations for each attribute (the right part in the refutation) are represented by H_{a_i} and corresponds to a radix tree.

5.2 *Parallelism Through Tuples*

This alternative is based on the fact that sometimes a search of FDs for a specific attribute can be needed. For instance, find all the refutations with attribute a_i as the dependent attribute. The approach consists in generating threads that go over a well-defined range on the tuples of the relation r. Thus, every refutation found by a thread will need to be stored in a common radix tree.

Formally, given a set of attributes $A = \{a_1, a_2, \dots, a_k\}$, an instance of relation r and a number of threads p; a range b is defined as $b = |r|/p$, where $0 \le b < |r|$.

A global radix tree is also defined with restricted access for writing and shared access for reading. Each process is given a range of contiguous tuples that will be its scope of search. The search of refutations starts when the main process takes a tuple as a pivot and sends this tuple to each thread that has the labor of finding refutations within its range. The radix tree is modified by getting exclusive access if and only if the refutation to add is maximal.

```
Input  : relation $r$, set of attributes $A$, number of processes $P$
Output: sets of refutations $H_1, \dots, H_k$
$i = 0$;
$j = 0$;
$range = |r|/P$;
for $a \in A$ do
    $H_a = \emptyset$;
    parallel for $t \in r$ do
        $i = j + 1$;
        $j = j + range$;
        findRefutations($a, t, i, j$);
    end
end
function findRefutations(a, t, i, j)
    for $t' \in (i, j)$ do
        if $t(a)! = t'(a)$ then
            $h = \emptyset$;
            for $a_j \in \{A - \{a\}\}$ do
                if $t(a_j) == t'(a_j)$ then
                    $h = h \cup a_j$;
                end
            end
            if $h$ is maximal then
                $lock()$;
                addToRadixTree($H_a, h$);
                $unlock()$;
            end
        end
    end
end
```

Algorithm 4: Parallel RB-FD through tuple ranges using shared radix trees (RB-FD-rtree-tup).

For an attribute D and a range b each thread computes exactly $\Theta(k * n * b)$ operations in finding refutations. Finding the refutations for all the attributes the complexity is $\Theta(k^2 * n * b)$. The Algorithm 4 presents the parallel solution through the set of attributes.

6 Experimental Results

A measurement of the average-over-time of the number of threads simultaneously running is introduced as the figure of merit that characterizes the efficiency of a Multi-threaded execution in a Many-Core computer. The objective is to distinguish, for a certain problem and execution, what is the proportion of the total running time for which threads execute in parallel and achieve some progress in their work.

Consider that blocking mechanisms are used to guarantee exclusive access to certain memory location. These mechanisms usually have three main states:

- Acquiring permission: The thread asks for permission to access the protected memory location. This state can consider blocking.
- Performing actions with permission: The thread performs actions on the protected memory location.
- Releasing the permission: The thread releases the permission after finishing its actions on the protected memory location.

It can be seen that the only blocking state is the first one, meaning that there can be no progress on the actions that the thread has to perform. For instance, the Lock blocking mechanism considers that only one writer thread can have the lock at a time. Thus the blocking state consists on repeatedly and unsuccessfully atomic operations by the thread to change the lock state from *unlock* to *lock*.

Formally, let Γ be the total running time of a thread and let β be the time the thread spends in blocking state, with a total of p threads running concurrently, the average of parallel running time ($APRT$) is defined as follows:

$$APRT = \frac{\sum_{i=1}^{p} \Gamma_i - \beta_i}{p}$$

$APRT$ and the level of speedup are used to analyze the performance of the parallel algorithms described previously. We carried out a set of experiments using both of the proposals with the introduction of the radix tree.

Algorithms were implemented in the C++11 programming language. The experiments were carried out on a Dual 12 Core Xeon Haswell, with a total of 24 physical cores running at 2.60 GHz. Hyperthreading was disabled. The computer runs Linux 3.19.0-26-generic, in 64-bit mode. This machine has per-core L1 and L2 caches of sizes 32 and 256 KB, respectively and a per-processor shared L3 cache of 30 MB, with a 256 GB DDR RAM memory and 1 TB SSD. Algorithms were compared in terms of running times using the usual high-resolution (nanosecond) C functions in *time.h*.

The datasets (input) used are from UCI Machine Learning Repository [1], in particular we use the PAMAP2 Physical Activity Monitoring datasets.

6.1 Radix Tree Performance

One of the first improvement for RB-FD that was introduced in the previous section was the use of a radix tree instead of a matrix to store maximal refutations. This new data structure allows refutation compression by using the similar refutations' prefix as a single node.

A number of 12 relation instances where used to carry out some experiments to measure the performance with the goal of seeing what is the improvement achieved by using this new data structure. Figure 3 shows the running time (in seconds) of RB-FD using the original matrix H and the new version using a radix tree (RB-FD-rtree). It is clear to see that RB-FD-rtree presents a significant less running time when increasing the number of tuples in the datasets. A similar behavior occurs when increasing the number of attributes. Furthermore, as analyzed previously, the memory usage is reduced notoriously when using the radix tree. For instance, when storing 30,000 maximal refutations the memory utilization is 10 times smaller than using an array representation for the matrix H (Fig. 4).

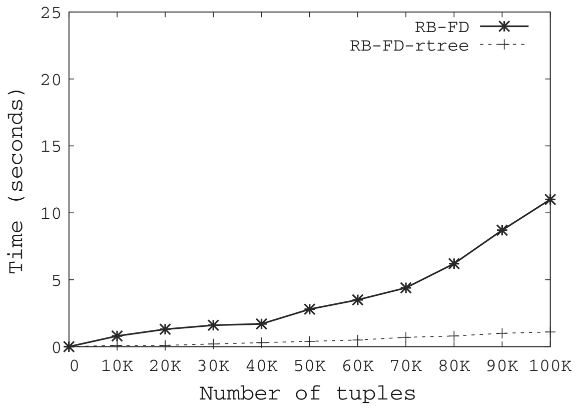

Fig. 3 Running time of RB-FD and RB-FD-rtree

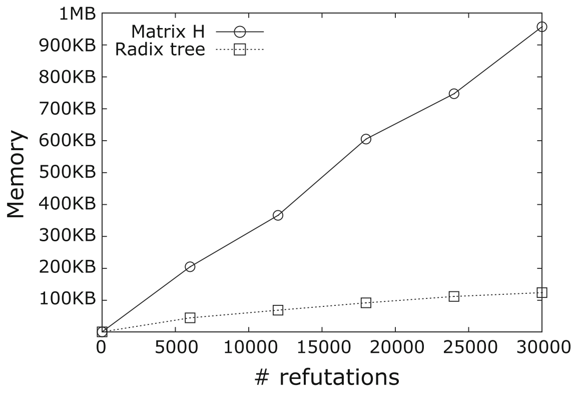

Fig. 4 Memory usage of Matrix H and radix tree storing refutations

6.2 Parallel Alternatives

Table 3 shows the times achieved by RB-FD on its three versions: RB-FD-rtree (sequential algorithm) and the parallel versions RB-FD-rtree-att (parallelization through attributes) and RB-FD-rtree-tup (parallelization through tuples). This experiment corresponds to the finding of functional dependencies on a datasets with 502,182 tuples and 32 attributes. Figure 5 shows the speedup from the sequential version RB-FD-rtree when increasing the number of cores for RB-FD-rtree-att and RB-FD-rtree-tup. Up to 12 cores, it can be seen that the speedup of the parallelization through tuples grows slowly, then it stops growing and decreases its speedup a bit and continues fluctuating between a speedup of 2 and 3. The main reason for this phenomenon is that increasing the number of threads in RB-FD-rtree-tup produces data contention and more lock operations on the shared radix tree. The data contention does not occur with RB-FD-rtree-att, which presents a constant growth, since there is no shared radix tree, due to each thread has its own radix tree.

The parallelization over tuples (RB-FD-rtree-tup) works with a simple Lock as a mechanism for allowing exclusive access to the radix tree when a new maximal

Table 3 Running times (in seconds) varying the number of cores

# cores	RB-FD rtree	RB-FD rtree-att	RB-FD rtree-tup
1	4.41	4.41	4.41
4	–	2.08	1.7
8	–	1.94	1.04
12	–	1.46	0.76
16	–	1.86	0.66
20	–	1.62	0.56
24	–	1.90	0.44

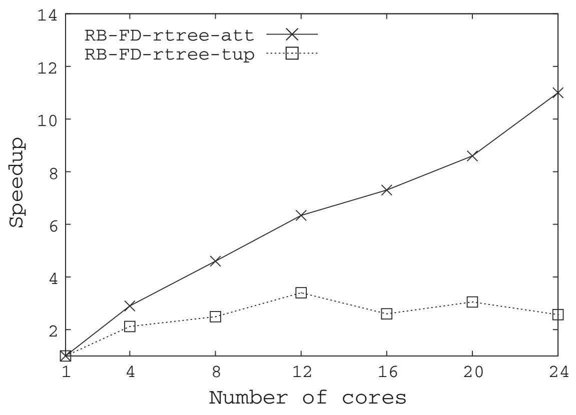

Fig. 5 Speedup of parallel alternatives varying the number of cores

Table 4 Average of blocking and total time for threads running RB-FD-rtree-tup

# cores	Blocking time	Total time	APRT
4	0.018	0.22	0.91
8	0.030	0.156	0.79
12	0.05	0.132	0.63
16	0.048	0.106	0.57
20	0.064	0.114	0.43
24	0.082	0.126	0.35

refutation is found. Therefore it sounds interesting to see what is the APRT achieved when different number of threads are finding FDs in a dataset.

Table 3 shows the blocking and total times achieved by the algorithm when threads work on the same attribute (right-part of the refutation). As explained in Algorithm 4 the number of tuples is divided by the number of threads and each thread has to find refutations on its own range. These executions correspond to the same from the previous experiments, but focusing only on RB-FD-rtree-tup and the blocking and total times (Table 4).

According to the results, when increasing the number of cores and threads shorter running time is achieved. However after 12 cores, the running time stops decreasing and keeps stable. This behavior is explained by the addition of more blocking time as the number of cores and threads are increased. Therefore smaller APRT values are obtained and the greater blocking times are added. In other words, with more threads they have to perform less work but they suffer of blocking and waiting times (Fig. 6).

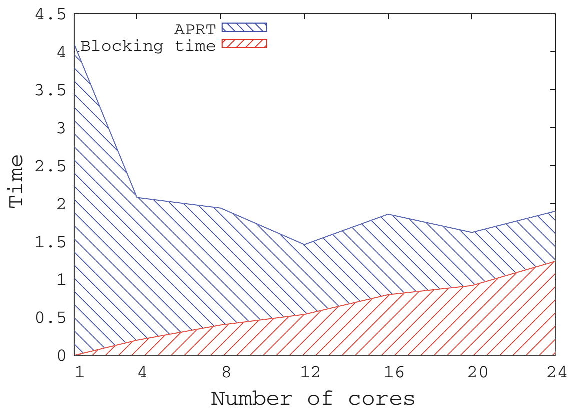

Fig. 6 Running time characterized by APRT and blocking time

Acknowledgments We are grateful to Prof. Gilberto Gutiérrez and Dr. Pablo Sáez for their valuable guidance into the problem of finding functional dependencies. Thanks also to EMC Inc. in Irvine for lending us computing resources.

References

1. Bache, K., Lichman, M.: UCI Machine Learning Repository (2013)
2. Baixeries, J.: A formal concept analysis framework to mine functional dependencies. Workshop on Mathematical Methods for Learning. Como, Italy (2004)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
4. Distel, F., Sertkaya, B.: On the complexity of enumerating pseudo-intents. Discrete Appl. Math. **159**(6), 450–466 (2011)
5. Flach, P., Savnik, I.: Database dependency discovery: a machine learning approach. AI Commun. **12**(3), 139–160 (1999)
6. Fuentes, J., Sáez, P., Gutiérrez, G., Scherson, I.D.: A method to find functional dependencies through refutations and duality of hypergraphs. Comput. J. bxu047 (2014)
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2008)
8. Huhtala, Y., Karkkainen, J., Porkka, P., Toivonen, H.: Tane: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100 (1999)
9. Knuth, D.E.: The art of computer programming, vol. 3, 2nd edn. In: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc, Redwood City, CA, USA (1998)
10. Novelli, N., Cicchetti, R., Fun: An efficient algorithm for mining functional and embedded dependencies. In: Database Theory—ICDT, vol. 1973. Springer, London, United Kingdom, 189–203 (2001)