Solving hypertree structured CSP : Sequential and parallel approaches

Mohammed Lalou [1], Zineb Habbas [2], and Kamal Amroun [3]

[1] University of Bejaia, e-mail : mohammed.lalou@gmail.com
[2] LITA, University of Metz, e-mail : zineb@univ-metz.fr
[3] University of Bejaia, e-mail : $k_$amroun25@yahoo.fr

Abstract

Solving $\mathcal{CSP}$ is in general $\mathcal{NP}$-$\mathcal{C}$omplete. However, there are various subsets of $\mathcal{CSP}$s that can be solved in polynomial time. Some of them can be identified by analyzing their structure. Unfortunately the proposed methods for exploiting these structural proprieties are not efficient in practice. So exploiting these structural properties for solving $\mathcal{CSP}\int$ is a crucial challenge. In this paper, we propose efficient algorithms which exploit these structural proprieties, for both sequential and parallel resolutions. Some experiments done on academic benchmarks show the efficiciency of our approach.

1 Introduction

A CSP consists of a set V of variables, the domains D of these variables and a set C of constraints over these variables. The objective is to assign values in D to the variables in such a way that all constraints are satisfied. $\mathcal{CSP}\int$ are known to be $\mathcal{NP}$-$\mathcal{C}$omplete. Considerable efforts have been made to identify tractable classes. One approach is based on exploiting structural properties of the constraints network. If the $\mathcal{CSP}$ is tree structured then it can be solved in polynomial time. Many techniques have been developed to transform a $\mathcal{CSP}$ into a tree or an equivalent hypertree structured $\mathcal{CSP}$, we can cite [4, 6, 1]. The Generalized Hypertree Decomposition method is the most general method. However the algorithm proposed to solve the hypertree structured $\mathcal{CSP}$ is not efficient in practice. In this work, we propose to improve this algorithm. Mainly we propose two algorithms: an improved sequential algorithm and a parallel algorithm. The sequential algorithm is based on the hashing technique, while the parallel algorithm explores the pipeline technique and the parallel tree contraction algorithm between the nodes to achieve the semi-join operation. In order to validate our proposition we first compare the sequential algorithm proposed in this paper with the basic algorithm proposed in [5] . Based on some experiments done on benchmarks from the literature, we observed a promising gain in terms of cpu time with our sequential approach. More particularly, good results are observed when large size relations are considered. To validate our parallel algorithm, we have proposed a simulation model using logical

Proceedings of the 16th International RCRA workshop (RCRA 2009):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Reggio Emilia, Italy, 12 December 2009

time assumptions. The experimental results outline the practical efficiency of this algorithm and its performances in term of space memory.

This paper is organized as follows : section 2 gives preliminaries of constraint satisfaction problems and well known $\mathcal{CSP}$ decomposition methods by developing more particulary the (generalized) hypertree decomposition method which is the most general one. In section 3, we present our sequential algorithm called S_HBR (for Sequential Hash Based Resolution) which solves the hypertree structured $\mathcal{CSP}$ and we give some experiments of our proposition with respect to the algorithm proposed in [5]. In section 4, we present our parallel algorithm and in section 5, we give some experimental results of this algorithm. Finally, in section 6, we give our conclusion and some perspectives to this work.

2 Preliminary notions

The notion of Constraint Satisfaction Problem CSP was introduced by [7].

Definition 1. Constraint Satisfaction Problem*: A constraint satisfaction problem is defined as a 3-tuple* $\mathcal{P} = \langle X, D, C \rangle$ *where :*
$X = \{x_1, x_2, ..., x_n\}$ *is a set of n variables.*
$D = \{d_1, d_2, ..., d_n\}$ *is a set of finite domains; a variable* x_i *takes its values in its domain* d_i*.*
$C = \{C_1, C_2, ..., C_m\}$ *is a set of m constraints. Each constraint* $\mathcal{C}_i$ *is a pair* $(S(C_i), R(C_i))$ *where* $S(C_i) \subseteq X$*, is a subset of variables, called* ***scope*** *of* C_i *and* $R(C_i) \subset \prod_{x_k \in S(C)_i} d_k$ *is the constraint relation, which specifies the allowed values combinations.*

A **solution** of a $\mathcal{CSP}$ is an assignment of values to variables which satisfies all constraints.

Definition 2. Hypergraph*: The constraint hypergraph [2] of a* $\mathcal{CSP}$ $\mathcal{P} =< X, D, C >$ *is given by* $\mathcal{H} =< V, E >$ *where E is a set of hyperedges corresponding to the scopes of the constraints in C, V is the set of variables of* $\mathcal{P}$*.*

In this paper, hyperedges($\mathcal{H}$) is the set of the hyperedges of the hypergraph $\mathcal{H}$. If h is a hyperedge, then var(h) is the set of variables of h.

Definition 3. *A* join tree *for a hypergraph* $\mathcal{H}$ *is a tree T whose nodes are the hyperedges of* $\mathcal{H}$*, such that, when a vertex v of* $\mathcal{H}$ *occurs in two hyperedges e1 and e2 then v occurs in each node of the unique path connecting e1 and e2 in the tree T.*

Definition 4. Hypertree: *A hypertree for a hypergraph $\mathcal{H}$ is a triple $< T, \chi, \lambda >$ where $T = (N, E)$ is a rooted tree, and χ and λ are labelling functions which associate each vertex $p \in N$ with two sets $\chi(p) \subseteq var(\mathcal{H})$ and $\lambda(p) \subseteq hyperedges(\mathcal{H})$. If $T' = (N', E')$ is a subtree of T, we define $\chi(T') = \bigcup_{v \in N'} \chi(v)$. We denote the set of vertices N of T by* vertices*(T) and the root of T by* root*(T). T_p denotes the subtree of T rooted at the node p.*

Proposition 1. *A $\mathcal{CSP}$ whose structure is acyclic can be solved in polynomial time [6].*

The goal of all structural decomposition methods is to transform a $\mathcal{CSP}$ into an equivalent acyclic $\mathcal{CSP}$ which can be solved in polynomial time. A decomposition method D associates to each hypergraph $\mathcal{H}$ a parameter $\mathcal{D}$-width called the width of $\mathcal{H}$. The method $\mathcal{D}$ ensures that for a fixed k, each $\mathcal{CSP}$ instance with $\mathcal{D}$-*width* $\leq k$ is tractable and then can be solved in polynomial time. Among these methods, the generalized hypertree decomposition (GHD) dominates all the other structural decomposition methods. In the next paragraph, we present the (generalized) hypertree decomposition method.

Definition 5. *A* generalized hypertree decomposition *[10] of a hypergraph $\mathcal{H} =< V, E >$, is a hypertree $HD =< T, \chi, \lambda >$ which satisfies the following conditions :*

1. *For each edge $h \in E$, there exists $p \in vertices(T)$ such that $var(h) \subseteq \chi(p)$. We say that p covers h.*
2. *For each variable $v \in V$, the set $\{p \in vertices(T) | v \in \chi(p)\}$ induces a connected subtree of T.*
3. *For each vertex $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$.*

A hypertree decomposition *of a hypergraph $\mathcal{H} =< V, E >$, is a generalized hypertree decomposition $HD =< T, \chi, \lambda >$ which additionally satisfies the following special condition :*
For each vertex $p \in vertices(T)$, $var(\lambda(p)) \bigcap \chi(T_p) \subseteq \chi(p)$.

The width *of a (generalized) hypertree decomposition $< T, \chi, \lambda >$ is $max_{p \in vertices(T)} |\lambda(p)|$. The (generalized) hypertree width $(g)hw(\mathcal{H})$ of a hypergraph $\mathcal{H}$ is the minimum width over all its (generalized) hypertree decompositions.*

A hyperedge h of a hypergraph $\mathcal{H} = \langle V, E \rangle$ is **strongly covered** in $\mathcal{HD} = \langle \mathcal{T}, \chi, \lambda \rangle$ if there exists $p \in vertices(\mathcal{T})$ such that the vertices in h are contained in $\chi(p)$ and $h \in \lambda(p)$. A (generalized) hypertree decomposition $\mathcal{HD} = \langle \mathcal{T}, \chi, \lambda \rangle$ of $\mathcal{H} = \langle V, E \rangle$ is called **complete** if every hyperedge h of $\mathcal{H}$ is strongly covered in $\mathcal{HD}$.

3 A sequential algorithm

3.1 The basic algorithm

The sequential resolution of a $\mathcal{CSP}$ represented by its hypertree decomposition is given by the following algorithm [5] (see algorithm 1), called in this paper B_A algorithm (for Basic algorithm).

Algorithm 1 B_A algorithm

1: **Input** : $\mathcal{HD} = \langle \mathcal{T}, \chi, \lambda \rangle$
2: **step1**: complete the hypertree decomposition HD
3: **step2**: Solve each λ-term using multi-join operation.
4: **step3**: Solve the resulting CSP using semi-join operation.
5: **Output**: A solution of the given problem.

The second step consists in Solving each sub-problem represented as the λ-term in each node of the hypertree. As a result, we have a new hypertree whose the λ-terms are replaced by a unique constraint relation $\mathcal{R}_p$ corresponding to the projection on the variables in $\chi(p)$ of the join of the constraint relations in $\lambda(p)$. More formally $\mathcal{R}p = (\bowtie_{t \in \lambda(p)} t)[\chi(p)]$. This operation is expensive. The third step consists in semi-join operation which is an other expensive operation.

3.2 The S_HBR algorithm

Some notations : ϑ_i represents the intersection variables of the constraint $\mathcal{C}_i$ with the set of all variables in the union of $\mathcal{C}_j$ where $j < i$ in a given order. $\mathcal{HT}_i$ is the hash table associated to R_i.

The S_HBR for $\mathcal{S}equential\ \mathcal{H}ash\ \mathcal{B}ased\ \mathcal{R}esolution$ algorithm, is formally presented by Algorithm 2. It proceeds in two steps : The SP_HBR for $\mathcal{S}$ub $\mathcal{P}$roblem $\mathcal{H}$ash $\mathcal{B}$ased $\mathcal{R}$esolution algorithm (lines 3 to 6) is the optimization of the join operation. The A_HBR for ($\mathcal{A}cyclic\ \mathcal{H}ash\ \mathcal{B}ased\ \mathcal{R}esolution$) algorithm (line7) is an optimization of the classical Acyclic solving algorithm [3].

We now give more details about SP_HBR and A_HBR algorithms.

a) SP_HBR **algorithm**: The result of the SP_HBR execution is a join-tree whose every node n contains only one table $\mathcal{R}_n$ composed of the tuples satisfying all sub-problems constraints. Nodes of this join-tree are represented as follows: $\mathcal{R}_n = \langle \chi(p), \mathcal{H}rel(p) \rangle$ where $\chi(p)$ is the set of variables of the node p, and $\mathcal{H}rel(p)$ is the constraint relation generated by the join operation. Additionally, for the leaves nodes, $\mathcal{H}rel(p)$'s tuples are hashed on intersection variables with the parent node of p. This algorithm decomposes each node of the join tree into connected components. Then it applies the join operation algorithm for each component using the hash join

Algorithm 2 Sequential hash Based Resolution(S_HBR)

1: **Input**: a hypertree $< \mathcal{T}, \chi, \lambda >$ of a hypergraph $< \mathcal{H} = V, E >$ where $\mathcal{T}$ is a tree, χ associates to each t of $\mathcal{T}$ a set of nodes $\chi(t) \subset V$, and λ, a set of arcs $\lambda(t) \subset E$
2: **Output** : A solution.
3: **Step1**:
4: **for** each node n of $\mathcal{T}$ **do**
5: Apply the SP_HBR algorithm on the node n.
6: **end for**
7: **Step2**: Apply A_HBR algorithm on $\mathcal{T}$.

principle. For doing that, we establish an order among relations in the same component. Then, we hash each relation R_i of the constraint C_i, except the first one, on intersection variables with the relations of the constraints that are before C_i in this order. The join operation is applied using the *Join* procedure. These procedures cannot be described in this paper for restricted space reason. We illustrate SP_HBR by the following example.

Example 1. *We consider a node p of a hypertree $\mathcal{HD}$ labelled as follows:*

$\lambda(p) = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5, \mathcal{C}_6\}$, *with:* $\mathcal{S}(\mathcal{C}_1) = \{a, b, c\}$, $\mathcal{S}(\mathcal{C}_2) = \{i, j\}$, $\mathcal{S}(\mathcal{C}_3) = \{a, d\}$, $\mathcal{S}(\mathcal{C}_4) = \{d, b, f\}$, $\mathcal{S}(\mathcal{C}_5) = \{f, g\}$, $\mathcal{S}(\mathcal{C}_6) = \{i, k\}$.

There are two connected components $R_1 = \{\mathcal{C}_1, \mathcal{C}_3, \mathcal{C}_4, \mathcal{C}_5\}$ *and* $R_2 = \{\mathcal{C}_2, \mathcal{C}_6\}$. *According to the order of the constraints in each component, we have:*

$\vartheta_3 = \mathcal{S}(\mathcal{C}_3) \cap \mathcal{S}(\mathcal{C}_1) = \{a\}$, $\vartheta_4 = \mathcal{S}(\mathcal{C}_4) \cap \{\mathcal{S}(\mathcal{C}_1) \cup \mathcal{S}(\mathcal{C}_3)\} = \{b, d\}$, $\vartheta_5 = \mathcal{S}(\mathcal{C}_5) \cap \{\mathcal{S}(\mathcal{C}_1) \cup \mathcal{S}(\mathcal{C}_3) \cup \mathcal{S}(\mathcal{C}_4)\} = \{f\}$, $\vartheta_6 = \mathcal{S}(\mathcal{C}_6) \cap \mathcal{S}(\mathcal{C}_2) = \{i\}$.*(if C is a hyperedge, then S(C) denotes the scope of C).*

Therefore, for the first component, the $\mathcal{R}_3$'s tuples will be hashed on the set of intersection variables $\vartheta_3 = \{a\}$, those of $\mathcal{R}_4$ and $\mathcal{R}_5$ on, respectively, $\vartheta_4 = \{b, d\}$ and $\vartheta_5 = \{f\}$. For the second component, the tuples of $\mathcal{R}_6$ will be hashed on $\vartheta_6 = \{i\}$. After computing a cartesian product of the components, the resulting tuples t are hashed according to the intersection variables with the parent node of p, if p is a leaf. So, t will be ready for the semi-join operation in $\mathcal{A}$-$\mathcal{HBR}$ algorithm. We do not add t to its related partition P unless if $P = \phi$. Thus, we will have one and only one tuple in each partition by eliminating the duplicated tuples. This elimination has no impact on the resolution because, for the $\mathcal{A}$-$\mathcal{HBR}$ algorithm, each parent node is filtered on its sons [1]. *Moreover, we are interested to get only one solution for the $\mathcal{CSP}$.*

b) The A_HBR algorithm: The A_HBR algorithms tests the consistency of a constraint acyclic network and generates a solution if it exists.

[1] It is sufficient to have only one tuple by partition in the son node relations.

This algorithm takes the hypertree resulting from the SP_HBR algorithm as its input. The A_HBR algorithm (see. *Algorithm 3*) proceeds in two steps: the CONTRACT step and the S_SEARCH step. The first step contracts the deep-rooted hypertree $\mathcal{HT}$. A semi-join operation is associated to each contraction operation. The result is a directional from root toward leaves arc consistent hypertree $\mathcal{HT}'$ (line 3). The second step is the solution search operation, which develops the solution from the $\mathcal{HT}'$'s root to leaf nodes (line 4).

Algorithm 3 Acyclic Hash Based Resolution algorithm (A_HBR)

1: **Input**: A hypertree $\mathcal{HT} = \langle \mathcal{T}, \chi, \lambda \rangle$
2: **Output** : Determine a directional arc consistent and generate a solution.
3: $\mathcal{HT}'$ = CONTRACT ($\mathcal{HT}$)
4: S_SEARCH ($\mathcal{HT}'$),

The *CONTRACT algorithm* strengths the directional arc consistency of the input hypertree. Given a hypertree, a contraction operation is realized between each leaf node and its parent. For each contraction operation we hash each tuple t of a parent node (p) relation on intersection variables with its sons leaves nodes a_i. We denote h_i the hashed values. If a_i's hash tables partitions related to h_i are not empty , we hash t on intersection variables of p with its parent node and save it. Otherwise, we eliminate t. The leaves nodes a_i are marked and considered as pruned. We make the same with the obtained hypertree, and so on, until all the nodes, excepted the root, will be marked. The final result of the contraction operation of a parent node p with its sons leaves nodes, is a node $p' = \langle \chi(p'), \mathcal{H}rel(p') \rangle$, whose relation is arc consistent with those of the son nodes relations, and hashed on the intersection variables of p with its parent node. If the hashed relation $\mathcal{H}rel(p')$ is empty, the problem has no solution.
The S_SEARCH algorithm is a classical *Backtrack free* algorithm applied to the directional arc consistent hypertree resulting from the $CONTRACT$ algorithm execution.

3.3 Complexity of S_HBR

The complexity of the SP_HBR algorithm is

$$\sum_{i=1}^{h-1} (r * P^i) \tag{1}$$

Where r is the the maximum relation size, h is the hypertree width, P is the maximum number of tuples in the partition of hashing on the variables of ϑ (ϑ represents the intersection variables of $\mathcal{C}_i$ with all variables in $\mathcal{C}_j$ where $j < i$). For the A_HBR algorithm, its complexity is $O(nP^2)$. Where n is

the number of constraints in the obtained CSP. But in our case, we have only one tuple by partition, so the complexity of A_HBR algorithm is only $O(n)$.

3.4 Experiments

We have implemented the S_HBR algorithm and the basic algorithm [5] (noted here B_A). We have tested the two solvers on a set of $\mathcal{CSP}$ benchmarks. This benchmarks collection is represented using the new *XCSP 2.1* format proposed by *The organizational committee of the* 3^{th} *international competition of the* $\mathcal{CSP}s$ *solvers* [2][8]. The solvers inputs are $\mathcal{CSP}s$ instances and the corresponding hypertrees in *GML* format. For this, we have exploited a *GML* Parser proposed by *Raitner and Himsol*[3] and another one, proposed by *Roussel*[4], for *XCSP* file. The experiments are made on *Linux* using an *Intel Pentium IV*, with *2.4 GHz* of *CPU* and *600 Mb* of *RAM*. In our results, $|V|$, $|E|$ and $|R|$ represent respectively the variables number, the constraints number and maximum number of tuples by relation associated to a given $\mathcal{CSP}$. $\mathcal{N}_d$ and $\mathcal{HTW}$ are respectively the nodes number of the hypertree and the hypertree width. $\mathcal{S}$-$\mathcal{HBR}(sec)$ illustrate the $\mathcal{S}$-$\mathcal{HBR}$ computational time in seconds, and $B\_A(sec)$ the time of the B_A algorithm. The table 1 summarizes our experiments. The hash function used in this paper is

$$f(t) = \sum_{i=1}^{n}((x_i + 1) * 10^{ln(f(t))+1}) \tag{2}$$

We observe that $\mathcal{S}$-$\mathcal{HBR}$ clearly **improves** the B_A algorithm in terms of CPU time for all the instances.

The S_HBR algorithm gets more better results for the instances which have large relations, because it browses only a sub- part of the relation rather than the totality one as in B_A.

4 The parallel Algorithm

In this section, we introduce a parallel version of our previous sequential algorithm S_HBR. We called it P_HTR for Parallel Hypertree Resolution algorithm and it is described by the algorithm 4. This algorithm uses both **pipeline technique** and the **parallel tree contraction technique**. The pipeline avoids the storage of intermediate results so it reduces the memory space explosion. The parallel tree contraction is the most well known adapted approach for the semi-join operation. However the known parallel

[2]http://www.cril.univ-artois.fr/ lecoutre/research/benchmarks/benchmarks.html
[3]http://www.cs.rpi.edu/ puninj/XGMML/GML-XGMML/gml-parser.html
[4]http://www.cril.univ-artois.fr/ roussel/CSP-XML-parser/

$\mathcal{CSP}$						B_A (sec.)	S_HBR (sec.)
Name	|V|	|E|	N_d	$\mathcal{HTW}$	|R|		
3-insertions-3-3	56	110	86	7	3	>1000	127
domino-100-100	100	100	50	2	100	27	7
domino-100-200	100	100	50	2	200	103	28
domino-100-300	100	100	50	2	300	241	61
series-7	13	42	39	4	42	>1000	106
hanoi-7	126	125	125	1	6558	71	9
haystacks-06	36	95	88	3	8	9	4
haystacks-07	49	153	144	4	9	514	66
langford-2-4	8	32	31	4	8	11	3
Renault	101	134	81	2	48721	780	20
bf-1355-e-63	532	339	223	6	3124	96	11
bf-1355-g-63	532	339	223	6	7775	561	36
bf-2670-b	1244	1354	733	8	31	>1000	77
bf-2670-c	1244	1354	641	13	31	>1000	125

Table 1: Experimental results of $\mathcal{S}$-$\mathcal{HBR}$ algorithm

tree contraction considers at each iteration, for each parent node with only one son node leaf. To increase the parallelism degree in this operation, we developed a new alternative of this technique, based on partitioning the parent node relation (the critical resource) between all processors, which allows an asynchronous updating.

The P_HTR(algorithm 4) procedure proceeds in two steps. The first one concerns the parallel resolution of sub-problems using the P_SBR algorithm (line 4). The second step is the resolution of the whole problem. It is performed using the $\mathcal{P}$_$\mathcal{S}$olving algorithm (line 6).

Algorithm 4 P_HTR ($\mathcal{P}$arallel $\mathcal{H}$yper$\mathcal{T}$ree $\mathcal{R}$esolution) algorithm

1: **Input**: a hypertree $\langle \mathcal{T}, \chi, \lambda \rangle$ of a hypergraph $\mathcal{H} = (\mathcal{V}(\mathcal{H}), \mathcal{E}(\mathcal{H}))$
2: **Output** : The solution of the problem
3: **for** each node n of $\mathcal{T}$ **in parallel do**
4: P_SBR (n) // *Apply the* P_SBR algorithm on the node n.
5: **end for**
6: $P\_Solving$(T') // Apply $P\_Solving$ algorithm on $\mathcal{T}'$ ($\mathcal{T}'$ is the obtained hypertree in the step 3).

4.1 The P_SBR algorithm

The P_SBR for Parallel Sub Problem Resolution algorithm uses the hash technique. The P_SBR algorithm proceeds as the SP_HBR algorithm. After establishing an order on the constraints, it builds a pipeline line composed of join operators whose the first relation is the *probe* relation and the remaining ones are *build* relations. The join operation is performed according to the hash pipelined join principle. Thus, each build relation is hashed on its intersection variables (ϑ_i) with the union of the previous relations.

The parallelism in this algorithm is provided by the Pipeline. At each level of the pipeline corresponds a join operation with two operands. The first one is R_1 (the probe relation) for the first operator, $R_{tmp}[i-1]$ for the i^{th} operator. The second one is the build relation R_i for the i^{th} operator. We make a join operation between each tuple of the first operand and all the tuples of the second operand. The obtained tuples from the i^{th} operator are saved in the temporary relation of $(R_{tmp}[i+1])$ of the following operator.

4.2 *P_Solving* algorithm

P_Solving algorithm is the parallel version of *A_HBR* algorithm. It consists in two operations, $P\_CONTRACT$ (for Parallel CONTRACT), and PS_SEARCH (for Parallel SEARCH). The first operation concerns the parallelization of the CONTRACT operation, while the second one corresponds to the parallel version of the S_SEARCH operation.

Algorithm 5 *P_Solving* algorithm

1: **Input**: A join tree $\mathcal{T}$.
2: **Output** : The solution for the problem.
3: $\mathcal{T}' = P\_CONTRACT\ (\mathcal{T})$.
4: $PS\_SEARCH(\mathcal{T}')$.

4.2.1 $P\_CONTRACT$ algorithm

The $P\_CONTRACT$ operation strengths the directional arc consistency in the input hypertree. It is based on two operations, P_RAKE (for Parallel RAKE), and $P\_COMPRESS$ (for Parallel COMPRESS), where $RAKE$ and $COMPRESS$ are the basic operations of a parallel tree contraction [9]. We explain these two operations.

P_RAKE **operation**: is based on the fact that the existence of only one resource is the reason of the sequential execution. This resource is the parent node relation. This resource must be manipulated simultaneously by all processors. Therefore, it must be shared between them. Thus, this technique consists in partitioning the parent relation between all processors which compute the semi-join operation of this last relation with its son nodes relations. These partitions are manipulated periodically between processors.

$P\_COMPRESS$ **operation** : is a pipelined execution of COMPRESS operation. It is applied on a sequence of nodes or a chain. Let $\{n_1, n_2, ..., n_j\}$ be a chain of a tree $\mathcal{T}$ where n_{i+1} is the n_i's only son, the application of $P\_COMPRESS$ to this chain results in a new tree contracted $\mathcal{T}'$ in which the node n_1 is transformed thus:

- $\chi_{T'}(n_1) = \chi_T(n_1)$

- $Hrel_{T'}(n_1) = \Pi_{\chi(n_1)}(Hrel_T(n_2) \bowtie Hrel_T(n_3) \bowtie \cdots \bowtie Hrel_T(n_j))$.

It makes a pipeline of join operations between the chain's nodes, and at each time when it takes a tuple from the probe relation, it joins it with all tuples of the corresponding partition in the hash table of the build relation. Then it communicates it to the probe relation of the next operator. After the execution of the last operator, it projects the tuples results on the variable set χ of each node of the chain, it hashes it on the intersection variables of each node with its parent node, and puts the tuple result in the corresponding node relation.

4.2.2 PS_SEARCH algorithm

PS_SEARCH algorithm is the parallel version of S_SEARCH algorithm. It works in the same way, except that, each time, after taking a tuple from a parent node, the research of corresponding tuples in the son nodes is made in parallel.

4.3 Complexity of P_HTR

For a binary hypertree, the complexity of the P_SBR algorithm is

$$O(r * P^{h-1}) \tag{3}$$

using $O(h)$ operations in a PRAM of type EREW. r is the the maximum relation size, h is the hypertree width, P is the maximum number of tuples in the partition of hashing on the variables of ϑ (ϑ represents the intersection variables of $\mathcal{C}_i$ with all variables in $\mathcal{C}_j$ where $j < i$). For the $P\_Solving$ algorithm, its complexity is $O(log\frac{n}{2})$ using $O(n)$ operations. n is the hypertree size (number of nodes).

5 Experiments of parallel algorithm

We have developed a simulation model for P_SBR and $P\_Solving$ algorithms based on logical time assumptions.

5.1 Simulation model

Our system is composed of simple modules described as classical sequential programs, which are collected and executed in a parallel way by the simulator. A module is the set of operations performed by a processor we called a *process*. For the internal working of a process, we split operations that can be performed in three main types : *Reading* a tuple, *Searching* in hash tables including the tuples join operation and *Writing* a result tuple. These operations are executed sequentially between them and in parallel with those

of the other processes. Each process explores the maximum of potential parallelism if its environment (its input variables) do not conflict during the other process execution. For a pipeline execution, if the processes are not adjacent, there is no conflict. Otherwise, we use the semaphore mechanism to synchronize them.

5.2 Simulation process

Each operation or each event is performed in an interval of time and the simulator must therefore keep a logical time counter. The asked question is : When we increment this counter, and what is the relation between this counter and the events execution time (physical counter)? We make a step-by-step simulation so that for each step (logical time unit) all operations which can be performed at the same time start as follows :

1. The simulator executes all processes and it increments the logical time counter at each iteration.
2. For all processes that are not in conflict, the simulator increments the physical time counter.
3. After a logical time step, the simulator updates all the shared variables.
4. It restarts a new step by considering the new variable states.

5.3 Experimental protocol

We developed a simulation of P_HTR algorithm in order to check: first, if it gives good temporal and spatial complexities and second, if the new approach of parallel tree contraction proposed in this paper can be considered as an interesting alternative of the one considered in [9]. In order to compare our results, we simulate $\mathcal{PTAC}$ algorithm. In this section, experimentations are divided in three categories :

1. Simulation of P_SBR algorithm in order to estimate its interest in term of space memory, since the sub-problems resolution is the source of a bad spatial complexity in general.
2. Simulation of P_RAKE in order to evaluate its performances.
3. Simulation of *P_Solving* algorithm in order to estimate its temporal optimization, and to measure the contribution of the new parallel tree contraction technique *P_Rake* by leading a comparison with the one proposed in ($\mathcal{PTAC}$ algorithm).

The simulation has been accomplished on a PC under *Linux 6.0.52 version* with CPU *Intel Pentium IV 2.4 GHz* and *512 Mb* of RAM. We use a

set of benchmarks *CSP* which exist in the literature. We take the average of the data obtained after several executions. In our results, $|V|$, $|E|$ et $|r|$ represent respectively the variables number, the constraints number and the maximum relations cardinality. N_d, N_f and HTW are respectively, the nodes number, the leaf nodes number and the hypertree width. N_p represents the processors number and NbS represents the number of pipeline stages.

In order to be hardware independent, measure units are *Tuple* for space and Operation (Reading, Searching or Writing) for time.

5.4 P_SBR algorithm simulation

The table 2 presents the **space memory** required for a pipeline (R_P_SBR) and no pipeline (R_join) execution of the join operations for different *CSP* instances (consistent and inconsistent).

CSP								
Name	|V|	|E|	Nd	HTW	$\|R\|$	$NNW > 2$	$\|R\_join\|$	$\|R\_P\_SBR\|$
geom-30a-4	30	81	70	4	4	20	12168	36
haystacks-07	49	153	144	4	9	27	109368	48
Renault	101	134	81	2	48721	3	979	3
pret-60-25	60	40	28	5	4	19	3288	40
pret-150-60	150	100	50	5	4	50	6136	114
pret-150-75	150	100	54	5	4	54	6552	127
queen-5-5-3	25	160	7	10	2	7	1582	38
haystacks-06	36	95	88	3	8	18	7870	18
langford-2-4	8	32	31	4	8	5	3520	9
pigeons-7	7	21	19	3	6	4	6600	7
series-6	11	30	27	3	30	5	2920	5
queen-12	12	66	61	6	12	6	36180	24
mug-100-25-4	100	166	133	3	31	32	26762	32

Table 2: P_SBR algorithm simulation of $\mathcal{CSP}$ instances

$NNW > 2$ denotes the number of nodes in the hypertree with width more than 2. We observed that the gain in space memory is very important in instances for which the pipeline executions number (NNW>2) is important (*haystacks-07*). We also observed that the gain depends on the number of nodes in hypertree (*mug-100-25-4*). It is also proportional to the number of tuples by relation. As the two factors that influence on *CSP* problems complexity are the constraints relations size, and the structural decomposition width, and in order to estimate the contribution of P_SBR, we evaluate in figure 1, in (a) and (b), the P_SBR algorithm behavior w.r.t respectively the constraints relations size and the stages number in pipeline for two classes of *CSP* problems: *aim-100-6* and *full-Insertion*.

N.B: P_SBR algorithm forms a pipeline in each node of the hypertree. So, the stages number in the pipeline is the number of constraints in the

node minus one.

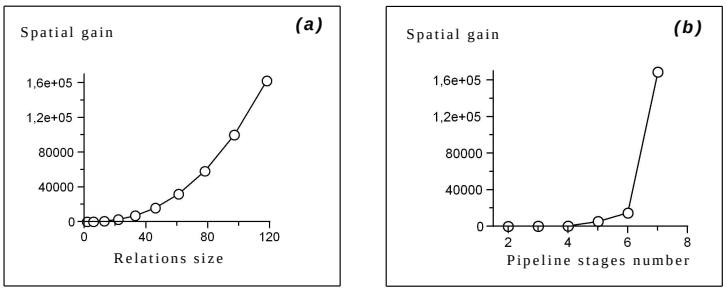

Figure 1: spatial performances of P_SBR according to (a) relations size (b) stages number in pipeline

Figure 1 shows that more the relations size is important more the memory space gain is considerable. A similar interpretation is illustrated for the stages number in pipeline.

5.5 P_RAKE algorithm simulation

P_RAKE tries to optimize the CPU time of the whole problem. We present here a simulation of this operation by computing its efficiency $Eff = \frac{T_s}{N_P * T_p}$where N_P is the processors number and T_s(reps. T_p) the time of sequential (reps. parallel) resolution of the sub- hypertree.

Sub-hypertree					
n	N_p	$\|R\|$	T_s	T_p	Eff
1	2	196	1034	550	0.94
2	4	4500	230720	59974	0.96
3	3	27000	31401	10911	0.96
4	4	12000	95095	24875	0.95
5	5	18000	156884	33061	0.95
6	6	680	9812	1659	0.98
7	6	87808	1055270	176943	0.99
8	8	7776	90125	12252	0.92
9	9	18144	297846	34410	0.96
10	10	46656	820947	84798	0.97

Table 3: P_RAKE algorithm efficiency

Table 3 presents the obtained results for some sub-hypertrees of different *CSP* instances. We observed a good efficiency of P_RAKE operation from 1 to ten processors. This algorithms allows a load balancing by sharing tuples to be filtered between all processors.

N.B: The number of processors used to contract a sub-hypertree is equal to the number of semi-join operations which is equal to the stages number

in pipeline.

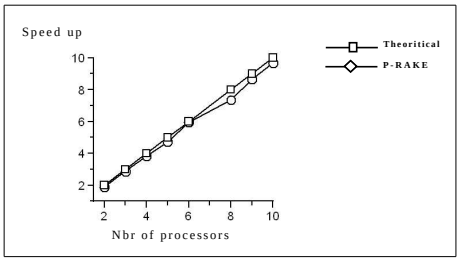

Figure 2: P_RAKE algorithm speedup

The P_RAKE algorithm speedup is presented in Figure 2. It is obvious that it is close to theoretical speedup (linear).

5.6 $P\_Solving$ algorithm simulation

Before comparing $P\_Solving$ with A_HBR, it seems natural to compare $P\_CONTRACT$ algorithm with an algorithm applying the tree contraction.

CSP							$A-HBR$	PTAC		$P-Solving$	
Name	$\|V\|$	$\|E\|$	N_d	HTW	$\|R\|$	N_f	T	N_c	T	N_c	T
hayst06	36	95	88	3	8	23	16736	13	253	10	158
dom200	100	100	50	2	200	1	3632	48	3632	48	3632
hanoi-6	62	61	59	1	2148	2	260	37	169	37	169
mug25-4	100	166	133	3	31	26	119	12	47	9	38
myc-3-4	11	20	15	4	12	4	267	6	164	3	42
myc-4-4	23	71	48	6	6	12	124	8	315	4	299
Renault	101	134	81	2	48721	12	7758	26	38167	9	1456
series-6	11	30	27	3	30	5	622	8	256	4	155
hole-6	42	133	132	2	8	127	1970	85	1201	1	29

Table 4: *P-Solving* algorithm simulation

N_c represents the number of contraction operations performed and T the computational time. We have considered 1000 operations as a measure unit. Table 4 gives the CPU time of both A_HBR algorithm and $PTAC$ algorithm (CONTRACT operation) and the one of $P\_Solving$ ($P\_CONTRACT$ operation). It results that $P\_CONTRACT$ generates a gain in temporal complexity inversely proportional to the number of contractions. More the number of the leaf nodes contracted is big more the number of contraction is less and more the gain in time is important (*myc-3-4*, *Renault* and *hole-6*), and vice versa, which is the case for *hayst06* and *mug25-4*. We can also remark for the problem *hole-6*, the CPU time gain (29 instead of 1201) is considerable. This is due to the fact the hypertree is contracted on only one step, instead of taking 81 steps.

6 Conclusion

In this paper, we have presented sequential and parallel algorithms to solve CSPs by exploiting their structural properties. The algorithms exploit the hash technique. For the sequential algorithm, we have done some experiments on benchmarks from the literature and the results we have obtained are promising. Good results are observed when the number of tuples of relations is important. The parallel algorithm is based on the notion of pipeline and parallel tree contraction which is a parallel version of the well known tree contraction. We performed simulations on some benchmarks and the results are very good. Future works include the execution of our parallel algorithm on a parallel machine.

References

[1] Gyssens, M., Cohen D., Jeavons, P.: A unified theory of structural tractability for constraint satisfaction problems. J. Comp. and Sys. Sci. 74, 721–743 (2007)

[2] Dechter, R.: Constraint Processing. Morgan Kaufmann, (2003)

[3] Dechter, R.: Constraint Networks. Encyclopedia of Artificial Intelligence, pages 276-285, (1992)

[4] Dechter, R., Pearl, J.: Tree clustering for constraint networks. Art. Int. 38, 353–366, (1989)

[5] Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural csp decomposition methods. Art. Int. 124, 243–282 (2000)

[6] Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. Art. Int. 66, 57–89 (1994)

[7] Montanari, U.: Networks of constraints: Fundamental properties and applications to pictures processing. Inf. Sci. 7, 95–132 (1974)

[8] XML Representation of Constraint Networks Format XCSP 2.1, http://www.cril.univ-artois.fr/CPAI08

[9] Karp, R. M. and Zhang, Y.: Randomized Parallel Algorithms for Backtrack Search and Branch-and-Bound Computation, Journal of the Association for Computing Machinery, 40, 3, 765–789 (1993)

[10] Gottlob, G. and Leone, N. and Scarcello, F.: Hypertree Decompositions: A survey, Proceedings of MFCS '01,(2001),