2015 IEEE 27th International Conference on Tools with Artificial Intelligence

Using Parallel Singleton Arc Consistency to enhance Constraint Solving

Nebras GHARBI
CRIL - CNRS UMR 8188,
Artois University,
Rue de l'université, 62307 Lens cedex, France
Email: gharbi@cril.fr

***Abstract*—In the latest years and with the advancements of the multicore computing world, the constraint programming community tried to benefit from the capacity of new machines and make the best use of them through several parallel schemes for constraint solving. In this paper, we present a new approach using parallel consistencies to enhance the classical solving process. Specifically, we propose an approach where a master process tries to solve a constraint satisfaction problem while using the results of consistency tests done by auxiliary workers so as to avoid some useless branching.**

INTRODUCTION

In constraint programming, several search/inference algorithms were proposed in order to speed up the search process sequentially. With the development of parallel programming techniques, the Constraint Satisfaction Problem (CSP) community tried to benefit from such parallel architecture when solving constraint problems. An interesting survey of parallel methods in the literature for constraint solving is proposed in [1].

We can distinguish two main categories of approaches for parallel constraint solving: search space splitting and portfolios. The first category consists in distributing different parts of the search tree over available cores [2], [3], [4], [5], [6]. As an early work, in [7], Schulte used a work splitting technique to reach a linear speed-up. Work stealing, a more popular approach, handles the issue of balancing the load of workers dynamically. This technique is used in several solvers like Comet or PMiniSAT. In [8], Bordeaux et al. proposed a distributed based approach that splits the search space by adding "hashing" constraints to the original model in order to avoid communication between workers. More recently, Régin et al. [9] proposed another approach called embarrassingly parallel search, which decomposes the initial problem into many small sub-problems that are available to the workers. Because there are many sub-problems, a good balancing is observed in practice.

In the second category, solvers compete against each others for solving problems and may also cooperate via message passing operations in order to find a solution like, e.g., in [10], [11]. To the best of our knowledge, there are two CSP portfolio solvers CPHydra [12] and Sunny-CP [13].

In this paper, we propose to explore another way to use a parallel architecture in which a main solver is helped by side workers that partially establish some consistencies, which are otherwise too heavy to be maintained by the main solver.

This paper is organized as follows. We describe the architecture of our system in Section II. Next, we explore both master's and workers' side respectively in Section III and Section IV. In Section V, we explain how our architecture is supposed to enhance the search process, before concluding with experimental results in Section VI.

I. TECHNICAL BACKGROUND

A (discrete) *Constraint Network* (CN) N is a finite set of n variables, denoted by $vars(N)$, "interconnected" by a finite set of e constraints. Each *variable* x has a *domain* which is the finite set of values that can be assigned to x. The *initial* domain of a variable x is denoted by $dom^{init}(x)$ whereas the *current* domain of x is denoted by $dom(x)$; we always have $dom(x) \subseteq dom^{init}(x)$.

Each *constraint* c involves an ordered set of variables, called the *scope* of c and denoted by $scp(c)$, and is semantically defined by a *relation*, denoted by $rel(c)$, which contains the set of tuples allowed for the variables involved in c.

Definition A constraint c is *Generalized Arc Consistent* (GAC) iff $\forall x \in scp(c), \forall a \in dom(x)$, there exists at least one support for (x, a) on c. A *Constraint Network* N is GAC iff every constraint of N is GAC.

Enforcing GAC is the task of removing from domains all values that have no support on a constraint. Many algorithms have been devised for establishing GAC according to the nature of the constraints.

Definition A constraint c is *Singleton Arc Consistent* (SAC) [14] iff $\forall x \in scp(c), \forall a \in dom(x)$, $GAC(P|_{x=a}) \neq \bot$ such that $P|_{x=a}$ is the obtained constraint network P' after reducing $dom(x)$ to a. A *Constraint Network* N is SAC iff every constraint of N is SAC.

We now introduce the concepts of *literal* and *level* useful to describe our approach.

Definition A *literal* is a pair (x, a_i) where x is a variable of the problem ($x \in vars(N)$) and a_i is a value of the domain of x ($a_i \in dom(x)$).

Definition A search tree is composed of different *levels* which represent the number of variables composing the problem ($vars(N)$). Each *level* corresponds to an assignment of a

1082-3409/15 $31.00 © 2015 IEEE
DOI 10.1109/ICTAI.2015.17

variable. Different variables can be assigned in the same level but in different branches of the search tree.

II. Architecture

Our approach is based on a master/workers architecture where the master is a sequential CSP solver and the different workers help their master during the search process. This main solver transmits its current instantiation to its side workers, which will try to infer relevant information by exploiting different levels of consistencies. As soon as new facts are discovered, they are transmitted to the main solver that takes them into account as soon as possible. Our goal is to have a synchronization between the main solver and the side workers as lightweight as possible. In this current work, both the main solver and the side workers (threads running on different cores) run on the same host.

Singleton Arc Consistency is a strong consistency that is very expensive to enforce during search [15]. Since we use a parallel architecture, we can benefit from its power in order to enforce SAC by several workers. The values that are discovered SAC-inconsistent are transmitted to the master in order to avoid them in its future assignments. In fact, each master and worker has its own copy of the problem. The master solves the problem whereas the workers enforce SAC on some values of the problem which are called *literals*. With each master and worker is associated an *Assignments Stack*. The master stores all decisions made in its own stack while the workers only copy them to their stack, each time there are new decisions. This stack describes the state of the problem of each entity. Workers get literals from a *Literals Queue* which is a shared data structure between all of them. If enforcing SAC on these literals produces new facts, these latters are put into the *Messages Queue* which is a shared data structure between the master and the workers. The master in his turn extracts messages from the *Messages Queue* in order to exploit them in its solving process by avoiding failures.

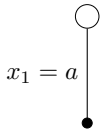
III. Master's side

In our approach, the master runs a classical CSP solver using only two additional data structures: *Messages Queue* and *setOfInferences*. In this section, we, first, explain the decisions storage in the *Assignments Stack*. Then, we explain how the messages are managed thanks to the *Messages Queue* and the *setOfInferences* data structures.

A. The Assignments Stack

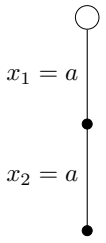
The *Assignments Stack* stores all decisions made by the solver. They might be positive (assignment) or negative (refutation) decisions. Each positive decision defines a level to which corresponds a time-stamp indication at which time this assignment was taken. This time-stamp is given by a global counter (on 64 bits to avoid any overflow) which is initialized to 0 and incremented each time a decision is taken. This information is used to identify the modifications of the *Assignments Stack* and is further described from the workers side. In fact, a time-stamp is only associated with positive decisions which is a feature of AbsCon [16] solver. Positive decisions are stored in the stack in a consecutive order. When a backtrack occurs, the value causing it is removed from the domain of its variable and this refutation is added to the previous level.

Figure 1 gives an illustration of the *Assignments Stack* management when different decisions are made. Positive decisions, which correspond to levels of the search tree, are stored in the stack in a consecutive order. When a backtrack occurs, e.g. $x_2 \neq a$ in Figure 1c, the value causing the backtrack is removed from the domain of its variable and this refutation is added to the previous level. The time-stamp of the last positive decision before backtrack does not change.



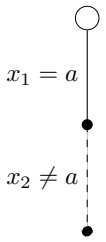
Level	Time-stamp	Decision
1	$ts = 1$	$x_1 = a$

(a) $x_1 = a$.



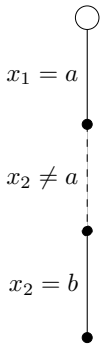
Level	Time-stamp	Decision
1	$ts = 1$	$x_1 = a$
2	$ts = 2$	$x_2 = a$

(b) $x_2 = a$.



Level	Time-stamp	Decision
1	$ts = 1$	$x_1 = a$ $x_2 \neq a$

(c) $x_2 \neq a$.



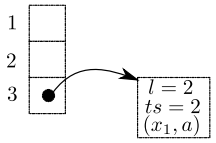
Level	Time-stamp	Decision
1	$ts = 1$	$x_1 = a$ $x_2 \neq a$
2	$ts = 3$	$x_2 = b$

(d) $x_2 = b$.

Fig. 1: Master's side: managing the *Assignments Stack*.

B. The Messages Queue and the Set Of Inferences

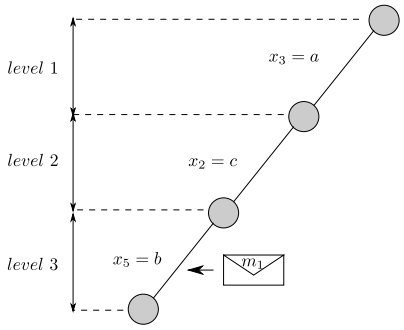
Before a decision is taken, the master checks if some information has been inferred by the other workers. This is



(a) The *setOfInferences*.

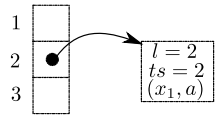
Level	Time-stamp	Decision
1	1	$x_3 = a$
2	2	$x_2 = c$
3	3	$x_5 = b$
		$x_1 \neq a$

(b) The *Assignment stack*.



(c) The search tree.

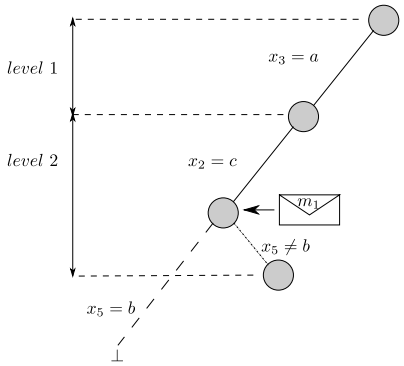
Fig. 2: A master's search tree at level 3.



(a) The *setOfInferences*.

Level	Time-stamp	Decision
1	1	$x_3 = a$
2	2	$x_2 = c$
		$x_5 \neq b$
		$x_1 \neq a$

(b) The *Assignment stack*.



(c) The search tree.

Fig. 3: The master backtracking to level 2.

done by checking the content of the *Messages Queue*. When a message is extracted from this stack, the master first checks if the inference is still relevant in the current state of the solver (the message could be outdated because the master backtracked in between). According to the master, a message is still relevant iff the level of the test is *higher* (in the search tree) than the master solver level and the time-stamps are coherent (detailed in Algorithm 1). If the message is not relevant, it is ignored. Otherwise, the inference is taken into account.

Algorithm 1: isValidMessage(*msg*: Message): Boolean

```
1 if msg.level ≤ currentLevel and
    msg.timeStamp = timeStamp[msg.level] then
2     return true // the message msg is still
      relevant;
3 return false;
```

In fact, a message contains three main components:

- *level*: the level in which the property "the literal is SAC-inconsistent" is relevant;
- *timeStamp*: the time-stamp of *level* when the SAC test was enforced;
- *literal*: the pair (variable,value) to avoid in future assignments.

Once the master figures out that the message is still relevant, it stores it in its own data structure, called *setOfInferences*. In fact, *setOfInferences* is a table where *setOfInferences*[i] defines the messages extracted and used at level i. Thanks to this data-structure, the master can re-use these inferences in higher levels, when a backtrack occurs. Figures 2 and 3 describes the use of *setOfInferences*. At level 3, the master receives a new message m_1=$\{$*level*=2, *timeStamp*=2, *literal*=$(x_1, a)\}$. This message is relevant with respect to the master current state. The master can, thus, benefit from this message by deleting this value (as long as the master does not backtrack). Value a is, then, removed from $dom(x_1)$ and the message is stored in *setOfInferences*. It happens that a backtrack occurs to the level 2 which implies the restoration of the previous state of the problem at level 2: the inference application of the message m_1 is undone when backtracking. Since m_1 remains relevant after backtracking, the master re-applies it at level 2.

Algorithm 2 describes how the master manages the messages received from the different workers and the inferences already stored in *setOfInferences*. In fact, the master checks *setOfInferences* only if a backtrack occurs (Lines 1 - 7). The *hasBacktracked* parameter indicates iff the master has backtracked or not. If it is the case, the master re-applies all the inferences stored starting from its current level after backtracking. After, eventually, checking the stored inferences, the master checks the new messages in *Messages Queue*. It,

Algorithm 2: applyMessages(*hasBacktracked*: Boolean)

```
1  if hasBacktracked then
2      foreach i ∈ [currentLevel+1,levelBeforeBacktrack] do
3          foreach msg : setOfInferences[i] do
4              if isValidMessage(msg) then
5                  remove literal.val from dom(literal.var);
6                  add msg to setOfInferences[currentLevel];
7          clear setOfInferences[i];
           // removing the inferences stored at level i
8  foreach msg : msgStack do
9      if isValidMessage(msg) then
10         remove literal.val from dom(literal.var);
11         add msg to setOfInferences[currentLevel];
12     remove msg;
```

first, verifies if the message is relevant. If it is the case, the literal is removed and the inference is added to *setOfInferences* at the current level of search (Lines 5 - 6 and 10 - 11). If the message is no more relevant, it is removed from the Messages Queue (Line 12).

IV. Workers' side

The workers use three main data structures.

A. The Assignments Stack

The workers obtain initially a copy $\mathscr{P}'$ of the problem instance $\mathscr{P}$ that is handled by the master. In a loop, they copy incrementally the *Assignments Stack* of the master (i.e. the list of decisions taken by the master), reproduce the filtering made by the master after these decisions in order to reach the same state as the master and then run their own (partial) consistency. The incremental copy of the *Assignments Stack* consists in, first, identifying the part of the *Assignments Stack* which is identical in the master and in the worker's copy and then copying every decision taken by the master after this common part. It is performed in the following way. The worker first obtains the current stack pointer of the master and then identifies the last decision which has the same time-stamp in the master and in the worker. Then, the worker copies each decision of the master after this last common decision. Since the master may have backtracked in between, the worker then checks that the last decision that it copied still has the same time-stamp as in the master. If this is not the case (i.e. the master backtracked during the copy), the worker restarts its copy of the *Assignments Stack*. In practice, this does not happen too often (less than 1% of the total copies). Therefore, our approach guarantees that inconsistent copies are never used because when such a copy is obtained, the worker simply asks for a new one.

Figures 4 and 5 describe two cases of the copying process. In Figure 4, the master continued its solving process and, thus, there are new decisions to be copied in the worker's *Assignments Stack*. The last common positive decision between the master and the worker is $x_1 = a$. All decisions made after this one are added to the worker's stack.

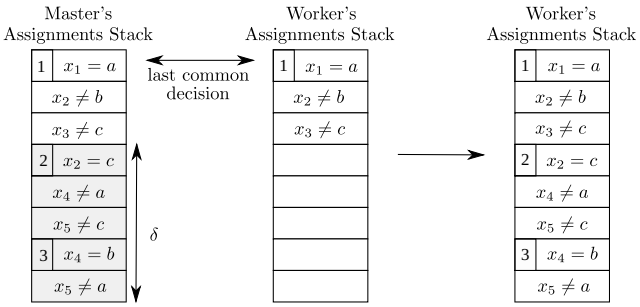


Fig. 4: Copying process (case1: the same branch).

In the case of Figure 5, the master has backtracked and started new other decisions. The worker has to remove all decisions until the common one. In fact, a backtrack on the variable x_2 has occurred and, thus, a negative decision is added to the first positive decision ($x_1 = a$). The worker has to update its stack by removing the old decisions that are no more relevant, add the new negative decisions of the last common positive decision ($x_1 = a$ in our case) and then add incrementally the new ones (including refutations).

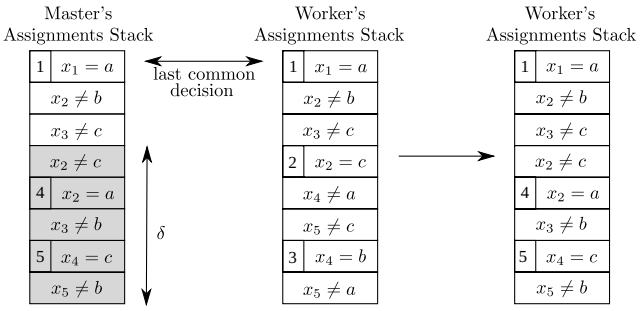


Fig. 5: Copying process (case2: backtracking).

B. The Messages Queue

Once a worker has a stable copy of the master's state, it picks a literal (x, a) from a global queue of literals (described in Section IV-C) and checks if (x, a) is SAC-consistent, i.e., if no domain wipe-out (global inconsistency) is detected when taking the decision $x = a$ and running constraint propagation, we assume that GAC is achieved. If it is not, the worker places in the *Messages Queue* the data $x \neq a$ together with a decision level and a time-stamp where this information could be inferred (as detailed in Section III-B). After testing SAC on a literal (x, a), it is put back to the literals queue in order to test its consistency later.

Algorithm 3 describes the SAC enforcing on the literals set of the queue. We just focus, in this section, on the message producing process. When a literal $(variable,value)$ is SAC-inconsistent (the SAC test returns *false*), it means that removing this literal enables the master to avoid some useless

Algorithm 3: enforceConsistencyOnLiterals(P_i: Constraint Network of slave i, *LiteralsQueue*: Queue of literals)

```
1  valueFound ← false;
2  while ¬valueFound do
3      literal ← getLiteral(literalsQueue);
4      if ¬assigned(literal.var) and
       |dom(literal.var)| > 1 then
5          valueFound ← true;
6      else
7          literal.countdown- -;
8          insert literal to literalsQueue;
   // while the problem is not solved,
   there is always a value to test
9  assign literal.val to literal.var;
10 consistent ← GAC(P, literal.var);
11 backtrack();
   // going back to the previous level in
   order to test more pairs
12 if ¬consistent then
13     literal.priority++;
14     putMessage(currentLevel, timeStamp[level],
       literal);
       // Putting the inferred result in
       the queue of messages
15 literal.countdown- -;
16 insert literal to literalsQueue;
```

search sub-tree and, thus, speeds up the solving process. The worker transmits this information to its master (Line 14). A message is then sent to the master indicating the literal to avoid, the current level of the worker where the SAC test is done and its time-stamp.

C. The Literals Queue

In order to perform the SAC tests, the set of literals of the problem are put together in a *Literals Queue*. The different workers cooperate to examine each possible literal as often as possible. Since some literals are more likely to become SAC-inconsistent, we use priorities in order to test these literals more often than the other ones. Therefore, to each literal in the *Literals Queue* is associated a priority, which is incremented when the literal is identified as SAC-inconsistent (Line 13 of Algorithm 3). To manage this priority, we use the Completely Fair Scheduling (CFS) algorithm [17] used in Linux scheduler. The CFS scheduler ensures executing the process that has used the least amount of time at the first place and, thus, the processes are ordered according to the execution time spent.

In our approach a literal has two main properties:

- *countdown* counter: it indicates the number of times a literal could be tested. Each time we test a literal, we decrease its *countdown* counter (Line 15 of Algorithm 3);
- *priority* counter: it indicates how interesting a literal is. If a SAC test proves the inconsistency of a literal, its priority counter is increased (Line 13 of Algorithm 3). It is important to test the same literal in other branches of the search tree since it may infer important information to be exploited by the master.

Workers perform the SAC tests as follows: they first get a literal and remove it from the queue. Then, they check if the variable is already assigned or is a singleton variable (Line 4 of Algorithm 3). If it is the case, it is useless to enforce SAC on this literal and they put back the literal in the queue (by decreasing its priority). Otherwise, we perform the test.

V. Enhancing the search process

The search algorithm used by the master is the classical MAC search algorithm which maintains GAC during search. Added to that, we include extracting messages before each assignment: a call of Algorithm 2 is made in order to benefit from the facts discovered by workers. In this way, the SAC-inconsistent values are removed from their respective domains which enables to avoid them.

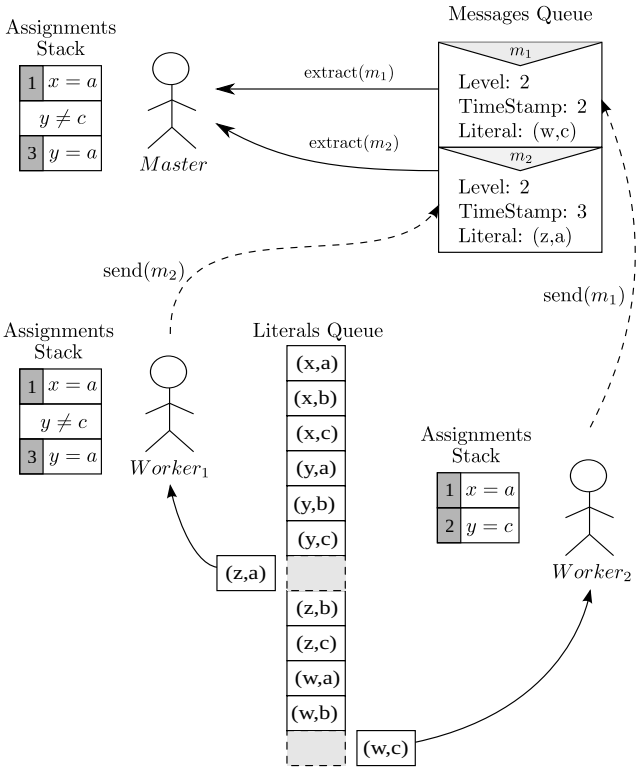


Fig. 6: Illustration of the architecture.

Figure 6 illustrates an example where the master has taken two positive decisions so far. The first assignment $x = a$ is made at time-stamp 1. Then, the master tried $y = c$ at time-stamp 2, but immediately obtained an inconsistency. Therefore, it removes this decision and adds the negative decision $y \neq c$ to the previous level. The latest decision is $y = a$ at time-stamp 3. $Worker_1$ has copied the master's stack at this point, while $Worker_2$ has copied the stack at an earlier point where

the decision $y = c$ was taken. Each worker picks a literal from the *Literals Queue*. $Worker_2$ picks (w, c) and detects that is SAC-inconsistent. A message m_1 indicating that $w \neq c$ can be inferred at decision level 2 (with time-stamp 2) is posted. $Worker_1$ tests (z, a) and proves that it is SAC-inconsistent. Therefore, it posts a message m_2 indicating that $z \neq a$ can be inferred at decision level 2 (with time-stamp 3). The master finds out, when extracting messages, that m_1 is outdated because it has backtracked since $Worker_2$ has copied the decisions of its *Assignments Stack*. The message m_1 is then removed. The message m_2 is relevant with respect to the problem state of the master. This latter can benefit from this information until it changes its decision at level 2.

VI. Experimental results

In order to test the practical interest of our approach, we have conducted experimentation with AbsCon solver using a cluster of bi-quad cores Xeon processors at 2.66 GHz node with 16GiB of RAM under Linux. We have used the search algorithm MAC, equipped with $dom/ddeg$ [18] as variable ordering heuristic and $lexico$ as value ordering heuristic, to solve instances of different problems[1]. The choice of variable ordering heuristic is made in order to guarantee that the different approaches have the same search path. In Table I, we have compared the wall-clock time (wck) in second(s) and the number of nodes (#nodes) for both sequential approaches and the parallel ones. For the sequential approaches, we have used MAC, SAC+MAC where SAC is enforced only at pre-processing, and MSAC that enforces SAC during the search. For the parallel approach, we have used 1, 3 and 7 workers (and of course, one master), and the number of useful messages sent from the workers to the master is displayed (#msgs). A time-out of $1,800$ seconds was set per instance ; when an instance cannot be solved within this limit, TO is indicated in the table. In our tests, we just focused on *wck time* as we supposed that we can waste *CPU time* as possible.

Compared with a MAC solver, the interest of using a parallel approach is visible on these instances. The number of visited nodes is highly decreased for *queen-12-12-14*, *cc-10-10-2* and *qk-12-5-mul*. The wall-clock time is also decreased for *cc-10-10-2* and *qk-12-5-mul*. One interest of our approach is that SAC tests are interleaved with search, which means that we do not have to wait for the completion of the pre-processing to start the actual search, and still, we benefit from the discovery of inconsistent values. In a sense, we use a kind of *anytime* version of SAC.

We investigate the efficiency of used messages on the speed-up obtained when using our parallel approach compared to the classical MAC solver. For the Crossword instance (cw-ogd-vg12-15), the inferences made by the workers are too limited for being useful compared with a MAC solver. Even though, using a parallel approach remains interesting compared to SAC+MAC or MSAC in term of wall-clock time. On this instance, it appears that, for SAC+MAC, the most of wck time (more than 368 seconds) is spent at pre-processing. This explains why MSAC is not able to solve the problem in $1,800$ seconds. For the Chessboard Coloration instance (cc-10-10-2), the parallel approach achieves the best wall-clock time with an average of messages number around $20,000$. For the Langford and Queen instances, the number of messages is greater than the Chessboard Coloration (cc) instance. However, this does not imply in anyway that the parallel approach is quicker. To conclude, analyzing these results shows clearly that the speed-up obtained is not correlated with to the number of used messages when using our parallel approach compared to the classical MAC solver.

[1]available at http://www.cril.univ-artois.fr/∼lecoutre/benchmarks.html

Instance		#nodes	wck time	#msgs
cc-10-10-2	MAC	542,776	47.899	
	SAC+MAC	542,776	48.23	
	MSAC	**4960**	49.091	
	Par(7)	73,684	20.846	23,979
	Par(3)	139,351	**19.49**	22,240
	Par(1)	315,035	33.774	16,512
cw-ogd-vg12-15	MAC	4,224	**60.62**	
	SAC+MAC	4,648	418.031	
	MSAC	TO		
	Par(7)	**4116**	239.897	412
	Par(3)	4241	132.056	158
	Par(1)	4216	94.94	71
langford-3-13	MAC	764,944	29.142	
	SAC+MAC	764,944	**28.531**	
	MSAC	TO		
	Par(7)	**750,105**	47.007	33,987
	Par(3)	759,166	30.465	15,073
	Par(1)	763,373	28.797	4,994
queen-12-12-14	MAC	2,211,140	**64.122**	
	SAC+MAC	2,211,140	65.649	
	MSAC	**61,179**	170.871	
	Par(7)	2,000,032	97.09	164,826
	Par(3)	2,083,863	68.492	110,223
	Par(1)	2,173,009	70.391	44,101
qK-12-5-mul	MAC	2,017,288	44.877	
	SAC+MAC	0	0.438	
	MSAC	0	**0.401**	
	Par(7)	**959**	1.69	2,568
	Par(3)	1845	0.843	2,138
	Par(1)	99,721	3.615	9053

TABLE I: Results for the sequential and parallel approaches for solving a few selected instances.

Table II describes the average wall-clock time for several series of instances. Times are mentioned for a classical MAC solver, MSAC, SAC+MAC and also our parallel approach using 7 workers. The instances that reach time-out, their wck-time is penalized by $1,800 * 3$ seconds. Our parallel approach has the best solving wall-clock time on 3 series out of 8 which outweigh the results obtained with MSAC and SAC+MAC. However, we have been disappointed in not reducing significantly the solving time compared to a MAC solver. In fact, using such a strong consistency is supposed to infer important information that can be used by a classical solver to reduce its search tree.

In order to understand the reasons behind the fruitless results (compared to MAC) of our approach, we decide to

Series	#inst	MAC	MSAC	SAC+MAC	Par(7)
langford	17	77.828	830,090	**76.995**	397.311
queen	7	845.799	4,659.207	2,443.690	**319.756**
queensKnight	22	3,446.051	**160.044**	166.904	1,475.197
rand-8-20-5-18-800	10	**20.221**	204.801	22.192	34.837
fapp25-2230	12	46.246	619.829	49.241	**43.191**
ewddr2-10-by-5	10	**1.518**	6.888	2.362	3.046
cc	10	53.004	67.663	54.017	**13,422**
BlackHole-4-4-e	10	**0.493**	11.354	0.498	1.156

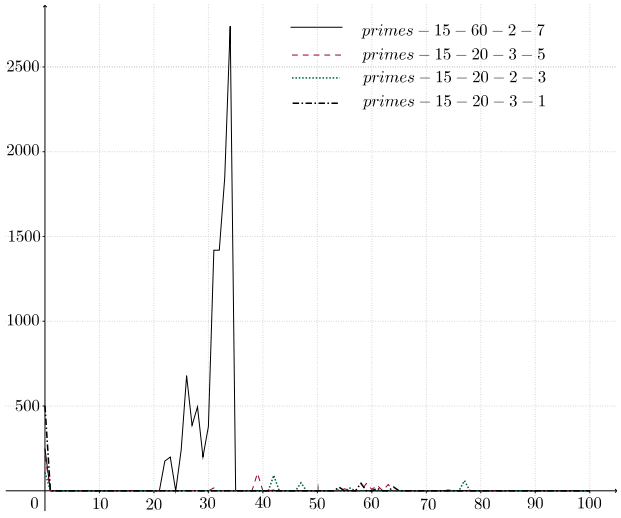
TABLE II: The average wall-clock time for series of instances with dom/ddeg.

make more experiments. On the one hand, we supposed that the SAC tests are costless (*costlessMSAC*). To do that, we stored the inferences made when maintaining classical SAC during search (MSAC). We, then, use the previous stored inferences (oracle of inferences) while maintaining arc consistency (MAC). We get, thus, the best case times of maintaining SAC during search. On the other hand, we compared *costlessMSAC* with a classical MAC solver and estimated the speed-up ratio (MAC / *costlessMSAC*). In fact, on 876 instances of several hard problems, only 3% of the instances of the conducted experimentation have a speed-up ratio greater than 10. Whereas, the other 94% of the instances have a speed-up ratio less than 2. This leads us to assume that maintaining SAC during search is not as effective as supposed in reducing the search tree. These experiments show that maintaining SAC during the search can reduce the search tree only for a few instances.

Instance	costlessMSAC	MAC	Speed-up ratio
2-fullins-5-4	11.44	13.683	1.196
langford-3-14	93.257	229.071	2.456
crossword-m1c-lex-vg6-7	16.527	51.253	3.101
bdd-21-133-18-78-11	2.427	12.224	5.037
rand-8-20-5-18-800-13	3.857	51.407	13.328
cc-10-10-2	3.088	75.784	24.541
half-n25-d5-e56-r7-1	4.703	198.238	42.151
primes-10-20-2-5	1.202	51.973	43.239
val10-43	0.772	43.983	56.973

TABLE III: The wall-clock time and speed-up for some selected instances with costlessMSAC and MAC (dom/ddeg).

Table III describes the results of some selected instances of the previous experimentation: comparing MAC to costlessMSAC. These instances are presented in an increasing order of their speed-up. Obviously, the difficulty of instance resolution does not affect the obtained speed-up ratio. For example, the *langford-3-14* instance is more difficult than *val10-43*, whereas the speedup ratio for *langford-3-14* is significantly less than *val10-43*.



(a) The effective SAC tests per level.

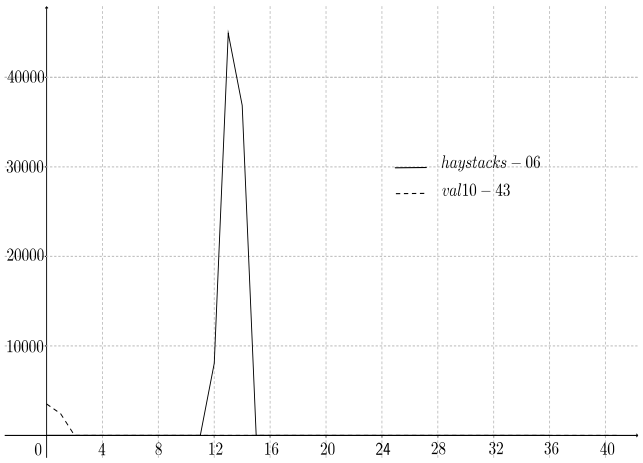
Instance	costlessMSAC	MAC	speed-up ratio
primes-15-60-2-7	1.194	37.384	31.310
primes-15-20-3-5	0.875	15.833	18.095
primes-15-20-2-3	0.756	1.254	1.659
primes-15-20-3-1	0.596	0.903	1.515

(b) The wall-clock time and the speed-up ratio.

Fig. 7: Comparison between costlessMSAC and MAC for primes instances (100 variables, domain size= 46).

Figure 7a describes the number of effective SAC tests that contribute to the reduction of the search tree and, thus, helping the solver during its solving process. These results are more detailed in Figure 7b. In fact, we chose instances composed of the same number of variables (100 per instance). All variables have the same domain size of 46. However, these instances have different speed-up. Obviously, the *primes-15-60-2-7* instance which has the greatest speed-up ratio, has also the greatest number of effective SAC tests during the 32 first levels. This number reaches 2740 tests for the level 32. For the other instances, the greatest number of effective SAC tests are mostly during pre-processing whereas the remaining SAC tests are distributed during the search in few amounts. Obviously, using a large number of SAC tests in order to reduce the search space is a property of some instances for which maintaining *costlessSAC* is efficient.

Figure 8a describes two instances with different speed-up ratio (wck-time and speed-up are detailed in Figure 8b). These results show, clearly, that the previous assumption is not always true. In fact, *haystacks-06* instance benefits from the greatest number of SAC tests during search but does not reach an important speed-up compared to MAC. However, *val10-43*



(a) The effective SAC tests per level.

Instance	costlessMSAC	MAC	speed-up ratio
haystacks-06	0.88	1.43	1.625
val10-43	0.772	43.983	56.973

(b) The wall-clock time and the speed-up ratio.

Fig. 8: Comparison between costlessMSAC and MAC for two instances of different speed-up ratio.

instance reaches a speed-up of almost 57 compared to MAC benefiting from a few amount of SAC tests at pre-processing and its first level.

Furthermore, using a high number of SAC tests at the first level of the search tree does not guarantee a high speed-up which the case for *primes-15-20-2-3* and *primes-15-20-3-1* instances (Figure 7a) compared to *val10-43* instance.

CONCLUSION

Among the various ways to parallelize the resolution of CSP instances, this paper explores an architecture where a sequential solver receives help from other workers, which establish consistencies that are too heavy to be maintained by the master alone. In our approach, the workers cooperate to establish SAC while the master keeps searching. The exchanges between the master and the workers are maintained at a minimum to limit the impact on the master. Clearly, maintaining SAC, is relevant only on few instances for which different factors are involved in the reduction of the search tree. Therefore, a natural perspective of this work is to incorporate other various consistencies in the system in order to infer wiser facts.

ACKNOWLEDGMENTS

This work has been supported by both CNRS and OSEO (BPI France) within the ISI project 'Pajero'.

REFERENCES

[1] I. Gent, C. Jefferson, I.Miguel, N. Moore, P. Nightingale, P. Prosser, and C. Unsworth. A preliminary review of literature on parallel constraint solving. In *Proceedings of PMCS'11, workshop on parallel methods for Constraint Solving*, 2011.

[2] L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 443–448, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[3] G. Chu, C. Schulte, and P. Stuckey. Confidence-based work stealing in parallel constraint programming. In IanP. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241. Springer Berlin Heidelberg, 2009.

[4] G. Chu, P. Stuckey, and A. Harwood. PMiniSAT: A Parallelization of MiniSAT 2.0. Technical report, 2008.

[5] T. Schubert, M. Lewis, and B. Becker. Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2008.

[6] F. Burton and M. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 187–194. ACM, 1981.

[7] C. Schulte. Parallel search made simple. In *Proceedings of TRICS'00*, pages 41–57, 2000.

[8] L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *Proceedings of IJCAI'09*, pages 443–448, 2009.

[9] J.-C. Régin, M. Rezgui, and A. Malapert. Embarrassingly parallel search. In *Proceedings of CP'13*, pages 596–610, 2013.

[10] Y. Hamadi. Optimal distributed arc-consistency. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming CP99*, volume 1713 of *Lecture Notes in Computer Science*, pages 219–233. Springer Berlin Heidelberg, 1999.

[11] Y. Hamadi. Distributed, interleaved, parallel and cooperative search in constraint satisfaction networks. Technical report, Information Infrastructure Laboratory, HP Laboratories Bristol, 2002.

[12] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of AICS'08*, 2008.

[13] R. Amadini, M. Gabbrielli, and J. Mauro. Sunny-cp: a sequential cp portfolio solver. SAC, 2015.

[14] R. Debruyne and C. Bessiere. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.

[15] C. Lecoutre and P. Prosser. Maintaining singleton arc consistency. In *Proceedings of CPAI'06 workshop held with CP'06*, pages 47–61, 2006.

[16] C. Lecoutre and S. Tabary. Abscon 109: a generic CSP solver. In *Proceedings of the 2006 CSP solver competition*, pages 55–63, 2007.

[17] T. Li, D. Baumberger, and S. Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 65–74, New York, NY, USA, 2009. ACM.

[18] C. Bessiere and J.-C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.