Asynchronous Consistency Maintenance with Reordering

Marius-Călin Silaghi and **Djamila Sam-Haroud** and **Boi Faltings**
Swiss Federal Institute of Technology at Lausanne
{silaghi,haroud,faltings}@lia.di.epfl.ch

Abstract

When a search problem is solved by a distributed network of agents, it is desirable that this search exploits asynchronism as much as possible. This increases parallelism, allows agents to maintain privacy and makes the process robust against timing variations. Recent work on bringing constraint satisfaction techniques to distributed environments has focused either on backtrack search or on local consistency. A high degree of asynchronism can be reached when the two techniques are combined. A generic protocol for maintaining consistency during distributed search is presented. It allows: *i)* distributed consistency to be enforced concurrently at all levels of the search trees and *ii)* decisions about instantiations and agent reordering to be taken asynchronously by the agents. We show how its completeness can be ensured during backtracking with polynomial space requirements when a large class of reordering policies is used. A comparative experimental evaluation of several instances of this protocol is presented.

1 Introduction

A constraint satisfaction problem (CSP) is defined as a set of variables taking their values in particular domains and subject to constraints that specify consistent value combinations. Solving a CSP amounts to assigning to its variables values from their domains so that all the constraints are satisfied. Distributed constraint satisfaction problems arise when the constraints or variables come from a set of independent but communicating agents.

Maintaining local consistency with dynamic reordering during backtrack search (e.g. [Sabin and Freuder, 94]) is one of the most powerful techniques for solving centralized CSPs. While in that setting, instantiation and consistency enforcement steps alternate sequentially, much more elaborate combination schemes can be proposed for distributed CSPs by enabling the agents to act asynchronously. Asynchronism is desirable since it gives the agents more freedom in the way they can contribute to search and enforce their privacy policies. Namely, before the deadline of an agent for giving some information, other agents' announcements may preempt the need of undesired disclosures. It also increases both parallelism and robustness. In particular, robustness is improved by the fact that the search may still detect unsatisfiability even in the presence of crashed agents.

In this paper we target problems with finite domains. We consider that each agent A_i wants to satisfy a local CSP, CSP(A_i). The agents may keep their constraints private but publish their interest on variables. We distinguish four facets of asynchronism. Thoroughly studied previously are:

a) *deciding instantiations* of variables by distinct agents. The agents can propose different instantiations asynchronously (e.g. Asynchronous Backtracking (ABT) [Yokoo *et al.*, 98]).

b) *enforcing consistency*. The distributed process of achieving "local" consistency on the global problem is asynchronous (e.g. Distributed Arc Consistency [Zhang and Mackworth, 91]).

No polynomial space asynchronous complete algorithm was known for:

c) *maintaining consistencies asynchronously*. Instantiations and asynchronous local consistency enforcements no longer alternate sequentially. Consistency can be enforced concurrently at all levels in the distributed search trees. Domain reductions are used as soon as available.

d) *performing reordering*. Dynamic reordering can be decided asynchronously by several agents. Asynchronous Weak Commitment [Yokoo *et al.*, 98] is a powerful technique, but requires exponential space in the number of variables for completeness.

We propose a generic protocol, Multiply Asynchronous Search (MAS), which integrates asynchronisms of type a, b, c, and d, and requires polynomial space. We denote by $MAS_{(\pm,\pm,\pm,\pm)}$ versions of MAS that behave with asynchronisms of different types. "$+$" stands for presence and "$-$" for absence of some type of asynchronism, respectively a, b, c or d, according to the corresponding position. Completeness and correctness of the protocol is demonstrated and four of its instances are experimentally evaluated.

2 Background & Definitions

Asynchronous Aggregation Search (AAS) is a general complete protocol for solving distributed CSPs with polynomial

space requirements [Silaghi *et al.*, 2000]. It allows for all known asynchronous instantiation schemes. In AAS, as presented further in this section, the agents exchange messages about sets of values for combinations of variables (aggregates). In this paper we refer to an assignment of a variable x by an agent A_i as a *proposal of A_i on x*. The next definitions are borrowed from AAS:

Definition 1 *An* **assignment** *is a triplet (x_j, s_j, h_j) where x_j is a variable, s_j a set of values for x_j, $s_j \neq \emptyset$, and h_j a history of the pair (x_j, s_j).*

The history guarantees a correct message ordering. It determines if a given assignment is more recent than another. Let $a_1 = (x_j, s_j, h_j)$ and $a_2 = (x_j, s'_j, h'_j)$ be two assignments for the variable x_j. a_1 is *newer* than a_2 if h_j is more recent than h'_j. The newest assignments received by an agent A_i define its **view**, view(A_i). An **aggregate** is a set of assignments. Let V be an aggregate and vars(A_i) the variables of CSP(A_i). The set of tuples disabled from CSP(A_i) by V is formally $\mathbf{T_i(V)}=\{t \mid t=(x_1=v_t^1, ..., x_n=v_t^n), \forall j, x_j \in \text{vars}(A_i);\ \forall u \neq j, x_j \neq x_u;\ \text{n}=|\text{vars}(A_i)|; \exists k \in [1..n], (x_k, s_k, h_k) \in V, v_t^k \notin s_k\}$.

Definition 2 *$V' \rightarrow \neg T_i(V)$ is a* **nogood entailed for $\mathbf{A_i}$ by its view V**, *denoted $\mathbf{NV_i(V)}$, iff $V' \subseteq V$ and $T(V') = T(V)$.*

Definition 3 *An* **explicit nogood** *has the form $\neg V$, or $V \rightarrow$ fail, where V is an aggregate.*

The information in the received nogoods that is essential for completeness can be stored compactly in a polynomial space structure called conflict list nogood.

Definition 4 *A* **conflict list nogood (CL) for $\mathbf{A_i}$** *has the form $V \rightarrow \neg T$, where $V \subseteq$ view(A_i) and T is a set of tuples:*
$T=\{t \mid t=(x_{t^1}=v_t^1, ..., x_{t^{n_t}}=v_t^{n_t}), \forall k, x_{t^k} \in$ vars(A_i)$\}$, such that T can be represented by the structures of a centralized backtracking algorithm.

Orders and Histories The agents communicate using channels without message loss. In order to obtain instantiation asynchronism (type a), with no infinite loops, AAS uses a strict order $\prec$ on agents as proposed for ABT. In the sequel of the paper, A_i^j denotes the agent A_i with the position (**priority**) $j, j \geq 0$, when the agents are ordered by $\prec$. If $j > k$, we say that A^j has a higher priority than A^k. A^j is then a successor of A^k, and A^k a predecessor of A^j. The asynchronism obtained in this way is restricted to proposals made on distinct variables. By also using *histories*, AAS allows asynchronous proposals on the same variables by different agents. An agent A_i^j associates a history $h_{x,a}^j$ to each proposal, a, it makes on any variable x. The history $h_{x,a}^j$ is constructed by prefixing the history of the newest assignment on x proposed by a predecessor and known to A_i^j, if any, to the pair $|j : l|$. l is the value of a counter for the number of proposals of A_i^j on x. A pair $|j_1{:}l_1|$ has priority over a pair $|j_2{:}l_2|$ if either $j_1 < j_2$, or $j_1 = j_2$ and $l_1 > l_2$.

A history h_1 is **more recent** than a history h_2 if $\exists p \geq 0$ such that h_1 and h_2 have identical prefixes of length p and h_1 contains at position p a pair having priority over a pair found at position p in h_2. Similarly, if h_1 is a prefix of h_2, $h_1 \neq h_2$ then h_2 is **more recent** than h_1. This conventions define a total order on histories. An assignment with history h_x^j built by A^j for a variable x is **valid** for an agent $A^m, m \geq j$ if no other history known by A^m and built by agents $A^k, k \leq j$ for some assignment of x, is more recent than h_x^j. A nogood is valid if all the assignments contained in its premise are valid.

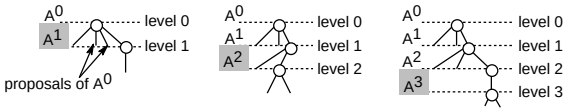

Figure 1: *Distributed search trees: simultaneous views of distributed search seen by A^1, A^2, and A^3, respectively.*

The AAS protocol is defined by the next messages[1]:

- `ok()` messages having as parameter an aggregate, V.
- `nogood()` messages announcing an explicit nogood.
- `addlink(vars)` messages transporting a set of variables. They are sent from agent A^j to agent A^i, $j > i$ and inform A^i that A^j is interested in the variables $vars$.

`ok(`V`)` messages announce proposals of domains for a set of variables and are sent from agents with lower priorities to agents with higher priorities. The proposal is sent to all successor agents interested in it. Let the set of valid assignments known to the sender A_i be denoted known(A_i). $V \subseteq$ known(A_i). Any tuple not in T_i(known(A_i)) must satisfy the local constraints of the sender A_i and its valid nogoods[2]. An agent maintains its view and a valid CL and always enforces its CL and its nogood entailed by the view. Generally, an assignment has to be built and added to V by A_i only if the newest assignment for the same variable known by A_i does not have the same set of values.[3]

`nogood` messages are sent from agents with higher priorities to agents with lower priorities. If given its constraints and valid nogoods an agent can find no proposal, in finite time it sends an explanation under the form of an explicit nogood $\neg N$ via a `nogood` message to the highest priority agent that has built an assignment in N. An empty nogood signals failure of the search. On the receipt of a valid nogood that negates its last proposed aggregate, V, an agent knows that proposal V is refused. Any received valid explicit nogood is merged into the maintained CL using an inference technique.

Distributed Consistency (DC) A centralized local consistency algorithm prunes from the domain of variables locally inconsistent values. A computed restricted domain is called **label**. Let the **union** of CSPs and constraints be a CSP containing all the constraints and variables referred in arguments. Let P be a Distributed CSP with the agents $A_i, i \in \{1..n\}$ and let $C(P)$ be $\cup_{i \in \{1..n\}}$CSP(A_i). Let $\mathcal{A}$ be a centralized local consistency algorithm (e.g. arc-, bound-consistency). We denote by DC($\mathcal{A}$) a distributed consistency algorithm that computes by exchanging value eliminations the same labels for P

[1] Some technical details from AAS are omitted.

[2] Except for constraints about which A_i knows that a successor enforces them (as in ABT).

[3] Exceptions appear for the first proposal made by A_i after nogoods of certain types are discarded (see AAS).

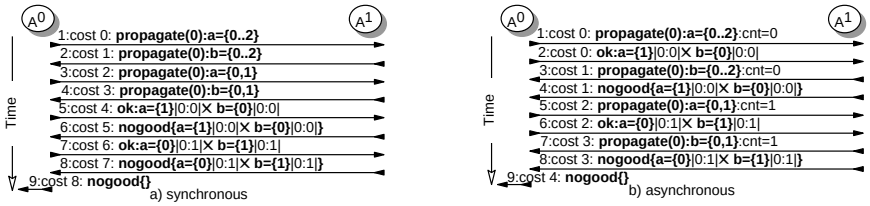

Figure 2: *Comparative traces for distributed consistency maintenance.*

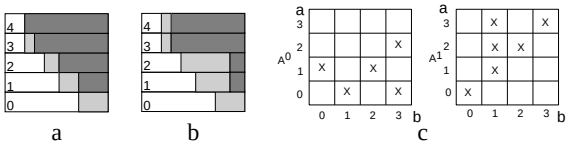

Figure 3: *a,b) Synchronism vs. asynchronism in consistency maintenance seen at A^4; c) A simple problem where running search and consistency maintenance asynchronously helps.*

than $\mathcal{A}$ for $C(P)$. If such labels were computed for P, we say that P becomes DC consistent. Generic instances of DC($\mathcal{A}$) are denoted by DC. Typically with DC, the maximum number of generated messages is a^2vd and the maximum number of sequential messages is vd (v:number of variables, d:domain size, a:number of agents).

Each agent has its own perception of the distributed search. In this perception, the search spaces associated to arcs of the search tree are defined by proposals from its predecessors. In Figure 1 is shown a simultaneous view of three agents. Only A^1 knows the fourth proposal of A^0. A^2 has not yet received the third proposal of A^1 consistent with the third proposal of A^0. However, A^3 knows that proposal of A^1. Since it has not received anything valid from A^2, A^3 assumes that A^2 agrees with A^1. In practice it may happen that an agent sees only partly the valid proposal of another agent. The depth in distributed search trees is referred to as **level**. Histories help agents to eventually get coherent views at the same level.

3 Multiply Asynchronous Search (MAS)

In this paper, instantiation asynchronism (type a) is understood as in AAS. Consistency asynchronism (type b) is obtained using DC algorithms.

Asynchronous consistency maintenance Let us consider a problem with variables $x_i, i \in [0..k]$. We assume that for this problem AAS instantiates at each node all these variables. V_i denotes the aggregate with the valid assignments at level i as known by A^4. In AAS V_k must be included in V_{k-1}. Figure 3 schematizes the way in which the size of the label of x_0 evolves when instantiations are interleaved with DC. In Figure 3a instantiations and DC alternate sequentially [Tel, 99] (protocol we refer to as synchronous MDC) while in Figure 3b they are performed asynchronously. The white box numbered by k stands for the label for x_0 obtained at level k of the search. The dark gray boxes represent the values eliminated by the valid assignment of x_0 in an aggregate. The light gray boxes represent the other values of x_0 eliminated by DC. In Figure 3a, the consistency computation at level k is started only after a new instantiation is proposed by agent A^{k-1}. Moreover, the instantiation of agent A^k can be done only after the end of consistency computation at level k.

In a distributed setting, for making a proposal (assignment), there is no need to wait in a node until all the consistencies in the previous levels on the same branch converge. We show that equivalent labels can be computed asynchronously. In fact, at a given level value elimination results from one of the following processes: instantiation, DC at this level, or inheritance from DCs at previous levels along the same branch. We propose to run all these processes concurrently (the light gray boxes in Figure 3b can overlap).

Let us now illustrate the behavior of asynchronism of type c, concerning maintaining consistencies, using a simple example with two agents handling a single constraint each. The constraints involve the same variables, a and b, and are shown in Figure 3c. Checked positions in constraint tables stand for feasible tuples. Figure 2 compares the traces of message passing for this example between synchronous and asynchronous variants of maintaining distributed consistency. The basic algorithm behaves as AAS except that, in addition to `ok` and `nogood` messages, it also allows for `propagate` messages which inform the agents about domain reductions computed by the DC processes. In Figure 2, *propagate(k):x=s* is a message informing the receiver that the label s has been computed by DC at level k for the variable x. The cost refers to the number of sequential messages required. cnt=j tells that the labels were tagged at sender with the counter j. In Figure 2a, a proposal cannot be made (message 5) before the convergence of DC which requires 4 sequential `propagate` messages. Still, the same amount of search remains to be done. The synchronous search requires eight sequential messages. Figure 2b shows that the whole search space can be exhausted with four sequential messages with asynchronism of type c.

Asynchronous reordering By properly using histories we can also implement asynchronism of reordering with polynomial space requirements. An **order**, $\langle o, h\rangle$, is given by a sequence o of distinct agents $A^0..A^k$. All the agents that do not appear in o are considered to be ordered by their name and follow A^k. A history, h, is attached to each order decided by an agent. This history is constructed exactly as for instantiations and the valid order is the newest. All the decisions taken within a certain order are tagged with that order. Figure 4 shows how for the problem in Figure 5 dynamic reordering is achieved at each node. The problem has 3 variables and is defined by 3 agents with one constraint each. The heuristic

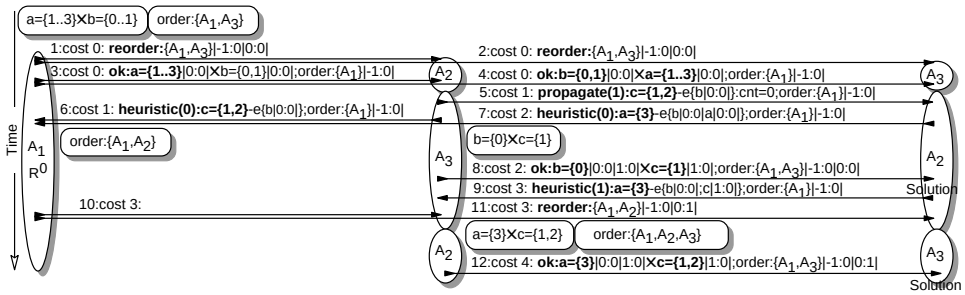

Figure 4: *A simple run of consistency with asynchronous reordering.* `addlink` *messages are redundant for this heuristic.*

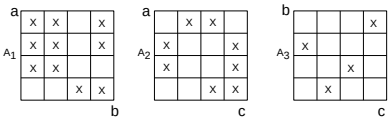

Figure 5: *A problem with 3 agents.*

in this example consists of choosing the next agent as the one with the least number of allowed tuples (least *volume*). To allow an agent A^k to compute the volumes of its successors, `heuristic` messages are used. They are sent to A^k by all agents that prove value eliminations for levels lower than or equal to $k+1$. The reordering decisions are communicated to agents $A^j, j > k$. In Figure 4 the ovals represent decisions taken by an agent. We assume that before the shown trace, some agent has decided the `order:{`A_1`}|-1:0|` establishing the identity of A^0 as A_1. Agent A_1 decides the shown instantiation and order and informs appropriately its successors by sending `reorder` and `ok` messages. Upon receipt of an `ok` message from A_1, A_3 sends a propagate message to A_2 with a reduced domain for *c*. Once A_2 receives this reduction, the domain of a becomes $\{3\}$. This information is reported to A_1 by messages 6 and 7 and allows A_1 to reconsider the order. The reordering decision sent by agent A_1 via message 2 is discarded at A_3 when the decision in message 10 arrives with a newer history.

4 The MAS protocol

We assume that a message does not necessarily need to head directly to its target agent. Its content can travel indirectly to the target via several agents. However, we require the content to arrive at destination in finite time. Parts of the content of a message may become *invalid* due to newer available information. The receiver can discard the invalid incoming information, or can reuse invalid nogoods with alternative semantics (e.g. as redundant constraints). However, when the channel (intermediary agents) holds information that it knows invalid, it is allowed to discard that information.

4.1 Maintaining Consistency Asynchronously

The `ok`, `nogood` and `addlink` messages are as in AAS. In addition, the agents may exchange information about nogoods inferred by DCs. This is done using `propagate` messages.

Definition 5 *A* **consistency nogood** *for a level k and a variable x has the form $V \rightarrow (x \in l^k_x)$ or $V \rightarrow \neg(x \in s \setminus l^k_x)$. V is an aggregate and may contain for x an assignment (x,s,h), $l^k_x \subset s$. Any assignment in V must have been proposed by predecessors of A^k. l^k_x is a label, $l^k_x \neq \emptyset$.*

The `propagate` messages take as parameters a list of consistency nogoods, the reference k of a level and a counter of the `propagate` messages sent by the sender. They are sent to all interested agents $A^i, i \geq k$. The nogoods are meant to allow the agents A^i to compute DC consistent labels on the problem obtained by integrating the most recent proposals of the agents $A^j, j<k$. A_i may receive valid consistency nogoods of level k with the variables *vars*, *vars* not in vars(A_i). A_i must send `addlink` messages to all agents $A^{k'}, k'<k$ not yet linked to A_i for all *vars*. In order to achieve consistencies asynchronously, besides the structures of AAS, implementations can maintain at any agent A^u_i, for any level k, $k \leq u$:

- The aggregate, V^i_k, of the newest valid assignments proposed by agents $A^j, j<k$, for each interesting variable.
- For each variable x, $x \in$vars(A_i), for each agent $A^t_j, t \geq k$, the last consistency nogood sent by A_j for level k, denoted $cn^k_x(i,j)$, if valid. It has the form $V^k_{j,x} \rightarrow (x \in s^k_{j,x})$.

$cn^k_x(i,i)$ is recomputed by inference (e.g. using local consistency techniques) for each variable x for the problem $P_i(k)$. Let $cn^k_x(i,.)$ be $(\cup^{t \leq k}_{t,j} V^t_{j,x}) \rightarrow (x \in \cap^{t \leq k}_{t,j} s^t_{j,x})$.

$$P_i(k) := \mathrm{CSP}(A_i) \cup (\cup_x cn^k_x(i,.)) \cup \mathrm{NV}_i(V^i_k) \cup \mathrm{CL}^i_k$$

CL^i_k is the CL of A^u_i if $k=u$ and an empty set of constraints otherwise. $cn^k_x(i,i)$ is stored and sent to other agents by `propagate` messages iff any constraint of $\mathrm{CSP}(A_i)$ was used for its logical inference from $P_i(k)$ and its label shrinks.

We now prove the correctness, completeness and termination properties of $MAS_{(+,+,+,-)}$. We only use DC techniques that terminate. Techniques to achieve such a behavior are outside the scope of this paper. (see [Zhang and Mackworth, 91; Baudot and Deville, 97]).

Lemma 1 *$MAS_{(+,+,+,-)}$ is correct, complete, terminates.*

Proof summary. Completeness: All the nogoods are generated by logical inference from existing constraints. Therefore, if a solution exists, no empty nogood can be generated.

No infinite loop: By quiescence of a group of agents we mean that none of them will receive or generate any valid

nogoods, new valid assignments, or addlink messages. We prove by induction on increasing i the **property**: *In finite time t^i either a solution or failure is detected, or all the agents $A^j, 0 \leq j \leq i$ reach quiescence in a state where they are not refused a proposal satisfying* $\text{CSP}(A^j) \cup \text{NV}_j(\text{view}(A^j))$.

Let this be true for the agents $A^j, j<i$. A^i receives the last valid `ok` message at time t^i_o. $\exists t^i_v$, $t^i_v \geq t^i_o$ such that after t^i_v, view(A^i) and all $V^u_k, k \leq i$ of any agent A_u are no longer modified. $\text{CL}^u_k, k \leq i$ cannot be invalidated after t^i_v. Since DCs were assumed to terminate, they terminate after each modification of a CL^u_k. Since the number of such modifications after t^i_v is bounded, after a finite time no consistency nogood is received any longer by A^i for levels $k \leq i$. Only one valid explicit nogood can be received for a proposal since the proposal is immediately changed on such an event. Similarly, there is a finite number of valid nogoods that can be received by A^i for any of its proposals made after t^i_v (and after t^i_o).

If one of the proposals is not refused by incoming nogoods, and since the number of such nogoods is finite, the induction step is correct. If all proposals that A^i can make after t^i_o are refused or if it cannot find any proposal, A^i has to send according to rules inherited from AAS a valid explicit nogood $\neg N$ to somebody. If N is empty, failure is detected and the induction step is proved. Otherwise $\neg N$ is sent to a predecessor $A^j, j<i$. From here, as proved for AAS, it results that either empty nogood is detected, or A^i will receive a new proposal. This contradicts the assumption that the last `ok` message was received by A^i at time t^i_o and the induction step is proved.

The property can be attributed to an empty set of agents and it is therefore proved by induction for all agents.

Correctness: All valid proposals are sent to all interested agents and stored there. At quiescence all the agents know the valid interesting assignments of all predecessors. If quiescence is reached without detecting an empty nogood, then all the agents agree with their predecessors and their intersection is nonempty and correct as proved for AAS. □

If the agents using $MAS_{(+,+,+,-)}$ maintain all the valid consistency nogoods that they have received, then DCs in $MAS_{(+,+,+,-)}$ converge and compute a local consistent global problem at each level. If not all the valid consistency nogoods that they have received are stored, but some of them are discarded after inferring the corresponding $cn^k_x(i,i)$, some valid bounds or value eliminations can be lost when a $cn^k_x(i,i)$ is invalidated. Different labels are then obtained in different agents for the same variable. These differences have as result that the DC at the given level of $MAS_{(+,+,+,-)}$ can stop before the global problem is DC consistent at that level.

Among the consistency nogoods that an agent computes itself from its constraints, $cn^k_x(i,i)$, let it store only the last valid one for each variable. Let A_i also store only the last valid consistency nogood, $cn^k_x(i,j)$, sent to it for each variable $x \in \text{CSP}(A_i)$ at each level k from any agent A_j. Then:

Proposition 1 *DC($\mathcal{A}$) labels computed at quiescence at any level with the help of* `propagate` *messages are equivalent to $\mathcal{A}$ labels when computed in a centralized manner on a processor. This is true whenever all the agents reveal consistency nogoods for all minimal labels, l^k_x, which they can compute.*

Proof. In each sent `propagate` message, the consistency nogood for each variable is the same as the one maintained by the sender. Any assignment invalid in one agent will eventually become invalid for any agent. Therefore, any such nogood is discarded at any agent, iff it is also discarded at its sender. The labels known at different agents, being computed from the same nogoods, are therefore identical and the distributed consistency will not stop at any level before the global problem is local consistent in each agent. □

Proposition 2 *The minimum space an agent needs with $MAS_{(+,+,+,-)}$ for insuring maintenance of the highest degree of consistency achievable with DC is O($a^2v(v+d)$). With bound consistency, the required space is O($(av)^2$).*

Proof. The space required for storing all valid assignments is O(dav) for values and O(av) for histories (stored separately). The agents need to maintain at most a levels, each of them dealing with v variables, for each of them having at most a last consistency nogoods. Each consistency nogood refers at most v assignments in premise and stores at most d values in label. The stack of labels requires therefore O($a^2v(v+d)$). The space required by CL and by the algorithm for solving the local problem depends on the corresponding technique (e.g. chronological backtracking requires O(v)). CL also refers at most v assignments in its premise. □

4.2 Reordering Agents Asynchronously

For allowing asynchronous reordering, each message receives as additional parameter an order. New optional messages are:

- `heuristic` messages for heuristic dependent data.
- `reorder` messages announcing a new order, $\langle o, h \rangle$.

The `heuristic` messages are intended to offer data for reordering decisions. The parameters depend on the used reordering heuristic. The `heuristic` messages can be sent by any agent and have to be sent to the agents $R^k, k \geq -1$. R^k is the reordering leader of the level k, namely the one deciding which agent will have the position $k+1$. The set $\{R^k\}$ may or may not be included in $\{A^i\}$. The agents $\{R^k\}$ need not be distinct. The identity of R^k may be modified by R^{k-1}. `heuristic` messages may only be sent by an agent to R^k in a bounded time after having received an `ok` from $A^j, j \leq k$ or a `propagate` message of level $k'', k'' \leq k+1$.

If `reorder` messages are used, the agents are required to store all the valid assignments proposed by their predecessors, for all the variables they are interested in. A `reorder` message is sent by R^k to all the agents that are defined by its parameter to have the priority $k+1, ..., n$ and to the agents $R^j, j>k$. Any `reorder` message is sent within a bounded time after a `heuristic` message is received. The prefix of k+1 agents of an order decided by R^k must be the same with the one in the newest order received by R^k. R^k appends to histories of orders he builts a pair $|k : l|$ where l is a counter of the orders it proposes. Assignments associated with an order, o_o, older than the newest received one, o_n, and built by *possibly reordered agents* or their successors, are invalidated. The first possibly reordered agent is A^{k+1}, where k is the lowest

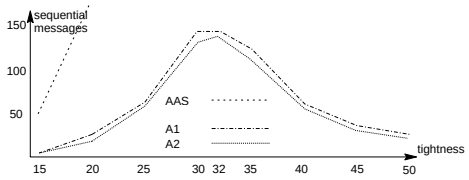

Figure 6: *Asynchronous maintaining consistencies. Results averaged over 500 problems per point.*

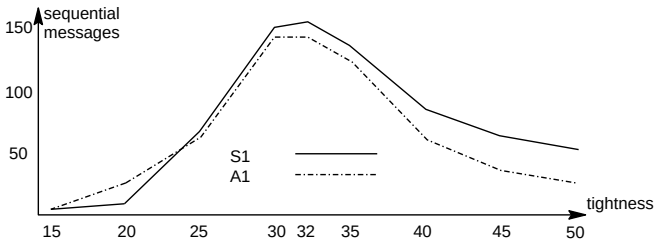

Figure 7: *Synchronous vs. Asynchronous. Results averaged over 500 problems per point.*

level whose reordering leader, R^k, has added a pair $|k : l|$ making a difference between the histories of o_o and o_n.

`addlink` messages are no longer sent only to predecessors, but to successors as well. When the heuristic in example of Figure 4 is used, `addlink` messages are redundant.

To use dynamic asynchronous reordering, besides the structures of $MAS_{(+,+,+,-)}$, each agent needs to store an order. This is the newest order it has received. The structures that have to be maintained by R^k, as well as the content of the `heuristic` messages depend on the reordering heuristic. In our example (Figure 5), we consider the case where agents announce all the interested reordering leaders of all the labels they send. In Figure 4, the reordering leader of level k, R^k corresponds to the agent A^k. R^{-1} corresponds to the agent A_1. The volume of A_i is estimated as the product of the size of domains for all variables in vars(A_i). R^i maintains for all variables and for the levels k, $k{\leq}i{+}1$ similar structures to those of $MAS_{(+,+,+,-)}$ at agent A^u_j for vars(A^u_j) and levels t, $t{\leq}u$. The space complexity remains the same as with $MAS_{(+,+,+,-)}$.

Proposition 3 *MAS is correct, complete and terminates.*

Proof summary. Completeness and **correctness** are proved using the same reasoning as for lemma 1. **No infinite loop:** We prove by induction on increasing i the same property as in the proof of lemma 1 where quiescence now also refers to `reorder` messages and to any `propagate` message of level $k{\leq}i{+}1$. Let all agents $A^j, j{<}i$ reach quiescence before time t^{i-1}. Reasoning as for lemma 1, $\exists t^i_h$ after which no propagate of level k, $k{\leq}i$ and therefore no heuristic message toward any R^u, $u{<}i$ is exchanged. Then, R^u becomes fixed, receives no message, and announces its last order before a time t^i_r. After t^i_r the identity of $A^k, k{\leq}i$ is fixed. A^i receives the last new assignment or order at time t^i_o. Reasoning as in lemma 1 for A^i and R^i, it follows that after t^i_o, within finite time, the agent A^i either finds a solution and quiescence or exhausts its possibilities and sends a valid explicit nogood to somebody. From here, the induction step is proven as in lemma 1. R^{-1} is always fixed and the property is true for the empty set. The termination results by induction on i. □

We see in Figure 4 that the solution detection method proposed for AAS can detect solutions before quiescence. However, if all the solutions are required, that solution detection algorithm may detect several times the same solutions. A quiescence detection algorithm can avoid this inconvenient.

5 Experiments

We have implemented the techniques and heuristic presented here and we have run experiments comparing four versions of MAS. The DC we used maintains bound-consistency. In each agent, computation at lower levels is given priority over computations at higher levels. We generated randomly problems with 15 variables of 8 values and graph density of 20%. Their constraints were randomly distributed in 20 subproblems for 20 agents. Figure 6 shows their behavior for variable tightness (percentage of feasible tuples in constraints), averaged over 500 problems per point. We have tested two versions of MAS, A1 and A2. A1 asynchronously maintains bound consistency at all levels. A2 is a relaxation where agents only compute consistency at levels where they receive new labels or assignments, not after reduction inheritance between levels. The versions of MAS maintaining some consistency proved to be more than 10 times better in average, even for the easy points where AAS requires less than 2000 sequential messages. A2 was slightly better than A1 in average (excepting at tightness 15%). In these experiments we have stored only the minimal number of nogoods. The nogoods are the main gain of parallelism in asynchronous distributed search. Storing additional nogoods was shown for AAS to strongly improve performance of asynchronous search. As future research topic, we foresee the study of new reordering and nogood storing heuristics. Tests show that the reordering heuristic we used is in average three times better than its reverse on this type of problems at tightness 32%.

Since DCs terminate in finite time, synchronous MDC is an instantiation of MAS, more exactly $MAS_{(-,+,-,-)}$. However, the asynchronism in $MAS_{(-,+,-,-)}$ is limited to small slices since the search itself is actually synchronous and any new decision has to be synchronized among all agents. $MAS_{(-,+,-,+)}$ does not exist since the notion of asynchronous reordering makes no sense with synchronous search. We have designed a version of $MAS_{(-,+,-,-)}$ that is based on aggregations similarly to AAS. Our version is denoted S1 in Figure 7. In S1, termination of DCs is detected using a technique requiring 1 additional sequential message per search node. Each agent A_i that is being instantiated launches DC on known(A_i) after each instantiation, respectively on its currently allowed domains before changing its instantiation. A_i starts a new DC by sending the current labels for all variables to all uninstantiated agents. All the modified labels are sent by the agents running DC to A_i. After DC ends without domain wipe-out, A_i chooses the agent with the next position in the search to be an agent with the smallest volume among all those that are not instantiated. All the labels are then sent to the next agent. The first DC is started by the agent A_0

which then chooses the agent with the first position. S1 was in average slightly worse than asynchronous versions, excepting very over-constrained problems.

6 Conclusions

We have presented MAS, a new distributed search protocol, which allows for maintaining distributed consistency with a high degree of parallelism and allows for asynchronous dynamic agent reordering. MAS covers two frustrating holes in the literature. It allows for the first really asynchronous algorithm with polynomial space complexity that maintains consistency. It also allows for the first asynchronous algorithm that is complete and enables agents to concurently and asynchronously propose agent reordering with polynomial space complexity. MAS is based on AAS. It is a generalization of the best-known distributed complete search algorithms with polynomial memory requirements. The experiments have shown that the overall performance of MAS is significantly improved compared to that of AAS. MAS has much potential in practice, it accommodates a higher number of agents and larger problems than AAS and fully exploits the aggregation capability of AAS. It has polynomial space complexity requirements for a large class of reordering heuristics.

References

[Baudot and Deville, 97] B. Baudot and Y. Deville. Analysis of distributed arc-consistency algorithms. Technical Report RR-97-07, U. Catholique Louvain, 97.

[Chandy and Lamport, 85] K.-M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS'85*, 1(3):63–75, 85.

[Kumar, 1985] D. Kumar. A class of termination detection algorithms for distributed computations. In N. Maheshwari, editor, *5th Conf. on Foundations of Software Technology and Theoretical Computer Science*, number 206 in LNCS, pages 73–100, New Delhi, 1985. Springer.

[Mattern, 87] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, (2):161–175, 87.

[Sabin and Freuder, 94] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125–129, 94.

[Silaghi *et al.*, 2000] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proc. of AAAI2000*, pages 917–922, 2000.

[Tel, 99] G. Tel. *Multiagent Systems, A Modern Approach to Distributed AI*, chapter Distributed Control Algorithms for AI, pages 539–580. MIT Press, 99.

[Yokoo *et al.*, 98] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 10(5):673–685, 98.

[Zhang and Mackworth, 91] Y. Zhang and A. K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, pages 394–397, 91.

7 Annexes

7.1 Termination detection

In synchronous MDC we need to detect the quiescence of the agents. In previous work, distributed CSP algorithms have detected quiescence using the techniques for distributed snapshot presented in [Chandy and Lamport, 85]. This algorithms have the characteristic that termination tests need to be initialized by agents suspecting quiescence. If the interest is to detect quiescence as quickly as possible, other methods are more appropriate.

We present here a termination detection method which only requests one sequential message after the quiescence of the monitored protocol. This technique is well adapted to consistency maintenance procedures for distributed CSPs. Related termination detection protocols are already known to the distributed systems community [Kumar, 1985; Mattern, 87] and their proof also applies here.

Each agent A_i maintains a counter $c_{i,j}$ for outgoing messages towards each other agent A_j and a counter $c^{j,i}$ for incoming messages from each agent A_j. When A_i becomes idle, this counters are sent at any modification to the agent T checking the termination. When T detects that $c_{i,j} = c^{i,j}$ for all i and j, then T can announce termination. Since the counters can only increase, there is no need of time stamps or FIFO channels since the highest counter value is always the newest. Several termination detection stages can succeed synchronously one after another, as happens with the successive consistency rounds in synchronous MDC. We may want to distinguish them. A simple flag with two values that switches at the start of a new round has to be attached to each message. If each agent receives a message in each round, this is sufficient to announce the agents that a new stage begins and that the local counters must be reset to 0 (1 for the incoming link announcing the new round). Otherwise the flag has to be replaced by an increasing counter of the stages. In both cases, the current value of the flag (counter) is attached to the messages requesting backtracking in order to update the knowledge of agents with lower positions.

The messages need not be sent directly to T. Each agent keeps a list Lc of counters that it has received. Lc contains only the last received pair of vectors of counters for each agent. If A_i has to send a messages to a set of agents A, it chooses an agent A_j in A. A_i attaches to the message sent to A_j the modified counters in Lc, received from other agents, as well as and its own modified counters. Then A_i clears Lc. If A was empty after A_i has received some messages, it sends its counters and its Lc to T and also clears Lc. When Lc is large enough to fill the payload of a message, the network charge is reduced if an agent can sent the modified counters to T rather than sending them further to other agents. The size of Lc when the behaviour has to change is a function of the size $|m|$ of the messages sent to agents in A and also depends on the maximum unfragmented payload (MTU) that can be sent to T. The threshold should be $|Lc| \leq (MTU - |m| mod\ MTU) mod\ MTU$.