Classifying approaches based on Parallel Proposals in Asynchronous Search*

Marius C. Silaghi
Florida Institute of Technology (FIT)
Melbourne
msilaghi@fit.edu

Boi Faltings
Swiss Federal Institute of Technology (EPFL)
Lausanne
Boi.Faltings@epfl.ch

ABSTRACT

Real distributed systems have a limited parallelism capacity, i.e. maximum number of distributed processes of a given type that can be run concurrently without a decrease in performance. Once a capacity K is found for running distributed solvers, the remaining issue (that we address here) is to design mechanisms to dynamically split distributed constraint satisfaction problems (DisCSPs) to K asynchronous solvers such that no resource is wasted.

Parallelism and distribution are two distinct concepts that are confusingly close. Parallel Search (PS) refers in this work to the distribution of the search space and Distributed Asynchronous Search (DAS) to the distribution of the constraint predicates. A certain amount of parallelism exists in any distributed asynchronous search and it increases with the degree of asynchronism. Since distributed search is the only solution for certain classes of naturally distributed problems, Hamadi in [6], and Denzinger in [4] have independently proposed a smart way of increasing parallelism in distributed search (namely by running several distributed searches in parallel on different areas of the problem's search space).

Note that here we consider each distributed search process as a black box (different algorithms could run in different such cooperating processes), but classify the techniques for distributing the problem to these distributed search processes, coordinating competition for resources and aggregating results. We analyze here[1] analytically four different alternative ways in which one can integrate the idea of Parallel Search in Distributed Asynchronous Search, the first two following closely Hamadi's versions, while the other approaches identify various alternatives. We contribute methods to maximize use of resources while assuring awareness of limitations on resources such as bandwidth and CPU. A technique for dynamic allocation of search space to available resources is presented. This technique builds on the procedure for marking concurrent proposals for conflicting resources, that we have formalized in [19, 12].

*Most of this research was performed while the first author was at EPFL and was funded by the Swiss National Science Foundation.

[1]This article is a version of [16] updated with discussions about prior and subsequent work.

1. INTRODUCTION

Each real distributed system has a parallelism capacity, i.e. maximum number of distributed processes of a given type that can be run concurrently without a decrease in performance (such as FTP downloads of a file, or DisCSP solving). Once a capacity K(T) is found for running distributed processes of type T on a distributed system, the remaining problem (that we address here) is to design mechanisms to dynamically split DisCSPs to K(T) processes, such that no resource is wasted. E.g., for FTP downloads problem that coresponds to dynamically splitting the file into chunks to be concurrently downloaded, such that each time a process succesfully downloaded its chunk it receives as new sub-task a part of the remaining chunk of a slower process. Here we address the more complex case where the task is a DisCSP problem, the chunks are partitions of the (distributed) search space of the problem, and the solvers are asynchronous.

Distributed combinatorial problems can be modeled using the general framework of Distributed Constraint Satisfaction (DisCSP). A **DisCSP** is defined in [28] as: a set of agents, $A_1, ..., A_n$, where each agent A_i controls exactly one distinct variable x_i and each agent knows all constraint predicates relevant to its variable. The case with more variables in an agent can be obtained quite easily from here, while the case of one variable in several agents can be adapted as shown in [17]. Asynchronous Backtracking (ABT) [27] is the first *complete* and *asynchronous* search algorithm for DisCSPs, allowing for completeness guarantees with polynomial space complexity [9, 28]. In [22] we present a technique for maintaining consistency in asynchronous search. [19] describes a general technique that allows the agents to asynchronously and concurrently propose changes to their order.

Parallelism and distribution are two distinct concepts that are confusingly close. Parallel Search (PS) refers in this work to the distribution of the search space and Distributed Asynchronous Search (DAS) to the distribution of the constraint predicates. This is somewhat different from the definitions in [4]. Some parallelism exists in any distributed asynchronous search and it is expected to increase with the degree of asynchronism. However, in comparison to parallel search [14], the parallel effort in distributed asynchronous

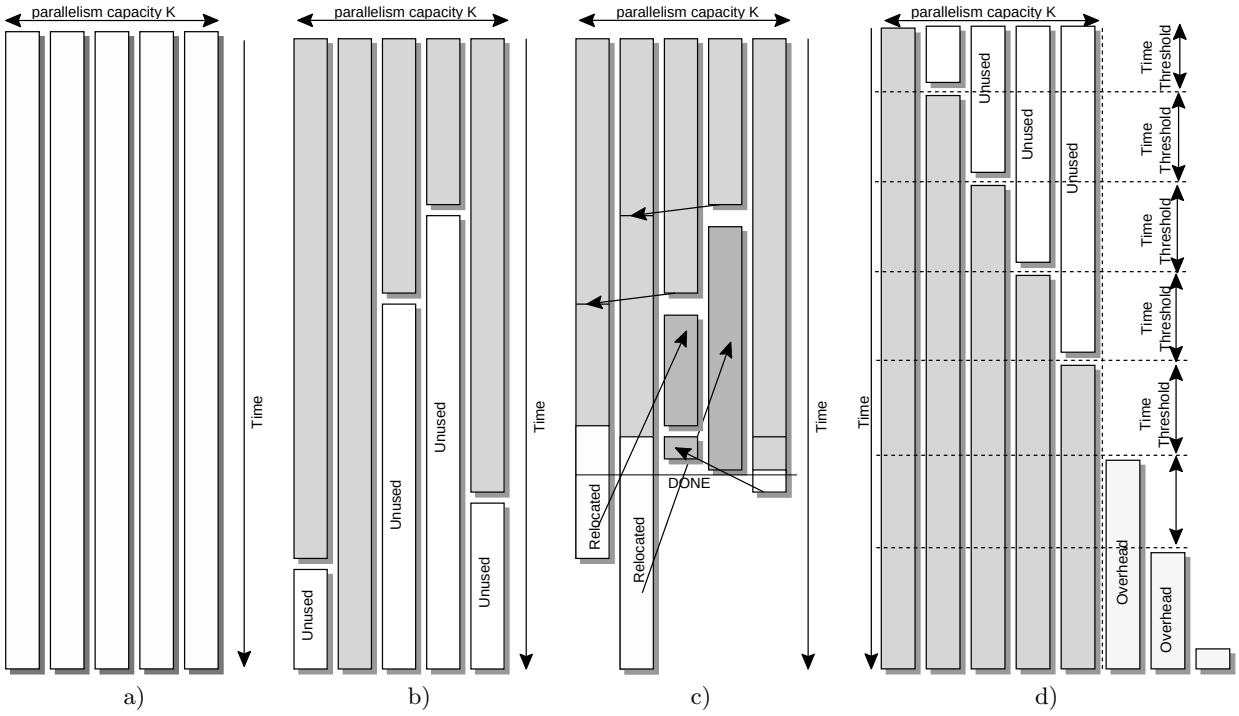


Figure 1: Parallelism capacity and approaches for handling it. Figure (a) pictures the idea that K computational slots are available in a distributed system with parallelism capacity K for a given protocol, e.g. ABT. Figure (b) shows that some resource slots are sometimes not used with the approach in [6, 7], when processes using them terminate early. Figure (c) shows the general idea we proposed in [16], namely that processes can be dynamically split to fill freed slots. Figure (d) depicts a more recent approach [31] where resource limitations are not taken into account, resulting in a epoch of resource under-usage and an epoch of context switching overhead.

search can be more redundant. Moreover, agents in DAS can have periods of inactivity which are much more important (longer and more frequent) than in parallel search. DAS is the only solution for certain classes of naturally distributed problems, such as problems with privacy [18, 5]. A smart idea independently introduced by Hamadi in [6, 7] and by Denzinger in [4] is to increase parallelism by running several distributed search processes in parallel on different areas of the search space. Hamadi's algorithm is called Interleaved Distributed Backtracking (IDIBT), while Denzinger provides the similar concept of *improvement on the competition approach*. The efficiency of this approach for DisCSPs was experimentally confirmed in [7, 15].

We assume that the distributed system (Internet) has a limited capacity, allowing for a maximum efficiency at a number of parallel DAS processes, K, dependent on the problem size and algorithm. We identify here analytically four distinct families of techniques for integrating the idea of parallel search in distributed asynchronous search, namely PAS1, PAS2, PAS3, and PAS4:

PAS1 Several DAS solving processes are run independently and concurrently on the same problem (eventually collaborating via the exchange and reuse of some nogoods). Also called "improving on the competition approach".

PAS2 The search space (Cartesian product of domains) of the initial problem is statically split in K partitions, prior to solving. A separate DAS is independently and concurrently launched for solving each partition of the search space. When a DAS finds a solution, the other DAS processes are stopped.

PAS3 The first agent has the task of dealer, i.e., to dynamically distribute the values in the domain of its variable to K running DAS processes, according to their needs (giving a new value to each DAS that exhausts its search space).

PAS4 A dynamic and fully asynchronous extension of PAS3 such that, even when the dealer agent exhausts its values, the search space continues to be split for offering new chunks to the DAS processes that exhaust their search space. This is done by dynamically and asynchronously letting dealers to designate new subdealers, to provide new chunks of search space to terminated DAS processes.

The first two approaches described here are similar to Hamadi and Denzinger's versions in [6, 7, 4], while the other two approaches seem original. Techniques for dynamic reallocation of the search space to available resources are presented to enable PAS3 and PAS4. The technique for PAS4 builds on the procedure for marking concurrent proposals for conflicting resources, that we have formalized in [19]. The more recent work in [31] independently introduces a

method related to PAS3 but which does not handle resource limitations (assuming infinite bandwidth and computation power).

The main motivation of this paper results from considering that before search it can be established that the optimal number of distributed DAS processes is K. A DAS process (set of negotiating local threads, containing one local thread in each agent) is the equivalent of a processor in parallel search approaches [14] and is called *slot* (i.e., slot of available resources). The tasks of the slots can be defined prior to search. For versions with dynamic reallocation of sub-problems to slots, we propose to handle these slots as *conflict resources* [19] and the agents make concurrently proposals about their allocation.

This kind of asynchronous search algorithms allow for parallel proposals which cannot be gathered into one Cartesian product. They are neither a generalization[2] nor an instance of AAS [17], since the different proposals can be considered separately in consistency maintenance. Our description builds on ABT since it is an algorithm easier to describe than its subsequent extensions. The techniques can nevertheless be integrated in a straightforward manner in most extensions of ABT, such as AAS and R-MAS [21]. In certain settings, especially in combination with R-MAS, parallel proposals can also offer additional opportunities for improving privacy besides improving efficiency.

Hamadi and Denzinger in [6, 4] also discuss improvements by exchanging nogoods between distributed processes running in parallel. While that is a good way to speed up different processes, it is a feature that depends on the nature of the algorithm used by the distributed processes. In this article we aim at presenting techniques for managing concurrent distributed processes in a way as independent as possible of the exact algorithms used by these processes. For this reason we avoid losing focus by delving too much in details about how to speed them up (with nogood exchanges, etc.).

1.1 Parallelism Concepts and Limitations

Our DisCSP solvers are designed for distributed systems where computation nodes belong to different users. Our concept of distributed system is defined as follows.

DEFINITION 1 (DISTRIBUTED SYSTEM). *A distributed system is composed of a set of nodes with limited computational capability and linked by communication channels with limited bandwidth.*

Processes can be "concurrently" run on mono-processor systems by time sharing and sometimes users launch such parallel threads with a misguided hope to increase performance. However, it is known that more parallel threads than the number of actual hardware processor of a system only slows down the overall performance [3]. This is due to overhead due to the context switching used to implement such parallelism. Peculiarities in operating systems schedulers sometimes slightly shift the number of useful processes, but the principle remains valid.

The same considerations generalize to the "parallel" running of distributed computational threads on a distributed system with limited bandwidth and limited computational power in nodes (e.g. as often reported for FTP transfers). Namely, beyond a certain amount of parallelism allowed by the available resources (and type of computation), the system is expected to chock due to overhead for context switching and maintenance. Each (distributed) system has a certain *parallelism capacity.*

DEFINITION 2 (PARALLELISM CAPACITY). *The parallelism capacity of a distributed system is defined by the number of parallel distributed processes (of a given type) that can run without decreasing the total efficiency due to overhead in context switching.*

Given a certain type T for distributed processes (a known algorithm, like ABT) and given a certain distributed system, it is complex to assess theoretically the expected parallelism capacity K of that system. Nevertheless, that capacity can be predicted through experimentation on the given system, as done by Hamadi in [7], and as done by Debes in [3].

In conclusion we will assume here that each given distributed system has a parallelism capacity $K(T)$ that can be found prior to running our distributed processes (see Figure 1.a). In practice, algorithms T may be designed to integrate the detection of $K(T)$ in an initial phase or dynamically, but that is outside the scope of this article.

1.2 Solvers started simultaneously

In [6, 7, 15], Hamadi uses a scheme where a number of K parallel distributed processes are simultaneously launched at the beginning for solving a given distributed CSPs. Two used approaches consist in either:

- having all processes solve the same problem (different orderings), or
- have the different processes run on different partitions of the search space of the problem.

The critique that can be brought to this approach is that some of these distributed processes may end before the others, and their resources are wasted during the remaining part of the computation (see Figure 1.b).

1.3 Reusing resources of terminated processes

Our main contribution (introduced in [16] and described here) consists of proposing approaches for:

1. dynamically splitting the not yet explored search space of the DisCSP to be solved, each time hardware resources are freed by a terminating distributed process
2. creating distributed processes for solving those new partitions, and allocating them to the freed resources.

The concept is pictured in Figure 1.c, showing that the termination of each process is used to split the task of some other still running process, using the available slot of computational resources. We will show several ways of doing this without renouncing to asynchronism.

1.4 Fallacies from disregarding resource constraints

It may be tempting to disregard the hardware limitations on computational resources and to go with the common (but erroneous) assumption that *the more parallelism the better.* The way in which such an assumption can be put in practice [31] is illustrated in Figure 1.d, namely continuously creating new processes at certain intervals of time, until the problem is solved. The expected consequences are that:

[2]If not built on AAS.

- at the beginning resources are insufficiently exploited (i.e., during the time when less than $K(T)$ processes are used).
- in the second part of the computation (if the problem is sufficiently hard to reach the moment when all resources are used), then too many processes start to be created and the system will suffer of excessive overhead from context switching and from context maintenance (just as a computer running too many processes). The space complexity is no longer be polynomial, either.

The way [31] goes around these problems is by using a simulator with infinite parallelism capacity in experimentation. Since in this work we assume real distributed systems (that do not have infinite parallelism capacity), we no longer discuss it in the rest of the article.

2. RELATED WORK

The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking (ABT)[27]. The approach in [27] considers that each agent maintains only one variable. More complex definitions were given later [29, 26]. Other definitions of DisCSPs [30, 23, 17] have considered the case where the interest on constraints is distributed among agents. [23] proposes versions that fit the structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) [17] algorithm actually extends ABT to the case where the same variable can be instantiated by several agents (e.g. at different levels of abstraction, or (dichotomous) splitting [21]) and an agent may not know all constraint predicates relevant to its variables. AAS offers the possibility to aggregate several branches of the search. Methods for ordering variables in distributed search are proposed in [25, 1, 19]. [6, 2] shows how `add-link` messages can be avoided in ABT. [8] studies the usefulness of Petri-Nets for analyzing asynchronous protocols.

The Parallel Search has been analyzed in [14, 10, 13, 6, 4]. It consists in dynamically splitting the problem and redistributing it to free processors. Important nogoods discovered by individual processors can be distributed and reused. [24] discusses how one can exchange nogoods between independent solvers running concurrently. Hamadi in IDIBT [6] and Denzinger in [4] explain a way to hybridize the two approaches by having several distributed search processes simultaneously explore different areas of the search space of a given problem. The efficiency of the approach is confirmed by experimentation described in [7].

3. ASYNCHRONOUS BACKTRACKING (ABT)

In asynchronous backtracking, the agents run concurrently and asynchronously. Each agent owns exactly one distinct variable. The variable of A_i is x_i. Each agent instantiates its variable and communicates the variable value to the relevant agents. Since here we don't assume generalized FIFO channels, in our version a **local counter**, $C_i^{x_i}$, is incremented each time a new instantiation is proposed, and its current value **tags** each generated assignment.

DEFINITION 3 (ASSIGNMENT). *An* assignment *for a variable x_i is a tuple $\langle x_i, v, c\rangle$ where v is a value from the domain of x_i and c is the* tag *value (current value of $C_i^{x_i}$).*

Given two assignments for the same variable, the one with the higher *tag* (attached value of the counter) is the **newest**. A static order is imposed on agents and we assume that A_i has the i-th position in this order. If $i>j$ then A_i has a *lower priority* than A_j and A_j has a *higher priority* then A_i.

RULE 1 (CONSTRAINT-EVALUATING-AGENT). *Each constraint C is evaluated by the lowest priority agent whose variable is involved in C.*

Each agent holds a list of **outgoing links** represented by a set of agents. Links are associated with constraints. ABT assumes that every link is directed from the value sending agent to the constraint-evaluating-agent.

DEFINITION 4 (AGENT_VIEW). *The* agent_view *of an agent, A_i, is a set containing the* newest *assignments received by A_i for distinct variables.*

Based on their constraints, the agents perform inferences concerning the assignments in their *agent_view*. By inference the agents generate new constraints called *nogoods*.

DEFINITION 5 (NOGOOD). *A* nogood *has the form $\neg N$ where N is a set of assignments for distinct variables.*

The following types of messages are exchanged in ABT: **ok?**, **nogood**, and **add-link**. An **ok?** message transports an assignment and is sent to a constraint-evaluating-agent to ask whether a chosen value is acceptable. Each **nogood** message transports a *nogood*. It is sent from the agent that infers a *nogood* $\neg N$, to the constraint-evaluating-agent for $\neg N$. An **add-link** message announces A_i that the sender A_j owns constraints involving x_i. A_i inserts A_j in its *outgoing links* and answers with an **ok?**.

The agents start by instantiating their variables concurrently and send **ok?** messages to announce their assignment to all agents with lower priority in their *outgoing links*. The agents answer to received messages according to the Algorithm 3 [19].

DEFINITION 6 (VALID ASSIGNMENT). *An assignment $\langle x, v_1, c_1\rangle$ known by an agent A_l is valid for A_l as long as no assignment $\langle x, v_2, c_2\rangle, c_2>c_1$, is received.*

A **nogood is invalid** if it contains invalid assignments.

4. PARALLEL PROPOSALS

In this section we describe the concept of *slots*. The slots are at the heart of parallel proposals in asynchronous search. The dynamic reallocation of the slots is discussed in subsequent sections.

4.1 Slots as abstract distributed processors

For simplicity, we assume that prior to search each agent allocates resources for handling K DAS processes involved in solving the current DisCSP (each DAS handled by a distinct local thread). This assumption can be slightly relaxed, as mentioned later. For an agent A_i, these resources/threads, are ordered and are identified using an additional index: $A_{i,k}, k \in [1..K]$.

DEFINITION 7. *The slot j is defined as the set of local threads $A_{i,j}, i \in [1..N]$ (Figure 2).*

when received *(***ok?**,$\langle x_j, d_j, c_{x_j}\rangle$*)* **do**
if(old c_{x_j}) return;
add(x_j,d_j,c_{x_j}) to *agent_view*;
eliminate invalidated nogoods;
check_agent_view;
end do.
when received *(***nogood**,A_j,$\neg N$*)* **do**
when any $\langle x, d, c\rangle$ in N is invalid (old c) **then**
send (**ok?**,$\langle x_i$, *current_value*, $C^i_{x_i}\rangle$) to A_j;
return;
when $\langle x_k, d_k, c_k\rangle$, where x_k is not connected, is contained in $\neg N$
send **add-link** to A_k;
add $\langle x_k, d_k, c_k\rangle$ to *agent_view*;
put $\neg N$ in *nogood-list* for x_i=d;
add other new assignments to *agent_view*;
0.1 eliminate invalidated nogoods;
old_value ← *current_value*;
check_agent_view;
when *old_value* = *current_value*
0.2 send (**ok?**,$\langle x_i$, *current_value*, $C^i_{x_i}\rangle$) to A_j;
end do.
procedure check_agent_view do
when agent_view *and* current_value *are not consistent*
if *no value in* D_i *is consistent with* agent_view **then**
backtrack;
else
select $d \in D_i$ where *agent_view* and d are consistent;
current_value ← d; $C^i_{x_i}$++;
send (**ok?**,$\langle x_i, d, C^i_{x_i}\rangle$) to lower priority agents in *outgoing links*;
end
end do.
procedure backtrack do
nogoods ← $\{V \mid V =$ inconsistent subset of *agent_view*$\}$;
when an empty set is an element of *nogoods*
broadcast to other agents that there is no solution, terminate this algorithm;
for every $V \in$ *nogoods*;
select (x_j,d_j,c) where x_j has the lowest priority in V;
send (**nogood**,A_i,V) to A_j;
eliminate invalidated explicit nogoods;
remove (x_j,d_j,c) from *agent_view*;
check_agent_view;
end do.

Algorithm 1: *Procedures of A_i for receiving messages in ABT with nogood removal.*

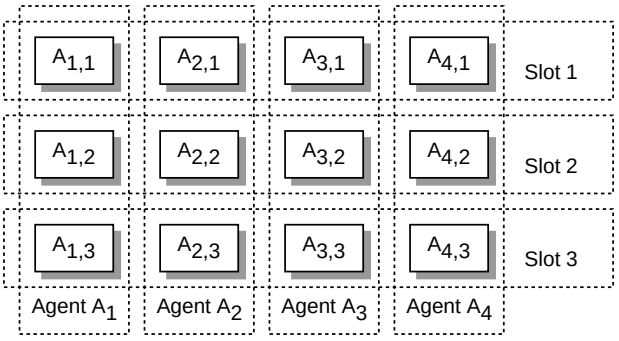


Figure 2: A slot is a set of abstract agents, one for each initial agent.

The agents own private constraints, but every process $A_{i,j}$ knows all the constraints of A_i. Therefore a slot can be used to perform a distributed computation independently from other slots. Any asynchronous protocol can be used in any slot, with the simple modification that the index of the current slot has to tag any message for identifying the target thread. Obviously, different distributed computations launched in such slots could exchange some nogoods to improve search similarly as computations on real processors do in [24]. This version will be referred to as Parallel Asynchronous Search I (PAS1). Techniques belonging to this family are described in [6, 7, 4].

When the order of the agents is different in distinct slots, the computational load of different agents can become more balanced.

Further in this paper we rather discuss techniques that distribute the search space among different slots. A family of nogood sharing techniques is naturally obtained when the nogoods involve common segments of the search tree.

4.2 Slots Statically Allocated (SSA)

The simplest way to distribute a search space among existing slots, is to statically split the domain of a variable prior to search and to distribute it among the slots. Imagine that the agents in Figure 2 work on a DisCSP P. Assume that in P, the domain of x_1, D_1 has at least K values (here $K = 3$). D_1 can then be split in K nonempty disjoint partitions, here $D_{1,1}, D_{1,2}, D_{1,3}$. Let P_i be the problem P where the domain of x_i is restricted to $D_{1,i}$. Any slot i can work independently on the problem P_i, eventually exchanging some nogoods as in PAS1. This technique can always be used for continuous domains.

When D_1 has less than K values, the splitting of the problem can continue with domains of subsequent variables. We want to balance the effort in distinct slots. The split has to ensure that the number of tuples (volume) of the search space in slots is not very different. A greedy approximate technique is to choose the allocation by a breadth first technique, calling greedy-split(1,K,P). The variables are ordered according to the descending size of their domains.

Procedure greedy-split(i,K,P)

- If $|D_i| \geq K$, split D_i in K partitions, as equally as possible. Return.
- If $|D_i| < K$, split P by splitting D_i in domains of one value. $p = K\%|D_i|$. For each obtained subproblem $P_{k,k>1}$, call greedy-split($i+1$,$K/|D_i| + (k \leq p)$,P_k).

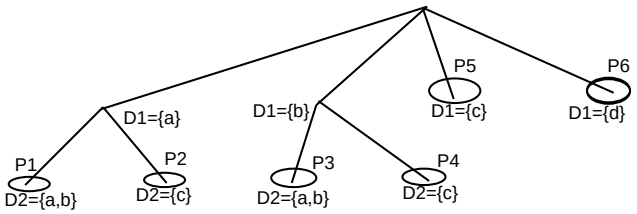


Figure 3: Weak performance of the greedy-split algorithm.

In the example of Figure 3, $K = 6$, $D_1 = \{a, b, c, d\}$ and $D_2 = \{a, b, c\}$. The problems obtained for slots are: $P_1 = \{D_1 = \{a\} \times D_2 = \{a, b\}\}$, $P_2 = \{D_1 = \{a\} \times D_2 = \{c\}\}$, $P_3 = \{D_1 = \{b\} \times D_2 = \{a, b\}\}$, $P_4 = \{D_1 = \{b\} \times D_2 = \{c\}\}$, $P_4 = \{D_1 = \{c\} \times D_2 = \{a, b, c\}\}$, $P_5 = \{D_1 = \{d\} \times D_2 = \{a, b, c\}\}$. Their size varies between 1 and 3.

In order to obtain a better equilibrium between the size of search spaces for slots, we introduce another heuristic. This is obtained by calling prime-split(K,P).

Procedure prime-split(K,P)

- Let a decomposition of K in prime numbers be $p_1 p_2, ..., p_n$. Choose (i, j) such that $|D_i|$ is divided by p_j. If this is not possible, choose $(i, j) = \underset{i,j}{\text{argmax}}[|D_i|/p_j]$. $[f]$ denotes the truncated integer of f. Among remaining competitor pairs, choose the one with highest p_j.
- If $|D_i| \geq p_j$, split D_i in p_j partitions, as equally sized as possible. For each obtained sub-problem P_k, call prime-split(K/p_j,P_k).
- If $|D_i| < p_j$, split P by splitting D_i in domains of one value. $p = p_j\%|D_i|$ For each obtained subproblem $P_{k,k>1}$, call prime-split($K/|D_i| + (k \leq p)$,P_k).

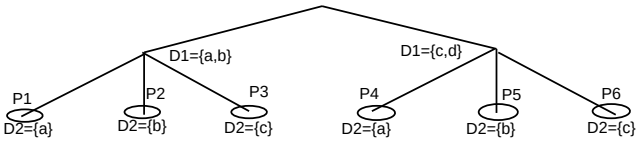


Figure 4: Results of the prime-split algorithm.

As shown in Figure 4, the algorithm prime-split can obtain better partitions. The protocol where the slots solve independently problems partitioned according to algorithms similar to those presented in this subsection are referred to as PAS2. As for PAS1, it is recommended to order the agents differently in distinct slots in order to balance their load.

4.3 Slots Statically Allocated to Agents (SSAA)

The main drawback in PAS2 is that the partitioning of the problem does not take into account the constraint predicates. One search space may be much harder than another and some slots can end their activity immediately. Now we propose to give certain agents power to split the search space among groups of slots. A hierarchy of agents can have a hierarchical control on the distribution in slots.

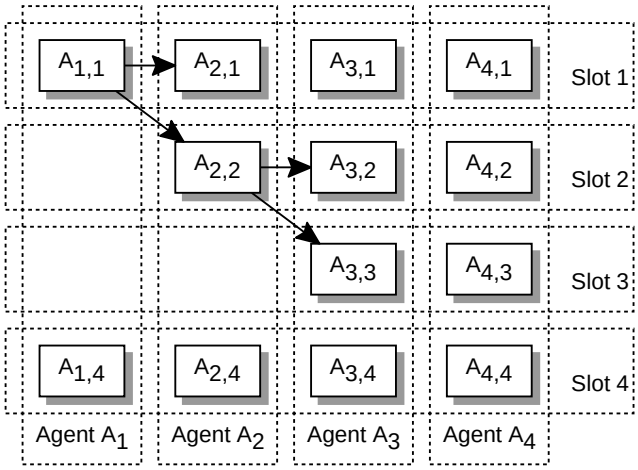


Figure 5: Agent-based static allocation.

The example in Figure 5 shows a case where the first process of agent A_1, $A_{1,1}$, takes the first position in all asynchronous search protocols for the slots 1 to 3. The second process of agent A_2, $A_{2,2}$, takes the second position in the asynchronous search protocols for the slots 2 and 3.

For this case, the initial domain D_1 of the variable x_1 of agent A_1 is statically split in two partitions: $D_{1,1}$ for the slots 1 to 3, respectively $D_{1,4}$ for the slot 4. The slot 4 behaves like in PAS2. $A_{1,1}$ starts by making two different proposals in parallel, by sending a set of **ok?** messages in the slot 1 and another set of **ok?** messages with the second instantiation of x_1 to the slots 2 and 3. $A_{2,2}$ also sends two sets of **ok?** messages, one to slot 2 and the other to slot 3. Whenever a proposal of one of these two agents is refused (e.g.. by a **nogood** message) in a slot, that agent sends a new proposal for that slot. Any **nogood** message (or **propagate** message in R-MAS) that has to be sent to A_2 by lower priority processes in slots 2 and 3, are sent to $A_{2,2}$.

Those from slots 2 and 3 towards A_1, are sent to the process $A_{1,1}$. This can be implemented very efficiently by defining the addresses of processes $A_{1,1}$, $A_{1,2}$, and $A_{1,3}$ (respectively $A_{2,2}$ and $A_{2,3}$), as synonyms.

The processes $A_{1,1}$ and $A_{2,2}$ are a bottleneck, but in general this drawback is reduced when the branching factor is low and the agents that are sources of branching have high priority. The computational load is dynamically distributed to different slots. The domain of x_2 is incrementally distributed to the slots 2 and 3 on request. Only when $A_{2,2}$ has exactly one valid proposal available, a possible value for x_2, then one of the slots 2 and 3 remains unused. The generalization of these rules for general trees of access to slots is obvious and the obtained protocol is called PAS3.

The only modification to the messages in ABT (and its extensions) is that each message has to be tagged with the name of its slot, so that the target process can be discriminated by the receiving agent. The procedures for receiving nogoods and the procedure `check_agent_view` have to be modified as shown in Algorithm 2.

ASSUMPTION 1. *We assume in the following that all the threads of an agent can share data.*

5. SIGNATURES

Now we recall [19] a *marking technique* that allows for the definition of a total order among the proposals made concurrently and asynchronously by a set of ordered agents on a shared resource (e.g. a label-AAS, an order-ABTR, an allocation of a slot).

DEFINITION 8. *A* **proposal source** *for a resource $\mathcal{R}$ is an an agent or a thread that can make specific proposals concerning the allocation (or valuation) of $\mathcal{R}$.*

We consider that an order $\prec$ is defined on *proposal sources*. The *proposal sources* with lower position according to $\prec$ have a higher priority. The *proposal source* for $\mathcal{R}$ with position k is noted $P_k^{\mathcal{R}}$, $k \geq x_0^{\mathcal{R}}$. $x_0^{\mathcal{R}}$ is the first position.

DEFINITION 9. *A* **conflict resource** *is a resource for which several agents can make proposals in a concurrent and asynchronous manner.*

Each *proposal source* $P_i^{\mathcal{R}}$ maintains a counter $\mathbf{C_i^{\mathcal{R}}}$ for the *conflict resource* $\mathcal{R}$. The markers involved in our *marking technique for ordered proposal sources* are called **signatures**[3]. They are similar in structure (but not in usage) with the vector clocks proposed by Mattern in [11]. A vector clock tagging a message is a list with the value of the counter of each agent, as known by a sender of a message at the moment when the message was sent (i.e., $|c_1, ..., c_n|$ where c_k is the value of $C_i^{\mathcal{R}}$ known by the sender). We historically used for the vector clocks in our signatures a compact representation where zero valued counters are not listed and instead each counter value is associated with its position.

DEFINITION 10. *A* **signature** *is a chain h of pairs, $|a{:}b|$, that can be associated to a proposal for $\mathcal{R}$. A pair $p{=}|a{:}b|$ in h signals that a proposal for $\mathcal{R}$ was made by $P_a^{\mathcal{R}}$ when its $C_a^{\mathcal{R}}$ had the value b, and it knew the prefix of p in h.*

[3]In early publications they were called *histories*.

A signature where the clock values are described in the format proposed by Mattern (i.e., a simple list) is denoted in our recent work *signatures vector clock*.

An order $\propto$ (read "precedes") is defined on pairs such that $|i_1{:}l_1| \propto |i_2{:}l_2|$ if either $i_1 < i_2$, or $i_1 = i_2$ and $l_1 > l_2$.

DEFINITION 11. *A signature h_1* **is newer than** *a signature h_2 if a lexicographic comparison on them, using the order $\propto$ on pairs, decides that h_1 precedes h_2.*

$P_k^{\mathcal{R}}$ builds a signature for a new proposal on $\mathcal{R}$ by prefixing to the pair $|k{:}\text{value}(C_k^{\mathcal{R}})|$, the newest signature that it knows for a proposal on $\mathcal{R}$ made by any $P_a^{\mathcal{R}}$, $a{<}k$. The $C_a^{\mathcal{R}}$ in $P_a^{\mathcal{R}}$ is reset each time an incoming message announces a proposal with a newer signature, made by higher priority *proposal sources* on $\mathcal{R}$. $C_a^{\mathcal{R}}$ is incremented each time $P_a^{\mathcal{R}}$ makes a proposal for $\mathcal{R}$.

DEFINITION 12. *A signature h_1 built by $P_i^{\mathcal{R}}$ for a proposal is* **valid** *for an agent A if no other signature h_2 (eventually known only as prefix of a signature h_2') is known by A such that h_2 is newer than h_1 and was generated by $P_j^{\mathcal{R}}$, $j \leq i$.*

For example, in Figure 6 the agent P_3^x may get messages concerning the same resource x from P_1^x and P_2^x. In Figure 6a, if the agent P_3^x has already received m_1, it will always discard m_3 since the *proposal source* index has priority. However, in the case of Figure 6b the message m_1 is the newest only if $k_{1f} < k_{1l}$ and is valid only if $k_{1f} \leq k_{1l}$. In each message, the length of the signature for a resource is upper bounded by the number of *proposal sources* for the *conflict resource*.

6. DYNAMIC ALLOCATION IN PARALLEL ASYNCHRONOUS SEARCH

Here we show how the marking technique presented in the previous section can be used by agents to make parallel proposals while dynamically allocating slots. In [19], an order on agents is modeled as a resource while each proposal defines guidelines for reordering and a recommended order. The guidelines from high priority agents have priority, and are followed by the recommended orders of lower priority agents that respect the valid guidelines.

To asynchronously and dynamically allocate slots to parallel proposals, we consider each slots as a *conflict resource*. The proposal sources for each slot consists of an ordered set of $N-1$ threads. The mapping of these threads to processes of initial agents can be modified identically as for reordering. Each proposal consists in:

- *a working slot*, and
- *a set of free slots.*

The free slots are the ones that can theoretically receive the control of this slot, but the working slot is the recommended one.

The next convention helps to aggregate messages containing proposals on the allocations of several slots into messages called **slots**.

CONVENTION 1. *By convention, the proposal sources for a slot, s, are delegated to the threads in the current working slot for s, and are ordered according to the current order of the processes in the asynchronous protocol.*

when received (**nogood**,$A_{j,slot}$,$\neg N$) **do**
when any $\langle x, d, c\rangle$ in N is invalid (old c) **then**
send (**ok?**,$\langle x_i$, *current_value*[*slot*], $C^i_{x_i}\rangle$) to $A_{j,slot}$;
return;
when $\langle x_k, d_k, c_k\rangle$, where x_k is not connected, is contained in $\neg N$
send **add-link** to A_k;
add $\langle x_k, d_k, c_k\rangle$ to *agent_view*;
put $\neg N$ in *nogood-list* for x_i=d;
add other new assignments to *agent_view*;
2.1 eliminate invalidated nogoods;
old_value ← *current_value*[slot];
check_agent_view;
when *old_value* = *current_value*[slot] (= d)
2.2 send (**ok?**,$\langle x_i$, *current_value*[*slot*], $C^i_{x_i}\rangle$) to $A_{j,slot}$;
end do.
procedure check_agent_view do
when agent_view *and any* current_value *are not consistent*
if *no value in* $D_{i,k}$ *is consistent with* agent_view **then**
if *no* current_value *is consistent with* agent_view **then**
backtrack
else
set inconsistent *current_value*s to -1
end
else
select $d \subseteq D_{i,k}$ where *agent_view* and d are consistent;
inconsistent *current_value*s ← elements of d;
for every *modified slot, s,* **do**
$C^i_{x_i}$(s)++;
send (**ok?**,$\langle x_i, d, C^i_{x_i}\rangle$) to lower priority processes of slot s
for agents in *outgoing links*
end do
end
end do.

Algorithm 2: *Procedures of $A_{i,k}$ for receiving nogoods in PAS3.*

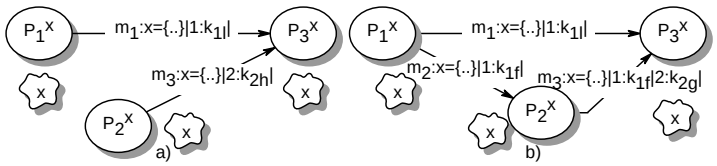


Figure 6: Simple scenarios with messages for proposals on a resource, x.

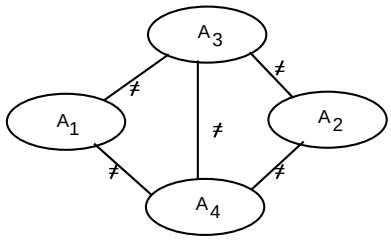
When a process is proposal source for several slots and the proposals for those slots are identical, those proposals need to be sent only once. The payload of the **slots** messages, consisting in a proposal on allocating available slots, tags each **ok?** and **nogood** message. Therefore no **slots** message is needed to announce allocation proposals to agents found in outgoing links, since those agents learn the newest proposals from the tags of the received **ok?** message.

The PAS4 family of algorithms contains protocols where:

- Proposals are made according to the previous conventions.
- When a reallocation is proposed, all the proposal sources for the corresponding slots, placed on higher positions, are announced. On the receipt of newer allocations, data tagged with invalidated signatures of slot allocation is removed.
- Each message is tagged with the newest allocation for the receiving slot, as known at sender. For **propagate** messages in DMAC and R-MAS, this corresponds to the tag of their level.
- A proposal source only makes a finite number of proposals on slot allocations after a proposal of variable instantiation was refused for the delegated process.
- In ABTR, the order of successor agents can only be modified when a reallocation of their slot is made. (In order to reorder the agents, a new proposal for reallocation has to be defined and it has to tag the proposal on order)

The pair added in the signature of a proposal on slots reallocations has the form$|(i : cS) : c|$, where cS is the slot of the process delegated as the proposal source which builds this pair. i is its position. c is the value of the counter of proposals for this proposal source.

PROPOSITION 1. *When the protocols used in slots are complete extensions of polynomial space ABT (e.g. AAS,*



1: $A_{1,1}$ —— **ok?**$\langle x_1, a, 1, (1, \{1, 2\})|(1 : 1) : 0|\rangle$ ——→ $A_{3,1}, A_{4,1}$

// message 1 proposes x_1=a to slots 1 and 2,
// allowing future agents to split it further

2: $A_{1,1}$ —— **ok?**$\langle x_1, b, 1, (3, \{3\})|(1 : 1) : 0|\rangle$ ——→ $A_{3,3}, A_{4,3}$

// message 2 proposes to use slot 3 for the assignment x_1=b

3: $A_{2,1}$ —— **ok?**$\langle x_2, a, 1, (1, \{1, 2, 3\})|(2 : 1) : 0|\rangle$ ——→ $A_{3,1}, A_{4,1}$

// message 3 is discarded upon reception because its signature
// is weaker than signatures of messages 1 and 2

4: $A_{1,1}$ —— **slots**$\langle (1, \{1, 2\})|(1 : 1) : 0|\rangle$ ——→ $A_{2,1}$

5: $A_{1,1}$ —— **slots**$\langle (3, \{3\})|(1 : 1) : 0|\rangle$ ——→ $A_{2,3}$

// messages 4 and 5 tell unlinked lower priority agent A_2 about the slot allocation

6: $A_{2,1}$ —— **ok?**$\langle x_2, a, 1, (1, \{1\})|(1 : 1) : 0|(2 : 1) : 0|\rangle$ ——→ $A_{3,1}, A_{4,1}$

// message 6 proposes to allocate slot 1 for the assignment x_2=a

7: $A_{2,1}$ —— **ok?**$\langle x_2, b, 1, (2, \{2\})|(1 : 1) : 0|(2 : 1) : 0|\rangle$ ——→ $A_{3,2}, A_{4,2}$

// message 7 proposes to allocate slot 2 for the assignment x_2=b

8: $A_{3,1}$ —— **ok?**$\langle x_3, b, 1, (1, \{1\})|(1 : 1) : 0|(2 : 1) : 0|\rangle$ ——→ $A_{4,1}$

// message 8 proposes the assignment x_3=b in slot 1 which leads
// to A_4 selecting $x_4 = c$, and terminating with a solution

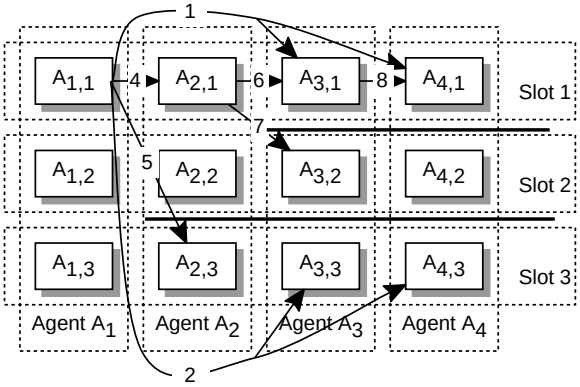


Figure 7: Example of a trace with PAS4.

R-MAS), PAS algorithms are complete, correct and terminate, and require only polynomial space.

Proof. The proof is obvious for PAS1-PAS3 and results from the corresponding properties of the used asynchronous algorithms, and on the completeness of the problem partitioning.

In PAS4, the working slots elected by the first agents cannot be continuously disturbed and interrupted until a solution is found or the proposal launched on them is refused. Whenever a reallocation is proposed, all involved processes are announced and they will update their proposals. When any complete extension of ABT (AAS, R-MAS) is used in slots, the termination of PAS4 results by induction. Namely, once a process and its predecessors are no longer refused, the reasoning applies to the process on the next position in the working slots. The completeness is a consequence of using only logic inference. The use of signatures for slot reallocations leads to coherent views in processes for each given allocation. The soundness is ensured by the fact that coherent views lead to generation of **nogood** messages at any contradiction. Actually, the complete extensions of ABT ensure that processing of such valid **nogood** messages leads to soundness when they are tagged with valid signatures.

The required space complexity is K times the highest space complexity required by the asynchronous protocols used in slots. □

In Figure 7 is given a simple example of a trace of PAS4 with ABT in 3 slots, for the shown coloring problem. When a message is sent to several agents, a single message is shown and the list of target agents is shown on the right-hand side. The proposal on the slots relevant for each message is shown in parentheses after the other parameters. It is followed by the signature for that proposal. All the processes start having by default available as free slots all the slots, and having the slot 1 as current working slot. The proposal sources in agent A_1 propose to split the free slots in two. These proposals are attached to the **ok?** messages that have to be sent to processes of agents A_3, A_4. They are sent by

slots messages to the agent A_2 since no other message is scheduled toward A_2. Meanwhile, the agent A_2 also made proposals in message 3, but their tag is recognized as invalid by the receiving processes which know tags from messages 1 and 2. The **slots** message 4 is delivered by the process $A_{2,1}$ to its both proposal sources for slots 1 and 2.

The example is shown only up to a point where a solution is found, but most slots are still working. The signature of the slot proposals in nogoods are trimmed as for the signature of proposals on orders in ABTR-wc1 [19]. The target slot for a nogood is computed as the slot in the last pair in the trimmed signature of the nogood. If nogoods would have to be sent from the process of A_3 in slot 2 to A_1, they would be sent to the process in slot 1 of A_1, as read in the pair $|(1:1):0|$ found in valid signatures.

6.1 Nogood reuse across reallocation (PAS4r)

Similarly with the nogood reuse across reordering [20], nogoods can be saved when new proposals for reallocations are received. For example, in Figure 7, the inferences resulting from assignments in message 3 can be temporarily stored as redundant constraints by all the working processes of agents A_3, and A_4. When new assignments arrive in messages 6 and 7, if nothing changes, the corresponding receiving processes only need to update the tags and recover the corresponding nogoods (e.g. this happens in the slot 1). Otherwise, the stored invalidated nogoods can be discarded (slot 2). The corresponding protocol is called PAS4r.

6.2 Dynamic reconfiguration in PAS4r

During search, a proposal of A_i might be refused and A_i may want to offer to other existing working slots the set of freed slots. A_i can do it by simply broadcasting the new proposal on the modified slots using **slots** messages with tags with incremented counters.

If the current proposal source wants to make a slot available to predecessor proposal sources, dedicated heuristic messages can be defined easily without modifying the properties of PAS4r. Let us look again at the example in Figure 7. If a nogood would be received by the process $A_{1,3}$, this could either propose c in slot 3, or allocate the slot 3 to the proposal in $A_{1,1}$. In the last case, the current computation in the slots already allocated to the current instantiations proposed by $A_{1,1}$ will not be disturbed by reallocation.

The proposal sources in the process in slot 1 for agent A_2, can detect after receiving nogoods for two proposals that A_2 can make only one more proposal and that they have two available slots in the current allocation. In this situation heuristic messages can be sent to proposal sources in A_1 such that the slot 2 can be reallocated (e.g. to the proposal in message 2).

7. CONCLUSIONS

We are motivated by the fact that each real distributed system has a parallelism capacity, i.e. maximum number of distributed processes of a given type that can be run concurrently without a decrease in performance. Once a capacity K(T) is found for running distributed asynchronous solvers of some type T on a distributed system, the remaining problem (that we addressed here) is to design mechanisms to dynamically split the search space of a DisCSP to K(T) concurrent asynchronous solvers such that no resource is wasted.

We analyzed existing techniques for exploiting parallelism in asynchronous search (extensions of ABT), and identified 4 classes of algorithms among those designed for realistic distributed systems (namely systems with finite resources). We have proposed techniques enabling two such classes of techniques, continuing ideas introduced in IDIBT [6] and in [4]. This family of techniques is called Parallel Asynchronous Search. Note that here we discuss techniques for distributing the problem search space to a set of concurrent distributed search processes (treated as black boxes), coordinating competition for resources and aggregating results, rather than caring how this processes work.

Our new technique are the only ones to allow for dynamic allocation of the problem search space to the optimal number of asynchronous search processes, such that none of them wastes time before the search ends.

8. ACKNOWLEDGEMENTS

This research was supported by the Swiss National Science Foundation under project number 21-52462.97.

9. REFERENCES

[1] A. Armstrong and E. F. Durfee. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *Proceedings of 15th IJCAI*, 1997.

[2] C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In *Proc. IJCAI DCR Workshop*, pages 9–16, 2001.

[3] Eric Debes. *Exploitation of Parallelism in General Purpose Processor Based Systems for Multimedia Applications*. PhD thesis, EPFL, Lausanne, Switzerland, December 2000.

[4] J. Denzinger. Distributed knowledge based search. IJCAI tutorial notes (MA2), 2001.

[5] E.C. Freuder, M. Minca, and R.J. Wallace. Privacy/efficiency tradeoffs in distributed meeting scheduling by constraint-based agents. In *Proc. IJCAI DCR*, pages 63–72, 2001.

[6] Youssef Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués*. PhD thesis, Université Montpellier II, Juillet 1999.

[7] Youssef Hamadi. Interleaved backtracking in distributed constraint networks. In *ICTAI*, pages 33–41, 2001.

[8] M. Hannebauer. On proving properties of concurrent algorithms for distributed CSPs. In *Proceedings of the CP Workshop on Distributed Constraint Satisfaction*, 2000.

[9] W. Havens. Nogood caching for multiagent backtrack search. In *AAAI Constraints and Agents Workshop*, 1997.

[10] Q.Y. Luo, P.G. Hendry, and J.T. Buchanan. Comparison of different approaches for solving distributed constraint satisfaction problems. Technical Report No. RR-93-74, University of Strathclyde, 1993.

[11] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[12] Viet Nguyen, Djamila Sam-Haroud, and Boi Faltings. Dynamic distributed backjumping. In *DCR Workshop*, pages 51–65, 2004.

[13] N. Prcovic. Un algorithm distribué pour la résolution des problèmes de contraintes en domaines finis. Technical Report CERAMICS 95.44, CERAMICS, Novembre 1995.

[14] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE*, 4(4), Apr 1993.

[15] G. Ringwelski and Y. Hamadi. Multi-directional distributed search with aggregation. In *IJCAI-DCR*, 2005.

[16] M.-C. Silaghi and B. V. Faltings. Parallel proposals in asynchronous search. Technical Report 01/#371, EPFL, August 2001.

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

[18] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with private constraints. In *Proc. of AA2000*, pages 177–178, Barcelona, June 2000.

[19] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. ABT with asynchronous reordering. In *IAT*, 2001.

[20] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizing ABT and AWC into a polynomial space, complete protocol with reordering. Technical Report #01/364, EPFL, May 2001.

[21] M.-C. Silaghi, D. Sam-Haroud, and B. Faltings. Polynomial space and complete multiply asynchronous search with abstractions. In *IJCAI-01 DCR Workshop*, pages 17–32, Seattle, August 2001.

[22] M.-C. Silaghi, Djamila Sam-Haroud, and Boi Faltings. Maintaining hierarchical distributed consistency. In *Workshop on Distributed CSPs*, Singapore, September 2000. 6th International Conference on CP 2000.

[23] G. Solotorevsky, E. Gudes, and A. Meisels. Algorithms for solving distributed constraint satisfaction problems (DCSPs). In *AIPS96*, 1996.

[24] Cyril Terrioux. Cooperative search and nogood recording. In *Proc. of IJCAI-01*, pages 260–265, 2001.

[25] M. Yokoo. Asynchronous weak-commitment search for solving large-scale distributed constraint satisfaction problems. In *1st ICMAS*, pages 467–318, 1995.

[26] M. Yokoo. *Distributed Constraint Satisfaction*. Springer, 2001.

[27] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *ICDCS*, pages 614–621, June 1992.

[28] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE TKDE*, 10(5):673–685, 1998.

[29] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of 3rd ICMAS'98*, pages 372–379, 1998.

[30] 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, 1991.

[31] R. Zivan and A. Meisels. Concurrent backtrack search on discsps. In *FLAIRS*, 2004.