Parallel Proposals in Asynchronous Search

Technical Report No. TR-01/371

Marius-Călin Silaghi and Boi Faltings
Laboratoire d'Intelligence Artificielle
Département d'Informatique
EPFL, Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne

E-mail: \{silaghi, faltings\}@lia.dii.epfl.ch
Fax: ++41-21-693.52.25

August, 2001

Parallelism and distribution are two distinct concepts that are confusingly close. Parallel Search refers in this work to the distribution of the search space and Distributed Asynchronous Search 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. However, in comparison to Parallel Search [10], the parallel effort in Distributed Asynchronous Search can be more redundant. Moreover, agents in Asynchronous Search can have periods of inactivity which are less frequent in Parallel Search. Since Distributed Search is the only solution for certain classes of naturally distributed problems, we show here how one can integrate the idea of Parallel Search in Distributed Asynchronous Search. A technique for dynamic reallocation of search space is then presented. This technique builds on the procedure for marking concurrent proposals for conflicting resources, that we have formalized in [11].
1 Introduction

Distributed combinatorial problems can be modeled using the general framework of Distributed Constraint Satisfaction (DisCSP). A DisCSP is defined in [21] 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 [12]. Asynchronous Backtracking (ABT) [20] is the first complete and asynchronous search algorithm for DisCSPs. A simple modification was mentioned in [6, 21] to allow for versions with polynomial space complexity. In [13] we present a technique for maintaining consistency in asynchronous search. [11] describes a general technique that allows the agents to asynchronously and concurrently propose changes to their order. Using a special type of markers, the completeness of the search is ensured with polynomial space complexity.

Parallelism and distribution are two distinct concepts that are confusingly close. Parallel Search refers in this work to the distribution of the search space and Distributed Asynchronous Search to the distribution of the constraint predicates. This is somewhat different from the definitions in [3]. A certain amount of parallelism exists in any Distributed Asynchronous Search and it increases with the degree of asynchronism. However, in comparison to Parallel Search [10], the parallel effort in Distributed Asynchronous Search can be more redundant. Moreover, agents in Asynchronous Search can have periods of inactivity which are much less important (less long and frequent) in Parallel Search. Since Distributed Search is the only solution for certain classes of naturally distributed problems, we show here how one can integrate the idea of Parallel Search in Distributed Asynchronous Search. A technique for dynamic reallocation of the search space is then presented. This technique builds on the procedure for marking concurrent proposals for conflicting resources, that we have formalized in [11].

The main idea of this paper results from considering that before search, each agent agrees on using a number of $K$ distributed processes (slots). A set of processes, containing one process from each agent, is the equivalent of a processor in Parallel Search approaches [10]. The tasks of the slots can be defined previous to search. For dynamic reallocation of the processes, these slots are considered here as conflict resource [11] and the agents make proposals about their allocation.

This is the first asynchronous search algorithm that allows for parallel proposals which cannot be gathered into one Cartesian product. It is also the first protocol where non-redundant parallelism is explicitly generated in Asynchronous Search. This is neither a generalization\footnote{If not built on AAS.} nor an instance of AAS, since the different proposals can be considered separately in consistency maintenance. Here we build 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 [14]. In certain settings, especially in combination with R-MAS, parallel proposals can also offer additional opportunities for improving privacy besides improving efficiency.

2 Related Work

The first complete asynchronous search algorithm for DisCSPs is the Asynchronous Backtracking (ABT) [20]. The approach in [20] considers that each agent maintains only one variable. More complex definitions were given later [22, 18]. Other definitions of DisCSPs [23, 16, 12] have considered the case where the interest on constraints is distributed among agents. [16] proposes versions that fit the structure of a real problem (the nurse transportation problem). The Asynchronous Aggregation Search (AAS) [12] 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 [14]) and an agent may not know all constraint predicates relevant to its variables. AAS offers the possibility to aggregate several branches of the search. An aggregation technique for DisCSPs is then presented in [8] and allows for simple understanding of privacy/efficiency mechanisms, also discussed in [4]. The strong impact of the ordering of the variables on distributed search is so far addressed in [19, 16, 1, 11]. [2] shows how add-1 link messages can be avoided in ABT. [5] studies the usefulness of Petri-Nets for analyzing asynchronous protocols.

The Parallel Search has been analyzed in [10, 7, 9, 3]. It consists in dynamically splitting the problem and redistributing it to free processors. Important nogoods discovered by individual processors can be distributed and reused. [17] discusses how one can exchange nogoods between independent solvers running concurrently.

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 1 (Assignment) An assignment for a variable $x_i$ is a tuple $(x_i, v, c)$ where $v$ is a value from the domain of $x_i$ and $c$ is the tag value (current value of $C_i^{x_i}$).

Among 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 $A_i$. 

2
when received \((\text{ok?},(x_j,d_j,c_{x_j}))\) do
  if old \(c_{x_j}\) return;
  add \((x_j,d_j,c_{x_j})\) to agent\_view;
  eliminate invalidated nogoods;
  \(\text{check\_agent\_view}\);
end do.
when received \((\text{nogood},A_j,-\bar{N})\) do
  when any \((x,d,c)\) in \(N\) is invalid (old \(c\)) then
    send \((\text{ok?},(x_i,\text{current\_value},C^i_{x_i}))\) to \(A_j\);
    return;
  when \((x_k,d_k,c_k)\), where \(x_k\) is not connected, is contained in \(-\bar{N}\)
    send \text{add\_link} to \(A_k\);
    add \((x_k,d_k,c_k)\) to agent\_view;
    put \(-\bar{N}\) in nogood-list for \(x_i=d\);
    add other new assignments to agent\_view;
  1.1 eliminate invalidated nogoods;
    old\_value \leftarrow \text{current\_value};
    \(\text{check\_agent\_view}\);
  when old\_value = \text{current\_value}
    \(\text{check\_agent\_view}\);
  1.2 send \((\text{ok?},(x_i,\text{current\_value},C^i_{x_i}))\) 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 \leftarrow d; \(C^i_{x_i}++\);
      send \((\text{ok?},(x_i,d,C^i_{x_i}))\) to lower priority agents in outgoing links;
    end
end do.
procedure backtrack do
  nogoods \leftarrow \{V \mid V = \text{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 \(V \in \text{nogoods}\); select \((x_j,d_j,c)\) where \(x_j\) has the lowest priority in \(V\); send \((\text{nogood},A_i,V)\) to \(A_j\);
    eliminate invalidated explicit nogoods;
    remove \((x_j,d_j,c)\) from agent\_view;
  \(\text{check\_agent\_view}\);
end do.
Algorithm 1: Procedures of \(A_i\) for receiving messages in ABT with nogood removal.
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 2 (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 3 (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 1 [11].

Definition 4 (Valid assignment) An assignment $\langle x, v_1, c_1 \rangle$ known by an agent $A_i$ is valid for $A_i$ 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. The next property is a consequence of the fact that ABT is an instance of AAS [12].

Property 1 If only one nogood is stored for a value then ABT has polynomial space complexity in each agent, $O(n^d)$, while maintaining its completeness and termination properties. $d$ is the domain size and $n$ is the number of agents.

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.
Figure 1: A slot is a set of abstract agents, one for each initial agent.

4.1 Slots as abstract distributed processors

For simplicity, we assume that prior to search each agent allocates $K$ processes for solving the current DisCSP. This assumption can be slightly relaxed, as mentioned later. For an agent $A_i$, these processes, are ordered and are identified using an additional index: $A_{i,k}, k \in \{1, K\}$.

Definition 5 The slot $j$ is defined as the set of processes $A_{i,j}, i \in 1, N$ (Figure 1).

The agents own private predicates, but every process $A_{i,j}$ knows all the predicates 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 process. Obviously, different distributed computations launched in such slots could exchange some nogoods to improve search similarly as computations on real processors do in [17]. This version will be referred to as Parallel Asynchronous Search I (PASI).

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 1 work on a DisCSP $P$. Assume that in $P$, the domain of $x_1$, $D_1$ has at least $K$ values (here $K = 3$). Let $P_1$ be the problem $P$ where the domain of $x_i$ is restricted to $D_{1,i}$. $D_1$ can then be split in $K$ nonempty disjoint partitions, here $D_{1,1}, D_{1,2}, D_{1,3}$. 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 equilibrate 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 |Di| ≥ K, split Di in K partitions, as equally as possible. Return.
- If |Di| < K, split P by splitting Di in domains of one value. \( p = K \% |D_i| \).
  For each obtained subproblem \( P_{h,k>1} \), call greedy-split(\( i+1, K/|D_i| + (k \leq p), P_k \)).

In the example of Figure 2, \( K = 6 \), \( D_1 = \{a, b, c, d\} \) and \( D_2 = \{a, b, c\} \). The problems obtained for slots are: \( P_1 = \{D_1 = \{a\} \times D_2 = \{a, b\}\} \), \( P_2 = \{D_1 = \{a\} \times D_2 = \{c\}\} \), \( P_3 = \{D_1 = \{b\} \times D_2 = \{a, b\}\} \), \( P_4 = \{D_1 = \{b\} \times D_2 = \{c\}\} \), \( P_5 = \{D_1 = \{b\} \times D_2 = \{a, b\}\} \), \( P_6 = \{D_1 = \{d\} \times D_2 = \{a, b, c\}\} \). 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 \( p_1p_2, ..., p_n \). Choose (i,j) such that \( |D_i| \) is divided by \( p_j \). If this is not possible, choose \((i,j) = \text{argmax}[|D_i|/p_j]. [f] \) 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 subproblem \( P_k \), 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 \( P_{h,k>1} \), call prime-split(\( K/|D_i| + (k \leq p), P_k \)).

As shown in Figure 3, 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 very 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.

The example in Figure 4 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. Instead, the computational load can be dynamically adjusted 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 \texttt{check\_agent\_view} have to be modified as shown in Algorithm 2.

\textbf{Assumption 1} We assume in the following that all the processes of an agent can share data.

Given the previous assumption, $A_{1,1}$ needs to send \texttt{ok?} messages only to the processes in the slot 2, instead of sending them to the slots 2 and 3. This reduces the number of exchanged messages, but special care is required in implementing the agents such that they do not become bottlenecks.

\section{Histories}

Now we recall \cite{11} 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).

\textbf{Definition 6} A proposal source for a resource $R$ is an entity (e.g. an abstract agent) that can make specific proposals concerning the allocation (or valuation) of $R$.

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 $R$ with position $k$ is noted $P^R_k$, $k \geq x^R_0$. $x^R_0$ is the first position.

\textbf{Definition 7} A conflict resource is a resource for which several agents can make proposals in a concurrent and asynchronous manner.

Each proposal source $P^R_k$ maintains a counter $C^R_k$ for the conflict resource $R$. The markers involved in our marking technique for ordered proposal sources are called \textbf{histories}.

8
when received \((\text{noobd}, A_{j, \text{slot}}, -N)\) do
  when any \((x, d, c)\) in \(N\) is invalid (old \(c\)) then
    send \((\text{ok?}, (x_i, \text{current value(slot)}, C^i_{x_i}))\) to \(A_{\text{j,slot}}\);
    return;
  when \((x_k, d_k, c_k)\), where \(x_k\) is not connected, is contained in \(-N\n    send \text{add-link} \) to \(A_k\);
    add \((x_k, d_k, c_k)\) to \text{agent view};
  put \(-N\) in \text{noobd-list} for \(x_i=d\);
  add other new assignments to \text{agent view};

  \begin{enumerate}
    \item \text{eliminate invalidated noobds};
      \text{old value} \leftarrow \text{current value(slot)};
      \text{check_agent_view};
    \item \text{when old value} = \text{current value(slot)} \(= d\)
      \text{send} \((\text{ok?}, (x_i, \text{current value(slot)}, C^i_{x_i}))\) to \(A_{\text{j,slot}}\);
  \end{enumerate}
end do.

\begin{algorithm*}
\begin{algorithmic}
\Procedure{check_agent_view}{}
  \If {no value in \(D_{i,k}\) is consistent with \text{agent view}}
    \If {no current value is consistent with \text{agent view}}
      \text{backtrack}
    \Else
      \set inconsistent \text{current values} to -1
    \EndIf
  \Else
    \select \(d \subseteq D_{i,k}\) where \text{agent view} and \(d\) are consistent;
    \set inconsistent \text{current values} \leftarrow \text{elements of } \(d\);
    \For {every modified slot, } \(s\), \Do
      \(C^i_{x_i}(s)++;
      \text{send} \((\text{ok?}, (x_i, d, C^i_{x_i}))\) to lower priority processes of slot \(s\)
      \For {agents in outgoing links}
        \EndFor
      \EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm*}

Algorithm 2: Procedures of \(A_{i,k}\) for receiving noobds in PARSER.

\textbf{Definition 8} \textbf{A history} is a chain \(h\) of pairs, \(|ab|\), that can be associated to a proposal for \(\mathcal{R}\). A pair \(p=|ab|\) in \(h\) signals that a proposal for \(\mathcal{R}\) was made by \(P^R_a\) when its \(C^R_a\) had the value \(b\), and it knew the prefix of \(p\) in \(h\).

An order \(\prec\) (read “precedes”) is defined on pairs such that \(|i_1;i_1| \prec |i_2;i_2|\) if either \(i_1 < i_2\), or \(i_1 = i_2\) and \(l_1 > l_2\).

\textbf{Definition 9} A history \(h_1\) \textbf{is newer than} a history \(h_2\) if a lexicographic comparison on them, using the order \(\prec\) on pairs, decides that \(h_1\) precedes \(h_2\).

\(P^R_h\) builds a history for a new proposal on \(\mathcal{R}\) by prefixing to the pair \(|k|value(C^R_h)|\), the newest history that it knows for a proposal on \(\mathcal{R}\) made by
any $P^R_a$, $a<k$. The $C^R_a$ in $P^R_a$ is reset each time an incoming message announces a proposal with a newer history, made by higher priority proposal sources on $\mathcal{R}$. $C^R_a$ is incremented each time $P^R_a$ makes a proposal for $\mathcal{R}$.

Figure 5: Simple scenarios with messages for proposals on a resource, $x$.

**Definition 10** A history $h_i$ built by $P^R_i$ for a proposal is valid for an agent $A$ if no other history $h_j$ (eventually known only as prefix of a history $h'_j$) is known by $A$ such that $h_j$ is newer than $h_i$ and was generated by $P^R_j$, $j \leq i$.

For example, in Figure 5 the agent $P^R_3$ may get messages concerning the same resource $x$ from $P^R_1$ and $P^R_2$. In Figure 5a, if the agent $P^R_3$ has already received $m_1$, it will always discard $m_3$ since the proposal source index has priority. However, in the case of Figure 5b 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 history 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 [11], 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 $N-1$ abstract agents. The delegations of these abstract agents 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, when a proposal source for a slot $s$, proposes $s$ as working slot, the proposed set of free slots is empty. The receiving slots interpret the proposals in this way, even if the set of free slots that they receive is not empty.*

**Convention 2** *By convention, the proposal sources for a slot, $s$, are delegated to the processes in the current working slot for $s$, and are ordered according to the current order of the processes in the asynchronous protocol.*

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.

By PAS4 we refer the protocol 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 histories 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 history of a proposal on slots reallocations has the form $(i : c) : c$, where $c$ 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. Termination detection can be run independently in distinct slots.

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

11
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 histories 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 histories.

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

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{graph_coloring.png}
\caption{A graph coloring problem with 3 colors, \{a,b,c\}.}
\end{figure}

In Figure 7 is given a simple example of a trace of PAS4 with ABT in 3 slots, for the coloring problem shown in Figure 6. 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 history 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 history of the slot proposals in nogoods are trimmed as for the history of proposals on orders in ABTR [11]. The target slot for a
nogood is computed as the slot in the last pair in the trimmed history 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 histories.

6.1 Nogood reuse across reallocation (PAS5)

Similarly with the nogood reuse across reordering [15], 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
consuming receiving processes only need to update the tags and recover the
consuming nogoods (e.g. this happens in the slot 1). Otherwise, the stored
invalidated nogoods can be discarded (slot 2). The corresponding protocol is
called PAS5.

6.2 Dynamic reconfiguration in PAS5

During search, a proposal of $A_i$ might be refused and $A_i$ may want to offer
to other working processes 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 PAS5. 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
propose 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).

Figure 7: Example of a trace with PAS4.

| 1: $A_{1,1}$ | ok?($x_1, a, 1, (1, 2))|(1 : 1 : 0) | $A_{3,1}, A_{4,1}$ |
| 2: $A_{1,3}$ | ok?($x_1, b, 1, (3, 3))|(1 : 1 : 0) | $A_{3,1}, A_{4,1}$ |
| 3: $A_{2,1}$ | ok?($x_2, a, 1, (1, 2, 3))|(2 : 1 : 0) | $A_{3,1}, A_{4,1}$ |
| 4: $A_{1,1}$ | slots($1, (1, 2))|(1 : 1 : 0) | $A_{2,1}$ |
| 5: $A_{1,3}$ | slots($3, (3))|(1 : 1 : 0) | $A_{2,3}$ |
| 6: $A_{2,1}$ | ok?($x_2, a, 1, (1))|(1 : 1 : 0) | $A_{3,1}, A_{4,1}$ |
| 7: $A_{2,2}$ | ok?($x_2, b, 1, (2, 2))|(1 : 1 : 0) | $A_{3,2}, A_{4,2}$ |
| 8: $A_{3,1}$ | ok?($x_3, b, 1, (1))|(1 : 1 : 0) | $A_{4,1}$ |
7 Conclusions

We have presented a family of techniques for introducing parallelism in Asynchronous Search (extensions of ABT). This family is called Parallel Asynchronous Search, and 5 members of its members are detailed. The techniques PAS1 and PAS2 are expected to work well especially for problems where the agents use distinct network connections and processors for their processes. PAS3 to PAS5 are reasonable mostly in combination with techniques such as R-MAS which allow for a better balance in computation and communication load, and give agents additional flexibility.

8 Acknowledgements

The authors are supported by the Swiss National Science Foundation under project number 21-52462.97.

References


