Synchronizing for Performance - DCOP algorithms *

Or Peri and Amnon Meisels
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, 84105, Israel
{*perio,am*}*@cs.bgu.ac.il*

Keywords: Distributed search: DCOP.

Abstract: The last decade has given rise to a large variety of search algorithms for distributed constraints optimization problems (DCOPs). All of these distributed algorithms operate among agents in an asynchronous environment. The present paper presents a categorization of DCOP algorithms into several classes of synchronization. Algorithms of different classes of synchronization are shown to behave differently with respect to idle time of agents and to irrelevant computation. To enable the investigation of the relation between the classes of synchronization of algorithms and their run-time performance, one can control the asynchronous behavior of the multi-agent system by changing the amount of message delays. A preliminary probabilistic model for computing the expected performance of DCOP algorithms of different synchronization classes is presented. These expectations are realized in experiments on delayed message asynchronous systems. It turns out that the performance of algorithms of a weaker synchronization class deteriorates much more when the system becomes asynchronous than the performance of more synchronized DCOP algorithms. The notable exception is that concurrent algorithms, that run multiple search processes, are much more robust to message delays than all other DCOP algorithms.

1 INTRODUCTION

Distributed constraint optimization problems (*DCOP*s) are composed of agents, each holding its local constraints network, that are connected by constraints among variables of different agents. Agents assign values to variables, attempting to generate a globally optimal assignment, minimizing the sum of costs of constraints between agents (cf. (Meisels, 2008; Modi et al., 2005; Yeoh et al., 2010)). To achieve this goal, agents check the value assignments to their variables for costs and exchange messages with other agents, to check the overall costs of their proposed assignments against the value of constraints with variables owned by different agents (Modi et al., 2005; Yeoh et al., 2010; Gershman et al., 2009).

A search procedure for an optimal assignment of all agents in a Distributed Constraints Optimization Problem (*DCOP*) is a distributed algorithm. All agents cooperate in search for a globally optimal solution. The search procedure involves assignments of all agents to all their variables and the exchange of messages among agents, to check the global cost of assignments arising from constraints between agents. DCOP algorithms differ greatly in their design. The present study defines classes of synchronization for DCOP algorithms. Each class is based on specific conditions that are proven to hold for any agent that performs an algorithm of the correlative synchronization class.

Computations of agents during search are aimed at one of two complementary goals. Either prove that some partial assignment can be consistently extended by additional assignments of the computing agent to its variables, or prove that the partial assignment cannot be extended in this way (and therefore cannot be extended to a complete solution). Either of these two options advances the complete search process. The former extends the search process and delivers the new partial assignment to an unassigned agent. The latter prunes the subtree under the current view, triggers a backtrack and as a result the search process moves to a different part of the search tree.

Any subtree that does not contain a solution can in general be proven to be so in more than one way and by more than one agent. Asynchronous algorithms enable the proofs to be given concurrently by multiple agents (Bessiere et al., 2005; Modi et al., 2005). A central idea of the present study is that one can think

*Supported by the Lynn and William Frankel center for Computer Sciences and the Paul Ivanier Center for Robotics and Production Management.

of additional such proofs if they are computed non-concurrently as *irrelevant computations*. The emphasis on the non-concurrency of the different computations of the same proof is important, because concurrent computations do not necessarily lengthen the non-concurrent run-time of the algorithm (Meisels, 2008; Zivan and Meisels, 2006c). Completely synchronous DCOP algorithms can be shown to guarantee that such irrelevant computations will never happen during search. However, the price paid by typical synchronous DCOP algorithms is the fact that all computations happen sequentially and therefore all computations contribute to the non-concurrent run time of these algorithms.

The categorization of algorithms is based on the amount of enforced coordination needed to make a decision of a value assignment. We show that, typically, synchronization results with *idle-time*, as agents are compelled to wait for their peers' actions before taking action. On the other hand, attempts to utilize the system's idle-time by allowing agents to assign their variables without coordinating (e.g. asynchronously) is shown to create *irrelevant computation*, some at the expense of idle-time as intended, but some can potentially lengthen the total run-time. The focal point of the present paper is the investigation of these two attributes of DCOP algorithms: the amount of idle time vs. irrelevant computation.

The asynchronous nature of a system that can be parametrized by the amount of delay of messages directly affects the amount of irrelevant computation. The amount of irrelevant computation may deteriorate the run-time performance of an asynchronous algorithm. This impact is analyzed for a synchronous and a less synchronous protocol under different system conditions. In contrast, a *Conc* mechanism which incorporates several synchronous search-processes is shown to shorten the idle-time imposed by a synchronous algorithm without inflicting irrelevant computation and is thus both more efficient and more resistant to system conditions.

Distributed constraints optimization problems (DCOPs) are presented in Section 2. Section 3 presents classes of synchronization for DCOP search algorithms and classifies several well known algorithms into these classes. All categorized DCOP algorithms are treated in three groups - DCOP algorithms that use a DFS tree (Modi et al., 2005; Chechetka and Sycara, 2006; Yeoh et al., 2010); algorithms that do not use a DFS tree (Gershman et al., 2009); and algorithms that incorporate multiple search processes (Meisels, 2008; Netzer et al., 2010). The computation of expected idle time for different algorithms, as a function of their synchronization class is presented in Section 4, along with a general probabilistic model of asynchronous idle time. The computation of the expected idle time is also presented in section 4. An extensive experimental evaluation, which compares the behavior of several DCOP algorithms in the presence of message delays is in Section 5. The results of the evaluation supports our thesis and theoretical analysis that claims that a strongly coordinated algorithm is more efficient than a weakly coordinated one. Interestingly, this holds in a completely asynchronous system. It also demonstrates the impact of the degree of the asynchrony of the system over the synchronous, asynchronous and 'Conc' mechanisms. Section 6 discusses the experimental behavior of the algorithms and a possible explanation for the outstanding performance and robustness of concurrent DCOP algorithms.

2 DISTRIBUTED CONSTRAINT OPTIMIZATION

A DCOP is composed of a set of n agents $A_0, A_2, ..., A_{n-1}$, along with a set $\{X_{i_j} : 0 \leq i \leq n-1, 1 \leq j \leq k_i\}$ of constrained variables. Each agent A_i is responsible for assigning the variables $X_{i_1}, X_{i_2}, ..., X_{i_{k_i}}$. A *constraint* or *relation* C is a function (typically termed the "cost" or the "gain") from a subset of the Cartesian product of the domains of the constrained variables that assumes real values. For a set of constrained variables $X_1, X_2, ..., X_l$, with corresponding domains of values for each variable $D_1, D_2, ..., D_l$, the constraint is a function $C : D_1 \times D_2 \times \cdots \times D_l \longrightarrow \mathbf{R}_+$ to the non-negative reals. A *binary constraint* C_{ij} between any two variables X_j and X_i is a function from a subset of the Cartesian product of their domains; $C_{ij} : D_i \times D_j \longrightarrow \mathbf{R}_+$. We denote that two variables X_i, X_j share a constraint by $C_{ij} \in C$.

In a distributed constraint optimization problem, constrained variables may belong to different agents (Modi et al., 2005; Gershman et al., 2009). Each agent has a set of constrained variables, i.e., a *local constraint network*.

An assignment is a pair $< \texttt{var}, \texttt{val} >$, where `var` is a variable of some agent and `val` is a value from `var`'s domain that is assigned to it. A *compound label* (or a partial solution) is a set of assignments of values to a set of variables. A *feasible solution* is a compound label assigning values to all variables. An *optimal solution P* of a DCOP is a feasible solution, that has a minimal global cost (or a maximal global gain) for all the constraints. Agents check assignments of values against non-local constraints by communicating with other agents through sending and receiving

messages. Agents exchange messages with their constrained agents whose assignments have a cost that arises from constraints (Bessiere et al., 2005). Agents connected by constraints are therefore called *neighbors*. The ordering of agents is termed *priority*, so that agents that are later in the order are "lower priority agents" (Yokoo, 2000; Bessiere et al., 2005).

The following simplifying assumptions are routinely made in studies of DCOPs and are assumed to hold in the present study (Yokoo, 2000; Bessiere et al., 2005):
(1) Each agent holds exactly one variable.
(2) Messages arrive at their destination in finite time.
(3) Messages sent by agent A_i to agent A_j are received by A_j in the order they were sent.

3 FORMAL ELEMENTS OF SYNCHRONIZATION

The intuitive notion of classes of synchronization for different DCOP algorithms is based on certain statements of partial consistency that the states of the algorithms guarantee. The following definitions of consistency states of DCOP algorithms relate to the idea of a search tree. A search tree is defined for a distributed search process as follows.

Definition 1. *The* Search Tree*(ST) of a DCOP algorithm is defined by a linear ordering of the agents $A_0,\ldots,A_{n-1}$ of the DCOP. This order defines a tree where each node at some level i, ($0 \leq i < n$) represents a partial assignment of $A_0,\ldots,A_{i-1}$ and each edge diverging from such a node adds a possible assignment of a value v_i of A_i ($v_i \in D_i$, where D_i is the domain of values of agent A_i) to the partial assignment.*

A search process over a ST is the following

Definition 2. *A* Search Process*(SP) of a DCOP is a distributed computation (composed of local computations by agents and of message passing among them) that scans some subtree of the problem's search-tree in search of the lowest cost global assignment.*

Typically, search processes attempt to find a partial assignment such that the sum of costs of the constraints among assigned agents in the partial assignment is less than or equal to some Upper-Bound: $\sum_{c_{ij}\in C} C(i,j) \leq UB$.

Now one can define the concept of a *backward-consistent* view of an agent

Definition 3. *Agents have views of assignments of agents that are ordered before them on the search tree. The view of agent A_i, of a DCOP search-process, is said to be* Backwards-Consistent*(BC) if for every agent A_j that is before A_i in the search tree (i.e., $j < i$ and $C_{ij} \in C$) that appears in A_i's view it is the case that $\sum_{l=0}^{j}\sum_{k<l,c_{kl}\in C} C(k,l) \leq UB$.*

In other words, a view of agent A_i that is backwards-consistent contains a partial-assignment of agents ordered before A_i whose total cost is not higher than the Upper-bound. In an asynchronous system, UB in definition 3 relates to the upper bound known to agents that are incorporated in A_i's view, as they send their assignment states to A_i. In an analogous way to definition 3 one can define a forward consistent view, where the partial assignment is consistent forward with unassigned agents.

Definition 4. *The view of agent A_i, of a DCOP search-process, is said to be* Forward-Consistent*(FC) if for every agent A_j that is before A_i in the search tree the following inequality holds: $\sum_{l=0}^{j}\sum_{k<l,c_{kl}\in C} C(k,l) + \sum_{l\geq i} LB_l \leq UB$. Where LB_l is the lower bound of the cost of agent A_l for its constraints with the partial assignment in agent A_i's view.*

A *forward-consistent*(FC) view is therefore also BC. It guarantees that there exists an assignment, for every agent ordered after agent A_i, that is consistent with this view's partial assignment. Note that assignments of lower-priority agents are not included in the view, and therefore may not be consistent with each other. An agent that has a consistent view of the search-process is guaranteed that the computation triggered by its decisions will not be proven irrelevant to finding a solution by a higher-priority agent.

Definition 5. *Let agent A_i be of lower-priority than some agent A_j. Let A_i perform computation based on an agent-view (or a cpa) containing A_j. If there exists an ordering of messages such that by the time the outcome of A_i's computation reaches agent A_j it already holds a proof that the current view cannot be part of an optimal solution, the computation performed by agent A_i is defined to be* irrelevant.

A synchronous algorithm is one whereagents always hold consistent views of the search-process, and thus do not perform irrelevant computations. Some asynchronous algorithms, on the other hand, do not keep any of the above properties and may potentially perform irrelevant computations. However, asynchronous algorithms are designed to perform computations concurrently, attempting to lower the overall non-concurrent runtime. Asynchronous DCOP algorithms can be shown to fall into one of the above classes and, as a result, incorporate different potential amounts of irrelevant computations.

Let us go over several clear example algorithms and their synchronization classes.

Synchronous Branch & Bound : In SyncBB (Hirayama and Yokoo, 1997) each agent in a serial order (e.g., in its turn) receives a CPA message, which is a partial assignment that is a *backwards-consistent* (BC) view of all (assigned) higher-priority agents. The receiving agent tries to add its own assignment to the CPA, verifies its backwards-consistency, and passes it on. This naive algorithm doesn't check forward-consistency and as a result is outperformed easily by algorithms that do. It is clear that the mechanism of SyncBB is completely sequential. No computation can be proven irrelevant to the search-process. Every consistency check either proves consistency of all higher priority agents whose assignments are part of the view that is the basis for the computation, or removes a non empty part of the search process by proving an inconsistency.

Synchronous Forward Bounding: SFB takes a more eager approach, using a Forward-Bound mechanism. Upon receiving a CPA message, the receiving agent is guaranteed to have a forward-consistent (FC) view of the search-process. Its procedure is to select an assignment (expand the *cpa*), verify consistency backwards, and send FB requests forward to receive FB estimates (a lower bound LB_l for each $l > i$) that will be summed up to a consistency-forward check against the current value of the Upper-Bound. Following this complete verification of forward-consistency, a CPA message is sent to the next agent. If no consistent value is found, the *cpa* has been proven (forward or backward) inconsistent and therefore cannot be part of any optimal solution. Clearly, the SFB algorithm assures *forward-consistent* views, and therefore it belongs to a higher synchronization class than SyncBB.

Asynchronous Forward-Bounding: AFB (Gershman et al., 2009) uses the same FB mechanism as SFB with one difference. Similarly to SFB, once an agent is done checking backwards-consistency on its newly assigned value, it sends FB requests to lower-priority agents requesting to verify forward-consistency. However, it does not wait for approval by all requested agents and the next agent is expected to immediately further expand the CPA and send FB-requests of its own. This implies that an agent A_i receiving a CPA message is only guaranteed *backward-consistency* of the view on the received CPA. As a result, the receiving agent's computations may be proven irrelevant if a higher-priority agent later revokes the received partial assignment. This can happen if the higher priority agent (say, agent A_j, whose assignment appears on the CPA) has received some FB estimate proving the view on the CPA to be forward-inconsistent. Such a message will cause a change of assignment of A_j into an alternative backward-consistent one (and the sending of a later-time-stamped CPA message). This means that for this specific case the agent that received the CPA might have performed an irrelevant computation.

The run-time performance cost of an irrelevant computation can vary, depending on the actual distributed computation taking place. It may be the case that an agent performs irrelevant computation instead of being idle, so that the non-concurrent run-time of the algorithm is not necessarily lengthened. However, two types of damage to the performance of the algorithm can be incurred. First, the agent's unavailability to accept new messages while performing an irrelevant computation may delay the non-concurrent run-time of the algorithm. Second, the communication load of the system increases by the addition of (irrelevant) messages. A higher communication load may cause deterioration in the delivery time of messages, as well as create additional irrelevant computation.

3.1 Algorithms that Use a DFS Tree

Some DCOP algorithms organize the agents into a pseudo-tree (also known as a DFS tree) in order to gain run-time performance by the resulting enhanced concurrency (Modi et al., 2005; Yeoh et al., 2010; Chechetka and Sycara, 2006). In a DFS tree, if two agents share a constraint they are in an ancestor-descendent relation. Any two agents which are not in such a relation need not check consistency directly with one another. This property enables the solving of unrelated sub-problems concurrently. It also means that the relation *higher-priority than* (lower-priority than) is undefined for any two nodes that are not related as descendant-ancestor. As a result, the use of a backward-consistent view by an agent does not suffice for the same type of guarantee against irrelevant computation as in the algorithms described above. At any instant of the search process traversing the subtree rooted at agent A_i its parent may be informed by another child of it (a sibling of agent A_i) of a cost that breaks its known Upper-Bound. The result of such an event will be a re-assignment of the parent which implies that the computation performed by all agents within the subtree rooted at agent A_i is irrelevant.

Non-commitment Branch and Bound: In NCBB (Chechetka and Sycara, 2006) when agent A_i receives a search message from its parent, it has a view of the SP that is backwards-consistent. A_i then invokes multiple *subtreeSearches* (which are similar to requesting an FB-estimate) one per each child and value it has. The invocations of subtree searches are done iteratively, keeping every child occupied with some calculation over a value still considered

relevant. In other words, for each non-busy child A_j of Agent A_i it selects a value (that was neither proven obsolete nor explored by that child in the current search) and initializes a $subtreeSearch_j$ to get an FB-estimate of the subtree rooted at A_j. For each returned lower bound (LB) estimate from a child's subtree, A_i replies with a search message - pending that the sum of this value's back-costs and the cost estimates received so far are still within bound, i.e. *partially consistent-forward*. This utilizes concurrency among children, as each subtree attempts to keep its children busy at all times, not waiting for forward-bounds from all children before invoking a search, or before exploring other values on one subtree prior to them being fully explored on another.

However, while one child returns an estimate or a cost accumulating to a subtotal cost that is higher than the known UB, other children may continue to explore this (now obsolete) value. This has the potential effect of delaying the search of other (relevant) values.

Branch-and-Bound ADOPT: In BnB-ADOPT (Yeoh et al., 2010; W. Yeoh and Koenig, 2009; Gutierrez et al., 2011) all agents start computation on their current views (initially empty ones) and do not wait for any message in order to start assigning values. Each agent sends a COST message to its parent, and VALUE messages to its children and pseudo-children (forward-edges). Every VALUE message from an ancestor may cause a change of assignment and further VALUE messages to descendants, and every COST message from a child triggers a COST message to the recipient's parent. The result is that VALUE messages are *not backwards-consistent*; Consider an agent A_i that receives a VALUE message from one of its pseudo-parents agent A_j which just switched value. Agent A_i may now hold a view with assignments of agents on the path from A_j to A_i, that were not checked for consistency against the VALUE reported by A_j. Furthermore, A_i will now choose a value and send its children and pseudo-children a VALUE message carrying its view and its new assignment, none of which was checked for consistency with all former assignments.

Asynchronous Distributed OPTimization: Alike the former, ADOPT (Modi et al., 2005) agents start computation on their current views (initially empty ones) and do not wait for a sign in order to start assigning values. Agents send COST messages to their parents, and VALUE messages to their children and pseudo-children (forward-edges). Here too VALUE messages are *not backwards-consistent*. However, unlike BnB-ADOPT, this mechanism takes a unique best-first approach, i.e. a change of assignment of some agent A_i is made whenever the currently explored value no longer has the best lower bound. This means that an agent may change its assignment from some value d_1 to a different one d_2 that currently has a better lower bound, after the subtree of d_1 was only partially explored. As a result, it may return to d_1 (and will need to reconstruct subtree cost) once d_2 is explored some more. This may happen many times during search, and effectively means that no assignment can be ruled out until the very end of the search-process, when the partial assignment becomes a full assignment and is still considered as the "best" one.

The last step we take in differentiating levels of synchronization requires the description of a weaker synchronization element for the case of BnB-ADOPT and we term it *depth-first search*. Searching in a depth-first manner enables the synchronization of pruning actions. When an agent receives a message that reports the assignment of a higher-priority agent that is different than its previous assignment, the receiving agent can be certain that the entire subspace of solutions that is rooted at the sender's former assignment is pruned from the search, never to be explored again. In other words, the receiving agent is informed that any computation it performs that is based on the former assignment(s) of the sender is necessarily irrelevant. The property that we term depth-first search differentiates the synchronization class of ADOPT from that of BnB-ADOPT, which has this property (as do protocols of a higher synchronization class). The bottom line of the above categorization of synchronization classes for DCOP algorithms is that ADOPT is the least synchronized of all DCOP algorithms. It does not belong even to the weakest class (depth-first search) and agents computing against a received partial assignment cannot be guaranteed that a replaced assignment of some ancestor will not be visited again and may still be relevant.

Figure 1 is a scheme of DCOP algorithms by their level of coordination. A Grade value of 1 describes an algorithm that considers (views) a backward (respectively, forward) expression of the compound assignment while expanding it. A grade 2 is given if the assignment is BC (respectively, FC). Note that SBB and SFB define the exact cases of BC and FC.

3.2 A Different Type of Concurrency

A different concept of concurrency is introduced into DCOP search by the **Concurrent Forward-Bounding** algorithm (Netzer et al., 2010). In ConcFB, idle time is reduced by splitting the search-tree into several disjoint subtrees, each explored by a different search-process(SP). The concept of a search process was first introduced for distributed constraints

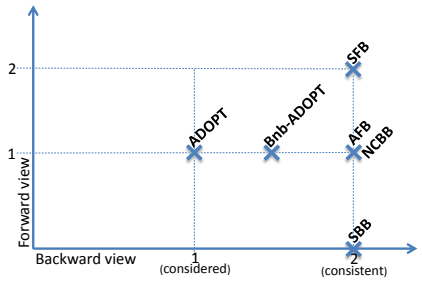

Figure 1: Synchronization levels.

satisfaction in (Zivan and Meisels, 2006a) and involves all of the agents. In order to improve efficiency, each SP has a different agent hierarchy, randomly or dynamically generated. When some SP finds a complete solution (better than the last one), all agents are informed of the new upper-bound (and the best assignment known). This causes all other SPs to consider a tighter UB, and prune faster partial-assignments of higher cost. Each SP performs an SFB search of its assigned subtree. As a result, each SP guarantees that all CPA messages are *forward-consistent* when sent and received. Consequently, each computation against a received CPA message is relevant. The only point in time where a computation can be irrelevant occurs when a new-solution message, carrying a lower UB than the one known, is on its way at the same time when an agent is busy expanding some partial assignment, that is carrying a cost higher than the new UB to arrive.

Any two concurrent SPs are independent of one another, as they search disjoint parts of the global search space. This is unlike an asynchronous mechanism of a single search process, where multiple simultaneous computations may relate to the same search space (*cpa*). Going back to the definitions of relevance the proof that a specific 'Conc' search process cannot lead to an optimal solution does not imply anything about the optimality of any other SP.

4 IDLE TIME

Let us take a closer look at the behavior of the three forward-bounding algorithms of the last section - SFB; AFB; ConcFB. These are an excellent test case, since the main difference between them is their level of synchronization. A good starting point is to examine the bottleneck of SFB: idle-time imposed by waits for approval of forward-consistency. Consider some agent A_i, presently holding the CPA with its partial assignment $CPA_{0,\ldots,i-1}$ and attempting an assignment. A_i's first move is to choose some value, verify backwards-consistency, which will consume i constraint checks, and then send FB requests to all agents A_j $(j > i)$ and await reply. The time (as measured in constraint checks) needed for a receiving agent A_j to handle an FB-request is: $|dom| \times (i+1)$, because A_j has $|dom|$ values to check, each against the $i+1$ agents $0, 1, ..., i$ assigned so far on the request's CPA. Thus, A_i will wait $|dom| \times (i+1)$ time for all replies. This is the waiting time caused solely by computation (message delivery times are accounted for later on).

Note that while agent A_i holds the cpa, all agents $A_0, \ldots, A_{i-1}$ are idle. For each agent A_k (where $k < i$), while A_i holds the CPA, A_k's idle time amounts to the number of tries for a value assignment performed by agent A_i times i for each try, and times $|dom| \times (i+1)$ which is A_i's waiting time for replies for each of its value assignment tries:

$$\begin{aligned} Tries(i) \times \{AssignTime(i) + FBtime(i)\} = \\ Tries(i) \times \{i + |dom| \times (i+1)\} \end{aligned} \quad (1)$$

$Tries(i)$ is trivially bounded by $|dom| \times Tries(i-1)$, where $Tries(-1) = 1$, but it is reduced both as i increases, as well as with search progress, as lower Upper-bounds are discovered and assignments of higher cost are pruned earlier. Based on the above considerations, the maximal total idle time during the search due to computations can be formulated as:

$$\sum_{i=0}^{n-1} \left(\sum_{j<i} Tries(i) \times (i + |dom| \times (i+1)) + \sum_{j>i} Tries(i) \times i \right) \quad (2)$$

Note that the inner sums are for j, but they are independent of it. It simply reflects the fact that every agent is idle during the computation performed by agent A_i. The above computation of idle time is not a non-concurrent time. On the contrary, it can add up to $n-1$ times the non-concurrent time (in SyncBB, a sum of this sort is exactly $n-1$ times the number of NCCCs). It reflects the total time wasted by n system nodes as they were pending for messages. This idle time needs to be utilized by the system and its minimization is a main goal of any concurrent algorithm.

The method of saving idle time employed by the AFB algorithm (and all other asynchronous branch & bound algorithms) saves *FBtime(i)* on the one hand, but sometimes significantly increases $Tries(i)$ on the other. This happens because less values are pruned by enforcing backwards-consistency (neglecting forward-consistency enforcement). AFB's increase of $Tries(i)$ (compared to SFB) triggers a ripple effect: it increases $Tries(i+1)$, since agent A_{i+1} sends an FB-estimate to A_i and expands the CPA and sends FB-requests of its own. This ripple continues

until a new FB-request message from A_i arrives at the top of the inbox of A_j $(j > i)$, when a new ripple is created. This does not only happen at the expense of idle time. When some agent A_j is occupied with a calculation on an irrelevant CPA, a newer CPA to explore (or estimate upon) can be waiting in its inbox, causing an increase of the sender's idle time.

The above analysis of idle time assumes that messages arrive instantanously, counting only computation time. In real-life distributed systems, where communication is transferred by a computer network, messages take a finite time to arrive at their destination. It turns out that once messages take time to be delivered, a more realistic analysis of the impact of synchronization class on non-concurrent run-time of DCOP algorithms can be performed.

4.1 Adding Message Delivery Times

Assume that message delivery consumes a random time, uniformly distributed in the range of $[0,...,m]$, time units. This may happen when network messages are sent by servers that respond differently to geographical distances between nodes. A uniform distribution model for such delays is compatible with some basic assumed symmetry.

Since message transfer time is not a constant, messages do not arrive all at once. This may cause (in AFB's case) some agents to receive an irrelevant FB-request, while others will eliminate it—receiving a newer FB attempt from a higher-priority agent first.

SFB's synchronous behavior dictates a wait for all agents to reply with an FB-response before making a decision. This means that the time spent from the moment agent A_i sends an FB-request and until all responses accumulate back is expected to be $max_{j>i}\{m_{i\to j} + m_{j\to i}\}$ where $m_{k\to l} \backsim U[0..m]$. The distribution function of the time needed for an FB-request and reply to arrive at lower-priority agents is therefore:

$$F^{(n-i)}(t) = \begin{cases} \left(\frac{t^2}{2m^2}\right)^{n-i} & ,0 \le t \le m \\ \left(\frac{t(4m-t)}{2m^2} - 1\right)^{n-i} & ,m \le t \le 2m \end{cases}$$

A graphical description of such a function is presented in figure 2. As i grows (less neighbors are ahead), the graph curves up for lower values of t, i.e. as there are less neighbors to send to (and receive from) messages, it is more likely that the maximal time for send and receive will be shorter. For the extreme case where $i = n-1$, where there is only one neighbor ahead, $Pr(send + receive \le m) = 0.5$, and the curve is simply wider.

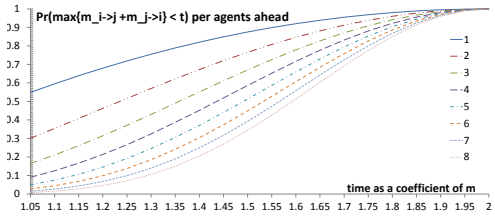

Figure 2: Distribution function of time for FB request and reply.

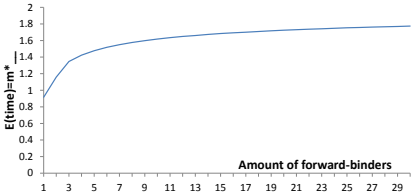

Figure 3: Expected time of FB-request-and-reply per amount of agents ahead.

In SFB's case, when some agent A_i sends an FB-request, it is expected to wait $E(max\{m_{i\to j} + m_{j\to i}\})$, which once again depends on the amount of agents ahead, as described in figure 3.

Upon success of an assignment, where FB-responses will prove the CPA to be forward-consistent, AFB is expected to save the time needed for all response-messages to arrive and the approval message from A_i to A_{i+1}, which amounts to: $E(max_{j>i} m_{j\to i} + m_{i\to i+1})$. Although this formula is dependent upon the amount of agents ahead, its outcome is in $[m..3m/2]$. The relation between maximal message delivery time m and computation time, denoted by c, is imperative to the comparison. If $c << m$, then upon success AFB saves up to $3m/2$ compared to SFB which waits for all responses. Upon failure of Forward-bounding, in this scenario, AFB does not waste any time, since any irrelevant message arriving before the next FB-request from A_i arrives while the analogous SFB agent is idle. When the new FB-request from i arrives, it may have to wait to be processed as much as: $(j-i)\cdot c$ time units (based on the maximal amount of irrelevant FB-requests that may arrive before it). If c is small enough compared to m, AFB has not lost any significant time.

In the other extreme $c >> m$. Upon success, AFB saves an insignificant amount of time (m), where failure in forward-bounding an assignment of some A_i generates a ripple of irrelevant messages from A_{i+1} which are expected to arrive to other agents A_j, $j > i+1$ just before the new FB-request A_i is about to send. This causes a delay of roughly c time units.

When $c \approx m$, if forward-bounding succeeds, AFB is expected to save up to m time units (pending, as above, on the amount of agents ahead), as SFB has

to wait for FB responses (maximal message time) before passing on the CPA. Note that the time c added to SFB is not insignificant, but A_{i+1} consumes the same time in AFB as it expands the CPA (and responds to the FB-request). Upon failure of forward-bounding, there is a significant cost to AFB's run time. This is because some agent A_j, $(j > i+1)$ may be occupied with irrelevant computation caused by messages from all agents $A_{i+1}, A_{i+2}, ..., A_{j-1}$ at the time the next message from A_i arrives. The actual amount of delay differs according to the precise relation between c and m and the amount of agents ahead, but by examining for instance, the delay imposed by A_{i+3} as it may receive irrelevant messages from A_{i+1} as well as from A_{i+2} causing it to become unavailable to process the next (relevant) FB-request from A_i as it arrives, it is clear that the search process could be delayed by up to $2c$ time units. Under the assumption that c is roughly $m/2$ (the expected message time), the expected delay imposed by A_{i+3} for each failure is slightly over $m/7$ ($0.14731 \cdot m$)[2]. Thus, for a success/failure ratio of $1:7$ or worse, SFB runs faster.

4.2 Comparing to ConcFB

Let us turn now to the method of decreasing idle time that is employed by the third algorithm - ConcFB. In ConcFB, the search space is divided into several disjoint sub-spaces, each explored by an independent Synchronous Search-Process. Each SP has a random or dynamically created order of agents, so while some agent awaits a reply or a CPA in one SP, it is kept busy computing for other SPs. This calls for some more memory, to keep track of each SP, but idle time is reduced, and there is less irrelevant computation. The only time where a computation may be irrelevant is when a New_Solution message is in the inbox queue of some agent A_i, carrying an UB so low that it is about to prune the CPA that A_i is currently computing upon. New_Solution messages are a lot rarer than both FB-requests and CPA messages. Empirically, problems with 12 agents, and 6 domain values each, have about 10 New_Solution messages during the entire search. The total amount of messages during the same concFB search is roughly 200,000. Consequently, this type of concurrency does not create massive irrelevant computation, and since it also evenly spreads the computation load (and order), it is less susceptible to the impact of message delays.

For better intuition, consider ConcFB's time utilization potential: As each SP is an SFB-like protocol, at any given time, a search-process may be (1)at some agent, expanding the cpa or (2)calculating FB estimates. A third option exists, where the SP awaits attendance in an occupied agent's inbox, which is why the mechanism must balance the amount of concurrent SPs, but this is also not directly relevant to the current analysis, that focuses on minimizing idle-time - keeping agents busy at one hand, and not increasing the amount of computation needed on the other. While an SP is in the assign phase (1) a single agent is computing thus only $1/n$ of the agents are active and the rest are idle (ignoring the existence of other SPs for the moment). While an SP is on an FB phase, taking the average case of FB estimate for a median agent A_i $(i = n/2)$, a fraction α of unassigned agents are neighbors of A_i and thus $\alpha/(0.5 \cdot n)$ agents are active at that time, and the rest are idle. Had we known the relation between time consumptions of (1) and (2) we could calculate the expected amount of idle agents at a random time, and moreover calculate the amount of concurrent SPs needed to maximize agent activity levels at all times (recall that agents are dynamically ordered, thus the load is expected to be evenly distributed). Increased system delay times obviously lowers the system's activity level, and therefore calls for some more concurrent SPs to compensate.

[2]Based on a complex probabilistic aggregation of probabilities for delay times the delays imposed in each possible message arrival time and order for $m = 10$, $(n-i) = 10$ and $c = 5$.

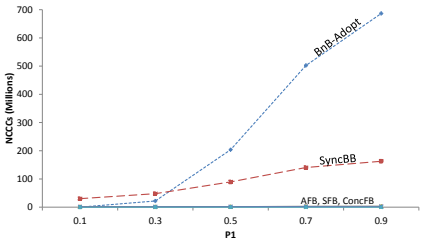

Figure 4: Runtime in NCCCs - synchronization classes.

5 EXPERIMENTAL EVALUATION

The first set of experiments, depicted in Figures 4 and 5, shows a categorical partition of algorithms into synchronization classes and the clear correlation between synchronization level and performance, measured by non-concurrent constraint-checks(Meisels et al., 2002) and network load. This experiment was run over problems with 10 agents and 6 domain values per agent. p_1 marks the probability for two agents to share a constraint, and constraint costs are randomly distributed in $[0, 1, ..., 100]$. For each p_1 value, 100 random problems have been generated and averaged.

To correlate between the class of synchronization and performance level, recall that BnB-

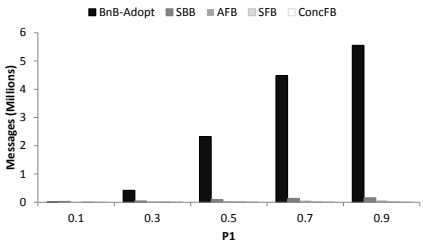

Figure 5: Total message count.

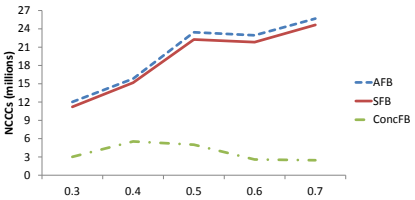

Figure 6: Synchronous vs. Asynchronous vs. concurrent - $c >> m$

Adopt(Gutierrez and Meseguer, 2010) was categorized as a *depth-first* class, which is stronger than ADOPT's class. ADOPT could not complete the search in this size of problems under our simulation limits. Higher than BnB-Adopt in synchronization level are *backwards-consistent* algorithms such as SyncBB, which is shown to perform better as problems become more dense. The other three algorithms (AFB, SFB, ConcFB) seem to perform similarly. This is due to the scale of the graphs in Figures 4 and 5. The relations between these three are clearer when compared separately, as in the following experiments.

The next group of experiments demonstrates the differences between SFB, AFB and ConcFB as analyzed in Section 4. It shows that under different network conditions, AFB and SFB may outperform one another, depending on the ratio between computation time c and message delivery time m. It also shows that a synchronized multiple-SP mechanism outperforms them both under all network conditions. These experiments were run on the same batches of problems, each problem has 12 agents and 6 domain values per agent, with p_1 values of $0.3, 0.4, 0.5, 0.6, 0.7$. For every p_1 value 50 random problems were averaged.

According to the description of Section 4, SFB is a forward-consistent search-process whose main drawback is idle-time. The mechanisms of the other two algorithms - AFB and ConcFB - offer to eliminate this drawback to achieve a faster search-algorithm.

Figure 6 shows the results of an experiment where messages arrive instantly, and "time" is measured in NCCCs. This simulation corresponds to the case where computation time is much longer than communication time. As estimated in Section 4, when $c >> m$ AFB's irrelevant computation does not happen only at the expense of idle-time. Agents may become unavailable to respond to relevant messages and lengthen the algorithm's runtime. ConcFB, on the other hand, has no irrelevant computation and its method of splitting the search-space into separate subspaces and searching those simultaneously turns out to perform much better.

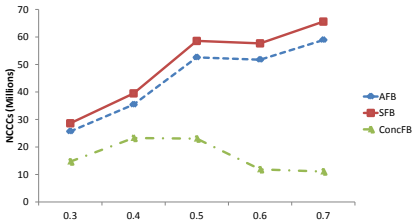

Figure 7: Synchronous vs. Asynchronous vs. concurrent - $c \approx m$

Figure 7 depicts the results of running the same problems of Figure 6 in a system where message delivery time is not zero (simulation is based on (Zivan and Meisels, 2006b; Zivan and Meisels, 2006c)). Each message is delayed randomly in the range of $[0, 1, ..., 100]$, where time is measured in NCCCs. This gives an expected delay of 50 time units per message, and is comparable to the average number of constraint-checks an agent performs per message (as sampled empirically). Similar experiments with normally distributed delays shows same relations, and were not brought here for lack of space

Message delay times accumulate in the NCCCs clock, and hence the total time measures increase dramatically. AFB is slightly faster than SFB in such a system. Not waiting for forward-bound was beneficial and had saved more time than the time consumed by irrelevant computation. In contrast, the runtime of the ConcFB algorithm stays better than the other two algorithms (as in the case of instantaneous masseges).

The last experiment focuses on very large message delays, so that message delivery takes a lot longer than computation time, i.e. $c << m$. Random delays were in the range of $[0, 1, ..., 1000]$ per message which accumulates to a Non-Concurrent Steps Count (NCSC) clock(Meisels et al., 2002). This clock actually measures the longest path in messages throughout the algorithm's run. Since computation time is irrelevant in the current scenario, it is a natural clock.

The graph demonstrates the claim of Section 4. In a system where messages have high cost, AFB performs better than SFB since irrelevant computation is done at the expense of idle-time. ConcFB shows resistance to very long delays. Instead of risking irrelevant computation, each SP explores a different part of the search tree, and whenever a high-quality solution is found, all other SPs can be pruned faster.

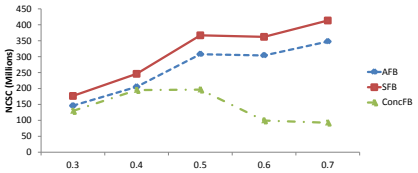

Figure 8: Synchronous vs. Asynchronous vs. concurrent, $c << m$.

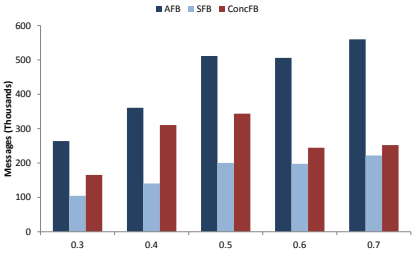

Figure 9: Synchronous vs. Asynchronous vs. concurrent - average network load by problem density.

Regarding the network's communication load, Figures 9 and 10 illustrate the typical load-ratio between the three algorithms. Considering the system's overall communication load as presented in Figure 9 both AFB and ConcFB exchange more messages than SFB. However, as shown in Figure 10, ConcFB spreads the load evenly between all agents (besides the first agent that merely initiates the search).

6 CONCLUSIONS

A classification of DCOP search protocols into classes of synchronization has been presented. The analysis of non-concurrent run-time of DCOP algorithms identifies a drawback of synchronization in the form of idle time and the trade-off with irrelevant computation that are the result of asynchronous algorithms attempting to decrease agents' idle time. The solution was given by the *Conc* mechanism. It uses several distinct search processes concurrently, each responsible for a different part of the search space. When the concurrent mechanism is applied to synchronous search-processes, irrelevant computations are avoided, as well as an overload of the network with redundant messages.

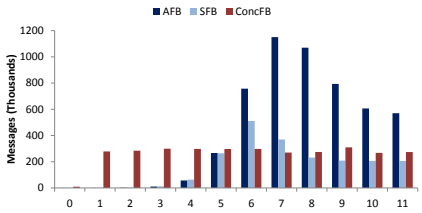

Figure 10: Synchronous vs. Asynchronous vs. concurrent - average network load by agent.

REFERENCES

Bessiere, C., Maestre, A., Brito, I., and Meseguer, P. (2005). Asynchronous Backtracking without adding links: a new member in the ABT Family. *Artificial Intelligence*, 161:1-2:7–24.

Chechetka, A. and Sycara, K. (2006). An any-space algorithm for distributed constraint optimization. In *Proc. AAAI Spring Symp. Distr. Plan Sched. Manag.*

Gershman, A., Meisels, A., and Zivan, R. (2009). Asynchronous Forward Bounding for Distributed COPs. *J. Artif. Intell. Res. (JAIR)*, 34:61–88.

Gutierrez, P. and Meseguer, P. (2010). Saving redundant messages in bnb-adopt. In *Proc. 24th AAAI Conf. Artif. Intell. (AAAI-10)*, pages 1259–1260.

Gutierrez, P., Meseguer, P., and Yeoh, W. (2011). Generalizing adopt and bnb-adopt. In *Proc. 23rd Intern. Joint Conf. Artif. Intell. (IJCAI-11)*, pages 554–559.

Hirayama, K. and Yokoo, M. (1997). Distributed partial constraint satisfaction problem. In *Proc. CP-97*, pages 222–236.

Meisels, A. (2008). *Distributed search by constrained agents*. Springer Verlag.

Meisels, A., Razgon, I., Kaplansky, E., and Zivan, R. (2002). Comparing performance of distributed constraints processing algorithms. In *Proc. AAMAS-2002 Workshop on Distributed Constraint Reasoning DCR*, pages 86–93, Bologna.

Modi, P. J., Shen, W., Tambe, M., and Yokoo, M. (2005). Adopt: asynchronous distributed constraints optimization with quality guarantees. *Artificial Intelligence*, 161:1-2:149–180.

Netzer, A., Meisels, A., and Grubshtein, A. (2010). Concurrent Forward Bounding for DCOPs. In *Proc. 12th intern. workshop Dist. Const. Reas. (DCR-10) (AAMAS-10)*, Toronto.

W. Yeoh, X. S. and Koenig, S. (2009). Trading off solution quality for faster computation in dcop search algorithms. In *Proc. 21st Intern. Joint Conf. Artif. Intell. (IJCAI-09)*, pages 354–360.

Yeoh, W., Felner, A., and Koenig, S. (2010). Bnb-adopt: An asynchronous branch-and-bound dcop algorithm. *J. Artif. Intell. Res. (JAIR)*, 38:85–133.

Yokoo, M. (2000). Algorithms for distributed constraint satisfaction problems: A review. *Autonomous Agents & Multi-Agent Sys.*, 3:198–212.

Zivan, R. and Meisels, A. (2006a). Concurrent search for distributed csps. *Artif. Intell.*, 170(4-5):440–461.

Zivan, R. and Meisels, A. (2006b). Message delay and asynchronous discsp search. *Archives of Control Sciences*, 16(2):221–242.

Zivan, R. and Meisels, A. (2006c). Message delay and discsp search algorithms. *Ann. Math. Artif. Intell. (AMAI)*, 46:415–439.