Interleaved Backtracking in Distributed Constraint Networks

Youssef Hamadi
Hewlett-Packards Labs
Filton road, Stoke Gifford, Bristol BS34 8QZ, United Kingdom
yh@hplb.hpl.hp.com

Abstract

The adaptation of software technology to distributed environments is an important challenge today. In this work we combine parallel and distributed search. By this way we add the potential speed-up of a parallel exploration in the processing of distributed problems. This paper extends DIBT, a distributed search procedure operating in distributed constraint networks [6]. The extension is twofold. First the procedure is updated to face delayed information problems upcoming in heterogeneous systems. Second, the search is extended to simultaneously explore independent parts of a distributed search tree. By this way we introduce parallelism into distributed search, which brings to Interleaved Distributed Intelligent BackTracking *(IDIBT). Our results show that 1) insoluble problems do not greatly degrade performance over DIBT and 2) superlinear speed-up can be achieved when the distribution of solution is nonuniform.*

Keywords: Distributed Constraint Satisfaction, Distributed AI, Collaborative Software Agents, Search

1 Introduction

The constraint satisfaction problem (CSP) is a powerful framework for general problem solving. It involves finding a solution to a constraint network; i.e., finding values for problem variables subject to constraints that are restrictions on which combinations of values are acceptable. This formalism has been extended to tackle distributed problems. In the distributed constraint satisfaction paradigm (DCSP) a problem is distributed between autonomous agents which are cooperating to compute a global solution. The raise in application interoperability combined to the move towards decentralized decision process in complex systems raise the interest for distributed reasoning. In this work we show how to enhance efficiency of distributed search.

The basic method to search for solution in a constraint network is depth-first backtrack search (DFS) [3], which performs a systematic exploration of the search tree until it finds an instantiation of values to variables that satisfies all the constraints. DFS has been extended to parallel-DFS to speed-up the resolution process [11]. Interestingly parallel-DFS showed that under some assumptions, speed-up could be superlinear.

In this paper we present *Interleaved Distributed Intelligent BackTracking* an algorithm performing parallel-DFS in DCSPs. Our algorithm interleaves the exploration of subspaces within each agent. Between distinct agents parallelism is achieved since they can consider distinct subspaces at the same time. Experiments show that 1) insoluble problems do not greatly degrade performance over DIBT and 2) on problems with nonuniform search space, IDIBT allows superlinear speed-up over DIBT.

In the following, we first give a basic definition of the CSP/DCSP paradigm, completed by a distinction between parallel and distributed search. Then, we present DisAO a distributed variable ordering method, and we describe and analyze IDIBT. Afterwards, we give an experimentation with random DCSPs and N-queens problems, followed by a general conclusion.

2 Background

2.1 Constraint satisfaction problems

A *binary constraint network* involves a set of n *variables* $\mathcal{X} = \{X_1, \dots, X_n\}$, a set of *domains* $\mathcal{D} = \{D_1, \dots, D_n\}$ where D_i is the finite set of possible *values* for variable X_i and $\mathcal{C}$ the set of binary constraints $\{C_{ij}, \dots\}$ where C_{ij} is a constraint between i and j. $C_{ij}(a,b) = true$ means that the association value a for i and b for j is allowed. Asking for the value of $C_{ij}(a,b)$ is called a *constraint check*. $G = (\mathcal{X}, \mathcal{C})$ is called the *constraint graph* associated to the network $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

A *solution* to a constraint network is an instantiation of the variables such that all the constraints are satisfied. The *constraint satisfaction problem* (*CSP*) involves finding a solution in a constraint network.

2.2 Distributed constraint satisfaction

A *distributed constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$ is a constraint network (binary in our case), in which variables and constraints are distributed among a set $\{Agent_1, \ldots, Agent_m\}$ of m autonomous sequential processes called *agents*. Each agent $Agent_k$ "owns" a subset A_k of the variables in $\mathcal{X}$ in such a way that $\mathcal{A} = \{A_1, \ldots, A_m\}$ is a partition of $\mathcal{X}$. The domain D_i (resp. D_j), the constraint C_{ij} (resp. C_{ji}) belongs to the agent owning X_i (resp. X_j)[1]. In the present work, we limit our attention to the extreme case, where there are n agents, each only owning one variable, so that $\mathcal{A} = \mathcal{X}$. Thus, in the following, $Agent_i$ will refer to the agent owning variable X_i. Of course, the instantiation of a single variable can relate the solution of an embedded large subproblem and in fact, each inter-agent constraint can represent a large set of constraints.

Initially, the graph of *acquaintances* in the distributed system matches the constraint graph. So, for an agent $Agent_i$, Γ is the set of its acquaintances, namely the set of all the agents $Agent_j$ such that X_j shares a constraint with X_i. The *distributed* CSP (*DCSP*) involves finding a solution in a distributed constraint network.

2.3 Communication model

For a DCSP, we assume the following communication model [12] (in every way classical for distributed systems). Agents communicate by sending messages. An agent can send messages to other agents if and only if it knows their address in the network. The delay in delivering messages is finite. For the transmission between any pair of agents, messages are received in the order in which they are sent. Agents use the following primitives to achieve *message passing* operations:

- $sendMsg(dest,$"m"$)$ sends message m to the agents in $dest$.
- $getMsg()$ returns the first unread message available.

2.4 Distributed v Parallel Search

Parallel backtrack search is used to speed-up the resolution process [11, 7]. Distributed backtrack search faces a situation where the whole problem is not fully accessible; resolution is enforced by collaboration between subproblems.

Both framework use several processing units. In parallel search, N processors concurrently perform backtracking in disjoint parts of a state-space tree. In distributed search, distinct subproblems are spread on several processing units and backtracking is performed by the way of collaboration.

Part a) of figure 1 presents an example of parallel exploration. Here, the problem is duplicated on two processors P_0 and P_1. P_0 is in charge of the subspace characterized by $X_1 = a$, P_1 explores the remaining space. During the computation, message passing is useless. However, since a processor can exhaust its task before another (good heuristic functions, filtering, ...), dynamic load balancing is used [11]. Usually, an idle unit asks a busy one for a part of its remaining exploration task.

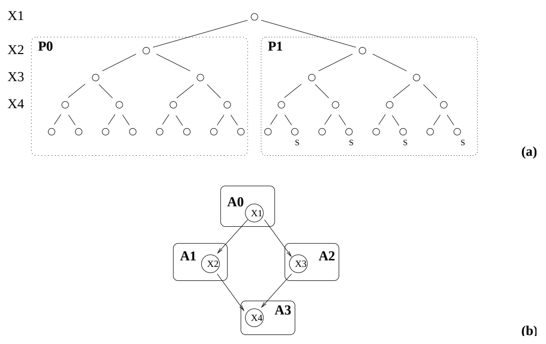

Figure 1. Tree searches: (a) parallel search, (b) distributed search

Part b) of the figure presents a distribution of this 4-variables problem between four autonomous agents. Here, state-space exploration uses local resolution for each subproblem with negotiation on the shared constraints (links).

In the following we show how to introduce the efficiency of parallel search in a distributed exploration.

3 Interleaved Distributed Intelligent BackTracking

We present here IDIBT as a generalization of DIBT. IDIBT mixes parallel and distributed search. The reader can report to [5] for more details on DIBT.

DIBT realizes a DFS between the agents of a distributed CSP. DFS is a general complete resolution technique widely used for its storage efficiency. Given a

[1]We suppose that the constraint network is such that $(\mathcal{X}, \mathcal{C})$ is a symmetric graph.

variable and value ordering, it generates successive instantiations of the problem variables. It tries to extend a partial instantiation by taking the next variable in the ordering and by assigning it a value consistent with previously assigned variables. If no value can be found for the considered variable, the algorithm backtracks. In the basic DFS scheme, it goes back to the previous variable in the ordering and changes its value. In some refined backtracking schemes, the algorithm jumps back to the origin of the failure.

Our framework is totally asynchronous but we need an ordering between related agents to apply the backtracking scheme which ensures completeness. In the following we present our distributed ordering method followed by the IDIBT search process.

3.1 Distributed Agent Ordering

The practical complexity of a search process is highly dependent on user's heuristic choices such as value/variable ordering. Usually these heuristics take advantage of domain-dependent knowledges. Each agent can use particular heuristics in the exploration of its subproblem. But in the DCSP, agents must collaborate to use an efficient ordering in the distributed search process. We are using here DisAO, a generic method for a distributed computation of any static agent ordering. With this algorithm, agents cooperatively build a global ordering between the subproblems. This ordering defines a hierarchical relation between the agents.

3.1.1 Algorithm

In our system, each agent locally computes its position in the ordering according to the chosen heuristic. Concretely, each agent determines the sets Γ^+ and Γ^-, respectively $children$ and $parent$ acquaintances, w.r.t. an evaluation function f and a comparison operator *op* which totally define the heuristic chosen. This is done in the lines 1 to 2 of algorithm 1. Notice that the evaluation function f can involve some communication between the agents. To avoid a complex communication behavior, it is better to use heuristics for which the associated function f involves only local communications; i.e., between neighbor agents.

After that, agents know their *children* (Γ^+) and *parents* (Γ^-) acquaintances. During the search, they will send instantiation value to Γ^+, and in case of dead-end, they will backtrack to the *first* agent in Γ^-. To achieve backtracking we need a total ordering on Γ^-. This is done in the second part of algorithm 1 (lines 3 to 4). Agents without children state that they are at level one, and they communicate this information to their acquaintances. Other agents take the maximum level value received from children, add one to this value, and send this information to their acquaintances. Now, with this new environmental information, each agent rearranges (total order) the agents in its local Γ^- set by increasing level. Ties are broken with agent tags. These total orders will be used during backtracking. Finally, for fitting each total order Γ^-, the constraint graph is extended with zero or more additional edges (lines 4). These new edges are tautological constraints. Their purpose is the enforcement of completeness by local search space initialization in the forward exploration phases (see section 3.3). We do not present details about this computation here. In summary, each agent communicates its ordered Γ^- set to its parents. These agents can locally modify their sets by adding lower (resp. higher) agents in their Γ^- (resp. Γ^+). This process is repeated until stabilization; i.e., no more Γ modification.

Algorithm 1: Distributed variable ordering

```
  begin
      % $\Gamma$ split;
1     $\Gamma^+ \leftarrow \emptyset$; $\Gamma^- \leftarrow \emptyset$;
      for each $Agent_j \in \Gamma$ do
          if ($f(Agent_j)$ op $f(self)$) then $\Gamma^+ \leftarrow \Gamma^+ \cup \{Agent_j\}$;
2         else $\Gamma^- \leftarrow \Gamma^- \cup \{Agent_j\}$;
      % $\Gamma^-$ ordering;
3     $max \leftarrow 0$;
      for ($i = 0; i < |\Gamma^+|; i++$) do
          $m \leftarrow getMsg()$;
          if ($m =$ value:$v$; from:$j$) then
              if ($max < v$) then $max \leftarrow v$;
      $max++$;
      sendMsg($\Gamma^-$, "value:$max$; from:self");
      sendMsg($\Gamma^+$, "position:$max$; from:self");
      for ($i = 0; i < |\Gamma^-|; i++$) do
          $m \leftarrow getMsg()$;
          if ($m =$ position:$p$; from:$j$) then $Level[j] \leftarrow p$;
      Order $\Gamma^-$ according to $Level[]$ ;
4     Extend $\Gamma^-$ ;
  end
```

Figure 2 gives an illustration of this distributed processing for the *max-degree* variable ordering heuristic. On the left side of the figure a constraint graph is represented. For achieving the max-degree heuristic, algorithm 1 must be called by each agent with the function $f(Agent_i) = |\Gamma_i|$ (where Γ_i is the set of acquaintances of $Agent_i$) and the comparison operator $op = '<'$. In case of ties, this operator can break them with agent tags.

Once algorithm 1 has been applied, the static variable ordering obtained is the one presented on the right side of Fig. 2. Arrows follow the ordering relation, which represents the instantiation transmission order of the search procedure. The link between $Agent_4$ and $Agent_3$ comes from the interconnection of $Agent_7$'s parents. $Agent_7$ will go back to X_3 then to X_4 if

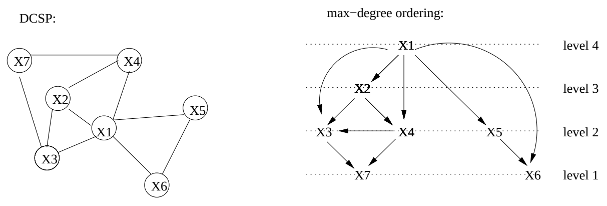

Figure 2. Distributed variable ordering

$Agent_3$ has no remaining solution. During forward exploration, a change in X_4 will be reported to X_3 and to X_7. These agents will then get back their whole search space.

3.1.2 Analysis

In the worst case, w.r.t. a fully connected network with n agents, the split of Γ uses $O(n)$. The exchange of **value** among the path of n agents use $O(n^2)$ messages; i.e., level one agent sends $n-1$ messages, level 2 agent $n-2$ and so on. These messages can overlap, this bring $O(n)$ local operations for performing these transmissions. The transmission of **position** messages is similar but from the top to the bottom. The extension of the ordering in the hierarchy adds no link but requires $O(n^2)$ message to exchange Γ^- sets. According to that, DisAO uses $O(n)$ local operations and $O(n^2)$ messages in the worst case[2].

PROPERTY 3.1 ()
$\forall A_i$, if $\exists A_j, A_k$ such that $A_j \to A_i$ and $A_k \to A_i$, then $\exists A_j \to A_k$ or $\exists A_k \to A_j$.

We have $A_j \to A_1, A_i$ and $A_k \to A_2, A_i$ with $A_1 \in \Gamma^-(A_i)$ and $A_2 \in \Gamma^-(A_i)$. By definition we have $f(A_1) opf(A_2)$ or $f(A_2) opf(A_1)$ then by Γ^- extension we have $A_2 \to A_1$ or $A_1 \to A_2$. We can follow the previous reasoning by considering $A_i = A_1$ or $A_i = A_2$.

PROPERTY 3.2 ()
For a problem $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$, if (X, C) is connected, the directed graph computed with DisAO has an unique agent such that $\Gamma^- = \emptyset$.

The proof is straight-forward, if we consider 3.1. In a DisAO ordering, there is a unique source and the hierarchy is made of subproblems (involving several agents) organized in a global tree.

[2]Of course for predefined applications, the DisAO pre-processing step could be avoided.

Finally, we can remark that in the resulting ordering, at a particular level, unconnected agents are independent. Connected ones are linked by tautological constraints. This means that their information will just initialize the search space without loosing current instantiation. Hence, in each level, agents can perform parallel computations at the same time. This observation will be important when we will consider the complexity of distributed search.

3.2 IDIBT: Distributed and Parallel search

To add parallel search in our distributed framework, we must divide a search space in independent parts. In each part a distributed backtrack search will take place. In the system, we will have two kind of agent with distinct behaviors.

- a $Source$ agent, which will partition its search space in several subspaces called $Context$
- the remaining agents which will try to instantiate in each context.

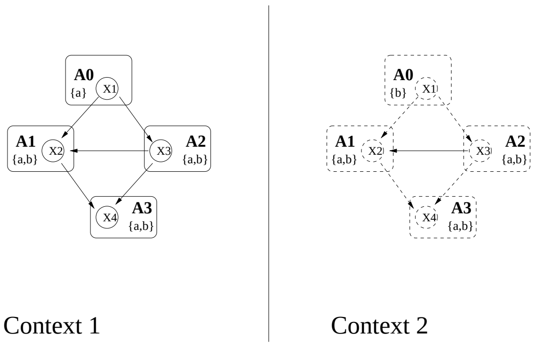

Figure 3. Interleaved search

For illustration purpose, consider the figure 3. Here the four-variables problem of figure 1 is presented for exploration between four agents using two resolution context. The source agent A_0 will use value a in the first context and value b in the second context. According to property 3.2, this agent is unique. Remaining agents will keep their local search space $\{a, b\}$.

There is no duplication of processing units here. Agents will successively consider search in the different contexts. This interleaving will be achieved by message passing operations. The context of resolution added within each message will allow an agent to successively explore the disjoint search spaces.

3.2.1 Algorithm

The global scheme of the search process is the following (see algorithm 2 and data structure below). In the initialisation phase (lines 1 to 3), the source agent divides its search space in NC subspaces. Remaining agents will use the same space D in each context. In each context c each agent instantiates its variable with respect to its parent constraints. Each timestamp counter $valueCpt_c$ is then set to one. After instantiation, the agent informs its children of its chosen value (message content starting by "**infoVal**").

Interactions start at line 4. Here each incoming message is interpreted in a particular context c (lines 5 and 6).

An "**infoVal**" message from acquaintance j is processed as follow (line 7). First the reported value is stored in $value[j]_c$ then the associated timestamp $valueCpt[j]_c$ is incremented. Finally the agent tries to get a value compatible with the new message. If a compatible value is found, an "**infoVal**" message with context c informs children of the new choice[3]. If no value satisfies the constraints with the agents in Γ^-, a backtrack message is sent in context c to the nearest parent (message content starting by "**btSet**" in line 8) . This message includes the local ordered set Γ^- of parent acquaintances, their level positions and agent beliefs about their timestamps $valueCpt[\Gamma^-]_c$.

The receiver of the backtrack message (line 9), checks the validity of it by comparing its timestamp with the reported one and by checking that shared acquaintances are reported with the same timestamps too (function $contextConsistency$). In case of different values, the sender and/or the receiver have not yet received some information. Backtrack decision could then be obsolete or badly interpreted. Here IDIBT differs from the original DIBT [6] which considers consistency between the receiver's value and the belief about it enclosed in the message. This assumption is correct if we assume that the network do not have different transmission time between distinct agents. However if transmission time are heterogeneous, the global test of IDIBT is more safe.

When the comparison matches, there are two possible behavior. If the agent can find a compatible $myValue_c$ in the remaining search space, this value is addressed to Γ^+ in line 10. If such a value cannot be found, we must consider two cases. The first one is an agent without possibility for backtracking, (line 11). This agent has detected problem insolubility in the subspace c. A message $noSolution$ in context c is sent to a *System* agent. This extra agent stops the distributed computation in context c by broadcasting a *stop* message in the whole multiagent system. With this information agents can stop the processing of context c messages. If **all** the context have no solution, the computation is finished. In addition, it can also stop the computation when a solution is found. A global state detection algorithm [1] is used to detect whole satisfaction. Global satisfaction occurs when in **a** particular context c, agents instantiated according to parent constraints are waiting for a message (line 5) and when no message with context c transits in the communication network; i.e., each local instantiation safisfies the local constraints.

If there exists a parent for backtracking, the agent address a backtrack message to the nearest agent in the ordered set union of Γ^- and the sender set (line 12). This new set is attached to the message with related information about agents for ensuring continuity of graph-based backjumping [2]. Now the sender is waiting for an incoming message, but if the goal is to maximize satisfaction in the system, the agent could get back its initial local search space and a value compatible with previous ones. This simple addition is a simple way toward a max-sat strategy.

[3]Of course, current value $myValue_c$ can already satisfy the constraints with j, in which case, information of children is useless.

Primitives and data structures

IDIBT uses the following structures and methods:

- NC is the number of resolution context. *self* is the agent running the algorithm, $D_{self,c}$ is its domain in context c. $myValue_c$ current value in the context c. $myCpt_c$ current instantiation number in context c. This value will be used as a timestamp in the system. $value[]_c$ stores parent acquaintances values in context c. $valueCpt[]_c$ stores for each parent the current instantiation number, in the right context.

- $getValue(type, c)$, if a compatible value is found, $myCpt_c$ is incremented. If $type$='info', returns the first value in $D_{self,c}$ compatible with agents in Γ^-, starting at $myValue_c$[4]. If $type$='bt', returns the first value after $myValue_c$ in $D_{self,c}$ compatible with agents in Γ^-.

- first(S) returns the first element of an ordered set S. With our application, returns the nearest agent in S. $merge$(s1,s2) takes two ordered sets and returns their ordered union.

- $contextConsistency(set, reportedValueCpt, c)$, set contains an ordered list of agents,

[4]The search for a new compatible value starts from the current value for keeping the maximum of previous work. For ensuring completeness, the values that are before $myValue_c$ in $D_{self,c}$ are put at the end of $D_{self,c}$.

$reportedValueCpt$ contains for each agent in set timestamps computed by the sender of the current message. This function ensures that, firstly reported timestamp for *self* is the good one; i.e., equal to $myCpt_c$, secondly that for the shared acquaintances agents, reported timestamps are the same than in $valueCpt[]_c$. This mechanism ensures that agents have the same beliefs about the shared parts of the system.

- The previous $sendMsg$ function becomes $sendMsg(dest, m, c)$, which sends message m to the agents in $dest$ in context c.

3.3 Analysis

Completeness

PROPERTY 3.3 ()
When an agent A_i changes its instantiation, agents A_j such that $\exists A_i \to A_j$ will reconsider their whole search space.

The proof is direct if we consider the algorithm 2. When an agent changes its value, Γ^+ agents receive it. These agents can keep their current instantiation or change it, but they always resume their local search space. By propagation of instantiations between agents, 3.3 is verified.

PROPERTY 3.4 ()
If A_i changes its instantiation according to a **btSet** message initially upcoming from A_j, each agent A_k such that $\exists A_i \to A_k \to A_j$ has exhausted its search space.

Consider an A_k which contradicts 3.4. Since we have a path between A_k and A_j, A_k will be included in the ordered union of Γ^- sets which gives the successive receivers of backtracking message. Now since A_i has received the message and since $\exists A_i \to A_k$, A_k has exhausted its search space too.

Properties 3.1, 3.3 and 3.4 ensure completeness of the exploration. They prove that according to the DisAO computed ordering, backtracking between agents is made in an exhaustive way.

Termination/Correctness

Termination is ensured by search exhaustivity and by the fact that DisAO order is acyclic. The use of a state detection algorithm [1] which stops the system when any context c is stuck on a solution gives correctness. Interestingly the use of several context within IDIBT do not significantly change the overhead brought by the

Algorithm 2: Interleaved Distributed Intelligent BackTracking

```
   begin
1      if ($\Gamma^- = \emptyset$) then Split domain $D$ in $D_{self,1}$ .. $D_{self,NC}$;
       for ($1 \le c \le NC$) do
           if ($\Gamma^- != \emptyset$) then $D_{self,c} \leftarrow D$;
2          $myValue_c \leftarrow$ getValue(info, c);
           $myCpt_c \leftarrow 1$;
           sendMsg($\Gamma^+$, "infoVal:$myValue_c$; from:self", c);
3          $end_c \leftarrow false$;
4      while ($\exists c | end_c = false$) do
5          $m \leftarrow$ getMsg();
6          $c \leftarrow m.context$;
           if ($m$ = stop) then $end_c \leftarrow true$;
7          if ($m$ = infoVal:$a$; from:$j$) then
               $value[j]_c \leftarrow a$;
               $valueCpt[j]_c ++$;
               $myValue_c \leftarrow$ getValue(info, c);
               if ($myValue_c$) then
                   sendMsg($\Gamma^+$, "infoVal:$myValue_c$; from:self", c);
               else
8                  sendMsg($first(\Gamma^-)$, "btSet:$\Gamma^-$; Values:$valueCpt[\Gamma^-]_c$", c);
9          if ($m$ = btSet:$set$; Values:$reportedValueCpt$) then
               if (contextConsistency(set, reportedValueCpt, c)) then
                   $myValue_c \leftarrow$ getValue(bt,c);
                   if ($myValue_c$) then
10                     sendMsg($\Gamma^+$, "infoVal:$myValue_c$; from:self", c);
                   else
11                     if ($\Gamma^- = \emptyset$ and $set = \emptyset$) then
                           sendMsg($system$, "noSolution", c);
                           $end_c \leftarrow true$;
                       else
                           $followSet \leftarrow merge(\Gamma^-, set)$;
12                         sendMsg($first(followSet)$, "btSet:$followSet$; Values:$valueCpt[\Gamma^-]_c \cup reportedValueCpt$", c)
   end
```

Chandy's method. In fact it is easy to generalize the method to manage the monitoring of the different context without raising the message passing overhead; i.e., each monitoring message includes the status of the different contexts.

Complexity

Search complexity is exponential in the number of variables. But in a distributed execution, rooms are open to use the relative independence between subproblems. This can enhance complexity results. In the following, $level_j$ represents the set of agents with a computed level j and h the highest level in the ordering.

DEFINITION 3.1 ()
A DisAO ordering is called *additive* if $\forall b \in level_j \mid 1 \leq j \leq h$, $\nexists$ agents $a, a' \in level_i$ with $1 < i \leq h \mid a \rightarrow b$ and $a' \rightarrow b$.

THEOREM 3.1 ()
A DCSP P with domain sizes d, using an additive DisAO ordering has a worst case time complexity,

$$O(\prod_{l=1}^{h} |level_l| \times d)$$

To prove that we must remark that with an additive ordering, during backtracking, the union of two Γ^- set do not include two agents at the same level. Then a backtracking occurs between distinct level and at each time considers at most d values. The whole problem is solved by considering at each level combinations of values. Since at each level, agents are independent, the number of possibilities is made by the sum of domains size.

When the ordering produced by DisAO is not additive, the complexity of a backtracking depends on the size of the longest path between agents. In the worst case we have an $O(d^n)$ complexity. We must remark here that DisAO was not made to construct additive hierarchies; its purpose was to add more parallelism by extracting subproblems independence. Nevertheless, we think that it must be possible to embolden parallelism while maximizing the additive property. This is futur work.

Remarks

- From distributed to parallel search: As we saw in section 2.4, parallel exploration allows simultaneous explorations of disjoint subspaces. This parallelism is achieved by duplication of processing units. Here we benefit from the asynchronism in the system. Since distinct agents can simultaneously consider and operate in different context, IDIBT realizes a parallel exploration of the search space too.

- Load balancing: When a particular context c detects insolubility, the search within it is canceled. In parallel search, the basic behavior when a subspace is exhausted is to rearrange the distribution of the work between processing units. This is normal since without such reallocation, some unit becomes idle while others are still working hard. Such load balancing process is automatic in IDIBT. In fact by doing nothing else that stopping current search in c, more cpu time and bandwidth are allocated for remaining subspaces. Of course, it is still possible to maintain NC explorations, the source agent has just to reallocate its subspaces.

- Ordered search: When DFS is used to find one solution, heuristics are useful to order the successors of a node. Within IDIBT, when the source agent splits its search space, he can use heuristics to allocate the best value as the first one in the first context, then the second best one as the first one in the second context, and so on. With this method, promising subspaces are more rapidly explored.

4 Experimentations

We made our experiments on a network of Linux workstations. The C++ algorithms use the MPI message passing library [8]. In all these experiments, each DCSP's variable was dedicated to a single computer.

4.1 Randoms problems

With randoms problems, we expected no speed-up upcoming from an exploration with several contexts. Nevertheless, these regular problems are helpful to evaluate and to define the overheads. We solved randoms problems with 15 variables, 8 values in each domain and connectivity between variables set to 30%. The tightness parameter has been changed from smallest to more important values, with a particular emphasis in the phase transition area. Each point in our experiments represents the median value took between 25 instances. Each instance was solved 5 times to limit the impact of message interleaving. That means 125 experiments for each point. Finally, each problem was solved with respectively 1, 2 and 4 context of resolution.

4.1.1 Time

Figure 4 presents median time results. With one context, IDIBT needs up to 1.69s. With two contexts, the time peak is 1.99s and with four it raises up to 2.85s. So as expected here, no speed-up came from the parallel exploration of these random problems. Nevertheless, this figure gives the shape of the overhead with several contexts. We can see that easy soluble instances are more difficult on several contexts than easy insoluble ones.

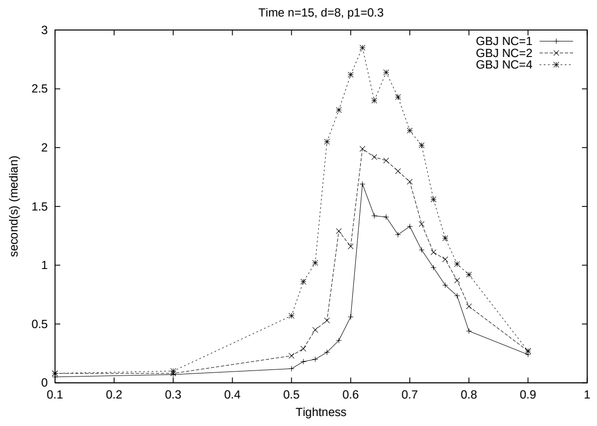

Figure 4. IDIBT, time (median)

Interestingly, the overhead of $NC = 2$ and $NC = 4$ is relatively small for problems located in the phase transition region. This result is confirmed by recent work on interleaved search [9].

4.1.2 Message passing and local operations

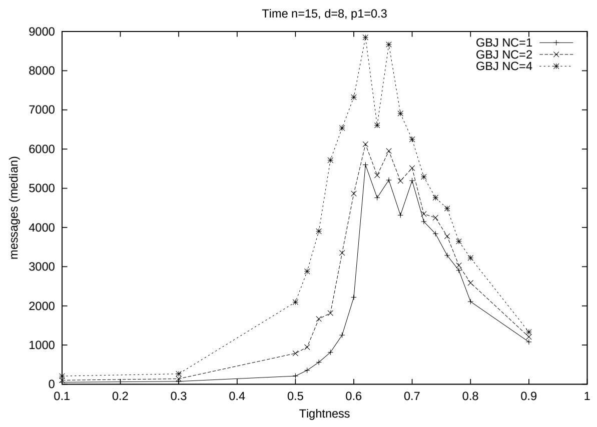

Figure 5. IDIBT, message passing (median)

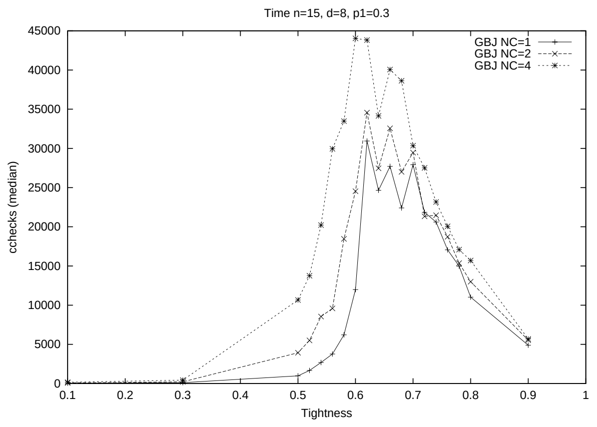

Figure 6. IDIBT, constraint checks (median)

Figures 5 and 6 present respectively the amount of messages and the amount of constraint checks performed in the system.

With one context, IDIBT uses up to 5600 messages, with two and four contexts it needs 6125 and 8851 messages. When we consider constraint checks, we have up to 30939, 34560 and 44032 respectively with one, two and four contexts. What we can say here is that explorations look very close according to these parameters. The previous observation about easy instances is reinforced here.

4.2 N-queens

In the second part of our experiments, we tried some instances of the classical N-queens problems. These problems have a nonuniform search space which authorize spectacular speed-up in parallel search [11]. For each instance we made 100 runs, the figure 7 presents the time results (log scale median). The number of context was set from one to eight.

We can see that 8-queens problem benefits from the multi-context running. With two context, the speed-up is the most important with 0.15s against 0.3s. The limit is reached with five context and 0.24s. With the 10-queens problem multi-context is useless. Resolution time is raising slowly with NC. Finally the 16-queens problem allows superlinear speed-up with two contexts (0.41s against 1.09s). Here, when NC is raising the behavior looks very chaotic. This was previously observed by [11].

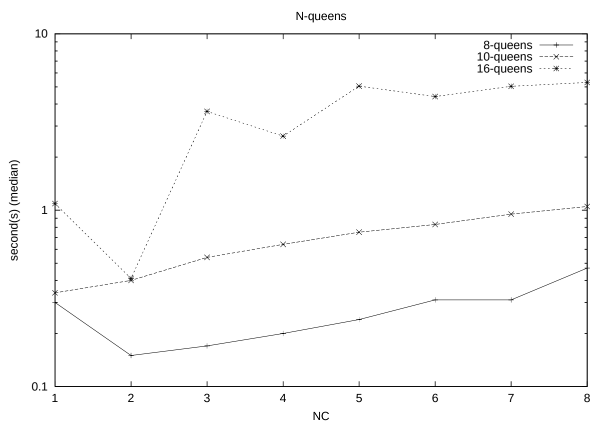

Figure 7. IDIBT, N-queens time (median)

5 Conclusion

We have presented IDIBT, as a generalization of DIBT a fully distributed asynchronous system for solving distributed CSPs. IDIBT can operate in heterogeneous networks where communications between agents are randomly delayed ($contextConsistency$ function). More importantly, IDIBT allows simultaneous exploration of disjoint search spaces.

From our experiments on random problems, we learned two things. The first one is that for easy soluble instances, the overhead of an exploration in several contexts is important. At contrary, when the instances are insoluble and easy, the relative overhead is limited compared to a classical exploration.

For problems with nonuniform search space, we saw as expected that our method can bring superlinear speed-up over DIBT. This result is important for applications where the accuracy of value ordering heuristics is limited and this, particularly at the beginning of a DFS exploration. With these distributed problems, IDIBT should be very effective.

The backjumping in the method uses the graph structure extended by DisAO to reach completeness. We prove both completeness and termination of IDIBT. Even if backjumping is systematic between related sub-problems, the relative independence between agents can be kept thanks to the DisAO ordering method. This backjumping can be easily extended to implement *conflict directed backjumping* [10].

Beyond the improvement of efficiency showed here, interleaving of context within search seems promising. In different context, several agents ordering could be used. This could be an answer to the difficulty of making efficient dynamic variable ordering in a distributed system. Each agent could implement cooperation between its local context by exchanging useful informations (instantiations, conflict-set, nogood, filtering [4],...). Here the principal drawback of cooperative frameworks (the cost of exchanging informations between processes) disappears since the exchanges occur within each agent. We are currently exploring this way.

Acknowledgements

The author wishes to thank the INRIA/HP icluster project for their computing facilities.

References

[1] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *TOCS*, 3(1):63–75, Feb 1985.

[2] R. Dechter. Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition. *AI*, 41(3):273–312, 1990.

[3] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 65.

[4] Y. Hamadi. Optimal distributed arc-consistency. In *Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, pages 219–233, 1999.

[5] Y. Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués*. PhD thesis, Université Montpellier II, 1999. (in french).

[6] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *ECAI*, pages 219–223, Aug 1998.

[7] W. Kornfeld. The use of parallelism to implement a heuristic search. In P. J. Hayes, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 575–580, Los Altos, CA, 24–28 Aug. 1981. William Kaufmann.

[8] M. P. I. F. MPIF. MPI: A message-passing interface standard. *Int. Journal of Supercomputer Applications*, 8(3/4), 1994.

[9] N. Prcovic and B. Neveu. Recherche à focalisation progressive. In *JNPC*, pages 191–204, 2000.

[10] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[11] V. N. Rao and V. Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):427–437, Apr 1993.

[12] M. Yokoo and K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In *ICMAS*, pages 401–408, Dec 1996.