Distributed, Interleaved, Parallel and Cooperative Search in Constraint Satisfaction Networks

Youssef Hamadi
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-21
March 8th , 2002*

distributed AI, constraint satisfaction, search

In this work, we extend the efficiency of distributed search in constraint satisfaction networks. Our method adds interleaving and parallelism into distributed backtrack search. Moreover, it has a filtering capacity that makes it open to cooperative work. Experimentations show that 1) the shape of phase transition with random problem can be characterized, 2) important speed-up can be achieved when the distribution of solutions is non uniform.

* Internal Accession Date Only
Approved for External Publication
© Copyright Hewlett-Packard Company 2002

Distributed, Interleaved, Parallel and Cooperative Search in Constraint Satisfaction Networks

Abstract

In this work, we extend the efficiency of distributed search in constraint satisfaction networks. Our method adds interleaving and parallelism into distributed backtrack search. Moreover, it has a filtering capacity that makes it open to cooperative work. Experimentations show that 1) the shape of phase transition with random problem can be characterized, 2) important speed-up can be achieved when the distribution of solutions is non uniform.

Keywords: Distributed AI, Constraint Satisfaction, Search

Introduction

In the distributed constraint satisfaction paradigm (DCSP) a problem is distributed between autonomous agents which are cooperating to compute a global solution. Agents are finding values for problem variables subject to constraints that are restrictions on which combinations of values are acceptable. Since the seminal work of (Yokoo, Ishida, & Kubawara 1990), the raise in application interoperability combined to the move towards decentralized decision process in complex systems raise the interest for distributed reasoning. Several distributed search method have been presented to tackle DCSP (Yokoo 1995; Armstrong & Durfee 1997). None of them ask the question of efficiency. In this work we show how to enhance the efficiency of distributed search. The basic method to search for solution in a constraint network is depth-first backtrack search (DFS) (Golomb & Baumert 65), which performs a systematic exploration of the search tree until it finds an assignment of values to variables that satisfies all the constraints. DFS has been extended to parallel-DFS to speed-up the resolution process (Rao & Kumar 1993). Parallelism was then the answer to the question of efficiency for sequential search. Interestingly parallel-DFS showed that under some assumptions, speed-up could be superlinear which means that sequential search is sub-optimal on some problems. Following these conclusions interleaved sequential DFS has been presented (Meseguer 1997). Here we give an answer to the problem of efficiency in any distributed search algorithm. Our method combines interleaved and parallel search for solution in DCSPs. The distributed backtrack search used acts like the well known sequential conflict-directed backjumping (CBJ) (Prosser 1993). Moreover, it takes advantage of the asynchronous relations between agents to perform a more refined and effective update of conflict-sets. Detected conflicts allows to some extent the removal of directed k-inconsistent values between agents. This filtering capacity is easily shared between all the interleaved subspaces of an agent which makes interleaved searches cooperating together to solve a problem. All the previous make search for solution in DCSPs more efficient. In the following, we first give a definition of the DCSP paradigm. We then present a distinction between distributed, parallel and interleaved search. Before presenting our distributed search procedure called IDIBT/CBJ-DkC, we give DisAO a distributed agent ordering method used during backtracking. Afterwards, we present a set of experimental results followed by a general conclusion.

Background

Distributed constraint satisfaction problems

A *distributed constraint network* $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$ involves a set of n *variables* $\mathcal{X} = \{X_1, \ldots, X_n\}$, a set of *domains* $\mathcal{D} = \{D_1, \ldots, D_n\}$ where D_i is the finite set of possible *values* for variable X_i, $\mathcal{C}$ the set of binary constraints $\{C_{ij}, \ldots\}$ 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 $(\mathcal{X}, \mathcal{D}, \mathcal{C})$. 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 our work we focus on the study and evaluation of inter-agents interactions. That is why 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 assignment 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

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

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

Communication model

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.

IDIBT: Distributed Interleaved and Parallel search

Parallel backtrack search is used to speed-up the resolution process (Rao & Kumar 1993; Kornfeld 1981). 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.

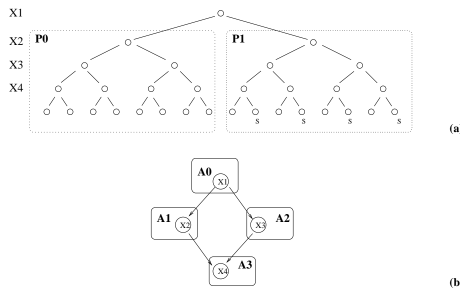

Figure 1: **Tree search:** (a) parallel search, (b) distributed search

Part a) of figure 1 presents a parallel exploration for a 4-variables CSP. 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 (Rao & Kumar 1993). Usually, an idle unit asks a busy one for a part of its remaining exploration task.

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

Interleaved sequential depth-first search (IDFS) tries to mimic a parallel exploration into a single process (Meseguer 1997). In its limited form, IDFS splits the top of a search tree into different subspaces and successively perform their exploration.

To combine interleaved and parallel search in our distributed framework, we divide a problem search space into independent subspaces. In each one a distributed conflict-directed 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.

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 subspaces. Parallelism is given by the observation that since agents are asynchronous, they can simultaneously operate in distinct subspaces.

All the previous makes the state space exploration of IDIBT/CBJ-DkC distributed, parallel and interleaved. Before closing this section, let us remark that sometimes distributed = parallel. In particular when the search space is flat i.e., no ordering used during the exploration of the subspaces. Distributed filtering is a good example of that (Hamadi 1999).

Conflicting agents in distributed search

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/CBJ-DkC search process.

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 between their subproblems. This ordering defines the hierarchical relation used during the distributed backtracking process.

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.

After that, agents know their *children* (Γ^+) and *parents* (Γ^-) acquaintances. During the search, they will send assignment value to children, and in case of dead-end, they

```
begin
      % Γ split;
1     Γ+ ← ∅; Γ− ← ∅;
      for each Agent_j ∈ Γ do
          if (f(Agent_j) op f(self)) then Γ+ ← Γ+ ∪ {Agent_j};
2         else Γ− ← Γ− ∪ {Agent_j};
      % Γ− ordering;
3     max ← 0;
      for (i = 0; i < |Γ+|; i++) do
          m ← getMsg();
          if (m = value:v; from:j) then
              if (max < v) then max ← v;
      max++;
      sendMsg(Γ−, "value:max; from:self");
      sendMsg(Γ+, "position:max; from:self");
      for (i = 0; i < |Γ−|; i++) do
          m ← getMsg();
          if (m = position:p; from:j) then Level[j] ← p;
      Order Γ− according to Level[] ;
4     Extend Γ− ;
end
```

Algorithm 1: **Distributed agent ordering**

will backtrack to the nearest conflicting agent in Γ^-. So, 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. 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).

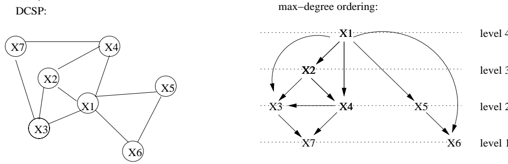

Figure 2: Distributed agents ordering

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. Once Algorithm 1 has been applied, the static variable ordering obtained is the one presented on the right side of Fig. 2.

Property 1 *$\forall A_i$, if $\exists A_j, A_k$ such that $A_j \rightarrow A_i$ and $A_k \rightarrow A_i$, then $\exists A_j \rightarrow A_k$ or $\exists A_k \rightarrow A_j$.*

We have $A_j \rightarrow A_1, A_i$ and $A_k \rightarrow A_2, A_i$ with $A_1 \in \Gamma^-(A_i)$ and $A_2 \in \Gamma^-(A_i)$. By definition we have $f(A_1) op f(A_2)$ or $f(A_2) op f(A_1)$ then by Γ^- extension we have $A_2 \rightarrow A_1$ or $A_1 \rightarrow A_2$. We can follow the previous reasoning by considering $A_i = A_1$ or $A_i = A_2$. The previous property expresses that the Γ^- extension step of DisAO break partial orders in the DisAO ordering. This will correlate forward and backtrack steps during distributed search.

IDIBT/CBJ-DkC: Using Conflicts Between Agents

Prosser's CBJ is directed by conflicts. This sequential algorithm stores with each variable i a "conflict-set" which keeps the subset of the past variables in conflict with some assignment of i. When a dead end occurs, CBJ jumps back to the deepest variable h in conflict with i. If a new dead end occurs, it jumps back to g, the deepest variable in conflict with neither h or i. Finally, each time CBJ jumps back from i to h, the variables j such that $h < j \leq i$ get back their search space and an empty conflict-set. In the following we detail the adaptation of this behavior in a distributed framework.

Principles Each agent will maintain a conflict-set which will be used during backjumping. But here, during a jump from i to a conflicting agent h our distributed framework can easily preserve previous work. While CBJ proactively reinitialize the search space and conflict-set of each variable j such that $h < j \leq i$, IDIBT/CBJ-DkC will do nothing. In fact, these updates are reduced and delayed as follow. When a value is addressed to i by an agent j, the receiver can selectively prune its conflict-set to keep conflicts unrelated with the new information from j. The elements of the local conflict-set are ordered ($<_o$) thanks to DisAO. These positions are computed from the bottom to the top. For each conflicting agent h, we have two configurations:

- $j <_o h$, the conflict with h is still valid. According to that the corresponding pruning in the local search space can be kept. Indeed, h is located in a position higher than the location of the sender. According to that, the new value for j cannot cancel previous decisions h.
- $h <_o j$, here the deletion of values raised by h's value is not independent from j's values. The local search space can recover these values and h is removed from the conflict-set.

This improvement makes IDIBT/CBJ-DkC close to backmarking (Gaschnig 77) and in a limited way close to dynamic backtracking (Ginsberg 1993). The previous behavior could be enhanced by cutting dependencies between variables during the detection of conflicts. Nevertheless, this behavior should be costly since the pruning made according to a particular acquaintance could not be used for successive tests.

Directed k-consistency and cooperating searches
CBJ has been extended to achieve directed k-consistency (Prosser 1993). We can do the same in the distributed framework. During backjumping, when an agent i addresses a conflict-set $\{h = v(h)\}$ to h, that mean that the value $v(h)$ is incompatible with any value from i. This value is arc-inconsistent and can be definitively removed from the domain of h. Here the asynchronism can make the backjump

obsolete; i.e., h has a new value. But this information is still useful and $v(h)$ must be removed[2].

To illustrate directed k-inconsistency, consider a configuration where an agent j exhausts its domain and constructs an ordered conflict-set $\{i = v(i), h = v(h)\}$. That mean that each value $v(j)$ is inconsistent with $v(h)$ or $v(i)$. Consider now that i has no conflict with its Γ^-. Now, assume that each new value addressed from i to j brings a new backjump to i. At the end, i exhausts its search space too and backjump to h which can definitively remove $v(h)$ from its domain.

Within IDIBT several DFS search are parallelized (see section). Each agent interleaves explorations and each k-inconsistency detected in a particular context can benefit to other contexts. This sharing makes the method cooperative. But while classical cooperative frameworks have to suffer from overheads to communicate useful informations (Hogg & Williams 1993), here the interleaving within each agent makes the sharing very efficient and easy.

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 initializes its conflict-set and assign its variable. Each timestamp counter $valueCpt_c$ is then set to one. After that, each agent informs 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. The function *updateSpace(j,c)* performs the conservative update of the local search space (see above). 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 conflicting agent (message content starting by "**btSet**" in line 8). This message includes the computed conflict-set and beliefs about the timestamps of its members $valueCpt[conflictSet_c]_c$. The receiver of the backtrack message (line 9) starts by considering the directed k-inconsistency detection in line 10. These detections are made without checking message validity. The full processing of backtracking demands then starts in line 11. The agent checks validity 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, this means that the sender and/or the receiver have not yet received some information. Backtrack decision could then be obsolete or badly interpreted. When the comparison matches, reported conflicts are merged with the local ones. The current value $myValue_c$ is added to the pruned values for the included agents; i.e., the corresponding pruning in the local search space. Then, if the agent can find a compatible $myValue_c$ in the remaining search space, this value is addressed to children in line 12. If such a value cannot be found, we must consider two cases. The first one is an agent without possibility for backtracking (empty conflict-set, line 13). 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 multi-agent system. With this information agents can stop the processing of context c messages[4]. 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 (Chandy & Lamport 1985) 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. If there exists a conflicting agent for backtracking, the agent addresses a backtrack message to the nearest agent in the augmented conflict-set (line 14).

Primitives and data structures

IDIBT/CBJ-DkC uses the following structures and methods:

NC is the number of contexts. 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. $conflictSet_c$, this set records the conflicting acquaintances in context c. $updateSpace(j, c)$, according to an *infoVal* message from j, implements in context c the selective update of the search space described above. $getValue(type, c)$, returns the first value in $D_{self,c}$ compatible with agents in Γ^-, starting at $myValue_c$. During this search, $conflictSet_c$ is updated. If a compatible value is found, $myCpt_c$ is incremented. 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. $remove(D, v)$ this function removes the value v, from the local domain D. It is used to achieve the removal of k-inconsistent values. $contextConsistency(set, reportedValueCpt, c)$, set contains an ordered list of agents, $reportedValueCpt$ 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

[2]Since IDIBT transmits timestamps instead of beliefs about assigned values, the sender adds the value which has to be deleted each time it address a singleton-conflict-set.

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

[4]And by the way give more bandwidth and CPU to remaining subspaces, i.e., implicit load-balancing. This behavior makes the search process more and more focused on promising branches.

$valueCpt[]_c$. 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.

```
    begin
1     if (Γ⁻ = ∅) then Split domain D in D_{self,1} .. D_{self,NC};
      for (1 ≤ c ≤ NC) do
          if (Γ⁻! = ∅) then D_{self,c} ← D;
          conflictSet_c ← ∅;
2         myValue_c ← getValue(info, c); myCpt_c ← 1;
          sendMsg(Γ⁺, "infoVal:myValue_c; from:self", c);
3         end_c ← false;
4     while (∃c|end_c = false) do
5         m ← getMsg();
6         c ← m.context;
          if (m = stop) then end_c ← true;
7         if (m = infoVal:a; from:j) then
              value[j]_c ← a; valueCpt[j]_c + +; updateSpace(j,c);
              myValue_c ← getValue(c);
              if (myValue_c) then
                  sendMsg(Γ⁺, "infoVal:myValue_c; from:self", c);
              else
8                 sendMsg(first(conflictSet_c),
                  "btSet:conflictSet_c;                    Val-
                  ues:valueCpt[conflictSet_c]_c", c);
9         if (m = btSet:set; Values:reportedValueCpt) then
10            if (|set| = 1) then remove(D_self, set(self));
11            if (contextConsistency(set, reportedValueCpt, c)) then
                  conflictSet_c ← merge(set, conflictSet_c);
                  myValue_c ← getValue(c);
                  if (myValue_c) then
12                    sendMsg(Γ⁺, "infoVal:myValue_c; from:self", c);
                  else
13                    if (conflictSet_c = ∅) then
                          sendMsg(system, "noSolution", c);
                          end_c ← true;
                      else
14                        sendMsg(first(conflictSet_c),
                          "btSet:conflictSet_c;            Val-
                          ues:valueCpt[conflictSet_c]_c", c)
    end
```

Algorithm 2: **IDIBT/CBJ-DkC**

Analysis

Property 2 *When an agent A_i changes its instantiation, agents A_j such that $\exists A_i \rightarrow A_j$ will reconsider their whole search space.*

When an agent changes its value, Γ^+ agents receive it. These agents can keep their current instantiation or change it. By using the *UpdateSpace* function, they filter their conflict-set and reinitialize some part of their local search space. By propagation of instantiations between agents, 2 is verified.

Property 3 *If A_i changes its instantiation according to a **btSet** message initially upcoming from A_j, each agent A_k (such that $A_k < A_i$ in the DisAO ordering) included in the conflict-set of agent A_j has exhausted its search space.*

According to backtrack chaining between related agents, this last property is obvious. Properties 1, 2 and 3 ensure completeness of the exploration. They prove that according to the DisAO computed ordering, backjumping between agents is made in an exhaustive way. Termination is ensured by the fact that DisAO orders are acyclic. The use of a state detection algorithm (Chandy & Lamport 1985) which stops the system when any context c is stuck on a solution gives correctness.

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

Definition 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 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 combinations is made by the sum of domains size. When the ordering produced by DisAO is not additive, the complexity of backtracking depends on the size of the longest path between agents. In the worst case we have an $O(d^n)$ complexity.

Experiments

We made our experiments on a cluster of linux workstations. The algorithms are made in C++ with the MPI message passing library. In all these experiments, each DCSP's variable was dedicated to a single computer.

Random problems

We solved randoms problems with 25 variables, 8 values in each domain and a 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 50 instances. Each instance was solved 3 times to limit the impact of message interleaving. That means 150 experiments for each point.

According to the search space regularity of these problems, we did not attempt any speed-up here. Nevertheless we can make some interesting observations. The interleaved/parallel overhead is more important for easy-solubles instances (at the left of the time peak) than for

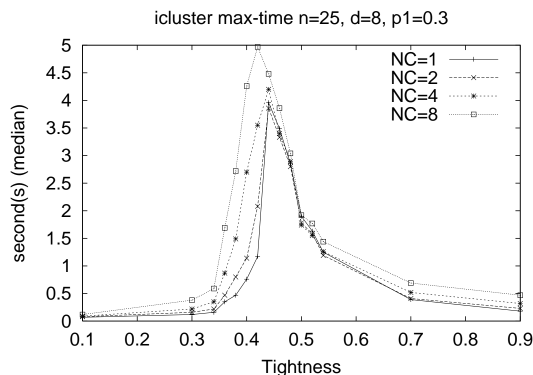

Figure 3: **IDIBT, 25 agents max-time (median)**

easy-insolubles ones (at the right of the peak). In the left part, small tightness and space regularity make classical distributed search (NC=1) able to easily found a solution, while the interleaved/parallel search just adds overheads. In the right part, insolubility is proved by checking the whole space. As we can see, this task is made with minimum overheads by the interleaved/parallel searches.

N-queens

The n-queens problem is well known for its non regular search space and this is very favorable to interleaved/parallel methods. For each instance the number of interleaved subspaces *NC* was set to 1, 2, 4 and 8.

	16-queens			*20-queens*		
NC	*time*	*msg*	*cchecks*	*time*	*msg*	*cchecks*
1	0.84s	4203	41167	13.17s	90114	1.4e6
2	**0.55s**	3171	24365	**4.46s**	29953	421031
4	1.62s	10256	96899	92.93s	176062	2.5e6
8	3.37s	22231	210801	-	-	-

Table 1: N-queens

Table 1 gives time, message passing and constraint checks results for the 16 and 20-queens problems. When the system uses two subspaces, speed-up are the most interesting (superlinear with 20 queens). The efficiency of parallel search is computed by dividing the speed-up by the number of added computing resources. Here we do not add more processing unit. Speed-up is then equal to efficiency (close to 3 with 20 queens). Such efficiency can be explained as follow. The first instantiation in the second subspace locates the first queen in the middle of the row. Thanks to this choice remaining agents are more constrained (this choice propagates on 2 diagonals) and global solution is more rapidly found. Such behavior reproduces in some way the now classical value ordering "middle" known for its performances with this problem.

Conclusion

Our algorithm searches for solution in distributed constraint satisfaction networks. It combines distributed, interleaved and parallel exploration. During exploration, it uses conflicts for backtracking and directed k-inconsistencies detection. This filtering has extended our method toward cooperation. While classical cooperative framework suffer from overheads and have to deal with questions like: when do we share an information? when do we stop cooperation? In our approach, the interleaving makes everything transparent, cooperation and even load balancing. Two mains conclusion were drawn from the experiments. First, we have characterized the shape of phase transition with random problems. Second, non regular search spaces proved the drastic efficiency of our combination over a single distributed search. This last point makes our method well suited for problems with poor heuristic function at the top of an agent hierarchy. Our algorithm is the first answer to the question of efficiency in distributed search. Its performances make it well suited for current and future distributed reasoning applications.

References

Armstrong, A., and Durfee, E. 1997. Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In *IJCAI*, 620–625.

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

Gaschnig, J. 77. A general backtracking algorithm that eliminates most redundant tests. In *IJCAI*, volume 1, 457.

Ginsberg, M. L. 1993. Dynamic backtracking. *JAIR* 1:25–46.

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

Hamadi, Y. 1999. Optimal distributed arc-consistency. In *CP*, 219–233.

Hogg, T., and Williams, C. P. 1993. Solving the really hard problems with cooperative search. In *AAAI*, 231–236.

Kornfeld, W. 1981. The use of parallelism to implement a heuristic search. In *IJCAI*, 575–580.

Meseguer, P. 1997. Interleaved depth-first search. In *IJCAI*, 1382–1387.

Prosser, P. 1993. Domain filtering can degrade intelligent backjumping search. In *IJCAI*, 262–267.

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

Yokoo, M.; Ishida, T.; and Kubawara, K. 1990. Distributed constraint satisfaction for DAI problems. In *Proc. of the 10th Int. Workshop on DAI.* chapter 9.

Yokoo, M. 1995. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *CP*, 88–102.