Optimal Distributed Arc-Consistency

Youssef Hamadi

LIRMM UMR 5506 CNRS-UMII
161, Rue Ada, 34392 Montpellier Cedex 5, France
hamadi@lirmm.fr

Abstract. This paper presents *DisAC-9*, the first optimal distributed algorithm performing the arc-consistency of a constraint network. Our method is optimal according to the number of message passing operations. This algorithm can firstly, give speedup over the fastest central arc-consistency algorithms. Secondly, achieve the fast processing of distributed constraint satisfaction problems (DCSP). Experimental results use classical benchmarks and large hard randoms problems. These results allow us to give the first characterization of the hardest instances of this distributed computation.

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 one of d values for n variables subject to constraints that are restrictions on which combinations of values are acceptable. It is widely used in artificial intelligence (AI), its applications ranging from machine vision to crew scheduling and many other fields (see [1] for a survey). Since it is an NP-complete task, many filtering techniques have been designed. These methods reduce the problem search space by pruning the set of possible values. Owing to their interesting complexities, arc-consistency algorithms which detect and eliminate inconsistencies involving all pairs of variables are widely used [2].

In the last years, the distributed constraint satisfaction (DCSP) paradigm emerges from the necessity of solving distributed AI problems. These distributed problems are more and more crucial because of the wide distribution of informations allowed by Internet facilities. So several authors tried to bring classical CSP procedures to the distributed framework [3, 4].

More relevant works on distributed arc-filtering were brought by P. R. Cooper and M. J. Swain who proposed in [5] two massively parallel (nd processes) versions of AC-4 [6]. More recently, T. Nguyen and Y. Deville [7] distributed AC-4 in a more realistic framework since their granularity varies from 1 to n process(es).

Nevertheless all these works do not address the real difficulty of distributed systems which is the complexity of message passing operations. S. Kasif who has studied the complexity of parallel relaxation [8] concludes that discrete relaxation is inherently a sequential process. This means that due to dependencies between

deletions, a distributed/parallel algorithm is likely to perform a lot of messages since it is likely to perform in a sequential fashion.

We have learned from all the previous works and focused our work on message passing reduction. Our DisAC-9 algorithm is optimal in the number of message passing operations. This major improvement over the previous works was possible by the exploitation of the bidirectionality property of constraint relations which allows agents to induce acquaintances deletions. Since bidirectionality is a general property of constraints, DisAC-9 is a general purpose algorithm. Worst time complexity is $O(n^2d^3)$ which allows us to reach optimality in message operations, nd. According to that, our method can achieve the fast processing of DCSPs and even give major speed-up in the processing of large CSPs.

The paper is organized as follows. We first give background about constraints, arc-consistency and message passing in section 2. Section 3 presents previous works. Then we present the centralized algorithm AC-6. Next we introduce our distributed algorithm, DisAC-9. Theoretical analysis follows in section 6. Before concluding, we present experimental results in section 7.

2 Definitions

2.1 Constraint and arc-consistency background

A *binary CSP*, $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ involves, $\mathcal{X}$ a set of n variables $\{i, j, ...\}$, $\mathcal{D} = \{D_i, D_j, \ldots\}$ where each element is a variable's domain, $\mathcal{C}$ 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. A *solution* to a constraint satisfaction problem is an instantiation of the variables in a way that all the constraints are satisfied. A constraint network (CN) $\mathcal{G} = (\mathcal{X}, \mathcal{C})$ can be associated to any binary CSP.

Each element of a domain represents a *value*. A *label* (i, a) is the instantiation of the value a to the variable i. It is supported by a value b along C_{ij} *iff* $C_{ij}(a, b) = true$, b is called a *support* for a along C_{ij}. The value a is also a support for b along C_{ji}. A value a from D_i is *viable iff* $\forall C_{ij}$, there exists a support b for a in D_j. The domain $\mathcal{D}$ of a constraint network is *arc-consistent* for this CN if for every variable i, all values in D_i are viable. The maximal arc-consistent domain $Dmax\text{-}AC$ of a constraint network is defined as the union of all domains included in $\mathcal{D}$ and arc-consistent for this CN. This maximal domain is also *arc-consistent* and is computed by an arc-consistency algorithm.

2.2 Distributed constraint satisfaction background

A *binary distributed CSP*, $P = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A})$ involves a binary CSP, $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ partitioned among $\mathcal{A}$, a set of p autonomous agents ($1 \leq p \leq n$). Each agent owns the domains/constraints on its variables. An agent that owns variable i owns D_i, and the binary constraints C_{ij}.

We assume the following communication model [3]. Agents communicate by sending messages. An agent can send messages to other agents *iff* he knows their

addresses in the network. For the transmission between any pair of agents, messages are received in the order in which they were sent. The delay in delivering messages is finite.

For readers unfamiliar with exchanges in distributed systems, we present in table 1 the real data transmission costs for some nowadays parallel computers [9]. In common local area networks (LAN), these costs are orders of magnitudes largers. These values are computed in an empty communication network, this means that these are lower bounds values. In real systems, the more you send messages, the more it takes time to achieve transmissions.

Table 1. The real cost of message passing operations

	iPSC	nCUBE/10	iPSC/2	nCUBE/2	iPSC/860	CM-5
T_s [μs/msg]	4100	400	700	160	160	86
T_b [μs/byte]	2.8	2.6	0.36	0.45	0.36	0.12
T_{fp} [μs/flop]	25	8.3	3.4	0.50	0.033	0.33

The line T_s represents the *start-up* cost of a message, T_b is the per/byte cost. We give as a reference point, T_{fp} which is the cost of one floating point operation on the machines. Each message reception brings to a cost, generally this cost is similar to T_s. So the estimated cost for a message m is, $\geq 2 * T_s + |m| * T_b$.

3 Previous works

We emphasize previous works performing distributed arc-filtering here. We do not present works which use the PRAM machine model like [10] since this model is not representative of current computing facilities which all use the message passing paradigm. We also do not present works which are not based on optimal central method [11]. In table 2 we summarize the headlines of 3 methods. These methods propose a distributed implementation of the AC-4 sequential algorithm. The *#ps* column gives the granularities, *first step* gives the complexity of the first step of the methods since these are two step procedures like the central AC-4. Then, *complexity* gives the methods full complexities and *#msg* the number of message passing operations required by a worst case execution.

Informally a distributed algorithm is *local*, if its messages transmissions are limited i.e. its transmissions do not involve the whole processes. This definition of locality is sufficient for our use. For a comprehensive work on locality in distributed systems, see [12]. The column *local* specifies if a method is local.

AC Chip This algorithm presented in [5] defines a VLSI circuit. It uses a very fine granularity and it does not use explicit message passing operations. Both allows a pretty good complexity. Nevertheless it would need $O(n^2d^2)$ initializing steps for loading the network in the digital chip.

Table 2. Distributed arc-consistency algorithms

Algo.	#ps	first step	complexity	#msg	local
AC Chip	nd	nd	$O(nd)$	-	-
ACP	nd	nd	$O(nd \log_2(nd))$	n^2d^2	no
DisAC-4	$1 \leq p \leq n$	$(n^2d^2)/p$	$O((n^2d^2)/p)$	nd	no

ACP This is the software version of the previous algorithm [5]. It uses nd synchronous processes which communicate by messages operations. Each message is sent with $O(\log_2(nd))$ steps and is received by the whole processes (excepted the sender). The method is not local.

DisAC-4 This coarse grain method uses p processes [7] sharing an Ethernet network. This allows "efficient" broadcasting of messages between processes. Nevertheless, this brings to an underlying synchronism since "collision networks" cannot carry more than one message in a given time. If we consider a more realist framework for the method, each broadcast requires p messages which brings to n^2d messages.

4 The arc-consistency algorithm AC-6

We briefly describe here Bessière's AC-6 algorithm [13]. This method introduces a total ordering between values in each domain, it computes one support (the first one) for each label (i, a) on each constraint C_{ij}. That to make sure of the current viability of (i, a). When (j, b) is found as the smallest support of (i, a) on C_{ij}, (i, a) is added to S_{jb} the list of values currently having (j, b) as smallest support. Then, if (j, b) is deleted, AC-6 looks for a new support in D_j for all values in S_{jb}. During this search, the method only checks values greater than b. The algorithm incrementally computes support for each label on each constraint relation. On average cases, it performs less checks than AC-4 whose the initialization step globally computes and stores the whole support relations.

In the following, we adapt this incremental behavior to our distributed framework, moreover we drastically improve efficiency by reaching optimality in the messages transmissions.

5 The distributed arc-consistency algorithm DisAC-9

The key steps of this new distributed algorithm are the following. According to local knowledge, each agent that owns a set of variables starts computing inconsistent labels. After this computation, *selected labels* are transmitted. This means that only the deleted labels that induce new deletions are sent to an acquaintance. After this initializing step, each agent starts the processing of incoming messages. These messages will modify its local knowledge about related

acquaintances. This will allow it to update support information for its labels. Naturally, this can lead to new deletions and maybe to new outcoming messages. Since communications occurs between acquaintances agents which are defined by agents that share binary constraints, DisAC-9 is local (see section 3).

5.1 Minimal message passing

Constraints bidirectionality allows agents to partially share acquaintances knowledge: since $C_{ij}(a,b) = C_{ji}(b,a)$, an agent owning i but not owning j knows any C_{ji} relations. This knowledge can be used to infere over acquaintances deletions.

More generally, *agent-1* currently deleting (i,a) computes for any acquaintance *agent-2*, labels (j,b) having (i,a) as support. If another local value (i,a') can support (j,b), it is useless to inform *agent-2* for (i,a) deletion. Otherwise, *agent-2* must delete (j,b) as an outcome of the deletion of (i,a). It must be informed of the local deletion.

5.2 A complete example

Figure 1 illustrates this on a particular DCSP, in this problem, two agents share a constraint C_{12}. The constraint relation is represented in the left part of the figure by the micro-structure graph. In this graph, allowed pairs of values are linked. In the right part, we represent the support relation computed by the two agents. An arrow goes from value a to value b if b supports a.

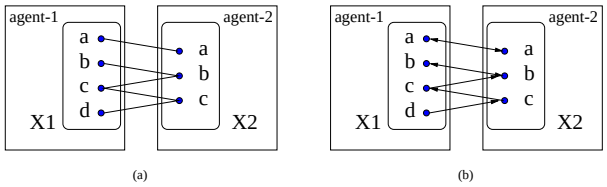

Fig. 1. Message minimizing in a DCSP

For *agent-1*, support lists are the following, $S_{2a} = \{(1,a)\}$, $S_{2b} = \{(1,b),(1,c)\}$ and $S_{2c} = \{(1,d)\}$. The second agent, *agent-2* computes $S_{1a} = \{(2,a)\}$, $S_{1b} = \{(2,b)\}$, $S_{1c} = \{(2,c)\}$ and $S_{1d} = \{\}$.

An extra data structure is used during the message minimization process. $first_j[i][a] = b$ tracks that according to C_{ij}, (j,b) is the first label supporting (i,a) in D_j. The $first$ table of agent-1 contains the following values, $first_2[1][a] = a$, $first_2[1][b] = b$, $first_2[1][c] = b$, $first_2[1][d] = c$. For agent-2, $first_1[2][a] = a$, $first_1[2][b] = b$ and $first_1[2][c] = c$.

Assuming the loss of $(1, c)$ (according to some incoming event) let us consider the information of *agent-2* about this deletion. We can just consider in D_2 values greater or equal to $first_2[1][c]$, the first support for $(1, c)$. Among these values, we must check the ones possibly supported by $(1, c)$. We find $(2, b)$ and $(2, c)$. For $(2, b)$ we determine $(1, b)$ as possible support and for $(2, c)$ we find $(1, d)$. According to these local computations, *agent-1* states that any message about $(1, c)$ deletion is useless. Such a message could update *agent-2* knowledge but it cannot lead to new domain reductions.

Now assuming the loss of $(1, b)$, we detect that $(2, b)$ must be deleted by *agent-2* since there are no more supports in D_1. Hence, we must send deletion information to *agent-2*. But if we only inform for $(1, b)$ deletion, *agent-2* by considering S_{1b} will compute $(1, c)$ as new support for $(2, b)$ since it is always assuming $(1, c)$ as a viable value. One solution is to inform for all previous deletions. In this example, *agent-1* informs in the same message for deletion of $(1, b)$ and $(1, c)$. As a result *agent-2* will delete $(2, b)$ but it will not inform for this deletion *agent-1* since the agent knows that $(1, b)$ was deleted. It will remove $(2, c)$ from S_{1c} and add it to S_{1d}.

This example shows that by performing more local computations, an agent can avoid outcoming messages. This exchange between local work and message transmissions is quite interesting in a distributed environment (see table 1).

5.3 Algorithm

After this illustration, we present the related algorithm. Our distributed system contains $1 \leq p \leq n$ autonomous agents (cf. 2.2) (plus an extra agent performing termination detection see section 6).

The knowledge of agent k $(1 \leq k \leq p)$ is represented/handled by the following data structures/primitives.

Data structures

- *Acquaintances*, agents sharing constraints with k are referenced in this set
- *localVar*, this set stores the variables for k
- $localD_i$ this set stores the current domain of i $(\forall i \in localVar)$
- $localM_i$ boolean state vector, keeps tracks of deleted values from the initial domain D_i $(\forall i \in localVar)$
- *linkedVar*, this set stores the non local variables which are linked to local variables by a constraint relation
  linkedVar=$\{i \in \mathcal{X}$, tq $i \notin localVar$ and $\exists j \in localVar$ tq $C_{ij} \in \mathcal{C}\}$
- $linkedM_i$ boolean state vector, keeps track of external variables deleted labels, $(\forall i \in linkedVar)$
- A set of lists, $S_{jb} = \{(i, a) | a \in D_i$ and b is the smallest value in D_j supporting $(i, a)\}$
- *first* an integer table, $first_j[i][a] = b$ tracks that according to C_{ij}, (j, b) is the first label supporting (i, a)

- *localList*, *extList*, these lists store the deleted labels, the second one is used for external propagation
- *sendSet*, vector of sets, stores outcoming deletions informations
- *inform*, boolean table, $inform_{acc}[i]$ means that *acc* has been selected for $localD_i$ transmission

Constant time primitives

- *affected(j)* returns the acquaintances that owns the variable j. *owns(A,i)* returns *true* if agent A owns the variable i
- *higher(D)* returns the last element of the domain D. if $D = \emptyset$ returns -1. *next(a,M)*, M is a state vector, returns the smallest indice ind, greater than a | M[ind]=true
- *get(S)* returns the first element from S. *addTo(S,a)* inserts a in S. *emptyP(S)* returns *true* if S is empty
- *localDUpdate(i,a)* this primitive realizes severals important updates. It removes the value a from $localD_i$ then it stores this deletion in $localM_i$ ($localM_i[a] \leftarrow false$). After performing these updates, the method checks the size of $localD_i$. If ($localD_i = \emptyset$), a *stop* message is broadcasted to the whole processes and termination occurs with the detection of problem inconsistency

Message passing primitives

- *getMsg()* blocking instruction, returns the first incoming message
- *broadcast(m, P)* the message m is broadcasted to processes in the set P, it uses $O(log_2(|P| + 1)$ operations

Each agent starts with a call to the *main* procedure (see algo. 1). This procedure has two steps. In the first one, agent's knowledge allows the filtering of local domains. For each local variable i, each value a in $localD_i$ is checked for viability. To achieve this test, agent looks for support for each related constraint C_{ij}. The *nextSupport* procedure (see algo. 3) looks for the first support according to C_{ij}. If this procedure returns with *emptySupport=false*, then b is the first support for (i, a) in D_j. We can add (i, a) to local support list S_{jb} and update *first* for later use (lines 9 and 10). Otherwise, (i, a) is non-viable. A call to *localDUpdate* (line 6) removes this label and test for domain wipe-out. Then the deletion is stored in both *localList* and *extList* (lines 7,8).

At the end of this initialization step, the *main* procedure processes deletions by calling *processLists* (see line 11). This procedure first, uses *localList* for the local propagation of deletions. This is done by calling the *deletionProcessing* procedure (see algo. 4) for each deleted label (j, e). This algorithm, removes labels (i, a) from S_{je}, then for each viable one ($a \in localD_i$), a call to *nextSupport* looks for a new support (j, c) greater than (j, e). If such a new support is found, (i, a) is added to S_{jc}. Otherwise, the procedure updates $localD_i$ and stores (i, a) in both *localList* and *extList* (lines 2,3,4).

Algorithm 1: DisAC-9 main

```
    begin
        localList ← ∅; extList ← ∅
        for each i ∈ localVar do
1           localD_i ← D_i
2           for each a ∈ D_i do localM_i[a] ← true; S_ia ← ∅
        for each j ∈ linkedVar do
3           for each a ∈ D_j do linkedM_j[a] ← true; S_ja ← ∅
4           for each i ∈ localVar do
                for each a ∈ D_i do first_j[i][a] ← 0
        %
        % first step, init. supports
        for each arc (i, j) | i ∈ localVar do
            for each a ∈ localD_i do
                b ← 0
5               nextSupport(j,i,a,b,emptySupport)
                if emptySupport then
6                   localDUpdate (i, a)
7                   addTo (localList, (i, a))
8                   addTo (extList, (i, a))
                else
9                   addTo (S_jb, (i, a))
10                  if j ∈ linkedVar then first_j[i][a] ← b
        % internal and external propagations
11      processLists (localList, extList)
        %
        % second step, interactions
        termination ← false
        while !termination do
12          m ← getMsg()
            switch m do
13              case stop
                    termination ← true
14              case deletedLabels:set
                    while !emptyP(set) do
15                      (j, M_j) ← get(set)
                        for each a ∈ D_j do
16                          if linkedM_j[a] ≠ M_j[a] then
17                              linkedM_j[a] ← false
18                              addTo (localList, (j, a))
19                  processLists (localList, extList)
    end
```

After these local propagations, *processLists* considers each locally deleted label from *extList*. For each label (j, e), a call to *selectiveSend* allows the agent to find which acquaintances to inform.

The *selectiveSend* procedure implements the previously detailed minimizing. While considering the deleted label (i, a), it finds for each non local variable j such that C_{ij} exists, the acquaintance *acc* that owns j (see line 2). Then, if *acc* has not be previously selected for $localD_i$ transmission ($inform_{acc}[i] = false$), it considers labels (j, b) supported by (i, a). Otherwise it is useless to check $localD_i$ transmission to *acc*.

We start the exploration of these labels by checking the current support of (i, a) ($first_j[i][a]$) since it is useless to consider previous labels (line 4). Two

Algorithm 2: DisAC-9 processLists

```
    Algorithm: processLists(in-out: localList, extList)
    begin
1       for each j ∈ Acquaintances do
            for each i ∈ localVar | ∃k ∈ linkedVar and owns(j,k) and C_ik do
2               inform_j[i] ← false
3       while !emptyP(localList) do
            (j, e) ← get(localList)
4           deletionProcessing (j,e)
        while !emptyP(extList) do
5           (j, e) ← get(extList)
6           selectiveSend (j, e, sendSet)
7       for each j ∈ Acquaintances do
            if sendSet_j ≠ ∅ then broadcast (j, deletedLabels : sendSet_j)
    end
```

Algorithm 3: DisAC-9 nextSupport

```
    Algorithm: nextSupport(in: j,i,a; in-out: b; out: emptySupport)
    begin
1       if b ≤ higher(D_j) then
2           if j ∈ localVar then stateVector ← localM_j
            else stateVector ← linkedM_j
3           while !stateVector[b] do b++
            emptySupport ← false
4           while !emptySupport and !C_ij(a, b) do
                if b < higher(D_j) then b ← next(b, stateVector)
                else emptySupport ← true
        else emptySupport ← true
    end
```

boolean are used, $endD_j$ is used to stop checking against j values, *selected* is set to stop checking against *acc*.

Among j values, we can restrict to viable ones ($linkedM_j[b]$=*true*) (line 6). Then if $C_{ij}(a, b)$ holds *true*, the procedure looks for another possible support (i, a'). If there is no (i, a') label such that $C_{ij}(a', b)$, *acc* is selected for message transmission (line 9). We add the pair $(i, localM_i)$ in $sendSet_{acc}$. This inclusion of the current domain of i allows the transmission of previously non informed labels deletions. Of course this can lead to long messages but the sizes of messages are not prohibitives comparing to their start-up costs (see table 1).

After these computations, *processLists* respectively addresses its *sendSet* set to each acquaintance (line 7).

The second step of the *main* procedure allows agents to revise their knowledge according to incoming messages (line 12). There are two kinds. The *stop* messages which stop the processing. These messages can result from both a domain wipe-out (see *localDUpdate* primitive) or can be addressed by an extra agent, called *System*. This agent is in charge of global termination detection. In our system, we use a global state detection algorithm [14]. So global termination occurs with a domain wipe-out or when the p agents are waiting for a message and when no message transits in the communication network.

Algorithm 4: DisAC-9 deletionProcessing

```
    Algorithm: deletionProcessing(in: j,e; in-out: localList,extList)
    begin
        while !emptyP(S_je) do
            (i, a) ← get(S_je)
1           if a ∈ localD_i then
                c ← e + 1
                nextSupport(j, i, a, c, emptySupport)
                if emptySupport then
2                   localDUpdate (i, a)
3                   addTo (localList, (i, a))
4                   addTo (extList, (i, a))
                else
                    addTo (S_jc, (i, a))
5                   if j ∈ linkedVar then first_j[i][a] ← c
    end
```

Algorithm 5: DisAC-9 selectiveSend

```
    Algorithm: selectiveSend(in : (i, a); in-out : sendSet)
    begin
1       for each arc (i, j) | j ∈ linkedVar do
2           acc ← affected(j)
3           if !inform_acc[i] then
4               b ← first_j[i][a]
                endD_j ← false
                selected ← false
5               while !selected and !endD_j and b ≤ higher(D_j) do
6                   if linkedM_j[b] and C_ij(a, b) then
                        endD_i ← false
                        otherSupport ← false
                        a' ← 0
7                       while !otherSupport and !endD_i and a' ≤ higher(localD_i) do
8                           if localM_i[a'] and C_ij(a', b) then otherSupport ← true
                            else
                                if a' < higher(localD_i) then a' ← next(a', localD_i)
                                else endD_i ← true
9                       if !otherSupport then selected ← true
10                  if b < higher(D_j) then b ← next(b, D_j)
                    else endD_j ← true
11              if selected then
12                  addTo (sendSet_acc, (i, localM_i))
13                  inform_acc[i] ← true
    end
```

The processing of incoming deletions messages (*deletedLabels:set*) starts with line 14. Remember that agents receive a set of pairs $(j, localD_j)$. They temporary store these pairs in (j, M_j). Then by comparing their local beliefs stored in $linkedM_j$ and the new reported situation they can detect new deletions (line 16). Agents store these new deletions and add them in their own *localList*. After considering the whole incoming *set*, a call to *processLists* allows the local propagation of externals deletions.

6 Analysis

6.1 Complexities

The worst time execution of a distributed algorithm occurs when it proceeds with a sequential behavior. With our DisAC-9, this occurs when only one value is deleted at a time. This leads to nd successive deletions between $p = n$ processes. In $O(nd^2)$ our *selectiveSend* procedure allows the detection of the unique outcome of each deletion. While receiving a deletion message each agent updates $linkedM[j]$ in $O(d)$, then *processLists* leads to one call to *deletionProcessing*. The S_{je} list contains exactly one label and *nextSupport* uses $O(d)$ steps in search of a new support. So each deletion conducts to $O(nd^2 + d)$ constant time operations. Since there are $O(nd)$ successive deletions, the overall time complexity is $O(n^2d^3)$ with exactly nd point-to-point messages.

With n variables, each one using d values, each agent data structures use $O(nd)$ space. Since there are n agents in the worst case, the total amount of space is $O(n^2d)$.

6.2 Termination

Termination is detected by the System agent, this agent performs a classical global detection state algorithm [14]. With our system it detects a situation where each agent is waiting for an incoming message and no message transit between agents. Domain wipe-out is another case of algorithm termination (see *localDUpdate*).

6.3 Correctness and message passing optimality

We present here the main steps of complete proofs (see [15] for proofs). For correctness we must ensure at the termination. (i) $\forall a \in localD_i, (i, a) \in$ *Dmax-AC*, this can be achieved by considering the deletions made by the algorithm.
(ii) $\forall (i, a) \in$ *Dmax-AC*, $a \in localD_i$, suppose that a current deletion (j, b) is in *Dmax-AC*, then, an inductive proof which suppose that all the previous deletions are out of *Dmax-AC*, gives (j, b) out of *Dmax-AC*.

For optimality, we must show that
(i) each *deletedLabels* message induces at least a label deletion for the receiver. This is exactly what is checked by the *selectiveSend* procedure.
(ii) each label suppression made during the interactions step is the outcome of an incoming *deletedLabels* message. We must observe that during the second step, deletions are made by *deletionProcessing* which is called by *processLists* which is always called for the processing of a *deletedLabels* message.

7 Experimentations

We have made ours experiments on an *IBM sp2* supercomputer[1] (*C++* language and *MPI* message passing library). We have used two algorithms against DisAC-9. First the sequential AC-6 to assess distribution benefits. Then to strictly evaluate our message minimizing we have built a modified version of our algorithm called DisAC-6 which does not use our minimizing method:

- each deleted label (i, a) is selected for diffusion to *acc* if this agent owns a variable j such that C_{ij} exists
- *sendSet* are composed of (j, a) labels so reception of deletions messages is straight forward ($linkedM[j][a] \leftarrow false$ and *localList* inclusion).

This simplified implementation has a worst time complexity $O(nd(n + d))$ requiring n^2d messages. Since all deleted labels are transmitted to acquaintances, it also allows comparisons with previous works (see section 3). Nevertheless, this new method is local since transmissions occur with respect to the CN. According to that, we think that DisAC-6 should outperform previous works.

7.1 Hard random problems

We have generated randoms problems with 90 variables, domains sizes set to 20, and connectivity p_1 set to 0.2, 0.5 and 0.8. Tightness p_2 varies from 0.1 to 0.9. For the distributed methods we set *p=90* which is the finest granularity in our framework since each agent owns one variable and *p=15* (each agent owns 6 variables). For each connectivity/tightness pair we have used 100 randoms instances.

We detail results with $p_1 = 0.5$. Figure 2 presents median time in seconds for the three algorithms. These results show the efficiency of DisAC-9[2]. With 90 agents, our procedure complexity peak is 1.06 seconds, while DisAC-6 needs up to 22.23 seconds.

Figure 3, presents the amount of *deletedLabels* messages processed in the whole system. DisAC-9 process up to 2029 messages while the second method process up to 32493 messages. Interestingly we can observe that in DisAC-9 messages transmissions start at $p_2 = 0.5$. Of course this minimizing has a cost in constraint checks (see figure 3), but it is limited to 4.41 times.

AC-6 time peak is 0.6 second while DisAC-9 with 15 agents uses up to 0.17 second. For $p2 \in \{0.71, 0.72, 0.73\}$, DisAC-9 with 90 agents is faster than AC-6.

Figure 2 allows some surprising new results. The time peaks of the methods are not located at the same tightness value. For the sequential AC-6, it occurs for $p_2 = 0.71$, for the methods using 90 agents, the time peak is located at $p_2 = 0.69$ while with 15 agents, the peak occurs at $p_2 = 0.7$. Our data results show 0%,

[1] Message passing latency is $35\mu s$, bandwidth is *90Mo/s*, each CPU is a *PowerPC* $120Mhz$.

[2] We can remark that experimentations on Ethernet networks give much larger speed-up since communications are slower than on the sp2.

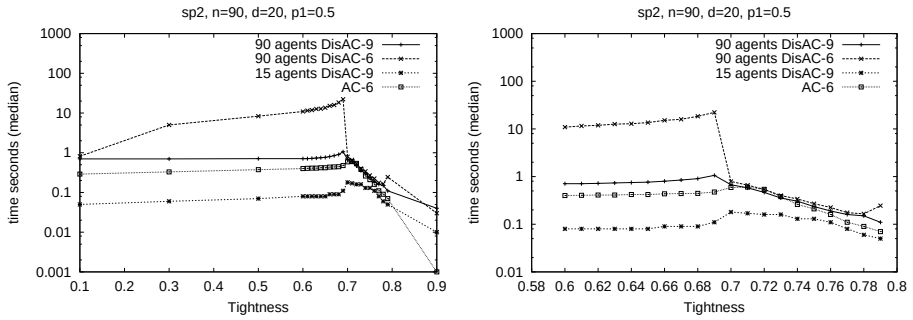

Fig. 2. Median time in seconds (the right part is a zoomed view)

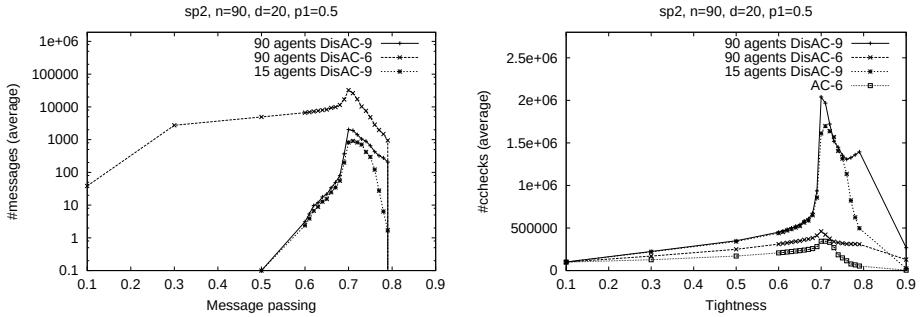

Fig. 3. Message passing (left) and constraints checks (right)

11%, 88% and 100% of inconsistent instances respectively at $p_2 = 0.68$, 0.69, 0.7 and 0.71.

To give a general explanation of this phenomenon, let us consider AC-6. It checks the set of arcs in a sequential way. So if an instance is inconsistent according to the last arc (i, j), the detection is very expensive. While with our distributed methods using n agents, the two agents respectively owning i and j rapidly detect inconsistency and immediately broadcast the end of the processing. With intermediate granularities, the previous reasoning occurs. But since the methods owns severals variables and consider their arcs successively, they are less efficient in these detections. The previous explains why inconsistent instances are relatively easier for distributed methods (see [15] for a complete analysis).

Table 3 presents computational peaks results. The first table gives results with $p = 90$. The second table gives speed-up over central AC-6. For $p_1 = 0.8$, DisAC-6 exceeds sp2 message passing capacities[3] for $p_2 \geq 0.63$, so comparisons are not available with this connectivity parameter.

[3] Overflow in message buffering capacities.

Table 3. Random problems computational peaks results

	time			#msg			#ccks		
p_1	0.2	0.5	0.8	0.2	0.5	0.8	0.2	0.5	0.8
DisAC-6 90 agents	10.52s	22.23s	-	13660.20	32493	-	206243	460272	-
DisAC-9 90 agents	0.79s	1.06s	1.50s	1663	2029	2431	761619	2.03e6	3.50e6
ratio	13.23	21	-	8.20	16	-	0.27	0.23	-
AC-6	0.28s	0.6s	0.91s	-	-	-	171370	344953	501970
DisAC-9 15 agents	0.09s	0.17s	0.25s	865.4	908.8	823.5	660964	1.69e6	2.60e6
ratio	3.06	3.53	3.64	-	-	-	0.26	0.20	0.19

7.2 The zebra problem

This classical benchmark uses 25 variables, 5 labels per variable ($n = 25, d = 5$) and 61 binary constraints. In our experimentations we have used $p \in \{2, 3, 4, 5, 10, 25\}$ which brings $11, 16, 27, 11, 43, 61$ inter-agents constraints. For each granularity we have used one hundred runs.

Table 4. The zebra problem

Algorithm	$p =$	1	2	3	4	5	10	25
AC-6	time	0.004s	-	-	-	-	-	-
	#msg	-	-	-	-	-	-	-
	#ccks	1161	-	-	-	-	-	-
DisAC-9	*time*	-	0.0052s	0.07s	0.0104s	0.0102s	0.0176s	0.0524s
	#msg	-	6	17	21.18	19.3	43.56	64.02
	#ccks	-	1463	1667.18	1815.58	1391.5	2444.58	3098.82
DisAC-6	time	-	0.008s	0.0092s	0.0096s	0.011s	0.0328s	0.1616s
	#msg	-	8	34	44.42	36.84	135.4	321
	#ccks	-	1248	1298.5	1290.06	1251.62	1347.24	1356

Table 4 gives executions results. According to the small size of this problem, distributed methods do not give any speed-up over the central algorithm. But speed-up is not our first goal since we also try to give efficient methods for distributed problems processing. As expected, DisAC-9 is faster than the other algorithm and particularly for finest granularities. We can also observe that the performance of the methods are strongly dependent of the number of inter-agents constraints.

8 Conclusion

We have provided DisAC-9, a coarsely grained distributed algorithm performing the arc-filtering of binary constraint networks. Our method is optimal in message passing operations which are critical in distributed systems[4]. DisAC-9 allows ef-

[4] In [15], we have extended our algorithm for achieving optimality in the number and in the size of the messages.

ficient processing of DCSP and with coarser granularity, give speed-up on the central AC-6. Surprisingly the experiments allow a characterization of computational peaks in distributed arc-consistency problems. Hard instances do not occur at the same tightness for CSP and for DCSP. Inconsistent problems are relatively easier for distributed procedures and the location of their time peaks depends of their granularities. This phenomenon requires a full theoretical study since it could allow efficient cost predictions in distributed frameworks.

As a practical extension of this work, we are combining it with a distributed search procedure. Since these kinds of combination have brought interesting results in the central framework [2], they should be studied with distributed CSPs. This will prune inconsistent parts of a distributed search space and by the way could save critical message passing operations.

References

1. Simonis, H.: A problem classification scheme for finite domain constraint solving. In: Proc. of the CP'96 workshop on Constraint Programming Applications: An Inventory and Taxonomy. (1996) 1–26
2. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: ECAI. (1994) 125–129
3. Yokoo, M., Hirayama, K.: Distributed breakout algorithm for solving distributed constraint satisfaction problems. In: ICMAS. (1996) 401–408
4. Hamadi, Y., Bessière, C., Quinqueton, J.: Backtracking in distributed constraint networks. In: ECAI. (1998) 219–223
5. Cooper, P.R., Swain, M.J.: Arc consistency: Parallelism and domain dependence. AI **58** (1992) 207–235
6. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence **28** (1986) 225–233
7. Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. In: First Int. Workshop on concurrent Constraint Satisfaction. (1995)
8. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. AI **45** (1990) 275–286
9. Culler, D.E., Liu, L.T., Martin, R.P., Yoshikawa, C.: LogP performance assessment of fast network interfaces. IEEE Micro (1996)
10. Samal, A., Henderson, T.: Parallel consistent labeling algorithms. Int. J. Parallel Program. **16** (1987) 341–364
11. Prosser, P., Conway, C., Muller, C.: A distributed constraint maintenance system. In: Proc. of the 12th Int. Conf. on AI. (1992) 221–231
12. Naor, M., Stockmeyer, L.: What can be computed locally ? SIAM J. Comput. **24** (1995) 1259–1277
13. Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence **65** (1994) 179–190
14. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed systems. TOCS **3** (1985) 63–75
15. Hamadi, Y.: Traitement des problèmes de satisfaction de contraintes distribués. PhD thesis, Université Montpellier II (1999) (in french).