A Distributed Arc-Consistency Algorithm

T. Nguyen, Y. Deville

Département d'Ingénierie Informatique, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium

Abstract

Consistency techniques are an efficient way of tackling constraint satisfaction problems (CSP). In particular, various arc-consistency algorithms have been designed such as the time optimal AC-4 sequential algorithm of Mohr and Henderson [8]. In this paper, we present a new distributed arc-consistency algorithm, called DisAC-4. DisAC-4 is based on AC-4, and is a coarse-grained parallel algorithm designed for distributed memory computers using message passing communication. Termination and correctness of the algorithm are proven. Theoretical complexities and experimental results are given. Both show linear speedup with respect to the number of processors. The strong point of DisAC-4 is its suitability to be implemented on very common hardware infrastructures like networks of workstations and/or PCs as well as on intensive computing parallel mainframes.

1 Introduction

Constraint satisfaction problem appears in many areas like artificial intelligence, operational research and hardware design. Informally, a CSP is composed of a finite set of variables, each of which is taking values in an associated finite domain, and a set of constraints between these variables. The constraints restrict the values the variables can simultaneously take. Resolving a CSP consists in finding one or all complete assignments of values to variables satisfying all the constraints. Numerous methods have been proposed to solve CSPs namely search procedures.

CSPs as formulated above are usually NP-complete. Search algorithms solving CSPs generate exponential sized search spaces. For this reason, it has been suggested to apply preprocessing techniques which would reduce the size of these search spaces.

Node, arc and path-consistency are preprocessing techniques which run in polynomial times. These algorithms have found wide application in artifi-

cial intelligence, image processing, pattern recognition and programming languages.

We restricted ourselves to arc-consistency. Many sequential algorithms have been designed to solve arc-consistency: Waltz [16] filtering algorithm; Mackworth's [6,7] AC-1, AC-2 and AC-3 algorithms; Mohr and Henderson's [8] AC-4 algorithm which is optimal in time complexity (AC-4 runs in $O(n^2d^2)$ where n is the number of variables, and d is the size of the largest domain); AC-5, a generic algorithm, proposed by Van Hentenryck, Deville and Teng [15], which can exploit properties of particular classes of constraints, yielding an $O(n^2d)$ time complexity.

When an arc-consistency algorithm has to distribute computing to a set of parallel processes, they have to access data structures and exchange results of their local computation. A first natural way is to use a shared memory. Several authors followed this shared memory approach. Henderson and Samal [12] proposed parallel versions of AC-1, AC-3 and AC-4 algorithms for shared memory parallel computers. A theoretical study shows that the parallel version of AC-4 with nd processors runs in $O(nd)$. They implement the algorithms on a BBN Butterfly multiprocessor. Since using nd processors is unrealistic, they used p processors, and experimented a linear speedup (that is $O(n^2d^2/p)$) on their example. Cooper and Swain [3] implemented and experimented a similar parallel AC-4 algorithm on the Connection Machine CM-2. The complexity is $O(nd\log(nd))$ due to communication overheads. They also gave a massively parallel algorithm expressed as a digital circuit. Kasif [5] proved the inherent sequentiality of arc-consistency algorithms. It means that with a polynomial number of processors, one cannot expect less than a $O(nd)$ time complexity. Zhang and Mackworth formulate a CSP as a *dual network* within arcs correspond to variables and nodes to constraints. They explored algorithms to compute consistency of such a network. They parallelized these algorithms and implemented them on a network of transputers [19,20].

Numerous authors studied distributed backtracking algorithms for solving CSP. Yokoo and his colleagues [18,17] formalize *cooperative distributed problem solving* (CDPS) as *distributed constraint satisfaction problems* (DCSP); they proposed distributed backtracking algorithms to solve DCSP. In [4], Collin and his colleagues examines the possibility of using uniform model of computation to design self-stabilizing distributed backtracking algorithms. In an uniform model of computation, all the processing agents are indistinguishable.

Share memory hardwares are not very common on existing computing infrastructures. Most of them are distributed memory environments, like processors connected by a local area network. Often in such environments shared memory is unavailable or has to be simulated, which can be costly.

In this paper, we give DisAC-4, a coarse-grained parallel algorithm designed on the basis of AC-4 for a distributed memory computer using message passing communication. We propose a method to efficiently exchange information between the cooperating processes. We prove termination and correctness of DisAC-4 and state its theoretical time complexity ($O(n^2d^2/p)$). The strong point of DisAC-4 is its suitability to be implemented on very common hardware infrastructures like workstations connected by an Ethernet network. Experimental results show a linear speedup.

This paper is organized as follows. Definitions and notations are presented in Section 2. To be self-contained, Section 3 introduces AC-4 algorithm. Section 4 describes a distributed memory parallel computer model and asynchronous message passing operations. The DisAC-4 algorithm, its correctness proof and its theoretical complexity are given in Section 5. Section 6 shows some experimental results. Finally, we state conclusions in Section 7.

2 Preliminaries

Most of our notations are taken from [13].

Definition 1 *A CSP is defined by a triplet* $(\mathcal{Z}, \mathcal{D}, \mathcal{C})$ *where:*

- $\mathcal{Z}$ *is a finite set of* variables $\{x_1, x_2, \ldots, x_n\}$*;*
- $\mathcal{D}$ *is a function which associate to each variable* x_i *a finite set of objects. The set of objects corresponding to a variable* x_i *is noted* D_i *and called* domain *of* x_i*. Elements of* D_i *are possible* values *of* x_i*;*
- $\mathcal{C}$ *is a set of constraints restricting values that variables can simultaneously take. Each constraint* $C^*_{i_1 i_2 \ldots i_k}$ *defined on variables* $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ *is associated to a predicate* $C_{i_1 i_2 \ldots i_k}$*. The atom* $C_{i_1 i_2 \ldots i_k}(v_{i_1}, v_{i_2}, \ldots, v_{i_k})$ *is true when assigning simultaneously values* $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ *to* $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ *variables do not transgress the constraint.*

In this paper, we restricts ourselves to binary CSPs. It is proven that every CSP can be transformed to an equivalent binary CSP [10]. A CSP $(\mathcal{Z}, \mathcal{D}, \mathcal{C})$ is binary when constraints of $\mathcal{C}$ are binary. Such a CSP may be associated to a directed graph.

Definition 2 *A binary CSP* $(\mathcal{Z}, \mathcal{D}, \mathcal{C})$ *is associated to a* directed graph *as follows: (i) each variable* x_i *of* $\mathcal{Z}$ *is associated to a* node i*; (ii) each constraint* C^*_{ij} *corresponds to a* directed arc (i, j)*. For simplicity, we make hypothesis that if a constraint* C^*_{ij} *is associated to arc* (i, j)*, there is a constraint* C^*_{ji} *associated to arc* (j, i) *such that* C^*_{ij} *and* C^*_{ji} *are the same except the fact that their arguments are interchanged.*

Definition 3 *A* label $\langle i, v \rangle$ *denotes the assignment of value* $v \in D_i$ *to variable* x_i.

In the following definitions, $arc(G)$ denotes the set of arcs of a graph G, and $node(G)$ the set of its nodes.

Definition 4 *Let* $(\mathcal{Z}, \mathcal{D}, \mathcal{C})$ *be a binary CSP and* G *its associated graph. A value* $v \in D_i$ *is* arc-consistent *or a label* $\langle i, v \rangle$ *is* arc-consistent *iff*

$$\forall\, (i,j) \in arc(G), \exists w \in D_j : C_{ij}(v, w)$$

Definition 5 *Let* $(\mathcal{Z}, \mathcal{D}, \mathcal{C})$ *be a binary CSP and* G *its associated graph. An arc* $(i,j) \in arc(G)$ *is* arc-consistent *iff*

$$\forall v \in D_i, \exists w \in D_j : C_{ij}(v, w)$$

Definition 6 *A binary CSP is* arc-consistent *iff all the arcs of its associated graph are arc-consistent.*

Definition 7 *Let* G *be the graph associated to a binary CSP* $(\mathcal{Z}, \mathcal{D}, \mathcal{C})$ *and* $\mathcal{D}'$ *such that*

$$\forall i \in node(G) : D'_i \subseteq D_i$$

$\mathcal{D}'$ *is an* arc-consistent domain *of* G *iff the binary CSP* $(\mathcal{Z}, \mathcal{D}', \mathcal{C})$ *which is also associated to graph* G *is arc-consistent. Furthermore,* $\mathcal{D}'$ *is the* largest arc-consistent *domain iff there no different arc-consistent domain* $\mathcal{D}''$ *such that*

$$\forall i \in node(G) : D'_i \subset D''_i \subseteq D_i$$

3 The arc-consistency algorithm AC-4

Arc consistency can be achieved by using the following domain restriction operation:

$$\forall (i,j) \in arc(G) : D_i \leftarrow \{v \mid v \in D_i \wedge \exists w \in D_j : C_{ij}(v, w)\}$$

One simply needs to go through each arc (i,j), and for each value v of head node i check whether one can find a value w in tail node j compatible with v (that is $C_{ij}(v,w)$ is satisfied). All the values of node i which fail to conform to this condition are removed. Several passes over the arcs of $arc(G)$ are often necessary and arc-consistency is fulfilled only whenever no value has been

deleted during the last pass. In the last two decades, numerous algorithms have been constructed to compute arc-consistency. Combining previous works, Mackworth [6] formulated the first arc-consistency algorithm AC-1 and two of its variants AC-2 and AC-3. Mohr and Henderson [8] proposed AC-4, an improvement of precedent algorithms.

AC-4 is based on the following observation: a value v of some node i is arc-consistent if it has a support in its neighboring nodes. More precisely, let $\langle i, v \rangle$ be a label. The set of supports of $\langle i, v \rangle$ is the following set of labels:

$$SupportOf_{\langle i,v \rangle} = \{\langle j, w \rangle \mid (i, j) \in arc(G) \wedge w \in D_j \wedge C_{ij}(v, w)\}$$

A label $\langle i, v \rangle$ becomes inconsistent when at some neighboring node j, there is no label $\langle j, w \rangle$ in $SupportOf_{\langle i,v \rangle}$. AC-4 uses a data structure $counter[\langle i, v \rangle][j]$ containing the number of labels at node j supporting $\langle i, v \rangle$:

$$counter[\langle i, v \rangle][j] = \#\{\langle j, w \rangle \mid \langle j, w \rangle \in SupportOf_{\langle i,v \rangle}\}$$

When this counter is zero, the label $\langle i, v \rangle$ is detected as inconsistent. AC-4 uses another data structure, $Support_{\langle j,w \rangle}$, containing all the labels supported by $\langle j, w \rangle$:

$$Support_{\langle j,w \rangle} = \{\langle i, v \rangle \mid \langle j, w \rangle \in SupportOf_{\langle i,v \rangle}\}$$

This data structure is used when the label $\langle j, w \rangle$ is detected as inconsistent.

As depicted by Fig. 1, the algorithm consists of two stages: building the appropriate data structures (*Support* and *counter*), and pruning the inconsistent labels. All inconsistent labels are put in a set called *List*. These are processed in the second stage of the algorithm. For each inconsistent label $\langle j, w \rangle$ of *List*, the set containing label $\langle i, v \rangle$ which are supported by $\langle j, w \rangle$ is examined. Since $\langle j, w \rangle$ is inconsistent and has to be deleted, variable $counter[\langle i, v \rangle][j]$ is decremented and checked. If it falls down to zero, label $\langle i, v \rangle$ is detected as inconsistent and so appended to *List* for later processing.

AC-4 possesses the following properties: (i) it computes the largest arc-consistent solution; (ii) its time and space complexities (for complete graphs) are $O(n^2d^2)$ where n is the number of nodes and d the size of the largest domain; (iii) its time complexity is optimal. An explanation of those features and a proof of correctness of the algorithm can be found in [8].

```
    program AC-4;
01  begin
02    List := {}; Support_⟨_,_⟩ := {};
03    (* Step 1: Build supports *)
04    for each (i, j) ∈ arc(G) do
05      for each v ∈ D_i do
06      begin
07        counter[⟨i, v⟩][j] := 0;
08        for each w ∈ D_j do
09          if C_ij(v, w) then
10          begin
11            counter[⟨i, v⟩][j] := counter[⟨i, v⟩][j] + 1;
12            Support_⟨j,w⟩ := Support_⟨j,w⟩ ∪ {⟨i, v⟩}
13          end;
14        if counter[⟨i, v⟩][j] = 0 then
15        begin List := List ∪ {⟨i, v⟩}; D_i := D_i − {v} end
16      end;
17    (* Step 2: Remove unsupported values *)
18    for each ⟨j, w⟩ ∈ List do
19    begin
20      List := List − {⟨j, w⟩};
21      for each ⟨i, v⟩ ∈ Support_⟨j,w⟩ such that v ∈ D_i do
22      begin
23        counter[⟨i, v⟩][j] := counter[⟨i, v⟩][j] − 1;
24        if counter[⟨i, v⟩][j] = 0 then
25        begin List := List ∪ {⟨i, v⟩}; D_i := D_i − {v} end
26      end
27    end
28  end.
```

Fig. 1. The AC-4 algorithm.

4 Asynchronous message passing model

Distributed memory parallel computers have become increasingly common. They are built with processors sharing only a *communication network*. Running processes can exchange information by using *channels*. A channel is an abstraction of a physical communication network; it provides a communication path between processes.

With asynchronous message passing, communication channels are unbounded queues of messages. A process appends a message to the end of a channel queue by executing a **send** operation. Since the queue is unbounded, execution of **send** operation does not block the sender. A process receives a message from a channel by executing **receive** statement. Execution of **receive** delays the receiver until the channel is non-empty; then the message at the front of

the channel is removed and stored in variables local to the receiver. Because channels are modelized as FIFO queues, one usually makes the following hypothesis: (i) there is no loss of message; (ii) messages are received in the order they have been sent. These assumptions are rather strong but often met by the currently existing distributed programming platforms such as PVM [11].

Many different notations have been proposed for asynchronous message passing. We use notations proposed in [1]. A parallel program is composed of processes which run concurrently and share global variables.

program *Example*;
global variables declaration
process P_1 :: Body of P_1 process
process P_2 :: Body of P_2 process
...

In a distributed memory computer using asynchronous message passing communication, the only available global variables are channels. A channel is a queue of messages that have been sent but not yet received. A channel declaration has the form:

chan $ch(id_1 : Type_1, id_2 : Type_2, \ldots, id_n : Type_n)$

id_k and $Type_k$, $k \in [1..k]$, are names and types of data fields in messages transmitted via the channel. Field names are optional. Since channels are an abstraction of an underlying network, access rules have to be defined following the topology of the network. It is the task of the programmer to determine within a program which processes can access a channel to write or to read a message.

Channels are accessed by means of two basic primitives: **send** and **receive**. A process sends a message to channel ch by executing

send $ch(expr_1, expr_2, \ldots, expr_n)$

An other process can receive this message by executing

receive $ch(arg_1, arg_2, \ldots, arg_n)$

To determine whether a channel is currently empty, a process calls the boolean-valued function

empty(ch)

In most local area networks such as Ethernet or Token Ring, each processor is directly connected to every other one. Such communication networks often support a broadcast operation, which transmits a message from one processor

to all processors. Let $ch[1..n]$ be an array of channels, and let each channel $ch[i]$ be associated to a process P_i. Each process broadcasts a message by executing

broadcast $ch(expr)$

The effects is thus the same as executing n **send** operations in parallel, with each sending a message to a different channel. Usually, on Ethernet or Token Ring local area networks, the cost of a **broadcast** is similar to the cost of a **send**.

Data with different types can be exchanged within a same channel by using an **union** type constructor as shown in the following example:

type $DataTypeFlag = \mathbf{enum}(\mathtt{tag}_1, \mathtt{tag}_2, \ldots, \mathtt{tag}_n)$
type $DataType = \mathbf{union}(id_1 : Type_1, id_2 : Type_2, \ldots, id_n : Type_n)$
chan $ch(kind : DataTypeFlag, data : DataType)$

If a process sends a message by using operation

send $ch(\mathtt{tag}_k, expr)$

the receiver process may get the message and decode it by executing statements

receive $ch(kind, arg)$;
case $kind$ **of**
begin
  $\mathtt{tag}_1$: Process arg as a $Type_1$ data;
  $\mathtt{tag}_2$: Process arg as a $Type_2$ data;
  ...
  $\mathtt{tag}_n$: Process arg as a $Type_n$ data
end

The channel ch accepts data of types $Type_1, Type_2, \ldots, Type_n$; the receiver process checks the value of $kind$ to determine the type of the content of arg.

5 The distributed algorithm DisAC-4

5.1 The algorithm

In DisAC-4 algorithm, we distribute the computation among p processes that we designate by $Worker[1]$, $Worker[2], \ldots,$ $Worker[p]$. They run the same code

```
    program DisAC-4;
    process Worker[k : 1..p] ::
01  begin
02    List := {}; Support_⟨_,_⟩ := {};
03  |  InitComWorker();
04  |  ToSendList := {};
05    (* Step 1: Build supports *)
06  |  for each (i, j) ∈ arc(G) such that i ∈ MyNodes(k) do
07      for each v ∈ D_i do
08      begin
09        counter[⟨i, v⟩][j] := 0;
10        for each w ∈ D_j do
11          if C_ij(v, w) then
12          begin
13            counter[⟨i, v⟩][j] := counter[⟨i, v⟩][j] + 1;
14            Support_⟨j,w⟩ := Support_⟨j,w⟩ ∪ {⟨i, v⟩}
15          end;
16        if counter[⟨i, v⟩][j] = 0 then
17        begin
18          List := List ∪ {⟨i, v⟩}; D_i := D_i − {v};
19  |       ToSendList := ToSendList ∪ {⟨i, v⟩}
20        end
21      end;
22    (* Step 2: Remove unsupported values *)
23  |  for ever do
24  |  begin
25      for each ⟨j, w⟩ ∈ List do
26      begin
27        List := List − {⟨j, w⟩};
28        for each ⟨i, v⟩ ∈ Support_⟨j,w⟩ such that v ∈ D_i do
29        begin
30          counter[⟨i, v⟩][j] := counter[⟨i, v⟩][j] − 1;
31          if counter[⟨i, v⟩][j] = 0 then
32          begin
33            List := List ∪ {⟨i, v⟩}; D_i := D_i − {v};
34  |         ToSendList := ToSendList ∪ {⟨i, v⟩}
35          end
36        end
37      end;
38  |     SendMessage(ToSendList);
39  |     ReceiveMessage(List)
40  |  end
41  end.
```

Fig. 2. The DisAC-4 algorithm: The *Worker* processes.

but on different data. In this paper, we will assume $1 \leq p \leq n$, n is the number of domains of the CSP.

Fig. 2 describes the algorithm of the workers. The bold line numbers give the modifications with respect to AC-4. Let $Worker[k]$, $k \in [1..p]$, be a worker process. $Worker[k]$ handles a set of nodes and associated domains which are given by the $MyNodes(k)$ function call. Like AC-4, computations of $Worker[k]$ consists in two parts. In the first stage, it independently builds the local data structures *counter* and *Support*, and detects local inconsistent labels. In the second stage, the inconsistent labels are treated. They may either be locally generated inside the process or produced by the other workers. Hence, $Worker[k]$ has to transmit its locally detected inconsistent labels to the other workers and collect inconsistent labels sent by them.

The space complexity of DisAC-4 is also $O(n^2d^2)$. The *List* variable of AC-4 can be seen as a share memory data structure, simulated by message passing. The other data structures of AC-4 are partitioned among the p workers. More precisely, the data structures of process $Worker[k]$ are the following:

- $counter[\langle i, \_\rangle][\_]$ with $i \in MyNodes(k)$.
- $Support_{\langle j,w \rangle}$ that only contains labels of the form $\langle i, \_\rangle$ with $i \in MyNodes(k)$. Hence, the number of elements of $Support_{\langle i,w \rangle}$ is at most nd/p where p is the number of workers.
- D_i with $i \in MyNodes(k)$.

In fact, in the first stage of computation, $Worker[k]$ needs all the domains. But only domains D_i with $i \in MyNodes(k)$ will have to be updated. The other domains will not be used in the second stage. Since p, the number of *Worker* processes, is assumed to be $1 \leq p \leq n$, the space complexity of the domains is $O(n^2d)$. This does not influence the global $O(n^2d^2)$ space complexity. Note that the partition and construction of the *Support* data structure ensures that $i \in MyNodes(k)$ in line 28 of the algorithm.

We adopted the hypothesis of fully connected processors architecture in which **broadcast** operation is available. Let $ch[0..p]$ be an array of channels and let $ch[k]$ be the channel associated to $Worker[k]$. The purpose of $ch[0]$ will be explained further. Each process can send a message in any channel but a process can only receive a message via its dedicated channel.

As we mentioned above, $Worker[k]$ has to: (i) broadcast its locally detected inconsistent labels (contained in *ToSendList*) to the other workers; (ii) update its *List* structure with inconsistent labels sent by them. Broadcasting the *ToSendList* is done at lines 38 by executing the *SendMessage* procedure; updating *List* is performed at line 39 by calling the *ReceiveMessage* procedure. These two routines (Fig. 3 and 4) can be seen as simulating a shared memory data structure for the *List* variable of AC-4.

```
    procedure SendMessage(ToSendList);
01  begin
02    (* Broadcast ToSendList *)
03    if ToSendList ≠ {} then
04    begin
05      broadcast ch(list, k, ToSendList);
06      ToSendList := {}
07    end
08  end.
```

Fig. 3. The DisAC-4 algorithm: The *SendMessage* procedure.

```
    procedure ReceiveMessage(List);
01  begin
02    (* Block until a message arrives *)
03    await not empty(ch[k]);
04    (* Process incoming messages *)
05    while not empty(ch[k]) do
06    begin
07      receive ch[k](kind, arg1, arg2);
08      case kind of
09      begin
10        stop : exit process Worker[k];
11        list : if arg1 ≠ k then List := List ∪ arg2
12      end
13    end
14  end.
```

Fig. 4. The DisAC-4 algorithm: The *ReceiveMessage* procedure.

```
    program DisAC-4;
    process Controller ::
01  begin
02    DetectTermination();
03    broadcast ch(stop, _, _)
04  end.
```

Fig. 5. The DisAC-4 algorithm: The *Controller* process.

The task of the *Controller* process (Fig. 5) consists in: (i) detecting the termination of the computation calling the *DetectTermination* procedure; (ii) signaling the termination to the workers by mean of a `stop` message. Informally, the computation ends whenever each worker has treated all the labels detected as inconsistent and no more inconsistent labels are generated.

We first state a sufficient condition for the *DetectTermination* procedure ensuring the partial correctness of DisAC-4. We postpone the implementation of this *DetectTermination* routine.

Lemma 1 *Assume no* `stop` *message broadcasted by the Controller process. Suppose all Worker processes are executing ReceiveMessage procedure and waiting for a message at line 3, and all the channels ch are empty. Then, DisAC-4 has built the largest arc-consistent solution.*

Proof. Since the algorithm of the workers is based on AC-4, it is sufficient to show that each label $\langle i, v\rangle$ detected as inconsistent by some worker has been treated by all the other workers. Suppose that some process $Worker[k]$, $k \in [1..p]$, has not treated a label $\langle i, v\rangle$ detected as inconsistent. Because all workers are waiting for a message at line 3 of the *ReceiveMessage* procedure, this label $\langle i, v\rangle$ has been broadcasted because a *SendMessage* call precedes any *ReceiveMessage* call (lines 38–39). Since the local *List* set of $Worker[k]$ is empty, $\langle i, v\rangle$ must be contained in a message that $Worker[k]$ did not yet receive. Therefore, we conclude that the channel $ch[k]$ is not empty what is contradictory with the hypothesis. □

Theorem 1 (Partial correctness) *Assuming the DetectTermination routine only returns whenever conditions mentioned in Lemma 1 are satisfied, then algorithm DisAC-4 is partially correct.*

Proof. Consequence of Lemma 1 as algorithm DisAC-4 only terminates after the *DetectTermination* procedure returns. □

To guarantee the end of the computation and thus the total correctness of DisAC-4, a correct implementation of the *DetectTermination* procedure has to be provided, that will be done in the next section.

Theorem 2 (Time complexity) *Assume the DetectTermination procedure instantaneously returns whenever the conditions described in Lemma 1 are fulfilled. The time complexity of DisAC-4 is* $O(n^2d^2/p)$.

Proof.

– **Step 1** In the first step of their algorithm, the workers run in parallel and independently. Hence, for this first step, the time complexity of the global

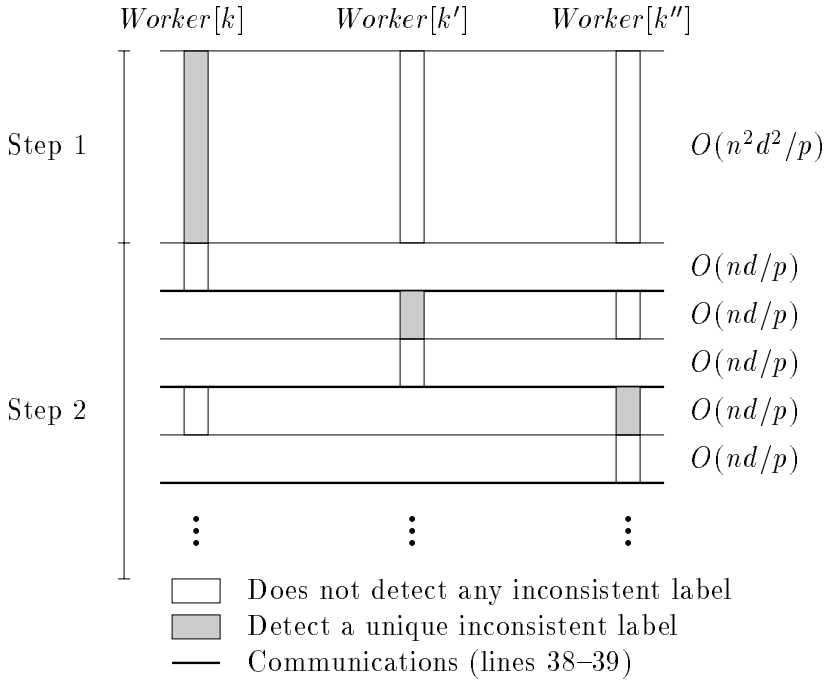

Fig. 6. DisAC-4: Worst case execution.

computation is the complexity of a worker. The **for** loop at line 6 examines at most n^2/p arcs. The **for** loops of lines 7 and 10 each iterate on d values. Hence, the time complexity of Step 1 is $O(n^2d^2/p)$.

– **Step 2** The worst case is the following (Fig. 6):
  · At the end of Step 1, only one label is detected as inconsistent by some $Worker[k]$, $k \in [1..p]$.
  · Step 2 of the other workers is void and they are all waiting for a message at line 39.
  · Step 2 of $Worker[k]$ treats the unique inconsistent label of *List*. Since *List* contains only one element and $Support_{\langle j,w \rangle}$ at most nd/p labels, the nested **for** loops of lines 25 and 28 run in $O(nd/p)$ time. In the worst case, we assume that no more inconsistent labels are detected at this stage. $Worker[k]$ sends the inconsistent label (line 38) and waits at line 39.
  · The other workers receive the singleton of $Worker[k]$ and put it in their *List* variables (line 39). They reexecute Step 2 in parallel in $O(nd/p)$ time. As result of these computations, we assume another single inconsistent label detected by some $Worker[k']$, $k' \in [1..p]$. $Worker[k']$ locally treats this label and updates its *counter* variables; it takes again an $O(nd/p)$ time. It broadcasts its *ToSendList* singleton at line 38.
  · The same scenario happens and so on, until all labels are detected as inconsistent.

  Since there are at most nd labels, we conclude that the time complexity of Step 2, excluding communications, is $O(n^2d^2/p)$.

– **Communications** Step 2 includes the time spent by the workers in the *SendMessage* and *ReceiveMessage* procedures.
  · Assuming a (very bad) implementation of **broadcast**, the complexity of such a **broadcast** is bounded by p successive **send** calls. Time spent by a worker for such a sending is proportional to the size of its *ToSendList*. The sum of successive sizes of *ToSendList* is bounded by nd. Hence a complexity of $O(pnd)$ for all the executions of *SendMessage* by this worker.
  · The complexity of *ReceiveMessage*, excluding waiting time, is proportional to the size of arg_2 (lines 7 and 11). The sum of the successive sizes of arg_2 is bounded by nd. Hence a complexity of $O(nd)$ for all the executions of *ReceiveMessage* by a worker.

  The complexity of DisAC-4 is then $O(n^2d^2/p + pnd)$, yielding $O(n^2d^2/p)$ assuming $p \leq \sqrt{nd}$. □

Theorem 2 shows a linear speedup of DisAC-4 with respect to the number of workers. The DisAC-4 algorithm is not intended to be massively parallel. The constraint $p \leq \sqrt{nd}$ is obviously met when $n \leq d$ (since $p \leq n$).

5.3 The Controller process

We may use a distributed snapshot algorithm to detect the end of the computations. By such an algorithm, a process in a distributed system determines a global state of the system during a computation. In our case, this process is called *Controller* and the state of interest corresponds to the conditions described in Lemma 1. But such a general algorithm is very heavy and complex to carry out; moreover, it could degrade the linear speedup of DisAC-4. Therefore, we rather choose to design a specialized and optimized termination detection algorithm exploiting the particular features of our fully connected communication topology and the arc-consistency computation of AC-4.

The task of the *Controller* process consists in counting the number of `list` typed messages that have been exchanged by the workers and checking whether each of them has treated a same number of `list` messages. In this case, it broadcasts a termination order to the workers as all inconsistent labels have been detected and treated. The correctness of this termination detection will be proven later.

The controller fulfills its task by means of the *DetectTermination* routine (Fig. 7). It is built over two other routines: the *CollectList* procedure and the *Polling* boolean function (Fig. 8 and 9). In the *CollectList* procedure, the controller attempts to get (without blocking) messages from its dedicated channel $ch[0]$: it checks the state of the channel before executing a **receive** statement. If the operation is successful and the received message contains

```
    procedure DetectTermination();
01  begin
02    InitComController();
03    terminated := false;
04    while not terminated do
05    begin
06      CollectList();
07      terminated := Polling()
08    end
09  end.
```

Fig. 7. The DisAC-4 algorithm: The *DetectTermination* procedure.

inconsistent labels, the variable *mastercounter* is incremented. If no message arrives after `max` successive attempts, the procedure returns. By calling the *Polling* function, the controller broadcasts a timestamped polling request to the workers, and waits for their replies. If k correctly timestamped replies are collected, the polling operation is successful and *Polling* returns `true`. Else the controller receives a message containing inconsistent labels, the *mastercounter* variable is incremented and *Polling* immediately returns with value `false`. Inside the *DetectTermination* procedure, the controller successively calls the *CollectList* and *Polling* routines until the *Polling* function returns value `true`. The procedure *InitComController* (Fig. 10) initializes the *mastercounter* and *timestamp* variables of the controller.

5.4 The Worker processes revisited

We modify the *SendMessage* and *ReceiveMessage* procedures allowing the workers to cooperate with the controller in detecting the termination of the computation. These procedures are shown in Fig. 11 and 12. The differences with their simplified versions presented in the precedent section are highlighted by bold line numbers.

Answering a polling request with a message containing a `pollreply` tag is only performed at certain conditions as shown by line 10 of the *SendMessage* routine. First, a polling request must have been sent by the controller. Then *ToSendList* must be empty otherwise some inconsistent labels have not yet been treated by the other workers. Next, *List* must be empty as the worker must have finished processing all the inconsistent labels at its disposal before answering a polling request. Finally, *localcounter* of the worker (the number of `list` messages it has received so far) must be equal to *mastercounter* (the number of `list` messages controller has received). Otherwise, either some `list` messages already received by the controller have still to be received and treated by the worker, or some `list` messages already received and treated

```
    procedure CollectList();
01  begin
02    (* Collect inconsistent labels lists *)
03    SuccessiveFailedAttempts := 0;
04    while SuccessiveFailedAttempts ≤ max do
05    begin
06      if empty(ch[0]) then
07      begin
08        SuccessiveFailedAttempts := SuccessiveFailedAttempts + 1;
09        sleep(afewtime)
10      end
11      else
12      begin
11        receive ch[0](kind, arg1, arg2);
12        if kind = list then
13        begin
14          SuccessiveFailedAttempts := 0;
15          mastercounter := mastercounter + 1
16        end
17      end
18    end
19  end.
```

Fig. 8. The DisAC-4 algorithm: The *CollectList* procedure.

by the worker have still to be received by the controller.

The routine *InitComWorker* initializes the *localcounter* and *pollrequest* variables (Fig. 13).

5.5 Properties of the algorithm (cont'd)

Lemma 2 *Suppose that the Controller process collects p correctly timestamped replies after having broadcasted a polling request. Let ListMessages be the set of all* `list` *messages sent by the Worker processes before replying to the polling request, and let C be the value of mastercounter as sent by the Controller process in the considered polling request. The following assertions are true:*

- **Assertion 1** *The Controller process has received all the messages of ListMessages.*
- **Assertion 2** $\#ListMessages = C$
- **Assertion 3** *For all* $k \in [1..p]$*, the value of* $Worker[k].localcounter$ *at the polling reply equals C.*
- **Assertion 4** *Let be a* $Worker[k]$*,* $k \in [1..p]$*. Let m be a* `list` *message sent*

```
    function Polling() : Boolean;
01  begin
02    (* Process a polling *)
03    timestamp := timestamp + 1;
04    broadcast ch(pollrequest, timestamp, mastercounter);
05    nbpollreply := 0;
06    while nbpollreply < p do
07    begin
08      receive ch[0](kind, arg1, arg2);
09      case kind of
10      begin
11        pollreply : if arg1 = timestamp
12                    then nbpollreply := nbpollreply + 1;
13        list : begin
14                 mastercounter := mastercounter + 1;
15                 return false
16               end
17      end
18    end;
19    return true
20  end.
```

Fig. 9. The DisAC-4 algorithm: The *Polling* function.

```
    procedure InitComController();
01  begin
02    mastercounter := 0; timestamp := 0
03  end.
```

Fig. 10. The DisAC-4 algorithm: The *InitComController* procedure.

by Worker$[k]$*. If* $m \notin$ *ListMessages, then there exists another* `list` *message that is not contained in ListMessages.*

- **Assertion 5** *Let* m *be a* `list` *message. If* $m \notin$ *ListMessages, then* m *is not sent by any worker.*
- **Assertion 6** *The channels contain no* `list` *message.*

Proof.

- **Assertions 1 and 2** Because channels guarantee messages reception in the order they have been sent, the controller has received all the messages sent by the workers before they reply to its polling request. Hence the size of *ListMessages* is the value C of *mastercounter* as sent in the polling request.
- **Assertion 3** Because *Worker*$[k]$ replied to the polling with a `pollreply` (line 10 in *SendMessage* procedure), its *localcounter* equals *mastercounter* of the controller, as sent in the polling request (that is C).

```
    procedure SendMessage(ToSendList);
01  begin
02    (* Broadcast ToSendList and/or reply to polling request *)
03    if ToSendList ≠ {} then
04    begin
05      broadcast ch(list, k, ToSendList);
06      ToSendList := {};
07 |    pollrequest := false
08    end
09 |  else
10 |    if pollrequest and List = {} and localcounter = mastercounter then
11 |    begin
12 |      send ch[0](pollreply, timestamp, _);
13 |      pollrequest := false
14 |    end
15  end.
```

Fig. 11. The DisAC-4 algorithm: The *SendMessage* procedure revisited.

```
    procedure ReceiveMessage(List);
01  begin
02    (* Block until a message arrives *)
03    await not empty(ch[k]);
04    (* Process incoming messages *)
05    while not empty(ch[k]) do
06    begin
07      receive ch[k](kind, arg1, arg2);
08      case kind of
09      begin
10        stop : exit process WORKER[k];
11        list : begin
12                 if arg1 ≠ k then List := List ∪ arg2;
13 |               localcounter := localcounter + 1
14              end;
15 |      pollrequest : begin
16 |                       pollrequest := true;
17 |                       timestamp := arg1;
18 |                       mastercounter := arg2
19 |                     end
20      end
21    end
22  end.
```

Fig. 12. The DisAC-4 algorithm: The *ReceiveMessage* procedure revisited.

```
    procedure InitComWorker();
01  begin
02    localcounter := 0; pollrequest := false
03  end.
```

Fig. 13. The DisAC-4 algorithm: The *InitComWorker* procedure.

- **Assertion 4** Since $m \notin ListMessages$, it is sent by $Worker[k]$ after having replied to the polling request and calling *SendMessage* at line 38. Hence, message m contains inconsistent labels that $Worker[k]$ has detected after having processed inconsistent labels contained in another `list` message m'. This latter message m' is not counted in the value of *localcounter* of $Worker[k]$ whenever its was replying to the polling request. If $m' \notin ListMessages$, the assertion is proven. Else, there must exists another `list` message $m'' \notin ListMessages$, otherwise Assertions 2 or 3 would be false.
- **Assertion 5** Suppose $m \notin ListMessages$ and sent by some $Worker[k]$, $k \in [1..p]$. Applying Assertion 4 successively, we have an infinite sequence of different `list` messages $m', m'', \ldots$ not belonging to *ListMessages*. This is impossible the number of `list` messages is infinite since the set of labels is finite.
- **Assertion 6** Consequence of Assertions 3 and 5. □

Lemma 3 *The Controller and the Worker processes terminate.*

Proof. Let be a $Worker[k]$, $k \in [1..p]$. It does not reply to the latest polling request from the controller whenever it calls *SendMessage* if: (i) its *List* is not empty; (ii) its *localcounter* does not equal C, the value of *mastercounter* of the controller when this latter was sending its request. $Worker[k]$ is constructed such that (i) cannot remain infinitely true. Two alternatives are possible when (ii) is verified:

- $Worker[k].localcounter < C$: There are `list` messages already received by the controller and not yet treated by $Worker[k]$; this latter will wait for these messages and process them before replying to the controller.
- $Worker[k].localcounter > C$: $Worker[k]$ has treated `list` messages which the controller has not received whenever it was broadcasting its polling request. The controller will receive them waiting for replies. Its polling process will fail. Therefore, $Worker[k]$ do not have to reply to the polling request of the controller.

Hence in the *Polling* function, the **while** loop of line 6 always terminates. Since there is at most nd labels, the controller receives at most nd `list` messages. *Polling* can be called at most nd times and therefore **while** loop in $DetectTermination$ (line 4) always ends. The fact that the workers terminate whenever the controller ends is trivial. □

Theorem 3 (Total correctness) *DisAC-4 is totally correct.*

Proof. Theorem 1 ensures the partial correction of the algorithm. Lemma 2 proves the *DetectTermination* procedure is built to detect the correct termination conditions. In an other hand, this procedure does not influence the underlying arc-consistency computations. Lemma 3 guarantees the termination of all processes. □

Let us now analyze the theoretical time complexity of our full DisAC-4 algorithm, including the detection of termination. Without such a detection, Theorem 3 showed a complexity of $O(n^2d^2/p)$ for DisAC-4.

Theorem 4 (Time complexity) *If the number of polling requests that the Controller process has sent is $O(n^2d^2/p^2)$, then the complexity of DisAC-4 is $O(n^2d^2/p)$.*

Proof. Let R the number of polling requests sent by the controller. Each worker has to treat R `pollrequest` messages. The complexity of a worker becomes $O(n^2d^2/p+R)$. With the hypothesis on R, we still have a complexity of $O(n^2d^2/p)$ for the workers.

The complexity of the controller depends on the number of messages. This number is bounded by $(nd+R+p\times R)$; these three terms correspond respectively to the number of inconsistent labels, the number of polling requests and the number of polling replies. With the hypothesis on R, the time complexity of the controller becomes $O(n^2d^2/p)$.

Hence time complexity is $O(n^2d^2/p)$ for DisAC-4. □

In a real implementation of DisAC-4, the number of polling requests will depends on the constants `max` and `afewtime`. Choosing high values for these constants ensures the fulfillment of the hypothesis of Theorem 4, hence yields a theoretical complexity of $O(n^2d^2/p)$; neverthless, this can induce high execution times. As we will see in the next section, it is easy to find empirically values for `max` and `afewtime` ensuring both the application of Theorem 4 and good performances.

5.6 Granularity

In DisAC-4, we choose a coarse-grained approach for parallelism ($p \leq n$). Hence the domain D_i of a node i is handled by a single process. This reduces

communication overheads compared with approaches such as [12,3] where data are split among processes.

In the proposed DisAC-4 algorithm, communication between the *Worker* processes is reduced to its simplest form. Each of them sends its detected inconsistent labels and receives inconsistent labels detected by the other only if its has nothing else to do (lines 38–39). This choice has been motivated by our objective of a coarse-grained parallelism.

From a theoretical point of view, calls to *SendMessage* and *ReceiveMessage* could also be added anywhere in the algorithm of the workers without altering its correctness nor its time complexity. Experimental results showed that such an increase of the number of messages does not influence the performance. The potential advantage of more parallelism is reduced by the overhead induced by such new messages.

6 Experimental results

We have implemented DisAC-4 on Sun workstations connected to an Ethernet local network. PVM [11] provides us a distributed message passing programming environment which is widely used in intensive parallel computing. PVM proposes tools which allow asynchronous message passing operations as described in Section 3. All our program code is written in C.

Since PVM supports heavyweight processes, each processor deals with only one process. We implemented *MyNodes* function in such a way that either $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$ successive domains are distributed to each *Worker* process. To compare parallel algorithm performance, a quotient called *speedup* is defined by setting

$$speedup = \frac{\text{Execution time of sequential AC-4}}{\text{Execution time of DisAC-4 with } k \text{ processors}}$$

We ran DisAC-4 algorithm with three kinds of CSPs: N-queens problem, reverse N-queens problem, and the benchmark CSP used in [3]. N-queens problem consists in seeking ways to place N queens on a $N \times N$ chessboard, one queen per row, so that each pair of queens *does not attack* each other. The problem is formalized as CSP with variables $x_1, x_2, \ldots, x_N$ where x_i is the position of the queen in row i. Initial domains and constraints associated to the problem are:

$$\forall i \in node(G) : D_i \subseteq \{1, 2, \ldots, n\}$$

$$\forall i,j \in node(G) : i \neq j \Rightarrow (i,j) \in arc(G)$$
$$\forall (i,j) \in arc(G), \forall v \in D_i, \forall w \in D_j :$$
$$C_{ij}(v,w) \equiv (v \neq w) \wedge (|i-j| \neq |v-w|)$$

The constraints impose that two queens cannot be one a same column nor on a same diagonal.

In opposite, in reverse N-queens problem, each pair of queens *does attack* each other; the constraints are:

$$\forall (i,j) \in arc(G), \forall v \in D_i, \forall w \in D_j :$$
$$C_{ij}(v,w) \equiv (v = w) \vee (|i-j| = |v-w|)$$

Reverse N-queens provides a better test-bed for consistency algorithms than N-queens [9].

We call N-CS the benchmark used by Cooper and Swain in [3]. This CSP is defined by the following domains and constraints:

$$\forall i \in node(G) : D_i \subseteq \{1,2,\ldots,n\}$$
$$\forall i,j \in node(G) : i \neq j \Rightarrow (i,j) \in arc(G)$$
$$\forall (i,j) \in arc(G), \forall v \in D_i, \forall w \in D_j : C_{ij}(v,w) \equiv \neg(|j-i| = 1 \wedge v > w)$$

Enforcing arc-consistency for N-CS, AC-4 and DisAC-4 build *Support* data structures whose sizes are close to maximal i.e. respectively nd and nd/k elements. Another feature of N-CS benchmark is that most of inconsistent labels are detected in the Step 2 of the algorithms.

Note that in our three benchmark problems, the initial domains are not fully specified. This is necessary as our objective is not to find a solution to these problems (what would require an explicit enumeration or labeling), but only to apply (once) an arc-consistency algorithm. In order to ensure the detection of inconsistent labels and constraint propagation, we have initially restricted some of the domains before applying the arc-consistency algorithm. Such a situation is representative of the use of arc-consistency within a search procedure.

Fig. 14, 15 and 16 represent the speedups obtained with these CSPs. The right side graphs depict speedups of the algorithm taken globally for different problem sizes; the left side graph describe speedups obtained with its different parts. The ideal case is to obtain a linear speedup with a slope equal to one.

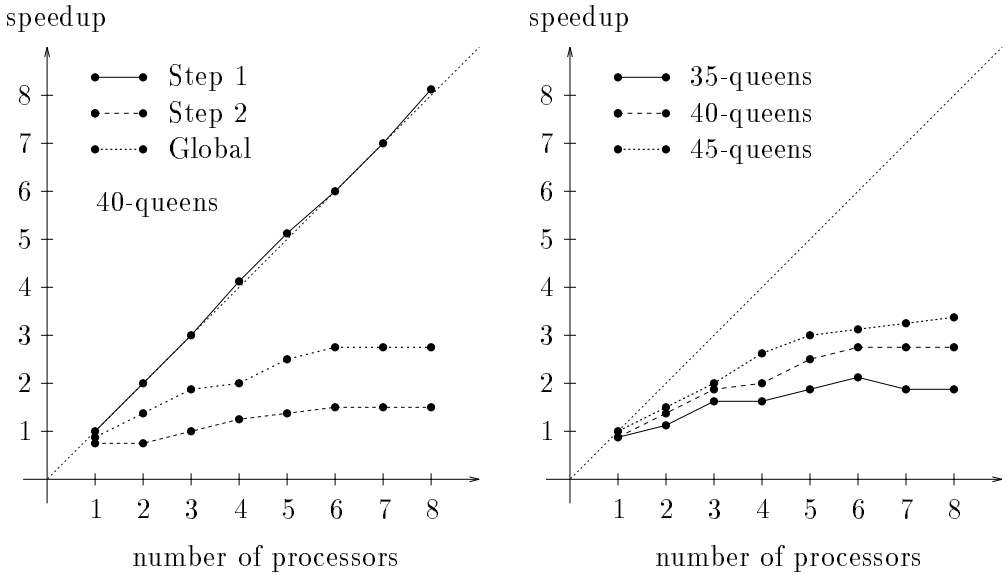

Fig. 14. N-queens benchmark - Speedup vs. number of processors.

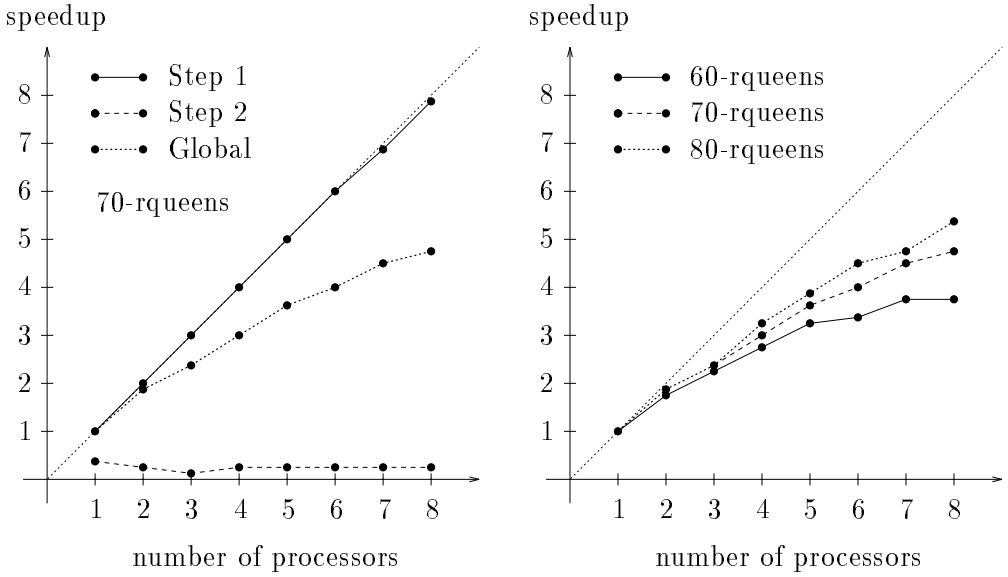

Fig. 15. Reverse N-queens benchmark - Speedup vs. number of processors.

The Step 1 of the algorithm gives speedups close to ideal. The fact that the *Worker* processes run in parallel and independently explains this feature.

In Step 2, the workers have to cooperate and exchange their inconsistent labels. Communications throughout the network are slow with respect to the computing speed of the processors. On the other hand, Step 2 is inherently sequential. Hence, speedups are worse than in Step 1. Note that in the case of reverse N-queens problem, we do not have any speedup but a slowdown. The number of labels which are examined in Step 2 is very low, about 0.01% of

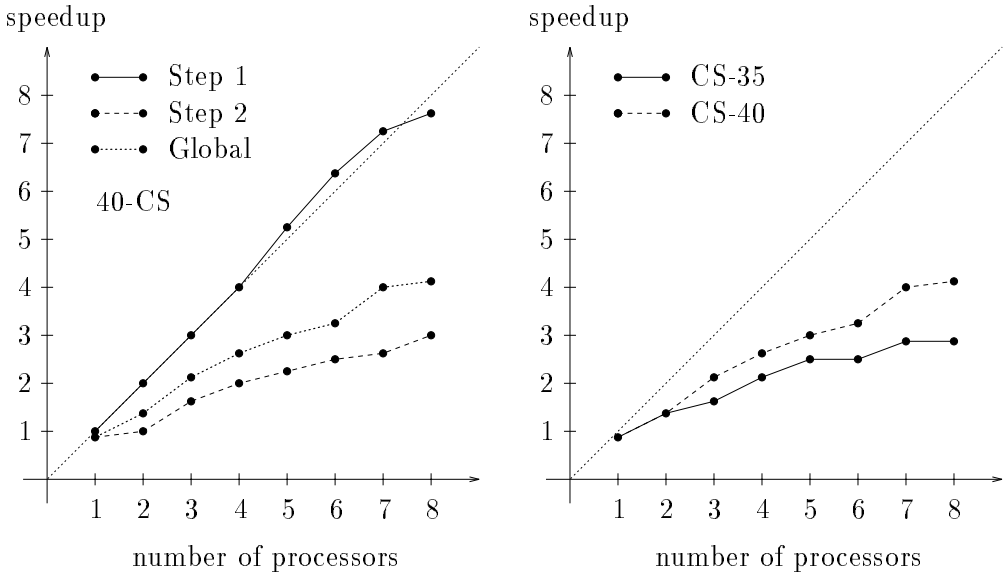

Fig. 16. N-CS benchmark - Speedup vs. number of processors.

the n^2d^2 labels that *Support* sets can theoretically contain. Also the amount of computations is small. Hence, in AC-4, Step 2 is very fast. It is slower in DisAC-4 due to communication overheads. But this slowdown does not completely degrade the global performance of DisAC-4 because time spent in Step 2 is negligible with respect to time spent in Step 1.

With N-queens problem and N-CS benchmark, Step 2 has to examine a large amount of labels contained in *Support* sets. They are treated sequentially in AC-4 and in parallel in DisAC-4. This generates speedup despite communication latencies. Global computation speedups are better whenever the size of the CSPs grows up. With very large sized CSPs, we observed unexpected superlinear effects i.e. speedup greater than the number of processors. This astonishing fact can be explained easily on basis on the following conjecture: whenever the number of processors is small, the amount of necessary memory within each processor is very large and important processing time is spent in swapping. The superlinearity is not inherent to the algorithm. It is due to limitations (memory size of computers) of the computing environment where our experiences were carried out.

In the implementation of DisAC-4, values have to be provided to the constants `max` and `afewtime` within the *CollectList* procedure. In our experiments, the parameter `max` has been set to its minimal value 1. Experimental results showed that other choices do not change the performance. With high values of `max`, the *Controller* process does busy waiting (line 4 **while** loop of the *CollectList* procedure) instead of being waiting for messages at line 8 of the *Polling* function.

The parameter `afewtime` has been set to 1 second. Experimental results showed that variations of this value (1 second + δ, where δ equals 1 or 2 seconds for instance) only extend the total computation time by about δ. The only impact of δ is thus the time between the two last pollings.

Various experiments have been made for different values of `max` and `afewtime`. The resulting number of polling requests generated by the controller was always low (less than 10), even with `afewtime` set to zero. Hence, the hypothesis of Theorem 4 are fulfilled. We also observed that the number of pollings did not affect the global performance of the algorithm.

7 Conclusion

We have presented a coarse-grained parallelization of AC-4 algorithm for a distributed memory computer whose processors communicate by asynchronous message passing. The DisAC-4 algorithm has been designed under the hypothesis of a full connection between the processors, i.e. each processor may directly communicate with another. We prove the termination and the correctness of the algorithm, and stated its complexity. The DisAC-4 algorithm has also been implemented with PVM tools, and experimented on workstations connected by an Ethernet network. The theoretical complexity of DisAC-4 is $O(n^2d^2/k)$ and the experimental results corroborate the expected linear speedup with respect to the number of processors.

We are currently investigating improvements of our DisAC-4 algorithm. We are examining the possibility of sharing domain D_i data structures. The non sharing of these data structures could generate more computations in DisAC-4 with respect to AC-4. It is due to the fact that in Step 1 of DisAC-4, whenever some worker updates its associated domains D_i, it does not directly propagate these modifications to the other workers. Hence, they do not take them into account building their *Support* sets. Nevertheless, we are aware that mechanisms allowing sharing domain data structures could generate overheads which would ruin the potential speedup. We are also considering other topologies than full connection topology for processor connection.

We are studying DisAC-4 implementation on parallel mainframes built for intensive computing (HP PA-RISC 9000 nodes connected by an FDDI network, Convex Exemplar supercomputer).

In an other hand, we are also working on DisAC-3 and DisAC-6 algorithms. They are respectively the distributed versions of AC-3 and Bessiere's [2] AC-6 algorithms. With respect to DisAC-4, they are less memory demanding. Comparisons between DisAC-3, DisAC-4 and DisAC-6 are currently in progress.

We believe DisAC algorithms are a suitable and efficient parallel way to achieve arc-consistency on distributed memory computers, as found on very common hardwares such as networks of workstations and/or PCs. As arc-consistency has important applications in constraint logic programming over finite domain [14,15], we intend to investigate how a parallel arc-consistency algorithm could be part of a parallel constraint logic programming language for finite domains.

Acknowledgments

The authors thank the anonymous reviewers for their helpful comments and suggestions on earlier versions of this text.

References

[1] G. R. Andrews, *Concurrent Programming: Principles and Practice* (The Benjamin/Cummings Publishing Company, Redwood City, CA, 1991).

[2] C. Bessière, Arc-consistency and arc-consistency again, *Artif. Intell.* **65** (1994) 179–190.

[3] P. R. Cooper and M. J. Swain, Arc consistency: parallelism and domain dependence, *Artif. Intell.* **58** (1992) 207–235.

[4] Z. Collin, R. Dechter and S. Katz, On the Feasibility of Distributed Constraint Satisfaction, in: *Proceedings IJCAI-91* (1991) 318–324.

[5] S. Kasif, On the parallel complexity of discrete relaxation in constraint satisfaction networks, *Artif. Intell.* **45** (1990) 275–286.

[6] A. K. Mackworth, Consistency in networks of relations, *Artif. Intell.* **8** (1977) 99–118.

[7] A. K. Mackworth and E. C. Freuder, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artif. Intell.* **25** (1985) 65–74.

[8] R. Mohr and T. C. Henderson, Arc and path consistency revisited, *Artif. Intell.* **28** (1986) 225–233.

[9] B. A. Nadel, Constraint satisfaction algorithms, *Comp. Intell.* **5** (1989) 188–224.

[10] F. Rossi, C. Petrie and V. Dhar, On the Equivalence of Constraint Satisfaction Problems, in: *Proceedings ECAI-90* (1990) 550–556.

[11] Oak Ridge National Laboratory, *PVM 3 User's guide and reference manual* (Oak Ridge, TN, 1993).

[12] A. Samal and T. C. Henderson, Parallel Consistent Labeling Algorithms, *Int. Journal of Paral. Prog.* **16** (1987) 341–364.

[13] E. Tsang, *Foundations of Constraint Satisfaction* (Academic Press, London, UK, 1993).

[14] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming* (The MIT Press, Cambridge, MA, 1989).

[15] P. Van Hentenryck, Y. Deville and C.-M. Teng, A generic arc-consistency algorithm and its specializations, *Artif. Intell.* **57** (1992) 291–321.

[16] D. Waltz, Understanding line drawings of scenes with shadows, in: P. H. Winston, ed., *The Psychology of Computer Vision* (MacGraw-Hill, New York, NY, 1975) 19–91.

[17] M. Yokoo, Constraint relaxation in distributed constraint satisfaction problems, in: *Proceedings of Int. Conf. on Tools with AI - 93* (1993) 56–63.

[18] M. Yokoo, E. H. Durfee, T. Ishida and K. Kuwabara, Distributed constraint satisfaction for formalization distributed problem solving, in: *Proceedings of Int. Conf. Dist. Syst. - 92* (1992) 614–621.

[19] Y. Zhang and A. K. Mackworth, Parallel and distributed algorithms for finite constraint satisfaction problems, in: *Proceedings of IEEE Symp. Paral. and Dist. Proc. - 91* (1991) 394–397.

[20] Y. Zhang and A. K. Mackworth, Parallel and Distributed Finite Constraint Satisfaction, *Technical Report 92-30* (Department of Computer Science, University of British Columbia, Vancouver, B. C. Canada, 1992).