Chaotic Iteration for Distributed Constraint Propagation

Eric Monfroy, Jean-Hugues Réty

CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
E-mail: `eric@cwi.nl, rety@cwi.nl`
Web: http://www.cwi.nl/~eric, http://www.cwi.nl/~rety

Keywords: distributed constraint propagation, chaotic iteration, distributed constraint satisfaction problem

Abstract

We propose a generic framework for distributed constraint propagation based on the notion of chaotic iteration. Our algorithm applies to distributed constraint satisfaction problems, and also leads to significant speed-ups on distributions of constraint satisfaction problems.

1 Introduction

Constraint propagation is one of the most important techniques for finding solutions to sets of constraints. Constraint propagation algorithms usually aim at reaching some *local consistency* by gradually reducing a Constraint Satisfaction Problem (CSP) to another one that is equivalent but simpler. These algorithms are generally designed for specific constraint domains and a generic approach was proposed in [1, 2] based on the notion of chaotic iteration, a general technique used for computing limits of iterations of finite sets of functions.

In this paper, we present a generic framework for distributed constraint propagation. Our motivations are threefold. First, Distributed Constraint Satisfaction Problems have recently appeared as a challenging framework [13, 10] with many applications in Distributed Artificial Intelligence. Second, splitting a CSP and solving it on a distributed system may result in significant speed-ups due to parallel computation. Finally, we are convinced that our framework is suitable for the description of cooperation between constraint propagation over different domains [3], and/or different constraint propagation techniques over a single domain.

Most of the studies on distributed constraint satisfaction aim at defining distributed backtracking-based search algorithms [4, 13, 10]. These studies do not tackle the question of distributed constraint propagation. However, efficient sequential constraint solvers usually combine constraint propagation and search algorithms. One can reasonably think that this is also true for distributed solvers.

We propose a generic algorithm for distributed constraint propagation that is independent from search algorithms, and thus, can be combined with the previously cited works. Our framework is generic: 1) it can be instantiated for propagation on a particular domain by specifying reduction functions for this domain, and 2) it can be instantiated with numerous strategies. The algorithm we propose is based on the Chaotic Iteration algorithm (CI) [1, 2]. We first generalize the CI algorithm, and then propose a distributed version (namely the Distributed Chaotic Iteration algorithm, DCI in short). The correctness of the DCI algorithm and its implicit termination on finite domains is proven in [7]. We realized a prototype implementation for solving systems of non-linear polynomial equations based on interval arithmetic and box-consistency [12]. Experimental results show significant speed-ups due to distribution of work among several computing agents and gradual communication of refined results among these agents.

The paper is organized as follows. In Section 2, we recall and generalize the chaotic iteration framework first proposed in [1]. Section 3 describes DCI, our distributed version of the (generalized) CI algorithm. The prototype we have realized as well as some experimental results are described in Section 4. Finally, we conclude, and discuss related work and future work in Section 5.

2 Chaotic Iteration for Constraint Propagation

Our formalization of the constraint propagation framework follows [1]. By a constraint satisfaction problem (in short CSP), we mean a sequence of domains together with a set of constraints. In our framework, domains of variables are elements of an algebraic structure called *join partial order*, and we abstract from constraints to focus on the domain reduction process. To this end, we assume a set of *domain reduction functions* (in short drf) deduced from the set of constraints. drfs are functions reducing domains of variables. Constraint propagation gradually proceeds by applying drfs.

2.1 Preliminaries

Definition 2.1 *A partial order $\langle D, \sqsubseteq\rangle$ is a $\sqcup$-po (join partial order) if*

- *$\langle D, \sqsubseteq\rangle$ has a least element $\perp \in D$.*
- *for all increasing sequences $d_0 \sqsubseteq d_1 \sqsubseteq d_2 \ldots$ of elements of D, the least upper bound $\sqcup_{i\geq 0} d_i$ exists.*

- *for all $a, b \in D$, the least upper bound $a \sqcup b$ exists.*

A partial order $\langle D, \sqsubseteq\rangle$ satisfies the *finite chain property* if all increasing sequences $d_0 \sqsubseteq d_1 \sqsubseteq d_2 ...$ of elements of D satisfy: $\exists j \geq 0$ such that $\forall i \geq j, d_i = d_j$. Clearly, the cartesian product of a finite sequence of $\sqcup$-po's is a $\sqcup$-po and, the cartesian product of a finite sequence of $\sqcup$-po's satisfying the finite chain property is a $\sqcup$-po satisfying the finite chain property.

Let us consider a finite sequence of pairwise distinct variables V and a sequence of $\sqcup$-po's $(\langle D_v, \sqsubseteq_v\rangle)_{v\in V}$. We note $\langle D, \sqsubseteq\rangle$ the cartesian product of the elements of the sequence $(\langle D_v, \sqsubseteq_v\rangle)_{v\in V}$.

Definition 2.2 *A* scheme on *V is a sequence of pairwise distinct elements of V.*

For sake of simplicity, we sometimes use set predicates on schemes. In particular, if s, s' are schemes, $s \subseteq s'$ means that all elements of s are in s'.

Let s be a scheme on V, we note $\langle D_s, \sqsubseteq_s\rangle$ the cartesian product of the $\sqcup$-po's $(\langle D_v, \sqsubseteq_v\rangle)_{v\in s}$. We note $\perp_s$ the least element of D_s. Since this does not generate confusion, we will often omit the index s on $\sqsubseteq_s$.

We also need a projection operator. Let us consider $d \in D_s$, s being a scheme on V. For each variable $v \in s$, $d[v] \in D_v$ is the projection of d on v. Projections are extended to schemes as follows: let s, s' be two schemes on V, and $d \in D_s$. Then, $d[s']$ is the element of $D_{s'}$ that satisfies $\forall v \in s'$, $d[s'][v] = d[v]$ if $v \in s$, and $d[s'][v] = \perp_v$ if $v \notin s$

Definition 2.3 *A* domain reduction function *(in short drf) on $(V, \langle D, \sqsubseteq\rangle)$ is a triple (is, os, f) where is and os are schemes on V (respectively the* input scheme *and* output scheme *of f) and f is a monotonic function $f : D_{is} \to D_{os}$.*

We will sometimes denote the drf (is, os, f) by f. If $d \notin D_{is}$ then $f(d)$ denotes $f(d[is])$.

Definition 2.4 *A* problem *is a triple $(V, \langle D, \sqsubseteq\rangle, F)$ where V is a finite sequence of pairwise distinct variables, $\langle D, \sqsubseteq\rangle$ is the cartesian product of a sequence of $\sqcup$-po's $(\langle D_v, \sqsubseteq_v\rangle)_{v\in V}$ and F is a finite set of drfs on $(V, \langle D, \sqsubseteq\rangle)$.*

This notion of *problem* formalizes the standard notion of CSP at a more abstract level, dedicated to the description of constraint propagation using domains reduction functions. Let us consider a CSP $P = \langle Dom_1, .., Dom_n; C\rangle$ where the Dom_i's are the domains of variables and C is a set of constraints. For $i \in [1, n]$, consider the partial order $\sqsubseteq_i$ defined by: $\forall X, Y \subseteq Dom_i, X \sqsubseteq_i Y$ iff $X \supseteq Y$. Then, we have: for $i \in [1, n]$, $\langle P(Dom_i), \sqsubseteq_i\rangle$ is a $\sqcup$-po with $\perp_i = Dom_i$ and $X \sqcup_i Y = X \cap Y$.

Depending on the constraint domain and the propagation to achieve, a set of domain reduction functions is derived from the set of constraints. The set of constraints itself is no longer needed: constraint propagation proceeds by applying drfs to the domains.

Let us consider, for instance, arc consistency over a binary constraint $c \subseteq Dom_{v1} \times Dom_{v2}$. The two following drfs are sufficient for reaching arc consistency: $drf_1 = (v1.v2, v1, f_1(X, Y) = X')$, where $X' = \{a \in X \mid \exists b \in Y, (a, b) \in c\}$, and $drf_2 = (v1.v2, v2, f_2(X, Y) = Y')$, where $Y' = \{b \in Y \mid \exists a \in X, (a, b) \in c\}$. This leads to the problem $(v_1.v_2, \langle P(Dom_{v_1}), \supseteq\rangle \times \langle P(Dom_{v_2}), \supseteq\rangle, \{drf_1, drf_2\})$. Enforcing arc consistency on c is achieved by applying these two drfs until a fixed point is reached.

2.2 The CI algorithm

Let us consider a problem $P = (V, \langle D, \sqsubseteq\rangle, F)$ to be reduced. The Chaotic Iteration algorithm provides a general framework for computing the limit of the application of the functions in F. The version of the CI algorithm we propose in Figure 1 generalizes the original algorithm [1] according to the following points:

- In [1], drfs have equal input and output schemes. We extend the algorithm to non-endomorphic functions.
- In [1], functions are supposed to be monotonic and inflationary. We no longer require inflationarity: whenever a function is applied, our algorithm considers the least upper bound of the result of the function, and the current value. Thus, $f(d) \sqcup d$ still is inflationary.
- We define the update of G (the set of candidate drfs to be applied) as a generic function (see Figure 2 for an example). Instead of giving a fixed update, we just specify the minimal requirements the *update* function must satisfy for the algorithm to be correct. This allows our algorithm to be generic with respect to (sound) strategies.

funct $CI(P = (V, \langle D, \sqsubseteq\rangle, F))$
  $d := \perp_V$
  $G := F$
  while $G \neq \emptyset$ **do**
    $(is, os, g) := select(G)$
    $G := G \setminus \{(is, os, g)\}$
    $d' := d \sqcup g(d[is])[V]$
    $G := update(P, G, d, d')$
    $d := d'$
  od
  return(d)

Figure 1: The CI algorithm

The *select* function must satisfy: $select(G) \in G$. The *update* function must satisfy the following properties:

1. $d = d' \Rightarrow update(P, G, d, d') \subseteq G$
2. $\forall (is_f, os_f, f) \in F \setminus update(P, G, d, d')$,
   $f(d'[is_f]) \sqsubseteq d'[os_f]$, or
   $f(d'[is_f]) \sqsubseteq (\sqcup_{g\in update(P,G,d,d')} g(d')[V])[os_f]$

Condition 1 ensures termination: when no change has occurred, no function is added to G. Thus, if a fixed point is reached, G eventually becomes empty. Condition 2 ensures correctness: each function in $F \setminus G$ either cannot modify the domains of the variables, or is subsumed by some functions in G. This latter point allows us to take into account weaker functions as in [6].

A strategy is then defined by a specific *update* function together with a specific *select* function (see Section 4 for an instantiation of the *select* and *update* functions to specify the standard max-domain strategy). Note that some strategies may require the algorithm to be extended with some extra data structures (such as the history of applied drfs, or some notion of cost associated to drfs).

```
funct update(P = (V, ⟨D, ⊑⟩, F), G, d, d')
  foreach v ∈ V do
      if d[v] ≠ d'[v] then
         foreach (is_f, os_f, f) ∈ F do
             if v ∈ is_f then G := G ∪ {(is_f, os_f, f)} fi
         od
      fi
  od
  return(G)
```

Figure 2: A particular *update* function

2.3 Correctness and termination

We now study partial correctness and termination of the CI algorithm.

Definition 2.5 *An execution of the CI algorithm is said to be* fair *if:*
if $g \in G$ holds at some iteration i, then there exists an iteration j with $j > i$ such that $g \notin G$ holds.

The following proposition directly comes from [1] [1]. The proofs need only little modification to be adapted to our framework. We refer the interested reader to [1].

Proposition 2.1

1. *Every terminating execution of the CI algorithm computes in d the least fixed point of the function h on D defined by:*

$$h(x) := \bigsqcup_{f \in F} (f(x)[V] \sqcup x)$$

2. *If all $\langle D_v, \sqsubseteq_v \rangle$, $v \in V$ satisfy the finite chain property, then every execution of the CI algorithm terminates.*

3. *If some fair execution of the CI algorithm terminates then all fair executions terminate.*

Item 2 is particularly interesting for constraint propagation: it ensures termination for finite domains.

3 Distributed Chaotic Iteration

We propose in this section a distributed version of the CI algorithm. Informally, we consider several agents executing CI algorithms extended with communication features. Locally, each agent manages a subset of drfs and a subset of variables (local variables), and communicates/consumes information to/from other agents via asynchronous message passing. Globally, the system still computes the limit of the application of the entire set of drfs. All agents are homogeneous and execute the same algorithm.

We first define a *n-distribution* of a problem to be a partition of its variables and its functions satisfying some independence property: the input scheme of each drf must be composed of local variables only. This allows drfs to be locally computed since they only need local resources. On the other hand, variables in output schemes can still belong to other agents: their domains are eventually transmitted.

[1] To be more precise, Items 1 and 2 correspond to Theorem 2.6 and Item 3 to Theorem 2.7 in [1].

3.1 n-Distribution

Consider a problem $P = (V, \langle D, \sqsubseteq \rangle, F)$.

Definition 3.1 *A* n-distribution *of $P = (V, \langle D, \sqsubseteq \rangle, F)$ is a set of schemes on V $\{V_1, .., V_n\}$ and a set of sets of drfs $\{F_1, ..., F_n\}$ satisfying:*

- *$\{V_1, ..., V_n\}$ is a partition of V.*
- *$\{F_1, ..., F_n\}$ is a partition of F.*
- *$\forall i \in [1, n]$, $\forall (is, os, f) \in F_i$, $is \subseteq V_i$*

Consider a n-distribution $\{V_1, .., V_n\}$, $\{F_1, ..., F_n\}$ of a problem P. Then, for $i \in [1, n]$, a variable v is said to be *external* to i if v is not a variable of V_i, and v belongs to the output scheme of some drf of F_i. More formally:

Definition 3.2 *For all $i \in [i, n]$, the set Ext_i of variables external to i is defined by:*

$$Ext_i = \{v \in V \setminus V_i \mid \exists (is, os, f) \in F_i, v \in os\}$$

The global set of external variables is: $Ext = \bigcup_{i \in [1,n]} Ext_i$.

Note that for $i \in [1, n]$, $P_i = (V_i \cup Ext_i, \langle D_{V_i \cup Ext_i}, \sqsubseteq \rangle, F_i)$ is a problem by itself.

A natural way of distributing a CSP consists in imposing a partition of its constraints, or of its variables. Given a n-partition of the drfs (or of the variables), every problem can be turned into an "equivalent" n-distribution by duplicating variables and extending functions to handle communication (local drfs are extended to produce information on duplicated variables). In [7], we give a method for distributing problems, and we prove an equivalence result between a problem and its distribution.

3.2 The algorithm

Since several drfs belonging to different agents may produce information on the same external variable, we consider for each external variable $v \in Ext$ a many to one communication channel ch_v. A message on the channel ch_v is an element of D_v, and communication is asynchronous. The $send(ch, d)$ (respectively $d := read(ch)$) primitive sends the domain d on the channel ch (respectively reads the domain d on the channel ch). We suppose messages to be delivered in finite time. Our algorithm does not require the order of messages to be preserved.

The Distributed Chaotic Iteration Algorithm is given Figure 3. The parameters $V, \langle D, \sqsubseteq \rangle, F, V_{in}, V_{out}$ of the DCI function must satisfy:

1. $V_{in} \subseteq V$, $V_{out} \subseteq V$, $V_{in} \cap V_{out} = \emptyset$,
2. $\langle D, \sqsubseteq \rangle$ is the cartesian product of the $\sqcup$-po's $\langle D_v, \sqsubseteq_v \rangle$ for $v \in V$,
3. and each $(is, os, f) \in F$ satisfies $is \subseteq V \setminus V_{out}$ and $os \subseteq V$.

The *update* function must satisfy:

1. $d = d' \Rightarrow update(P, G, d, d') \subseteq G$
2. $\forall (is_f, os_f, f) \in F \setminus update(P, G, d, d')$,
   $f(d'[is_f]) \sqsubseteq d'[os_f]$, or
   $f(d'[is_f]) \sqsubseteq (\sqcup_{g \in update(P,G,d,d')} g(d')[V])[os_f]$

```
funct DCI(P = (V, ⟨D, ⊑⟩, F), V_in, V_out)
  d := ⊥_V
  G := F
  V_send := ∅
  while true do
        sel := select(G, V_in, V_send)
        if (sel = (is, os, g)) ∈ G
           then G := G \ (is, os, g)
                e := g(d[is])
                d' := d ∪ e[V]
                foreach v ∈ os ∩ V_out do
                        if d[v] ≠ d'[v]
                           then V_send := V_send ∪ {v}
                        fi
                od
                G := update(P, G, d, d')
                d := d'
        fi
        if sel ∈ V_in
           then e := read(ch_sel)
                d' := d ⊔ e[V]
                G := update(P, G, d, d')
                d := d'
        fi
        if sel ∈ V_send
           then V_send := V_send \ {sel}
                send(ch_sel, d[sel])
        fi
  od
  return(d)
```

Figure 3: The DCI algorithm

And the *select* function must satisfy: $select(G, V_{in}, V_{send}) \in G \cup V_{in} \cup V_{send}$, and if $v = select(G, V_{in}, V_{send}) \in V_{in}$, then there is a message waiting on the channel ch_v. The *select* function is a blocking function: if $G = \emptyset$, $V_{send} = \emptyset$, and for all $v \in V_{in}$ there is no message on the channels ch_v, then $select(G, V_{in}, V_{send})$ blocks until some message arrives on at least one of these channels.

An agent thus runs a local problem and can receive/send domains from/to other agents. V_{send} contains external variables to which locally produced information is transmitted. The *select* function is extended to *send* and *read*. This allows our algorithm to be generic with respect to strategies: *send* and *read* can either be immediately executed or postponed.

3.3 Distributed computation with the DCI algorithm

A *DCI agent* is an agent that locally executes the DCI function of Figure 3. A *DCI system* for a n-distribution $V_1, .., V_n$ and $F_1, ..., F_n$ of a problem $P = (V, \langle D, \sqsubseteq \rangle, F)$ is composed of n DCI agents $a_1, \ldots, a_n$ such that for all $i \in [1, n]$, a_i runs the local problem $P_i = (V_i \cup Ext_i, \langle D_{V_i \cup Ext_i}, \sqsubseteq \rangle, F_i)$. Thus, each agent a_i computes $DCI(P_i, Ext \cap V_i, Ext_i)$. As stated in Proposition 3.1, the DCI system computes the same result as $CI(P)$.

We don't address in this paper the question of detection of termination. We consider below implicit termination [2]: an execution of a DCI system terminates if all the DCI agents it is composed of are blocked on their *select* functions and no message is currently transmitted.

Definition 3.3 *An execution of a DCI agent is said to be* fair *if:*

- *if $g \in G$ holds at some iteration i, then there exists an iteration j with $j > i$ such that $g \notin G$ holds,*
- *if $v \in V_{send}$ holds at some iteration i, then there exists an iteration j with $j > i$ where v is selected,*
- *and every incoming message is eventually read.*

Definition 3.4 *An execution of a DCI system is said to be* fair *if all the executions of the DCI agents it is composed of are fair.*

Proposition 3.1

1. *Every terminating execution of a DCI system computes in $\sqcup_{i \in [1,n]} d_i[V]$ the least fixed point of the function h on D defined by [3]:*

$$h(x) := \bigsqcup_{f \in F} (f(x)[V] \sqcup x)$$

2. *If for all $v \in V$, $\langle D_v, \sqsubseteq_v \rangle$ satisfies the finite chain property, then every execution of the DCI system terminates.*
3. *If some fair execution of a DCI system terminates then all fair executions terminate.*

Proofs of Proposition 3.1 are given in [7]. Just as for Proposition 2.1, Item 2 ensures termination for constraint propagation over finite domains.

4 Implementation and experimental results

4.1 Implementation

Our implementation of the DCI algorithm is a distributed extension of an implementation of the generalized CI algorithm for solving non-linear polynomial systems over real numbers [6] using interval propagation. Based on interval arithmetic and interval analysis [8], the interval version of Newton's method aims at enclosing zeros of interval functions by intervals of real numbers. Newton's method is used to obtain box-consistency [12], an approximation of arc-consistency for real numbers.

Assume a CSP $\langle D, C \rangle$ such that C is a set of polynomial multivariate equalities, and D is a sequence of real intervals (one for each variable of C). Then, for each variable v of each constraint $c \in C$ we consider a drf that enforces box-consistency of c w.r.t. v. Applying the CI algorithm to this set of drfs and to D returns a set D' of reduced domains such that the CSP $\langle D', C \rangle$ is "smaller" than $\langle D, C \rangle$ and is box-consistent.

In the following, we refer to the sequential implementation of the CI algorithm as S_Int, to the DCI system as D_Int, and to one of its agents as d_Int. The implementations of D_Int, d_Int, and S_Int are written in Maple [5]. d_Int is an extension of S_Int with communication features. D_Int consists of several independent d_Int processes that

[2] Explicit termination in this framework can be detected using termination detection algorithms (see for instance [11]).

[3] For $i \in [1, n]$, d_i is the current value of the variable d of agent a_i.

communicate through channels using *send* and *read* functions (implementations of the read and send primitives of Algorithm 3).

The strategy of a d_Int process is an extension of the standard "max-dom" strategy (selection of the drf that reduces the variable with the current largest domain). The strategy gives greater importance to communication and tries to maximize data-exchanges among agents:

1. if V_{send} is not empty, then select and apply the send function related to the variable with the largest domain.

2. if V_{send} is empty, then select a drf w.r.t. max-dom; for each shared variable of the input scheme of this drf, apply the necessary read function; then, apply the drf.

The *update* function of a d_Int agent is a refinement of the *update* function presented in Figure 2 that takes into account idempotency of the drfs (whenever an idempotent drf is successfully applied, it is not added to G) and does not consider V but only the output schemes of the drfs.

4.2 Experimental results

We tested our prototype implementation using various traditional benchmarks from interval propagation [12]. Given a benchmark, the problem is distributed among n well balanced sub-problems: the sharing out of the computing task among the n d_Int processes is as fair as possible, the number of variables is equitable, but we do not try to minimize/maximize the number of shared variables. For a given number n of processes, each problem was tested several times over several n-distributions. Thus, we give average results.

pb	*c*	*v*	*drf*	1	2	3	4	5	6	7	8	9	10
*i*1	10	10	40	0	8	10	13	16	15	15	15	14	14
*i*2	20	20	80	0	16	–	32	28	–	–	–	–	22
*i*3	20	20	80	0	16	–	22	28	–	–	–	–	22
*i*4	10	10	40	0	8	–	–	16	–	–	–	–	14
*i*5	10	10	60	0	8	–	13	16	–	–	–	–	14
*bb*4	4	4	13	0	3	7	8	–	–	–	–	–	–
*bb*8	8	8	40	0	5	–	12	–	–	–	23	–	–

Figure 4: Static properties of problems

Figure 4 shows some "static" properties of the problems. Columns *pb*, *c*, *v*, and *drf* represent the name of the problem, the number of constraints, the number of variables, and the number of drfs respectively. The other columns represent the average number (rounded to the nearest integer) of output channels per d_Int process when the problem is distributed among 1, 2, ...,10 agents (i.e.,the average number of sharing of variables per process, knowing that a variable can be shared between more than 2 processes). We did not test all n-distributions, because for some n, either there is no fair n-distribution, most of possible n-distributions are not fair, or the local problems become too small for the granularity of D_Int.

Figure 5 shows the speed-ups we obtain by distributing CSPs. For a given problem p, a curve represents the ratio c_1/c_2 w.r.t. the number of d_Int agents where:

- c_1 is the average cpu time of the worst d_Int process used in D_Int for solving p,

- c_2 is the cpu time of S_Int for solving p.

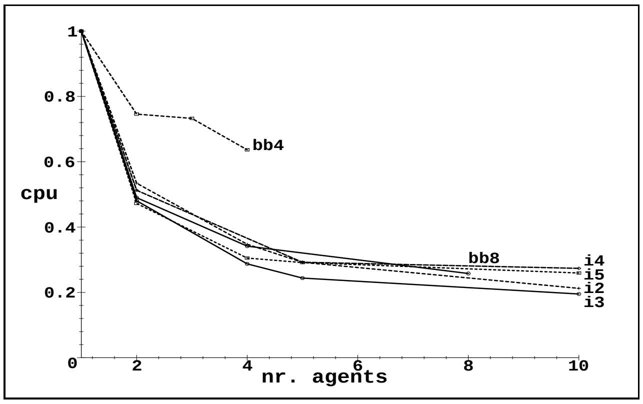

Figure 5: Speed-ups of DCI w.r.t. the number of agents

The speed-up is rather constant w.r.t. the number of agents independently from the problem. The main reason is that each distribution used for these tests is well-balanced.

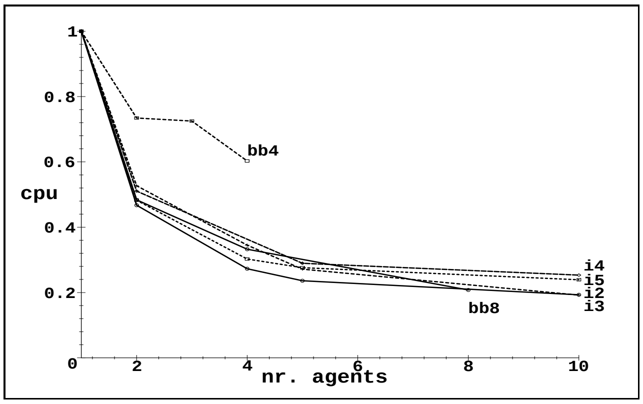

Figure 6: Decreasing of the calculation time

Figure 6 is analogous to Figure 5, but the cpu time is replaced by the calculation time, i.e., the cpu time used by the drfs only. The speed-up for calculation time follows the progression of the speed up of the cpu time. Thus, we can conclude that the ratio between calculation and communication is a constant for an average of fair distributions.

This is more obvious on Figure 7 where the speed up of cpu time (solid line with crosses) and the speed up of calculation time (dash line with boxes) are represented on the same schema. Figure 7 illustrates computations for the problem *i*1. Notice that distribution is made among 1 to 10 processes. However, only few distributions among 3,4,6,7,8, and 9 processes are well-balanced, and thus, the cpu average of the worst process is computed with few data.

Although the ratio between calculation and computation is a constant in average, some well-balanced distributions give some better speed-ups. Let us take the *kin*2 problem [12] as an example. An average on well-balanced distributions (between 2 processes) gives a cpu time speed-up of 0.59, with an average number of output channels per processes of 6. But with a well-balanced distribution that minimizes the number of output channels (3 per processes), we obtain a speed-up of 0.36, i.e.,the cpu time of the worst d_Int is nearly a third of the cpu time of the sequential execution. We observe here a very good cooperation between processes: the global strategy happens to be very efficient. In this case,

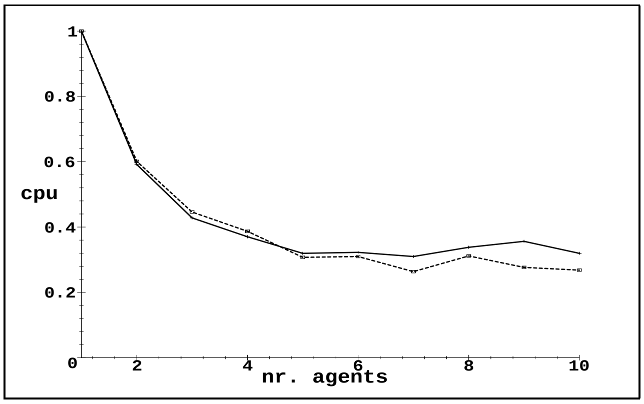

Figure 7: Cpu and calculation times

minimizing the number of shared variables is beneficial. We observed that speed-ups depend on a lot of parameters such as problems density, sharing of variables, computing tasks, etc. However, searching for the "best" well-balanced distribution is out of the scope of this paper.

5 Conclusion

In this paper, we proposed a distributed framework based on the notion of chaotic iteration. Our framework is generic in the sense that it can be instantiated by a particular constraint propagation method (i.e., specification of the drfs and of the computation domain), and by specific strategies (i.e., scheduling of applications of drfs and of communications). Our algorithm is proved to be correct. Using our distributed prototype implementation, we observed significant speed-ups on distributions of constraint satisfaction problems.

In [14, 15], Zhang and Mackworth study the parallelization and distribution of the AC-3 algorithm for constraint propagation. They propose a version with one constraint only per node, and a coarse-grain version for a particular class of constraint networks. Their distributed algorithm is a special case of our DCI algorithm.

A distributed version of AC-4 [9] (an algorithm for constraint propagation) has been proposed by Nguyen and Deville. However, their algorithm can hardly be compared with DCI since it is not based on domain reduction functions.

Our results meet two of our motivations: solving DCSPs, and efficiently solving CSPs with distributed computation. Concerning our third motivation, we are convinced that our framework is well suited for cooperation of constraint propagation over different domains, or cooperation between different constraint propagation techniques. Cast functions (converting a variable from one type to another, e.g. truncation of a real number to get an integer) can naturally be formalized as drfs, and thus, can create bridges between constraint propagation over different domains.

We plan to study how local strategies (scheduling of drfs and communications in a process) can be combined in order to get a global constraint propagation strategy. We have observed on the Example kin2 of Section 4 that a distributed strategy can be very efficient.

Acknowledgements

We are indebted to K. R. Apt for many fruitful discussions.

References

[1] K.R. Apt. From Chaotic Iteration to Constraint Propagation. In *Proc. of 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, pages 36–55. Springer-Verlag, LNCS 1256, 1997. Invited lecture.

[2] K.R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 1998. In Press.

[3] F. Benhamou and W. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 32(1):1–24, July 1997.

[4] Z. Collin, R. Dechter, and S. Katz. On the Feasibility of Distributed Constraint Satisfaction. In *Proc. of IJCAI'91*, pages 318–324, 1991.

[5] K. Geddes, G. Gonnet, and B. Leong. *Maple V : Language reference manual.* Springer Verlag, New York, Berlin, Paris, 1991.

[6] E. Monfroy. Using "Weaker" Functions for Constraint Propagation over Real Numbers. In *Proc. of the ACM Symposium on Applied Computing (SAC'99), San Antonio, Texas, U.S.A.* ACM Press, 1999.

[7] E. Monfroy and J.-H. Réty. A Distributed Chaotic Iteration Algorithm. In *Proc. of the Ercim/Compulog Workshop on Constraints, Amsterdam, The Netherlands*, 1998. Available at http://www.cwi.nl/~eric/ERCIM/Workshops/Workshop3/Program/index.html.

[8] R. E. Moore. *Interval Analysis.* Series in Automatic Computation. Prentice Hall, Englewood Cliffs, N. J., 1966.

[9] T. Nguyen and Y. Deville. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming*, 30(1-2):227–250, 1998.

[10] G. Solotorevsky and E. Gudes. Solving a Real-life Time Tabling and Transportation Problem Using Distributed CSP Techniques. In *AAAI'97 Workshop on Constraints and Agents*, pages 142–147, 1997.

[11] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[12] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a modeling language for global optimization.* The MIT Press, 1997.

[13] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. on Knowledge and Data Engineering*, 1998. To appear.

[14] Y. Zhang and A.K. Mackworth. Parallel and Distributed Algorithms for Finite Constraint Satisfaction Problems. In *Proc. of SPDP'91*, pages 394–397, 1991.

[15] Y. Zhang and A.K. Mackworth. Parallel and Distributed Finite Constraint Satisfaction: Complexity, Algorithms and Experiments. Technical Report 92-30, Dept. of Computer Science, University of British Columbia, Vancouver, Canada, 1992.