Journal of Discrete Mathematical Sciences & Cryptography
ISSN 0972-0529 (Print), ISSN 2169-0065 (Online)
Vol. 29 (2026), No. 3, pp. 1365–1379
DOI : 10.47974/JDMSC-2583

An asynchronous, message-efficient, distributed algorithm for enforcing partial consistency in information networks

Hera Antonopoulou *
Dimitris Papadopoulos †
Department of Management Science and Technology
University of Patras
Patras 26334
Greece

Yannis C. Stamatiou §
Department of Business Administration
University of Patras
Patras 26504
Greece

and

Computer Technology Institute and Press - Diophantus
Patras 26504
Greece

Abstract

In this work we present a totally asynchronous distributed algorithm for the problem of achieving arc-consistency in a network of constraints. The algorithm is designed for multicomputer systems. Since the number and size of messages are two factors of major concern when solving a problem distributively we have paid particular attention to designing the algorithm in a way that tends to reduce their number and size. Our approach relies on identifying whether a message can be suppressed without affecting the operation of the

* *ORCID-ID:* 0000-0002-3936-435X
† *ORCID-ID:* 0000-0002-0725-3954
§ *ORCID-ID:* 0000-0002-8925-9427

* *E-mail* :hera@upatras.gr (Corresponding Author)
† *E-mail* :dimfpap@upatras.gr
§ *E-mail* :stamatiu@upatras.gr

© Taru Publications

process to which it was addressed. This has the effect of keeping the number of exchanged messages low at a cost of some small extra computation time.

Subject Classification: *68W15.*

Keywords: *Constraint satisfaction, Arc-consistency, Parallel computation, Asynchronous algorithms, Distributed computation, Multiprocessors.*

1. Introduction

The *Constraint Satisfaction Problem* (or CSP for short) consists of a set of *variables* to be labeled by an element of a set of *permissible labels* as well as a set of *relations* that locally constrain the combinations of permissible label assignments. The aim is to find the label assignments for the variables (if they exist) that satisfy all the constraints simultaneously. The importance of this problem stems from its applicability to many diverse disciplines of computer science. It is easy to see that this problem is *NP-complete*, i.e. computationally intractable, since many NP-complete problems, such as graph colorability and 3-SAT, can be restated as CSP problems (see [20, 21, 22] - see, also, [1, 2, 5] for complexity related properties of such problems and [3, 4] for some mathematical results related to the algorithmic analysis of these problems). On the other hand, many structured subclasses are tractable (see, e.g., [8]).

A method that is frequently employed to reduce the search space of a CSP is to enforce some form of *local consistency* by deleting labels that cannot participate to any solution. One such method is to enforce *arc-consistency*, where labels that are compatible with *no* labels of *at least* one variable are discarded since it is clear that they cannot be part of a solution.

The computational complexity of arc-consistency has been extensively studied. Although it can be solved by efficient sequential algorithms (see, for example, [11 12, 13, 14, 24]), it is a difficult problem to parallelize. In [16] it is proved that arc-consistency is a P-complete problem (see [15]) by giving a logarithmic space reduction from the problem of Horn-formula satisfiability. However, for special cases there have been found efficient algorithms (see [17], [18]). Even for general arc-consistency, several good, albeit formally not *efficient*, parallelizations have been reported in the literature (see, for example, [25],[10],[8]). For distributed solutions that apply to *join-trees* of constraint networks, see [28].

In this paper we examine the solution of arc-consistency *distributively*. We do not assume that the constraint network is pre-processed in any way. The proposed algorithm is totally asynchronous and can be implemented on multicomputer systems. Due to the ever-increasing demand for fast computation with existing resources, the idea of using a network of computers for the solution of computationally demanding tasks seems attractive. In such an environment, however, there is a shift of interest from the computational complexity of the tasks that are assigned for execution to a computer and the communication cost that is incurred by the distances that separate them. In many cases, the cost of a distributed computation is overwhelmed by the communication cost that is necessitated by a distributed algorithm. Another crucial factor that affects the performance of distributed algorithms significantly, is the cost that is incurred by

the synchronization of the distributed processes. A distributed algorithm should minimize the effect of these factors on its performance. In this paper, we give an algorithm that is totally asynchronous and, at the same time, attempts to reduce the communication traffic among the distributed processes.

2. Statement of the problem

A binary *Constraint Satisfaction Problem* consists of a set V of n variables, $v_1, v_2, \dots, v_n$ that assume labels from the sets $D_1, D_2, \dots, D_n$ respectively (called the *domains*), and a set $\mathcal{C} = \{C_{i,j} | C_{i,j} \subseteq D_i \times D_j, 1 \leq i < j \leq n\}$ where each $C_{i,j}$ defines the set of permissible pairs of labels from v_i and v_j. The tuple $Cn = \{V, D_1, \dots, D_n, \mathcal{C}\}$ is called a *constraint network*. A *solution* of a binary CSP is an n-tuple of labels assigned to the variables $v_1, v_2, \dots, v_n$ that satisfy the binary constraints.

A constraint network Cn is arc-consistent if for every variable v_i, every label $d \in D_i$, and every neighbor $v_j \in Adj(i)$, there exists $d' \in D_j$ with $(d, d') \in C_{i,j}$. The *arc-consistency problem* (or ACP for short) is the problem of finding a *maximal*, with respect to the domains, arc-consistent subnetwork Cn_{AC} that is arc-consistent. (A subnetwork of Cn is a constraint network Cn' with the same variable set V, with domains $D'_i \subseteq D_i, i \in \{1, \dots, n\}$, and constraints $C'_{i,j} \subseteq C_{i,j}, 1 \leq i < j \leq n$.)

It will be helpful if we consider a constraint network as an n-partite graph, where partitions correspond to variables and nodes correspond to labels. In this context, an edge of this graph connects two nodes if the labels they represent are permitted by the binary constraint that constrains the corresponding variables. From now on, whenever we speak of a constraint network, we refer to this particular representation.

3. The model of computation

Our model of computation is the *fixed-connection network model*. The processing elements are interconnected into a *fixed* interconnection topology in the sense that the set of processing elements with which a processing element communicates *directly* (in one step) remains fixed throughout the computation. Each processing element is equipped with a small *fixed* number of bidirectional communication channels, i.e. a *bounded-degree network*, and possesses a constant amount of local memory for storing data and programs. Also, we do not assume that the processing elements are globally clocked.

A number of processes operate concurrently in each processing element, each performing a specific task. The data exchange between processes that are located in different processors is permitted exclusively through message passing and not through shared memory. Furthermore, it is assumed that a process known as the *communication daemon* is responsible for the routing of messages in each processing element, which is responsible for the communication between non-local processes. This process is dedicated to directing incoming messages that are intended for other processors to the appropriate output channel in order to reach the appropriate processes. This ensures that the message reaches its

destination after finitely many forwarding steps. The communication may be either *synchronous* or *asynchronous*. We assume that the messages received by the communication daemon of the receiving process are stored in a designated location in local memory, the *mailbox*, in the case where two processes communicate asynchronously. The process can subsequently query the mailbox as required in order to retrieve its messages from other processes. Whenever the state of the local computation dictates, the computational process on a processor may request from the daemon to forward to it the next available message of a specific type, specified by the process. If no message of this type is present, the computation process is not suspended. Instead, it is resumed, with the potential to perform an alternative action (if permitted by the implemented algorithm), as the requested message has not yet been received (or, in some cases, has not been sent at all). Last, we do not mandate that the processing elements be globally clocked. As a result, we necessitate the existence of a suitable communication protocol between neighboring communication daemons for synchronization.

The proposed algorithm is purely asynchronous. At no point of the computation is synchronization necessary. This has the effect of a large overlap between computation and communication.

4. The distributed algorithm

We now describe the distributed algorithm. Central to its operation is the notion of *support* (see [24]). Below, we state our model assumptions:

Assumption 4.1: (Model and Runtime Assumptions).

- Processes are asynchronous.
- Messages are reliable and eventually delivered (no loss, no duplication).
- FIFO per link is not required.
- Execution is fair: every continuously enabled action occurs infinitely often.
- Domains only shrink (no insertions). Constraints are undirected and static.
- Each variable is owned by exactly one processor (the processor shares form a partition of the variable set).
- Global termination is detected by a standard distributed termination detection scheme (e.g., Matternâ€™s algorithm).

We say that a label $d \in D_i$ (the domain of variable v_i) receives support from label $d' \in D_j$ (the domain of variable v_j) if $(d, d') \in C_{i,j}$, i.e. the label pair (d, d') is allowed by the corresponding binary constraint. As long as the label d receives support from at least one label of *every* other variable, it is considered a *valid* candidate for assignment to v_i. If d has no support from a variable, then d cannot be part of a solution and should be discarded. To eliminate all such labels, we iteratively delete labels that have lost support in (at least) one neighborâ€™s domain. The process continues until no further deletions occur, yielding the maximal arc-consistent subnetwork.

The central idea behind any parallelization of this iterative scheme is to effect label deletions in parallel. In our approach, we first partition the variables of the given constraint network equally among the available processing elements. However, if a label is discarded from a domain of a variable, then all the labels of the rest of the variables that are supported by the discarded label should be informed. This is where communication is introduced. It is desirable to dispense with process synchronization in order not to limit the attainable parallelism. The distributed algorithm below achieves both goals: it is totally asynchronous and it sends a notification only when a remote label loses its *last* support from the deleting variable (criticality filtering).

Representation: A constraint network is represented by two matrices:

- *The constraint matrix*. This is an $n \times n$ matrix whose (i, j) entry is 1 if the variables v_i and v_j are explicitly constrained. If all pairs of labels from v_i and v_j are permitted, then the entry (i, j) is 0. This lets each processor know its neighbors.
- *The compatibility matrix*. Let $N_D \sum_{i=1}^{n} |D_i|$ (for uniform domains, $N_D = nd$). This is an $N_D \times N_D$ 0–1 matrix that represents all binary relations. For any pair of variables v_i, v_j, we identify $C_{i,j} \subseteq D_i \times D_j$ with its 0–1 adjacency block over $D_i \times D_j$, where an entry 1 marks an allowed pair (ℓ, ℓ').

Each processor is assigned an equal number of variables. It receives: *Share* (owned variable indices), *Adj* (for each $i \in$ Share, the list of adjacent variables), *Varmap* (owner of each variable), and the relevant $C_{i,j}$ blocks. We maintain:

- $counter(i, j, l)$ on the *owner of* i: the number of supports for label $l \in D_i$ in D_j (authoritative for labels of v_i).
- $mirror_counter(j, i, l')$ on the *owner of* i: a local mirror of the number of supports in D_i for label $l' \in D_j$, used only to test *criticality* when deleting labels of D_i. Mirrors need not be globally synchronized.
- LocalQueue: the queue of (variable,label) pairs locally scheduled for deletion.

Communication API (final): We use a mailbox-based, asynchronous API:

- PutMessage$(p, i_{\text{src}}, j_{\text{dst}}, l_{\text{dst}})$: enqueue a deletion notice to processor p, meaning: "label l_{dst} of $v_{j_{\text{dst}}}$ has just lost its *last* support from $D_{i_{\text{src}}}$". Messages are buffered per destination and flushed in batches.
- IncomingBatch $:= GetMessage()$: returns one available batch from any sender (or the empty batch if none).
- External(i): TRUE iff v_i is owned by another processor.
- Sizeof$(\cdot)$, Empty$(\cdot)$: obvious.

We note that External(i), Sizeof$(\cdot)$, and Empty$(\cdot)$ are simply utility predicates (not used directly in the pseudocode).

Criticality test: When deleting (i, l), a remote label (j, l') needs a message *iff* it loses its *last* support in D_i:

$$Critical(i, l, j, l') \equiv ((l, l') \in C_{i,j} \wedge mirror_counter(j, i, l') = 1).$$

After removing l, we decrement the mirror to keep future tests correct.

Per-neighbor support counters: For each variable i, neighbor $j \in Adj(i)$, and local label $\ell \in D_i$, let $\text{counter}(i, j, \ell)$ be the number of labels $\ell' \in D_j$ with $(\ell, \ell') \in C_{i,j}$. Symmetrically, for each remote label (j, ℓ') we keep a local mirror $\text{mirror_counter}(j, i, \ell')$. A deletion of (i, ℓ) decrements $\text{mirror_counter}(j, i, \ell')$ for every ℓ' with $(\ell, \ell') \in C_{i,j}$. A remote notification is sent *iff* $\text{mirror_counter}(j, i, \ell')$ becomes 0.

Last, we denote by $D_j^{(0)}$ the snapshot of v_j's initial domain (prior to any deletions).

The algorithm follows below.

Algorithm 1: DAC: Asynchronous, message-efficient distributed arc-consistency

Input: Adjacency lists $Adj(\cdot)$, ownership map owner$(\cdot)$. For every owned variable i and each neighbor $j \in Adj(i)$, processor p receives the compatibility block $C_{i,j}$ and a snapshot of the initial neighbor domain $D_j^{(0)}$. Processor p does not maintain the live contents of D_j (unless it owns j). It relies on mirror counters and incoming deletion notices.

Output: Greatest arc-consistent domains $D_i^{\text{å}} \subseteq D_i$.

1. **Local state at processor p:**
   - For each owned i, arrays D_i and $\text{counter}(i, j, \ell)$ for all $j \in Adj(i), \ell \in D_i$.
   - For each *owned* i and each neighbor $j \in Adj(i)$, mirrors $\text{mirror_counter}(j, i, \ell')$ for all $\ell' \in D_j^{(0)}$.
   - Mailboxes per peer: buffered deletion notices.

Function Init( ):
  foreach *owned* i **do**
    foreach $j \in Adj(i)$ **do**
      foreach $\ell \in D_i$ **do**
        $\text{counter}(i, j, \ell) \leftarrow \left|\{\ell' \in D_j^{(0)} : (\ell, \ell') \in C_{i,j}\}\right|$

  foreach *owned* i **do**
    foreach $j \in Adj(i)$ **do**
      foreach $\ell' \in D_j^{(0)}$**do**
        $\text{mirror_counter}(j, i, \ell') \leftarrow \left|\{\ell \in D_i : (\ell, \ell') \in C_{i,j}\}\right|$

```
        foreach owned i do
            foreach ℓ ∈ D_i do
                if ∃ j ∈ Adj(i): counter(i, j, ℓ) = 0 then
                    enqueue (i, ℓ) into LocalQueue

        foreach neighbor q do
            FlushBufferedMessages(q)

    Function Delete ((i, ℓ));
        remove ℓ from D_i
        foreach j ∈ Adj(i) do
            foreach ℓ′ ∈ D_j^(0) with(ℓ, ℓ′) ∈ C_{i,j} do
                decrement mirror_counter(j, i, ℓ′)
                if mirror_counter(j, i, ℓ′) = 0 then
                    BufferMessage(owner(j), ``last-support lost for (j, ℓ′) due to (i, ℓ)")

    while true do
        if LocalQueue not empty then
            pop (i, ℓ); Delete ((i, ℓ))

        if IncomingBatch available then
            ProcessBatch (IncomingBatch)

        foreach peer q do
            if BufferNonEmpty(q) then
                FlushBufferedMessages(q)
        if local quiescence and Mattern detector says ``no messages in transit" then
            break

    Function ProcessBatch (batch):
        foreach notice ``last-support lost for (j, ℓ′) due to (i, ℓ)" in batch do
            counter(j, i, ℓ′) ← 0
            if counter(j, i, ℓ′) = 0 and ℓ′ ∈ D_j then
                enqueue (j, ℓ′) into LocalQueue
```

Termination detection: We can use Mattern's timestamp-based termination detection algotirhm given in [23] (see, also, [27]) integrated with our mailbox API. Each processor p maintains nonnegative integers S_p and R_p. Whenever a buffered batch of deletion notices is actually *sent* by PutMessage (i.e., the buffer for some destination is flushed and the batch is injected into the network), S_p is incremented by the number of $(\text{variable}, \text{label})$ tuples in that batch. Whenever GetMessage returns a nonempty batch, R_p is incremented by that batch size before the pairs are processed. Local quiescence requires: (i) LocalQueue is empty, (ii) all outgoing buffers are empty (nothing pending to flush), and (iii) the local mailbox has no messages. Global quiescence is declared by Matternâ€™s detector only when every processor is locally quiescent *and* the epoch/color check confirms $\sum_p S_p = \sum_p R_p$ (no messages in transit). The detector is orthogonal to

correctness: it neither reorders nor suppresses deletion notices and therefore cannot cause spurious survival or extra deletion of labels. It merely prevents early termination while application messages are in flight.

Theorem 4.2: (Correctness of DAC). *Under the assumptions in the Model of Computation, DAC terminates and the final domains are exactly the greatest arc-consistent subnetwork below the initial domains.*

Proof: Let $\mathcal{L} = \prod_{i=1}^{n} 2^{D_i}$ be ordered by componentwise inclusion. Define the pruning operator $F: \mathcal{L} \to \mathcal{L}$ by

$$(F(D))(i) = \{\, a \in D_i : \forall j \in Adj(i)\ \exists b \in D_j \text{ with } (a,b) \in C_{i,j} \,\}.$$

Safety. A label $a \in D_i$ is deleted by DAC only when there exists $j \in Adj(i)$ with $\text{counter}(i,j,a) = 0$, i.e., a has no support in D_j. Hence deletions preserve F-consistency with respect to the current domains.

Monotone descent and termination. Domains only shrink and $|D_1| + \cdots + |D_n$ is finite, so DAC terminates.

Greatest post-fixpoint. At termination no enabled deletion remains, so the result $D^{å}$ satisfies $D^* \subseteq F(D^*)$. Thus D^* is a post-fixpoint. Every DAC step is an instance of removing labels not in $F(D)$, thus by standard Tarski/Kleene arguments (see [26, 19] and the note after the end of this proof - see also the constraint-propagation account [6, 7]), the limit of any fair asynchronous schedule equals the greatest post-fixpoint below the initial D, which is the greatest arc-consistent subnetwork.

Moreover, at quiescence every remaining label has a support in every neighbor, hence $F(D^*) = D^*$.

On the use of Tarski/Kleene arguments in the proof of Theorem 4.2.

Let $\mathcal{L} = \prod_{i=1}^{n} 2^{D_i}$ be ordered by componentwise inclusion. The pruning operator $F: \mathcal{L} \to \mathcal{L}$ defined in the proof of Theorem 4.2 is *monotone*: if $D' \subseteq D$ then $F(D') \subseteq F(D)$. By Tarski's fixpoint theorem, the set of fixpoints of a monotone operator on a complete lattice forms a complete lattice, hence there exists a greatest post-fixpoint of F below the initial domains $D^{(0)}$ [26]. Moreover, since $\mathcal{L}$ is finite, iterating F under any fair ("chaotic") schedule of elementary deletions stabilizes in finitely many steps at the same fixpoint; this is the standard Kleene/Tarski convergence principle specialized to finite lattices [19, 6]. In particular, DAC performs only deletions of labels outside $F(D)$, so every fair asynchronous run converges to the greatest arc-consistent subnetwork below $D^{(0)}$, independently of message ordering or batching (see also the constraint-propagation account in [7, Ch. 3]).

5. Theoretical Analysis of Distributed Arc Consistency

In this section we provide estimates for the number of remaining labels per variable as well as the number of exchanged messages among processors.

We summarize the following notation for the parameters of the algorithm:

- n: Number of variables
- d: Domain size of each variable
- p: Arc probability (probability a binary constraint exists between variables)
- t: Constraint tightness (probability that a label pair is allowed)
- $k = p(n-1)$: Expected number of neighbors per variable
- P: Number of available processors ($\leq n$)
- $r = n/P$ (integer division): Number of variables assigned per processor

Theorem 5.1: *Let us assume a constraint network with n variables, each with domain size d, with binary constraints assigned with arc probability p and allowed label pairs, between variables (arcs), assigned independently with probability t (tightness). Then, the expected number of surviving labels per variable, after arc-consistency enforcement, is*

$$\mathbb{E}[surviving\ labels\ per\ variable] = d \cdot [1-(1-t)^d]^k.$$

Proof: Consider a label $a \in D_i$. For it to survive arc-consistency, it must be supported by at least one label in the domain of every neighbor $j \in \mathrm{Adj}(i)$.

For each neighbor j, the probability that none of the d labels in D_j support a is:

$$\Pr[\text{no support from } j] = (1-t)^d.$$

Hence, the probability that a has at least one support in D_j is:

$$S(a,j) = 1-(1-t)^d.$$

Assuming independent supports across neighbors, and that each variable has k expected neighbors, the survival probability of label a is:

$$\Pr[a \text{ survives}] = [1-(1-t)^d]^k.$$

Multiplying by d gives the expected number of surviving labels per variable.

Theorem 5.2: (Inter-processor message bound). *Let $G = (V, E)$ be the undirected constraint graph, with an edge $\{i, j\} \in E$ iff variables v_i and v_j are constrained (the relation $C_{i,j}$ viewed as a block over $D_i \times D_j$). Let $E_\times \subseteq E$ be the set of cross-processor edges whose endpoints are owned by different processors, and let $D_i^{(0)}$ denote the initial domain snapshot of v_i.*

Under the "last-support only" policy of DAC (a deletion notice $i \to j$ is sent only when some $\ell' \in D_j$ loses its last support in D_i), the total number of inter-processor deletion notices satisfies

$$M_{\text{inter}} \leq \sum_{\{i,j\} \in E_\times} (|D_i^{(0)}| + |D_j^{(0)}|) = \sum_{i \in V} |D_i^{(0)}| \deg_\times(i),$$

where $deg_\times(i)$ is the number of cross-processor neighbors of i. In particular, if all domains are of size d, then $M_{inter} \leq 2d\,|E_\times|$.

Proof: We count *inter-processor* deletion notices only. Local (intra-processor) updates are not counted.

Setup and invariants: Fix a cross-processor edge $\{i, j\} \in E_{\times}$. For each label $\ell' \in D_j^{(0)}$, the owner of i maintains the mirror

$$m_{i \leftarrow j}(\ell') \equiv \text{mirror}_{\text{counter}(j,i,\ell')} = \left|\{\ell \in D_i : (\ell, \ell') \in C_{i,j}\}\right| \text{at initialization},$$

i.e., the number of *initial* supports for ℓ' in D_i. The algorithm enforces the following invariants for every $\ell' \in D_j^{(0)}$ throughout the run:

(I1) $m_{i \leftarrow j}(\ell')$ is a nonnegative integer and is *monotonically nonincreasing*.

(I2) Each time a supporting label $\ell \in D_i$ with $(\ell, \ell') \in C_{i,j}$ is *deleted*, $m_{i \leftarrow j}(\ell')$ is decremented by exactly 1.

(I3) The algorithm sends a single notice $i \to j$ for the pair (j, ℓ') *iff* $m_{i \leftarrow j}(\ell')$ hits 0 for the *first* time (the "last-support" trigger). No further notices for (j, ℓ') are ever sent afterwards.

Invariant (I1) holds because deletions never add supports, (I2) follows from the per-deletion loop in Delete that scans only $\ell' \in D_j^{(0)}$ with $(\ell, \ell') \in C_{i,j}$, (I3) is by the BufferMessage guard $m_{i \leftarrow j}(\ell') = 0$ and because the counter never increases afterwards.

Basic facts: We use two simple facts that follow from the code:

(F1) Each local label (i, ℓ) is removed at most once. (It is enqueued on LocalQueue only when some $\text{counter}(i, \cdot, \ell)$ becomes 0, and the Delete handler removes it irrevocably.)

(F2) Each decrement recorded in (I2) corresponds to a *distinct* supporting label $\ell \in D_i$ for the fixed ℓ'. Hence $m_{i \leftarrow j}(\ell')$ can experience at most $m_{i \leftarrow j}(\ell')$ initial many decrements total, with the unique transition $1 \to 0$ occurring at most once.

Per-direction bound on a fixed cross edge: Fix $\{i, j\} \in E_{\times}$. Consider notices sent in direction $i \to j$. For each $\ell' \in D_j^{(0)}$, by (I1)–(I3) and (F2), the "last-support" condition $m_{i \leftarrow j}(\ell')$ hitting 0 can occur *at most once*. When it occurs, exactly *one* notice is sent. If it never occurs, no notice is sent. Therefore,

$$\#\{\text{notices } i \to j \text{ along } \{i, j\}\} \leq |\{\ell' \in D_j^{(0)}\}| = |D_j^{(0)}|.$$

By symmetry, the number of notices from $j \to i$ along the same edge is at most $|D_i^{(0)}|$.

Edge-level and global bounds: Summing the two directions on $\{i, j\}$ gives

$$\#\{\text{notices along } \{i, j\}\} \leq |D_i^{(0)}| + |D_j^{(0)}|.$$

Summing over all cross-processor edges and regrouping by endpoints yields

$$M_{\text{inter}} \leq \sum_{\{i,j\}\in E_\times} (|D_i^{(0)}| + |D_j^{(0)}|) = \sum_{i\in V} |D_i^{(0)}| \deg_\times(i),$$

where $\deg_\times(i)$ is the number of cross-processor neighbors of i. In particular, if $|D_i^{(0)}| = d$ for all i, then $M_{\text{inter}} \leq 2d\,|E_\times|$.

Robustness to asynchrony and batching: The bound is independent of delivery order, batching, or lack of FIFO per link. Messages are *triggered* solely by the 1→0 transition of a local, monotone counter (I1)–(I3). No scheduling policy can create an extra $1\to0$ transition for the same (j,ℓ'), hence no extra notice. Likewise, duplicates cannot be generated because after the first trigger the counter stays at 0 and the guard never re-enables.

Tightness: The bound is tight up to constants: for a fixed cross edge $\{i,j\}$, choose a compatibility block $C_{i,j}$ so that each $\ell' \in D_j^{(0)}$ has *exactly one* supporting $\ell \in D_i$ initially. If all those supporting ℓ–s are eventually deleted, then $m_{i\leftarrow j}(\ell')$ performs the 1→0 transition for every ℓ', producing exactly $|D_j^{(0)}|$ notices in direction $i \to j$ (and symmetrically in the opposite direction).

Remartk 5.3: (Tightness, assumptions, and corollaries). The bound is *worst-case tight* per cross-processor edge: for a fixed $\{i,j\} \in E_\times$, one can realize $|D_j^{(0)}|$ distinct notices $i \to j$ (and symmetrically $|D_i^{(0)}|$ from $j \to i$) by arranging that each $\ell' \in D_j^{(0)}$ has exactly one supporting label in D_i and those supports are deleted at any times or orders–FIFO is not required. No independence or randomness assumptions are used, i.e. the argument is purely combinatorial and holds under arbitrary asynchrony and reordering.

Two convenient corollaries follow immediately. (i) If all domains are uniform of size d, then $M_{\text{inter}} \leq 2d\,|E_\times|$. (ii) If the cross-processor degree is bounded by $\Delta_\times$, then $M_{\text{inter}} \leq \Delta_\times \sum_{i\in V} |D_i^{(0)}|$. Finally, batching or coalescing multiple (j,ℓ') notices into a single network packet can reduce *packets* sent, but not the logical notice count captured by the bound.

6. Discussion and directions for future research

In this paper, we presented a totally asynchronous distributed algorithm for achieving arc-consistency in networks of relations that, also, reduces the size of the message packets. The experimental results, which will be pursued next, are expected to be encouraging. This is attributed to the fact that asynchronicity permits considerable overlap with computation, a thing that affects positively the total execution time.

Some interesting questions that we will be addressed are the following:

- Build a simulation of the algorithm and evaluate its efficiency in practice or implement it on a parallel/distributed computer.
- How do the parameters p (arc probability) and t (tightness) affect the expected speed-up? It would be very interesting to derive expressions that relate these parameters to speed-up. This would enable one to have an estimate of the expected efficiency when solving arc-consistency distributively.
- What is the average time complexity of solving arc-consistency distributively? This, in general, is a difficult question. Distributed algorithms contain a large degree of non-determinism, as far as the exchange of messages is concerned. It is always the case that a message that is delayed in one run reaches its destination quickly at another run. However, if we make some assumptions about the probability distribution of these delays, we should be able to model the time complexity by suitable random processes.
- It would also be interesting to restate the arc-consistency problem in the framework described in [9].

Finally, the proposed distributed algorithm can be studied in the context of other problems such as the following:

- *Distributed Artificial Intelligence (AI) and Multi-Agent Systems:* Modern AI systems often involve multiple agents operating concurrently and making decisions based on shared constraints (e.g., in autonomous vehicles, drone fleets, or smart factories). The proposed algorithm can help maintain partial consistency of shared world models without requiring global synchronization, enabling more scalable and responsive systems.
- *Edge Computing and IoT Coordination:* In Internet of Things (IoT) environments, devices at the edge often need to make decisions based on local constraints while intermittently communicating with other devices. The distributed nature of this algorithm makes it ideal for ensuring constraint satisfaction among cooperating devices (e.g., load balancing in smart grids or traffic coordination in smart cities).
- *Cloud Resource Scheduling and Allocation:*

  Modern cloud infrastructures operate over distributed systems where resources (CPU, memory, storage) must be allocated to competing tasks subject to numerous constraints. Implementing arc-consistency via a message-efficient, asynchronous protocol can optimize this allocation in real time, especially in multi-cloud or hybrid environments.

- *Collaborative Filtering and Recommender Systems:* In recommendation systems, especially distributed ones (like federated learning-based platforms), partial consistency is often sufficient to deliver timely and relevant suggestions. This algorithm could help synchronize user and item

constraints across nodes without introducing latency from global synchronization.

- *Blockchain and Decentralized Computation:* The decentralized nature of blockchain systems, combined with their growing support for smart contracts and complex logic, makes them a fertile ground for constraint reasoning. The algorithm's ability to prune invalid options asynchronously is beneficial in scenarios such as distributed auction systems or decentralized identity management, where decisions are made collectively under local constraints.
- *Cybersecurity and Access Control:* In access control systems, especially those that span multiple administrative domains (e.g., zero trust architectures), enforcing consistency in access policies can be modeled as a CSP. Using a distributed, asynchronous algorithm ensures quick propagation of policy changes or revocations without central bottlenecks.

References

[1] H. Antonopoulou, "A user authentication protocol based on the intractability of the 3-coloring problem", *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 5, no. 1, pp. 17–21 (2002).

[2] H. Antonopoulou, "On threshold properties", *Journal of Discrete Mathematical Sciences and Cryptography*, vol. 7, no. 2, pp. 249–254 (2004).

[3] S. Antonopoulou, Y.C. Stamatiou, and M. Vamvakari, "An asymptotic expansion for the q-binomial series using singularity analysis for generating functions", *Journal of Discrete Mathematical Sciences and Cryptography,* vol. 10, no. 3, pp. 313–328 (2007).

[4] H. Antonopoulou, N. Glinos, and Y.C. Stamatiou, "An identity derived from the solution of a class of differential equations for the evolution of a key agreement protocol", *Journal of Discrete Mathematical Sciences and Cryptography,* vol. 14, no. 6, pp. 515–520 (2011).

[5] H. Antonopoulou, "Kolmogorov complexity based upper bounds for the unsatisfiability threshold of random k-SAT", *Journal of Discrete Mathematical Sciences and Cryptography,* vol. 23, no. 7, pp. 1431–1438 (2020).

[6] K. R. Apt, "The essence of constraint propagation", *Theoretical Computer Science*, vol. 221, pp. 179-210 (1999).

[7] K. R. Apt, Principles of Constraint Programming. Cambridge University Press (2003).

[8] D.A. Cohen, M.C. Cooper, and P.G. Jeavons, "Characterizing tractable constraints", *Artificial Intelligence*, vol. 65, pp. 347–361 (1994).

[9] D. Conforti, L. Grandinetti, R. Musmanno, M. Cannataro, G. Spezzano, and D. Talia, "A model of efficient asynchronous parallel algorithms on multicomputer systems", *Parallel Computing*, vol. 18, pp. 31–45 (1992).

[10] P.R. Cooper and M.J. Swain, "Arc consistency: parallelism and domain dependence", *Artificial Intelligence*, vol. 58, pp. 207–235 (1992).

[11] R. Dechter, "Constraint networks", in Encyclopedia of Artificial Intelligence, Second Edition, S. Shapiro Ed., New York, Wiley, pp. 276–285 (1992).

[12] R. Dechter and I. Meiri, "Experimental evaluation of preprocessing algorithms for constraint satisfaction problems", *Artificial Intelligence*, vol. 68, pp. 211–241 (1994).

[13] E.C. Freuder and D. Sabin, "Contradicting Conventional Wisdom in Constraint Satisfaction", in Proceedings of the Second Workshop on Principles and Practice of Constraint Programming, pp. 10–20 (1994).

[14] P. Van Hentenryck, Y. Deville, and C.-M. Teng, "A generic arc-consistency algorithm and its specializations", *Artificial Intelligence*, vol. 57, pp. 291–321 (1992).

[15] R.M. Karp and V. Ramachandran, "Parallel algorithms for shared-memory machines", in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., Amsterdam: Elsevier (1990).

[16] S. Kasif, "On the parallel complexity of discrete relaxation in constraint satisfaction networks", *Artificial Intelligence*, vol. 45, pp. 275–286 (1990).

[17] S. Kasif and A.L. Delcher, "Local consistency in parallel constraint satisfaction networks", *Artificial Intelligence*, vol. 69, pp. 307–327 (1994).

[18] L.M. Kirousis, "Fast parallel constraint satisfaction", *Artificial Intelligence*, vol. 64, pp. 147–160 (1993).

[19] S. C. Kleene, Introduction to Metamathematics. D. Van Nostrand, Princeton, 1952. Reprinted by North-Holland, Amsterdam, 1952. Dover, New York (2002).

[20] A.K. Mackworth, "Constraint satisfaction", in: Encyclopedia of Artificial Intelligence, S. Shapiro Ed., New York, Wiley, pp. 285–293 (1992).

[21] A.K. Mackworth and E.C. Freuder, "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems", *Artificial Intelligence,* vol. 25, pp. 65–74 (1985).

[22] A.K. Mackworth and E.C. Freuder, "The complexity of constraint satisfaction revisited", *Artificial Intelligence*, vol. 59, pp. 57–62 (1993).

[23] F. Mattern, "Algorithms for Distributed Termination Detection", *Distributed Computing,* vol. 2, pp. 161–175 (1987).

[24] R. Mohr and T.C. Henderson, "Arc and Path Consistency Revisited", *Artificial Intelligence*, vol. 28, pp. 225–233 (1986).

[25] A. Samal and T.C. Henderson, "Parallel consistent labeling algorithms", *International Journal of Parallel Programming*, vol. 16, no. 5, pp. 341–364 (1987).

[26] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications", *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285-309 (1955).

[27] S. Taylor, Parallel Logic Programming Techniques, Prentice-Hall (1989).

[28] Y. Zhang and A.K. Mackworth, "Parallel and distributed algorithms for finite constraint satisfaction problems", in Proceedings of 3rd IEEE Symposium on Parallel and Distributed Processing, Dallas, TX, pp. 394–397 (1991).

Received February, 2025