Towards Distributed Constraint Solving with CRDT

Hakan Hasan
SnT, University of Luxembourg
Luxembourg
hakan.hasan@uni.lu

Pierre Talbot
University of Luxembourg
Luxembourg
pierre.talbot@uni.lu

Abstract

Conflict-free replicated data types (CRDTs) provide a principled foundation for managing shared state under concurrent modification and asynchronous communication. However, their use in general and exact combinatorial optimization remains largely unexplored. In this paper, we build on the fact that the accumulation of global information in distributed constraint optimization can be expressed as a fixpoint computation over monotone shared state, directly aligning constraint solver's monotone global knowledge with the semantic foundations of state-based CRDTs. Building on this insight, we present a distributed constraint solver implementation combining GPU-based constraint solving with an MPI-based realization of CRDT using one-sided communication. Experimental results on multi-GPU and multi-node platforms demonstrate near-linear scaling, validating both the practicality and the semantic robustness of the proposed fixpoint-based coordination model.

CCS Concepts: • **Theory of computation → Distributed computing models**; **Constraint and logic programming**; • **Software and its engineering → Consistency**.

Keywords: constraint programming, CRDT, fixpoint computation, eventual consistency

ACM Reference Format:
Hakan Hasan and Pierre Talbot. 2026. Towards Distributed Constraint Solving with CRDT. In *13th International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3806077.3806699

1 Introduction

Conflict-free replicated data types (CRDTs) provide a principled foundation for managing shared state under concurrent modification and asynchronous communication in distributed systems [20, 21]. By organizing state within a lattice and restricting updates to monotone operations, CRDTs guarantee convergence under asynchronous execution. CRDTs have been successfully applied to storage systems [4] and collaborative applications [9], but their use in combinatorial optimization remains largely unexplored.

Constraint programming (CP) is a declarative paradigm for solving combinatorial decision and optimization problems [11], with applications ranging from scheduling and planning to verification and resource allocation. A CP solver operates by interleaving constraint propagation and search. Propagation derives new information implied by the problem's constraints, while search explores alternative subproblems through branching decisions when propagation alone is insufficient. This process can be understood as a backtracking algorithm in which search organizes these propagation computations into a tree of progressively refined states, and recursive exploration is guided by repeated propagation to stability (Section 2.2).

Most of the key optimisations of constraint solvers were primarily designed in the mental frame of sequential computation. Nevertheless, several parallel search strategies have been proposed. Some approaches parallelize search using a shared queue of nodes among threads [16, 19]. Other approaches aim to minimize coordination by decomposing the problem into many independent subproblems upfront, as in embarrassingly parallel search (EPS) [8, 12]. Parallel portfolio methods explore the same problem in parallel from different angles and are widely used in practice [17, 18]. While effective, these approaches mostly limit information sharing between workers or rely on centralized coordination.

In this paper, we argue that the accumulation of global information in distributed constraint optimization admits a fixpoint characterization over monotone shared state, closely aligned with the semantics of state-based CRDTs. Key forms of global information used in CP—most notably objective bounds and learned nogoods—are inherently monotone. This property makes them natural candidates for CRDT-style replication and asynchronous accumulation. This fixpoint-based view establishes a direct connection between distributed constraint solving and the theory of state-based CRDTs. To the best of our knowledge, this paper is the first to apply CRDT-style reasoning to the coordination of exact constraint optimization.

While CRDTs are not described explicitly in terms of fixpoint computation, their semantic foundations are closely related. Unlike classical fixpoint computations, CRDT state spaces are often unbounded and execution may not terminate; convergence is therefore defined semantically rather than operationally. In this work, we exploit the fact that

PaPoC '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2637-8/2026/04
https://doi.org/10.1145/3806077.3806699

distributed constraint optimization induces a bounded form of monotone information growth, where global knowledge refines toward a stable value of interest, such as an optimal bound. This makes fixpoint reasoning explicit and observable, while remaining fully consistent with CRDT semantics.

Fixpoint semantics has been used previously to design Turbo. Turbo is the first general-purpose constraint programming solver that executes entirely on GPU hardware using a parallel model based on concurrent constraint programming [25, 26]. Turbo's design emphasizes intrinsic parallelism, lock-free execution, and formal correctness. Although the usage of CRDTs are not explicitly referenced, Turbo's design is grounded in the same lattice-theoretic and fixpoint principles that underlie CRDT semantics.

We apply fixpoint-based model to distributed constraint optimization and present a high-performance implementation combining GPU-based constraint programming with MPI-based CRDT implementation. Each worker explores parts of the search space locally and independently, while objective bounds are shared asynchronously using monotone, idempotent updates implemented via one-sided MPI operations. The resulting solver avoids synchronization barriers and centralized control, while preserving correctness. Experiments demonstrate near-linear scaling across multiple GPUs and nodes.

This paper makes the following contributions:

- Investigation of state-based CRDT for distributed constraint solving (Section 4).
- High-performance implementation using MPI showing near-linear scaling (Section 5).
- Establishing a connection between CRDT semantics and prior work on asynchronous iteration.

2 Background

2.1 Lattices and Fixpoints

Let $\langle L, \leq \rangle$ be a partially ordered set (poset). A lattice is a poset in which for every pair of elements $x, y \in L$, both the least upper bound $x \sqcup y$, called their join, and the greatest lower bound $x \sqcap y$, called their meet, exist in L. The join and meet operations are associative, commutative, and idempotent. As a result, joins and meets are well defined independently of evaluation order. A lattice may additionally have a greatest element $\top \in L$ and a least element $\bot \in L$, satisfying $\forall x \in L, x \leq \top$ and $\forall x \in L, \bot \leq x$. A complete lattice is a lattice in which all subsets have a join and a meet.

Let $f \colon L \to L$ be a function over a lattice. The function f is monotone if $x \leq y \Rightarrow f(x) \leq f(y)$ for all $x, y \in L$. It is extensive if $x \leq f(x)$ for all $x \in L$. An element $x \in L$ is a fixpoint of f if $f(x) = x$. A fixpoint x is least if $x \leq y$ for every fixpoint y of f. When least fixpoints exist, they are uniquely determined by the order structure.

Chaotic iteration is a general technique for computing fixpoints of systems of monotone operators without imposing a fixed application order [3]. Let $\langle L, \leq \rangle$ be a complete lattice and let $f_1, \ldots, f_n$ be monotone functions $f_i \colon L \to L$. The combined effect of these operators can be captured by the function $F(x) = f_n \circ \ldots \circ f_1$. Rather than applying $f_1, \ldots, f_n$ iteratively in fixed order, chaotic iteration executes the functions in an arbitrary order. Formally, a chaotic iteration is a sequence $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ such that $x_{k+1} = f_{i_k}(x_k)$ for some $f_{i_k} \in \{f_1, \ldots, f_n\}$ where the choice of operator may vary at each step. Under standard fairness assumptions—namely, that no operator is permanently ignored—such iterations converge to the least fixpoint of F, which represents the state that is stable with respect to all operators in F.

Details on lattice theory can be found in [5].

2.2 Constraint Programming

Constraint programming is a declarative paradigm for solving combinatorial decision and optimization problems. It is particularly well suited to problems involving complex combinatorial structure and heterogeneous constraints. A problem in CP is specified by describing the variables involved, the domains of values they may take, and a set of constraints restricting the combinations of values that the variables can simultaneously take [2, 11].

In the following, we consider constraint programming over integer variables only. Let X be a finite set of variables and C be a finite set of constraints. A constraint network is a pair $P = \langle d, C \rangle$, where $d \colon X \to \mathcal{P}(\mathbb{Z})$ is a domain function. An assignment is a function $asn \colon X \to \mathbb{Z}$, and we denote the set of all assignments by **Asn**. A constraint c defines a relation $rel(c) \subseteq \mathbf{Asn}$, specifying the set of values allowed simultaneously over a subset of variables $X_c \subseteq X$. For example, $rel(x \neq y) = \{a \in \mathbf{Asn} | a(x) \neq a(y)\}$. The set of solutions of a constraint network is:

$$sol(d, C) \triangleq \{asn \in \mathbf{Asn} | \\ \forall c \in C, asn \in rel(c) \wedge \forall x \in X, asn(x) \in d(x)\}$$

The constraint satisfaction problem (CSP) is to find a solution $s \in sol(d, C)$. For example, the constraint network $\langle d, C \rangle$, where $X = \{x, y, z\}$, $d(x) = d(y) = d(z) = \{1, \ldots, 10\}$ and $C = \{x + 3 \leq y, x + 6 \leq z\}$. One possible solution is the assignment $\{x \mapsto 1, y \mapsto 5, z \mapsto 9\}$, since it satisfies both constraints.

Constraint optimization problems (COPs) extend CSPs by introducing an objective variable $z \in X$ whose value is to be minimized (or maximized). A feasible solution is a solution of the underlying CSP, and an optimal solution is one that minimizes the value of z.

Branch-and-bound. Branch-and-bound (B&B) is a systematic search algorithm used to solve optimization problems. A search state is defined by a domain function d' such that $d'(x) \subseteq d(x), \forall x \in X$. The algorithm explores the space of candidate solutions by recursively partitioning the search space into smaller mutually exclusive subproblems and using bounds on the objective variable to prune parts of it that

cannot contain an optimal solution. Let T be a rooted tree whose nodes correspond to subproblems. The root represents the original problem instance (domain function d). At any node $d' \in T$ a branching operator produces a set of subproblems $branch(d') = \{d'_1, d'_2\}$, each representing a refinement of the original search space (by fixing some decision variable or imposing additional constraints). During the search, the solver maintains a current best bound b on the objective variable, representing the smallest value of z found so far in any solution. Bounding is used to prune the search space. For any search state d' if constraint propagation implies that all assignments consistent with d' satisfy the inequality $z \geq b$ then the subproblem represented by d' can be discarded. The B&B algorithm terminates when all subproblems have either been explored or pruned. At that point, the current bound corresponds to the optimal value of the objective variable.

GPU Constraint Programming. The challenge of exploiting massive parallelism in GPUs for constraint solving has attracted recent interest. Turbo is a constraint programming solver that executes propagation and search entirely on GPU hardware using a parallel model based on concurrent constraint programming [26]. Turbo's design emphasizes intrinsic parallelism, lock-free execution, and formal correctness. Subsequent work has extended Turbo's performance by optimizing the representation of constraints for the GPU architecture [25].

2.3 Conflict-Free Replicated Data Types

Distributed systems often rely on replication to improve availability and scalability. In such systems, replicas may be updated independently and communicate asynchronously, without guarantees on message ordering or timing. Conflict-free replicated data types (CRDTs) [20, 21] are a class of replicated data types designed to ensure convergence under these conditions, without requiring synchronization between replicas.

There are two types of CRDTs: state-based and operation-based. In this paper we focus on state-based CRDTs. A state-based CRDT is defined by a tuple $\langle L, \leq, \sqcup, S, value, f_1, \ldots, f_n \rangle$, where $\langle L, \leq \rangle$ is a lattice[1], $\sqcup$ is the join operator of $\langle L, \leq \rangle$, S is a set of values, $f_1, \ldots, f_n$ are monotone and extensive functions over L, $value\colon L \to S$ returns the value modeled by the CRDT.

A replicated data type consists of a set of replicas, each maintaining a local copy of some abstract state. Replicas evolve by applying local updates and by incorporating information received from other replicas. The fundamental correctness property of CRDTs is strong eventual consistency: if two replicas have seen the same set of updates, possibly in different order, they will be in the same state [21].

[1]Formally, we do not require the meet operation $\sqcap$, so it is a join-semilattice, but for brevity we use the term "lattice"

For example, consider the grow-only counter (G-Counter), a simple state-based CRDT, that models a distributed counter that can only be incremented. Let n be the number of replicas. The G-Counter is then given by $G_B = \langle \mathbb{Z}^n, \leq_L, \sqcup, \mathbb{Z}, value, inc_1, \ldots, inc_n \rangle$, where $\langle L, \leq_L \rangle$ is the cartesian product lattice equipped with the componentwise order, $(x_1, \ldots, x_n) \sqcup (y_1, \ldots, y_n) = (\max(x_1, y_1), \ldots, \max(x_n, y_n))$, each replica $i \leq n$ can apply the update $inc_i((x_1, \ldots, x_i, \ldots, x_n)) = (x_1, \ldots, x_i + 1, \ldots, x_n)$, and the observable value is given by $value((x_1, \ldots, x_n)) = \sum_{i=1}^{n} x_i$. Consider three replicas, initially all in state $(0, 0, 0)$. Replica 1 performs two local increments, replica 2 performs three, and replica 3 performs one. As replicas exchange states asynchronously, they repeatedly apply the join operation. Regardless of the order of the local updates and the merges, all replicas eventually converge to the same state $(2, 3, 1)$, whose value is 6.

Each replica maintains a local state $s \in L$. Extensive updates are applied locally, that is, they produce a new state s' such that $s \leq s'$. Replicas periodically exchange their local states and incorporate received information using the join operator. Because the join operator is associative, commutative, and idempotent, repeated or reordered merges do not affect the result. Convergence follows from the fact that all replicas eventually compute the join of the same set of states. The resulting state is the least upper bound of all updates that have been generated.

CRDTs thus provide a semantic foundation for reasoning about asynchronous information sharing without synchronization. By structuring shared state as a lattice and restricting updates to be monotone and commutative, CRDTs ensure convergence independently of execution order. These properties make CRDTs a natural building block for coordination mechanisms in distributed systems, where global information must be accumulated safely under parallel and asynchronous execution.

3 Fixpoint-Based Coordination Model

This section introduces the coordination model underlying our distributed solver. The purpose of the model is to explain how global information can be shared across parallel workers without synchronization, and why this is sufficient for correctness. Rather than prescribing an implementation, the model provides a semantic description of how coordination in the context of constraint solving behaves under asynchronous execution.

3.1 Global Coordination State

In a distributed constraint solver, workers explore the search space independently. During this exploration, workers may discover information that is globally relevant, such as improved objective bounds. Coordination consists in accumulating such information so that it can be used by other workers, for instance to prune their search locally.

To reason about this process abstractly, we represent global coordination information as an element of a lattice $\langle L, \leq \rangle$. We associate the system with a conceptual global state $g \in L$, which represents the total information accumulated so far. This state is conceptual: it is not stored explicitly at a single location. Instead, each worker i maintains a local under-approximation $g^{(i)} \in L$, in the sense that $g^{(i)} \preceq g$. This reflects the information the worker i has received up to that point. Because communication is asynchronous, different workers may hold different approximations of the global state at any time.

Workers perform local computation independently and may generate updates $u \in L$, representing newly discovered global information. Updates are incorporated by applying the join operation, $g \leftarrow g \sqcup u$. Since L is a lattice, $\forall x, u \in L$ we have $x \leq x \sqcup u$. Intuitively, this guarantees that updates only add information and never invalidate previously accumulated knowledge.

This is central to the model. It ensures that global coordination can be expressed purely in terms of accumulating information, without requiring rollback, conflict resolution, or agreement on a common execution order. No assumptions are made about when updates are generated, how often they are communicated, or in what order they are applied.

3.2 Asynchronous Execution and Fixpoint Semantics

We now describe the semantics of the coordination process under asynchronous execution. We consider a system consisting of n workers. Let $U = \{u_1, u_2, \ldots\}$ denote the multiset of all updates generated by all workers during the execution. The intended semantic result of coordination is defined as the least upper bound of all these updates, $g^* = \bigsqcup U$. This definition specifies what the global state should represent once all relevant information has been taken into account, independently of how or when updates are applied.

An actual execution of the system does not compute g^* directly. Instead, updates are applied incrementally and asynchronously. For the purpose of reasoning, for each worker $i \leq n$, such an execution is a sequence $g_0^{(i)} \leq g_1^{(i)} \leq g_2^{(i)} \leq \ldots$, where each step corresponds to incorporating information. That information is either a single local update, $g_{k+1}^{(i)} = g_k^{(i)} \sqcup u_l$ for some update $u_l \in U$, or is the state of another worker, $g_{k+1}^{(i)} = g_k^{(i)} \sqcup g_l^{(j)}$ for a worker $j \leq n$, such that $j \neq i$. This sequence does not imply that the system enforces a global order on updates. Because the join operation is associative, commutative, and idempotent, the order in which updates are incorporated does not affect the final result. Duplicate updates have no effect, and delayed updates merely postpone the refinement of the global state. Any execution that eventually incorporates the same set of updates converges to the same value g^*.

This process admits a fixpoint interpretation. For each $u \in U$, define the monotone, extensive operator $f_u(x) = x \sqcup u$. The value g^* is the least fixpoint of the combined effect of all of operators f_u. Operationally, asynchronous execution corresponds to chaotic iteration of these per-update operators: at each step, some update u_k is incorporated into the local state. Under the assumption that every generated update is eventually incorporated, convergence to the same fixpoint is guaranteed.

From this perspective, asynchrony affects only performance-related aspects, such as the speed of convergence and the amount of redundant work performed by workers. It does not affect the correctness, which follows from the monotonic structure of the global state and its characterization as a fixpoint.

4 CRDT-based Distributed Constraint Solving

We now apply the coordination model to optimization problems. Without loss of generality, we focus on minimization problems, where the objective is to find a feasible assignment minimizing an objective function.

In this setting, the only globally shared information required for correctness is a bound on the objective value. Each feasible solution with objective value c implies that the optimal value is at most c. Such information is monotone: discovering a solution can only improve the bound, and no future computation can invalidate it.

Accordingly, we define a CRDT over a lattice $\langle L, \preceq \rangle$, where $L = \mathbb{Z} \cup \{\infty\} \cup \{-\infty\}$, ordered by the reverse of the usual numeric order, i.e., $x \preceq y$ if $x \geq y$ numerically. The join operation is defined as $x \sqcup y = \min(x, y)$. The bottom element ∞ represents the absence of any known solution, and the top element $-\infty$ represents an unbounded objective (there exists solution for any value of the objective). The set of modeled values by CRDT is $S = L$ and $value(b) = b$.

Each worker i maintains a local bound $b_i \in L$, initialized to ∞. When a worker discovers a feasible solution with cost c, it generates an update c, which is incorporated into the local state by the join operation and the new bound is asynchronously propagated to other workers.

Search and propagation may use stale bounds. A worker operating with a bound b_i larger than the current global bound may explore parts of the search space that would be pruned under a tighter bound. However, because bounds only restrict the search space and never eliminate feasible solutions incorrectly, such stale information affects only efficiency, not correctness.

The semantic global bound is given by $b^* = \sqcup\{c | c$ is the objective value of a feasible solution discovered during execution$\}$. By construction, this value is the least fixpoint of the bound-improvement process described in Section 3.2.

The coordination mechanism we use at the distributed level closely mirrors Turbo's internal mechanism at the GPU

level. In Turbo, the GPU computational units solve the problem concurrently while sharing a global objective bound stored in an atomic variable. Each unit performs updates using a monotone atomic operation. From a semantic perspective, Turbo's in-GPU optimization already realizes a fixpoint computation over a lattice, where the objective bound evolves monotonically under concurrent atomic updates. Our multi-GPU solver applies the same principle at a coarser granularity: independent solver instances exchange and merge bounds asynchronously using monotone one-sided MPI operations. In both cases, correctness follows from the same structural properties—monotonicity, idempotence, and order-independent joins—and asynchrony affects only convergence speed. This correspondence highlights that the proposed coordination model does not introduce a fundamentally new mechanism, but rather generalizes an existing fixpoint-based optimization pattern from intra-GPU parallelism to inter-GPU and inter-node execution.

5 Implementation

This section presents multi-GPU constraint optimization as a concrete case study for the coordination model described in Section 3. The CP solver Turbo focuses on single-machine GPU execution, our work extended Turbo's capabilities to run in multi-GPU and multi-node environment, abstracting bound sharing across distributed memory, and preserving solver correctness under asynchronous communication. The goal is not to optimize for a particular hardware configuration, but to demonstrate that the model naturally supports execution in a highly asynchronous environment, where fine-grained synchronization is impractical.

We consider a distributed execution environment composed of multiple compute nodes. Each node hosts several GPUs and runs one solver instance per GPU. Within each node, one designated process acts as a node leader for coordination across nodes. All other processes on the node communicate exclusively with their local leader.

5.1 Monotone Objective Coordination via One-Sided Communication

The coordination scheme described in Section 3 is realized using one-sided MPI communication [13]. In contrast to point-to-point messaging, one-sided communication allows a process to read or update a memory location exposed by another process without requiring its active participation. Updates are performed atomically and may be combined using predefined reduction operators.

In our implementation, the state of the objective bound is represented as a scalar value stored in a remote-memory-access (RMA) window and accessed exclusively through one-sided monotone accumulation operations. Each solver instance periodically retrieves and improves the bound, while concurrent updates are combined deterministically and atomically by the MPI runtime. No ordering or synchronization is imposed on these operations.

This communication pattern directly matches the lattice-based coordination model introduced in Section 3. Since the bound evolves monotonically, one-sided updates preserve convergence independently of timing or interleavings. The primary advantage of using one-sided MPI is that it eliminates coordination barriers. Solver instances do not need to exchange messages, acknowledge updates, or participate in collective operations.

By restricting global communication to monotone RMA updates, the implementation enforces the semantic separation assumed by the theory: all non-monotone reasoning remains local, while shared state evolves according to a convergence-guaranteed abstraction.

5.2 Gossip Dissemination of the Bound

To support asynchronous inter-node coordination of the global objective bound, we adopt a gossip-style dissemination protocol built on MPI RMA. Gossip protocols are decentralized communication schemes in distributed systems where nodes periodically exchange information with randomly selected peers. Inspired by epidemic algorithms for database replication, gossip enables efficient state propagation with eventual consistency and fault tolerance without centralized coordination [6].

In our multi-node solver, gossip is used to disseminate best bound updates between node leaders. Each node maintains local bound information and periodically engages in a randomized exchange with another leader from a set of known peers, merging received information using a monotone join operation (minimum). This approach ensures that, over repeated interactions, all nodes converge toward the same global best bound without requiring synchronization barriers or ordered communication.

Algorithm 1 Inter-node bound gossip

Require: W_b: shared MPI window for best bound; l: leader rank of the current node

```
1: while isSolving() do
2:   if rank = l then
3:     B_local = fetchAndUpdateMin(W_b, +∞, l);
4:     peer = selectRandomNodeLeader();
5:     B_peer = fetchAndUpdateMin(W_b, B_local, peer);
6:     fetchAndUpdateMin(W_b, B_peer, l);
7:   end if
8: end while
```

Table 1. MiniZinc score comparison across solvers

Solver	MiniZinc score
Turbo (16 GPUs)	30.7
Turbo (8 GPUs)	22.3
Turbo (4 GPUs)	21.9
Turbo (2 GPUs)	19.2
Turbo (1 GPU)	8.1

Table 2. Solvers metrics

Solver	nodes/sec	S_p	E_p
Turbo (16 GPUs)	6174470	14.67	92%
Turbo (8 GPUs)	2625910	6.24	78%
Turbo (4 GPUs)	1384340	3.29	82%
Turbo (2 GPUs)	711093	1.69	84%
Turbo (1 GPU)	420828	1.00	100%

6 Experimental Evaluation

6.1 Experimental Setup

Experiments were conducted on compute nodes, that are each equipped with 4 Tesla V100 SXM20 GPUs.

We compare our proposed multi-GPU solver with different number of GPUs. The evaluation set comprises 16 benchmark instances drawn from the MiniZinc Challenge 2022 [22], representing a diverse set of combinatorial optimization problems. Each run was limited to 300 seconds wall-clock time. [2]

6.2 Results

Table 1 reports the MiniZinc scores obtained by Turbo when executed with an increasing number of GPUs. As expected, increasing parallelism improves the overall solution quality within the fixed time limit.

Table 2 reports solver throughput in explored nodes per second, along with speedup and parallel efficiency relative to the single-GPU configuration. Throughput increases monotonically with the number of GPUs, confirming that the solver effectively exploits additional computational resources.

Because we rely on an EPS, subproblems are assigned to GPUs without dynamic redistribution. As a result, load imbalance may occur when some GPUs complete their assigned work earlier than others. This effect was observed in one benchmark instance, where near termination only a single GPU remained active in the 8- and 16-GPU configurations, while the remaining devices were idle until timeout. For all other instances, GPUs remained uniformly active or the problem was solved before timeout. As reported in Table 2, when the number of GPUs increases, efficiency decreases, which may partly be explained by such imbalance during late stages of the search, when few subproblems remain.

7 Future Work

While previous section focuses on objective bounds as the primary form of globally shared information in constraint optimization, the same fixpoint-based coordination model naturally extends to other forms of monotone global knowledge, most notably learned nogoods.

In constraint programming, a nogood represents a set of assignments that cannot occur in any solution. For example, for the constraint network $\langle d, C\rangle$, where $X = \{x, y, z\}$, $d(x) = d(y) = d(z) = \{1 \dots 10\}$ and $C = \{x + 3 \leq y, x + 6 \leq z\}$. The set of all assignments for which $x \mapsto 4, y \mapsto 6$ is a nogood, since it violates the constraint $x + 3 \leq y$.

Once derived, a nogood remains valid throughout the remainder of the search: future computation may discover additional nogoods, but never invalidate an existing one. Consequently, the set of known nogoods evolves monotonically and can be modeled as an element of a partially ordered set ordered by set inclusion.

Sharing nogoods asynchronously may lead workers to temporarily explore regions of the search space that could have been pruned earlier under a stronger nogood set. However, as with stale objective bounds, this affects only efficiency and not correctness: nogoods restrict the search space but never eliminate valid solutions. The fixpoint semantics therefore provides a unified explanation for the safe accumulation of both objective bounds and learned constraints under asynchronous execution.

8 Related Work

8.1 Lattice-Based Concurrency

Lattice-based structures have emerged in multiple communities as a mathematical foundation for concurrent and distributed computation. The mathematical framework of CRDTs shares strong parallels with shared-memory parallel programming models, such as LVars [10] and PCCP [26]. LVars and PCCP allow threads to communicate through monotone updates to shared state, guaranteeing determinism despite nondeterministic scheduling. The key distinction is that LVars enforces determinism through blocking reads, whereas PCCP allows non-blocking reads, similarly to CRDTs that allow nondeterministic observations of replica state to maximize availability.

8.2 Datalog

Datalog is a declarative logic programming language where programs consist of facts and rules, with evaluation defined as the least fixpoint of the rules application over the facts. This bottom-up evaluation ensures that the algorithm converges to a well-defined set of derived facts.

[2] Replicate: the version of the implementation used in this paper is available at https://github.com/haki-1/turbo/tree/papoc2026.

Consistency And Logical Monotonicity (CALM) is a principle which formalizes the connection between logical monotonicity and the ability to compute consistent outcomes without coordination in asynchronous systems [7]. The CALM principle states that a program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

The insight of the CALM principle has been applied in creation of languages like Bloom [1], which is based on a formal temporal logic called Dedalus, and grounded on Datalog.

A line of work similar to the idea of chaotic iterations also comes from the investigation of distributed Datalog evaluation. The naïve and semi-naïve evaluations in Datalog iteratively applies rules to derive new facts. In a distributed environment, this style of evaluation would require global barriers that significantly constrain parallel progress. To address this, pipelined semi-naïve evaluation is introduced in Network Datalog (NDlog), a relaxed evaluation strategy that processes tuples immediately upon arrival without waiting for global iteration barriers, yet provably produces the same minimal fixpoint results.

In recent years, there has been growing interest in the context of Datalog engines on GPUs [23, 24].

We should investigate more the relationship with Datalog languages and their implementation on GPU.

8.3 Chaotic Iterations in Constraint Programming

The notion of chaotic iteration has been used in the CP literature as a model for asynchronous constraint propagation. This approach reframes constraint propagation as a distributed chaotic iteration process that repeatedly applies domain reduction functions [14, 15].

9 Conclusion

We have presented a fixpoint-based coordination model for parallel and distributed constraint optimization and demonstrated the applicability of this model through a multi-GPU constraint optimization solver. Beyond the specific system presented here, the proposed model highlights a unifying principle underlying propagation, optimization, and coordination in parallel constraint solving. By making fixpoint semantics explicit, it opens the way toward more general coordination mechanisms for sharing monotone global knowledge, such as learned nogoods, across heterogeneous and distributed platforms.

We have presented the foundation of a distributed multi-GPU constraint solver using CRDT. We believe it opens new perspectives for the usage of CRDTs in scientific computations. Moreover, it shows that computing with lattices and fixpoint is a very general idea, useful in many communities.

Acknowledgments

We are grateful to the reviewers for the useful comments. This work is partially funded by the joint research programme UL/SnT–ILNAS on Technical Standardisation for trustworthy and sustainable ICT, Construction and Aerospace.

References

[1] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. *CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings*, 249–260.

[2] Krzysztof Apt. 2003. *Principles of constraint programming*. Cambridge university press.

[3] Gérard M. Baudet. 1978. Asynchronous Iterative Methods for Multiprocessors. *J. ACM* 25, 2 (April 1978), 226–244. doi:10.1145/322063.322067

[4] Cihan Biyikoglu. 2020. *Under the Hood: Redis CRDTs (Conflict-free Replicated Data Types).* White Paper. Redis Labs. https://lp.redislabs.com/rs/915-NFD-128/images/WP-RedisLabs-Redis-Conflict-free-Replicated-Data-Types.pdf

[5] B. A. Davey and H. A. Priestley. 2002. *Introduction to Lattices and Order* (2 ed.). Cambridge University Press.

[6] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. 1–12.

[7] Joseph M. Hellerstein. 2010. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.* 39, 1 (Sept. 2010), 5–19. doi:10.1145/1860702.1860704

[8] Marijn J.H. Heule, Oliver Kullmann, and Victor W. Marek. 2017. Solving Very Hard Problems: Cube-and-Conquer, a Hybrid SAT Solving Method. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 4864–4868. doi:10.24963/ijcai.2017/683

[9] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. 2019. Interleaving Anomalies in Collaborative Text Editors. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2019)*. ACM, Article 6. doi:10.1145/3301419.3323972

[10] Lindsey Kuper, Aaron Turon, Neelakantan R Krishnaswami, and Ryan R Newton. 2014. Freeze after writing: Quasi-deterministic parallel programming with LVars. *ACM SIGPLAN Notices* 49, 1 (2014), 257–270.

[11] Christophe Lecoutre. 2009. Constraint Networks: Techniques and Algorithms.

[12] Arnaud Malapert, Jean-Charles Régin, and Mohamed Rezgui. 2016. Embarrassingly parallel search in constraint programming. *Journal of Artificial Intelligence Research* 57 (2016), 421–464.

[13] Message Passing Interface Forum. 2025. *MPI: A Message-Passing Interface Standard Version 5.0.* https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf

[14] E Monfroy and JH Rety. 1998. A Distributed Chaotic Iteration Algorithm. In *Proc. of the Ercim/Compulog Workshop on Constraints, Amsterdam, The Netherlands*.

[15] Eric Monfroy and Jean-Hugues Rety. 1999. Chaotic iteration for distributed constraint propagation. In *Proceedings of the 1999 ACM symposium on Applied computing*. 19–24.

[16] Laurent Perron. 1999. Search Procedures and Parallelism in Constraint Programming. In *International Conference on Principles and Practice of Constraint Programming*.

[17] Laurent Perron and Frédéric Didier. 2025. *CP-SAT*. Google. https://developers.google.com/optimization/cp/cp_solver/

[18] Charles Prud'homme and Jean-Guillaume Fages. 2022. Choco-solver: A Java library for constraint programming. *Journal of Open Source Software* 7, 78 (2022), 4708. doi:10.21105/joss.04708

[19] Christian Schulte. 2000. Parallel Search Made Simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*. Singapore. https://chschulte.github.io/papers/schulte-trics-2000.html

[20] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report 7506. INRIA. http://hal.inria.fr/inria-00555588/

[21] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[22] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. 2014. The MiniZinc Challenge 2008–2013. *AI Magazine* 35, 2 (2014), 55–60. doi:10.1609/aimag.v35i2.2539

[23] Yihao Sun, Sidharth Kumar, Thomas Gilray, and Kristopher Micinski. 2025. Column-Oriented Datalog on the GPU. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 14 (Apr. 2025), 15177–15185. doi:10.1609/aaai.v39i14.33665

[24] Yihao Sun, Ahmedur Rahman Shovon, Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. 2025. Optimizing Datalog for the GPU. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 762–776. doi:10.1145/3669940.3707274

[25] Pierre Talbot. 2026. A GPU-based Constraint Programming Solver. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 14331–14341. doi:10.1609/aaai.v40i17.38448

[26] Pierre Talbot, Frédéric G Pinel, and Pascal Bouvry. 2022. A Variant of Concurrent Constraint Programming on GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 3830–3839.