Towards Distributed Constraint Solving with CRDT

Hakan Hasan
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg

Pierre Talbot
University of Luxembourg, Luxembourg

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.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Constraint and logic programming; Software and its engineering → Consistency

Keywords and phrases constraint programming, CRDT, fixpoint computation, eventual consistency

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Constraint programming (CP) is a declarative paradigm for solving combinatorial decision and optimization problems [8], with applications ranging from scheduling and planning to verification and resource allocation. Many of these problem domains are often inherently NP-hard, presenting significant computational demand. However, current CP solvers do not take advantage of distributed computations, leaving large-scale solving largely unexplored.

Several parallel search strategies have been proposed with the aim of improving the efficiency of the solvers. Some approaches parallelize search using a shared queue of nodes among threads [11, 14]. Other approaches aim to minimize coordination by decomposing the problem into many independent subproblems upfront, as in embarrassingly parallel search (EPS) [9, 6]. Parallel portfolio methods explore the same problem in parallel from different angles and are widely used in practice [13, 12]. While effective, these approaches mostly limit information sharing between workers or rely on centralized coordination.

Conflict-free replicated data types (CRDTs) provide a principled foundation for managing shared state under concurrent modification and asynchronous communication in distributed systems [16, 15]. By organizing state within a lattice and restricting updates to monotone operations, CRDTs guarantee convergence under asynchronous execution and have been successfully applied to storage systems [2] as well as collaborative applications [7].

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.

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 [19, 18]. 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 [4].

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 [8, 1].

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 $\mathbf{Asn}$. 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 $s \in sol(d, C)$. 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 (or maximizes) the value of z.

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 [19]. Turbo's design emphasizes intrinsic parallelism, lock-free execution, and formal correctness. Subsequent work has improved Turbo's performance by optimizing the representation of constraints for the GPU architecture [18].

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) [16, 15] 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 [16].

[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. 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

We present a coordination model for our distributed solver, providing a semantic framework for sharing global information across parallel workers without requiring synchronization. This model abstracts the behavior of coordination in asynchronous constraint solving rather than prescribing an implementation.

3.1 Global Coordination State

In a distributed constraint solver, workers explore the search space to find feasible or optimal solutions. During this process, they 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 $\langle L, \leq \rangle$, with a conceptual global state $g \in L$. Each worker i maintains a local under-approximation $g^{(i)} \in L$, in the sense that $g^{(i)} \preceq g$, reflecting the information it has received. Workers perform local computations independently and generate updates $u \in L$, which are incorporated via the join $g \leftarrow g \sqcup u$. Lattice properties ensure $x \leq x \sqcup u$ for all $x, u \in L$, so updates monotonically accumulate information without invalidating prior knowledge. No assumptions are made about communication timing or order.

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 incorporates either a local update, $g_{k+1}^{(i)} = g_k^{(i)} \sqcup u_l$ for some update $u_l \in U$, or another worker's state, $g_{k+1}^{(i)} = g_k^{(i)} \sqcup g_l^{(j)}$ for a worker $j \leq n$, such that $j \neq i$. Associativity, commutativity, and idempotence of $\sqcup$ ensure that the order of incorporation does not affect convergence: any execution eventually incorporating all updates reaches g^*.

This process admits a fixpoint interpretation. For each $u \in U$, define the monotone, extensive operator $f_u(x) = x \sqcup u$. Then g^* is the least fixpoint of the combined effect of all of operators f_u, and asynchronous execution corresponds to chaotic iteration of these 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

This section presents the design of our distributed constraint solver. It covers the coordination model for sharing global information across workers, and the decomposition strategy, which organizes the search space into independent subproblems for execution across GPUs.

4.1 Coordination

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.

We formalize this using 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. Workers may operate with stale bounds, which can lead to exploring parts of the search space that would be pruned under tighter bounds. However, since bounds restrict rather than eliminate solutions, stale information affects only efficiency, not correctness.

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.

4.2 Decomposition Strategy

Embarrassingly parallel search (EPS) is a parallelization strategy in which the search space is decomposed into a large number of independent subproblems that can be solved concurrently without exchanging search states [9]. A key motivation for EPS is the highly irregular and instance-dependent cost of solving individual subproblems, which makes static partitioning ineffective and favors over-decomposition relative to the number of available workers.

We adopt an on-demand variant of embarrassingly parallel search, similar in spirit to the GPU-level decomposition used in Turbo [19, 18], but applied at the level of independent solver instances across GPUs. Unlike classical EPS, which relies on an explicit pool of pre-generated subproblems and a centralized scheduler [9], independent subproblems in our solver are generated incrementally during execution and assigned to GPUs as they become available. Once assigned, a subproblem is solved entirely locally on a GPU using Turbo, without sharing propagation states or partial search trees with other devices.

5 Implementation

This section presents multi-GPU constraint optimization as a concrete case study for the coordination model described in Section 3. The solver Turbo targets single-GPU execution, we extend it to run in multi-GPU and multi-node environment, abstracting bound sharing across distributed memory, and preserving solver correctness under asynchronous communication.

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 [10]. Unlike 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 atomic 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 [5].

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

```
while isSolving() do
  if rank = l then
    B_local = fetchAndUpdateMin(W_b, +∞, l);
    peer = selectRandomNodeLeader();
    B_peer = fetchAndUpdateMin(W_b, B_local, peer);
    fetchAndUpdateMin(W_b, B_peer, l);
  end if
end while
```

6 Experimental Evaluation

Experiments were conducted on compute nodes equipped with 4 Tesla V100 SXM20 GPUs each. We evaluate our multi-GPU solver with different number of GPUs on 16 benchmark instances drawn from the MiniZinc Challenge 2022 [17], representing a diverse set of combinatorial optimization problems. Each run was limited to 300 seconds wall-clock time.[2]

Table 1 reports the MiniZinc scores obtained by Turbo when executed with an increasing number of GPUs and solver throughput in explored nodes per second, along with speedup

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

Table 1 Solvers metrics

Solver	MiniZinc score	nodes/sec	S_p	E_p
Turbo (16 GPUs)	30.7	6174470	14.67	92%
Turbo (8 GPUs)	22.3	2625910	6.24	78%
Turbo (4 GPUs)	21.9	1384340	3.29	82%
Turbo (2 GPUs)	19.2	711093	1.69	84%
Turbo (1 GPU)	8.1	420828	1.00	100%

and parallel efficiency relative to the single-GPU configuration. As expected, increasing parallelism improves the overall solution quality within the fixed time limit and 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 1, 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 CP, a nogood is a set of assignments that cannot occur in any solution. Once derived, a nogood remains valid: future computation may generate more, but never invalidate existing ones. Hence, the set of known nogoods evolves monotonically and can be modeled as a partially ordered set under inclusion. Asynchronous sharing of nogoods may cause temporary exploration of regions that could have been pruned, but, like stale objective bounds, this affects only efficiency, not correctness. Fixpoint semantics thus provides a unified framework for safely accumulating both objective bounds and learned constraints asynchronously.

8 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 believe this work 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.

References

1 Krzysztof Apt. *Principles of constraint programming*. Cambridge university press, 2003.

2 Cihan Biyikoglu. Under the hood: Redis crdts (conflict-free replicated data types). White paper, Redis Labs, 2020. URL: https://lp.redislabs.com/rs/915-NFD-128/images/WP-RedisLabs-Redis-Conflict-free-Replicated-Data-Types.pdf.

3 D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969. URL: https://www.sciencedirect.com/science/article/pii/0024379569900287, doi:10.1016/0024-3795(69)90028-7.

4 B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2 edition, 2002.

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

6 Marijn J.H. Heule, Oliver Kullmann, and Victor W. Marek. 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*, pages 4864–4868, 2017. doi:10.24963/ijcai.2017/683.

7 Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Interleaving anomalies in collaborative text editors. In *6th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2019. ACM, March 2019. doi:10.1145/3301419.3323972.

8 Christophe Lecoutre. Constraint networks: Techniques and algorithms. 2009.

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

10 Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*, June 2025. URL: https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf.

11 Laurent Perron. Search procedures and parallelism in constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, 1999.

12 Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

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

14 Christian Schulte. Parallel search made simple. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, Singapore, September 2000. URL: https://chschulte.github.io/papers/schulte-trics-2000.html.

15 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report 7506, INRIA, January 2011. URL: http://hal.inria.fr/inria-00555588/.

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

17 Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, 2014. URL: https://onlinelibrary.wiley.com/doi/abs/10.1609/aimag.v35i2.2539, doi:10.1609/aimag.v35i2.2539.

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

19 Pierre Talbot, Frédéric G Pinel, and Pascal Bouvry. A variant of concurrent constraint programming on gpu. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3830–3839, 2022.