Towards Distributed Constraint Solving with CRDT

Hakan Hasan Pierre Talbot

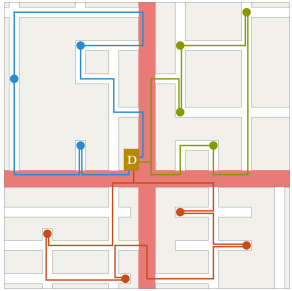
Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
hakan.hasan@uni.lu, pierre.talbot@uni.lu

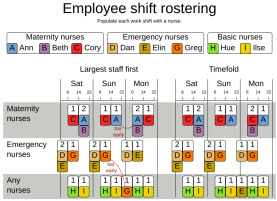
Workshop on Principles and Practice of Consistency for Distributed Data
27 April 2026

Constraint Programming

Constraint programming (CP) is about modeling and solving problems under constraints.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9





CP Modeling

australia.mzn

```
include "nvalue_fn.mzn";

enum REGION = { WA, NT, SA, Q, NSW, V, T };
enum COLOR = Color(1..card(REGION));

% Neighboring regions
array [_] of tuple(REGION, REGION): neighbors = [
  (WA,  NT),
  (WA,  SA),
  (NT,  SA),
  (NT,  Q),
  (SA,  Q),
  (SA,  NSW),
  (SA,  V),
  (Q,   NSW),
  (NSW, V)
];

% Color of each Region
array [REGION] of var COLOR: color;
% Number of colors used
var 1..card(COLOR): n_colors :: output = nvalue(color);

% Neighboring regions have different colours
constraint forall (pair in neighbors)
                  (color[pair.1] != color[pair.2]);

% Use as few colors as possible
solve minimize n_colors;
```

CP Modeling

australia.mzn

```
include "nvalue_fn.mzn";

enum REGION = { WA, NT, SA, Q, NSW, V, T };
enum COLOR = Color(1..card(REGION));

% Neighboring regions
array [_] of tuple(REGION, REGION): neighbors = [
  (WA,  NT),
  (WA,  SA),
  (NT,  SA),
  (NT,  Q),
  (SA,  Q),
  (SA,  NSW),
  (SA,  V),
  (Q,   NSW),
  (NSW, V)
];

% Color of each Region
array [REGION] of var COLOR: color;
% Number of colors used
var 1..card(COLOR): n_colors :: output = nvalue(color);

% Neighboring regions have different colours
constraint forall (pair in neighbors)
                  (color[pair.1] != color[pair.2]);

% Use as few colors as possible
solve minimize n_colors;
```

variables

CP Modeling

australia.mzn

```
include "nvalue_fn.mzn";

enum REGION = { WA, NT, SA, Q, NSW, V, T };
enum COLOR = Color(1..card(REGION));

% Neighboring regions
array [_] of tuple(REGION, REGION): neighbors = [
  (WA,  NT),
  (WA,  SA),
  (NT,  SA),
  (NT,  Q),
  (SA,  Q),
  (SA,  NSW),
  (SA,  V),
  (Q,   NSW),
  (NSW, V)
];

% Color of each Region
array [REGION] of var COLOR: color;
% Number of colors used
var 1..card(COLOR): n_colors :: output = nvalue(color);

% Neighboring regions have different colours
constraint forall (pair in neighbors)
                  (color[pair.1] != color[pair.2]);

% Use as few colors as possible
solve minimize n_colors;
```

domains

CP Modeling

australia.mzn

```
include "nvalue_fn.mzn";

enum REGION = { WA, NT, SA, Q, NSW, V, T };
enum COLOR = Color(1..card(REGION));

% Neighboring regions
array [_] of tuple(REGION, REGION): neighbors = [
  (WA,  NT),
  (WA,  SA),
  (NT,  SA),
  (NT,  Q),
  (SA,  Q),
  (SA,  NSW),
  (SA,  V),
  (Q,   NSW),
  (NSW, V)
];

% Color of each Region
array [REGION] of var COLOR: color;
% Number of colors used
var 1..card(COLOR): n_colors :: output = nvalue(color);

% Neighboring regions have different colours
constraint forall (pair in neighbors)
                  (color[pair.1] != color[pair.2]);

% Use as few colors as possible
solve minimize n_colors;
```

constraint

CP Modeling

australia.mzn

```
include "nvalue_fn.mzn";

enum REGION = { WA, NT, SA, Q, NSW, V, T };
enum COLOR = Color(1..card(REGION));

% Neighboring regions
array [_] of tuple(REGION, REGION): neighbors = [
  (WA,  NT),
  (WA,  SA),
  (NT,  SA),
  (NT,  Q),
  (SA,  Q),
  (SA,  NSW),
  (SA,  V),
  (Q,   NSW),
  (NSW, V)
];

% Color of each Region
array [REGION] of var COLOR: color;
% Number of colors used
var 1..card(COLOR): n_colors :: output = nvalue(color);

% Neighboring regions have different colours
constraint forall (pair in neighbors)
                  (color[pair.1] != color[pair.2]);

% Use as few colors as possible
solve minimize n_colors;
```

objective

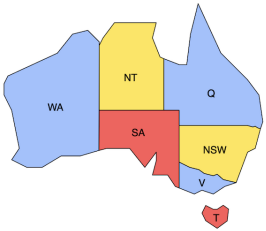
CP Modeling

australia.mzn

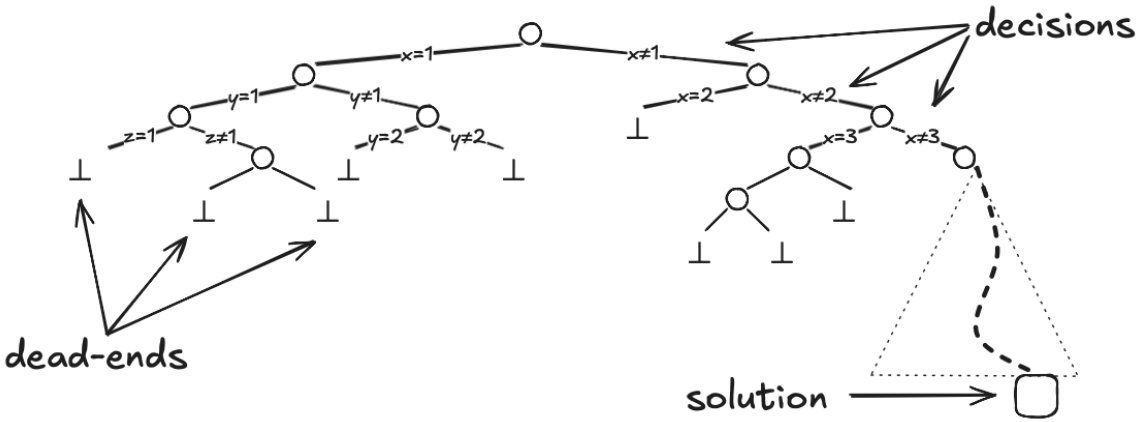
```
include "nvalue_fn.mzn";

enum REGION = { WA, NT, SA, Q, NSW, V, T };
enum COLOR = Color(1..card(REGION));

% Neighboring regions
array [_] of tuple(REGION, REGION): neighbors = [
  (WA,  NT),
  (WA,  SA),
  (NT,  SA),
  (NT,  Q),
  (SA,  Q),
  (SA,  NSW),
  (SA,  V),
  (Q,   NSW),
  (NSW, V)
];

% Color of each Region
array [REGION] of var COLOR: color;
% Number of colors used
var 1..card(COLOR): n_colors :: output = nvalue(color);

% Neighboring regions have different colours
constraint forall (pair in neighbors)
                  (color[pair.1] != color[pair.2]);

% Use as few colors as possible
solve minimize n_colors;
```

Running australia.mzn

```
color = [WA: Color(3), NT: Color(2), SA:
Color(1), Q: Color(3), NSW: Color(2), V:
Color(3), T: Color(1)];
n_colors = 3;
----------
==========
Finished in 177msec.
```



Backtracking Algorithm



Constraint Solvers

Google OR-Tools

OscaR
OPERATIONAL RESEARCH IN SCALA

chuffed/**chuffed**

The Chuffed CP solver

16 Contributors · 20 Issues · 123 Stars · 44 Forks

ptal/**turbo**

A constraint solver purely on GPUs (CUDA)

4 Contributors · 5 Issues · 40 Stars · 13 Forks

Constraint Solvers

Google OR-Tools

ACE

CHOCO

OscaR
OPERATIONAL RESEARCH IN SCALA

chuffed/**chuffed**

The Chuffed CP solver

16 Contributors · 20 Issues · 123 Stars · 44 Forks

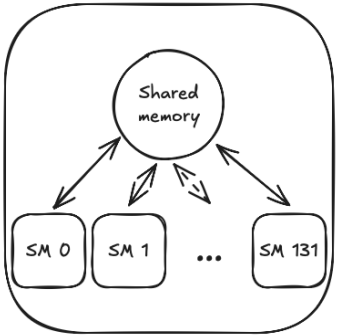
ptal/**turbo**

A constraint solver purely on GPUs (CUDA)

4 Contributors · 5 Issues · 40 Stars · 13 Forks

Turbo

A general purpose GPU CP solver.[1][2]
The underlying theoretical model is based on lock-free, asynchronous fixpoint iteration.



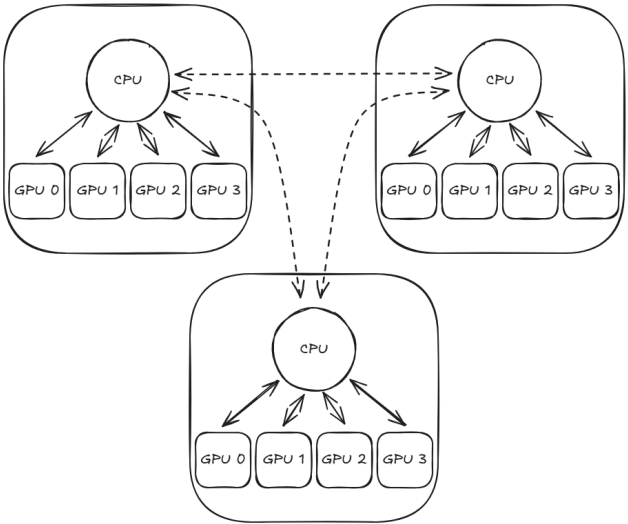
[1] P. Talbot et al., 'A Variant of Concurrent Constraint Programming on GPU', AAAI, 2022
[2] P. Talbot, 'A GPU-based Constraint Programming Solver', AAAI, 2026

Contributions

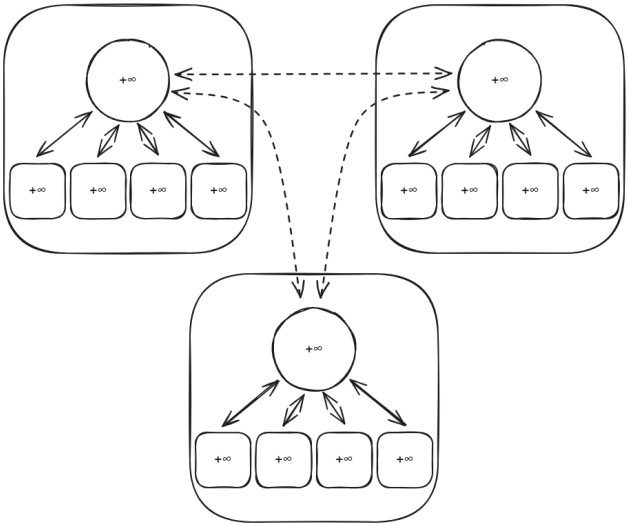
- Implementation of the first multi-GPU constraint solver.
- Investigation of state-based CRDT[3] for distributed constraint solving.
- High-performance implementation using MPI showing near-linear scaling.
- Establishing a connection between CRDT semantics and prior work on asynchronous iteration.

[3] M. Shapiro et al., 'Conflict-free replicated data types', SSS, 2011

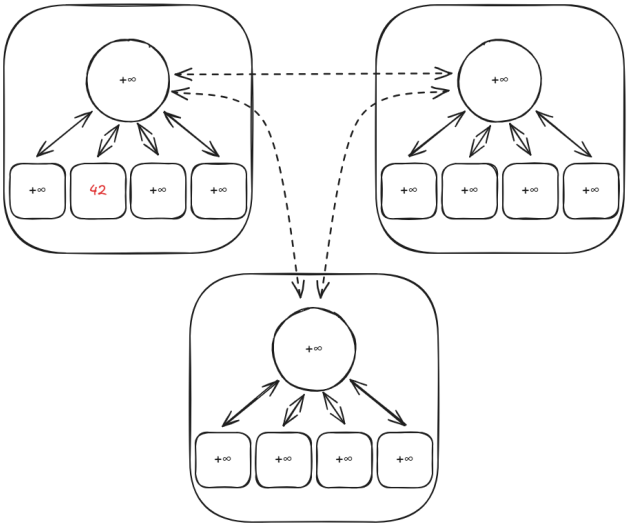
Counter CRDT for Bound Sharing



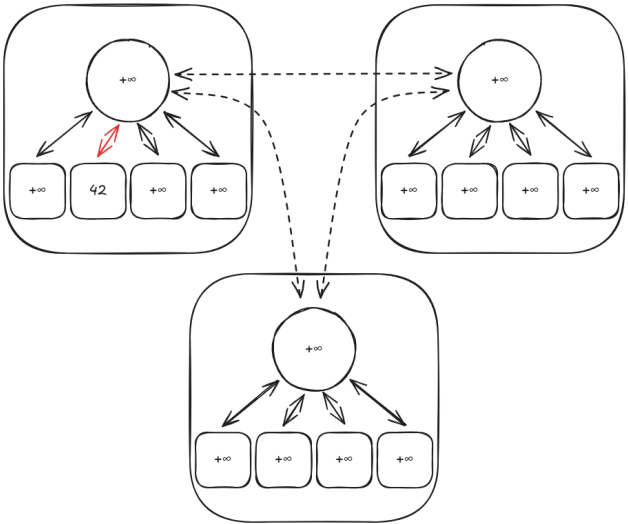
Counter CRDT for Bound Sharing



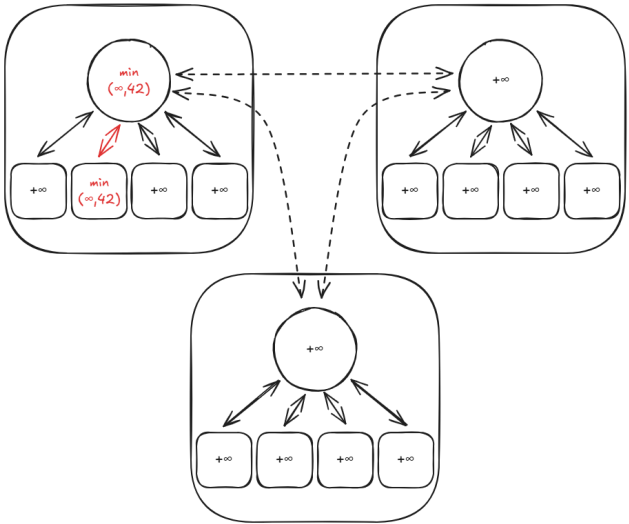
Counter CRDT for Bound Sharing



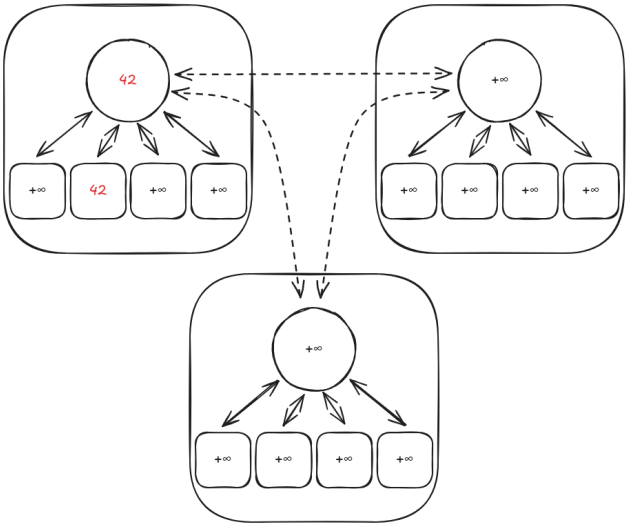
Counter CRDT for Bound Sharing



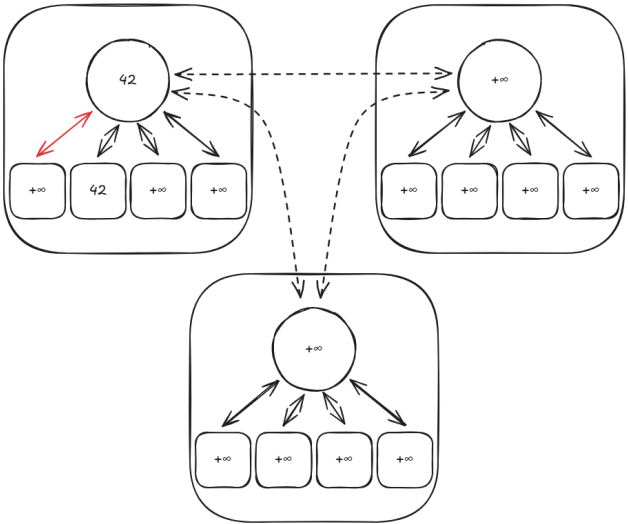
Counter CRDT for Bound Sharing



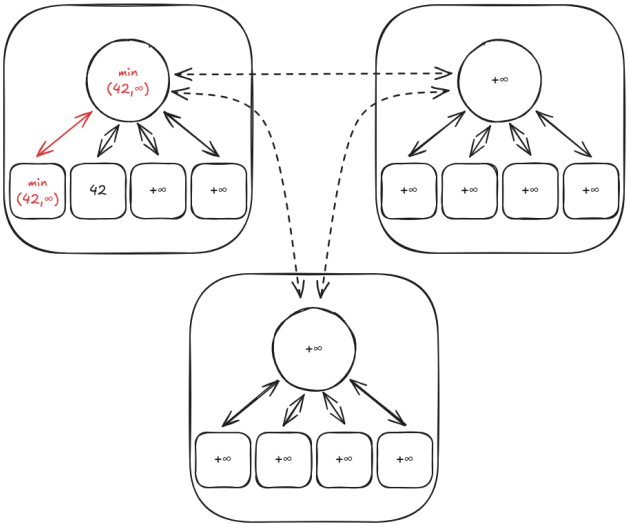
Counter CRDT for Bound Sharing



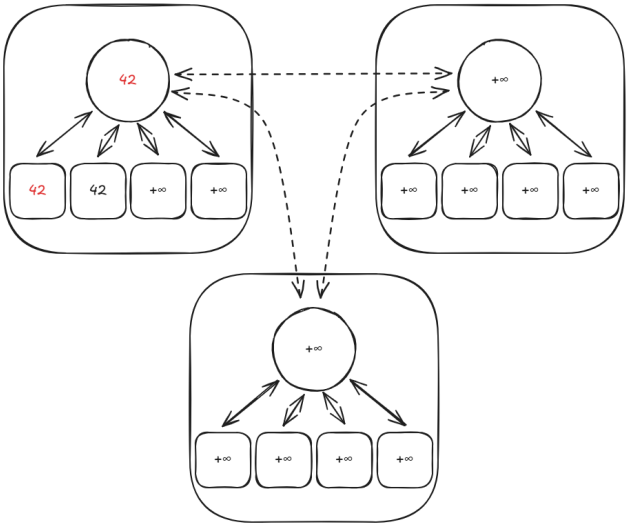
Counter CRDT for Bound Sharing



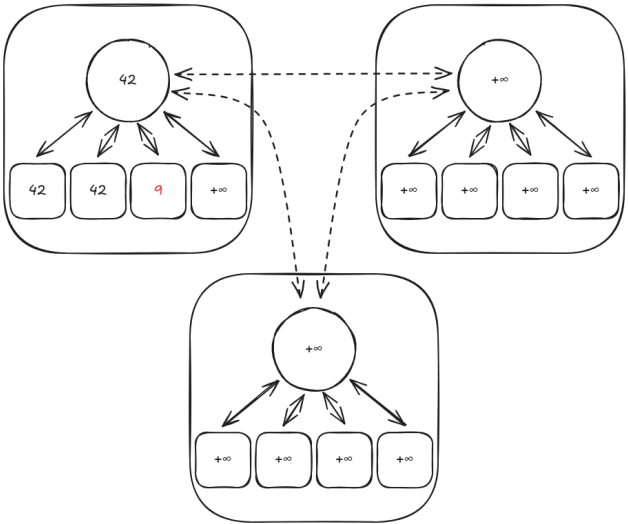
Counter CRDT for Bound Sharing



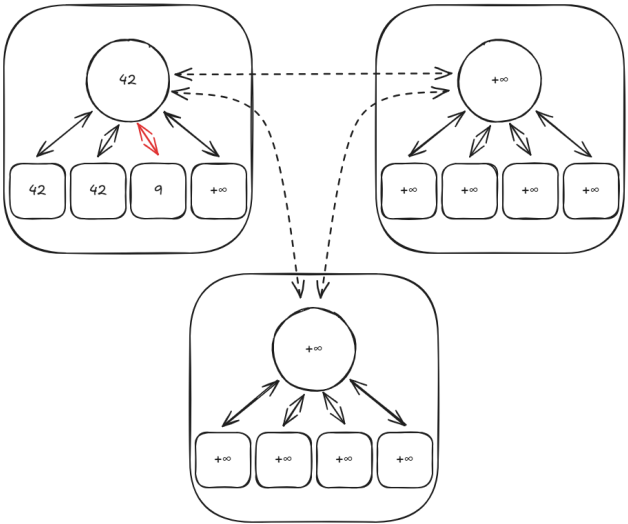
Counter CRDT for Bound Sharing



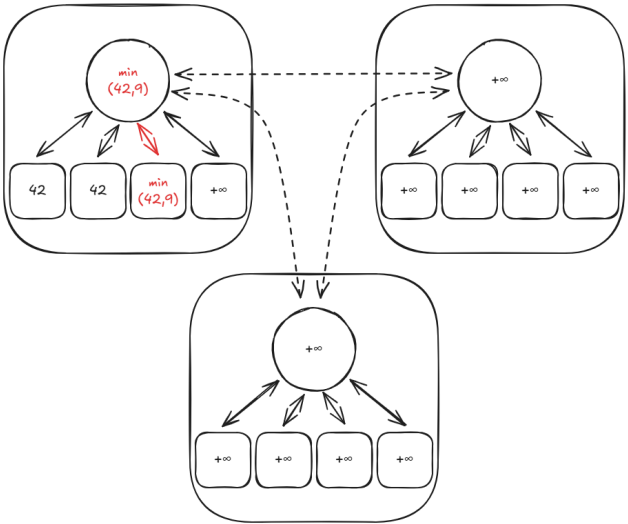
Counter CRDT for Bound Sharing



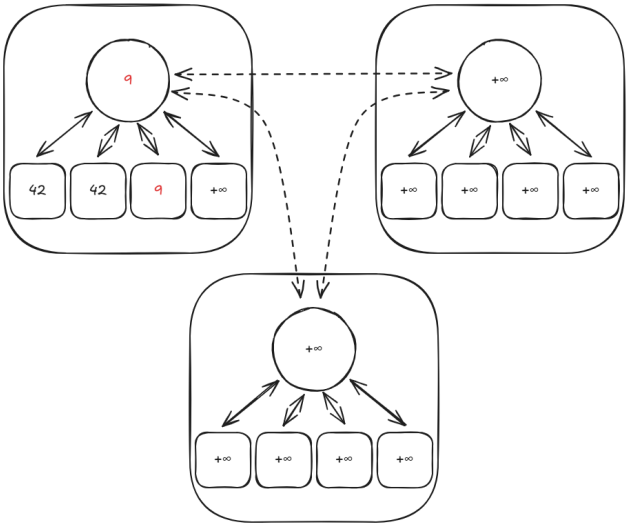
Counter CRDT for Bound Sharing



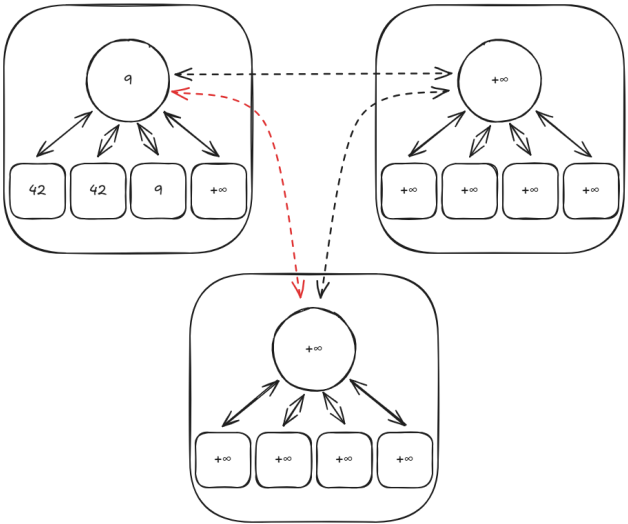
Counter CRDT for Bound Sharing



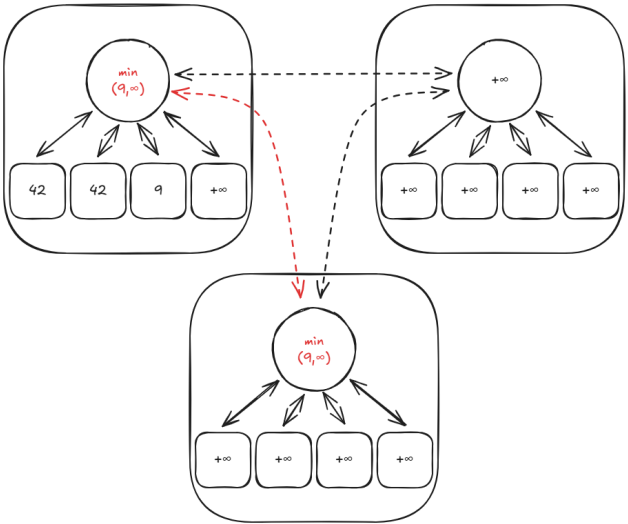
Counter CRDT for Bound Sharing



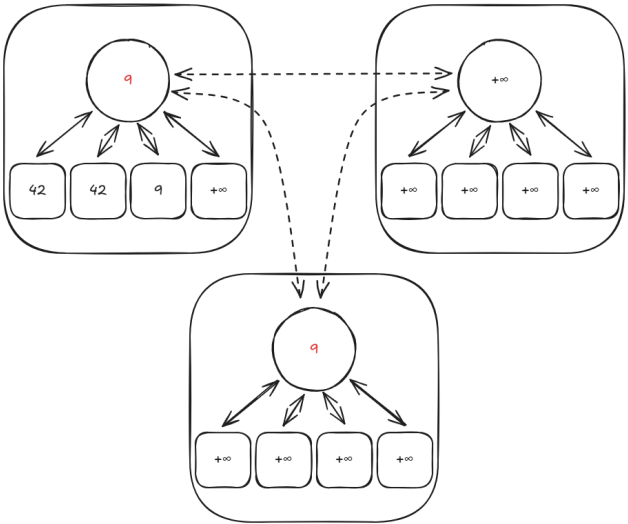
Counter CRDT for Bound Sharing



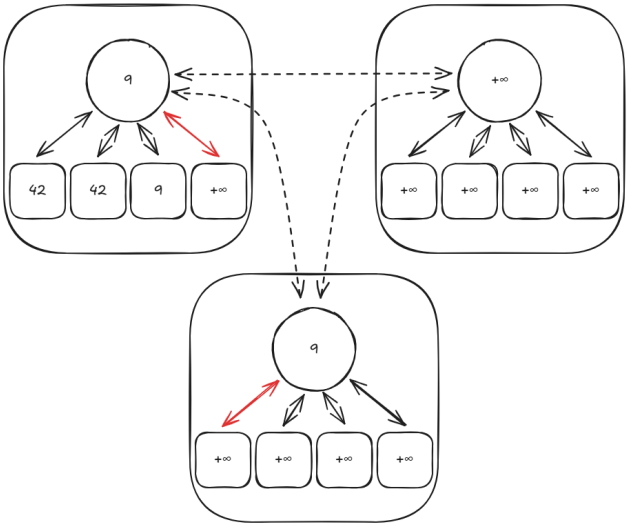
Counter CRDT for Bound Sharing



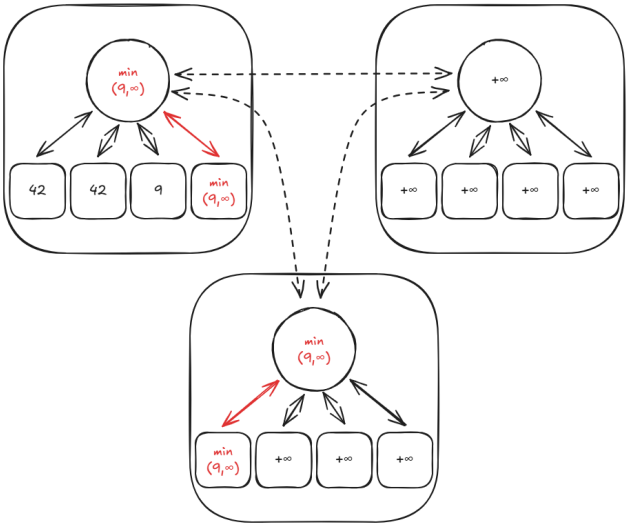
Counter CRDT for Bound Sharing



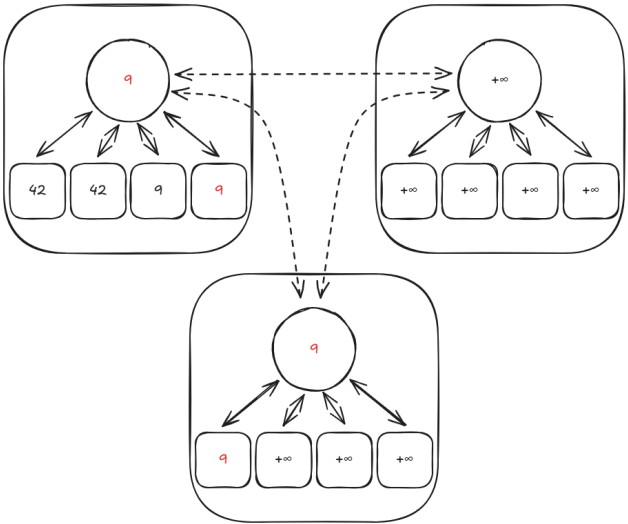
Counter CRDT for Bound Sharing



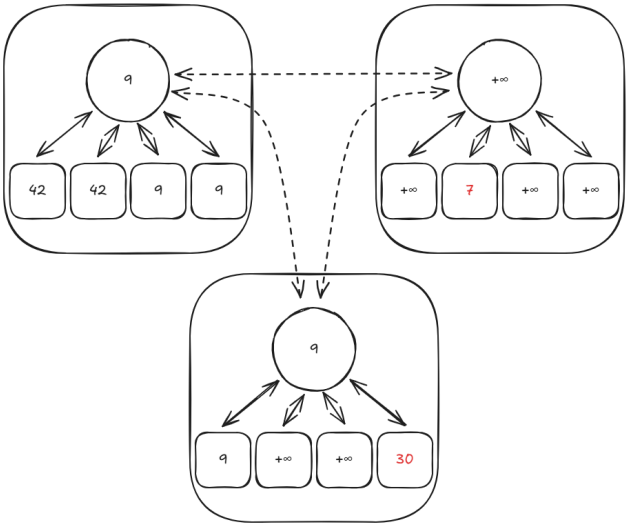
Counter CRDT for Bound Sharing



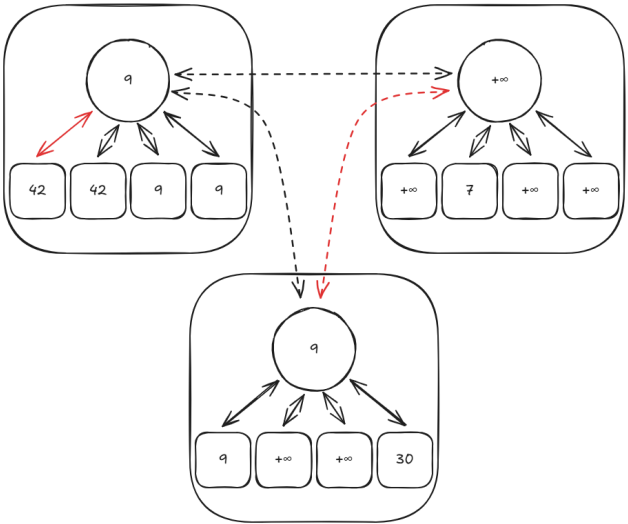
Counter CRDT for Bound Sharing



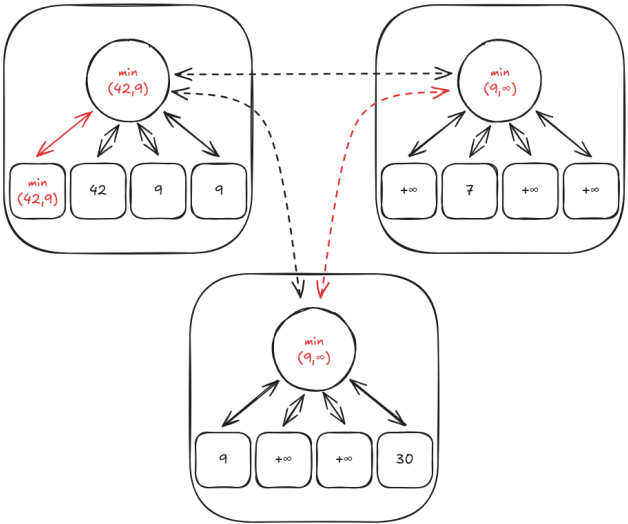
Counter CRDT for Bound Sharing



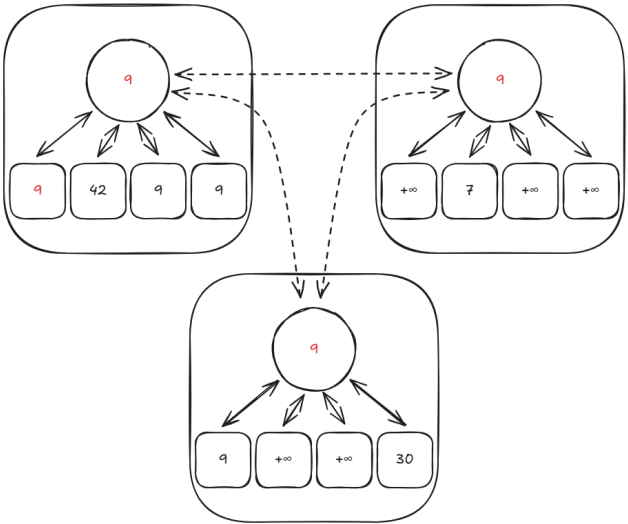
Counter CRDT for Bound Sharing



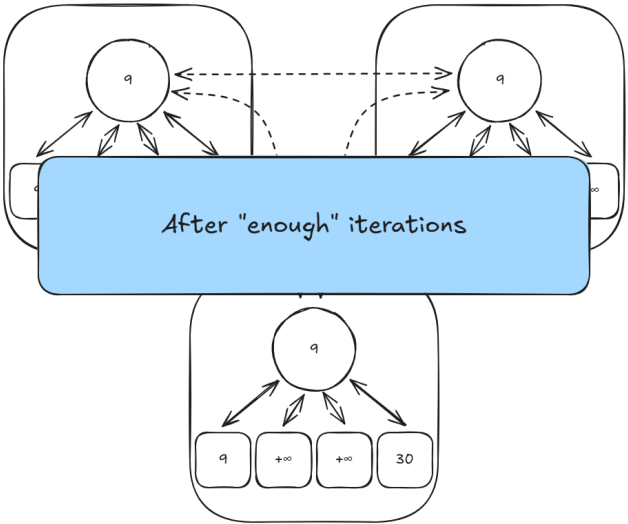
Counter CRDT for Bound Sharing



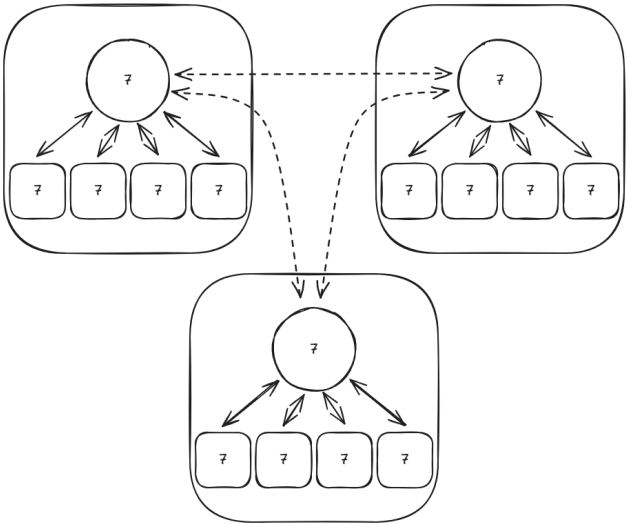
Counter CRDT for Bound Sharing



Counter CRDT for Bound Sharing



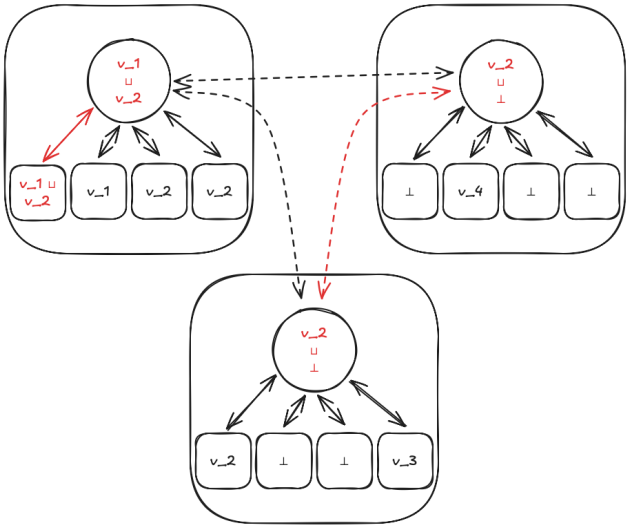
Counter CRDT for Bound Sharing



Beyond Bound Sharing: General Model

The solving process produces bounds that update the state monotonically. The accumulation of global information in distributed constraint optimization can be expressed as a fixpoint computation over monotone shared state.
The value of the best bound is the least fixpoint of the combined effect of all of join operators. Operationally, asynchronous execution corresponds to chaotic iteration of these per-update operators.

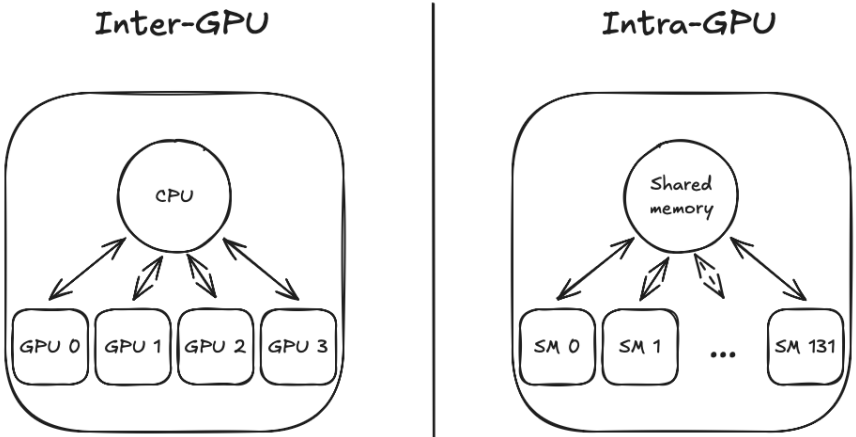
Fixpoint-Based Coordination Model



Implementation: CRDT-based Distributed Constraint Solving

- Local bound sharing and solver coordination are handled via one-sided MPI operations, ensuring asynchronous execution (in regard to the solving process).
- Hierarchical gossip-style protocol for periodic information exchange between nodes.

Single-GPU and Multi-GPU: One Coordination Mechanism



Single-GPU and Multi-GPU: One Coordination Mechanism

- **Single GPU (Intra-GPU):**
  - Threads share a common objective bound stored in an atomic variable.
- **Multi-GPU (Inter-GPU/Inter-Node):**
  - Independent solver instances exchange bounds asynchronously through one-sided MPI communication.
- **Same Logic, Different Scale:**
  - Both cases rely on the same principles: monotonicity, idempotence, and order-independent updates.
  - Fixpoint computation of the best bound.

Experimental Results

Benchmarks: 16 instances from a MiniZinc Challenge.
Time limit: 300s per run

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

Perspectives

Sharing nogoods

Definition

A nogood is an instantiation of a subset of variables that cannot be extended to a solution.

Example

$\neg(x = a \wedge y = b \wedge z = c)$
$\neg(w = c \wedge x = b \wedge y = b \wedge z = a)$

Recording nogoods permits to improve the process of exploring the search space.
Challenge: need to find a CRDT representation of nogoods.

Conclusion

- The first multi-GPU distributed CP solver.
- Reframing the sharing of global knowledge in constraint solving explicitly in CRDT semantics.
- Implementation of the CRDT bound-sharing concept using MPI over GPU constraint solver.
- Establishing a connection between CRDT semantics and prior work on asynchronous iteration.

Thank you for your attention!
Questions?