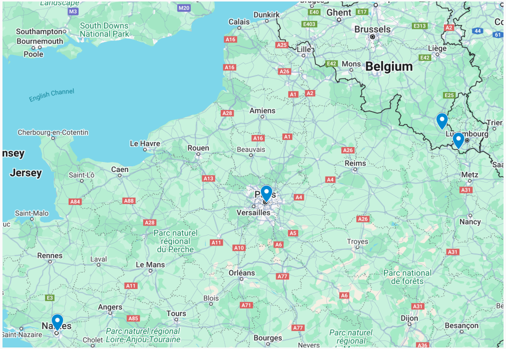
# Abstract Constraint Programming on GPU

Talk at Zhejiang University, ZLAIRE, Prof. Liao Group

**Pierre Talbot**
pierre.talbot@uni.lu
https://ptal.github.io
7th May 2025

University of Luxembourg

# Who am I?



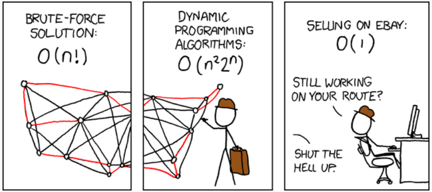

- 2014–2018: Ph.D., Sorbonne University, Paris
  - *Spacetime Programming: A Synchronous Language for Constraint Search.*
- 2018–2019: Postdoc, University of Nantes.
  - *Abstract Domains for Constraint Programming.*
- 2020–2023: Postdoc, University of Luxembourg
  - *A Lattice-Based Approach for GPU Programming.*
- 2023–: Research scientist, University of Luxembourg.
  - *Abstract Satisfaction.*

# My Research in a Nutshell!

I research on the “fusion” of...

Constraint reasoning + Abstract interpretation

(and lattice theory)





that gives us abstract satisfaction.

# My Research in a Nutshell!

I research on the "fusion" of...

**WHY?**

Accelerate constraint solving

**HOW?**

- Combining constraint solvers
- Constructing sound solving procedure over complex domains
- Constraint solving on GPUs

that gives us abstract satisfaction.

## My Research in a Nutshell!

I research on the "fusion" of...

**WHY?**

Accelerate constraint solving

**HOW?**

- Combining constraint solvers
- Constructing sound solving procedure over complex domains
- Constraint solving on GPUs $\Leftarrow$ TODAY

that gives us abstract satisfaction.

## TEAM

I have the pleasure to co-supervise and work with several Ph.D. students and postdocs.

Pierre Talbot

Thibault Falque

Hedieh Haddad

Manuel Combarro

Yi-Nung Tsao

Tobias Fischbach

Angelica Rings

Hakan Hasan

- Thibault Falque, postdoctoral researcher on the project COMOC, 2024-2025.
- Hedieh Haddad, Ph.D. candidate, *Hyperparameter Optimization of Constraint Solver*, 2023-2025.
- Manuel Combarro, Ph.D. candidate, *Multiobjective Constraint Programming*, 2023-2026.
- Yi-Nung Tsao, Ph.D. candidate, *Verification of Neural Networks by Abstract Interpretation*, 2023-2026.
- Tobias Fischbach, Ph.D. candidate, *Optimization of Quantum Circuits*, 2023-2026.
- Angelica Rings, Bachelor student, *A Musical Game to Understand the Challenge of Collaboration in Computing*, 2024-2025.
- Hasan Hakan, Master student, *Multi-GPU Parallel Constraint Programming*, 2025.

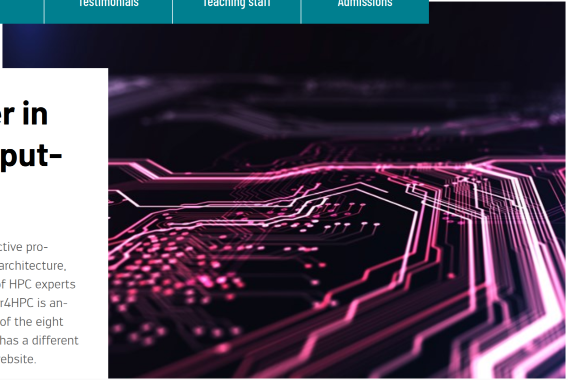
Home > Faculty of Science, Technology and Medicine > Study Programmes > Master in High Performance Computing

Study Programmes

# Master in High Performance Computing



## Your outstanding career in high-performance computing



The Master in High Performance Computing (MHPC) is a distinctive programme at the intersection of parallel programming, hardware architecture, and artificial intelligence. We are training the next generation of HPC experts in Luxembourg and Europe. Besides to the MHPC, the EUMaster4HPC is another programme where students earn a dual degree from two of the eight universities of the EUMaster4HPC consortium. EUMaster4HPC has a different application procedure, so be sure to check out the dedicated website.

# On the Menu

- **Abstract Satisfaction** (connection between logic and constraint reasoning)
- **Abstract Constraint Programming** (contribution, expressive reasoning framework)
- **Abstract Constraint Programming on GPU** (contribution, efficient reasoning framework)

# Abstract Satisfaction

# Syntax of First-Order Logic (FOL)

Let $S = \langle X, F, P \rangle$ be a *first-order signature* where $X$ set of variables, $F$ set of function symbols and $P$ set of predicate symbols.

| | |
|---|---|
| $\langle t \rangle ::= x$ | *variable* $x \in X$ |
| $\quad \mid \; f(t, \dots, t)$ | *function* $f \in F$ |

| | |
|---|---|
| $\langle \varphi \rangle ::= p(t, \dots, t)$ | *predicate* $p \in P$ |
| $\quad \mid \; \neg\varphi$ | *negation* |
| $\quad \mid \; \varphi \diamond \varphi$ | *connector* $\diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ |
| $\quad \mid \; \exists x, \; \varphi$ | *existential quantifier* |
| $\quad \mid \; \forall x, \; \varphi$ | *universal quantifier* |

Let $\Phi$ the set of well-formed formulas.

# Semantics of FOL

A *structure* $A$ is a tuple $(\mathbb{U}, [\![]\!]_F, [\![]\!]_P)$ where

1. $\mathbb{U}$ is a non-empty set of elements—called the *universe of discourse*,
2. $[\![]\!]_F$ is a function mapping function symbols $f \in F$ with arity $n$ to interpreted functions $[\![f]\!]_F : \mathbb{U}^n \to \mathbb{U}$, and
3. $[\![]\!]_P$ is a function mapping predicate symbols $p \in P$ with arity $n$ to interpreted predicates $[\![p]\!]_P \subseteq \mathbb{U}^n$.

An *assignment* is a function $X \to \mathbb{U}$ mapping variables to values. We denote the set of assignment by **Asn**. Let $\rho \in \textbf{Asn}$, we write $\rho[x \mapsto d]$ the assignment in which we updated the value of $x$ by $d$ in $\rho$.

# Entailment

The syntax and semantics are related by the ternary relation $A \vDash_\rho \varphi$, called the *entailment*, where $A$ is a structure, $\rho \in \mathbf{Asn}$ and $\varphi \in \Phi$. It is read as "the formula $\varphi$ is satisfied by the assignment $\rho$ in the structure $A$". We first give the interpretation function $[\![]\!]_\rho$ for evaluating the terms of the language:

$$\begin{aligned}
[\![x]\!]_\rho \quad & = \rho(x) \text{ if } x \in X \\
[\![f(t_1, \ldots, t_n)]\!]_\rho \quad & = [\![f]\!]_F([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho)
\end{aligned}$$

The relation $\vDash$ is defined inductively as follows:

$$\begin{array}{ll}
A \vDash_\rho p(t_1, \ldots, t_n) & \text{iff } ([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho) \in [\![p]\!]_P \\
A \vDash_\rho \varphi_1 \wedge \varphi_2 & \text{iff } A \vDash_\rho \varphi_1 \text{ and } A \vDash_\rho \varphi_2 \\
A \vDash_\rho \varphi_1 \vee \varphi_2 & \text{iff } A \vDash_\rho \varphi_1 \text{ or } A \vDash_\rho \varphi_2 \\
A \vDash_\rho \neg\varphi & \text{iff } A \vDash_\rho \varphi \text{ does not hold} \\
A \vDash_\rho \exists x,\ \varphi & \text{iff there exists } d \in \mathbb{U} \text{ such that } A \vDash_{\rho[x \mapsto d]} \varphi \\
A \vDash_\rho \forall x,\ \varphi & \text{iff for all } d \in \mathbb{U},\ \text{we have } A \vDash_{\rho[x \mapsto d]} \varphi
\end{array}$$

# Concrete Domain

Given a structure $A$, we define the *concrete interpretation function* as:

$$\llbracket . \rrbracket^\flat : \Phi \to \mathcal{P}(\mathbf{Asn})$$
$$\llbracket \varphi \rrbracket^\flat = \{\rho \in \mathbf{Asn} \mid A \vDash_\rho \varphi\}$$

- We call the *concrete domain* the set $\mathcal{P}(\mathbf{Asn})$ with $\llbracket . \rrbracket^\flat$.
- A *solution* of the formula $\varphi$ is an assignment $s \in \llbracket \varphi \rrbracket^\flat$.
- **Example** in the theory of standard integer arithmetics (and $X = \{x, y\}$):

$$\begin{aligned}
&\llbracket x < y \land x \geq 0 \rrbracket^\flat = \{ \\
&\quad \{x \mapsto 0, y \mapsto 1\} \\
&\quad \{x \mapsto 0, y \mapsto 2\} \\
&\ldots \\
&\quad \{x \mapsto 1, y \mapsto 2\} \\
&\ldots \\
&\}
\end{aligned}$$

# One Problem, Many Communities, Many Formalisms

Many communities emerged to solve the same problem: find $\rho$ such that $A \vDash_\rho \varphi$.

BUT they (generally) focus on different fragments of FOL:

- Propositional fragment (SAT): $(a \vee b) \wedge (\neg b \vee c)$ with $a, b, c \in \{0, 1\}$.
- Pseudo-Boolean fragment: $\sum_{1 \leq i \leq n} c_i * a_i \leq c_0$ with $a_i \in \{0, 1\}$ and $c_i$ some integers constants.
- Linear programming (LP): $\sum_{1 \leq i \leq n} c_i * b_i \leq b_0$ with $b_i \in \mathbb{R}$ and $c_i$ some real constants.
- Integer linear programming (ILP): $\sum_{1 \leq i \leq n} c_i * b_i \leq b_0$ with $b_i \in \mathbb{Z}$ and $c_i$ some integer constants.
- Mixed integer linear programming (MILP): $\sum_{1 \leq i \leq n} c_i * b_i \leq b_0$ with $b_i \in \mathbb{Z}$ or $b_i \in \mathbb{R}$ and $c_i$ some integer or real constants.
- Uninterpreted fragment (logic programming).
- Discrete constraint programming: $\langle X, D, C \rangle$ with $D_i \in \mathcal{P}_f(\mathbb{Z})$.
- Continuous constraint programming: $\langle X, D, C \rangle$ with $D_i \in \mathcal{I}(\mathbb{R})$.
- Satisfiability modulo theories (SMT).
- ...

# One Theory to Rule Them All?

- SAT [DHK13]
- SMT [DHK14]
- Logic programming [Cou20]
- Constraint programming ($\mathbb{R}$) [Pel+13]
- Constraint programming ($\mathbb{Z}$) [Tal+19]
- Linear programming [CH78]
- Answer set programming
- ...

$\longrightarrow$ Abstract domains

## What is an *abstract domain*?

It is a lattice with some operations.

## What is an *abstract domain*?

It is a lattice with some operations.

## What is a *lattice*?

A tuple $\langle S, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ where $S$ is a set.

## What is an *abstract domain*?

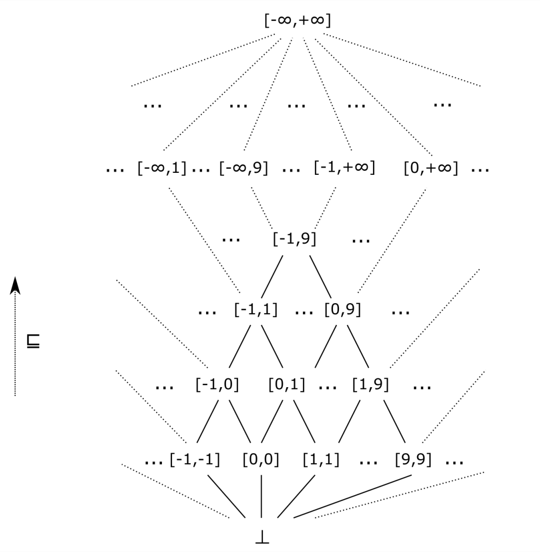
It is a lattice with some operations.

## What is a *lattice*?

A tuple $\langle S, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ where $S$ is a set.

## Example: Interval Lattice

- $S \triangleq \{[a,b] \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\} \cup \{\bot\}$
- $[a,b] \sqsubseteq [c,d] \Leftrightarrow a \geq c \wedge b \leq d$
- $\top \triangleq [-\infty, \infty]$
- $[a,b] \sqcap [c,d] \triangleq [\max\{a,c\}, \min\{b,d\}]$



# Simple Logic of Intervals

- **Logic**: $\Phi \triangleq x \leq k \mid x \geq k \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$. (only 1 variable)

# Simple Logic of Intervals

- **Logic**: $\Phi \triangleq x \leq k \mid x \geq k \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$. (only 1 variable)
- **Abstract interpretation**:
  - $[\![x \leq k]\!] \triangleq [-\infty, k]$
  - $[\![x \geq k]\!] \triangleq [k, \infty]$
  - $[\![\varphi \wedge \varphi']\!] \triangleq [\![\varphi]\!] \sqcap [\![\varphi']\!]$
  - $[\![\varphi \vee \varphi']\!] \triangleq [\![\varphi]\!] \sqcup [\![\varphi']\!]$

# Simple Logic of Intervals

- **Logic**: $\Phi \triangleq x \leq k \mid x \geq k \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$. (only 1 variable)
- **Abstract interpretation**:
  - $[\![x \leq k]\!] \triangleq [-\infty, k]$
  - $[\![x \geq k]\!] \triangleq [k, \infty]$
  - $[\![\varphi \wedge \varphi']\!] \triangleq [\![\varphi]\!] \sqcap [\![\varphi']\!]$
  - $[\![\varphi \vee \varphi']\!] \triangleq [\![\varphi]\!] \sqcup [\![\varphi']\!]$
- **Example**:
  - $[\![(x \leq 10 \wedge x \geq 0) \vee (x \geq 5)]\!]$
  - $[\![x \leq 10 \wedge x \geq 0]\!] \sqcup [\![x \geq 5]\!]$
  - $([\![x \leq 10]\!] \sqcap [\![x \geq 0]\!]) \sqcup [\![x \geq 5]\!]$
  - $([-\infty, 10] \sqcap [0, \infty]) \sqcup [5, \infty]$
  - $[0, 10] \sqcup [5, \infty]$
  - $[0, \infty]$

# Simple Logic of Intervals

- **Logic**: $\Phi \triangleq x \leq k \mid x \geq k \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$. (only 1 variable)
- **Abstract interpretation**:
  - $[\![x \leq k]\!] \triangleq [-\infty, k]$
  - $[\![x \geq k]\!] \triangleq [k, \infty]$
  - $[\![\varphi \wedge \varphi']\!] \triangleq [\![\varphi]\!] \sqcap [\![\varphi']\!]$
  - $[\![\varphi \vee \varphi']\!] \triangleq [\![\varphi]\!] \sqcup [\![\varphi']\!]$
- **Example**:
  - $[\![(x \leq 10 \wedge x \geq 0) \vee (x \geq 5)]\!]$
  - $[\![x \leq 10 \wedge x \geq 0]\!] \sqcup [\![x \geq 5]\!]$
  - $([\![x \leq 10]\!] \sqcap [\![x \geq 0]\!]) \sqcup [\![x \geq 5]\!]$
  - $([-\infty, 10] \sqcap [0, \infty]) \sqcup [5, \infty]$
  - $[0, 10] \sqcup [5, \infty]$
  - $[0, \infty]$
- **Soundness**: $[\![\varphi]\!]^\flat \subseteq [\![\varphi]\!]$ (compute all solutions).
- **Completeness**: $[\![\varphi]\!]^\flat \supseteq [\![\varphi]\!]$ (compute only solutions).
  Intervals are not complete: $[\![x \leq 10 \vee x \geq 15]\!] = [-\infty, \infty]$ (intervals cannot represent "holes").

# What About Multiple Variables?

We lift interval to a function $X \to \mathit{Itv}$ mapping variables to intervals where $\mathit{Itv}$ is the interval lattice.

Now, we can define (with $x \in X$ any variable):

- $[\![x \leq k]\!] \triangleq \{x \mapsto [-\infty, k]\}$.
- $[\![x \geq k]\!] \triangleq \{x \mapsto [k, \infty]\}$.
- ...

**Example**: $[\![x \leq 0 \wedge y \geq 0]\!] = \{x \mapsto [-\infty, 0], y \mapsto [0, \infty]\}$.

## What About Multiple Variables?

We lift interval to a function $X \to \mathit{Itv}$ mapping variables to intervals where $\mathit{Itv}$ is the interval lattice.
Now, we can define (with $x \in X$ any variable):

- $[\![x \leq k]\!] \triangleq \{x \mapsto [-\infty, k]\}$.
- $[\![x \geq k]\!] \triangleq \{x \mapsto [k, \infty]\}$.
- ...

**Example**: $[\![x \leq 0 \wedge y \geq 0]\!] = \{x \mapsto [-\infty, 0], y \mapsto [0, \infty]\}$.

How to compute solutions of more expressive logic?

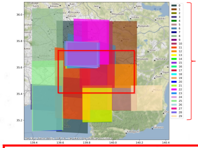
# Abstract Constraint Programming

# Constraint Programming

**Constraint programming**: FOL without quantifiers, $\mathbb{U} = \mathbb{Z}$ and arithmetic constraints.

- **Declarative paradigm**: specify your problem and let the computer solves it for you.
- **Many applications**: scheduling, bin-packing, hardware design, satellite imaging, ...
- **Constraint programming** is one approach to solve such combinatorial problems.
- Other approaches include SAT, linear programming, SMT, MILP, ASP,...

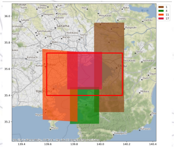
# Satellite image mosaic



After a query with certain parameters:

N = 30 satellite images

Find the cover with the minimum number of images (NP-Hard)



Build the mosaic

[1]

[1] Combarro et al., Constraint Model for the Satellite Image Mosaic Selection Problem, CP 2023

Constraint model of satellite imaging in MiniZinc:

New model Open Save Copy Cut Paste Undo Redo Shift left Shift right Run Solver configuratior Gecode 6.3.0

satellite.mzn satellite1.dzn

```
int: universe;

set of int: IMAGES = 1..images;
set of int: UNIVERSE = 1..universe;

array[IMAGES] of set of int: sets;
array[IMAGES] of int: costs;

constraint forall(u in UNIVERSE)(
  exists(i in IMAGES)(taken[i] /\ u in sets[i]));

array[IMAGES] of var bool: taken;

solve minimize sum(i in IMAGES)(costs[i] * taken[i]);
```

Output

Hide all dzn default

```
Running satellite.mzn, satellite1.dzn
taken = [true, true, false, true, true, false];
----------
==========
Finished in 114msec.
```

# Constraint Network

## Constraint Network

Let $X$ be a finite set of variables and $C$ be a finite set of constraints.
A *constraint network* is a pair $P = \langle d, C \rangle$ such that $d \in X \rightarrow Itv$ is the *domain* of the variables where $Itv$ is the set of intervals.

**Note**: It is just a "format" to represent quantifier-free logical formulas where variables have bounded domains.

## Example

$$\langle \{x \mapsto [0, 2], y \mapsto [2, 3]\}, \{x \leq y - 1\} \rangle$$

A solution is $\{x \mapsto 0, y \mapsto 2\}$.

## Constraints with Multiple Variables

- We already have: $[\![x \leq k]\!] \triangleq \{x \mapsto [-\infty, k]\}$.
- Also: $[\![x = k]\!] \triangleq \{x \mapsto [k, k]\}$.

How to interpret $[\![x = y]\!]$?

# Constraints with Multiple Variables

- We already have: $[\![x \leq k]\!] \triangleq \{x \mapsto [-\infty, k]\}$.
- Also: $[\![x = k]\!] \triangleq \{x \mapsto [k, k]\}$.

How to interpret $[\![x = y]\!]$?

- Unfortunately, without more information on $x$ and $y$, we must set $[\![x = y]\!] \triangleq \{x \mapsto [-\infty, \infty], y \mapsto [-\infty, \infty]\}$, which is the same than ignoring the constraint...

# Constraints with Multiple Variables

- We already have: $[\![x \leq k]\!] \triangleq \{x \mapsto [-\infty, k]\}$.
- Also: $[\![x = k]\!] \triangleq \{x \mapsto [k, k]\}$.

How to interpret $[\![x = y]\!]$?

- Unfortunately, without more information on $x$ and $y$, we must set $[\![x = y]\!] \triangleq \{x \mapsto [-\infty, \infty], y \mapsto [-\infty, \infty]\}$, which is the same than ignoring the constraint...

Solution: give more information to the interpretation function.

- $\mathcal{I}[\![.]\!] \in \Phi \times (X \to Itv) \to (X \to Itv)$
- $\mathcal{I}[\![x = y]\!]d \triangleq \{x \mapsto d(x) \sqcap d(y), y \mapsto d(x) \sqcap d(y)\}$

**Example**: Let $d = \{x \mapsto [0, 5], y \mapsto [5, 10]\}$, then $\mathcal{I}[\![x = y]\!]d = \{x \mapsto [5, 5], y \mapsto [5, 5]\}$.

# How to Deal with Conjunction?

- Before, we had $[\![\varphi \wedge \varphi']\!] \triangleq [\![\varphi]\!] \sqcap [\![\varphi']\!]$.
- Now, we can lift this function to

$$\mathcal{I}[\![\varphi \wedge \varphi']\!]d \triangleq \mathcal{I}[\![\varphi]\!]d \sqcap \mathcal{I}[\![\varphi']\!]d$$

# How to Deal with Conjunction?

- Before, we had $[\![\varphi \wedge \varphi']\!] \triangleq [\![\varphi]\!] \sqcap [\![\varphi']\!]$.
- Now, we can lift this function to

$$\mathcal{I}[\![\varphi \wedge \varphi']\!] d \triangleq \mathcal{I}[\![\varphi]\!] d \sqcap \mathcal{I}[\![\varphi']\!] d$$

Problem: $d$ must be copied... inefficient

# How to Deal with Conjunction?

- Before, we had $[\![\varphi \wedge \varphi']\!] \triangleq [\![\varphi]\!] \sqcap [\![\varphi']\!]$.
- Now, we can lift this function to

$$\mathcal{I}[\![\varphi \wedge \varphi']\!]d \triangleq \mathcal{I}[\![\varphi]\!]d \sqcap \mathcal{I}[\![\varphi']\!]d$$

Problem: $d$ must be copied... inefficient

- Instead, we can use functional composition:

$$\mathcal{I}[\![\varphi \wedge \varphi']\!]d \triangleq (\mathcal{I}[\![\varphi]\!] \circ \mathcal{I}[\![\varphi']\!])d$$

# Computing Solutions of Constraint Network

A constraint network $\langle d, C \rangle$ is a conjunctive collection of constraints. So we can compute the set of solutions using:

$$\mathcal{I}[\![c_1 \wedge c_2 \wedge \ldots \wedge c_n]\!] = \mathcal{I}[\![c_1]\!] \circ \mathcal{I}[\![c_2]\!] \circ \ldots \circ \mathcal{I}[\![c_n]\!]$$

**Example**: Let $\langle d, \{x = y, y = z\} \rangle$ be a constraint network with $d = \{x \mapsto [2,2], y \mapsto [1,2], z \mapsto [0,2]\}$, then:

$$\begin{array}{rl} & \mathcal{I}[\![x = y \wedge y = z]\!]d \\ = & (\mathcal{I}[\![x = y]\!] \circ \mathcal{I}[\![y = z]\!])d \\ = & \mathcal{I}[\![x = y]\!](\mathcal{I}[\![y = z]\!](d)) \\ = & \mathcal{I}[\![x = y]\!](\{x \mapsto [2,2], y \mapsto [1,2], z \mapsto [1,2]\} \\ = & \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [1,2]\} \end{array}$$

# Computing Solutions of Constraint Network

A constraint network $\langle d, C\rangle$ is a conjunctive collection of constraints. So we can compute the set of solutions using:

$$\mathcal{I}[\![c_1 \wedge c_2 \wedge \ldots \wedge c_n]\!] = \mathcal{I}[\![c_1]\!] \circ \mathcal{I}[\![c_2]\!] \circ \ldots \circ \mathcal{I}[\![c_n]\!]$$

**Example**: Let $\langle d, \{x = y, y = z\}\rangle$ be a constraint network with $d = \{x \mapsto [2, 2], y \mapsto [1, 2], z \mapsto [0, 2]\}$, then:

$$\begin{array}{rl} & \mathcal{I}[\![x = y \wedge y = z]\!]d \\ = & (\mathcal{I}[\![x = y]\!] \circ \mathcal{I}[\![y = z]\!])d \\ = & \mathcal{I}[\![x = y]\!](\mathcal{I}[\![y = z]\!](d)) \\ = & \mathcal{I}[\![x = y]\!](\{x \mapsto [2, 2], y \mapsto [1, 2], z \mapsto [1, 2]\} \\ = & \{x \mapsto [2, 2], y \mapsto [2, 2], z \mapsto [1, 2]\} \end{array}$$

We are not very precise... $z = [1, 2]$ instead of $z = [2, 2]$.

## Computing the Greatest Fixpoint

- $d_1 = \{x \mapsto [2,2], y \mapsto [1,2], z \mapsto [0,2]\}$.
- $d_2 = \mathcal{I}[\![x = y \wedge y = z]\!]d_1 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [1,2]\}$.

# Computing the Greatest Fixpoint

- $d_1 = \{x \mapsto [2,2], y \mapsto [1,2], z \mapsto [0,2]\}$.
- $d_2 = \mathcal{I}[\![x = y \wedge y = z]\!]d_1 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [1,2]\}$.
- **More precision?** We can apply the function again!
- $d_3 = \mathcal{I}[\![x = y \wedge y = z]\!]d_2 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [2,2]\}$.

# Computing the Greatest Fixpoint

- $d_1 = \{x \mapsto [2,2], y \mapsto [1,2], z \mapsto [0,2]\}$.
- $d_2 = \mathcal{I}[\![x = y \wedge y = z]\!]d_1 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [1,2]\}$.
- **More precision?** We can apply the function again!
- $d_3 = \mathcal{I}[\![x = y \wedge y = z]\!]d_2 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [2,2]\}$.
- **Again?** $d_3 = \mathcal{I}[\![x = y \wedge y = z]\!]d_3$, nothing changed! We reached a *fixpoint*.

# Computing the Greatest Fixpoint

- $d_1 = \{x \mapsto [2,2], y \mapsto [1,2], z \mapsto [0,2]\}$.
- $d_2 = \mathcal{I}[\![x = y \land y = z]\!] d_1 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [1,2]\}$.
- **More precision?** We can apply the function again!
- $d_3 = \mathcal{I}[\![x = y \land y = z]\!] d_2 = \{x \mapsto [2,2], y \mapsto [2,2], z \mapsto [2,2]\}$.
- **Again?** $d_3 = \mathcal{I}[\![x = y \land y = z]\!] d_3$, nothing changed! We reached a *fixpoint*.

For all formulas $\varphi$, $\mathcal{I}[\![\varphi]\!]$ is a *monotone function*.
Hence, we are guaranteed to find the greatest fixpoint, which is unique (Tarski theorem).

**Constraint propagation** is an approach to compute efficiently the *greatest fixpoint*:

$$\mathbf{gfp}_d\ \mathcal{I}[\![c_1]\!] \circ \ldots \circ \mathcal{I}[\![c_n]\!]$$

# Propagate and Search

The main algorithm behind constraint solvers:

> **function** $\textsc{solve}(d, \{c_1, \ldots, c_n\})$
>     $d \leftarrow \mathbf{gfp}_d\ \mathcal{I}[\![c_1]\!] \circ \ldots \circ \mathcal{I}[\![c_n]\!]$
>     **if** $\forall x \in X,\ d(x) = [v, v]$ **then return** $\{d\}$
>     **else if** $\exists x \in X,\ d(x) = \bot$ **then return** $\{\}$
>     **else**
>         $\langle d_1, \ldots, d_n \rangle \leftarrow \mathtt{split}(d)$
>         **return** $\bigcup_{i=0}^{n} \mathtt{solve}(d_i, C)$
>     **end if**
> **end function**

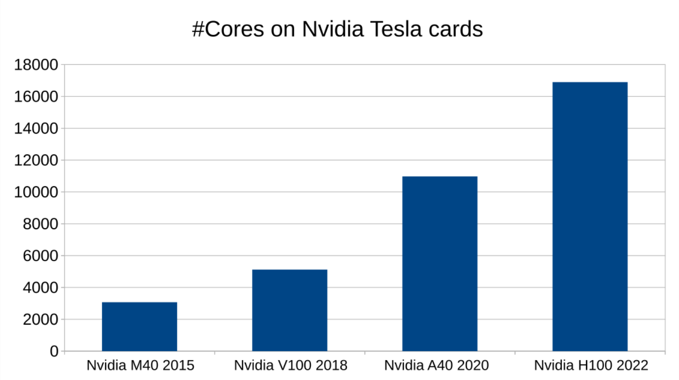
Thanks to the split function, the algorithm is **sound and complete**.

# Constraint Programming on GPU

# Why Constraint Programming on GPU?

# Why CP on GPU?

CPU clock speed is stagnating, GPU #cores is increasing quickly each year.




Easy speed-up: same code but faster.

# Why CP on GPU?

- Machine learning (deep learning, reinforcement learning, . . . ) has seen tremendous speed-ups (e.g. 100x, 1000x) by using GPU.
- Some (sequential) optimizations on CPU are made irrelevant if we can explore huge state space faster.

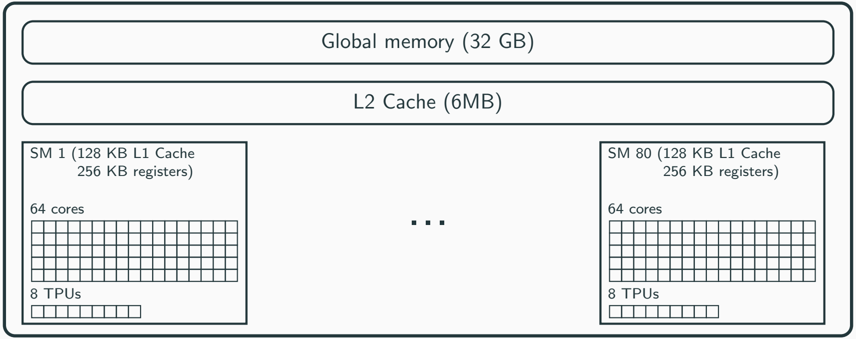
Can we replicate the success of GPU on machine learning applications to combinatorial optimization?

The rest of this talk:

- GPU Architecture.
- Challenges of Constraint Programming on GPU.
- Parallel Model of Computation.
- Ternary Constraint Network.

# GPU Architecture

# (Simplified) Architecture of the GPU Nvidia V100




**5120 cores on a single V100 GPU @ 1290MHz**

Whitepaper: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`

# Programming Challenges

- **Memory coalescence**: the way to access the data is important (factor 10).
- **Thread divergence**: each thread within a warp (group of 32 threads) should execute the same instructions.
- **Memory allocation** (dynamic data structures): costly on GPU, everything is generally pre-allocated.
- **Other limitations**: small cache, limited number of lines of code, limited STL...

# Example: Find the Minimum in an Array (CPU Style)

Each thread computes its local min (*map*), then we compute the min of all local min (*reduce*).

- Map:

$$\underbrace{\boxed{3}\boxed{22}\boxed{10}\boxed{23}}_{\text{Thread } 0,\ m_0 = 3}\underbrace{\boxed{21}\boxed{7}\boxed{91}\boxed{1}}_{\text{Thread } 1,\ m_1 = 1}\underbrace{\boxed{3}\boxed{10}\boxed{42}\boxed{11}}_{\text{Thread } 2,\ m_2 = 3}\underbrace{\boxed{8}\boxed{7}\boxed{32}}_{\text{Thread } 3,\ m_3 = 7}$$

- Reduce: $min([3, 1, 3, 7]) = 1$.

# Example: Find the Minimum in an Array (CPU Style)

Iteration 1:

- Map:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |

$\underbrace{3, 22, 10, 23}_{\text{Thread 0, } m_0 = 3}\ \underbrace{21, 7, 91, 1}_{\text{Thread 1, } m_1 = 1}\ \underbrace{3, 10, 42, 11}_{\text{Thread 2, } m_2 = 3}\ \underbrace{8, 7, 32}_{\text{Thread 3, } m_3 = 7}$

- Reduce: $min([3, 1, 3, 7]) = 1$.

# Example: Find the Minimum in an Array (CPU Style)

Iteration 2:

- Map:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |

$\underbrace{3, 22, 10, 23}_{\text{Thread 0, } m_0 = 3}$ $\underbrace{21, 7, 91, 1}_{\text{Thread 1, } m_1 = 1}$ $\underbrace{3, 10, 42, 11}_{\text{Thread 2, } m_2 = 3}$ $\underbrace{8, 7, 32}_{\text{Thread 3, } m_3 = 7}$

- Reduce: $min([3, 1, 3, 7]) = 1$.

# Example: Find the Minimum in an Array (CPU Style)

Iteration 3:

- Map:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |

$\underbrace{3, 22, 10, 23}_{\text{Thread 0, } m_0 = 3}\ \underbrace{21, 7, 91, 1}_{\text{Thread 1, } m_1 = 1}\ \underbrace{3, 10, 42, 11}_{\text{Thread 2, } m_2 = 3}\ \underbrace{8, 7, 32}_{\text{Thread 3, } m_3 = 7}$

- Reduce: $min([3, 1, 3, 7]) = 1$.

# Example: Find the Minimum in an Array (CPU Style)

Iteration 4:

- Map:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |

Thread 0, $m_0 = 3$ Thread 1, $m_1 = 1$ Thread 2, $m_2 = 3$ Thread 3, $m_3 = 7$

- Reduce: $min([3, 1, 3, 7]) = 1$.

# Optimization: Coalesced Memory Accesses

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |
| $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ |

Very important: up to an order of magnitude faster (10x).

# Optimization: Coalesced Memory Accesses

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |
| $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ |

Very important: up to an order of magnitude faster (10x).

# Optimization: Coalesced Memory Accesses

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |
| $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ |

Very important: up to an order of magnitude faster (10x).

# Optimization: Coalesced Memory Accesses

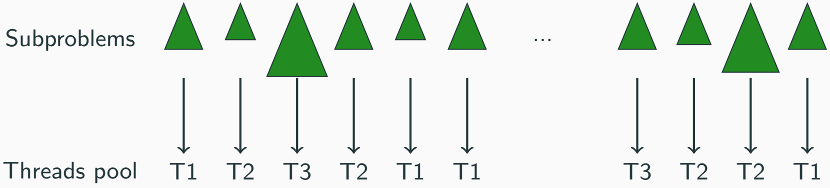
| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 10 | 23 | 21 | 7 | 91 | 1 | 3 | 10 | 42 | 11 | 8 | 7 | 32 |
| $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ | $T_3$ | $T_0$ | $T_1$ | $T_2$ |

Very important: up to an order of magnitude faster (10x).

# Challenges of Constraint Programming on GPU

# On CPU: Embarrasingly Parallel Search (EPS)[2]

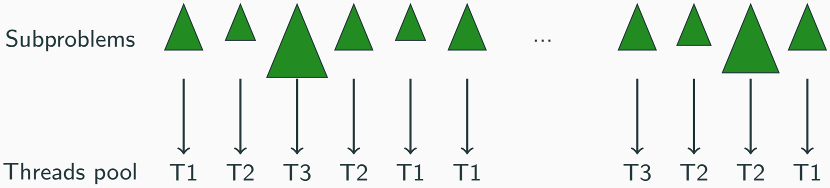
- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with $N$ the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.




[2]A. Malapert et al., 'Embarrassingly Parallel Search in Constraint Programming', JAIR, 2016

# On CPU: Embarrasingly Parallel Search (EPS)

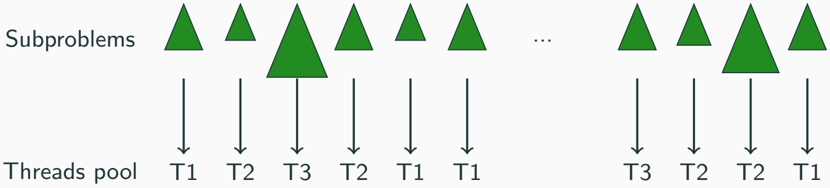
- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with $N$ the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.




$\Rightarrow$ **Other approach:** In modern solvers (e.g., Choco, OR-Tools), they use a portfolio approach (e.g., different *split* strategy on the *same problem*).

# On CPU: Embarrasingly Parallel Search (EPS)

- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with $N$ the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.




On GPU architectures, 1 subproblem per thread is not efficient (limited cache).

$\Rightarrow$ Need to parallelize propagation: $\mathbf{gfp}_d\ \mathcal{I}[\![c_1]\!] \circ \ldots \circ \mathcal{I}[\![c_n]\!]$.

# Where is the Challenge?

Parallelizing $\mathbf{gfp}_d\ \mathcal{I}[\![c_1]\!] \circ \ldots \circ \mathcal{I}[\![c_n]\!]$ is challenging because constraints share variables, and we have typical *shared state memory* issues such data races and inefficiencies.

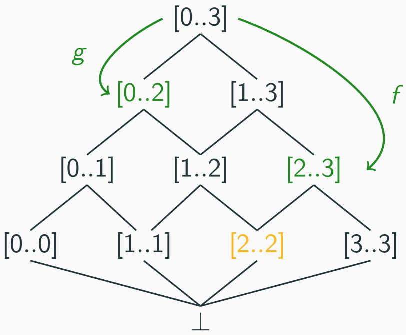
## Contributions

- **New parallel model of computation** to execute propagators in parallel[2]:
  $\mathbf{gfp}_d\ \mathcal{I}[\![c_1]\!] \parallel \ldots \parallel \mathcal{I}[\![c_n]\!]$
- **Ternary constraint network**: representation of constraints suited for GPU architectures[3].
- **First general constraint solver fully executing on GPU.**
  $\Rightarrow$ **Open-source**: Publicly available on `https://github.com/ptal/turbo`.

[2] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.
[3] P. Talbot, *Ternary Constraint Network for Efficient Integer Bound Propagation on GPU*, submitted, 2025.

# Parallel Model of Computation

# Parallel Model of Computation




- $f(x) \triangleq x \sqcap [2..\infty]$ models the constraint $x \geq 2$.
- $g(x) \triangleq x \sqcap [-\infty..2]$ models the constraint $x \leq 2$.
- Parallel execution: `f || g = [2..2]`

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$$\boxed{x = [-\infty, \infty]}$$

**Propagators:**

$$\begin{array}{lll} & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$\boxed{x = [-\infty, ?]}$

**Propagators:**

$$
\begin{array}{lll}
 & \boxed{x} \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\
\parallel & \boxed{x} \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!])
\end{array}
$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \land x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$\boxed{x = [-\infty, 5]}$

**Propagators:**

$$
\begin{array}{lll}
 & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\
\parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!])
\end{array}
$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.

$\Rightarrow$ **Solution**: Use atomic load and store!

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$$\boxed{x = [-\infty, 5]}$$

**Propagators:**

$$\begin{array}{llll} & \boxed{x} \leftarrow [-\infty, 4] & & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & \boxed{x} \leftarrow [-\infty, 5] & & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \land x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$$\boxed{x = [-\infty, 5]}$$

**Propagators:**

$$\begin{array}{llll} & x \leftarrow [-\infty, 4] & & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

**Issue 2: nondeterminism?** $x$ can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \land x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$\boxed{x = [-\infty, 4]}$

**Propagators:**

$$\begin{array}{lll} & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

**Issue 2: nondeterminism?** $x$ can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.

$\Rightarrow$ **Solution**: Use a fixpoint loop.

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \wedge x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$$\boxed{x = [-\infty, \boxed{5}]}$$

**Propagators:**

$$\begin{array}{lll} & \boxed{x} \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & \boxed{x} \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.
$\Rightarrow$ **Solution**: Use atomic load and store!
$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

**Issue 2: nondeterminism?** $x$ can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.
$\Rightarrow$ **Solution**: Use a fixpoint loop.

**Issue 3: progress?** What if $\mathcal{I}[\![x \leq 5]\!]$ is always "winning"?

# Example of Parallel Propagation

Let's consider $\mathcal{I}[\![x \leq 4 \land x \leq 5]\!] = \mathcal{I}[\![x \leq 4]\!] \parallel \mathcal{I}[\![x \leq 5]\!]$

**Memory:**

$$\boxed{x = [-\infty, 4]}$$

**Propagators:**

$$\begin{array}{lll} & x \leftarrow [-\infty, 4] & (\mathcal{I}[\![x \leq 4]\!]) \\ \parallel & x \leftarrow [-\infty, 5] & (\mathcal{I}[\![x \leq 5]\!]) \end{array}$$

**Issue 1: data race?** Parallel update of the same variable: upper bound of $x$.

$\Rightarrow$ **Solution**: Use atomic load and store!

$\Rightarrow$ **In CUDA**: Integer load and store are atomic by default!

**Issue 2: nondeterminism?** $x$ can be equal to $[-\infty, 4]$ or $[-\infty, 5]$ depending on the order of execution.

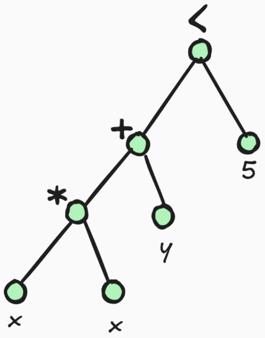
$\Rightarrow$ **Solution**: Use a fixpoint loop.

**Issue 3: progress?** What if $\mathcal{I}[\![x \leq 5]\!]$ is always "winning"?

$\Rightarrow$ **Solution**: Write in the memory only if the value is strictly lower ($\lceil x \rceil = v$ iff $v < \lceil x \rceil$).

# Ternary Constraint Network

# Representation of Propagators




- Represented using `shared_ptr` and `variant` data structures.

  ⇒ Uncoalesced memory accesses.

- Code similar to an interpreter:

```
switch(term.index()) {
  case IVar:
  case INeg:
  case IAdd:
  case IMul:
  // ...
```

  ⇒ Thread divergence.

# Ternary Normal Form (TNF)

We simplify the representation of constraints to ternary constraints of the form:

- `x = y <op> z` where `x`,`y`,`z` are variables.
- The operators are $\{+, /, *, \mathit{mod}, \mathit{min}, \mathit{max}, \leq, =\}$.

**Example**

The constraint $x + y \neq 2$ is represented by:

```
t1 = x + y
ZERO = (t1 = TWO)
```

`equivalent to` $\mathit{false} \Leftrightarrow (t1 = 2)$

where `ZERO` and `TWO` are two variables with constant values.

# Bytecode Representation

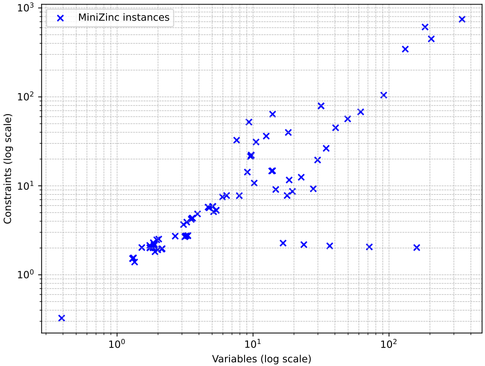
The ternary form of a propagator holds on 16 bytes:

```
struct bytecode_type {
  int op;
  int x;
  int y;
  int z;
};
```

- **Uniform representation** of propagators in memory $\Rightarrow$ coalesced memory accesses.
- Limited number of operators + sorting $\Rightarrow$ reduced thread divergence.

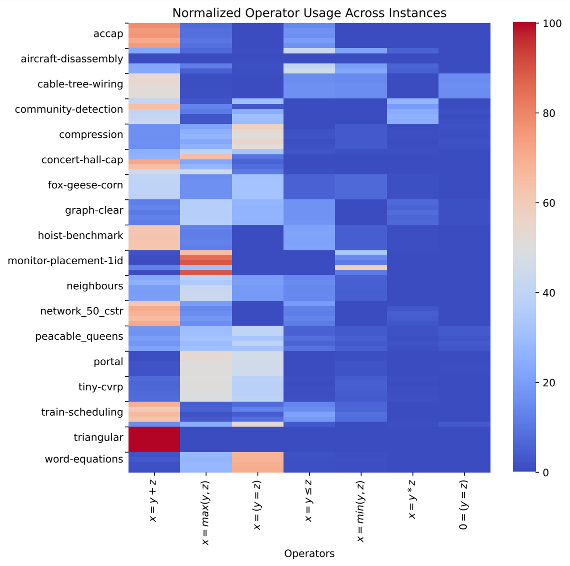
# Drawback of TNF: increase in number of propagators and variables.

Benchmark on the MiniZinc Challenge 2024 (89 instances)




The **median increase** of variables is 4.76x and propagators is 4.33x.

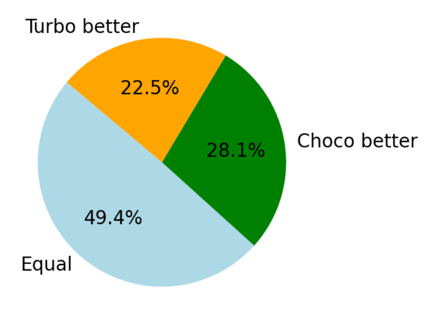
# Divergence?

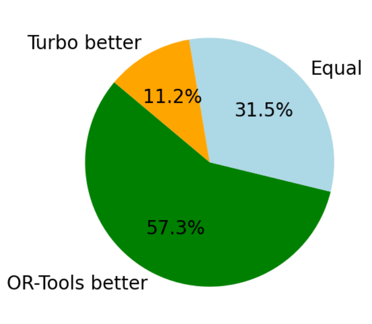


Normalized Operator Usage Across Instances
accap
aircraft-disassembly
cable-tree-wiring
community-detection
compression
concert-hall-cap
fox-geese-corn
graph-clear
hoist-benchmark
monitor-placement-1id
neighbours
network_50_cstr
peacable_queens
portal
tiny-cvrp
train-scheduling
triangular
word-equations
$x = y + z$
$x = max(y, z)$
$x = (y = z)$
$x = y \leq z$
$x = min(y, z)$
$x = y * z$
$0 = (y = z)$
Operators
100
80
60
40
20
0

# Benchmarks: Turbo vs Choco v4.18

Comparison of the best objective values found (timeout: 20 mins, GPU: H100).



# Conclusion

# Conclusion: Theoretical Parallel Model of Computation

*Data races occur rarely, so we should avoid working so much to avoid them.*

## Properties of the model

*A Variant of Concurrent Constraint Programming on GPU* (AAAI 2022)[4].

- Correct: Proofs that $P; Q \equiv P||Q$, parallel and sequential versions produce the same results.
- Restartable: Stop the program at any time, and restart on partial data.
- Weak memory consistency: Very few requirements on the underlying memory model $\Rightarrow$ wide compatibility across hardware, unlock optimization.

[4] `https://ptal.github.io/papers/aaai2022.pdf`

# Conclusion: Practical Implementation

## Turbo

- Simple: solving algorithms from 50 years ago.
  $\Rightarrow$ no global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency.
- Efficient: Almost on-par with Choco (algorithmic optimization VS hardware optimization).

https://github.com/ptal/turbo

# Conclusion: Practical Implementation

## Turbo

- Simple: solving algorithms from 50 years ago.
  $\Rightarrow$ no global constraints, nogoods learning, lazy clause generation, restart strategies, event-based propagation, trailing or recomputation-based state restoration and domain consistency.
- Efficient: Almost on-par with Choco (algorithmic optimization VS hardware optimization).

https://github.com/ptal/turbo

## But...

- Still lagging behind CP+SAT solvers, and SAT learning is inherently sequential...
- There is hope: ACE (a pure CP solver) show CP-only is still competitive in XCSP3 compet.

# Joint Projects?

- Abstract interpretation of abstract argumentation framework
  - Is it possible to use abstract interpretation for non-monotone logic?
  - Could be useful to combine abstract argumentation and FOL (not everything is non-monotone?).
- GPU-accelerated abstract argumentation
  - Does not seem to exist yet.
  - On CPU: "Cerutti, Federico, et al. *Exploiting parallelism for hard problems in abstract argumentation.*, AAAI-15"
  - Based on *strongly connected component* (SCC).
  - SCC is also useful for constraint reasoning (all-different constraint).
- Many smaller projects directly relevant to the topic of today (but not to abstract argumentation).