A GPU-based Constraint Programming Solver

The 40th Annual AAAI Conference on Artificial Intelligence (AAAI 2026)

Pierre Talbot
pierre.talbot@uni.lu
https://ptal.github.io
24th January 2026

University of Luxembourg

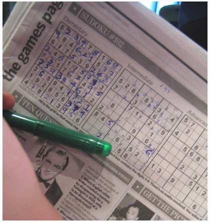
UNIVERSITÉ DU LUXEMBOURG

Constraint Programming

Constraint programming is a declarative approach to solve discrete constraint problems. In particular, it supports natively non-linear constraints.

$$x \in \{1, 2, 3\} \wedge y \in \{2, 3, 5\} \wedge x^2 - y = 5$$

A constraint solver can typically find a (best) solution satisfying the constraints.




What is the Problem?

Constraint solvers, and more largely combinatorial optimization, have not benefited yet from GPU architectures.

Why?

Support for GPU accelerate 2 251

Beneo Van

à or-tools-discuss

Occasionally, I found that I can apply for

Do you have plans to support GPUs?

Thanks

Laurent Perron

à or-tools-discuss

No, gpu are good on dense structures.

Sat is sparse by nature.

What is the Problem?

Constraint solvers, and more largely combinatorial optimization, have not benefited yet from GPU architectures.

Why?

For 50 years+, constraint solvers have been primarily designed for CPU architectures.

Have we really tried on GPU architectures??

State of the Art: Combinatorial Optimization on GPU

Do we have an **exact** and **general-purpose** constraint solver **running on GPU**?

- **Incomplete**, general-purpose, full GPU: Often population-based algorithms[1].
- Complete, **not general**, full GPU: Specific algorithms[2]
- Complete, general-purpose, **hybrid CPU/GPU**:
  - offloading to GPU specialized filtering procedures[3,4].
  - **cuOpt**: new MILP solver—relaxation on GPU, search on CPU[5].

[1] A. Arbelaez and P. Codognet, *A GPU Implementation of Parallel Constraint-Based Local Search*, PDP, 2014.
[2] Jan Gmys. Exactly Solving Hard Permutation Flowshop Scheduling Problems on Peta-Scale GPU-Accelerated Supercomputers. INFORMS Journal on Computing, 2022.
[3] F. Campeotto et al., *Exploring the use of GPUs in constraint solving*, PADL, 2014
[4] F. Tardivo et al., *Constraint propagation on GPU: A case study for the AllDifferent constraint*, Journal of Logic and Computation, 2023.
[5] Using primal-dual linear programming (PDLP).

Our Contributions

Turbo: a general and exact constraint solver fully executing on GPU (propagation + search).

Main Characteristics

- **Simple**: interval-based constraint solving + backtracking search (no global constraints, learning, restart, event-based propagation, ...).
- **Efficient?** Almost on-par with Choco on the quality of found objective bounds within 20mins. Can beat OR-Tools on some instances.

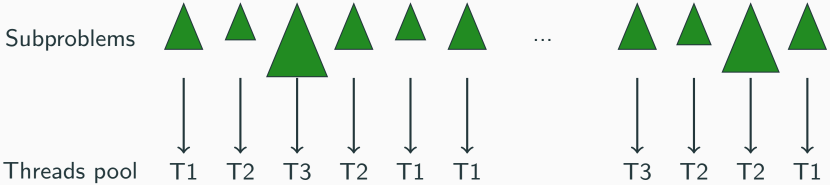
https://github.com/ptal/turbo

CPU-based Parallel Constraint Solvers

And Why The Same Techniques Do Not Work on GPU

On CPU: Embarrasingly Parallel Search (EPS)[6]

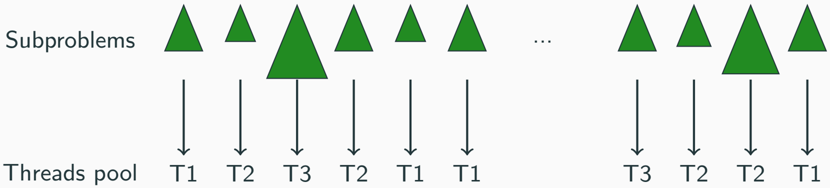
Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).



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

On CPU: Embarrasingly Parallel Search (EPS)

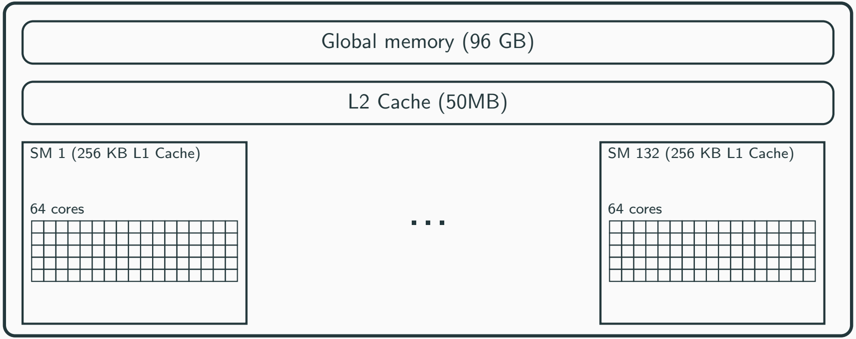
Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).



⇒ **Other approach:** portfolio approach (e.g., different search strategy on the *same problem*) as seen in Choco and OR-Tools.

Each thread works on its own copy of the problem.

On GPU: 1 Subproblem per Thread Takes too Much Memory



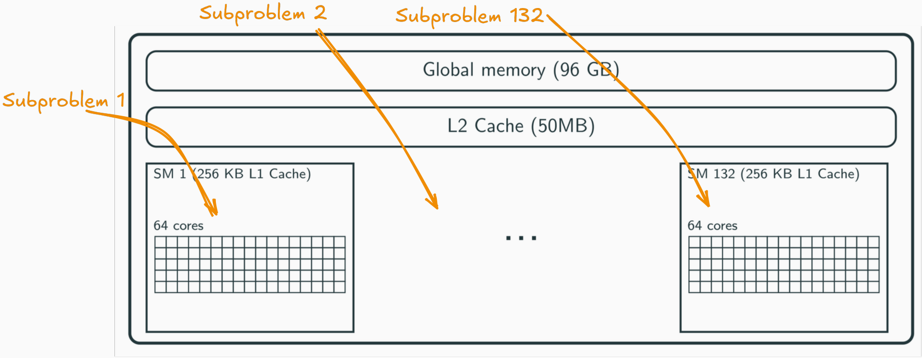
8448 cores grouped in 132 streaming multiprocessors (SM) of 64 cores each (H100).

$\Rightarrow$ **Oversubscribe** (to hide memory latency): 1024 threads per SM

135168 threads running in parallel!

For a 1MB constraint problem: 135GB of memory...

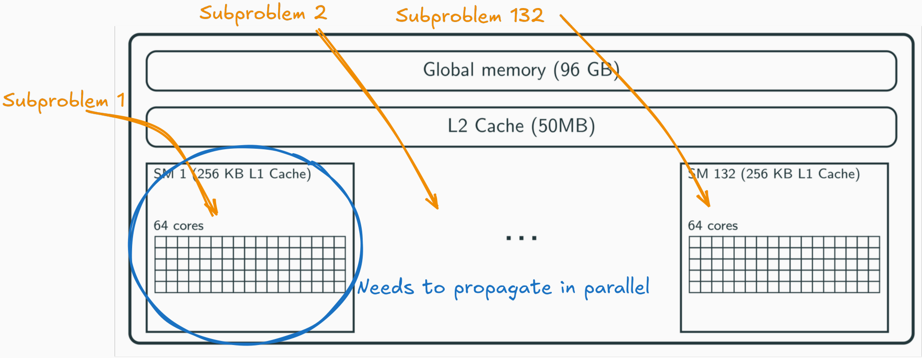
Search on GPU with EPS: one subproblem per SM[6]



To address load balancing issue, we create more subproblems than SMs (EPS). (more details in the paper for the algorithm).

[6] More precisely, one subproblem per GPU block.

Propagation on GPU



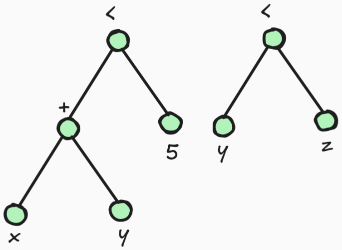
We proposed a correct model of lock-free parallel propagation[7].

[7] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

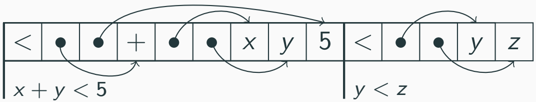
Representation of Constraints in AAAI 2022

Constraints: $x + y < 5$ and $y < z$

Syntax tree



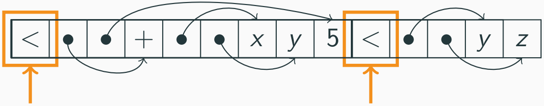
In memory



One Constraint per GPU Thread?

Constraint propagation is essentially traversing the syntax tree, but it is not efficient on GPU:

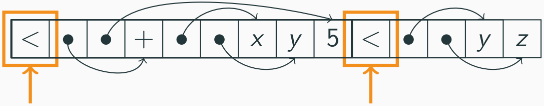
- **Non coalesced memory accesses**: 2 non-adjacent reads trigger 2 memory transactions.



One Constraint per GPU Thread?

Constraint propagation is essentially traversing the syntax tree, but it is not efficient on GPU:

- **Non coalesced memory accesses**: 2 non-adjacent reads trigger 2 memory transactions.

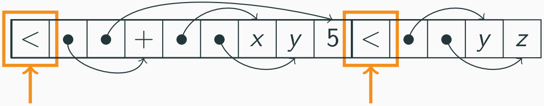


- **Load imbalance**: The first thread needs to do more work than the second.

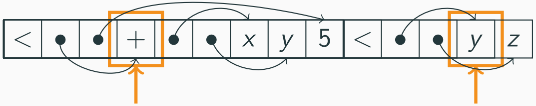
One Constraint per GPU Thread?

Constraint propagation is essentially traversing the syntax tree, but it is not efficient on GPU:

- **Non coalesced memory accesses**: 2 non-adjacent reads trigger 2 memory transactions.



- **Load imbalance**: The first thread needs to do more work than the second.
- **Thread divergence**: The code path is different on different kind of nodes.



Our Solution: Ternary Constraint Network

Ternary Constraint Network

Each constraint takes a regular form $x = y \odot z$ with $x, y, z \in X$ (no constant) and $\odot \in \{+, /, *, mod, min, max, \leq, =\}$.

Example

Constraints:

$$x + y < 5 \quad \rightsquigarrow \quad \boxed{t = x + y} \wedge t < 5$$

$$y < z \quad \rightsquigarrow \quad \boxed{y = u + z} \wedge u < 0$$

(introduce auxiliary variables t and u)

In memory

$+$	t	x	y	$+$	y	u	z
$t = x + y$				$y = u + z$			

One Constraint per GPU Thread!

Ternary constraints are compactly stored and efficiently accessed:

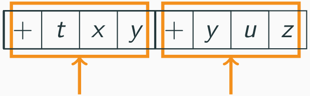
- **Coalesced memory accesses**: Adjacent vectorized 16-byte loads.

$+$	t	x	y	$+$	y	u	z

One Constraint per GPU Thread!

Ternary constraints are compactly stored and efficiently accessed:

- **Coalesced memory accesses**: Adjacent vectorized 16-byte loads.

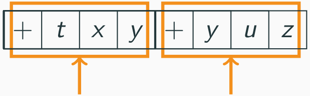


- **Uniform load balancing**: each thread performing the same work.

One Constraint per GPU Thread!

Ternary constraints are compactly stored and efficiently accessed:

- **Coalesced memory accesses**: Adjacent vectorized 16-byte loads.



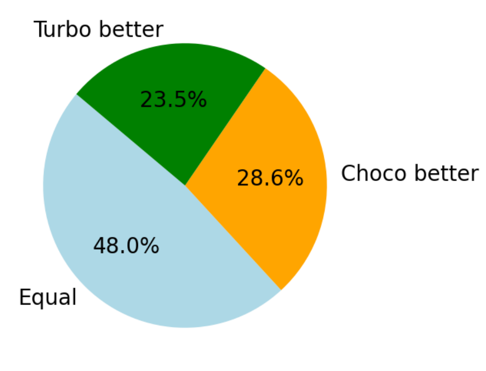
- **Uniform load balancing**: each thread performing the same work.
- **Thread divergence**: reduced by sorting constraints on $\odot$.

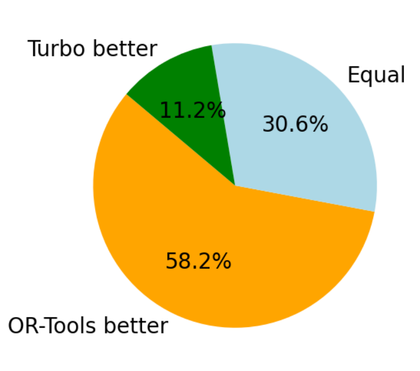
10x speed-up with ternary constraints.

Benchmark: 1-to-1 Comparison

On 98 instances of the MiniZinc 2024 competition.[8]

Comparison of the best objective values found. Timeout 20 minutes, GPU H100.





[8]1 instance unsat at root, 1 instance for which TCN is too large.

Conclusion

Turbo: General-purpose GPU constraint solver

- 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).
- Many possible optimizations to improve the efficiency, but need to be redesigned for GPU.

https://github.com/ptal/turbo

More Information...

Experimental Evaluation

On 98 instances of the MiniZinc 2024 competition.[9]

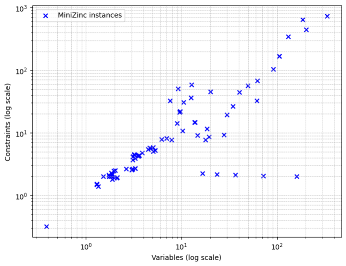
Timeout 20 minutes, CPU 64 cores, GPU H100.

solver	MiniZinc score	#Optimal
Or-Tools 9.9 (64 threads)	266.7	82
Choco 4.10.18 (64 threads)	190.8	44
Or-Tools 9.9 (fixed search)	119.9	37
Choco 4.10.18 (fixed search)	49.3	25
Turbo 1.2.8 (fixed search)	45.8	20

[9] 1 instance unsat at root, 1 instance for which TCN is too large.

Drawback of Ternary Constraint Networks

Benchmark on the MiniZinc Challenge 2024 (96 instances)[10].

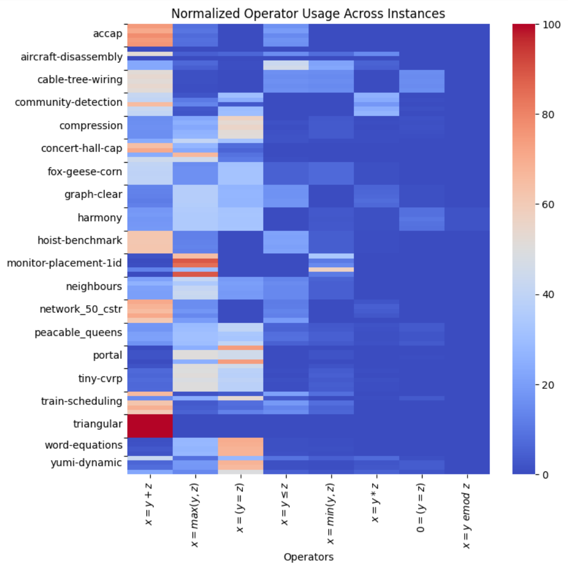


Increase in number of propagators and variables:

- The **median increase** of variables is 4.45x and propagators is 4.34x.
- The **maximum increase** of variables is 336x and propagators is 731x.

[10] Instances not solved during preprocessing.

Divergence?



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

Lock-free Parallel Propagation

Example of Parallel Propagation[11]

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

Memory:

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

Propagators:

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

[11] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

Example of Parallel Propagation[11]

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

Memory:

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

Propagators:

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

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

[11] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

Example of Parallel Propagation[11]

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

Memory:

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

Propagators:

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

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

$\Rightarrow$ **Solution**: fixpoint + fair scheduling + strict updates (`if (v < x.ub) { x.ub = v; }`).

[11] P. Talbot et al., *A Variant of Concurrent Constraint Programming on GPU*, AAAI, 2022.

GPU Fixpoint Algorithm

```
__device__ void fixpoint(Store& d, Props* props, int n) {
  __shared__ bool has_changed = true;
  // Keep going until no variable domain is modified.
  while(has_changed) {
    __syncthreads(); has_changed = false; __syncthreads();
```

GPU Fixpoint Algorithm

```
__device__ void fixpoint(Store& d, Props* props, int n) {
  __shared__ bool has_changed = true;
  // Keep going until no variable domain is modified.
  while(has_changed) {
    __syncthreads(); has_changed = false; __syncthreads();
    // Execute all propagators (similar to AC1)
    for(int i = threadIdx.x; i < n; i += blockDim.x) {
      has_changed |= props[i].propagate(d) ;
    }
    __syncthreads();
  }
}
```

GPU Challenges

- Coalesced memory accesses of the propagator representation `props[i]`.
- Avoiding divergence in `propagate`.