Ternary Constraint Network for Efficient Bound Propagation on GPU

Special Meeting 2

Pierre Talbot
pierre.talbot@uni.lu
https://ptal.github.io

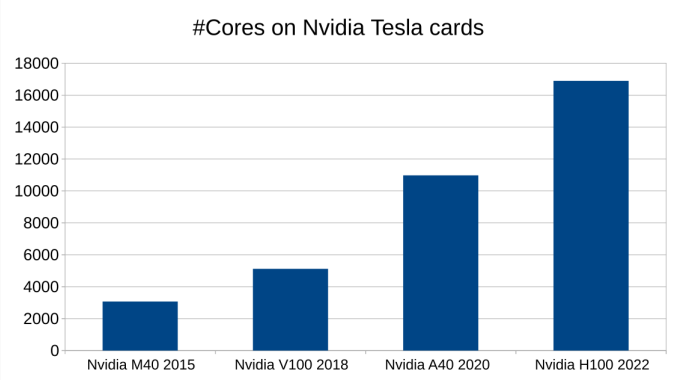
20 June 2025

University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

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?

cuOpt: Nvidia Routing Optimization

Record-Breaking NVIDIA cuOpt Algorithms Deliver Route Optimization Solutions 100x Faster

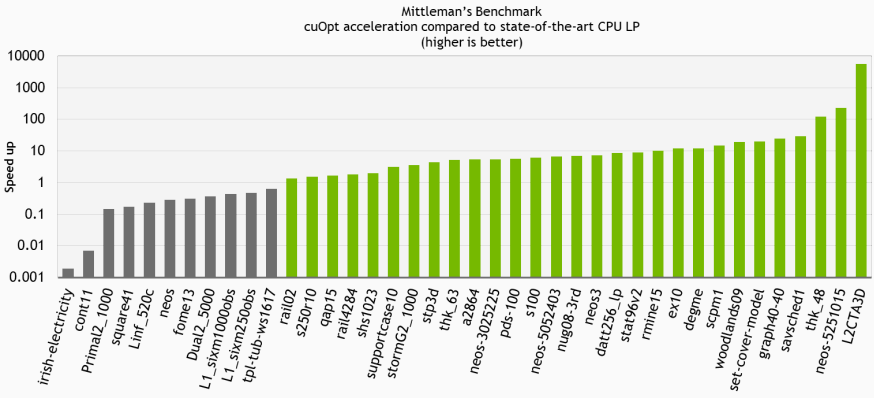
Mar 20, 2024

+20 Like Discuss (0)

By Akif Çördük, Piotr Sielski and Moon Chung

Cool, but specialized combinatorial algorithm for routing.

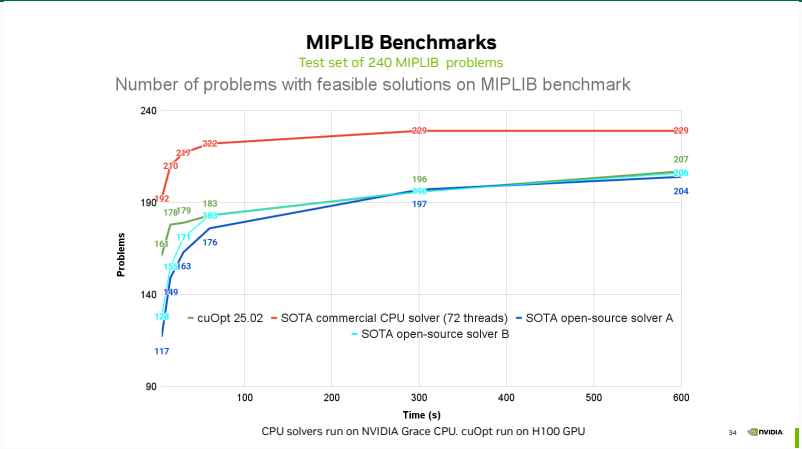
cuOpt: Nvidia Linear Programming Solver



Use a new linear solving algorithm primal-dual hybrid gradient (PDHG/PDPL) (2011, 2021) which relies on matrix multiplication.

https://developer.nvidia.com/blog/accelerate-large-linear-programming-problems-with-nvidia-cuopt/

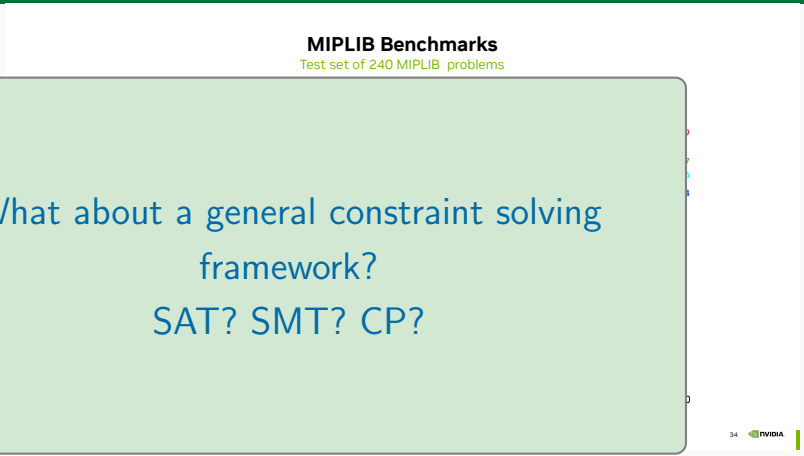
cuOpt: Nvidia Hybrid MIP Solver



Relaxation on GPU, search on CPU.

https://www.nvidia.com/en-us/on-demand/session/gtc25-s72290/

cuOpt: Nvidia Hybrid MIP Solver



Relaxation on GPU, search on CPU.

https://www.nvidia.com/en-us/on-demand/session/gtc25-s72290/

Combinatorial Optimization on GPU

Very scarce literature, usually:

- **Heuristics**: often population-based algorithms[1].
- **Limited set of problems**[2]
- **Limited GPU parallelization**: offloading to GPU specialized filtering procedures[3,4].
- **Limited expressivity**: solver with max 256 set variables[5].

For CP-based approach: not general and no proof of correctness.

[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] A. Dovier et al., *CUDA: Set Constraints on GPUs*, 2022.

Our Contributions

Contributions

- **Ternary constraint network**: representation of constraints suited for GPU architectures.
- **Search on GPU**: distributing one subproblem per GPU block.
- **First general constraint solver fully executing on GPU.**
  ⇒ **Open-source**: Publicly available on `https://github.com/ptal/turbo`.
  ⇒ **General**: Support MiniZinc and XCSP3 constraint models.
  ⇒ **Efficient?**: Almost on-par with Choco (23% better, 28% worst, 49% equal).

Constraint Programming on GPU

The rest of this talk:

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

Constraint Programming

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 $d \in X \to Itv$ is the *domain* of the variables where Itv is the set of intervals.

Example

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

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

Propagation

Let c be a constraint, then a propagator is a sound and monotone function:

- $\mathcal{I}[\![c]\!] \in (X \to Itv) \to (X \to Itv)$
- **Example**: let $d = \{x \mapsto [0,5], y \mapsto [5,10]\}$, then $\mathcal{I}[\![x = y]\!]d = \{x \mapsto [5,5], y \mapsto [5,5]\}$.

Let $\langle d, \{c_1, \ldots, c_n\}\rangle$ be a constraint network, propagation is the computation of the greatest fixpoint:

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

Propagate and Search

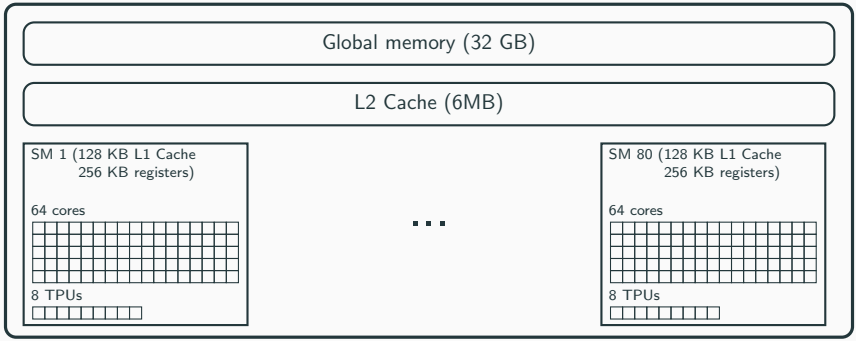
The main algorithm behind constraint solvers:

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
    $\langle d_1, \ldots, d_n \rangle \leftarrow \texttt{split}(d)$
    return $\bigcup_{i=1}^{n} \texttt{solve}(d_i, C)$
  end if
end function

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

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`

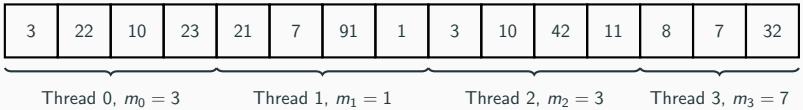
Programming Challenges

- **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:

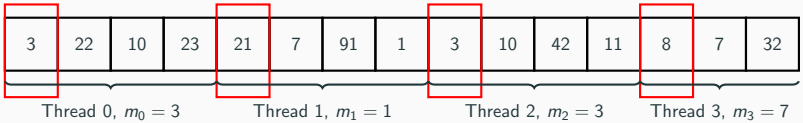


- Reduce: $min([3, 1, 3, 7]) = 1$.

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

Iteration 1:

- Map:

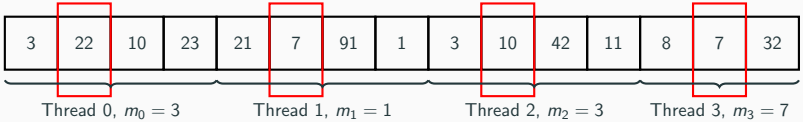


- Reduce: $min([3, 1, 3, 7]) = 1$.

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

Iteration 2:

- Map:

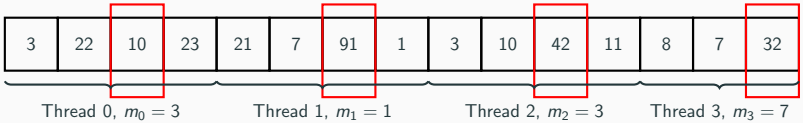


- Reduce: $min([3, 1, 3, 7]) = 1$.

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

Iteration 3:

- Map:

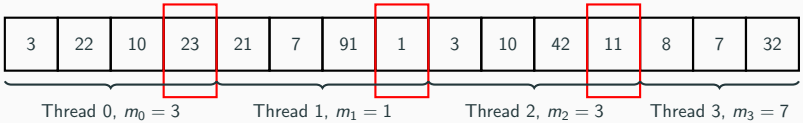


- Reduce: $min([3, 1, 3, 7]) = 1$.

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

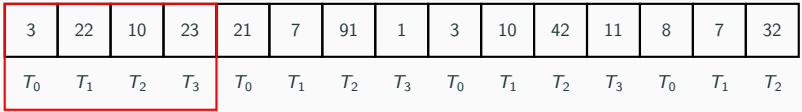
Iteration 4:

- Map:



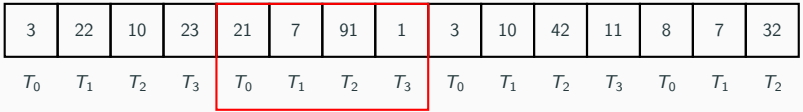
- Reduce: $min([3, 1, 3, 7]) = 1$.

Optimization: Coalesced Memory Accesses



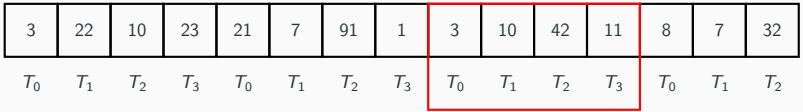
Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses



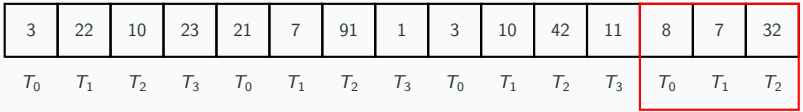
Very important: up to an order of magnitude faster (10x).

Optimization: Coalesced Memory Accesses



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

Optimization: Coalesced Memory Accesses

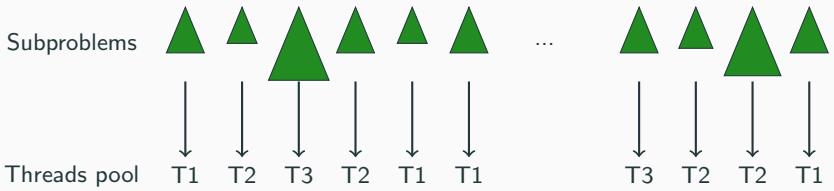


Very important: up to an order of magnitude faster (10x).

Challenges of Constraint Programming on GPU

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

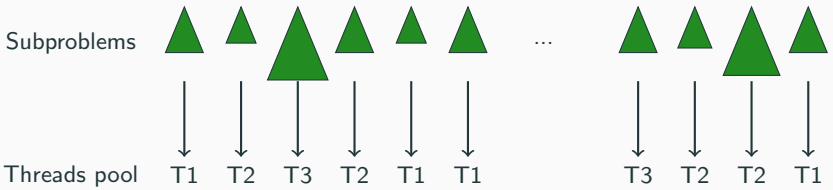
- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.



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

On CPU: Embarrasingly Parallel Search (EPS)

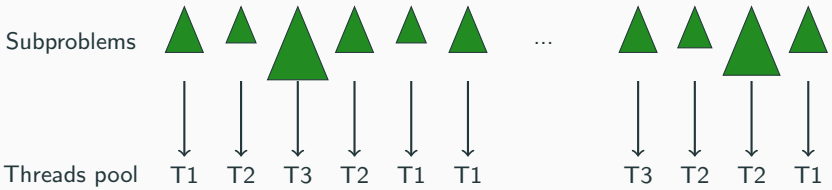
- **Idea**: Divide the problem into many subproblems beforehand (e.g. $N \times 30$ with N the number of threads).
- **Intuition**: Statistically, there is little chance a subproblem takes longer than the sum of the other subproblems.



⇒ **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 as data races and inefficiencies.

Solution

- **Lock-free parallel model of computation** to execute propagators in parallel[6]:
  $\mathbf{gfp}_d\ \mathcal{I}[\![c_1]\!] \parallel \ldots \parallel \mathcal{I}[\![c_n]\!]$
- **Ternary constraint network**: representation of constraints suited for GPU architectures.
- **First general constraint solver fully executing on GPU.**
  $\Rightarrow$ **Open-source**: Publicly available on `https://github.com/ptal/turbo`.

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

Parallel Model of Computation

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:

$x = [-\infty, \infty]$

Propagators:

$x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!])$

$\parallel$

$x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!])$

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:

$x = [-\infty, ?]$

Propagators:

$x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!])$

$\parallel$

$x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!])$

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

Example of Parallel Propagation

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

Memory:

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

Propagators:

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

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:

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

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

Memory:

$x = [-\infty, 5]$

Propagators:

$x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!])$

$\parallel$

$x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!])$

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

Memory:

$x = [-\infty, 4]$

Propagators:

$x \leftarrow [-\infty, 4] \quad (\mathcal{I}[\![x \leq 4]\!])$

$\parallel$

$x \leftarrow [-\infty, 5] \quad (\mathcal{I}[\![x \leq 5]\!])$

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**: fixpoint + fair scheduling + strict updates.

Strict updates: `if (v < x.ub) { x.ub = v; }`

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

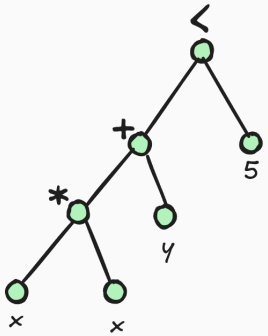
```
__device__ void fixpoint(Store& d, Props* props, int n) {
  __shared__ bool has_changed = true;
  // Keep going until no variable domain is modified.
  while(has_changed) {
    __syncthreads(); has_changed = false; __syncthreads();
    // Execute all propagators (similar to AC1)
    for(int i = threadIdx.x;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

Challenges

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

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 $\{+, /, *, mod, min, max, \leq, =\}$.

Example

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

```
t1 = x + y
ZERO = (t1 = TWO)                    equivalent to false ⇔ (t1 = 2)
```

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

Bytecode Representation

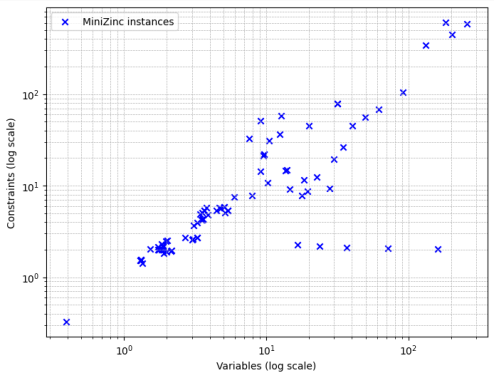
The ternary form of a propagator holds on 16 bytes:

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

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

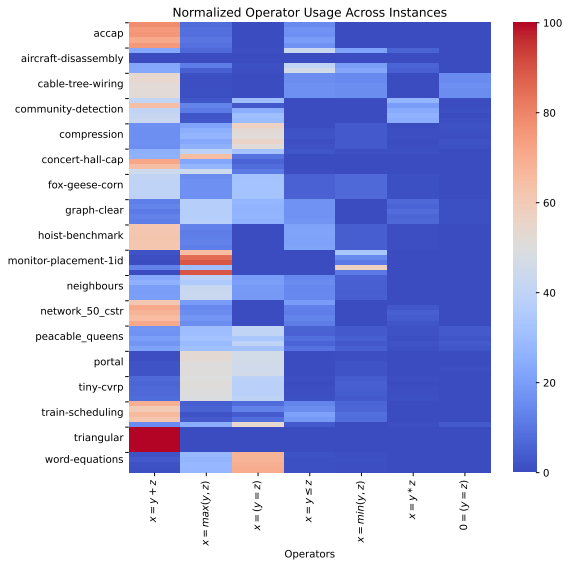
Drawback of TCN: increase in number of propagators and variables.

Benchmark on the MiniZinc Challenge 2024 (95 instances)



The **median increase** of variables is 4.46x and propagators is 4.85x.
The **maximum increase** of variables is 258x and propagators is 607x.

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 ≤ z
x = min(y, z)
x = y * z
0 = (y = z)
Operators
100
80
60
40
20
0

Search on GPU

Benchmarking

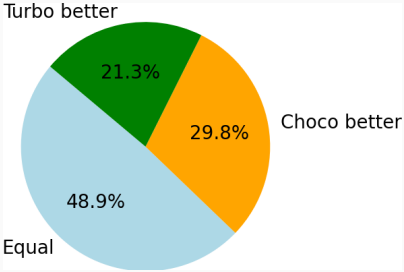
Experimental Evaluation

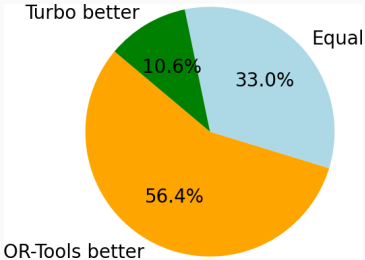
On 95 instances of the MiniZinc 2024 competition.
5 instances discarded (yumi-static) due to out of memory error with TCN.
Timeout 20 minutes, CPU 64 cores, GPU H100.

solver	MiniZinc score	#Optimal
Or-Tools 9.9 (64 threads)	312.9	80
Choco 4.10.18 (64 threads)	233.0	43
Choco 4.10.18 (free search)	159.5	32
Or-Tools 9.9 (fixed search)	130.2	36
Choco 4.10.18 (fixed search)	56.6	25
Turbo 1.2.8 (fixed search)	52.3	20

1-to-1 Comparison

Comparison of the best objective values found.





Conclusion

Conclusion

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

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.