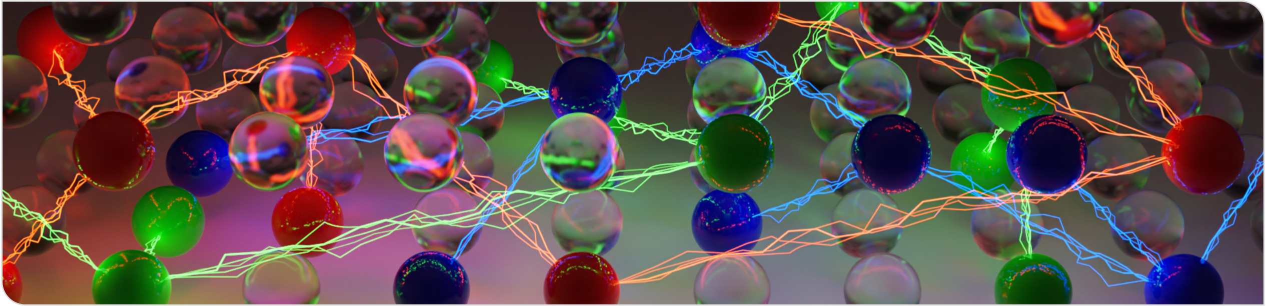
Karlsruhe Institute of Technology

Practical SAT Solving

Lecture 9: Parallel SAT Solving

T. Balyo, M. Iser, D. Schreiber | May 13, 2024



Outline

Parallel SAT solving approaches

- Basic search space splitting
- Clause sharing
- Cube&Conquer
- Portfolio solvers (without and with clause sharing)

A deep dive into Mallob

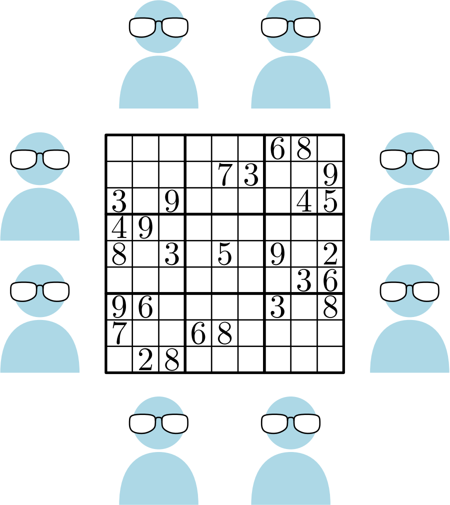
- Overview
- Scalable clause sharing
- Experiments and results

Parallel Portfolios: An analogy

The Assembly of Nerds

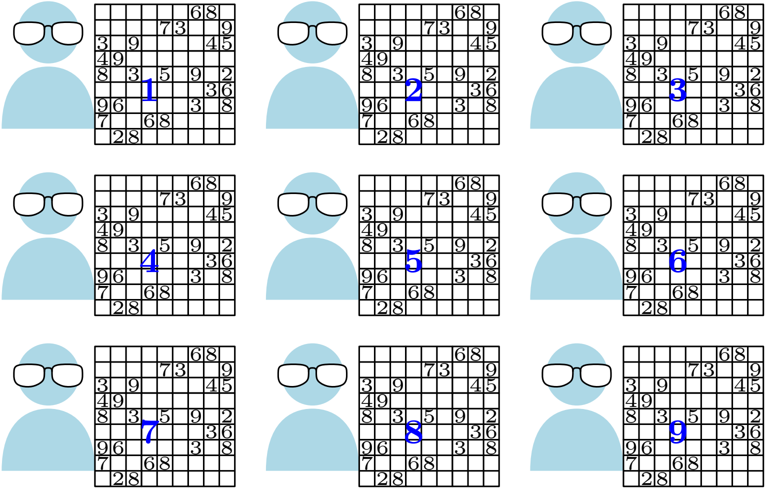
- Complex and large logic puzzle
- *n* puzzle experts at your disposition

How do we employ and “orchestrate” our experts?

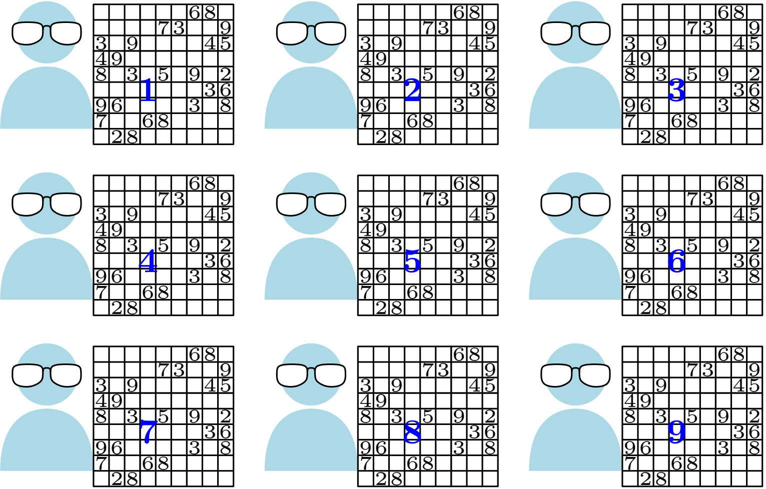


						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

Approach I: Search Space Partitioning



Approach I: Search Space Partitioning



- Partition search space at some decisions
  ⇒ Independent subproblems

Explicit Partitioning

1st Parallel DPLL Implementation by Böhm & Speckenmeyer (1994)

Explicit Load Balancing

- Completely distributed (no leader / worker roles)
- A list of partial assignments is generated
- Each process receives the entire formula and a few partial assignments
- Each process can be worker or balancer:
  - Worker: solve or split the formula, use the partial assignments
  - Balancer: estimate workload, communicate, stop
- Switch to balancer whenever worker is finished

Explicit Partitioning

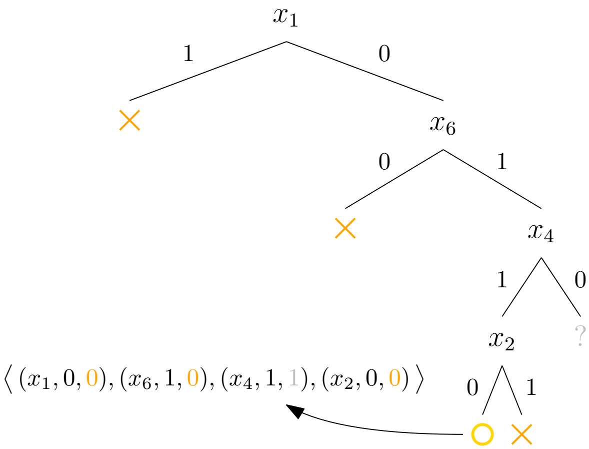
"**PSATO**: a Distributed Propositional Prover and its Application to Quasigroup Problems", Zhang et al., 1996

Centralized leader-worker architecture

- Communication only between leader and workers
- Leader assigns partial assignments using Guiding Path
  - Each node in the search tree is open or closed
    — closed = branch is explored / proven unsat
  - Leader splits open nodes and assigns job to workers
- Workers return Guiding Path when terminated by leader
- Modern features of fault tolerance, preemption of solving tasks

Explicit Partitioning

Guiding Path: List of triples (variable, branch, open)



$$\langle\, (x_1, 0, 0), (x_6, 1, 0), (x_4, 1, 1), (x_2, 0, 0) \,\rangle$$

Explicit Partitioning

SATZ (Jurkowiak et al., 2001) improves PSATO

Work stealing for workload balancing

- An idle worker requests work from the leader
- The leader splits the work of the most loaded worker
- The idle worker and most loaded worker get the parts

Clause Sharing Parallel Solvers

PaSAT (Blochinger et al., 2001)

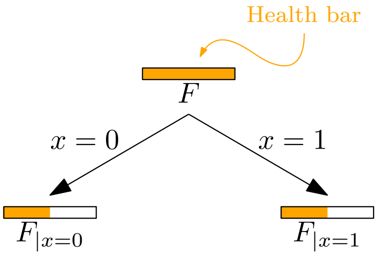
- First parallel CDCL with clause sharing
- Similar to PSATO/SATZ: leader/worker, guiding path, work stealing

ySAT (Feldman et al., 2004)

- First shared-memory parallel solver
- Multi-core processors started to be popular
- uses same techniques as the previous solvers (guiding path etc.)

... and many many more similar solvers

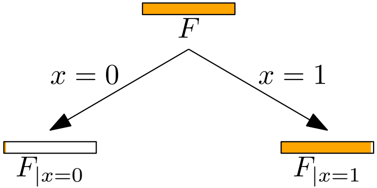
Problems with Partitioning



What we want: **Even splits**

- Split yields sub-formulas of similar difficulty
- Balanced partitioning of work
- Few or no dynamic (re-)balancing needed

Problems with Partitioning



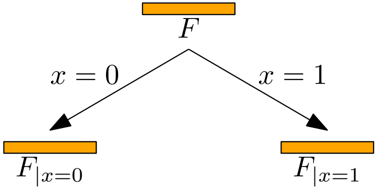
What we want: **Even splits**

- Split yields sub-formulas of similar difficulty
- Balanced partitioning of work
- Few or no dynamic (re-)balancing needed

Uneven splits

- One subformula is trivial, the other is just as hard as F
- Ping-pong effect for workers processing trivial formulae, communication / synchronization dominates run time

Problems with Partitioning



What we want: **Even splits**

- Split yields sub-formulas of similar difficulty
- Balanced partitioning of work
- Few or no dynamic (re-)balancing needed

Uneven splits

- One subformula is trivial, the other is just as hard as F
- Ping-pong effect for workers processing trivial formulae, communication / synchronization dominates run time

Bogus splits

- Both $F_{|x=0}$ and $F_{|x=1}$ are just as hard as F
- Divide&Conquer becomes Multiply&Surrender!

Cube and Conquer

The Cube&Conquer paradigm (Heule & Biere, 2011)

Generate a large amount (millions) of partial assignments (“**cubes**”)
and randomly assign them to workers.

Cube and Conquer

The Cube&Conquer paradigm (Heule & Biere, 2011)

Generate a large amount (millions) of partial assignments (“**cubes**”) and randomly assign them to workers.

- Unlikely that any of the workers will run out tasks
  $\Rightarrow$ Hope of good load balancing in practice
- Partial assignments are generated using a look-ahead solver (breadth-first search up to a limited depth)
- Best performance mostly with problem-specific decision heuristics

Cube and Conquer

The Cube&Conquer paradigm (Heule & Biere, 2011)

Generate a large amount (millions) of partial assignments (“**cubes**”) and randomly assign them to workers.

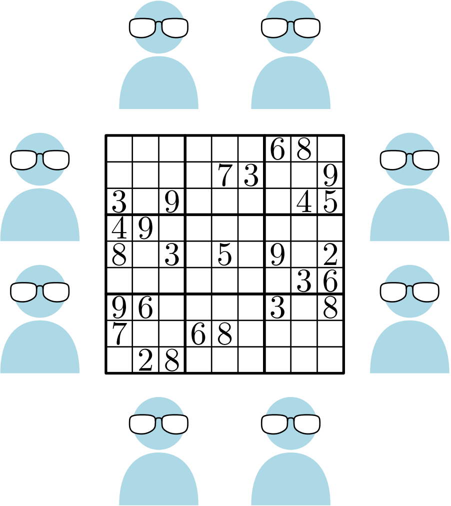
- Unlikely that any of the workers will run out tasks
  $\Rightarrow$ Hope of good load balancing in practice
- Partial assignments are generated using a look-ahead solver (breadth-first search up to a limited depth)
- Best performance mostly with problem-specific decision heuristics
- State-of-the-art for hard combinatorial problems
  - Used to solve the “Pythagorean Triples” problem ($\sim$200TB proof)
  - ... or more recently “Schur Number 5” ($\sim$2PB proof)
- Examples: March (Heule) + iLingeling (Biere) introduced in 2011; Treengeling (Biere)

Parallel Portfolios: An analogy

The Assembly of Nerds

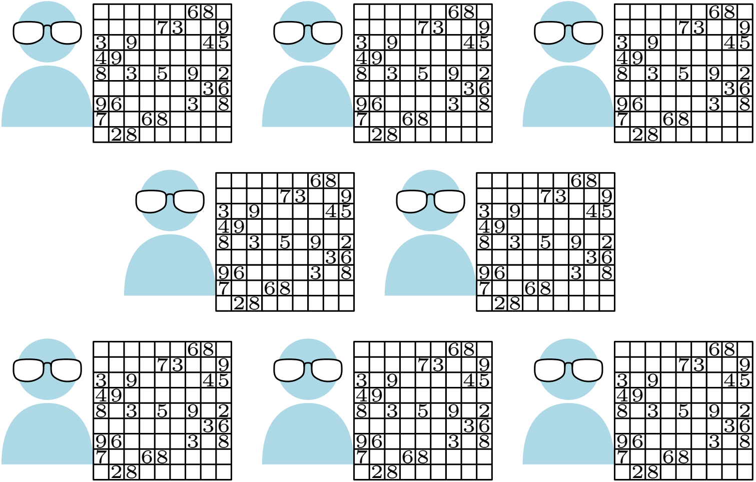
- Complex and large logic puzzle
- *n* puzzle experts at your disposition
  - — individual mindsets, approaches, strengths & weaknesses
  - — anti-social: work best if left undisturbed

How do we employ and “orchestrate” our experts?



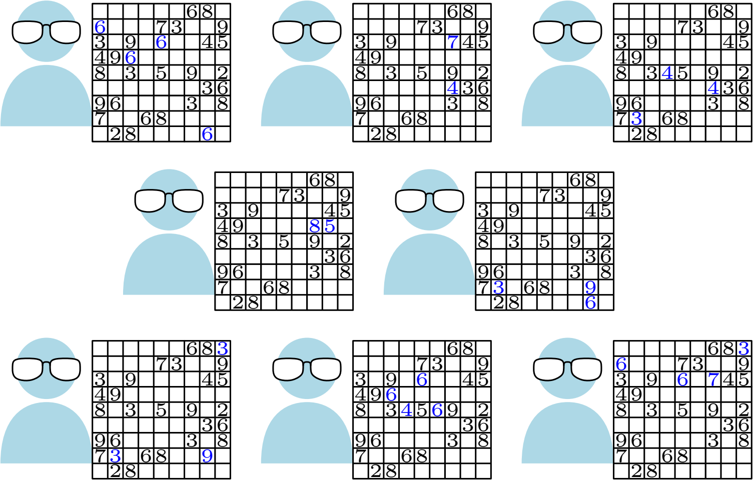
						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

Approach II: Pure Portfolio



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

Approach II: Pure Portfolio



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

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

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

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

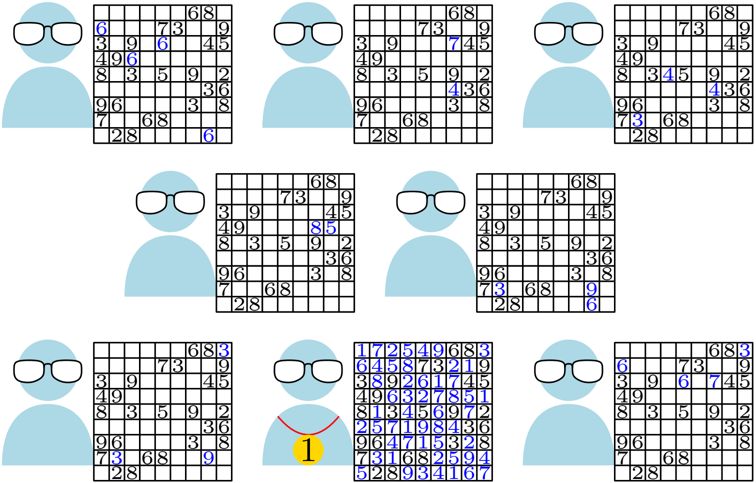
						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7	3		6	8			9	
	2	8					6	

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

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

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

Approach II: Pure Portfolio



Pure Portfolio: Oracle view vs. Speedup view

Virtual Best Solver (VBS) / Oracle

Consider n algorithms $A_1, \ldots, A_n$ where for each input x, algorithm A_i has run time $T_{A_i}(x)$.
The Virtual Best Solver (VBS) for $A_1, \ldots, A_n$ has run time $T^*(x) = \min\{T_{A_1}(x), \ldots, T_{A_n}(x)\}$.

Pure Portfolio: Oracle view vs. Speedup view

Virtual Best Solver (VBS) / Oracle

Consider n algorithms $A_1, \ldots, A_n$ where for each input x, algorithm A_i has run time $T_{A_i}(x)$.
The Virtual Best Solver (VBS) for $A_1, \ldots, A_n$ has run time $T^*(x) = \min\{T_{A_1}(x), \ldots, T_{A_n}(x)\}$.

Optimist: A pure portfolio simulates the VBS using parallel processing!

- On idealized hardware, we “select” best sequential solver for each instance

Pure Portfolio: Oracle view vs. Speedup view

Virtual Best Solver (VBS) / Oracle

Consider n algorithms $A_1, \ldots, A_n$ where for each input x, algorithm A_i has run time $T_{A_i}(x)$.
The Virtual Best Solver (VBS) for $A_1, \ldots, A_n$ has run time $T^*(x) = \min\{T_{A_1}(x), \ldots, T_{A_n}(x)\}$.

Optimist: A pure portfolio simulates the VBS using parallel processing!

- On idealized hardware, we "select" best sequential solver for each instance

Parallel speedup

Given parallel algorithm P and input x, the speedup of P is defined as $s_P(x) = T_Q(x)/T_P(x)$
where Q is the best available sequential algorithm.

Pure Portfolio: Oracle view vs. Speedup view

Virtual Best Solver (VBS) / Oracle

Consider n algorithms $A_1, \ldots, A_n$ where for each input x, algorithm A_i has run time $T_{A_i}(x)$.
The Virtual Best Solver (VBS) for $A_1, \ldots, A_n$ has run time $T^*(x) = \min\{T_{A_1}(x), \ldots, T_{A_n}(x)\}$.

Optimist: A pure portfolio simulates the VBS using parallel processing!

- On idealized hardware, we “select” best sequential solver for each instance

Parallel speedup

Given parallel algorithm P and input x, the speedup of P is defined as $s_P(x) = T_Q(x)/T_P(x)$
where Q is the best available sequential algorithm.

Pessimist: A pure portfolio never achieves actual speedups!

- There is always a sequential algorithm performing at least as well
- Consequence: Not resource efficient, not scalable

Pure SAT Portfolios

ppfolio: Winner of Parallel Track in the 2011 SAT Competition

- Just a bash script combining the best sequential solvers from 2010:

```
~$ ./solver1 f.cnf & ./solver2 f.cnf & ./solver3 f.cnf & ./solver4 f.cnf
```

- Bits by O. Roussel, the author of `ppfolio`:

— “*by definition the best solver on Earth*”
— “*probably the laziest and most stupid solver ever written*”

Pure SAT Portfolios

ppfolio: Winner of Parallel Track in the 2011 SAT Competition

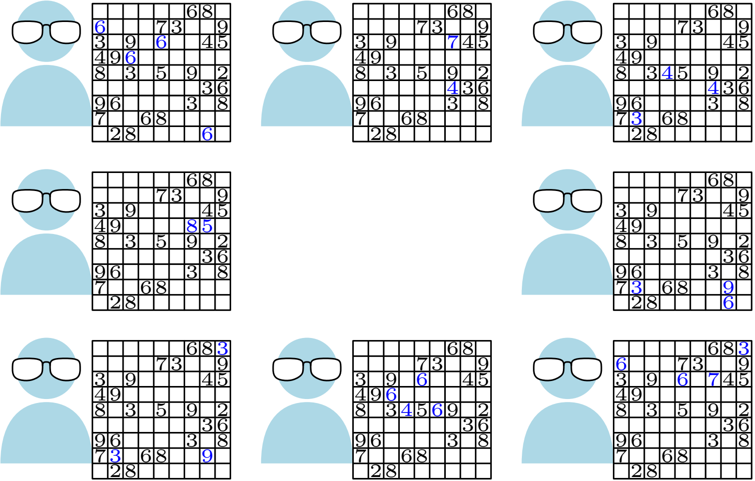
- Just a bash script combining the best sequential solvers from 2010:

```
˜$ ./solver1 f.cnf & ./solver2 f.cnf & ./solver3 f.cnf & ./solver4 f.cnf
```

- Bits by O. Roussel, the author of `ppfolio`:
  — "*by definition the best solver on Earth*"
  — "*probably the laziest and most stupid solver ever written*"

- Rationale: Different solvers are designed differently, excel on different instances
  — hope of orthogonal search behavior
- Pure portfolios no longer permitted in SAT Competitions

Approach II+: Cooperative Portfolio



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

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

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

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

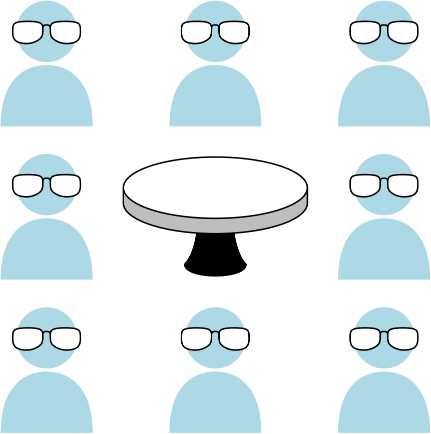
						6	8	
				7	3			9
3		9					4	5
4	9							
8		3		5		9		2
							3	6
9	6					3		8
7	3		6	8			9	
	2	8					6	

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

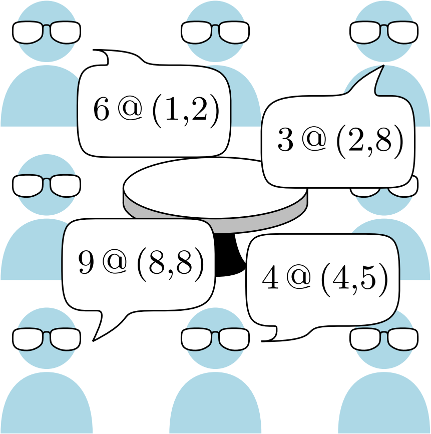
						6	8	
				7	3			9
3		9		6			4	5
4	9	6						
8		3	4	5	6	9		2
							3	6
9	6					3		8
7			6	8				
	2	8						

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

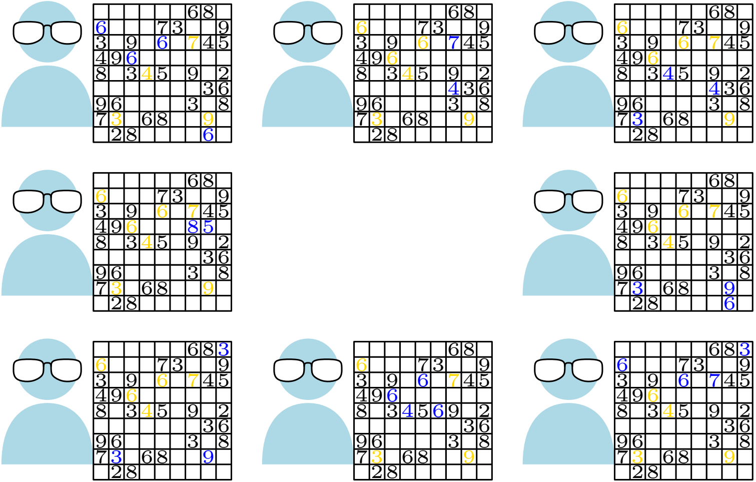
Approach II+: Cooperative Portfolio



Approach II+: Cooperative Portfolio



Approach II+: Cooperative Portfolio



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

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

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

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

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

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

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

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

Cooperative Portfolio

Assembly of Nerds, enhanced

- The experts periodically gather for brief standup meetings
- Via some protocol, the experts exchange the most valuable insights gained since the last meeting
- Solving continues — each expert may use the shared insights at their own discretion

Equivalent to “insights” in SAT solving:

Cooperative Portfolio

Assembly of Nerds, enhanced

- The experts periodically gather for brief standup meetings
- Via some protocol, the experts exchange the most valuable insights gained since the last meeting
- Solving continues — each expert may use the shared insights at their own discretion

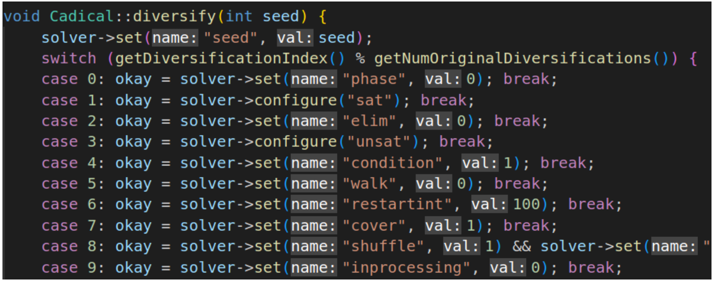
Equivalent to “insights” in SAT solving: **learnt (conflict) clauses**

- Explored branch of search space — safe to prune
- Potential step for deriving unsatisfiability

Clause Sharing Portfolios: Design Space

Portfolio considerations

- Which sequential solvers to employ?
- How to diversify solvers?
  — different search algorithms, selection heuristics, restart intervals, . . .
  — different random seeds, initial phases, input permutations, . . .



```
void Cadical::diversify(int seed) {
    solver->set(name: "seed", val: seed);
    switch (getDiversificationIndex() % getNumOriginalDiversifications()) {
    case 0: okay = solver->set(name: "phase", val: 0); break;
    case 1: okay = solver->configure("sat"); break;
    case 2: okay = solver->set(name: "elim", val: 0); break;
    case 3: okay = solver->configure("unsat"); break;
    case 4: okay = solver->set(name: "condition", val: 1); break;
    case 5: okay = solver->set(name: "walk", val: 0); break;
    case 6: okay = solver->set(name: "restartint", val: 100); break;
    case 7: okay = solver->set(name: "cover", val: 1); break;
    case 8: okay = solver->set(name: "shuffle", val: 1) && solver->set(name: "
    case 9: okay = solver->set(name: "inprocessing", val: 0); break;
```

Clause Sharing Portfolios: Design Space

Portfolio considerations

- Which sequential solvers to employ?
- How to diversify solvers?
  — different search algorithms, selection heuristics, restart intervals, . . .
  — different random seeds, initial phases, input permutations, . . .

Clause exchange considerations

- How often to share? (immediate/eager? delayed/lazy? periodic?)
- How many clauses to share? (fixed volume? fixed quality criteria?)
- Which clauses to share? (shortest? lowest LBD?)
- How to implement sharing? (all-to-all? leader-worker? some communication graph?)

Early Clause Sharing Portfolios

ManySAT (Hamadi, Jabbour, and Sais 2009)

- Hand-crafted diversification of four solver configurations
  — Restart policy, variable + polarity selection heuristic, . . .
- Eager exchange of clauses of length ≤ 8 via lockless queues

Early Clause Sharing Portfolios

ManySAT (Hamadi, Jabbour, and Sais 2009)

- Hand-crafted diversification of four solver configurations
  — Restart policy, variable + polarity selection heuristic, . . .
- Eager exchange of clauses of length ≤ 8 via lockless queues

Plingeling (Biere 2010)

- Portfolio over Lingeling configurations (shared-memory parallelism)
- Lazy exchange of information over “boss thread”
  — 2010: Unit clauses only
  — 2011: Unit clauses + equivalences
  — Since 2013: Unit clauses + equivalences + clauses of length ≤ 40, LBD ≤ 8
- Best parallel solver for many years

Massively parallel hardware?

Distributed computing

In distributed computing, several machines
(with no shared main memory) run together.
On each machine we run a number of processes,
each of which runs on a number of cores.
Processes commonly communicate by exchanging messages.

SuperMUC-NG: 6 336 nodes × 48 cores

Massively parallel hardware?

Distributed computing

In distributed computing, several machines (with no shared main memory) run together.
On each machine we run a number of processes, each of which runs on a number of cores.
Processes commonly communicate by exchanging messages.

SuperMUC-NG: 6 336 nodes × 48 cores

Large distributed systems (hundreds to thousands of cores) impose new requirements, challenges:

- No shared memory — communication protocols required
- Diminishing returns due to exhausted diversification of solvers

Massively parallel hardware?

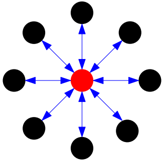
Distributed computing

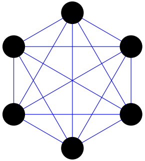
In distributed computing, several machines (with no shared main memory) run together.
On each machine we run a number of processes, each of which runs on a number of cores.
Processes commonly communicate by exchanging messages.

SuperMUC-NG: 6 336 nodes × 48 cores

Large distributed systems (hundreds to thousands of cores) impose new requirements, challenges:

- No shared memory — communication protocols required
- Diminishing returns due to exhausted diversification of solvers
- Some exchange schemes are conceptually not scalable
  - "Star graph": Master process collects, serves all exported clauses
  - Naïve (quadratic) all-to-all exchange of clauses





Massively parallel SAT portfolio

HordeSat (Balyo, Sanders, Sinz 2015)

- Decentralization: No single leader node / process
- Two-level (“hybrid”) parallelization
  — One or several processes on each machine
  — Multiple solver threads (+ communication thread) on each process

Massively parallel SAT portfolio

HordeSat (Balyo, Sanders, Sinz 2015)

- Decentralization: No single leader node / process
- Two-level (“hybrid”) parallelization
  — One or several processes on each machine
  — Multiple solver threads (+ communication thread) on each process
- Diversification options:
  — Native diversification (set of hand-crafted solver configurations)
  — Modifying some initial variable phases
  — Random seeds
- Periodic all-to-all clause exchange

HordeSat: Results

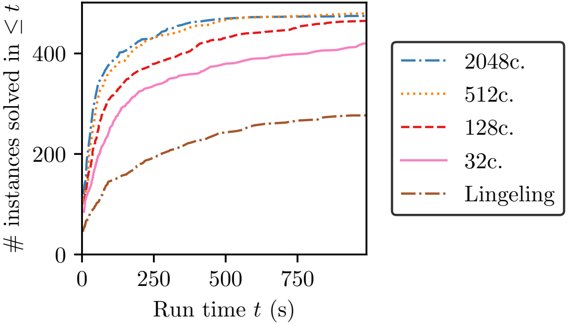
- Super-linear speedups for individual instances
  = speedup $>$ c on c cores!

HordeSat: Results

- Super-linear speedups for individual instances
  = speedup $> c$ on c cores!
  — SAT: “NP luck” – some solver got lucky
  — UNSAT: distributed memory accommodates more clauses than any sequential solver

HordeSat: Results

- Super-linear speedups for individual instances
  = speedup $> c$ on c cores!
  — SAT: “NP luck” – some solver got lucky
  — UNSAT: distributed memory accommodates more clauses than any sequential solver
- Median speedup: 3 at 16 cores, 11.5 at 512 cores
  — Efficiency: $11.5/512 \approx$ 2.2%
  — Deploying HordeSat is often not worth it
- No improvement beyond $\approx$ 500 cores



Data extracted from HordeSat paper

From HordeSat to Mallob

Research Question

How can we improve performance, (resource-)efficiency, and average response times of SAT solving in modern distributed environments?

From HordeSat to Mallob

Research Question

How can we improve performance, (resource-)efficiency, and average response times of SAT solving in modern distributed environments?

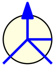
Result: Mallob

Mallob is a platform for SAT solving (*and other NP-hard problems*) with:

- multi-user, on-demand, malleable scheduling and solving of many problems at once
- the HordeSat paradigm re-engineered and made efficient
- state-of-the-art SAT performance from dozens to thousands of cores

Engineering a Scalable SAT Solver

Succinct clause sharing

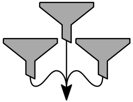


Hierarchical merging + duplicate detection

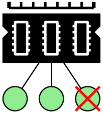
Global and adaptive admission criteria

Distributed clause filtering

Exact filtering of clauses shared before / from self



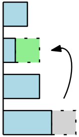
Memory Awareness



Reduction of solver threads

Negotiated memory panic

Adaptive buffering



Keep best clauses at expense of worse clauses

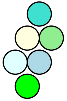
For export + import

Diversification

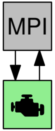
Glucose, Lingeling, CaDiCaL, Kissat

Clause shuffling

Noisy parameters



Controlling



Subprocess for solvers

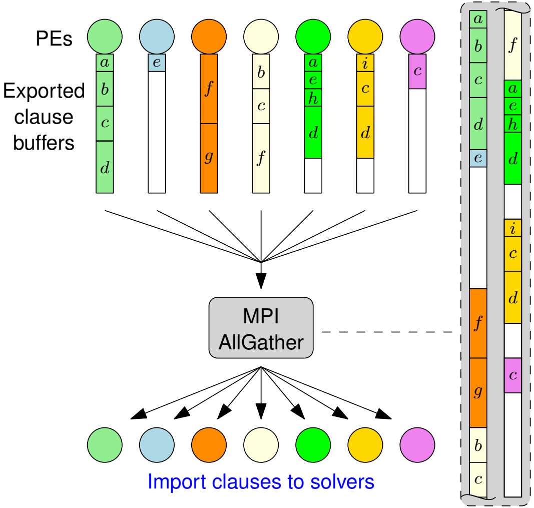
Seamless preemption and termination

Fault tolerance

Clause Exchange in HordeSat

Periodic collective operation **AllGather**

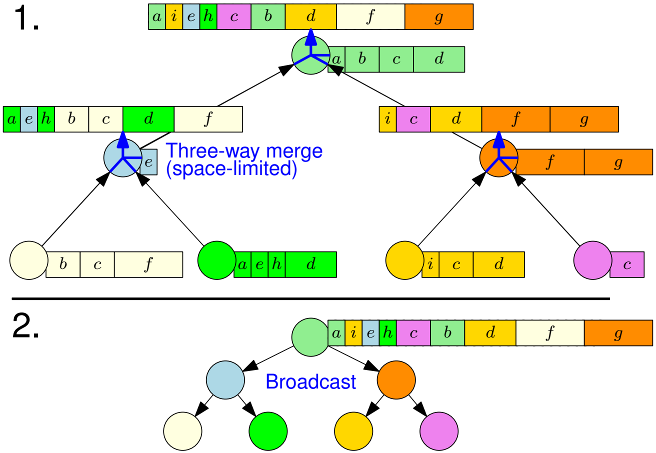
- Locally best clauses are shared with everyone
- Duplicate clauses
- “Holes” in buffer carrying no information
- Buffer grows proportionally with # proc.
  ⇒ Bottleneck w.r.t communication *and* local work



Clause Exchange in Mallob

Custom collective operation [SAT'21]

- Aggregate information along binary tree of processors
- Detect duplicates during merge
- Result is of compact shape
- Sublinear buffer size growth: Discard longest clauses as necessary



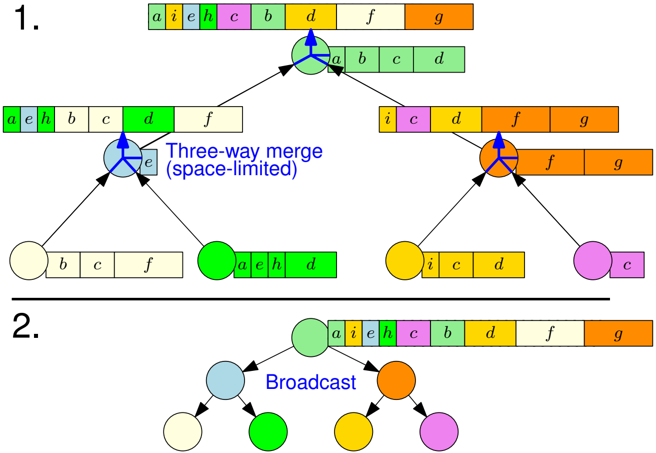
Clause Exchange in Mallob

Custom collective operation [SAT'21]

- Aggregate information along binary tree of processors
- Detect duplicates during merge
- Result is of compact shape
- Sublinear buffer size growth: Discard longest clauses as necessary

Observations

- Clause needs to meet global quality threshold to be shared successfully
- Quality threshold adapts to state of solving



1.
a i e h c b d f g
a b c d
a e h b c d f
i c d f g
e
Three-way merge (space-limited)
f g
b c f
a e h d
i c d
c
2.
a i e h c b d f g
Broadcast

Clause Filtering

The Problem

Given a shared clause c and a solver S, decide if S has received or produced c before (recently).

Previously: [HordeSat] [SAT'21]

- Bloom filters: fixed size, risk of false positives

Clause Filtering

The Problem

Given a shared clause c and a solver S, decide if S has received or produced c before (recently).

Previously: [HordeSat] [SAT'21]

- Bloom filters: fixed size, risk of false positives

Mallob'22+: Exact distributed filter [ISC'22]

- Process p remembers clauses it exported itself and tags their producing solver(s)

Clause Filtering

The Problem

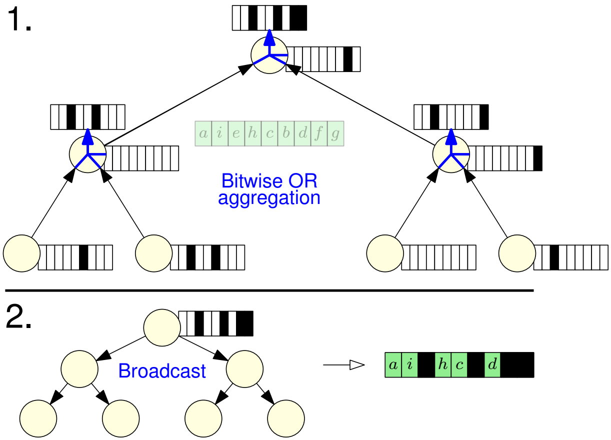
Given a shared clause c and a solver S, decide if S has received or produced c before (recently).

Previously: [HordeSat] [SAT'21]

- Bloom filters: fixed size, risk of false positives

Mallob'22+: Exact distributed filter [ISC'22]

- Process p remembers clauses it exported itself and tags their producing solver(s)
- Aggregate bit vector v where $v[i] := \bigvee_p (p \text{ remembers } c_i)$
- Only import clauses c_i for which $v[i] = \texttt{false}$



Clause Filtering

The Problem

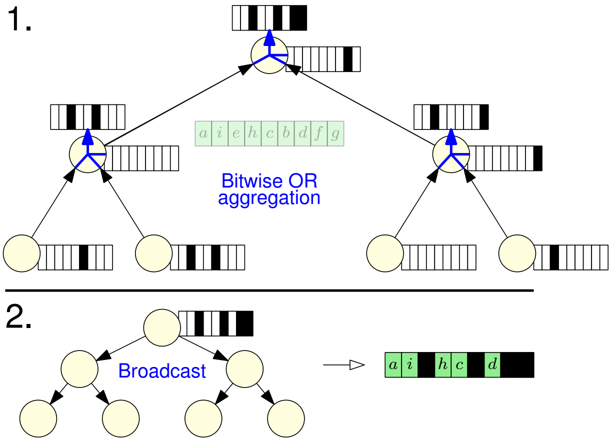
Given a shared clause c and a solver S, decide if S has received or produced c before (recently).

Previously: [HordeSat] [SAT'21]

- Bloom filters: fixed size, risk of false positives

Mallob'22+: Exact distributed filter [ISC'22]

- Process p remembers clauses it exported itself and tags their producing solver(s)
- Aggregate bit vector v where $v[i] := \bigvee_p (p \text{ remembers } c_i)$
- Only import clauses c_i for which $v[i] = \texttt{false}$
- Compensate for filtered clauses next sharing!



LBD Values

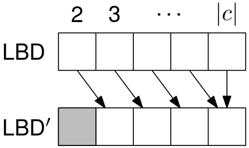
- Clause quality metric, central for whether to keep a clause
- Some solvers keep clauses with LBD 2 indefinitely
  — but expect a single solver's clause volume!

LBD Values

- Clause quality metric, central for whether to keep a clause
- Some solvers keep clauses with LBD 2 indefinitely
  — but expect a single solver's clause volume!
- Use original LBD values of imported clauses? [HordeSat]
  ⇒ Growing overhead (time, space) from low-LBD clauses
- Reset LBD values to maximum at import? [TopoSAT2]
  ⇒ Many clauses may be discarded very quickly

LBD Values

- Clause quality metric, central for whether to keep a clause
- Some solvers keep clauses with LBD 2 indefinitely
  — but expect a single solver's clause volume!
- Use original LBD values of imported clauses? [HordeSat]
  ⇒ Growing overhead (time, space) from low-LBD clauses
- Reset LBD values to maximum at import? [TopoSAT2]
  ⇒ Many clauses may be discarded very quickly



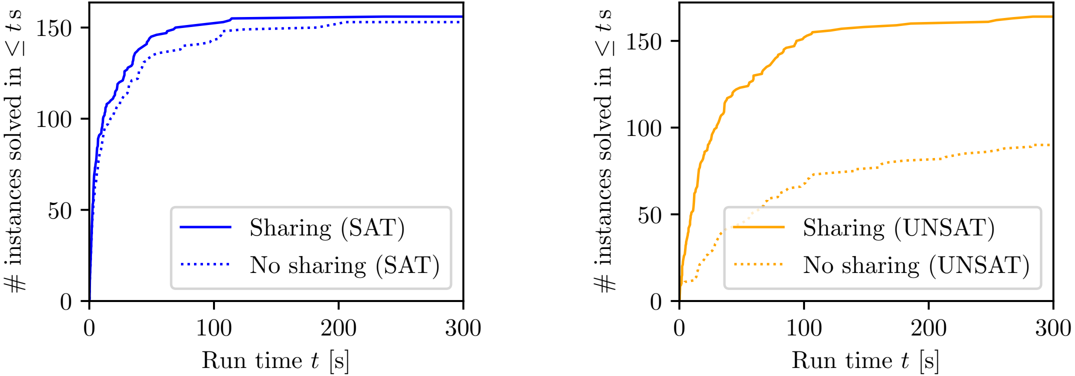
Our current approach: Increment each LBD before import

- Maintains LBD-based prioritization of clauses
- Solver keeps full control over its LBD-2-clauses
- “Regional clauses are the best!”

	Median RAM	PAR-2
Orig. LBD	108.8 GiB	75.7
Reset LBD	95.6 GiB	74.3
LBD++	97.3 GiB	72.9

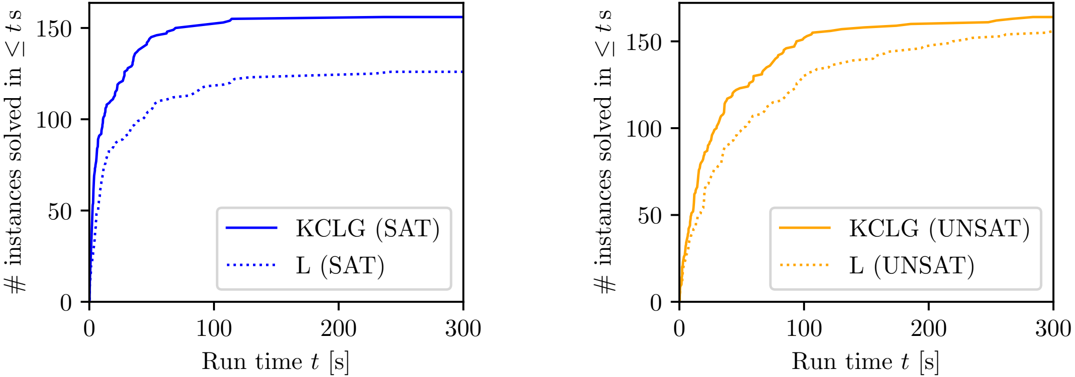
768 cores × 349 instances × 300 s

Merit of Clause Sharing, SAT vs. UNSAT



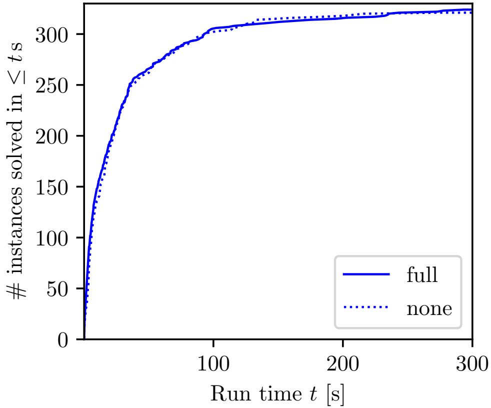
768 cores × 349 “solvable” instances from ISC 2022 × 300 s, portfolio “KCLG”

Merit of Diverse Portfolio, SAT vs. UNSAT



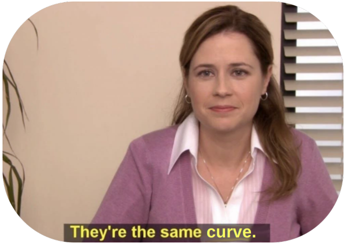
768 cores × 349 “solvable” instances from ISC 2022 × 300 s, with clause sharing

Merit of Diversification . . . None??

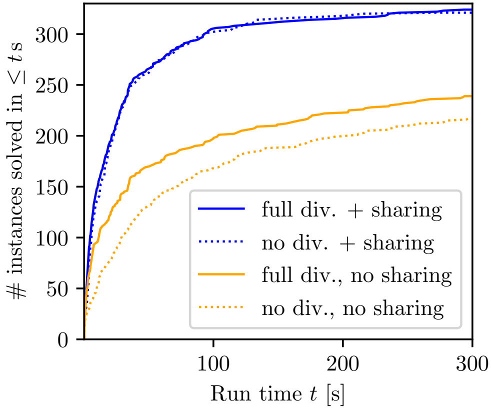


768 cores × 349 “solvable” instances from ISC 2022 × 300 s, portfolio “KCL”, with clause sharing!

- “full”: 36 solver configs + random seeds
  + noisy parameters + input permutation
  + a few solvers not importing clauses
- “none”: 36 solver configs, nothing else



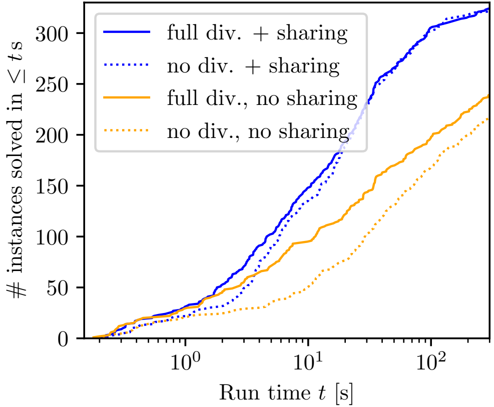
Merit of Diversification . . . None??



768 cores × 349 “solvable” instances from ISC 2022
× 300 s, portfolio “KCL”

- “full”: 36 solver configs + random seeds
  + noisy parameters + input permutation
  + a few solvers not importing clauses
- “none”: 36 solver configs, nothing else
- Without clause sharing diversification helps a lot!
- Clause sharing appears to absorb common diversification techniques! How?

Merit of Diversification … None??



768 cores × 349 “solvable” instances from ISC 2022 × 300 s, portfolio “KCL”

- “full”: 36 solver configs + random seeds + noisy parameters + input permutation + a few solvers not importing clauses
- “none”: 36 solver configs, nothing else
- Without clause sharing diversification helps a lot!
- Clause sharing appears to absorb common diversification techniques! How?
- Hypothesis:
  1. Shared clauses arrive at solvers at different times
  2. Solvers vary in when (and what) they import
  3. “Butterfly effect”
  4. Clause sharing as search space pruning: solvers won’t re-explore pruned branches!

Scaling and Speedups

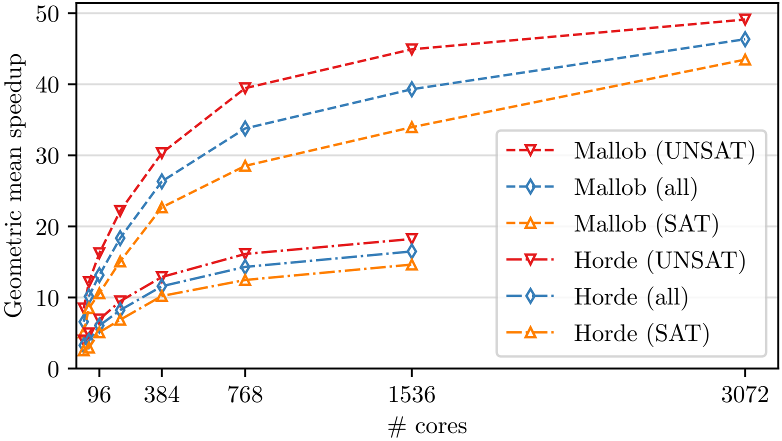
Updated HordeSat
(Lingeling)

vs.

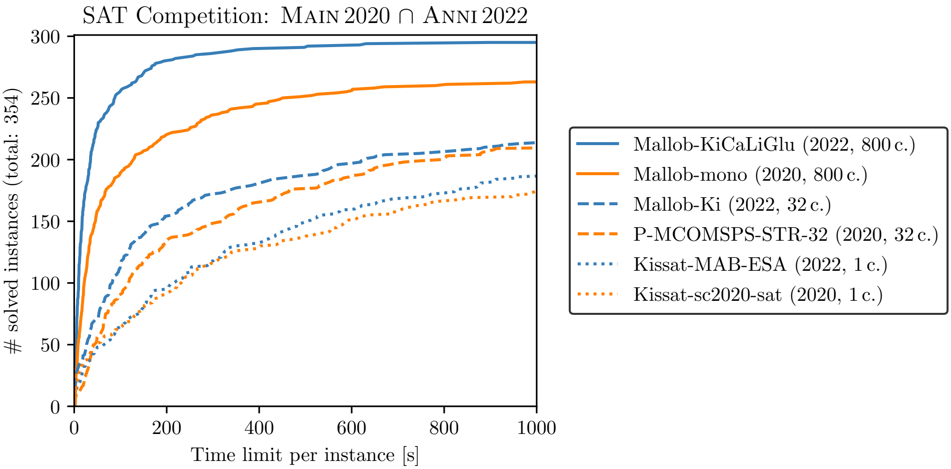
Mallob
(Kissat-CaDiCaL-Lingeling)

Sat Comp. 2021 benchmarks

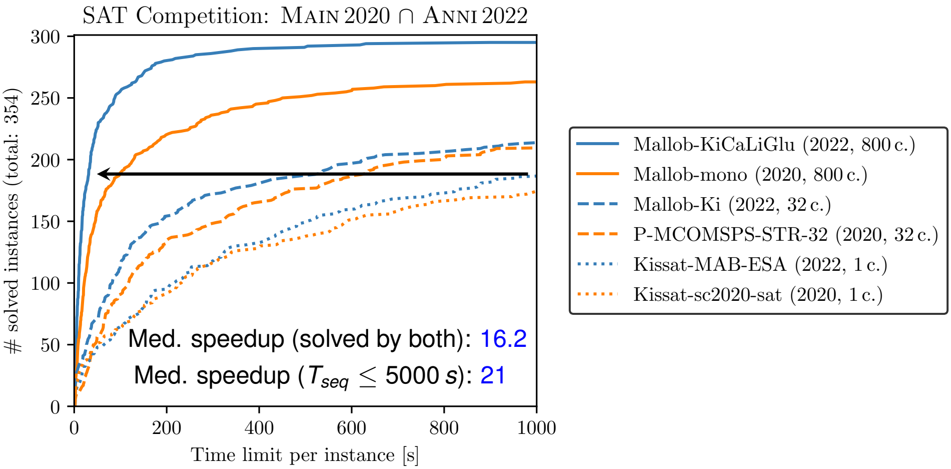
Sequential baseline:
`Kissat_MAB_HyWalk`
Seq. time limit: 115200 s
Par. time limit: 300 s



SAT Competition 2022



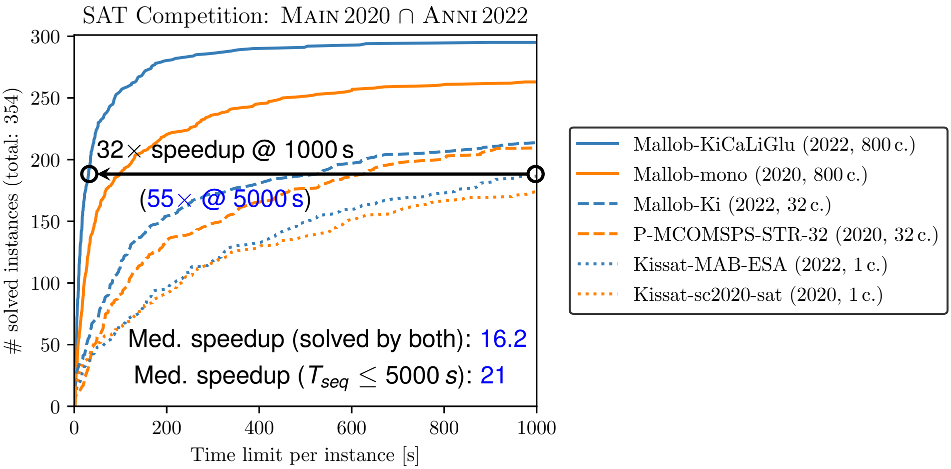
SAT Competition 2022



Med. speedup (solved by both): 16.2

Med. speedup ($T_{seq} \leq 5000\,s$): 21

SAT Competition 2022



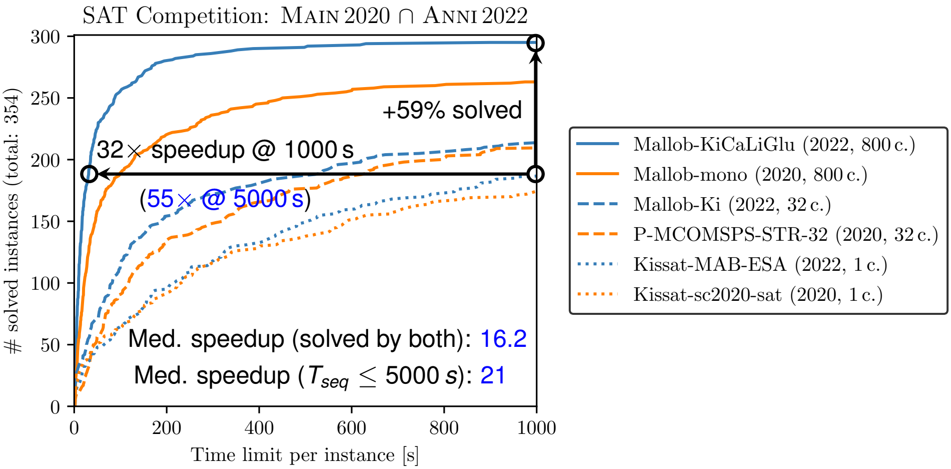
32× speedup @ 1000 s

(55× @ 5000 s)

Med. speedup (solved by both): 16.2

Med. speedup ($T_{seq} \leq 5000\,s$): 21

SAT Competition 2022



Better Efficiency?

Massive parallelism for a single formula

- Faster solving times
- Can resolve problems out of reach for sequential solvers
- Not that resource efficient (on average)

Better Efficiency?

Massive parallelism for a single formula

- Faster solving times
- Can resolve problems out of reach for sequential solvers
- Not that resource efficient (on average)

Solving many formulas in parallel

- Embarrassingly parallel
- Solving itself less powerful

Better Efficiency?

Massive parallelism for a single formula

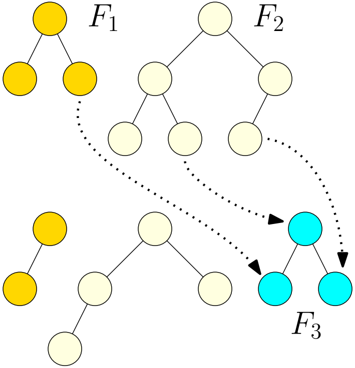
- Faster solving times
- Can resolve problems out of reach for sequential solvers
- Not that resource efficient (on average)

Solving many formulas in parallel

- Embarrassingly parallel
- Solving itself less powerful

Best of both worlds? [EuroPar'22]

- On demand scheduling of incoming (SAT) jobs
- Resize jobs during their execution as needed
- Few milliseconds to schedule an incoming job, full utilization whenever sufficient demand is present



Solving 400 Formulae on up to 6400 Cores

Problem statement

You allocate $x \in \{400, 1600, 6400\}$ cores for 2 h.
You have 400 formulae (SAT Comp. ’21) to solve. Go.

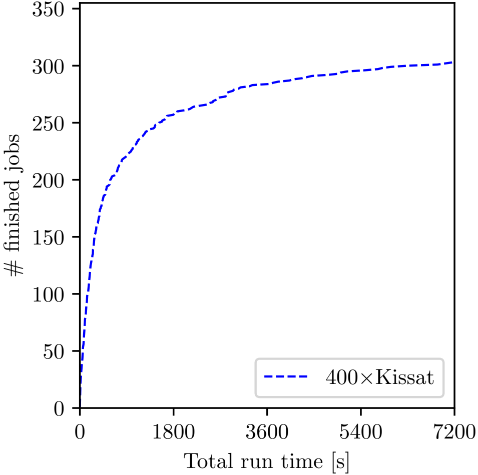
Solving 400 Formulae on up to 6400 Cores

Problem statement

You allocate $x \in \{400, 1600, 6400\}$ cores for 2 h.
You have 400 formulae (SAT Comp. ’21) to solve. Go.

Extreme 1: 400 Kissats in a trenchcoat

- No intra-job parallelism
- Embarrassingly parallel job processing (inter-job parallelism)
- Great resource efficiency



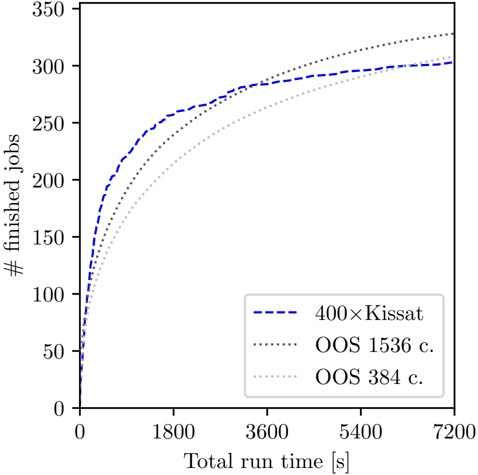
Solving 400 Formulae on up to 6400 Cores

Problem statement

You allocate $x \in \{400, 1600, 6400\}$ cores for 2 h.
You have 400 formulae (SAT Comp. ’21) to solve. Go.

Extreme 2: Massively parallel solving of each job

- One job at a time
- Assumption: Optimal Offline Schedule (OOS) — instances sorted by run time ascendingly
- No inter-job parallelism
- Maximum speedups from parallel SAT
- Poor resource efficiency



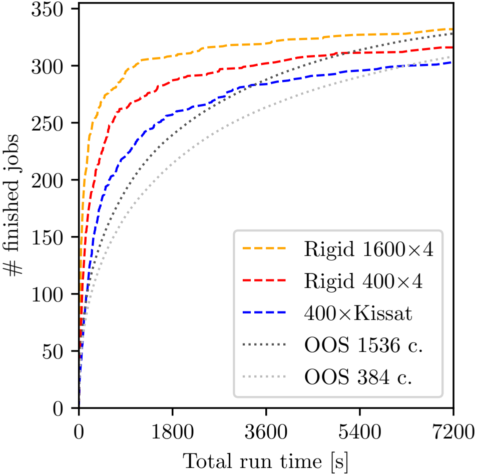
Solving 400 Formulae on up to 6400 Cores

Problem statement

You allocate $x \in \{400, 1600, 6400\}$ cores for 2 h.
You have 400 formulae (SAT Comp. ’21) to solve. Go.

Middle ground 1: Divide cores evenly among jobs

- Solid speedups at low-degree parallel SAT
- At the beginning, all cores are used
- After $<$ 15 min, $<$ 50% of cores are used



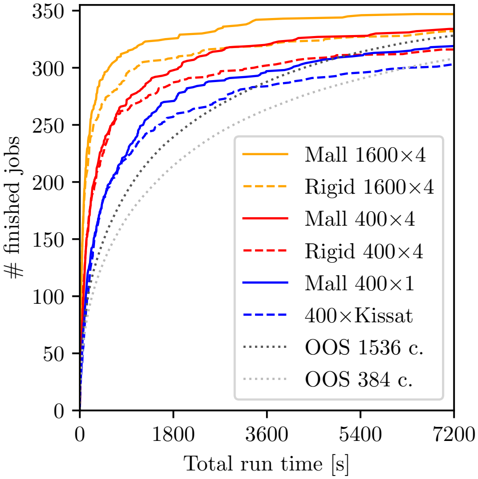
Solving 400 Formulae on up to 6400 Cores

Problem statement

You allocate $x \in \{400, 1600, 6400\}$ cores for 2 h.
You have 400 formulae (SAT Comp. '21) to solve. Go.

Middle ground 2: Divide cores dynamically among jobs

- Finishing jobs yield resources to remaining jobs — eventually exceeding 4× their initial resources
- Uses 100% of resources 100% of the time
- At 400 cores: Dominates 400× Kissat! — shows low overhead of scheduling



Mallob: Harvest

TACAS’23: UNSAT Proofs for Distributed Solvers

Issue

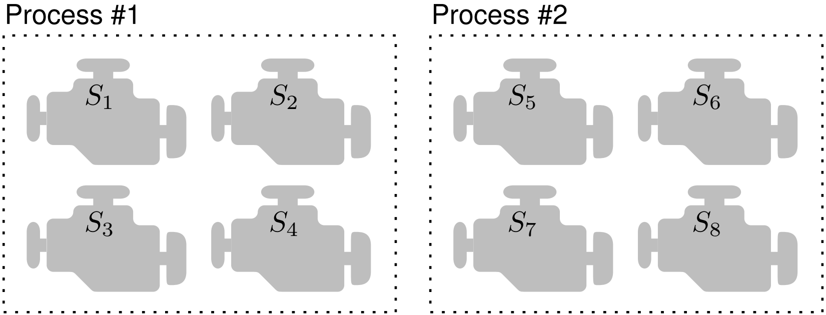
Parallel clause-sharing solvers do not support the production of unsatisfiability proofs.

- Real, practical issue
  - Some competition results of cloud solvers proved to be incorrect later!
  - Growing scale of computation ⇒ Growing probability of failures
- Prior approaches unsatisfactory
  - Limited to single machine
  - Not scalable at all

Objective

Introduce scalable production of unsatisfiability proofs for distributed clause-sharing SAT solvers, allowing to fully trust their results and exploit their power for critical applications.

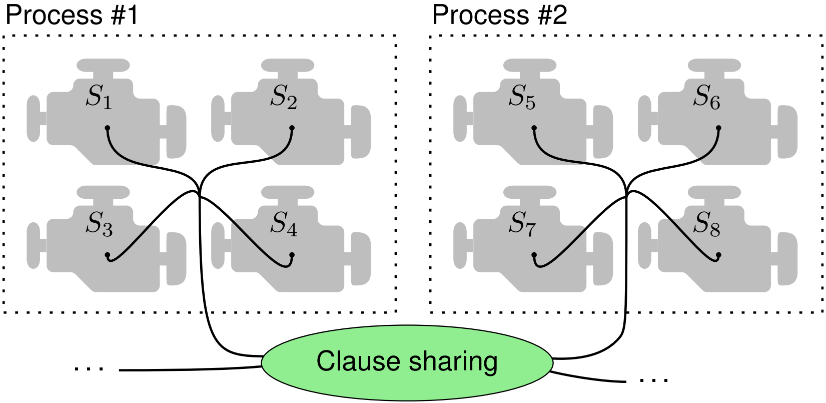
Background: Distributed Clause-Sharing SAT Solving



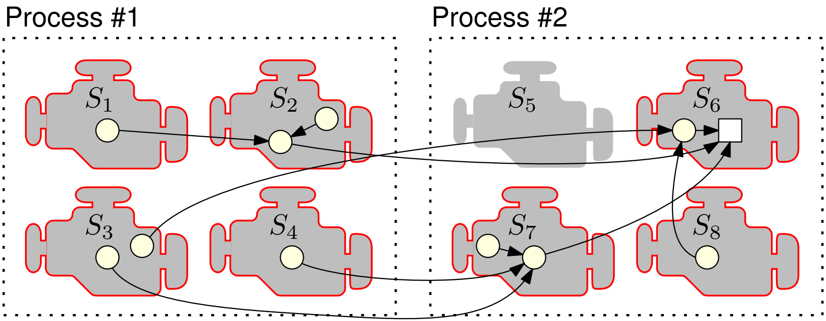
Portfolio of different CDCL solver configurations

$\approx$ producers of conflict clauses

Background: Distributed Clause-Sharing SAT Solving



Background: Distributed Clause-Sharing SAT Solving



Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1\, x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3\overline{x_4}$
add $x_1\, x_3$
add $\square$

+ compact format

+ prevalent in solvers

- costly checking

Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add □

- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} :=$ □ via c_{11}, c_{10}, c_1

- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Unique LRAT IDs across solvers?

Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

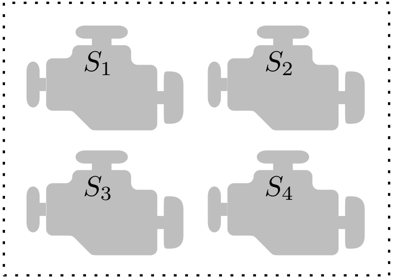
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Unique LRAT IDs across solvers?



10 original clauses

Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

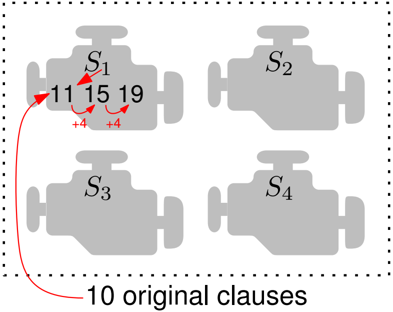
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Unique LRAT IDs across solvers?



Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

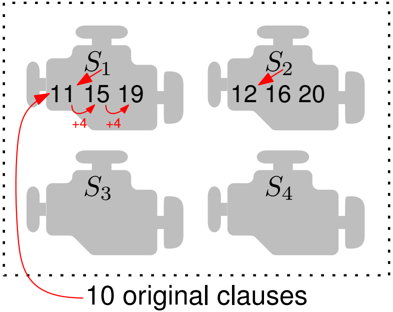
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Unique LRAT IDs across solvers?



Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

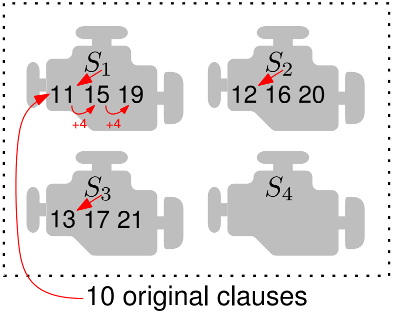
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Unique LRAT IDs across solvers?



Which Proof Format?

DRAT proof format

add $\overline{x_3}$
add $x_1 x_2$
add $\overline{x_1}$
delete $\overline{x_3}$
add $x_3 \overline{x_4}$
add $x_1 x_3$
add $\square$

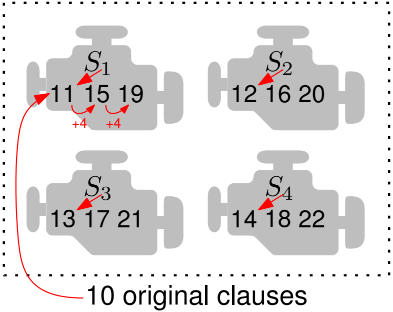
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

add $c_9 := \overline{x_3}$ via c_5, c_4
add $c_{10} := x_1 x_2$ via c_3, c_2
add $c_{11} := \overline{x_1}$ via c_6, c_9
delete c_9
add $c_{12} := x_3 \overline{x_4}$ via c_7, c_{11}
add $c_{13} := x_1 x_3$ via c_8, c_{12}
add $c_{14} := \square$ via c_{11}, c_{10}, c_1

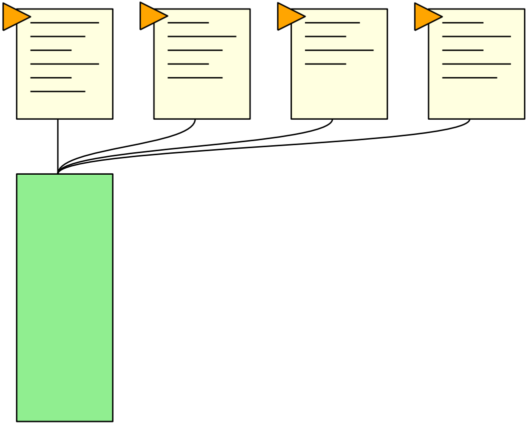
- + more efficient checking
- + unique IDs for clauses
- + explicit dependencies!

Unique LRAT IDs across solvers?



A Sequential Approach

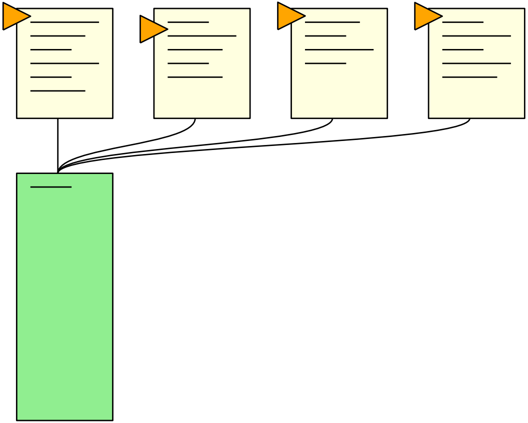
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output



A Sequential Approach

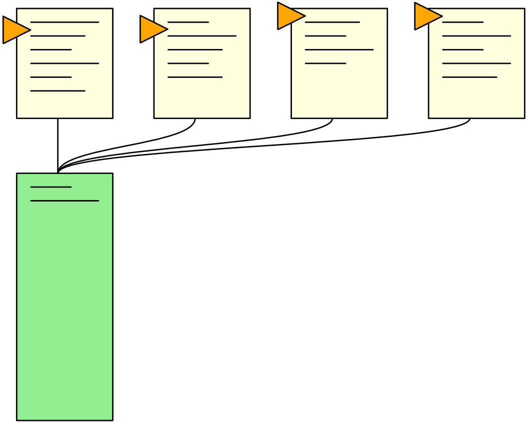
1. Combination

- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

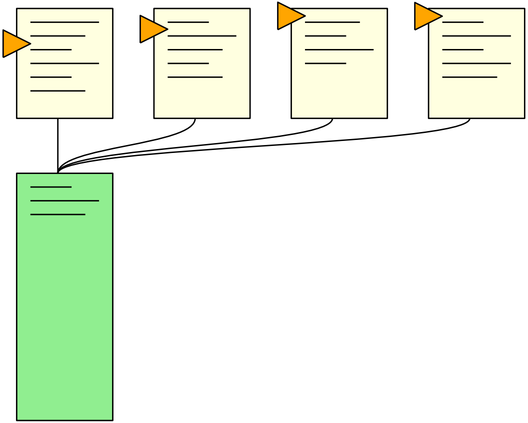
1. **Combination**
    - Read all partial proofs simultaneously
    - Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

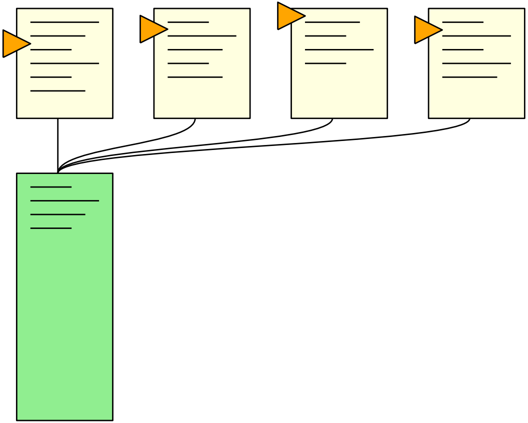
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

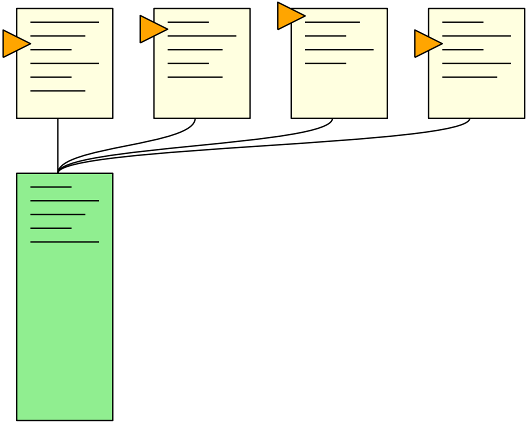
1. Combination

- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



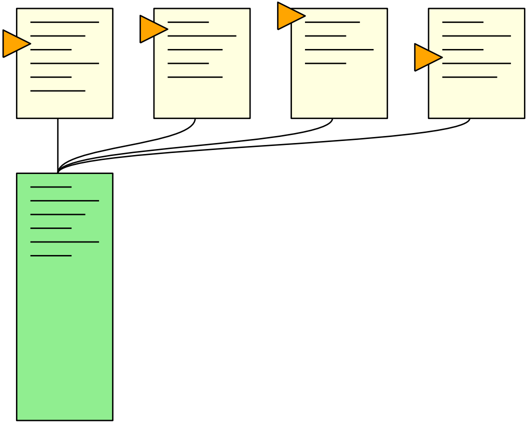
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



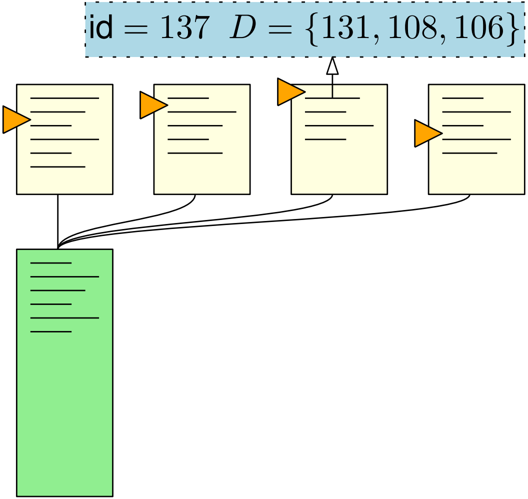
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output



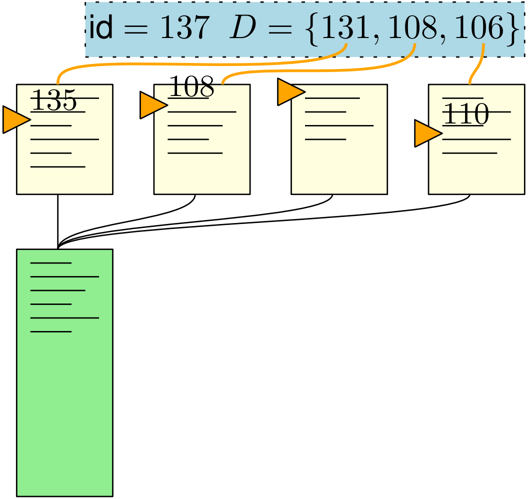
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



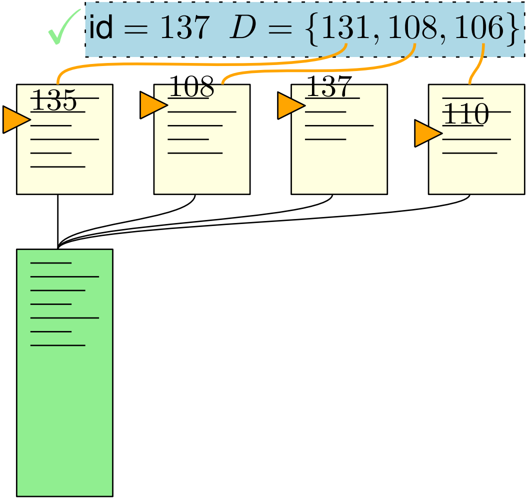
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



A Sequential Approach

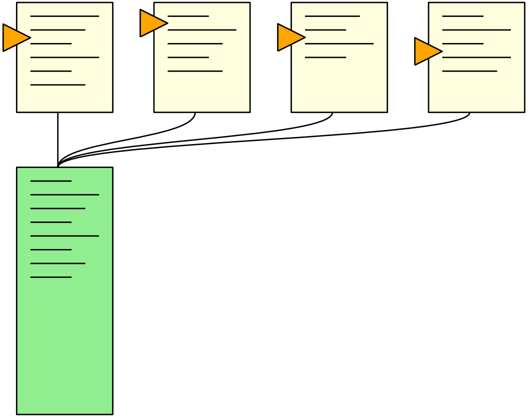
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



A Sequential Approach

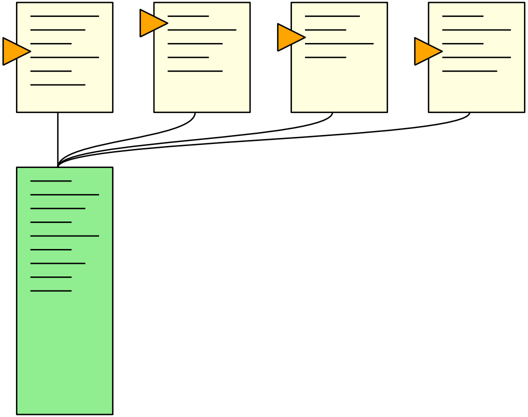
1. Combination

- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



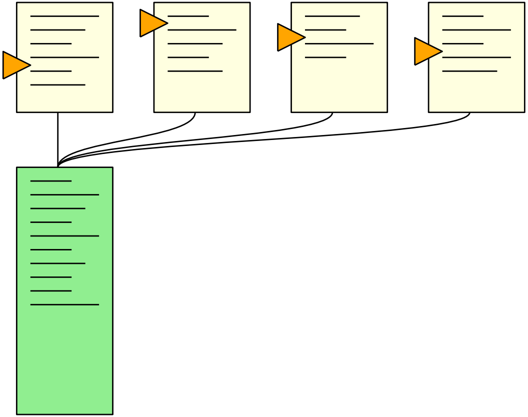
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



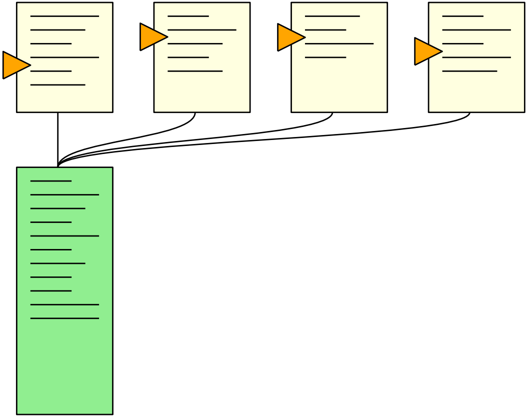
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



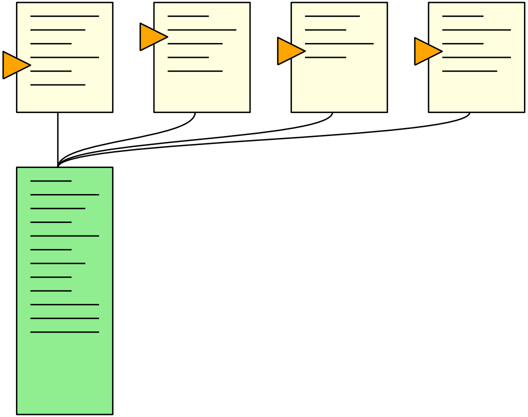
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



A Sequential Approach

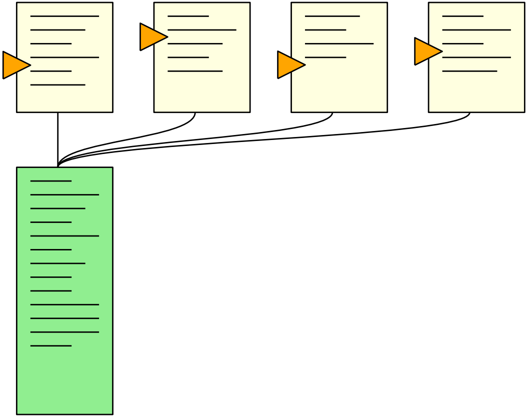
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output



A Sequential Approach

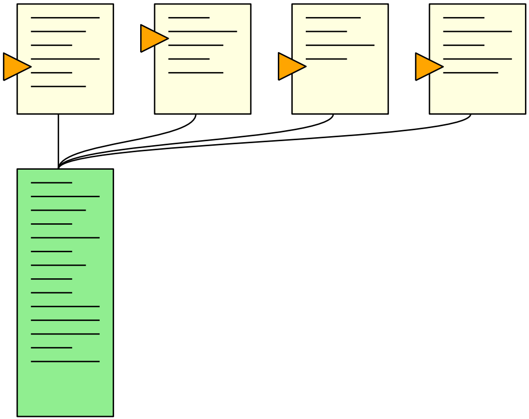
1. Combination

- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

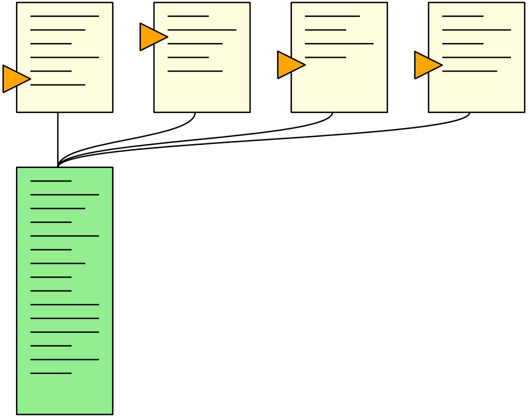
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output



A Sequential Approach

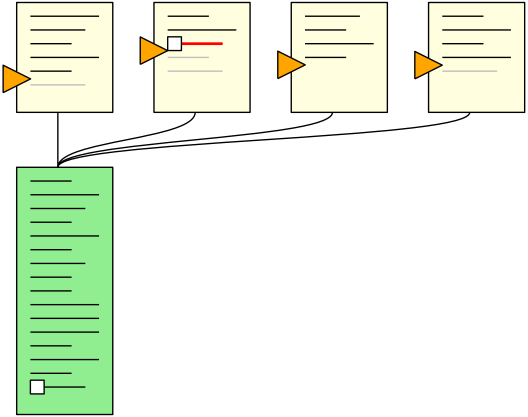
1. Combination

- Read all partial proofs simultaneously
- Output line ⇔ all dependencies d output



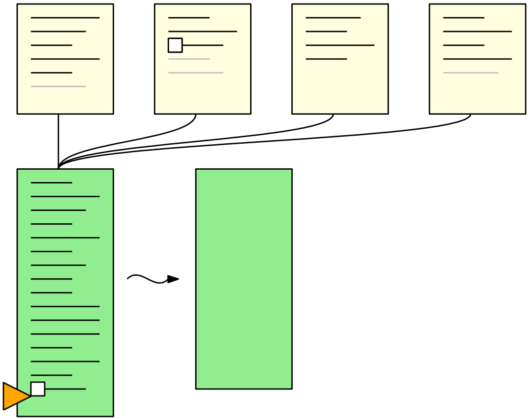
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



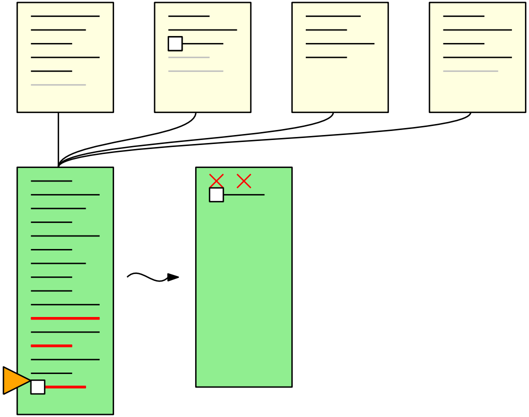
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front



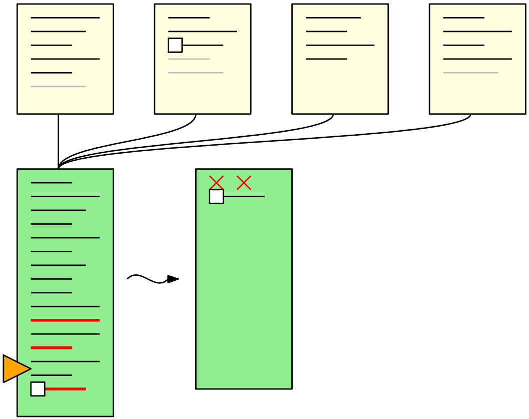
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\square)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R
     $\Rightarrow$ Output addition of c



A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R
     $\Rightarrow$ Output addition of c



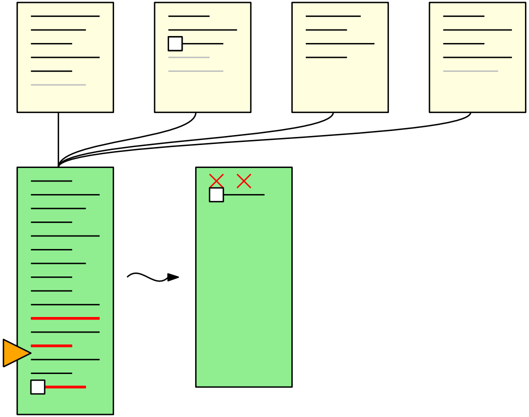
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

     $\Rightarrow$ For each dependency d of c, $d \notin R$:

     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c

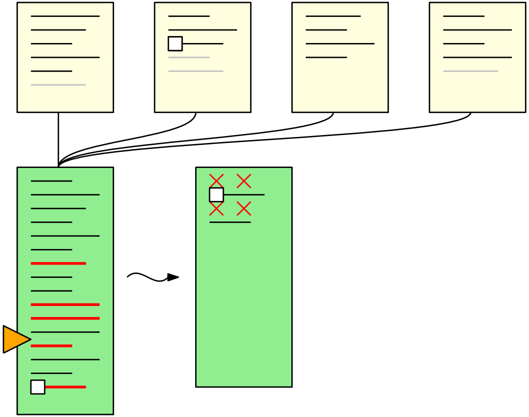


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause *c*: $id(c) \in R$?

     ⇒ For each dependency *d* of *c*, $d \notin R$:
     Output deletion of *d*, add *d* to *R*

     ⇒ Output addition of *c*

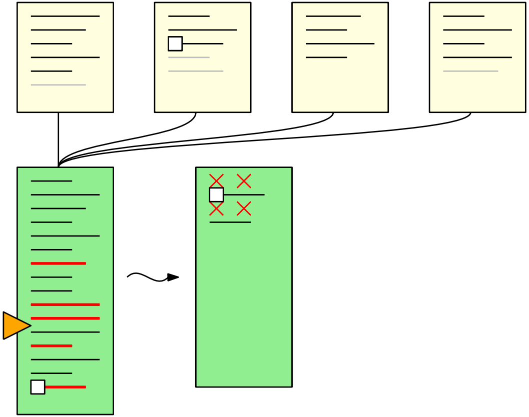


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

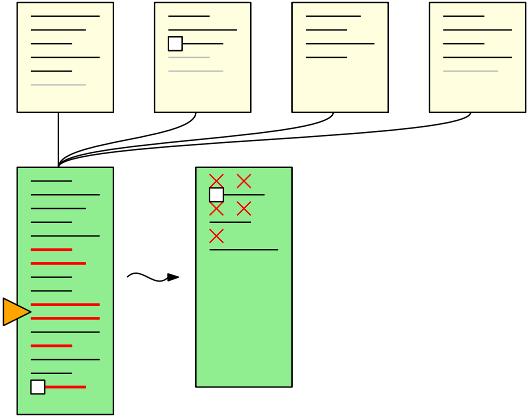
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c



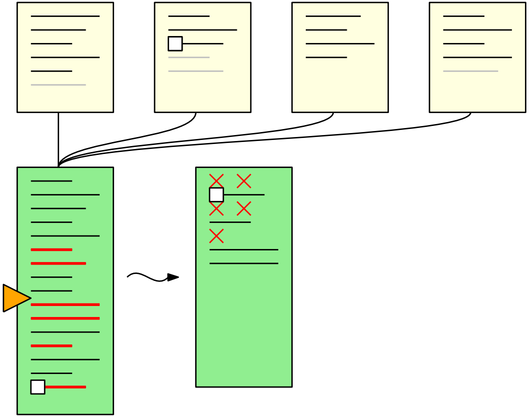
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\square)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R
     $\Rightarrow$ Output addition of c



A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R
     $\Rightarrow$ Output addition of c

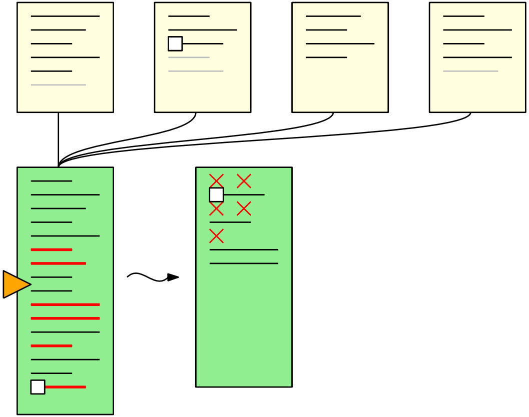


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c

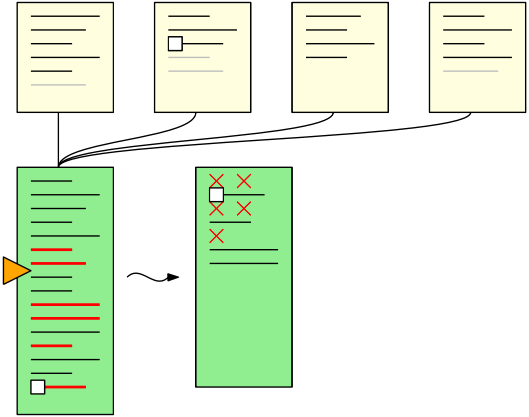


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

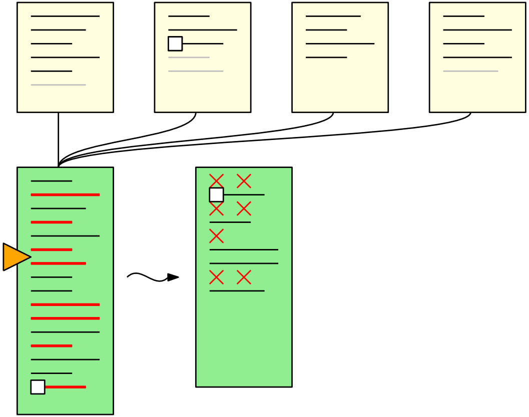
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c



A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R
     $\Rightarrow$ Output addition of c

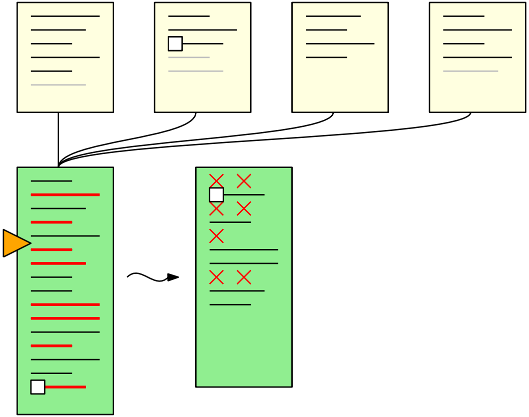


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

     ⇒ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     ⇒ Output addition of c

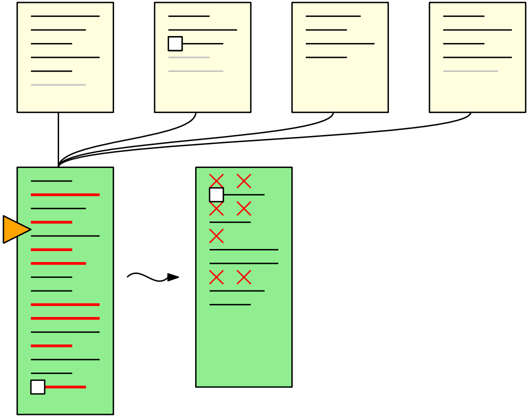


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\square)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c



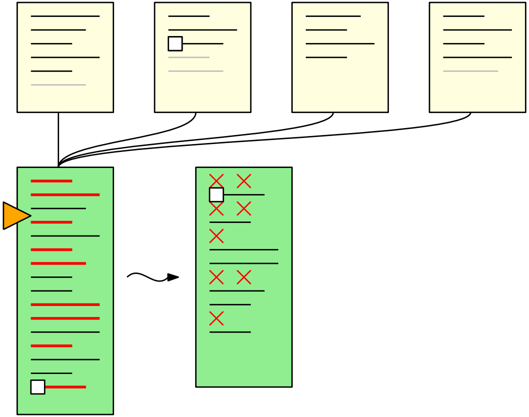
A Sequential Approach

1. Combination

- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output

2. Pruning

- Required clauses $R := \{id(\square)\}$
- Read combined proof from back to front
- @ Clause c: $id(c) \in R$?
  $\Rightarrow$ For each dependency d of c, $d \notin R$:
  Output deletion of d, add d to R
  $\Rightarrow$ Output addition of c

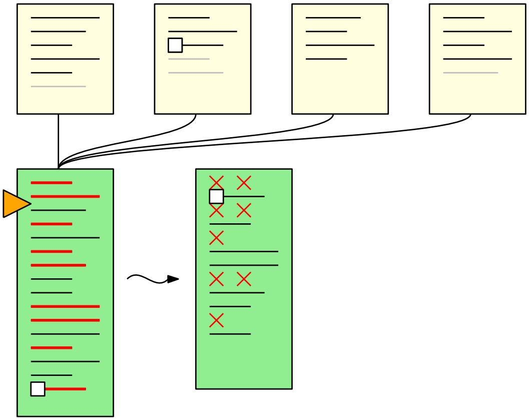


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause *c*: $id(c) \in R$?

     $\Rightarrow$ For each dependency *d* of *c*, $d \notin R$:
     Output deletion of *d*, add *d* to *R*

     $\Rightarrow$ Output addition of *c*

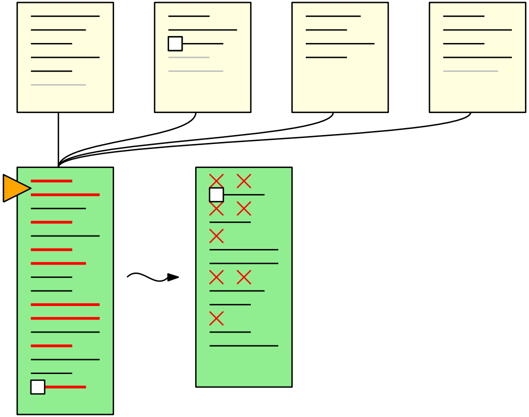


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\square)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c

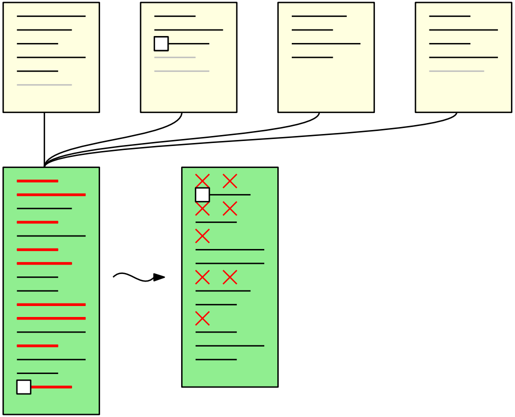


A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\square)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

     ⇒ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     ⇒ Output addition of c



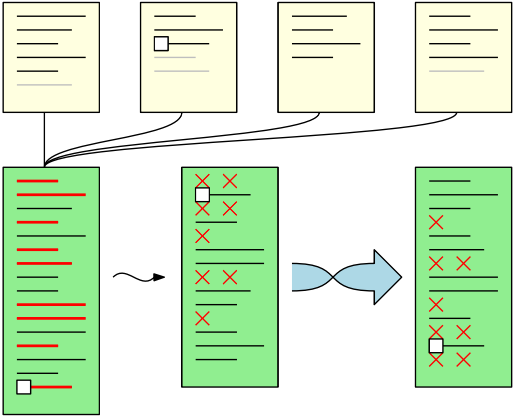
A Sequential Approach

1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output
2. **Pruning**
   - Required clauses $R := \{id(\square)\}$
   - Read combined proof from back to front
   - @ Clause c: $id(c) \in R$?

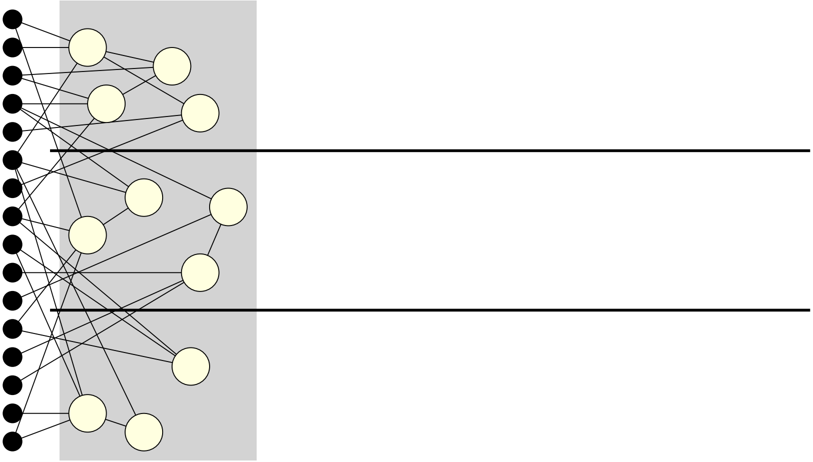
     $\Rightarrow$ For each dependency d of c, $d \notin R$:

     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c
3. **Reverse lines of pruned proof**

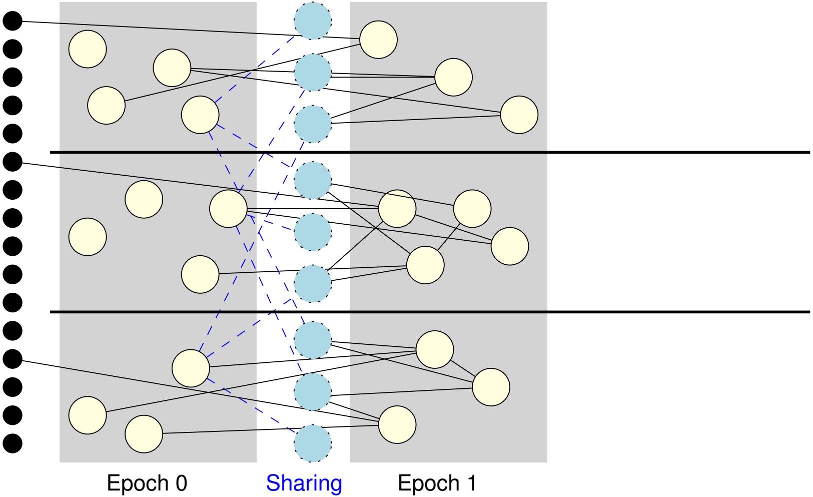


Distributed Pruning: Schematic Overview

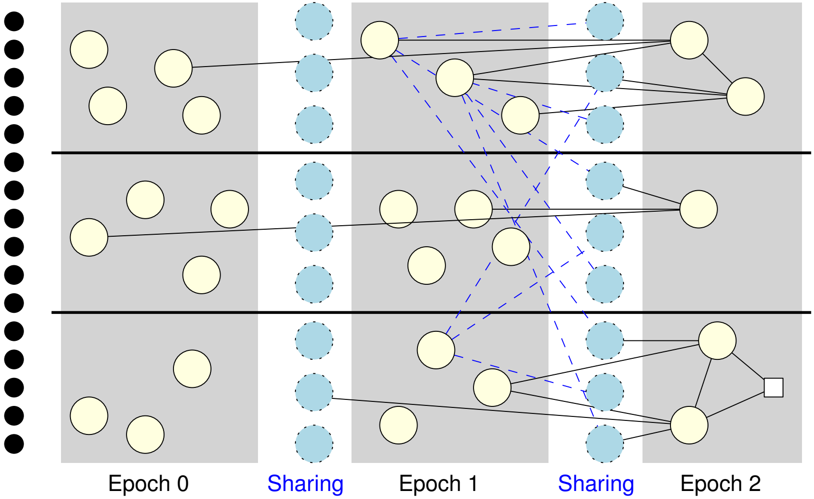


Epoch 0

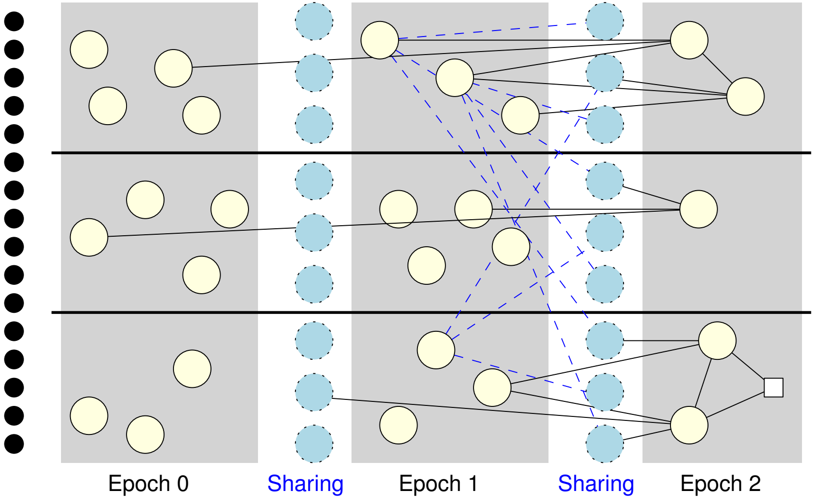
Distributed Pruning: Schematic Overview



Distributed Pruning: Schematic Overview

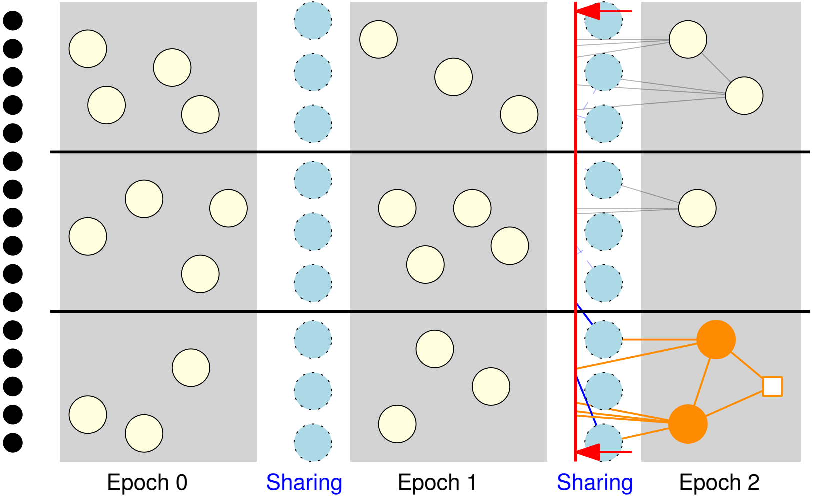


Distributed Pruning: Schematic Overview



First “prune”,
then combine!

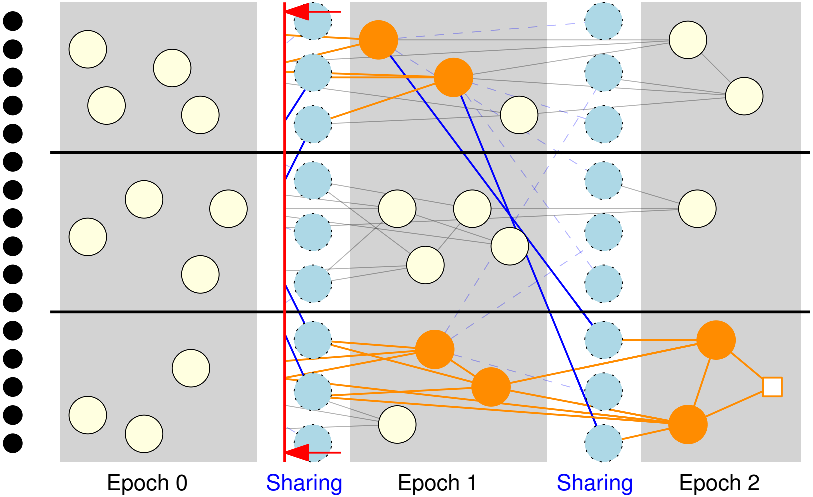
Distributed Pruning: Schematic Overview



First “prune”,
then combine!

Trace dependencies
epoch by epoch

Distributed Pruning: Schematic Overview

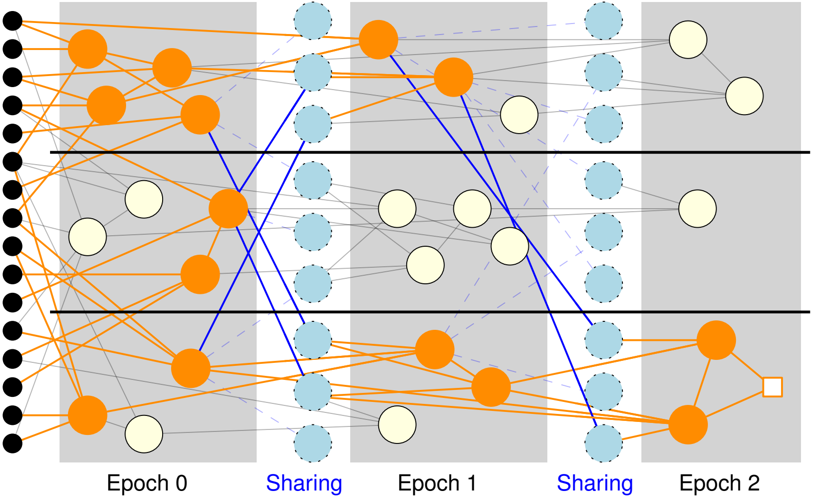


First “prune”,
then combine!

Trace dependencies
epoch by epoch

Redistribute remote IDs
at epoch borders

Distributed Pruning: Schematic Overview

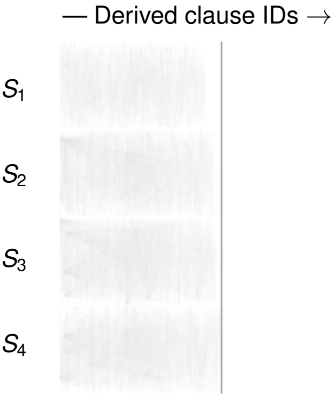


First “prune”,
then combine!

Trace dependencies
epoch by epoch

Redistribute remote IDs
at epoch borders

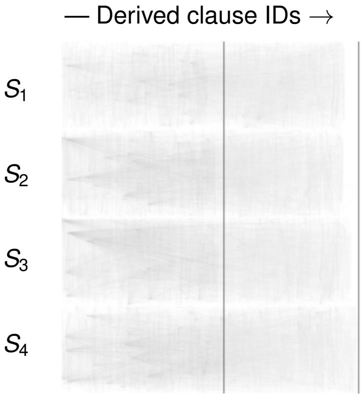
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

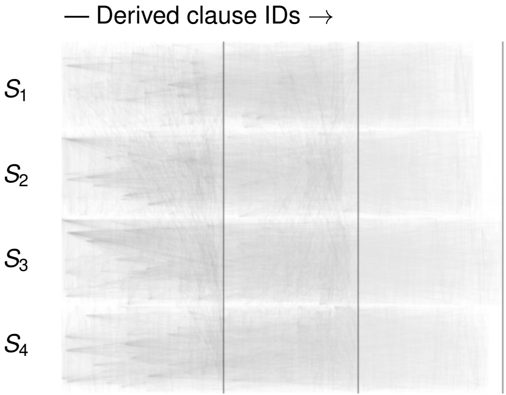
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

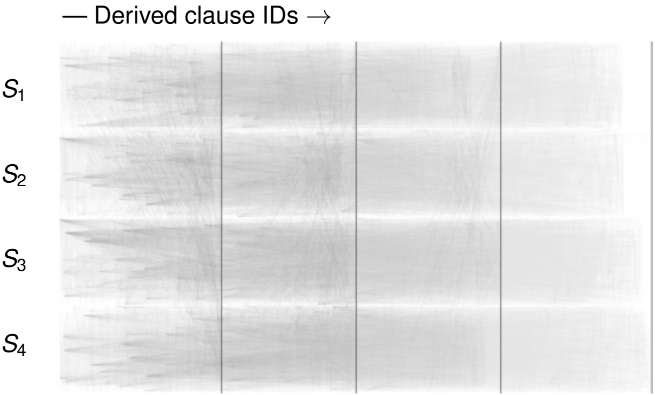
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

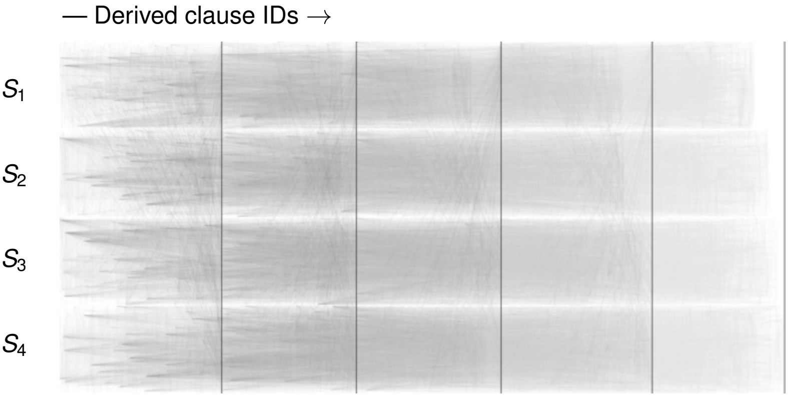
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

Distributed Pruning: Real Data

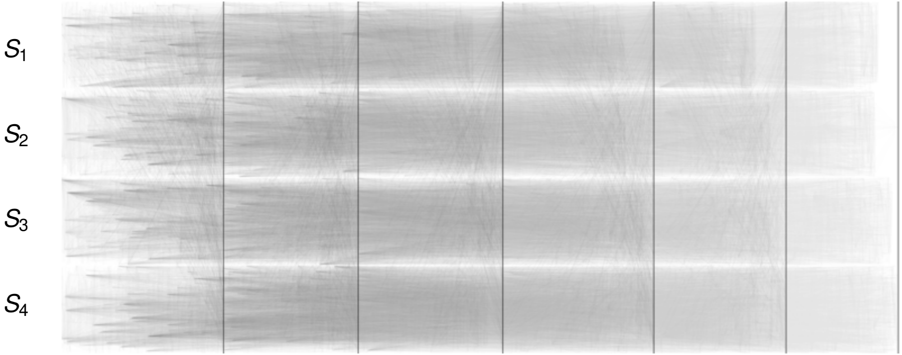


180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

Distributed Pruning: Real Data

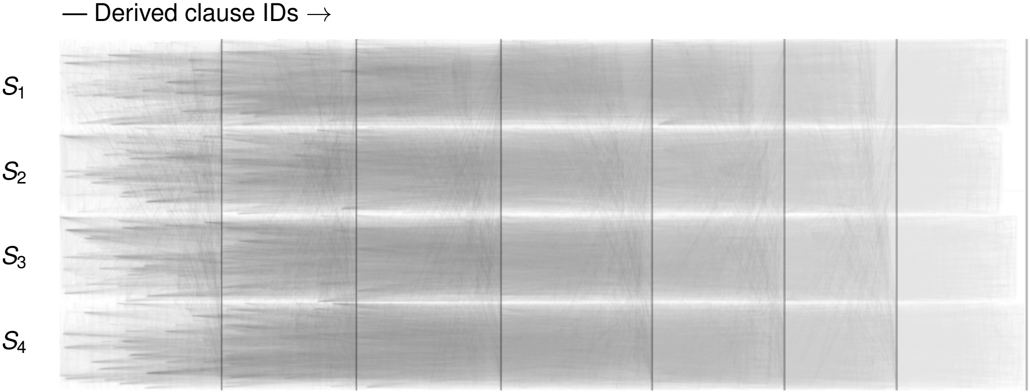
— Derived clause IDs →



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

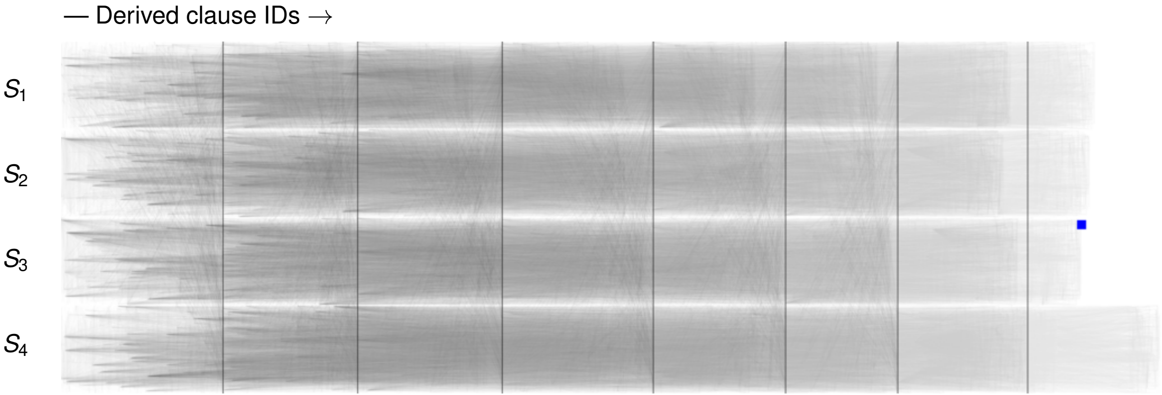
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

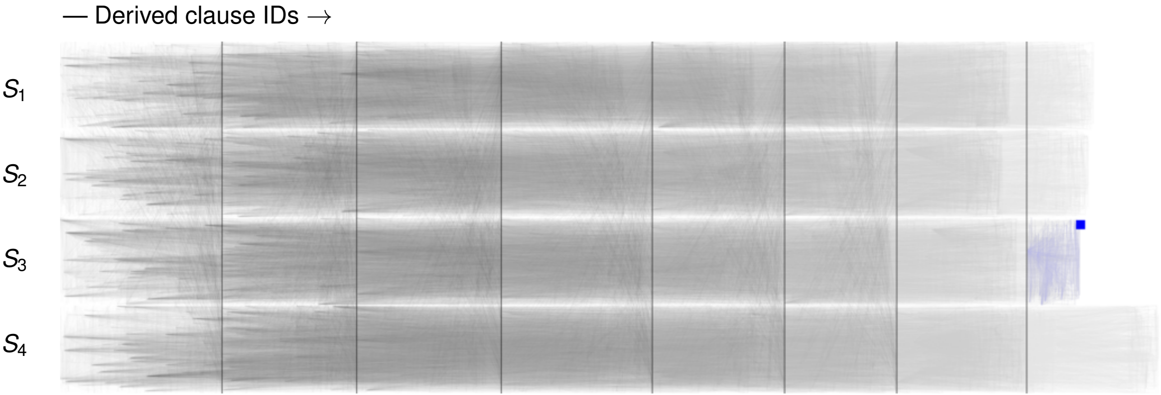
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

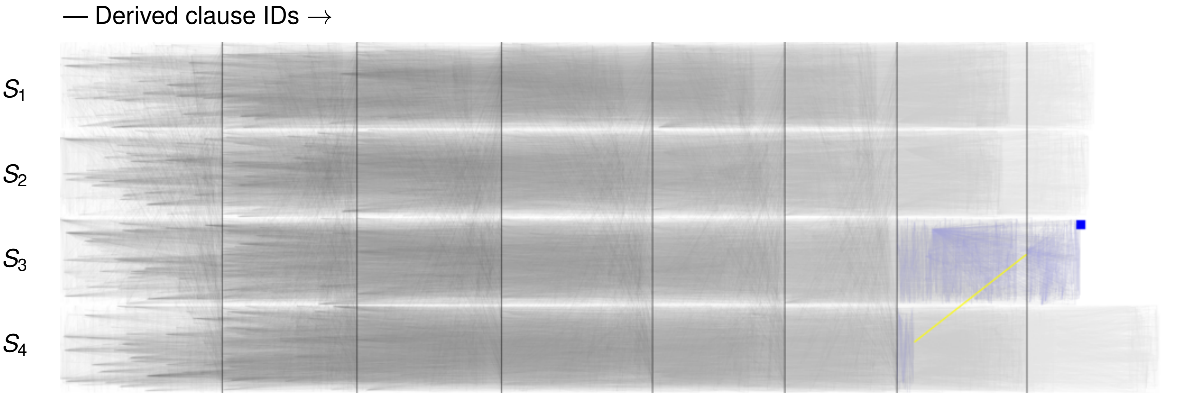
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

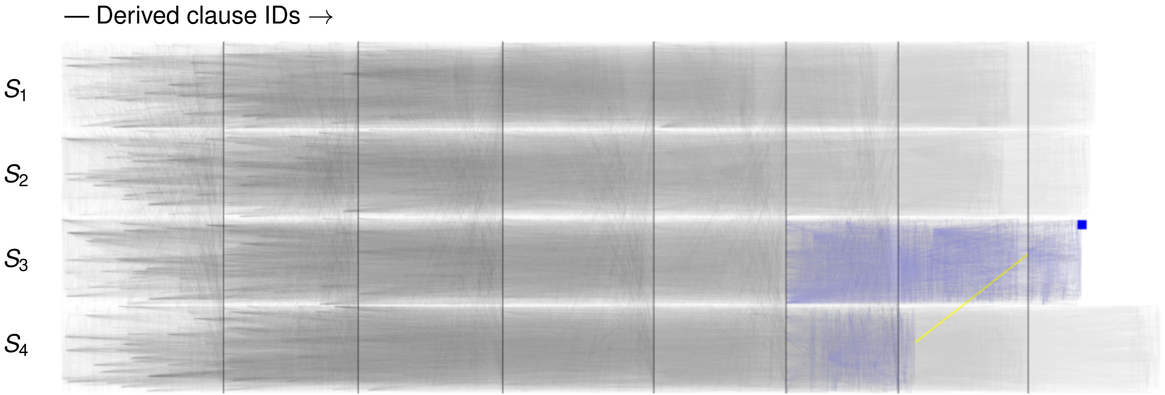
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

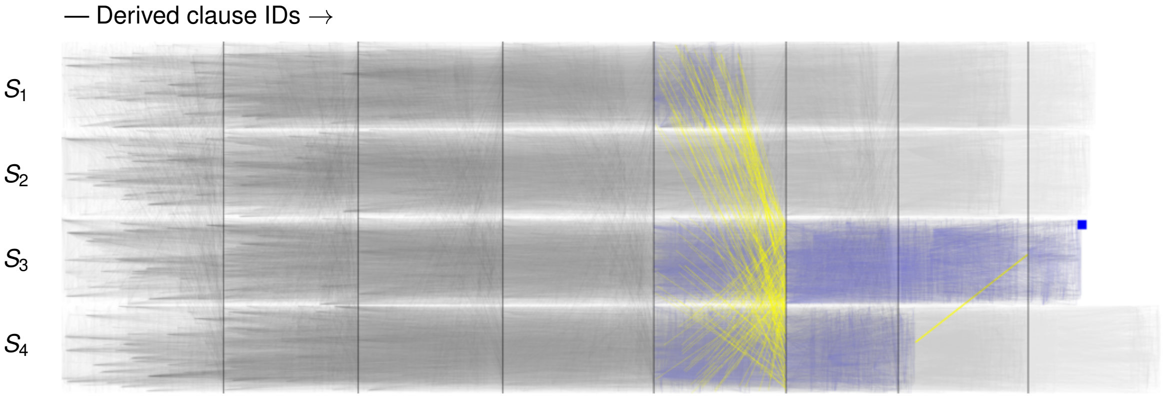
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

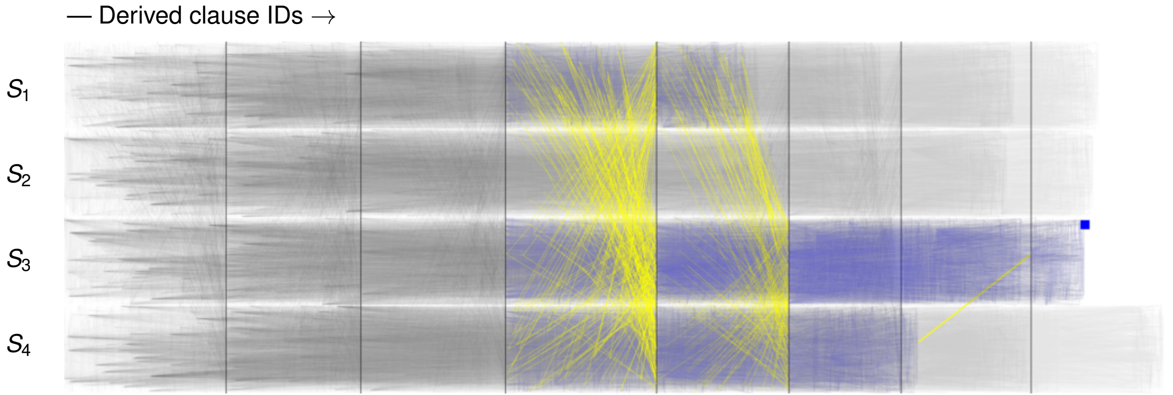
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

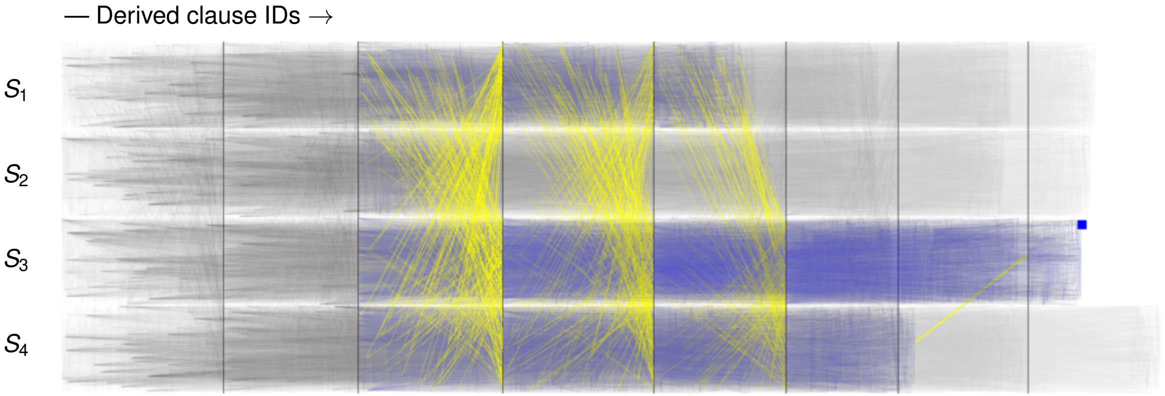
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

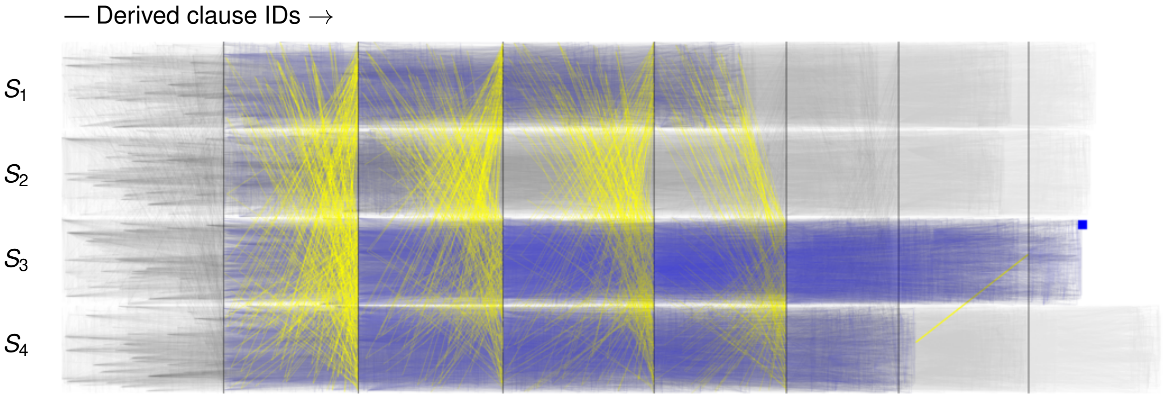
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

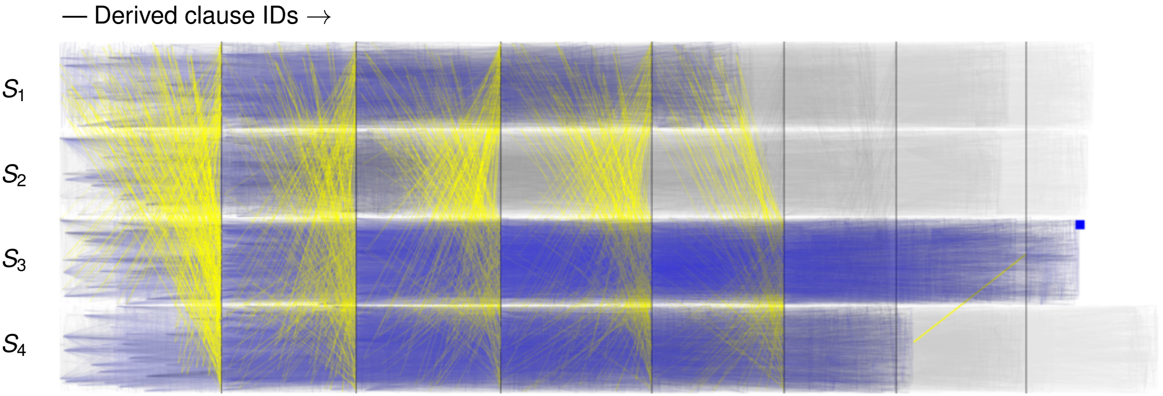
Distributed Pruning: Real Data



180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

Distributed Pruning: Real Data

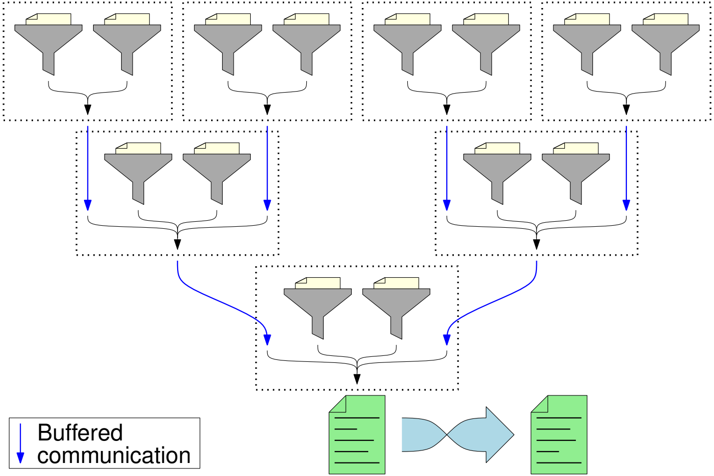


180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Rewind: Trace local required clause IDs, redistribute remote IDs just before reading their epoch of origin

Distributed Combination

- Hierarchically merge pruning output along tree of processors
- Root processor
  1. adds approximated “delete” lines
  2. writes stream into file
  3. reverses file



Experimental Setup (1/2)

Technology

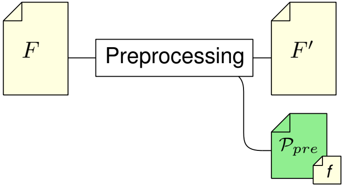
- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Experimental Setup (1/2)

Technology

- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Pipeline

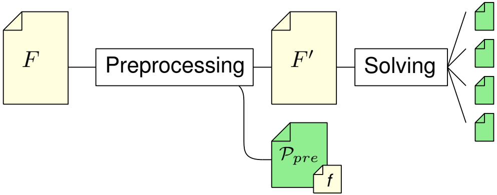


Experimental Setup (1/2)

Technology

- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Pipeline

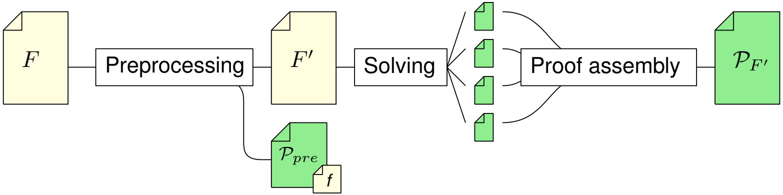


Experimental Setup (1/2)

Technology

- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Pipeline

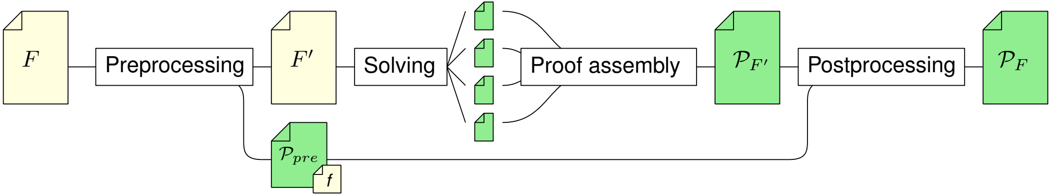


Experimental Setup (1/2)

Technology

- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Pipeline

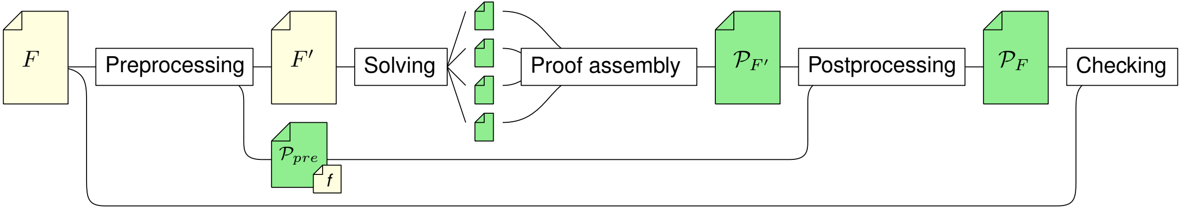


Experimental Setup (1/2)

Technology

- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Pipeline



Experimental Setup (2/2)

Comparison to prior work

- Shared-memory clause-sharing portfolios: Heule, Manthey, Philipp @ POS'14
  - Synchronized, moderated logging into shared DRAT proof
  - Solver not competitive ⇒ Simulate proof output, compare checking times only
- Sequential SAT solving: `Kissat_MAB-HyWalk` @ SAT Comp. 2022

Experimental Setup (2/2)

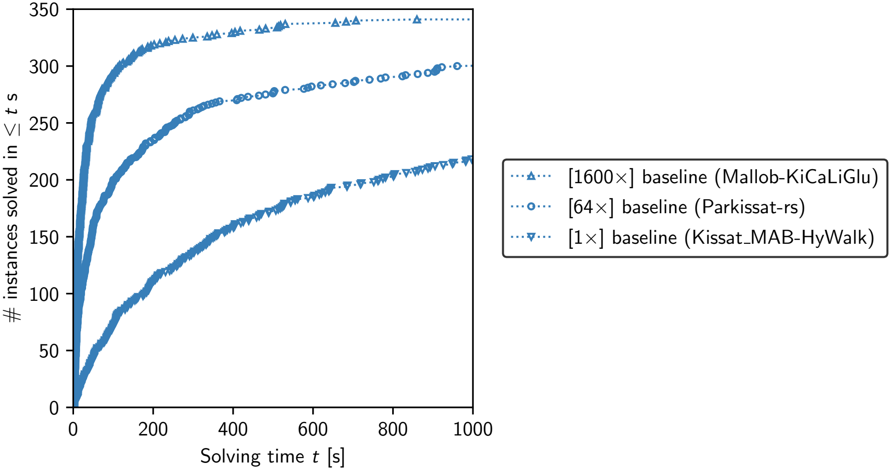
Comparison to prior work

- Shared-memory clause-sharing portfolios: Heule, Manthey, Philipp @ POS'14
    - Synchronized, moderated logging into shared DRAT proof
    - Solver not competitive ⇒ Simulate proof output, compare checking times only
- Sequential SAT solving: `Kissat_MAB-HyWalk` @ SAT Comp. 2022

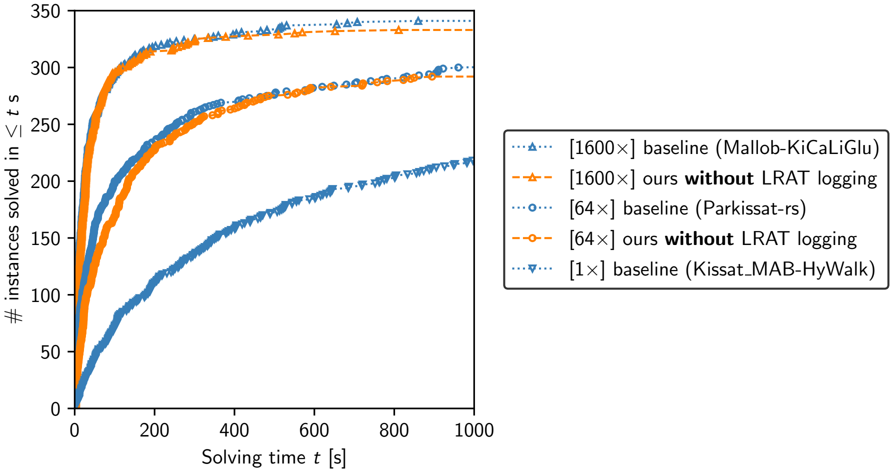
Resources

$$\left.\begin{array}{l}\text{1600× setup: 100× \texttt{m6i.4xlarge} EC2 instances (16 hwthreads, 64 GB RAM)}\\ \text{64× setup: 1× \texttt{m6i.16xlarge} EC2 instance (64 hwthreads, 256 GB RAM)}\\ \text{Sequential setup: One \texttt{m6i.4xlarge} EC2 instance}\end{array}\right\}\begin{array}{l}\leq 1000\,\text{s solving}\\ \leq 4000\,\text{s proof prod.}\end{array}$$

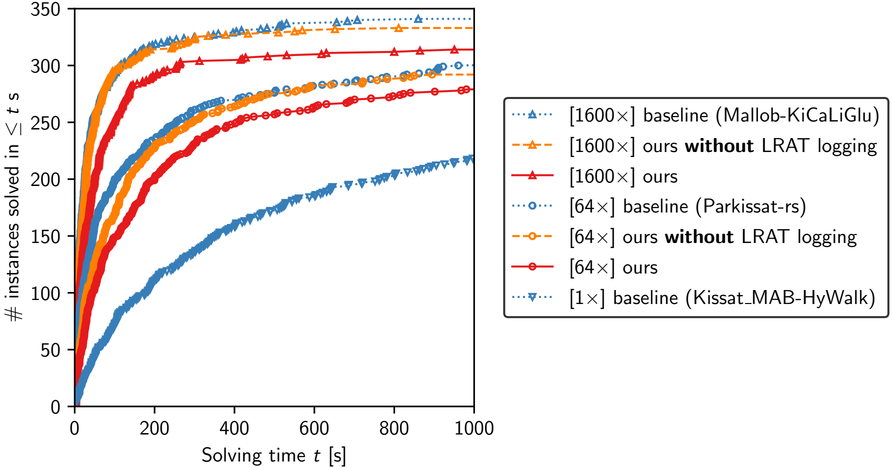
Evaluation: Solving Times



Evaluation: Solving Times

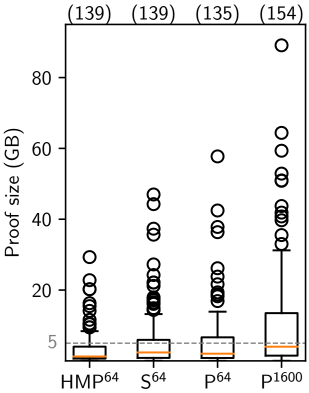


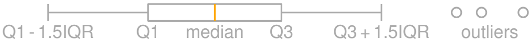
Evaluation: Solving Times



Evaluation: Proof Output

How large are the resulting proofs?

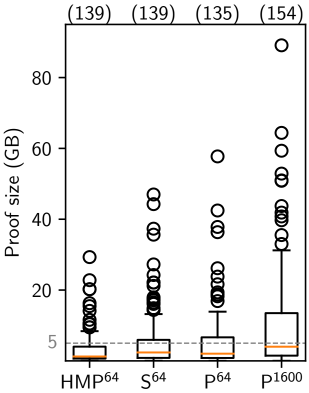




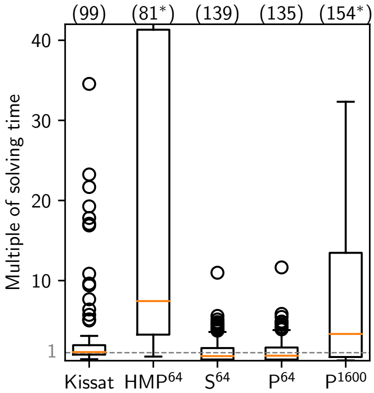
*Some data cut off

Evaluation: Proof Output

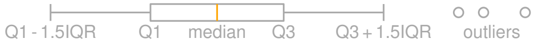
How large are the resulting proofs?



How fast can we check the proofs?



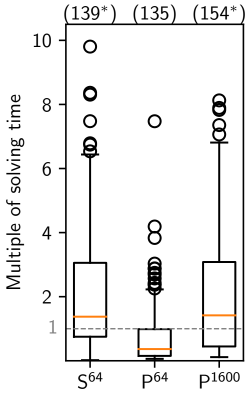
*Some data cut off

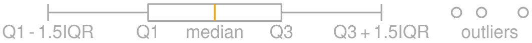


Q1 - 1.5IQR Q1 median Q3 Q3 + 1.5IQR outliers

Evaluation: Overhead

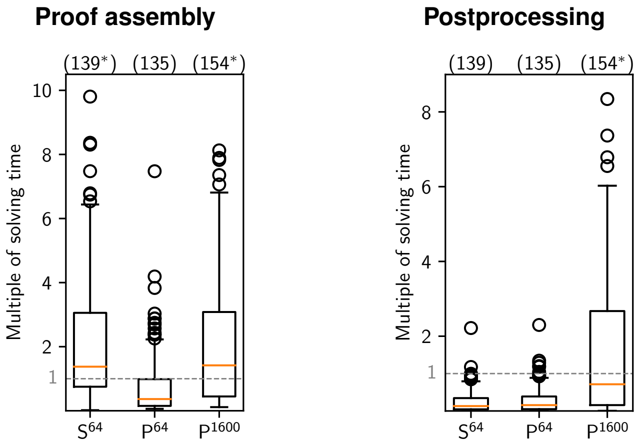
Proof assembly





*Some data cut off

Evaluation: Overhead



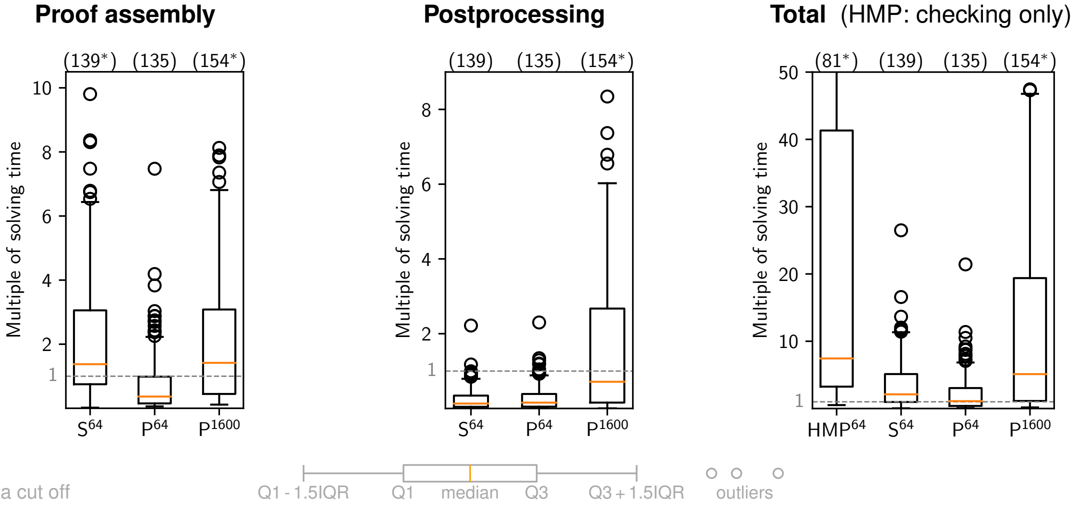
Proof assembly

Postprocessing

Q1 - 1.5IQR
Q1
median
Q3
Q3 + 1.5IQR
outliers

*Some data cut off

Evaluation: Overhead



*Some data cut off

Conclusion

Takeaways

- Popular parallelization approaches for SAT (“antisocial nerds” analogy)
  — Search space splitting, Cube&Conquer
  — Pure portfolio
  — Clause sharing portfolio
- All-to-all clause sharing can be very useful and scalable (up and down) if implemented well
  — huge for unsatisfiable, nice-to-have for satisfiable problems
  — diversifies solvers effectively in and of itself
- Exploit embarrassingly parallel job processing for interactive solving & best efficiency
- Emitting proofs of unsatisfiability is nontrivial and requires careful engineering

Conclusion

Takeaways

- Popular parallelization approaches for SAT (“antisocial nerds” analogy)
  - — Search space splitting, Cube&Conquer
  - — Pure portfolio
  - — Clause sharing portfolio
- All-to-all clause sharing can be very useful and scalable (up and down) if implemented well
  - — huge for unsatisfiable, nice-to-have for satisfiable problems
  - — diversifies solvers effectively in and of itself
- Exploit embarrassingly parallel job processing for interactive solving & best efficiency
- Emitting proofs of unsatisfiability is nontrivial and requires careful engineering

Recent and ongoing work

- Distributed incremental SAT solving with Mallob
- QBF solving with Mallob

`https://github.com/domschrei/mallob`

References

Publications

Balyo, T., Sanders, P., & Sinz, C. (2015). **Hordesat: A massively parallel portfolio SAT solver**. In Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, 2015, Proceedings 18 (pp. 156-172).

Biere, A. (2010). **Lingeling, Plingeling, Picosat and Precosat at SAT race 2010**.

Ehlers, T., & Nowotka, D. (2019). **Tuning parallel sat solvers**. Proceedings of Pragmatics of SAT, 59, 127-143.

Hamadi, Y., Jabbour, S., & Sais, L. (2010). **ManySAT: a parallel SAT solver**. Journal on Satisfiability, Boolean Modeling and Computation, 6(4), 245-262.

Michaelson, D., Schreiber, D., Heule, M. J., Kiesl-Reiter, B., & Whalen, M. W. (2023). **Unsatisfiability proofs for distributed clause-sharing SAT solvers**. In Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (pp. 348-366).

Roussel, O. (2012). **Description of ppfolio (2011)**. Proc. SAT Challenge, 46.

Sanders, P., & Schreiber, D. (2022,). **Decentralized online scheduling of malleable NP-hard jobs**. In Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing, 2022, Proceedings (pp. 119-135).

Schreiber, D. (2022). **Mallob in the SAT competition 2022**. Proc. SAT Competition, 38.

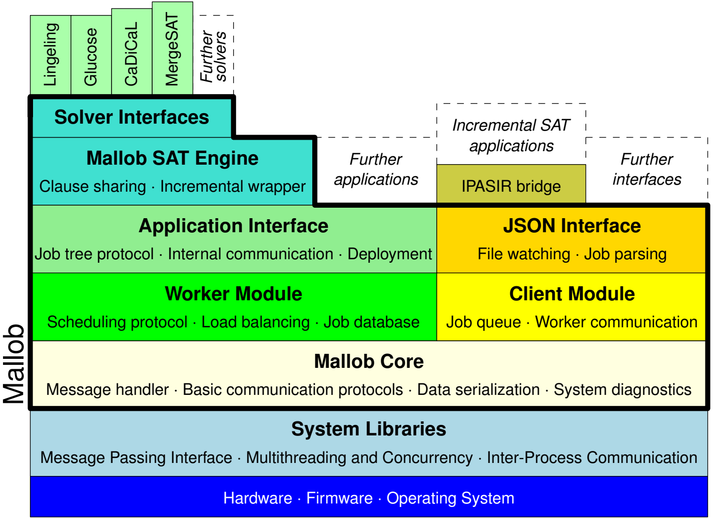
Schreiber, D., & Sanders, P. (2021). **Scalable SAT solving in the cloud**. In Theory and Applications of Satisfiability Testing–SAT 2021: 24th International Conference, 2021, Proceedings 24 (pp. 518-534).

External images

Slide 12, SuperMUC-NG: `https://doku.lrz.de/files/10745965/10745966/1/1684599593177/image2019-11-15_12-48-5.png`

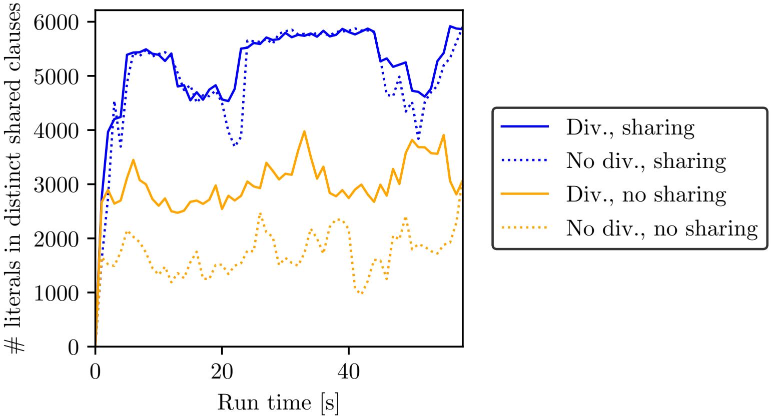
Slide 23, “They’re the same picture.” meme:
`https://cdn.eldeforma.com/wp-content/uploads/2020/08/theyre-the-same-picture-pam-the-office-meme-1024x580.png`

Mallob



Lingeling
Glucose
CaDiCaL
MergeSAT
Further solvers
Solver Interfaces
Mallob SAT Engine
Clause sharing · Incremental wrapper
Further applications
Incremental SAT applications
IPASIR bridge
Further interfaces
Application Interface
Job tree protocol · Internal communication · Deployment
JSON Interface
File watching · Job parsing
Worker Module
Scheduling protocol · Load balancing · Job database
Client Module
Job queue · Worker communication
Mallob Core
Message handler · Basic communication protocols · Data serialization · System diagnostics
Mallob
System Libraries
Message Passing Interface · Multithreading and Concurrency · Inter-Process Communication
Hardware · Firmware · Operating System

Sharing vs. diversification



4× default-configured Lingeling, random 3-SAT @ PT, 400 vars, no unused volume compensation

Scaling Experiments (2021)

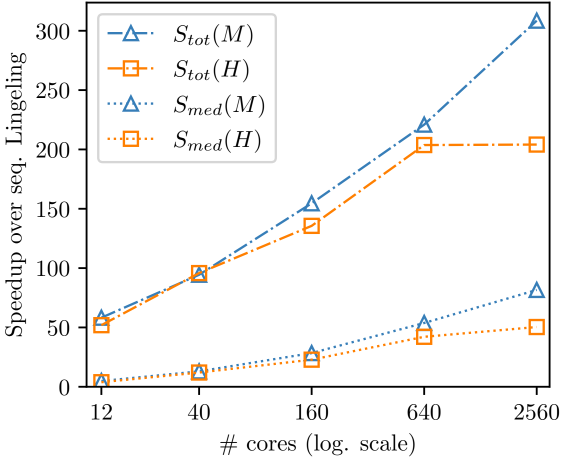
Mallob-mono$^{AnyLBD}_{sublin}$ vs. HordeSat$_{new}$

Speedups

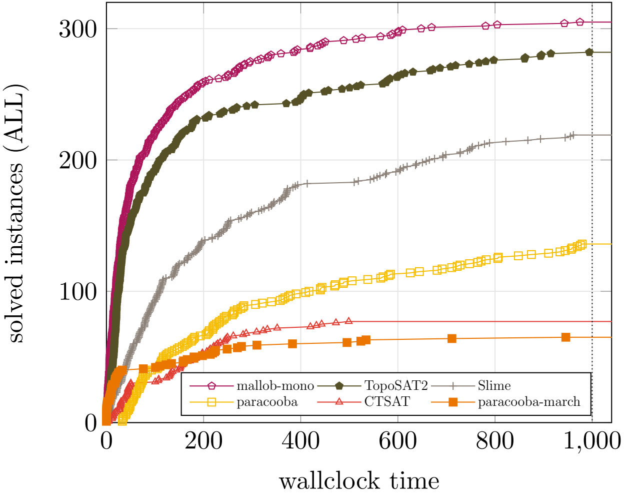
Instance F solved by parallel approach
$\Rightarrow$ Par. run time $T_{par}(F) \leq 300\,s$
$\Rightarrow$ Seq. run time $T_{seq}(F) \leq 50\,000\,s$
$(T_{seq}(F) := 50\,000\,s$ if unsolved)

Total speedup S_{tot}:
$\sum_F T_{seq}(F) \;/\; \sum_F T_{par}(F)$

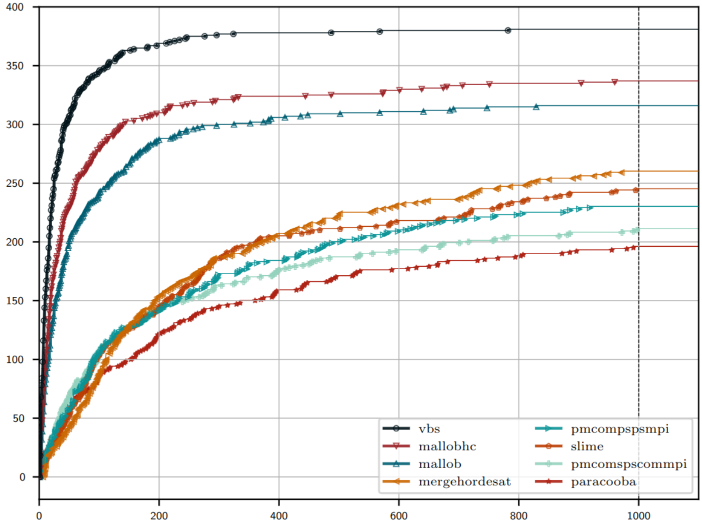
Median speedup S_{med}:
$median_F\{T_{seq}(F)/T_{par}(F)\}$



SAT Competition 2020 (Cloud Track)



SAT Competition 2021 (Cloud Track)



- MallobHC: mixed solver portfolio
- VBS of all Main track solvers solved 325 instances within 5000 s