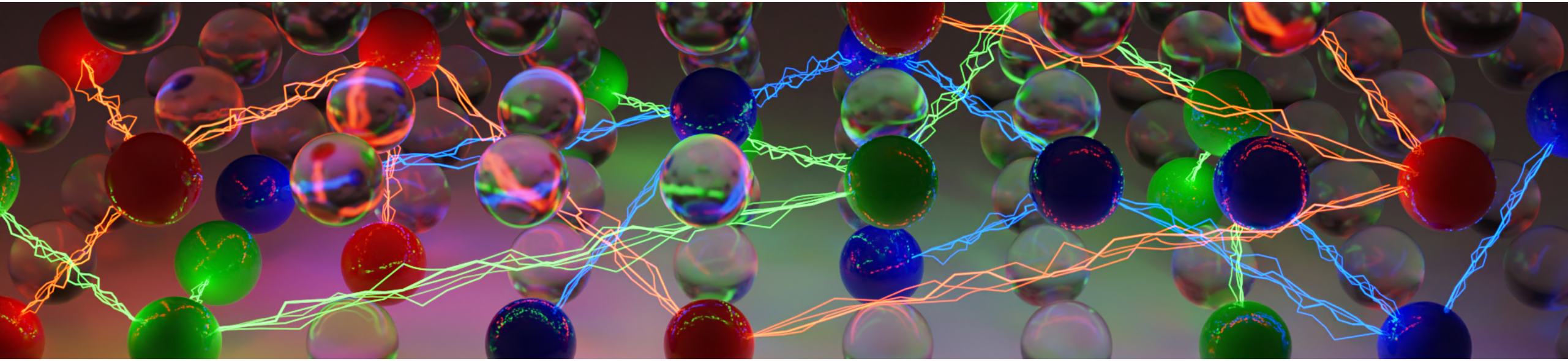
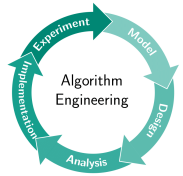


Practical SAT Solving



Lecture 8 – Parallel SAT Solving

Markus Iser, Dominik Schreiber | June 16, 2025

SAtRes
Scalable Automated Reasoning

Overview

Recap Lecture 7

- Propagation-based Redundancy notions
- Proof systems: Resolution, Extended Resolution, Blocked Clauses, Implication-based Redundancy
- Autarkies, Conditional Autarkies, and Satisfaction Driven Clause Learning (SDCL)

Today: Parallel SAT Solving

Parallel SAT solving approaches

- Basic search space splitting, Clause sharing, Cube&Conquer, Portfolio solvers

A deep dive into Mallob

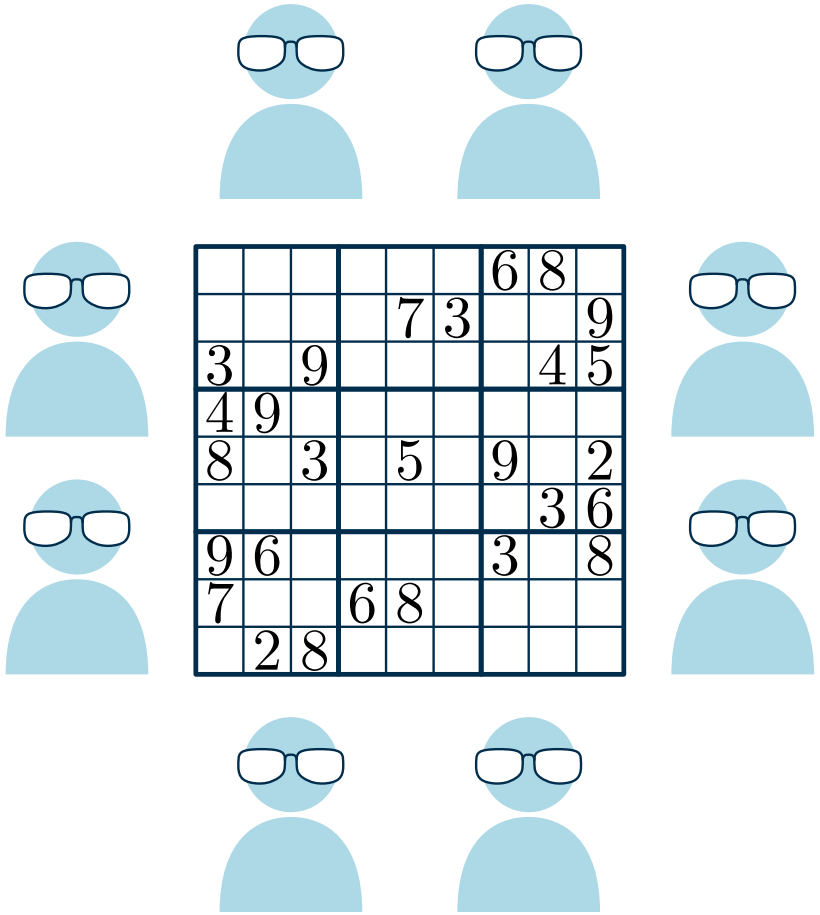
- Overview, Scalable clause sharing, Experiments and insights

Parallel Solving: An analogy

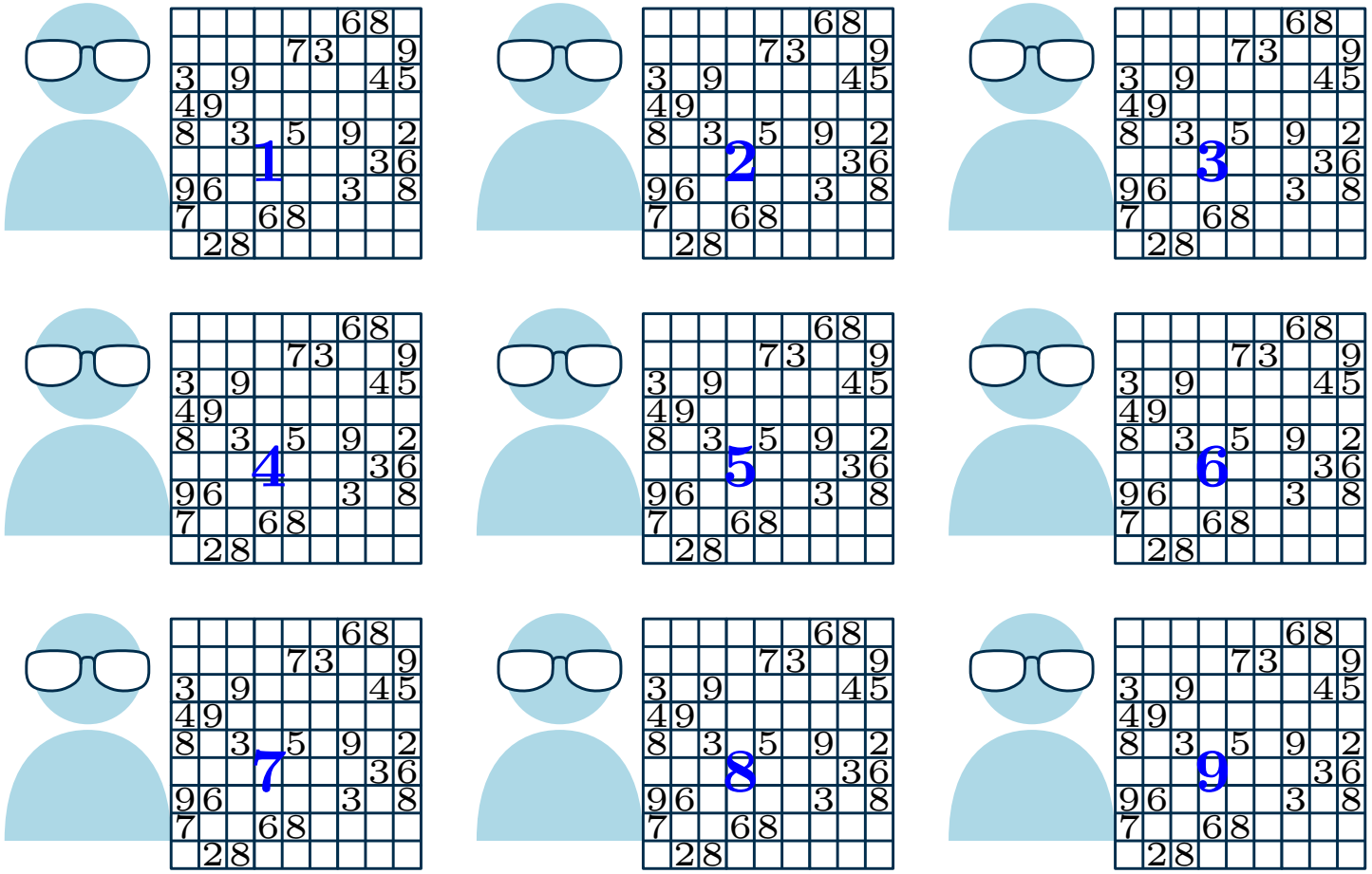
The Assembly of Nerds [23]

- Complex and large logic puzzle
- *n* puzzle experts at your disposition
  — anti-social: work best if left undisturbed

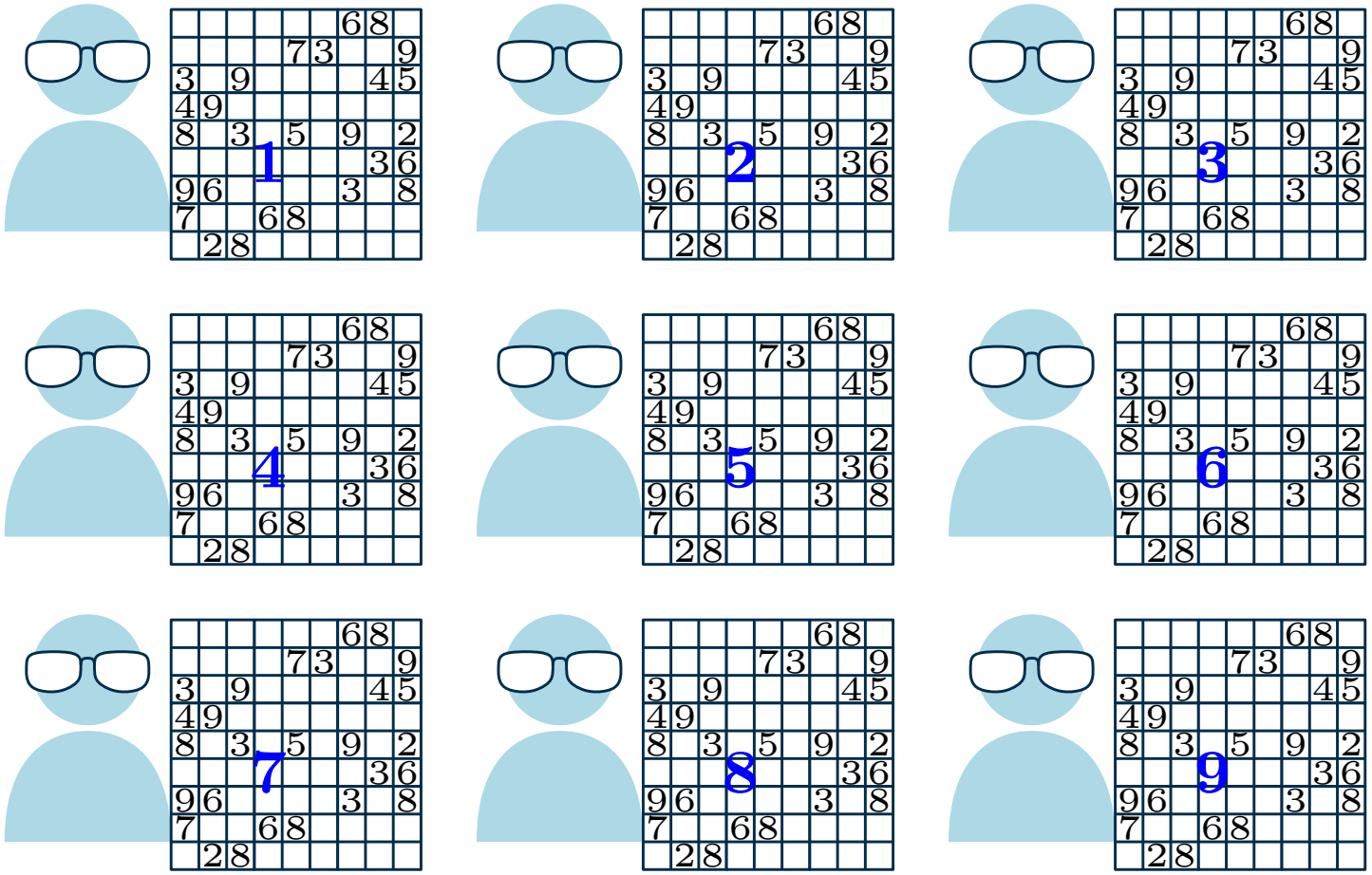
How do we employ and “orchestrate” our experts?



Approach I: Search Space Partitioning



Approach I: Search Space Partitioning



- Partition search space at some decisions
  $\Rightarrow$ Independent subproblems

Explicit Partitioning

Böhm & Speckenmeyer (1994-1996) [5]: 1st Parallel DPLL Implementation

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

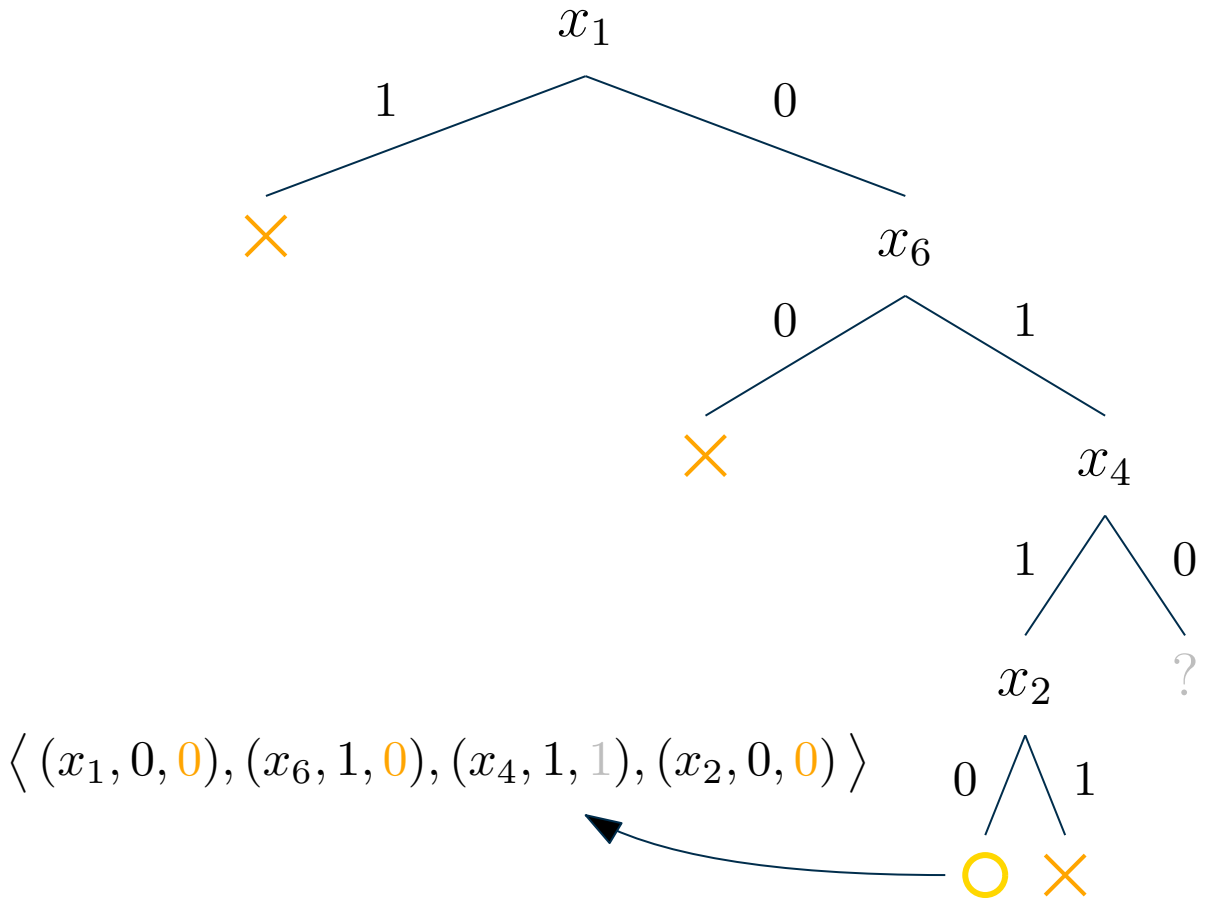
"**PSATO**: a Distributed Propositional Prover and its Application to Quasigroup Problems", Zhang et al. (1996) [26]

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)



Explicit Partitioning Solvers

SATZ (Jurkowiak et al., 2001) [15]: *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

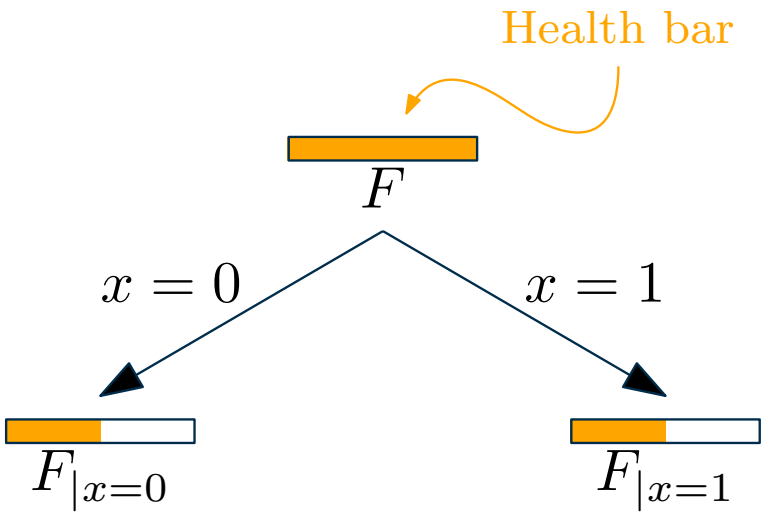
PaSAT (Blochinger et al., 2001-2003) [4]

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

ySAT (Feldman et al., 2004) [8]

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

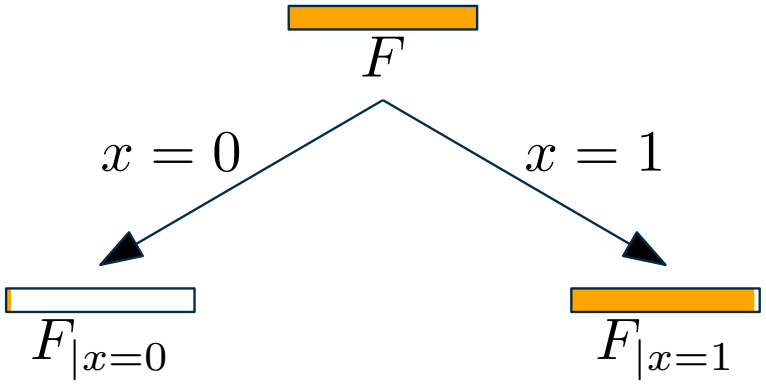
Problems with Partitioning [25]



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 [25]



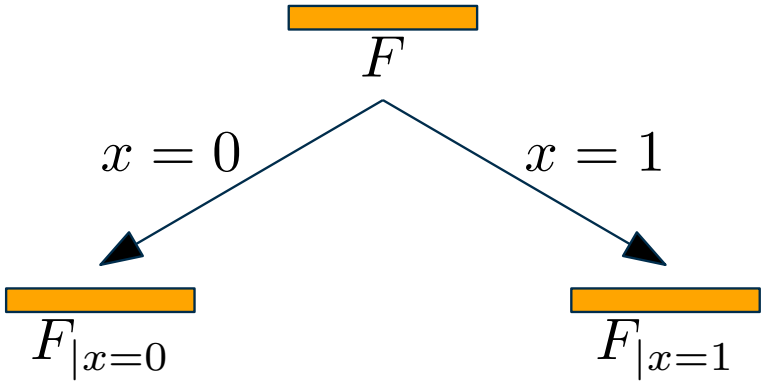
What we want: **Even splits**

- 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 [25]



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) [14]

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) [14]

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) [14]

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

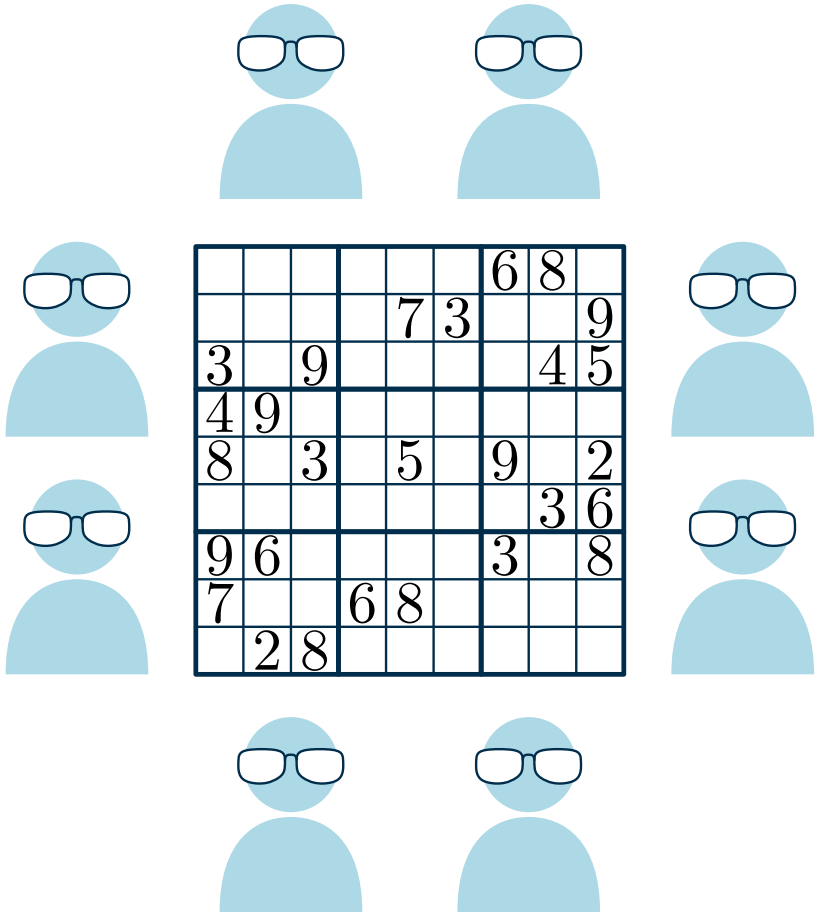
- Unlikely that any of the workers will run out tasks
  $\Rightarrow$ Hope of good load balancing in practice
- Partial assignments are generated using a look-ahead solver
  (breadth-first search up to a limited depth)
- Best performance mostly with problem-specific decision heuristics
- State-of-the-art for hard combinatorial problems
  - Used to solve the “Pythagorean Triples” problem ($\sim$200TB proof) [13]
  - ... or more recently “Schur Number 5” ($\sim$2PB proof) [12]
- Examples: March (Heule) + iLingeling (Biere) introduced in 2011; Treengeling (Biere) [2]

Parallel Portfolios: An analogy

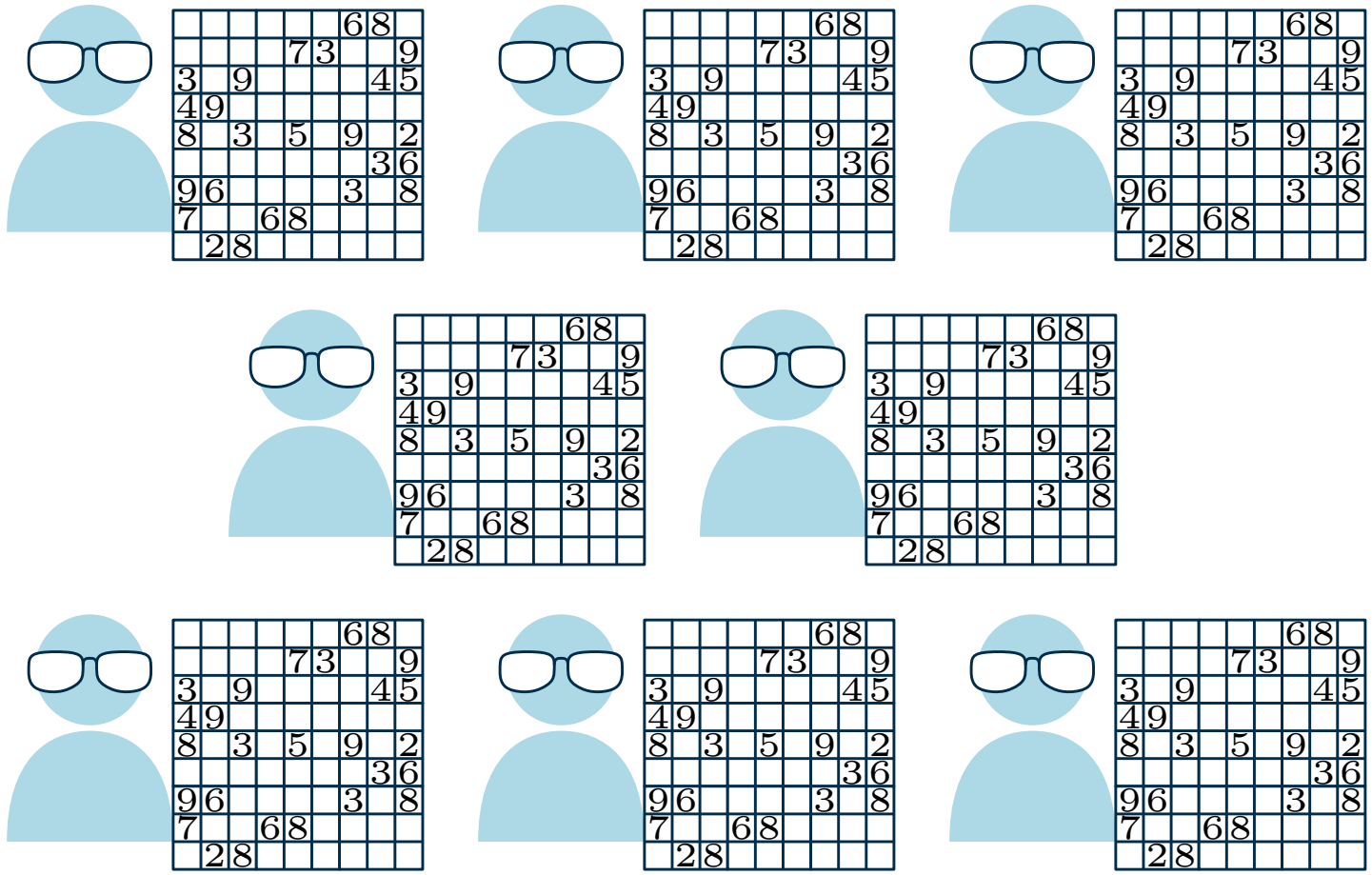
The Assembly of Nerds

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

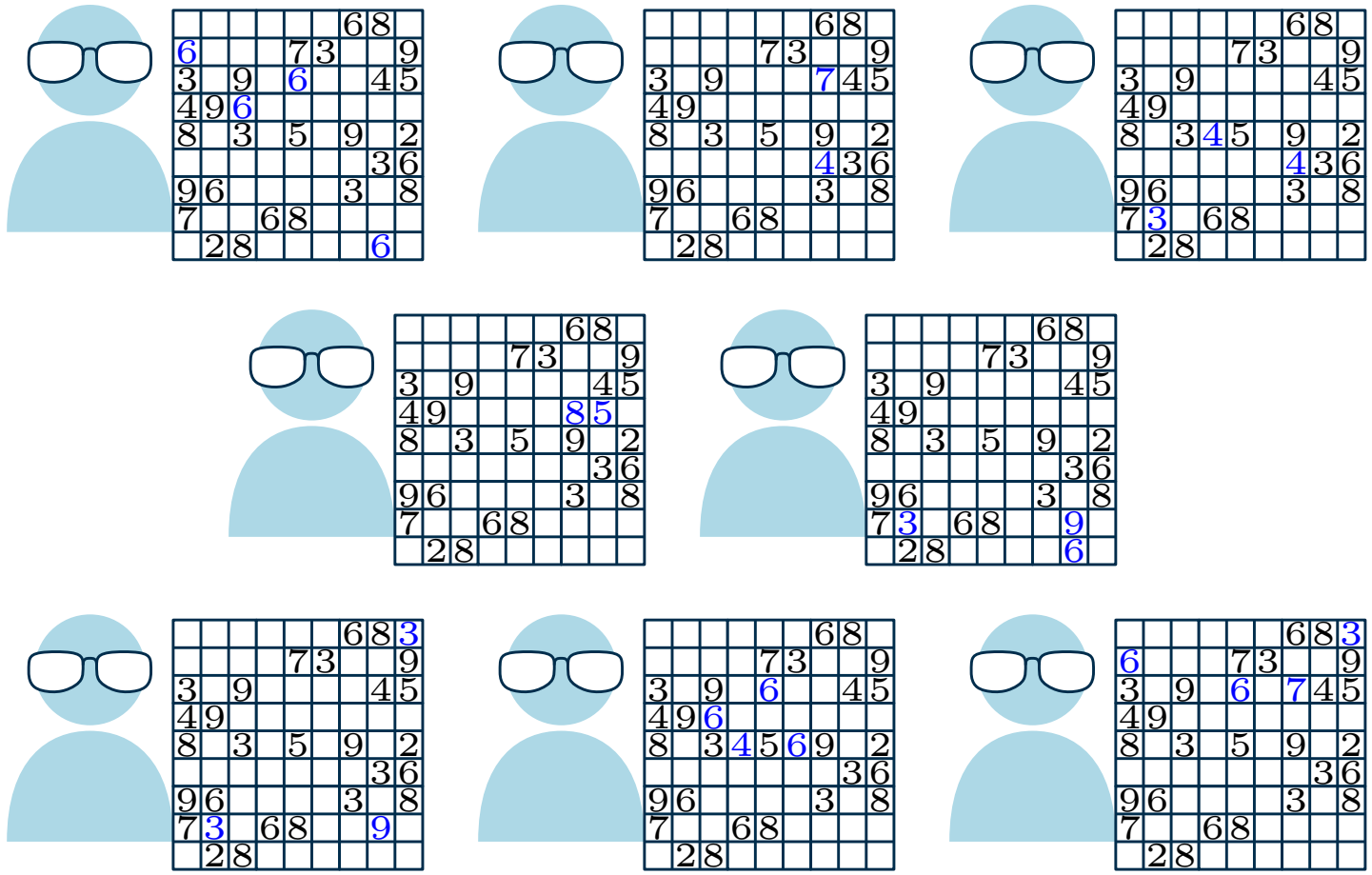
How do we employ and “orchestrate” our experts?



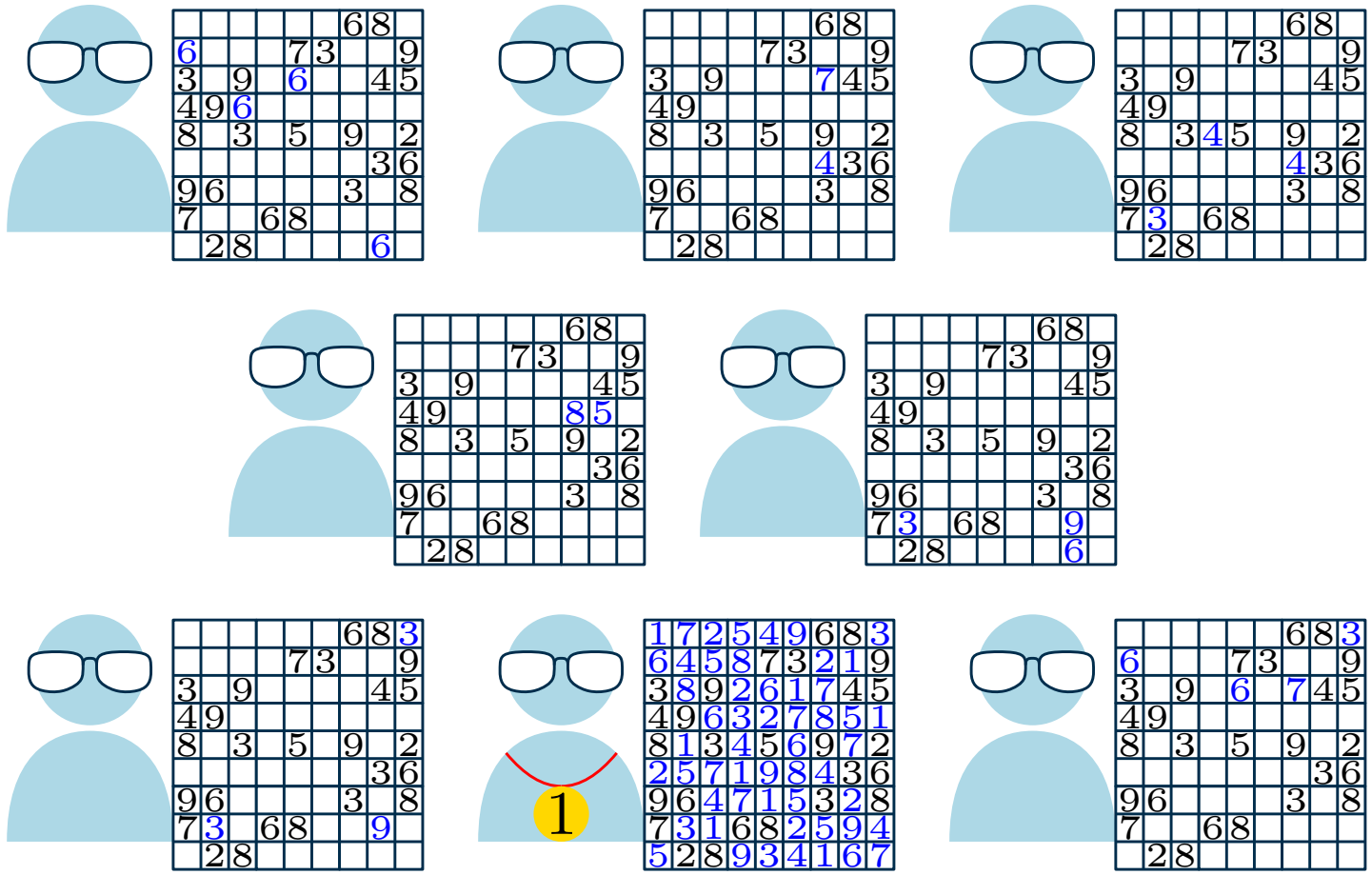
Approach II: Pure Portfolio



Approach II: Pure Portfolio



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 [20]

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 [20]

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 [18]

- 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*"

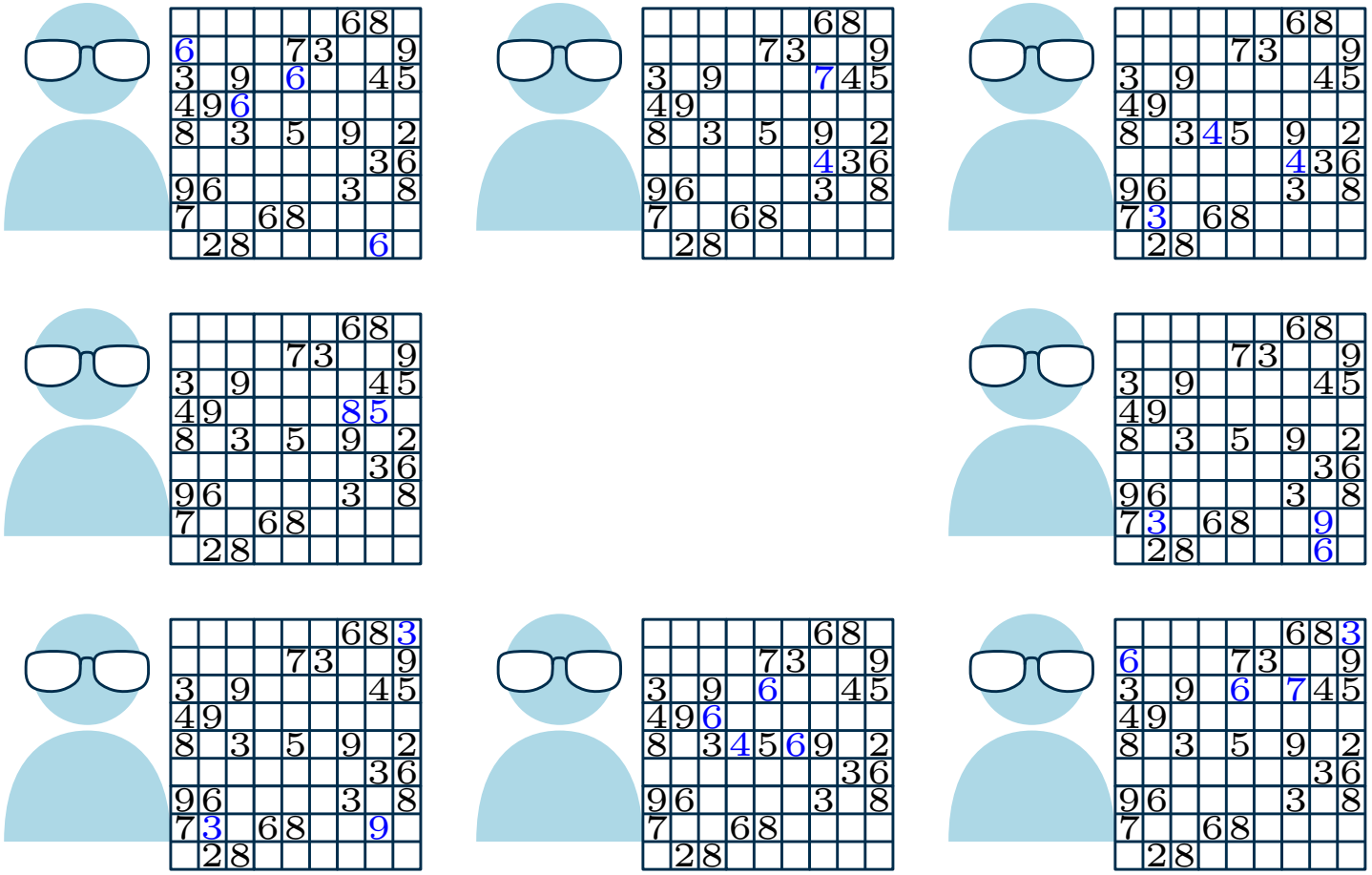
Pure SAT Portfolios

ppfolio: Winner of Parallel Track in the 2011 SAT Competition [18]

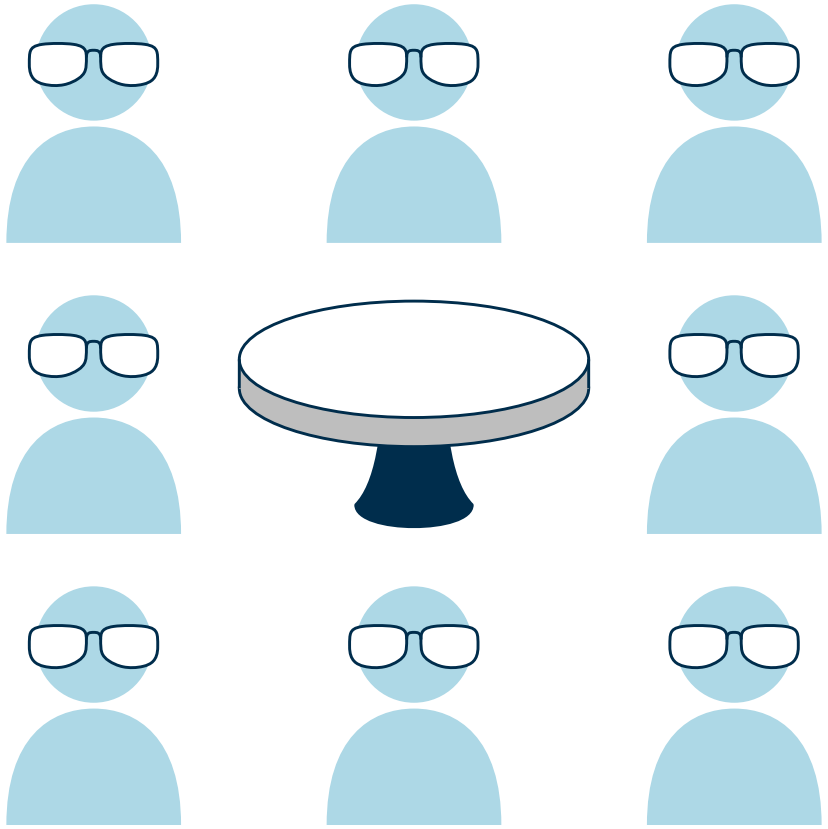
- Just a bash script combining the best sequential solvers from 2010:
  `~$ ./solver1 f.cnf & ./solver2 f.cnf & ./solver3 f.cnf & ./solver4 f.cnf`
- Bits by O. Roussel, the author of `ppfolio`:
  — "*by definition the best solver on Earth*"
  — "*probably the laziest and most stupid solver ever written*"

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

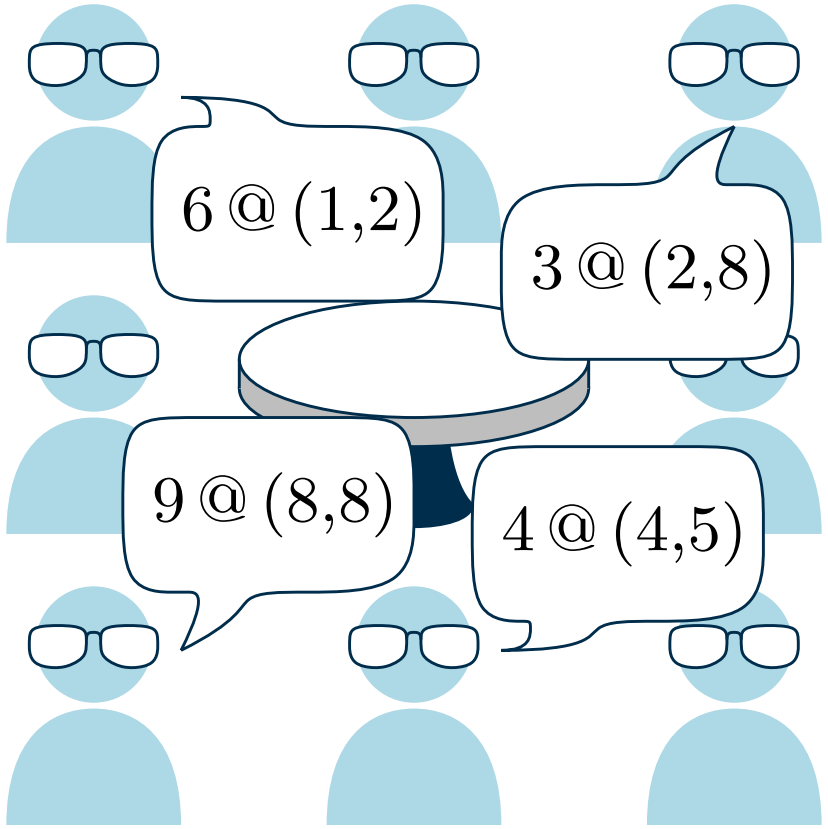
Approach II+: Cooperative Portfolio



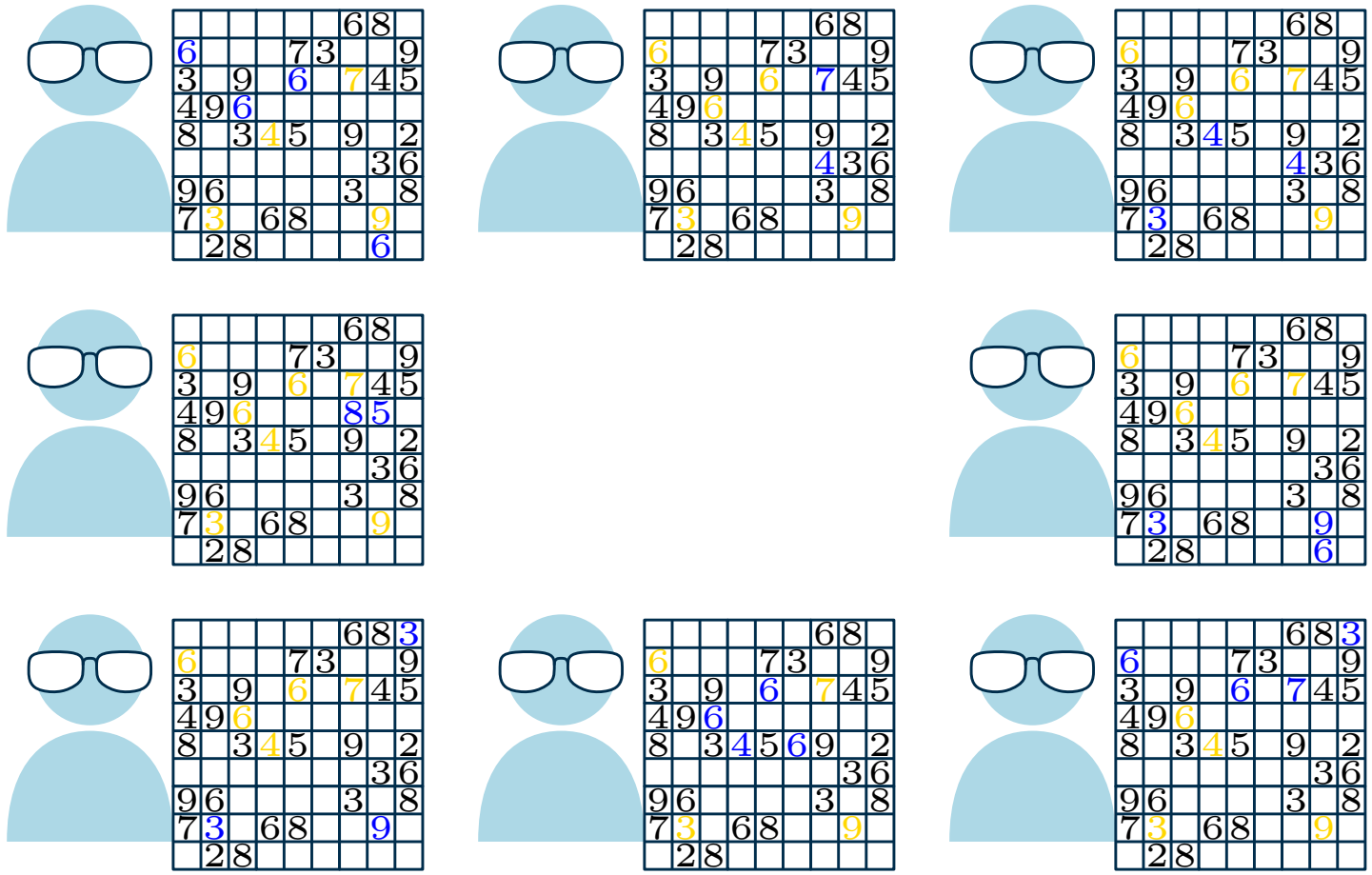
Approach II+: Cooperative Portfolio



Approach II+: Cooperative Portfolio



Approach II+: Cooperative Portfolio



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
- Result: Parallel search space pruning procedure

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
- Result: Parallel search space pruning procedure

Combination of portfolio idea + clause exchanges: **Clause sharing portfolio solvers**

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) [11]

- 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) [11]

- 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) [3]

- 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, High-Performance Computing (HPC)

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 $\times$ 48 cores

Massively parallel hardware?

Distributed computing, High-Performance Computing (HPC)

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 $\times$ 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?

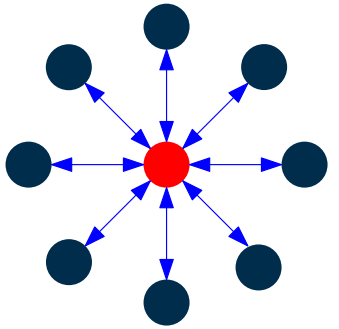
Distributed computing, High-Performance Computing (HPC)

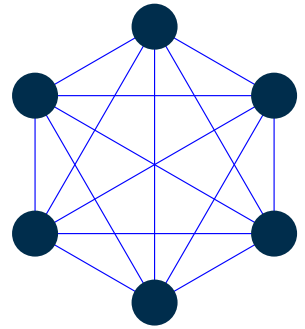
In distributed computing, several machines
(with no shared main memory) run together.
On each machine we run a number of processes,
each of which runs on a number of cores.
Processes commonly communicate by exchanging messages.

SuperMUC-NG: 6 336 nodes $\times$ 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 [7]
  - "Star graph": Master process collects, serves all exported clauses
  - Naïve (quadratic) all-to-all exchange of clauses
- HPC schedulers, administrators, committees are used to regular, easily parallelizable code with near-linear scaling





Massively parallel SAT portfolio

HordeSat (Balyo, Sanders, Sinz 2015) [1]

- 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) [1]

- 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

Clause Exchange in HordeSat

HordeSat's sharing logic

- Only clauses with LBD ≤ 2 are considered for sharing
  - Constraint is lifted successively if processes under-produce
- Solver threads write eligible clauses into shared buffer

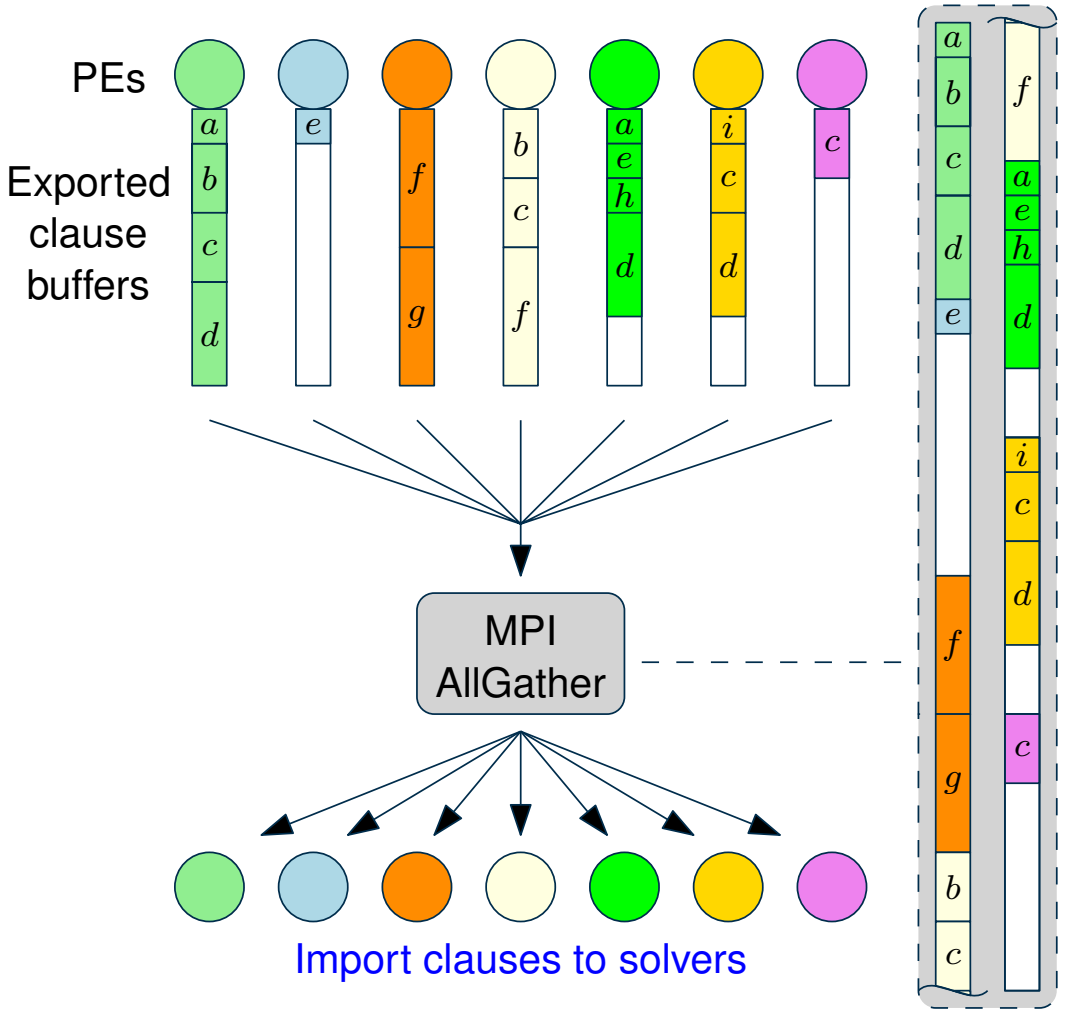
Clause Exchange in HordeSat

HordeSat's sharing logic

- Only clauses with LBD ≤ 2 are considered for sharing
  - Constraint is lifted successively if processes under-produce
- Solver threads write eligible clauses into shared buffer
- Each process' best clauses are shared with everyone
  - Limited to 1500 clause literals per process

$\Rightarrow$ Concatenation of p produced clause buffers

- Approximate, post-hoc filtering of clauses



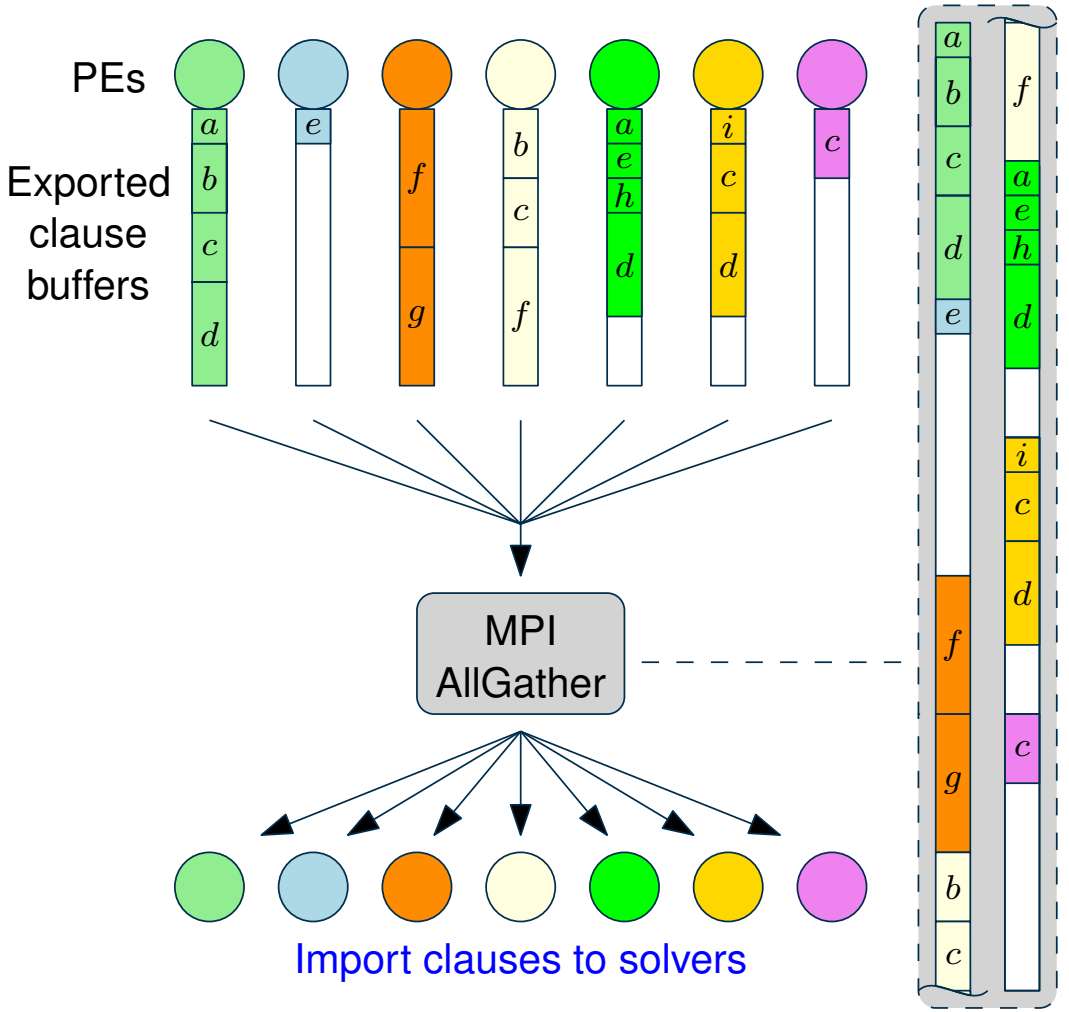
Clause Exchange in HordeSat

HordeSat's sharing logic

- Only clauses with LBD ≤ 2 are considered for sharing
  - Constraint is lifted successively if processes under-produce
- Solver threads write eligible clauses into shared buffer
- Each process' best clauses are shared with everyone
  - Limited to 1500 clause literals per process

$\Rightarrow$ Concatenation of p produced clause buffers

- Approximate, post-hoc filtering of clauses



Issues:

- Many ("high" LBD) clauses are not shared but discarded
- "Holes" in buffer carrying no information
- Duplicate clauses
- Buffer grows proportionally with # proc.
  $\Rightarrow$ Bottleneck w.r.t. communication, local work

HordeSat: Results

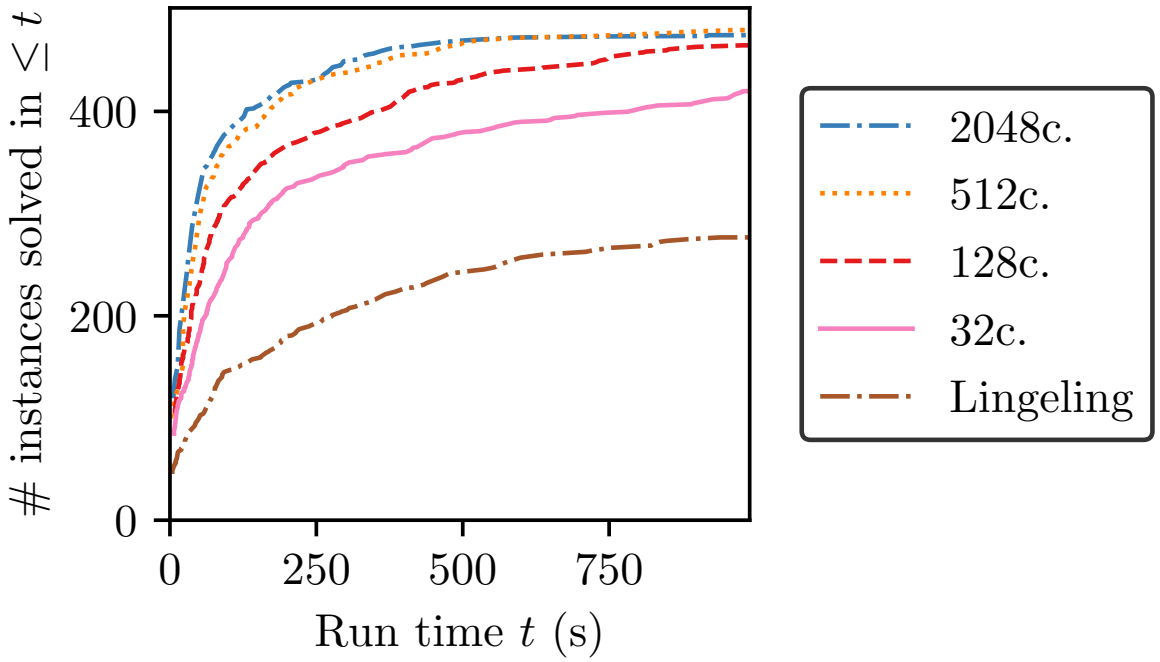
- Super-linear speedups for individual instances
  = speedup $> c$ on c cores! How?

HordeSat: Results

- Super-linear speedups for individual instances
  = speedup $> c$ on c cores! How?
  - SAT: "NP luck" – some solver got lucky
  - UNSAT: distributed memory accommodates more clauses than any sequential solver
  - General: sequential schedule of parallel algorithm may outperform sequential algorithm!

HordeSat: Results

- Super-linear speedups for individual instances
  = speedup $>$ c on c cores! How?
  - SAT: "NP luck" – some solver got lucky
  - UNSAT: distributed memory accommodates more clauses than any sequential solver
  - General: sequential schedule of parallel algorithm may outperform sequential algorithm!
- 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 [1]

From HordeSat to MallobSat

Research Question

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

From HordeSat to MallobSat

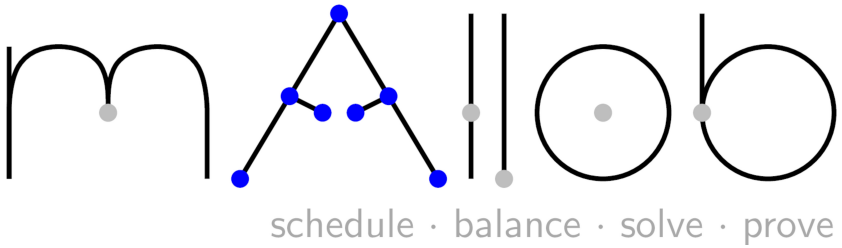
Research Question

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:

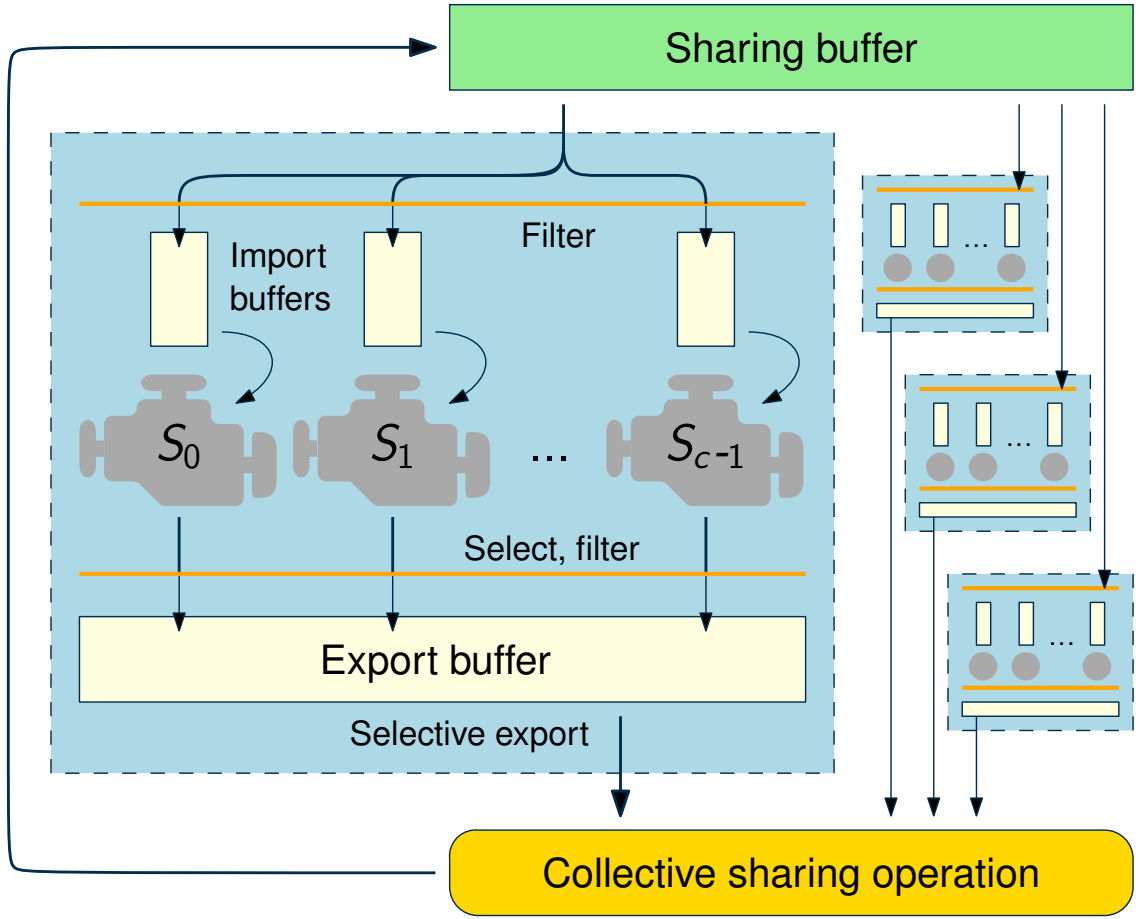
- multi-user, on-demand, malleable scheduling and solving of many problems at once [19]
- distributed SAT engine **MallobSat**: the HordeSat paradigm re-engineered and made efficient [24]
- state-of-the-art SAT performance from dozens to thousands of cores [23]



https://satres.kikit.kit.edu/research/mallob

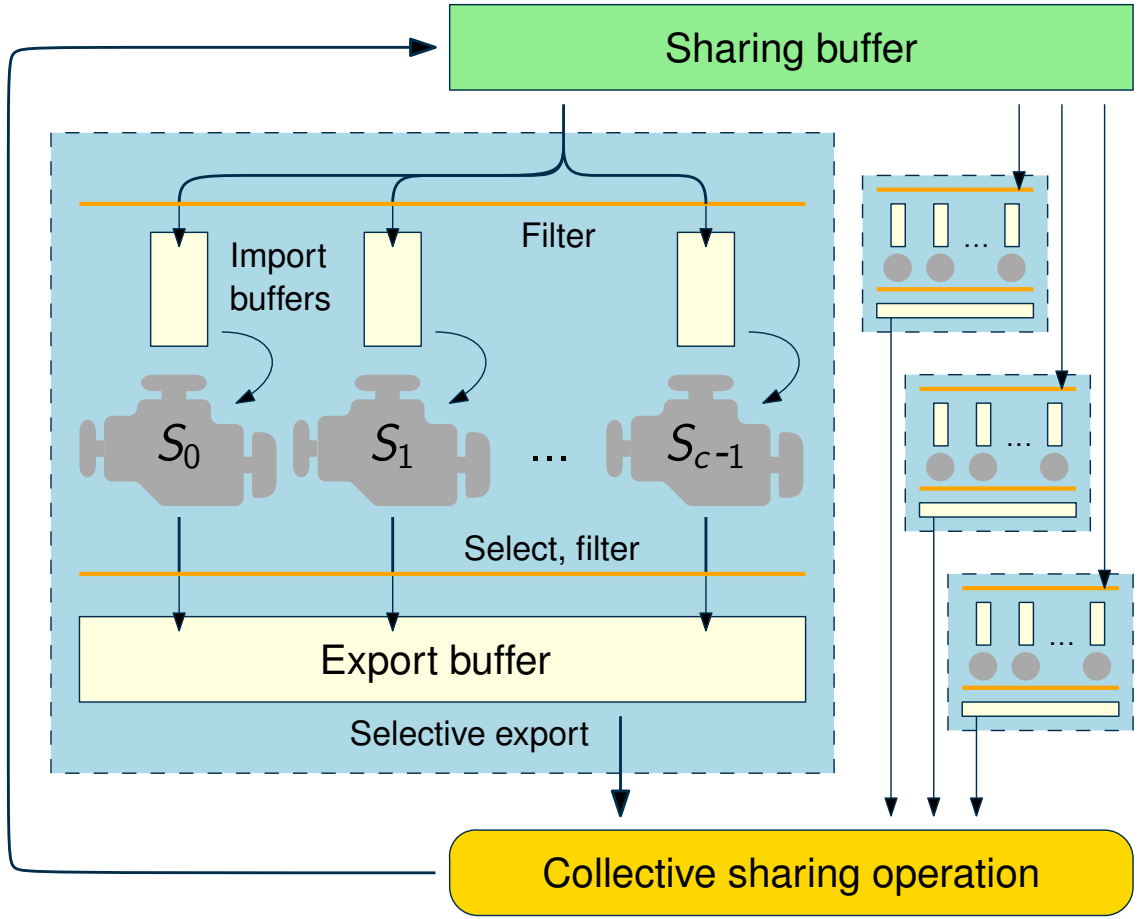
Design Decisions of MallobSat [23]

- **Fundament:** HordeSat
  - Two-level hybrid parallelization
  - Periodic all-to-all clause sharing



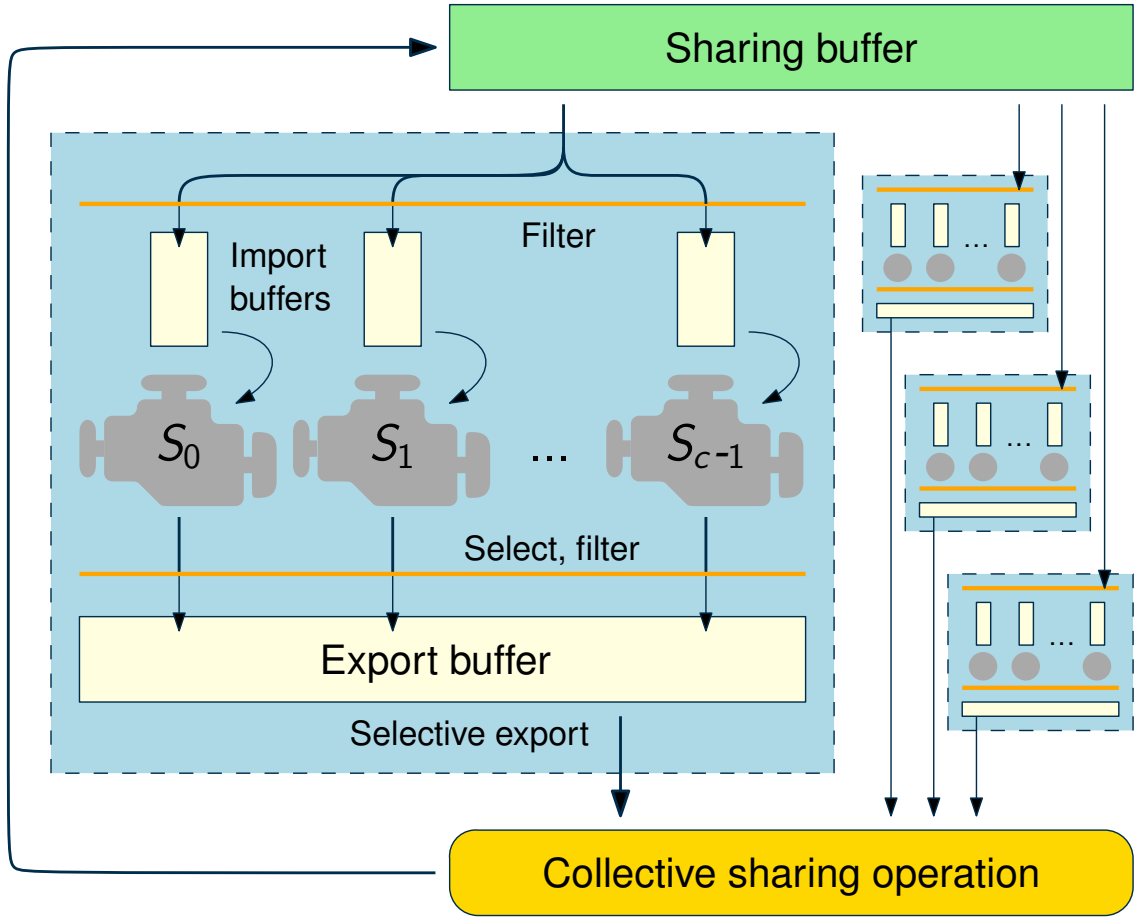
Design Decisions of MallobSat [23]

- **Fundament:** HordeSat
  - Two-level hybrid parallelization
  - Periodic all-to-all clause sharing
- Fix a certain sharing volume, spend it on the globally most useful **distinct** clauses



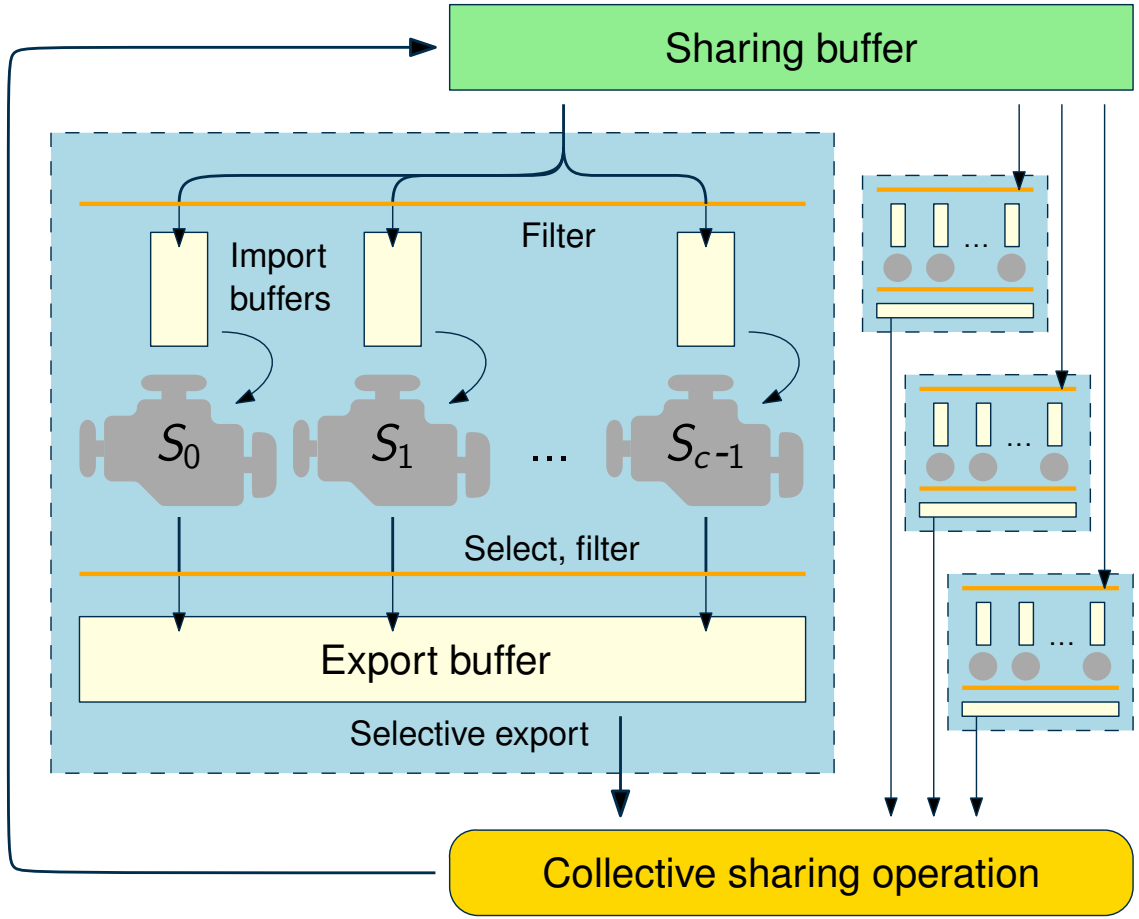
Design Decisions of MallobSat [23]

- **Fundament:** HordeSat
  - Two-level hybrid parallelization
  - Periodic all-to-all clause sharing
- Fix a certain sharing volume, spend it on the globally most useful **distinct** clauses
- Prioritize clauses by clause length over LBD
  - At all stages! Also in export / import buffering



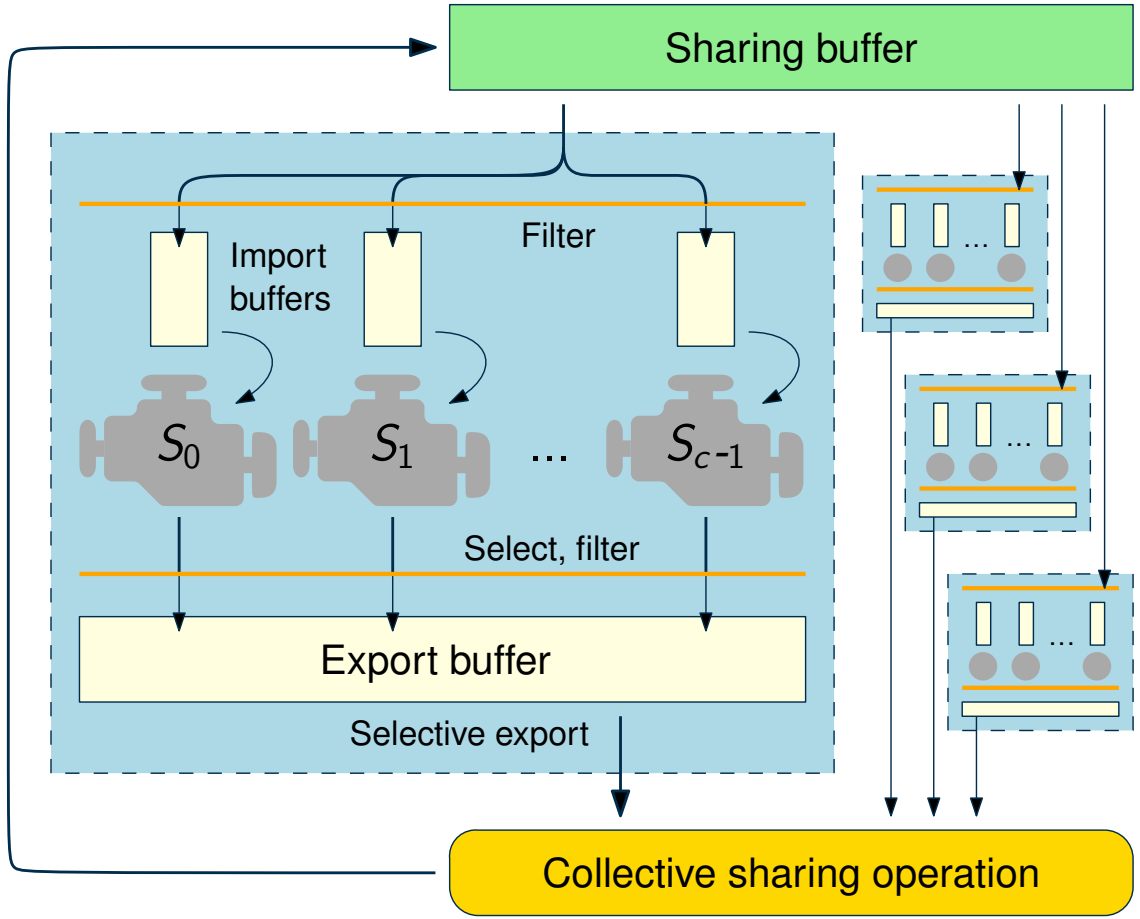
Design Decisions of MallobSat [23]

- **Fundament:** HordeSat
  - Two-level hybrid parallelization
  - Periodic all-to-all clause sharing
- Fix a certain sharing volume, spend it on the globally most useful **distinct** clauses
- Prioritize clauses by clause length over LBD
  - At all stages! Also in export / import buffering
- Minimize clause turnaround times



Design Decisions of MallobSat [23]

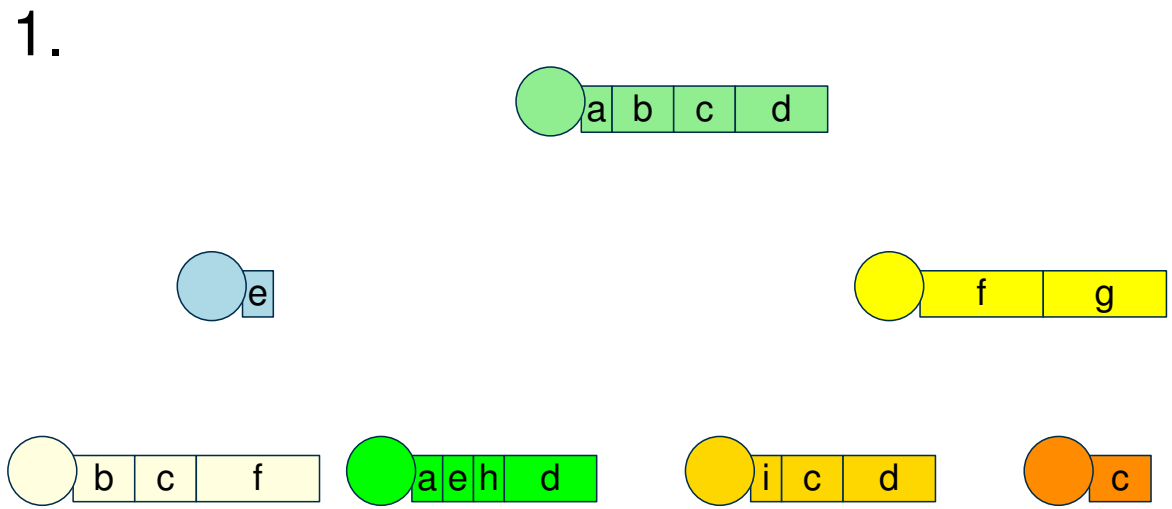
- **Fundament:** HordeSat
  - Two-level hybrid parallelization
  - Periodic all-to-all clause sharing
- Fix a certain sharing volume, spend it on the globally most useful **distinct** clauses
- Prioritize clauses by clause length over LBD
  - At all stages! Also in export / import buffering
- Minimize clause turnaround times
- Support fluctuating workers (*malleability*)



Clause Exchange in Mallob [23]

Custom collective operation

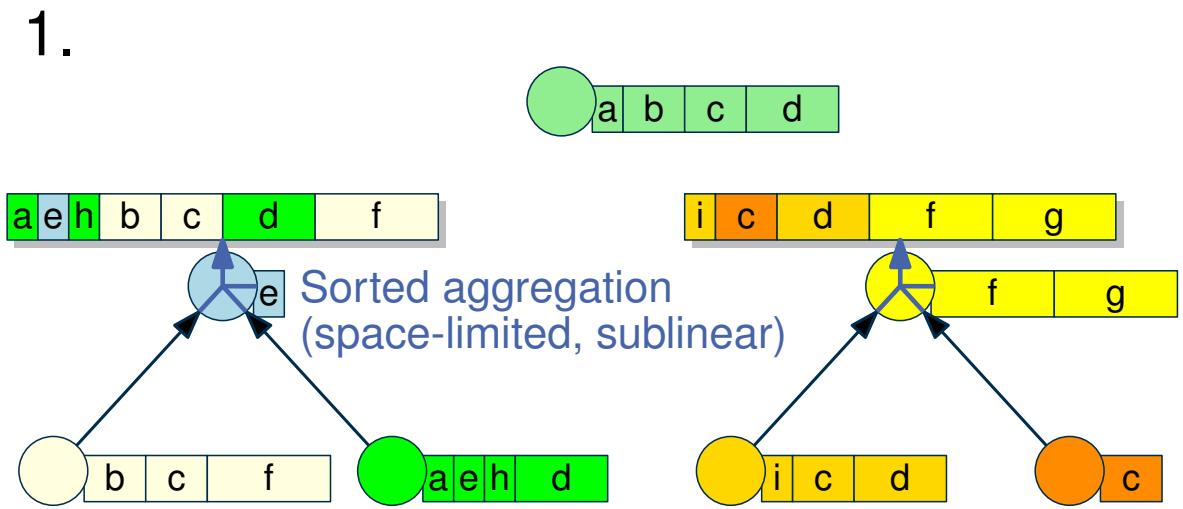
- Aggregate information along binary tree of processors
- Detect duplicates during merge
- Result is of compact shape
- Sublinear buffer size growth: Discard longest clauses as necessary



Clause Exchange in Mallob [23]

Custom collective operation

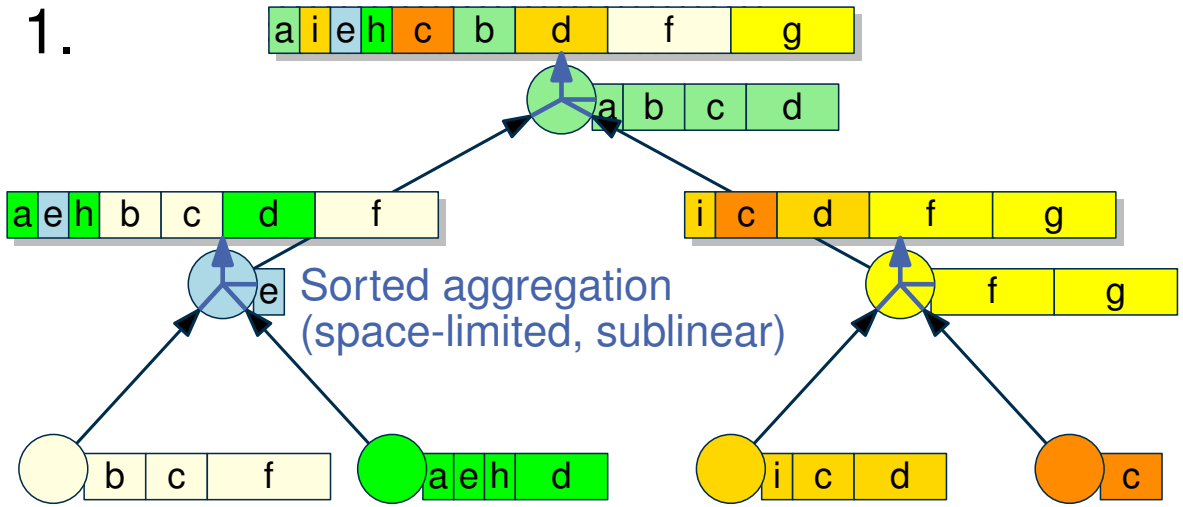
- Aggregate information along binary tree of processors
- Detect duplicates during merge
- Result is of compact shape
- Sublinear buffer size growth: Discard longest clauses as necessary



Clause Exchange in Mallob [23]

Custom collective operation

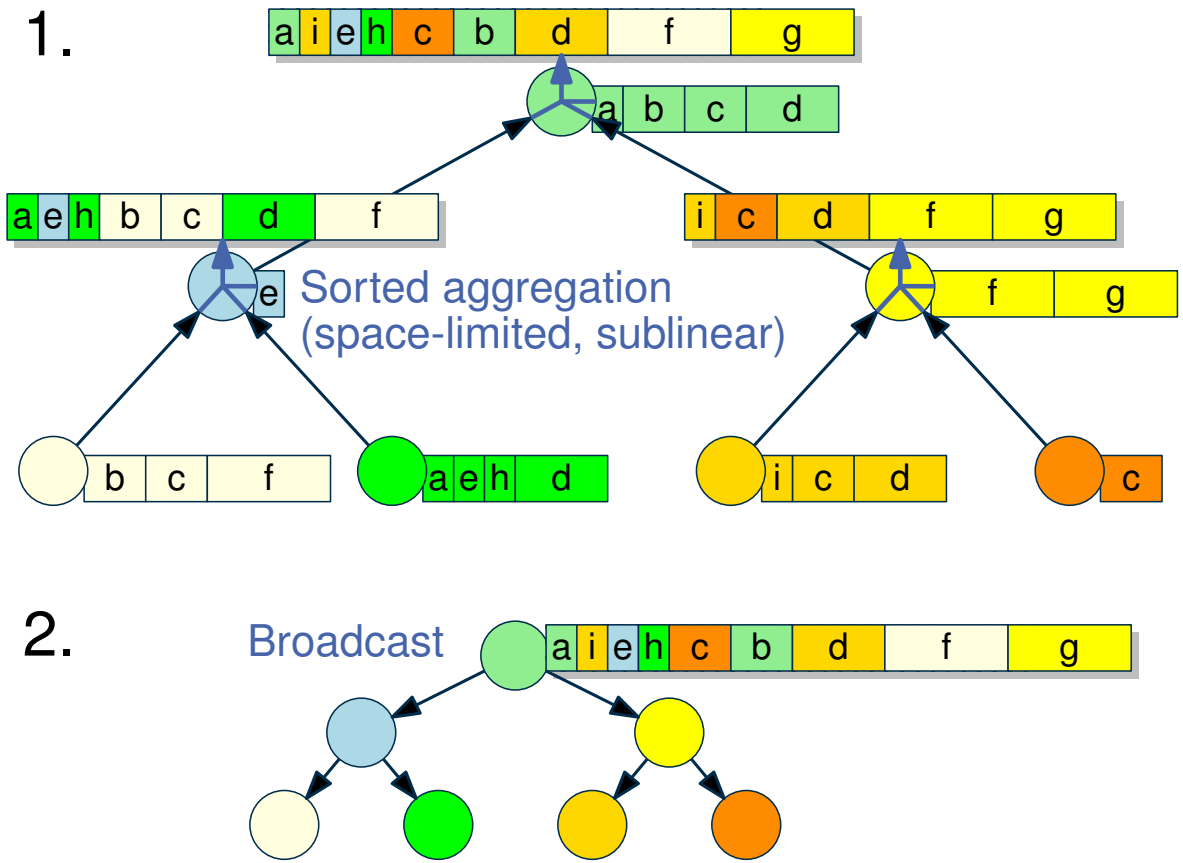
- Aggregate information along binary tree of processors
- Detect duplicates during merge
- Result is of compact shape
- Sublinear buffer size growth: Discard longest clauses as necessary



Clause Exchange in Mallob [23]

Custom collective operation

- 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



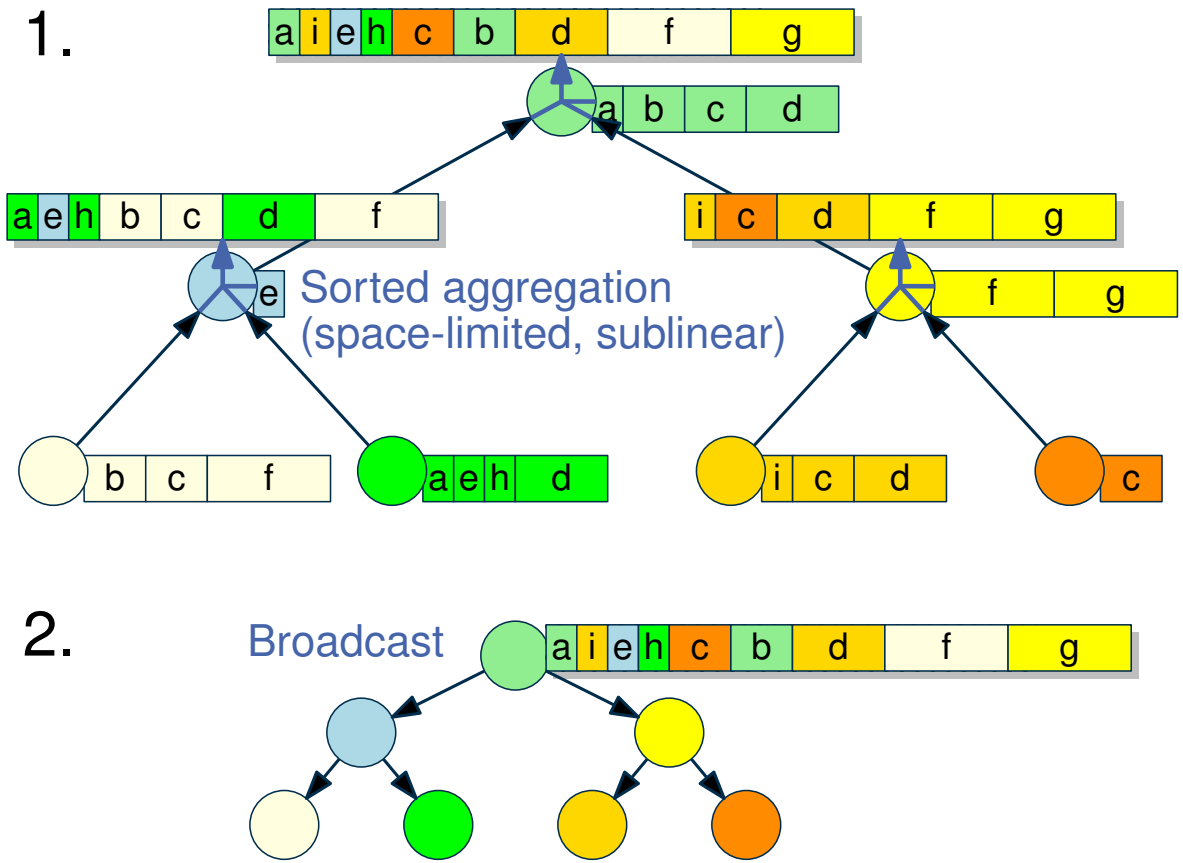
Clause Exchange in Mallob [23]

Custom collective operation

- 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



Clause Filtering

The Problem

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

Previously: [1, 24]

- 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: [1, 24]

- Bloom filters: fixed size, risk of false positives

Mallob'22+: Exact distributed filter [23]

- 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).

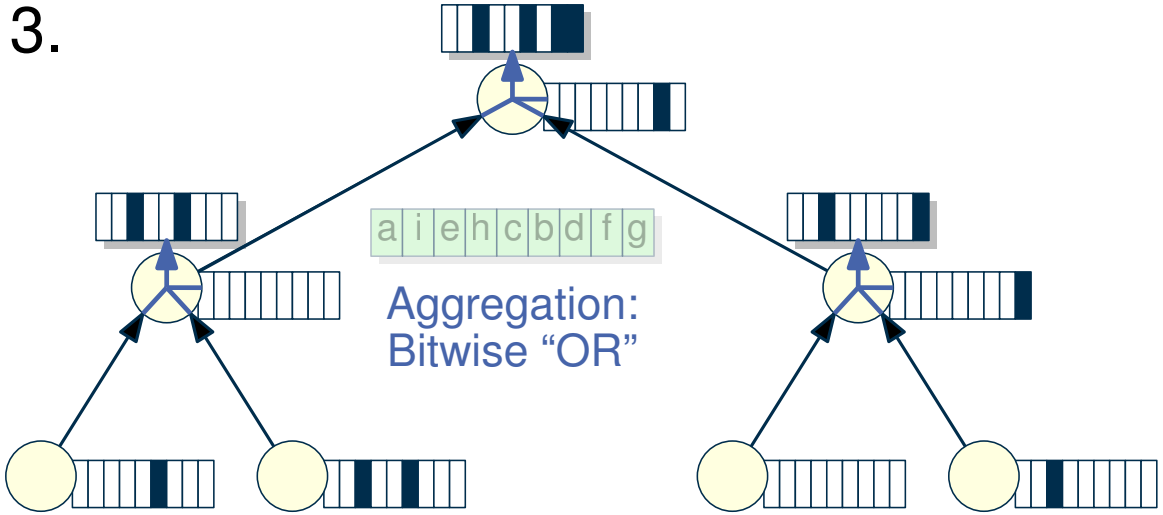
Previously: [1, 24]

- Bloom filters: fixed size, risk of false positives

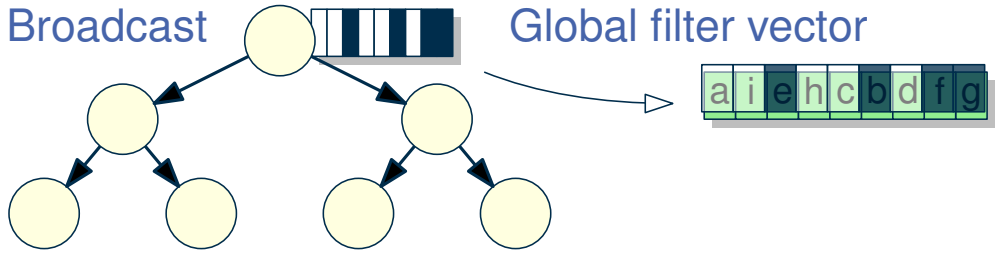
Mallob'22+: Exact distributed filter [23]

- 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 $v[i] = \texttt{false}$

3.



4.



Clause Filtering

The Problem

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

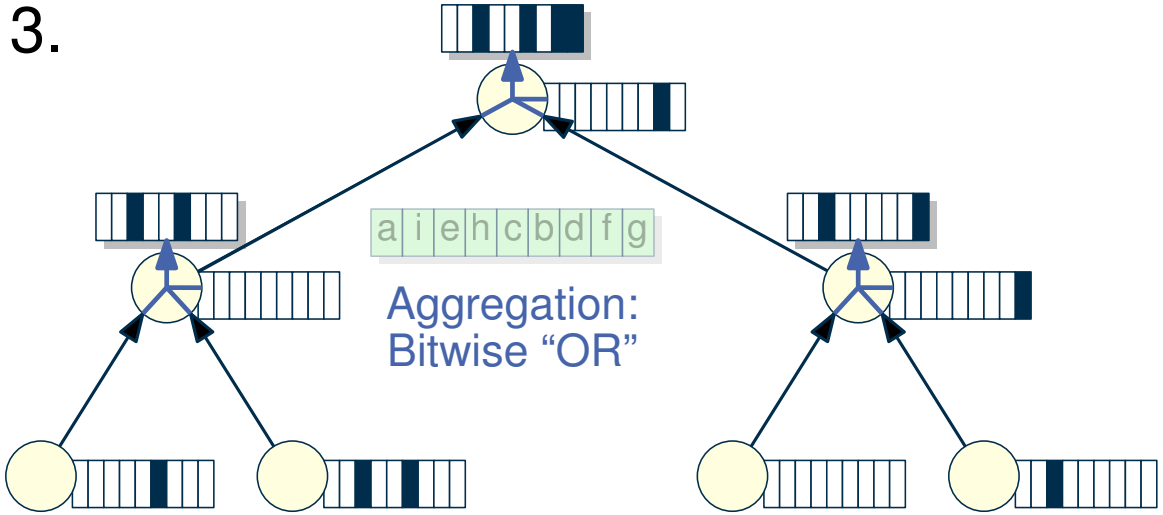
Previously: [1, 24]

- Bloom filters: fixed size, risk of false positives

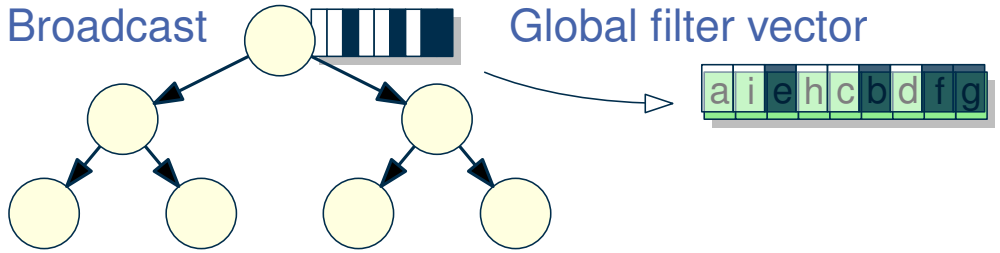
Mallob'22+: Exact distributed filter [23]

- 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 $v[i] = \texttt{false}$
- Use producer data to prevent mirroring clauses back to their producer(s)

3.



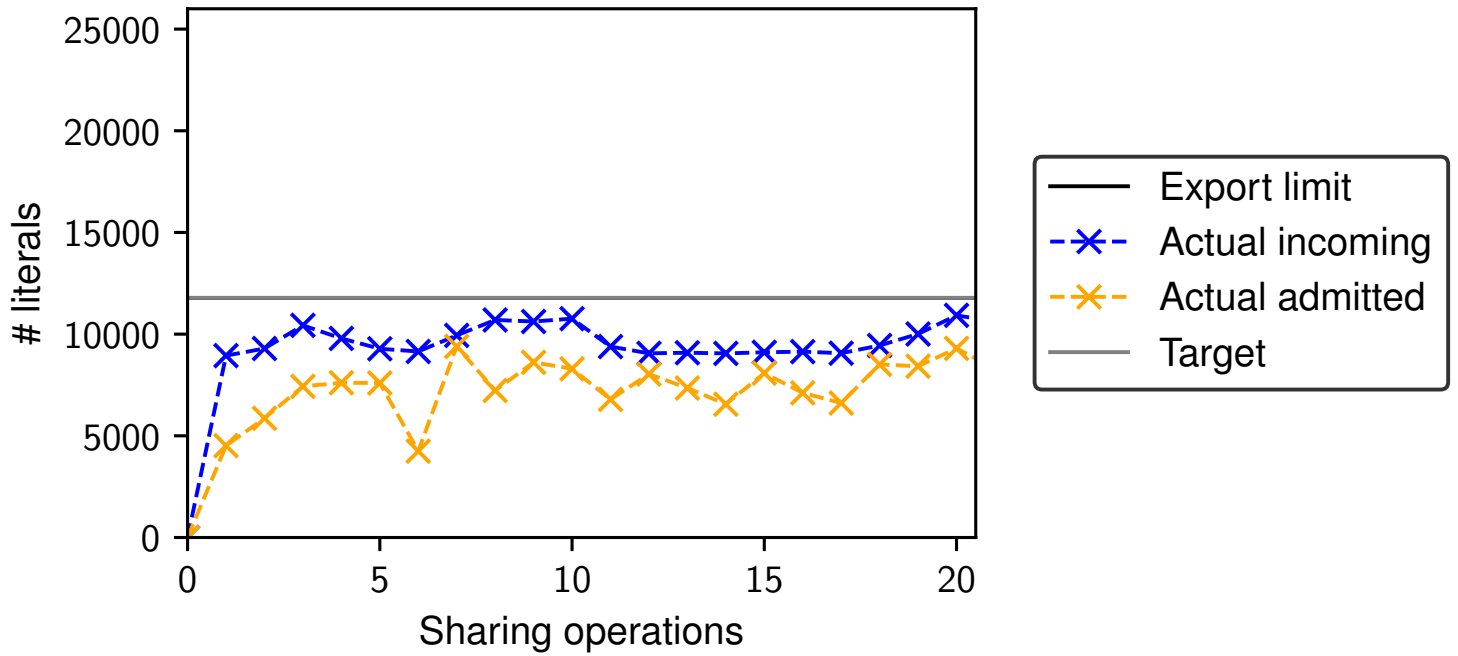
4.



Enforcing a Sharing Volume [23]

We want to share L literals per sharing but may only get $L' < L$ successfully shared literals. Why?

1. Processes didn't produce, export enough clauses
2. Duplicate clauses were detected and eliminated during aggregation
3. Distributed filter blocked some of the transmitted clauses

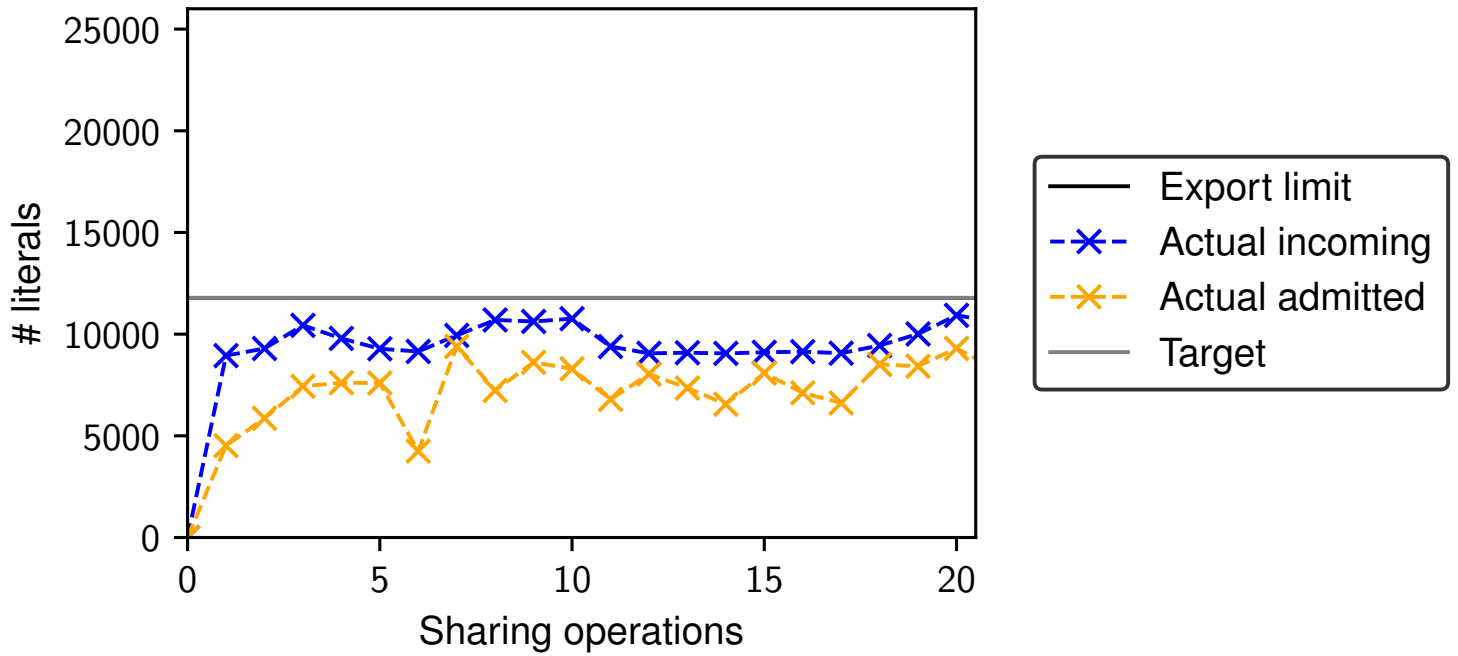


Enforcing a Sharing Volume [23]

We want to share L literals per sharing but may only get $L' < L$ successfully shared literals. Why?

1. Processes didn't produce, export enough clauses
2. Duplicate clauses were detected and eliminated during aggregation
3. Distributed filter blocked some of the transmitted clauses

Fix: Elastic compensation for sharing volume unused for algorithmic reasons (2. , 3.)

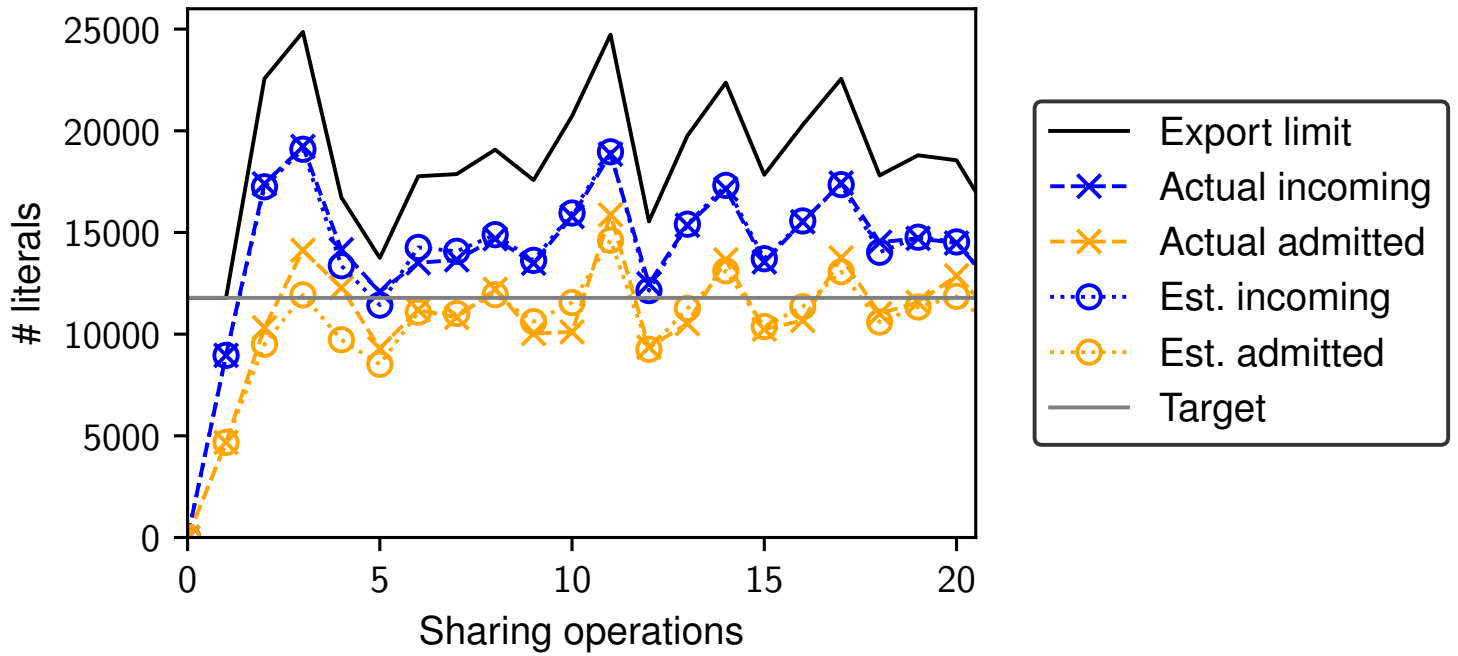


Enforcing a Sharing Volume [23]

We want to share L literals per sharing but may only get $L' < L$ successfully shared literals. Why?

1. Processes didn't produce, export enough clauses
2. Duplicate clauses were detected and eliminated during aggregation
3. Distributed filter blocked some of the transmitted clauses

Fix: Elastic compensation for sharing volume unused for algorithmic reasons (2. , 3.)



Handling LBD Values

- Seq. solving: central metric for whether to keep a clause
- **But:** LBD found by solver A not necessarily meaningful for solver B! $\rightarrow$ not as “global” as clause length

Handling LBD Values

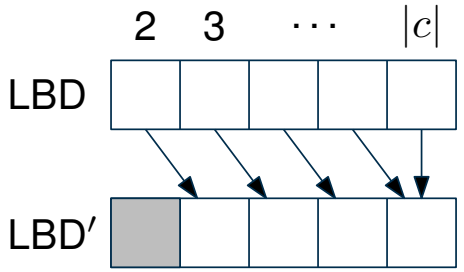
- Seq. solving: central metric for whether to keep a clause
- **But:** LBD found by solver A not necessarily meaningful for solver B! $\rightarrow$ not as “global” as clause length
- Some solvers keep clauses with LBD 2 indefinitely
  — but expect a single solver’s clause volume!
  $\Rightarrow$ Growing overhead (time, space) from low-LBD clauses

Handling LBD Values

- Seq. solving: central metric for whether to keep a clause
- **But:** LBD found by solver A not necessarily meaningful for solver B! $\rightarrow$ not as "global" as clause length
- Some solvers keep clauses with LBD 2 indefinitely
  — but expect a single solver's clause volume!
  $\Rightarrow$ Growing overhead (time, space) from low-LBD clauses

Possible approach: Increment each LBD before import [23]

- Maintains LBD-based prioritization of clauses
- Solver keeps more control over its LBD-2-clauses



	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 $\times$ 349 instances $\times$ 300 s

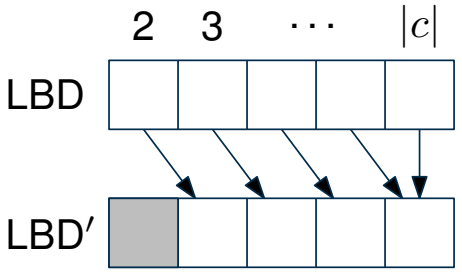
Handling LBD Values

- Seq. solving: central metric for whether to keep a clause
- **But:** LBD found by solver A not necessarily meaningful for solver B! $\rightarrow$ not as "global" as clause length
- Some solvers keep clauses with LBD 2 indefinitely
  — but expect a single solver's clause volume!
  $\Rightarrow$ Growing overhead (time, space) from low-LBD clauses

Possible approach: Increment each LBD before import [23]

- Maintains LBD-based prioritization of clauses
- Solver keeps more control over its LBD-2-clauses

But: Dropping LBD values alltogether performs just as well in latest experiments [22]



	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 $\times$ 349 instances $\times$ 300 s

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$?

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative
- **Geometric mean** $S_{geo} = \sqrt[|\mathcal{I}|]{\left(\prod_{i \in \mathcal{I}} s(i)\right)}$ – Meaningful, conservative, only for running times > 0

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative
- **Geometric mean** $S_{geo} = \sqrt[|\mathcal{I}|]{\left(\prod_{i \in \mathcal{I}} s(i)\right)}$ – Meaningful, conservative, only for running times > 0
- **Total** $S_{tot} = \frac{\sum_{i \in \mathcal{I}} T_{seq}(i)}{\sum_{i \in \mathcal{I}} T_{par}(i)}$ – Meaningful (“ratio of time saved”), emphasizes difficult inputs

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative
- **Geometric mean** $S_{geo} = \sqrt[|\mathcal{I}|]{\left(\prod_{i \in \mathcal{I}} s(i)\right)}$ – Meaningful, conservative, only for running times > 0
- **Total** $S_{tot} = \frac{\sum_{i \in \mathcal{I}} T_{seq}(i)}{\sum_{i \in \mathcal{I}} T_{par}(i)}$ – Meaningful (“ratio of time saved”), emphasizes difficult inputs

How do we handle instances with $T_{seq} =$ TIMEOUT?

Parallel Solver Evaluation Methodology

Benchmark “best” sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative
- **Geometric mean** $S_{geo} = \sqrt[|\mathcal{I}|]{\left(\prod_{i \in \mathcal{I}} s(i)\right)}$ – Meaningful, conservative, only for running times > 0
- **Total** $S_{tot} = \frac{\sum_{i \in \mathcal{I}} T_{seq}(i)}{\sum_{i \in \mathcal{I}} T_{par}(i)}$ – Meaningful (“ratio of time saved”), emphasizes difficult inputs

How do we handle instances with $T_{seq} =$ TIMEOUT?

- Reduce such occurrences by running sequential solver (much) longer than parallel solver

Parallel Solver Evaluation Methodology

Benchmark "best" sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative
- **Geometric mean** $S_{geo} = \sqrt[|\mathcal{I}|]{\left(\prod_{i \in \mathcal{I}} s(i)\right)}$ – Meaningful, conservative, only for running times > 0
- **Total** $S_{tot} = \frac{\sum_{i \in \mathcal{I}} T_{seq}(i)}{\sum_{i \in \mathcal{I}} T_{par}(i)}$ – Meaningful ("ratio of time saved"), emphasizes difficult inputs

How do we handle instances with $T_{seq} =$ TIMEOUT?

- Reduce such occurrences by running sequential solver (much) longer than parallel solver
- "Generously" assume as solved in time T, or apply penalty $k \cdot T$ – makes speedups difficult to interpret

Parallel Solver Evaluation Methodology

Benchmark "best" sequential solver (e.g., SAT Competition winner) and parallel solver on recent competition instances, with a fixed time limit T per instance.

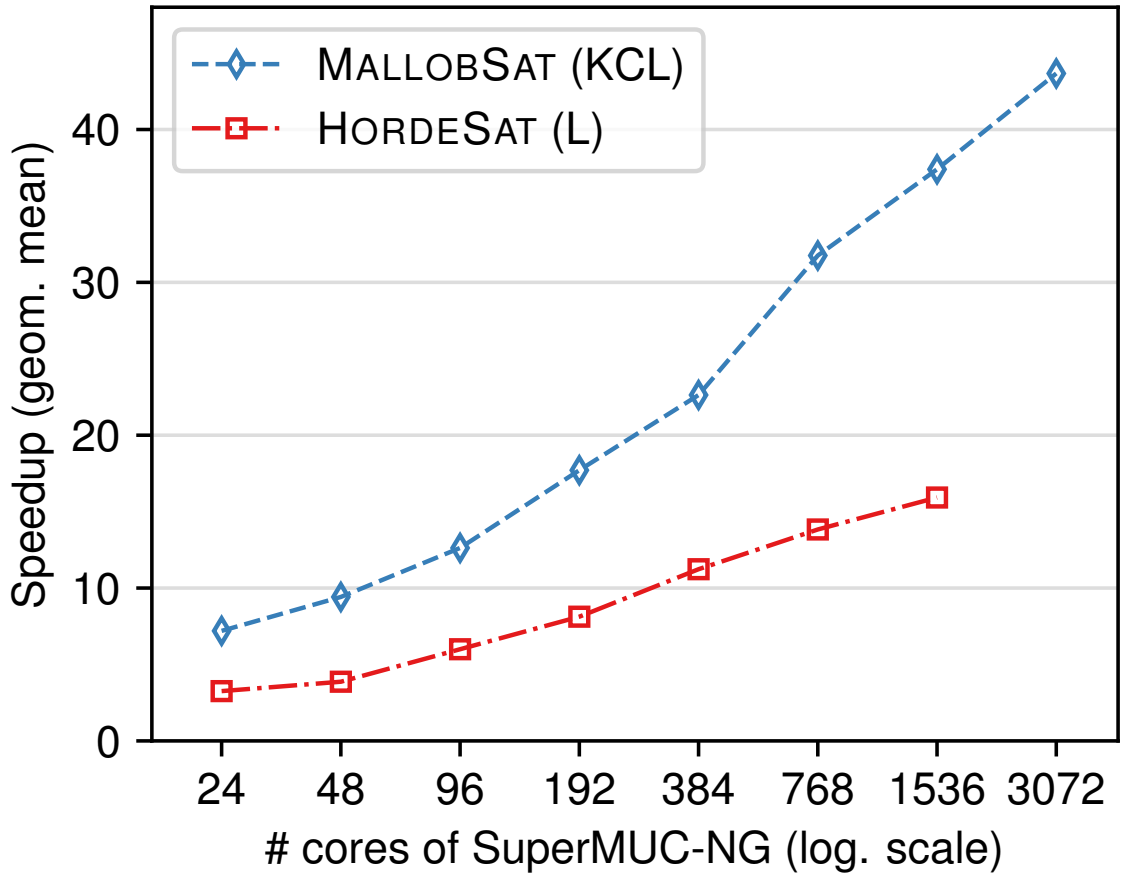
Speedup for a single instance i is $s(i) := T_{seq}(i)/T_{par}(i)$. How do we compute overall speedups over inputs $\mathcal{I}$?

- Arithmetic mean $S_{arit} = 1/|\mathcal{I}| \cdot \sum_{i \in \mathcal{I}} s(i)$? No statistical meaning!
- **Median** S_{med}: central item in sorted list of speedups – Meaningful, very conservative
- **Geometric mean** $S_{geo} = \sqrt[|\mathcal{I}|]{\left(\prod_{i \in \mathcal{I}} s(i)\right)}$ – Meaningful, conservative, only for running times > 0
- **Total** $S_{tot} = \frac{\sum_{i \in \mathcal{I}} T_{seq}(i)}{\sum_{i \in \mathcal{I}} T_{par}(i)}$ – Meaningful ("ratio of time saved"), emphasizes difficult inputs

How do we handle instances with $T_{seq} =$ TIMEOUT?

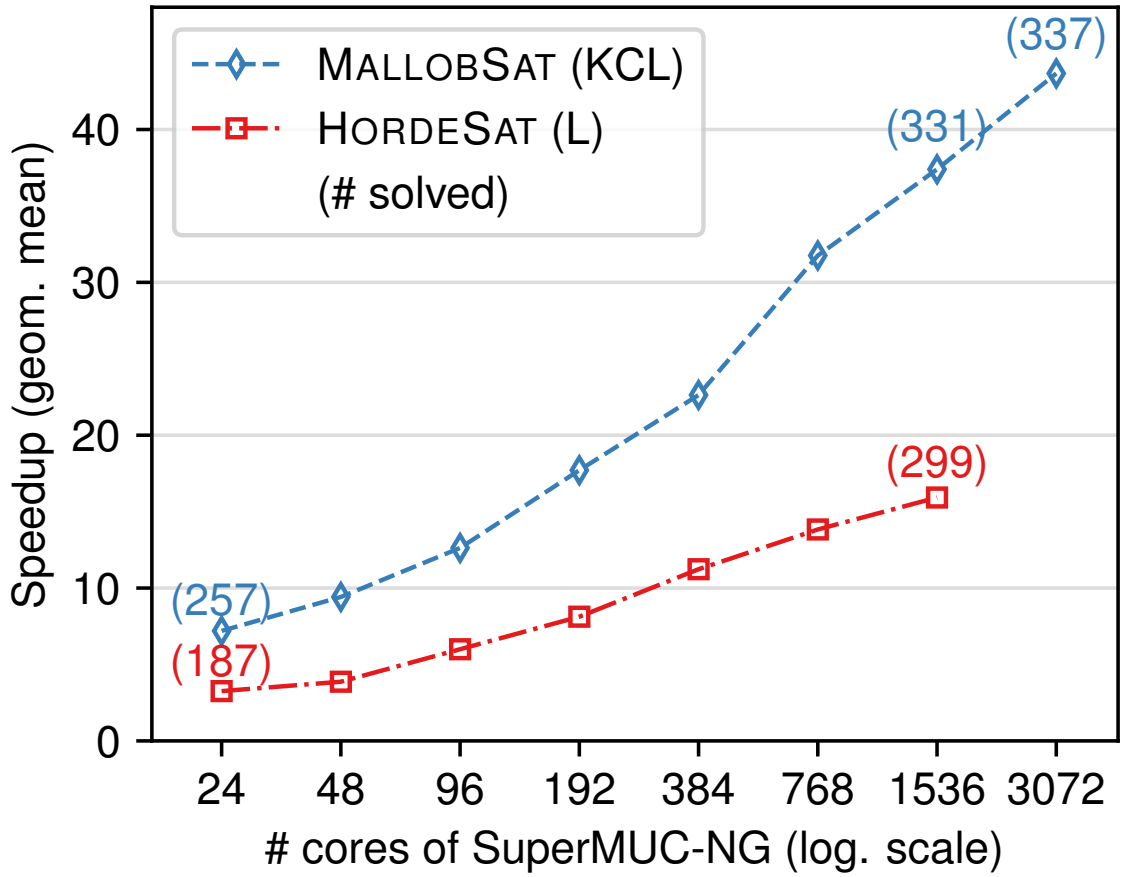
- Reduce such occurrences by running sequential solver (much) longer than parallel solver
- "Generously" assume as solved in time T, or apply penalty $k \cdot T$ – makes speedups difficult to interpret
- Omit from speedup calculation – clean separation of speedups from # solved instances

Scaling of MallobSat [23]



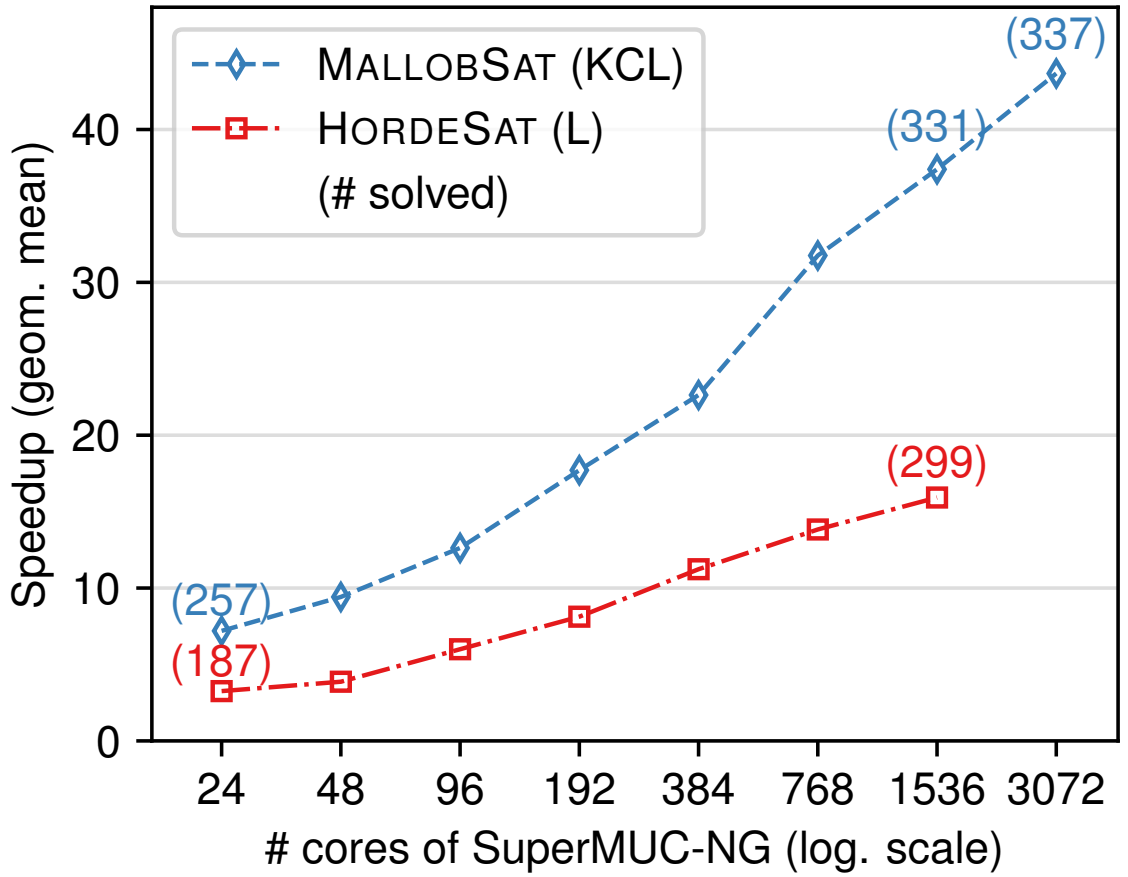
400 problems from SAT Comp. 2021 · Seq. baseline KissAT_MAB-HyWalk · Seq. limit 32 h (331 solved) · Par. limit 300 s

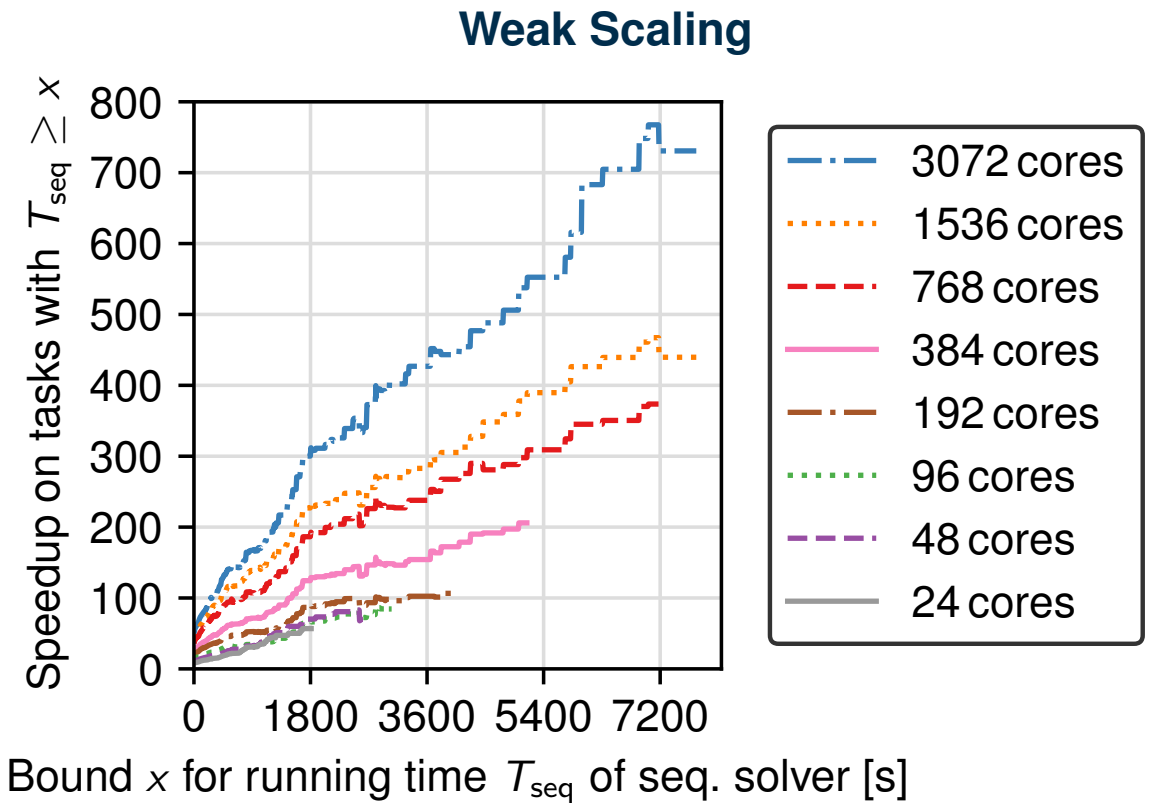
Scaling of MallobSat [23]



400 problems from SAT Comp. 2021 · Seq. baseline KissAT_MAB-HyWalk · Seq. limit 32 h (331 solved) · Par. limit 300 s

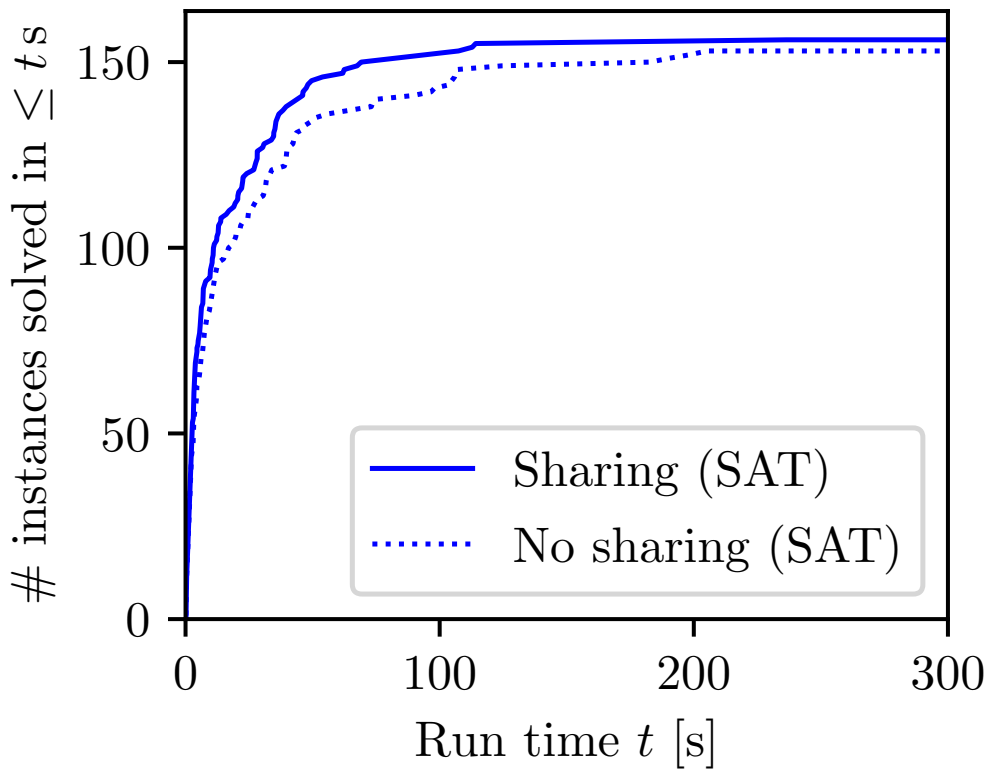
Scaling of MallobSat [23]

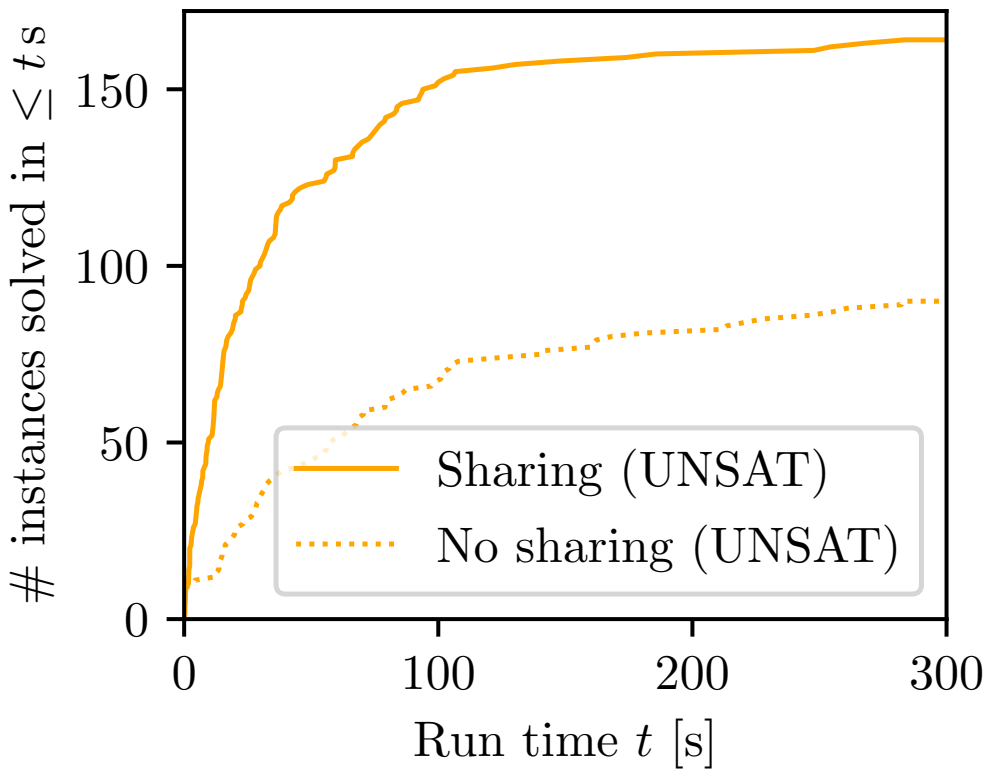




400 problems from SAT Comp. 2021 · Seq. baseline KissAT_MAB-HyWalk · Seq. limit 32 h (331 solved) · Par. limit 300 s

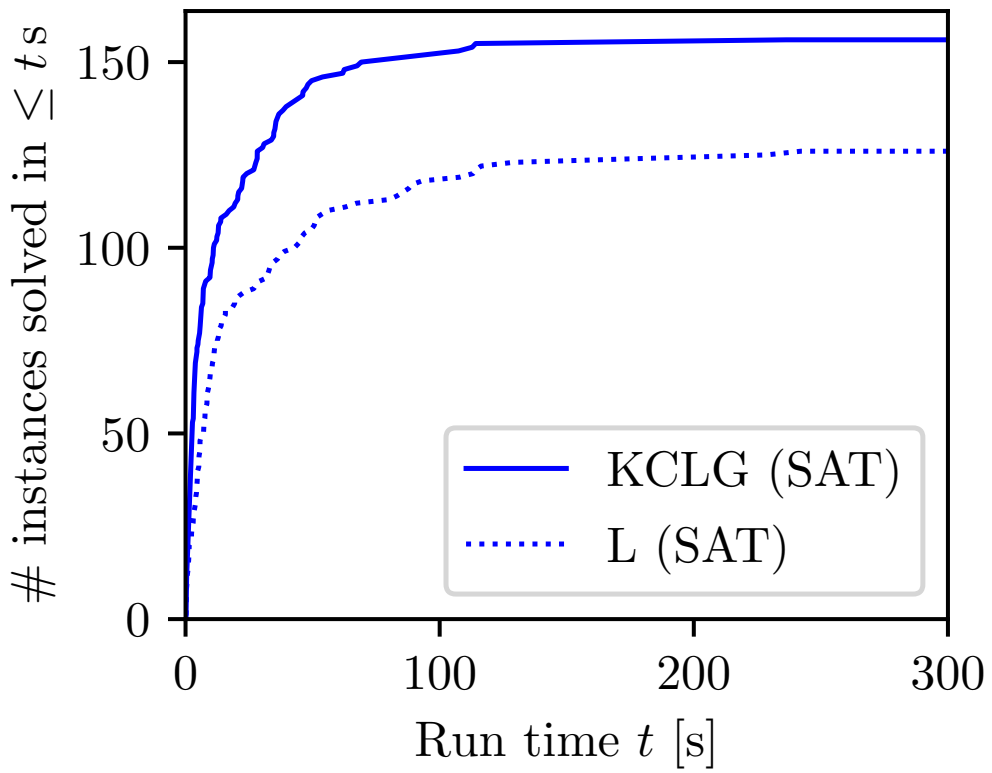
Merit of Clause Sharing, SAT vs. UNSAT

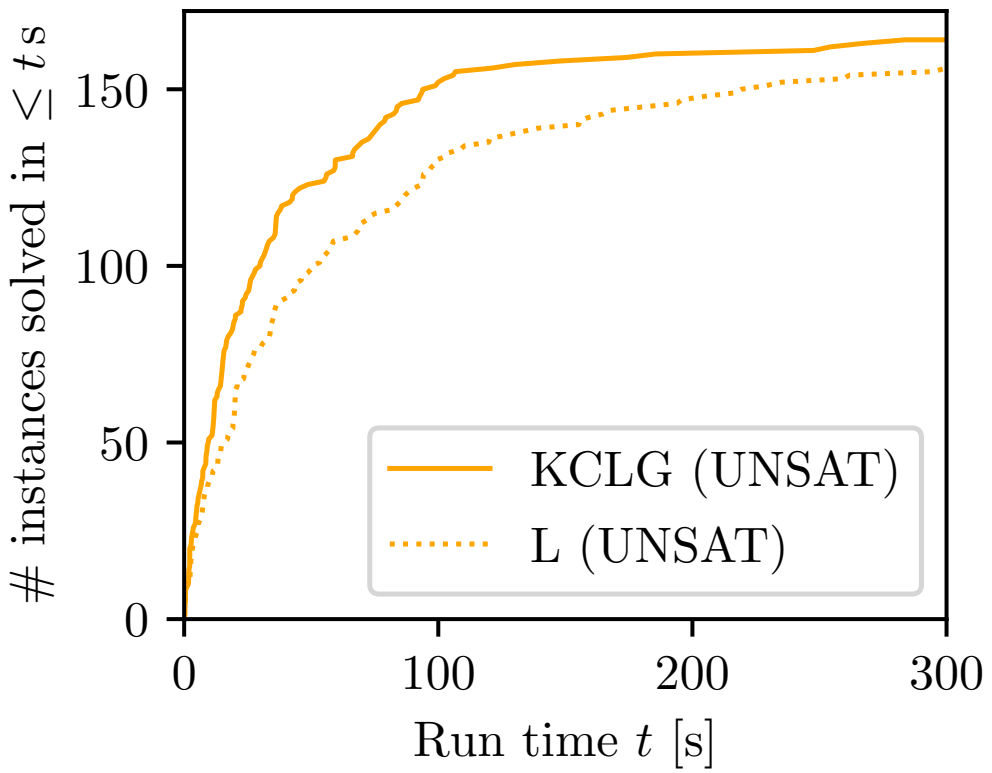




768 cores $\times$ 349 “solvable” instances from ISC 2022 $\times$ 300 s, portfolio “KCLG”

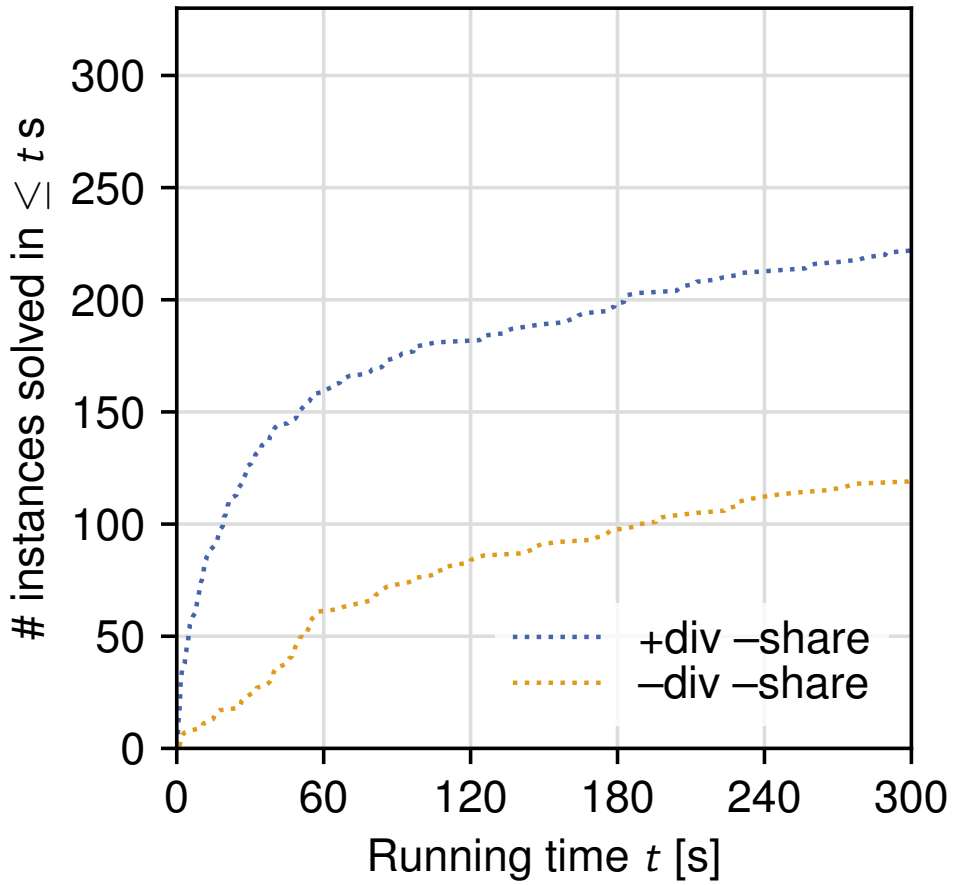
Merit of Diverse Portfolio, SAT vs. UNSAT





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

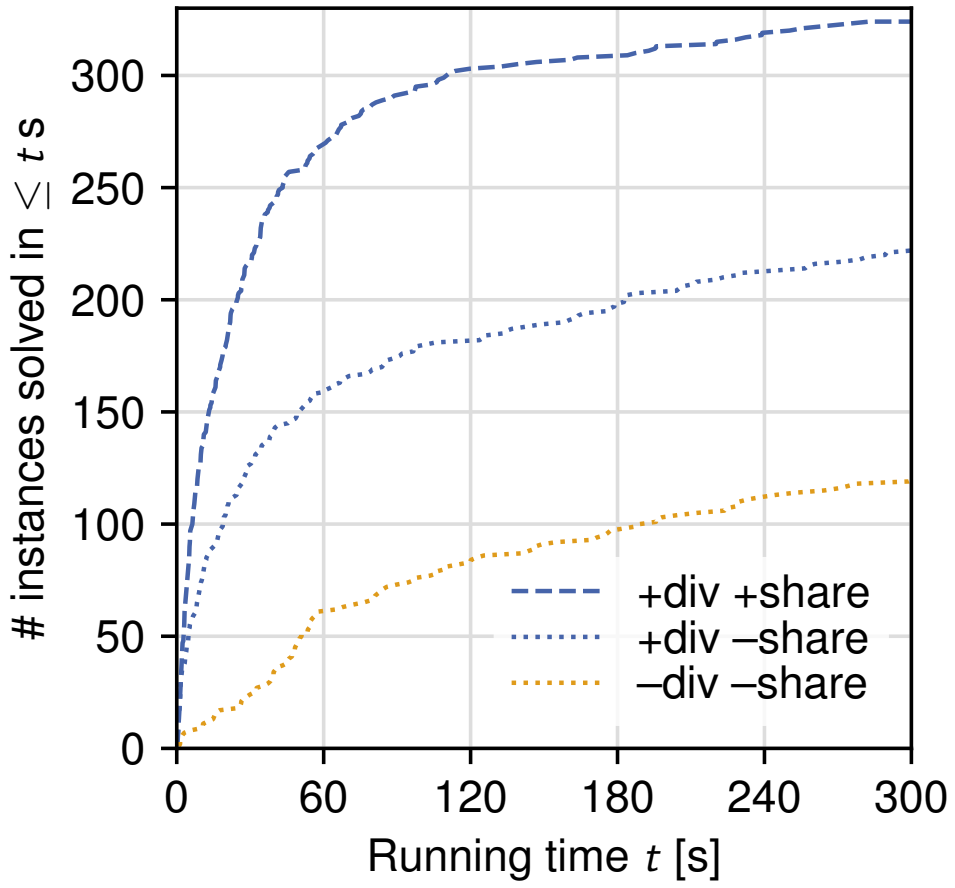
Diversification vs. Sharing @ 768 Cores [23]



- Without clause sharing, diversification is highly effective

349 problems from SAT Comp. 2022 · CaDiCaL only

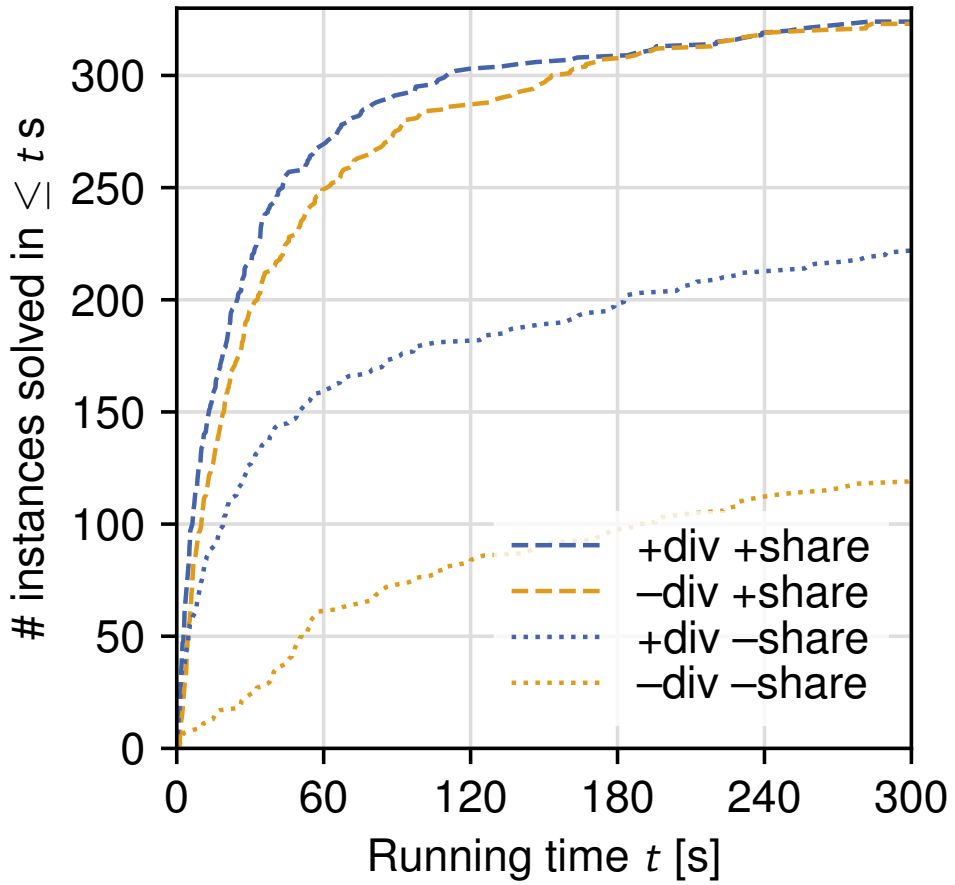
Diversification vs. Sharing @ 768 Cores [23]



- Without clause sharing, diversification is highly effective
- With sharing:

349 problems from SAT Comp. 2022 · CaDiCaL only

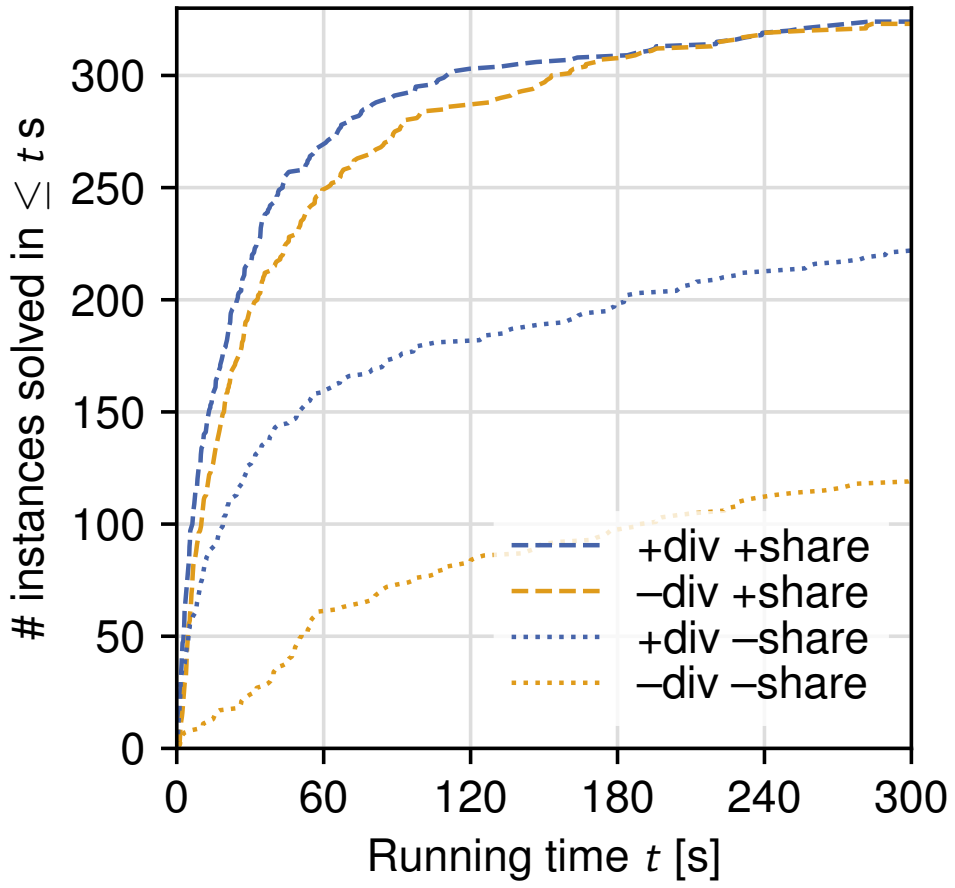
Diversification vs. Sharing @ 768 Cores [23]



- Without clause sharing, diversification is highly effective
- With sharing: 768× the same program performs well?!

349 problems from SAT Comp. 2022 · CaDiCaL only

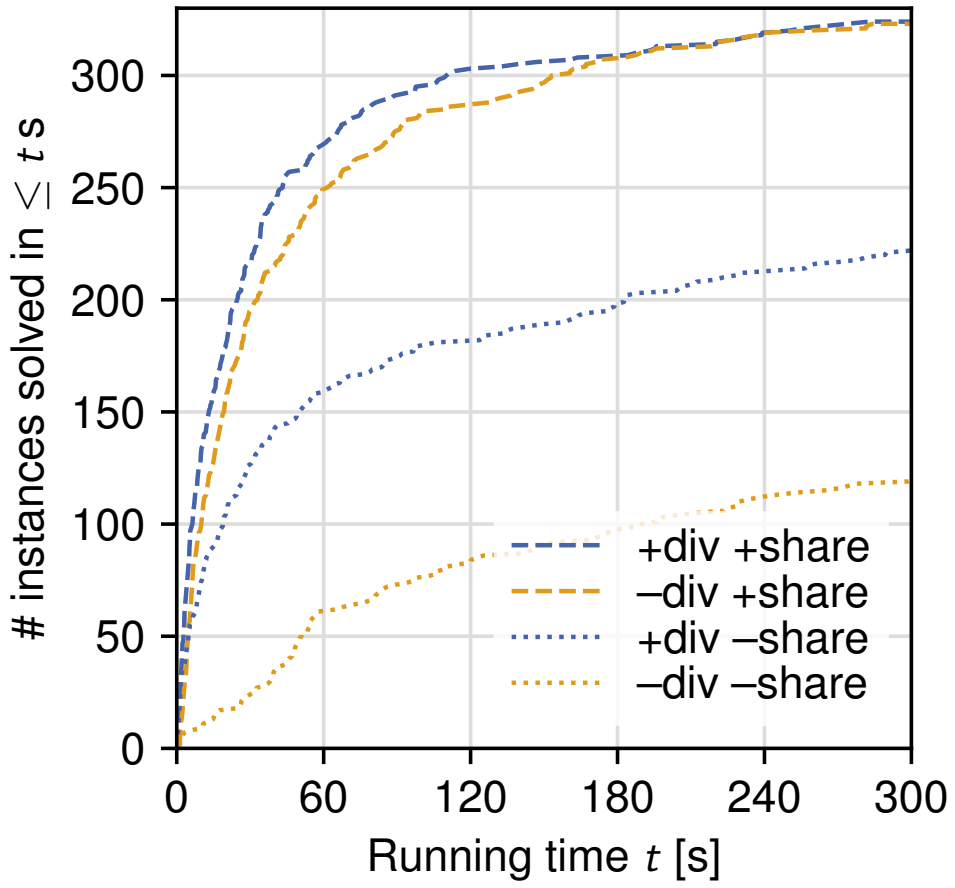
Diversification vs. Sharing @ 768 Cores [23]



- Without clause sharing, diversification is highly effective
- With sharing: 768× the same program performs well?!
  - ⇒ Clause imports deviate due to parallel execution
  - ⇒ "Butterfly effect" → effective sharing!

349 problems from SAT Comp. 2022 · CaDiCaL only

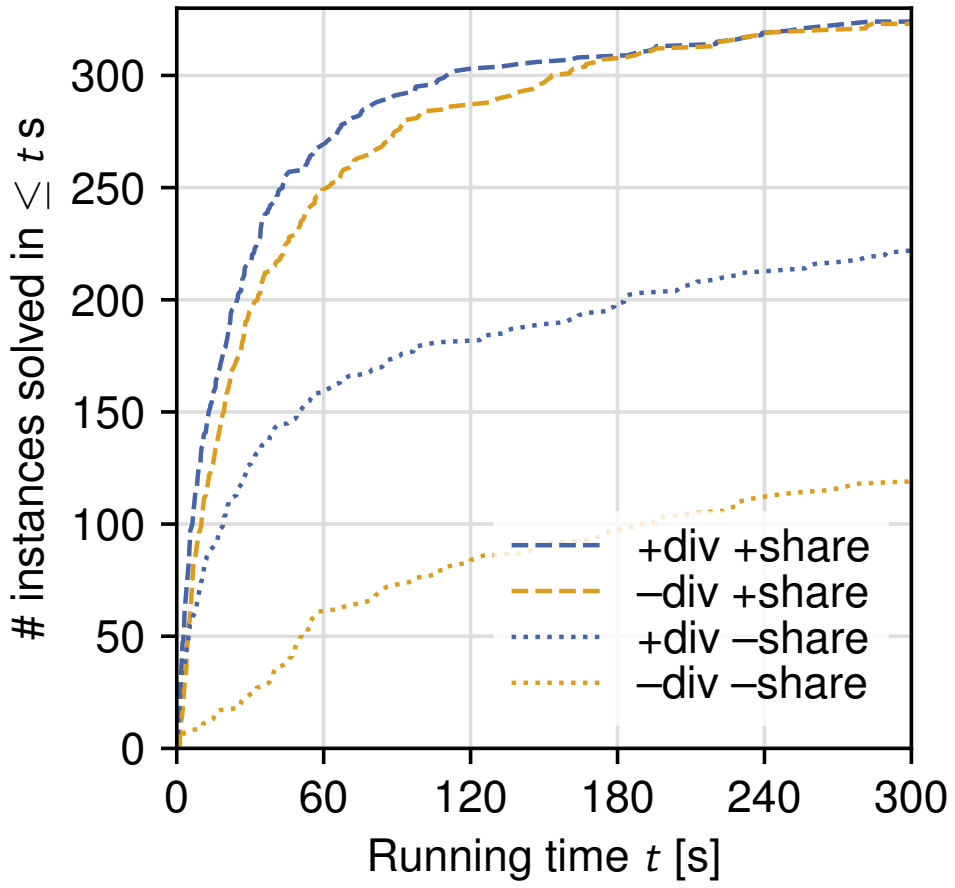
Diversification vs. Sharing @ 768 Cores [23]



349 problems from SAT Comp. 2022 · CaDiCaL only

- Without clause sharing, diversification is highly effective
- With sharing: 768× the same program performs well?!
  - ⇒ Clause imports deviate due to parallel execution
  - ⇒ "Butterfly effect" → effective sharing!
- Similar findings @ 3072 cores
  - Default CaDiCaL with primitive diversification (seeds, phases) performs competitively
  - Fully diversified portfolio without clause sharing does not

Diversification vs. Sharing @ 768 Cores [23]



349 problems from SAT Comp. 2022 · CaDiCaL only

- Without clause sharing, diversification is highly effective
- With sharing: 768× the same program performs well?!
  - ⇒ Clause imports deviate due to parallel execution
  - ⇒ "Butterfly effect" → effective sharing!
- Similar findings @ 3072 cores
  - Default CaDiCaL with primitive diversification (seeds, phases) performs competitively
  - Fully diversified portfolio without clause sharing does not

⇒ Portfolio of diverse configurations is dispensable

⇒ Clause sharing is essential

MallobSat: A Portfolio Solver?

Prevalent concept in literature: **Portfolio solver** *with clause sharing* / Clause-sharing portfolio

- "*a problem instance is independently given to a collection of solvers competing for a solution in parallel*"
  – Fichte et al., 2023 [9]
- "*each thread runs a different SAT solver on the same instance[, which] in combination with clause-sharing leads to surprisingly good performance for small portfolio sizes*"
  – Ozdemir et al., 2021 [17]

MallobSat: A Portfolio Solver?

Prevalent concept in literature: **Portfolio solver** *with clause sharing* / Clause-sharing portfolio

- "*a problem instance is independently given to a collection of solvers competing for a solution in parallel*"
  – Fichte et al., 2023 [9]
- "*each thread runs a different SAT solver on the same instance[, which] in combination with clause-sharing leads to surprisingly good performance for small portfolio sizes*"
  – Ozdemir et al., 2021 [17]

Our view, based on empirical observations:

MallobSat is a Clause-sharing solver *with diversification*

- Clause sharing = main driver of scalability
- Adding explicit diversification is beneficial but not essential
- Applicability to other solvers?

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

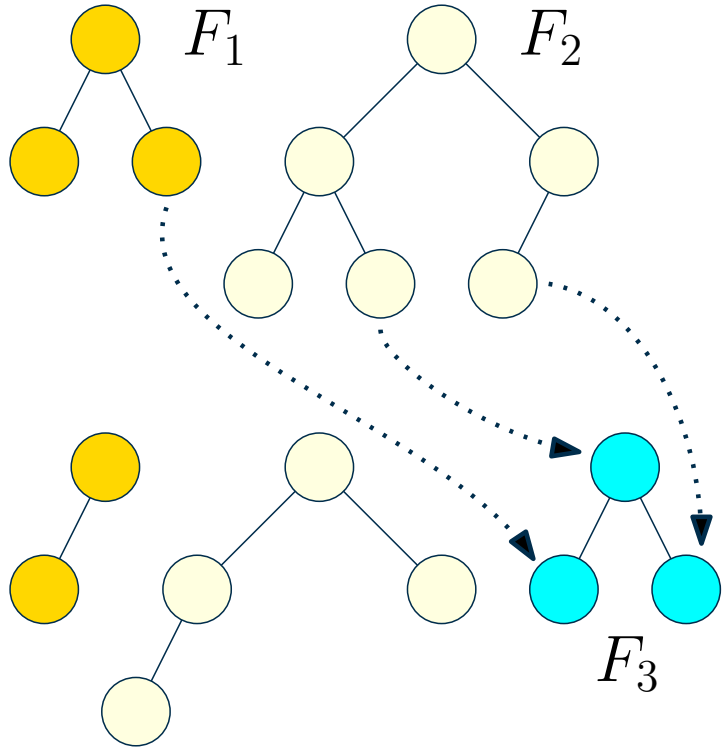
- Faster solving times
- Can resolve problems out of reach for sequential solvers
- Not that resource efficient (on average)

Solving many formulas in parallel

- Embarrassingly parallel
- Solving itself less powerful

Best of both worlds? [19]

- 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



Scheduling Many SAT Tasks [23]

Problem statement

Given $x \in \{400, 1600, 6400\}$ cores and 400 SAT tasks,
solve as many tasks as possible.

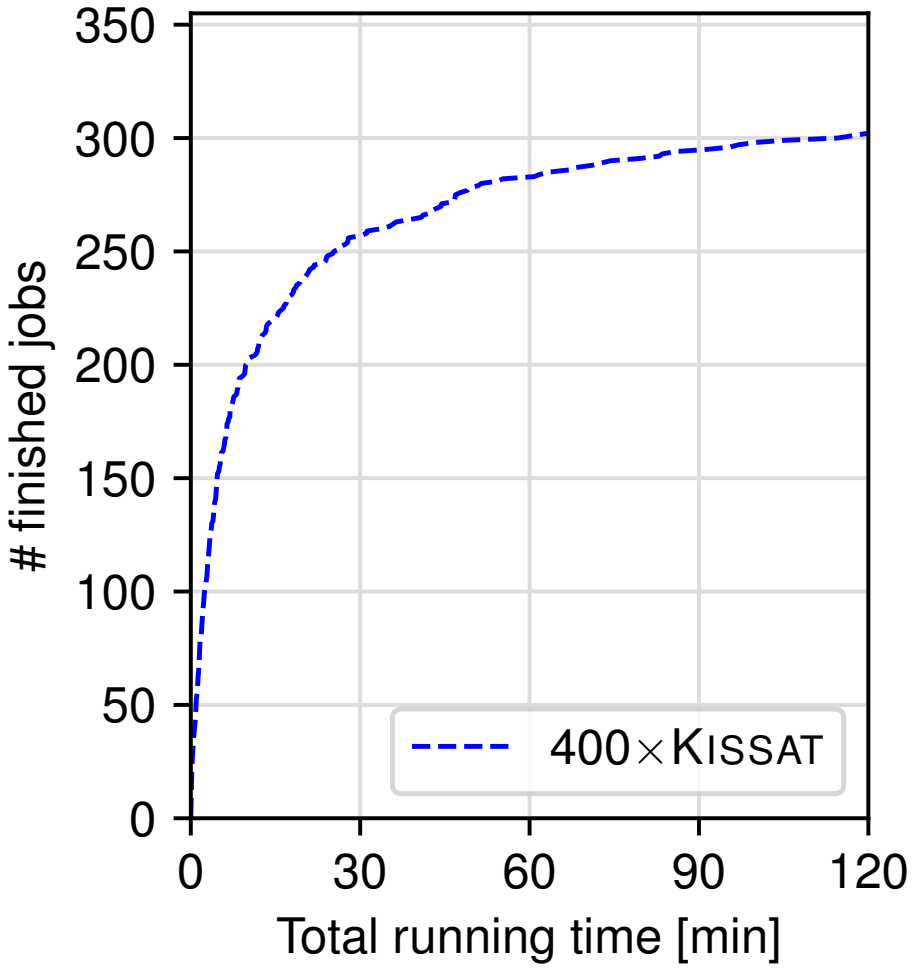
Scheduling Many SAT Tasks [23]

Problem statement

Given $x \in \{400, 1600, 6400\}$ cores and 400 SAT tasks, solve as many tasks as possible.

Extreme 1: 400 sequential solvers

- "Embarrassingly parallel" job processing
- Highly efficient (no redundant work)
- Sequential solving only $\rightarrow$ no scaling opportunity



Scheduling Many SAT Tasks [23]

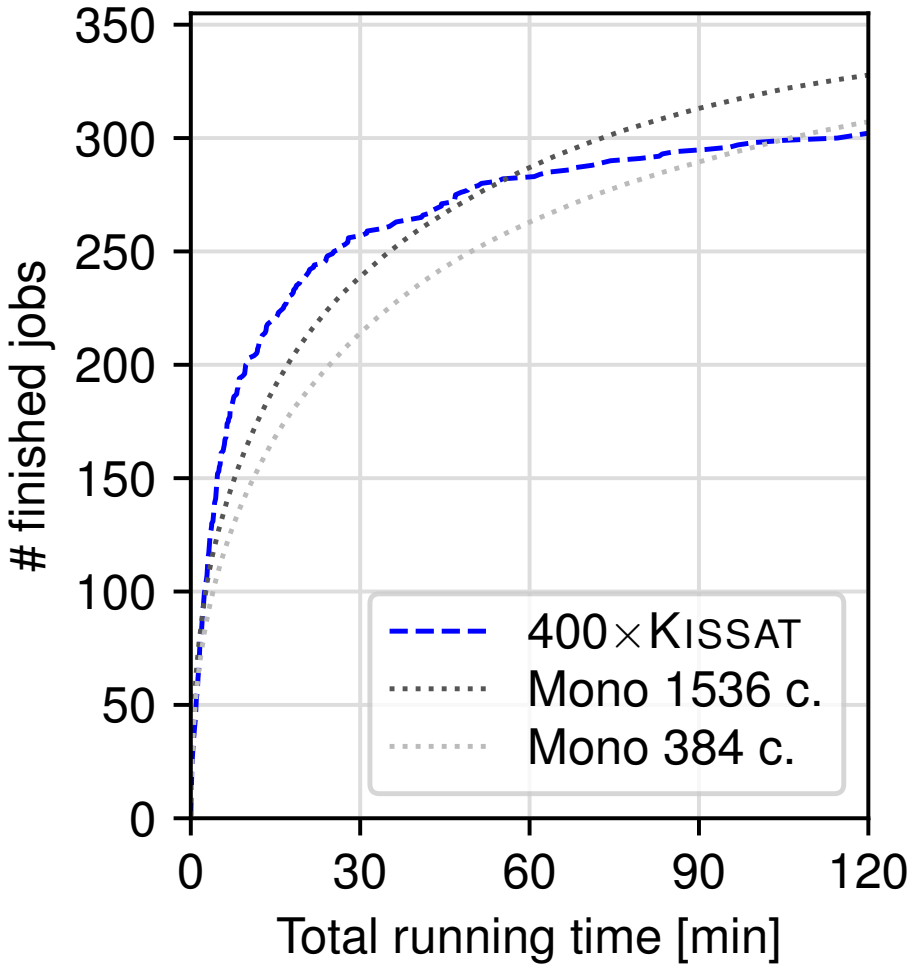
Problem statement

Given $x \in \{400, 1600, 6400\}$ cores and 400 SAT tasks, solve as many tasks as possible.

Extreme 1: 400 sequential solvers (400× Kissat)

Extreme 2: "Monolithic" parallel solving

- One job at a time
- Generous assumption: instances sorted by run time
- Maximizes per-instance speedups
- No task parallelism, poor efficiency



Scheduling Many SAT Tasks [23]

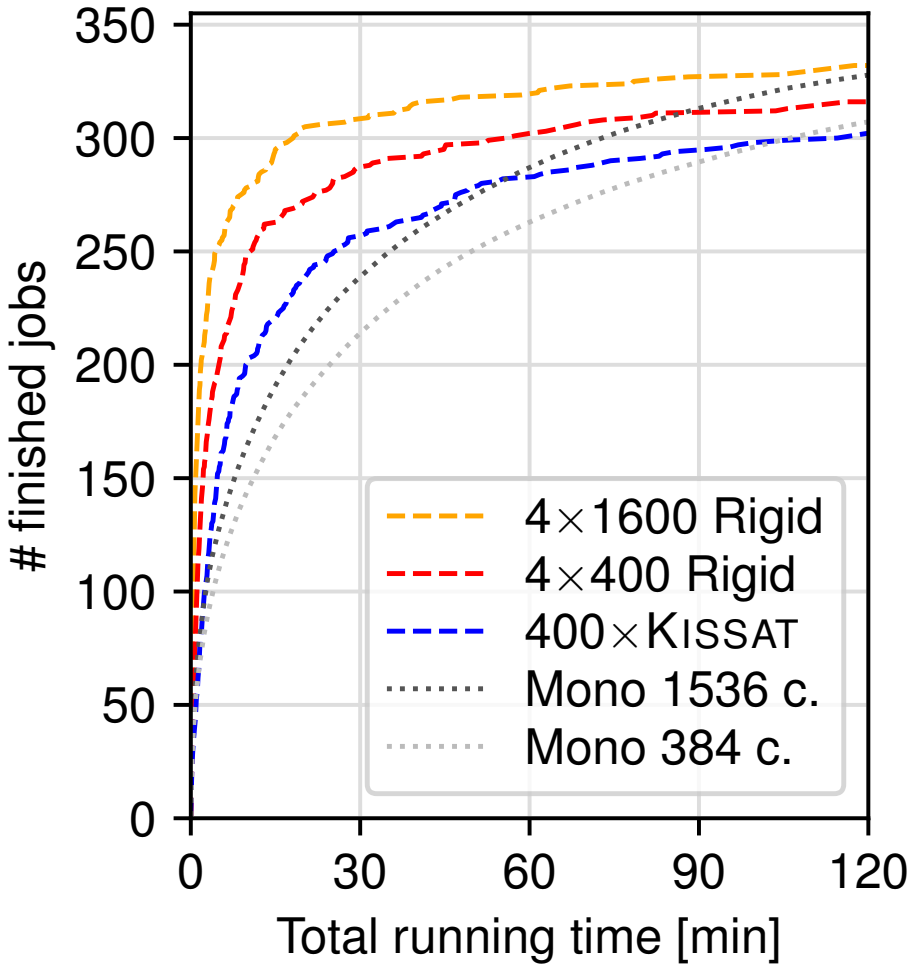
Problem statement

Given $x \in \{400, 1600, 6400\}$ cores and 400 SAT tasks, solve as many tasks as possible.

Extreme 1: 400 sequential solvers (400× Kissat)
Extreme 2: "Monolithic" parallel solving

Middle ground 1: Divide cores evenly among jobs

- Solid speedups at small-scale parallel SAT
- After 15 min, $>$ 50% of cores are idling



Scheduling Many SAT Tasks [23]

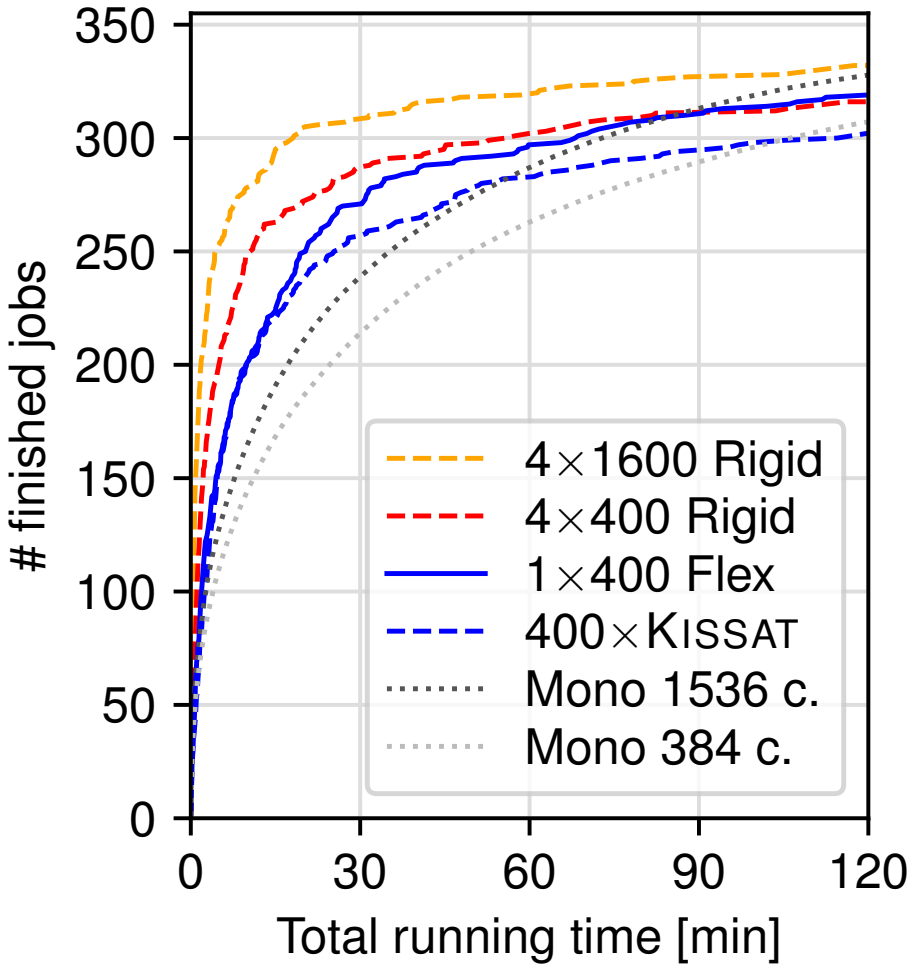
Problem statement

Given $x \in \{400, 1600, 6400\}$ cores and 400 SAT tasks, solve as many tasks as possible.

Extreme 1: 400 sequential solvers (400× Kissat)
Extreme 2: "Monolithic" parallel solving
Middle ground 1: Divide cores among jobs (rigid)

Middle ground 2: Divide cores dynamically among jobs

- Finishing jobs yield resources to remaining jobs



Scheduling Many SAT Tasks [23]

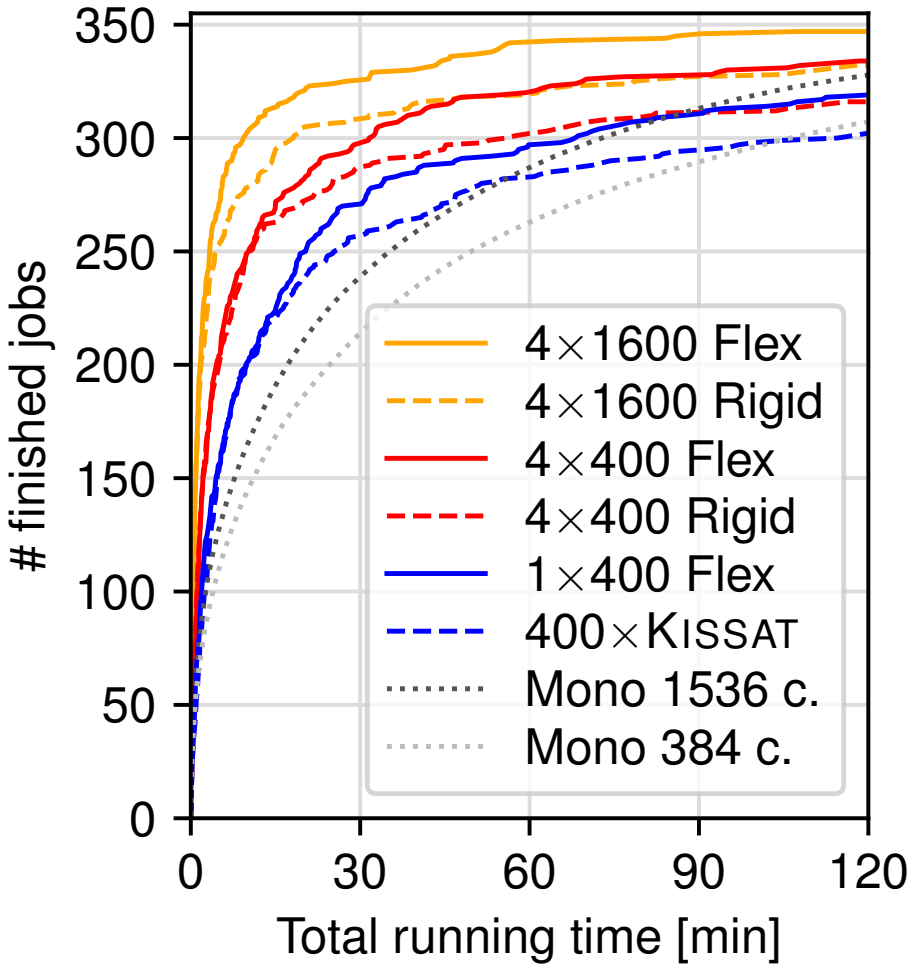
Problem statement

Given $x \in \{400, 1600, 6400\}$ cores and 400 SAT tasks, solve as many tasks as possible.

Extreme 1: 400 sequential solvers (400× Kissat)
Extreme 2: "Monolithic" parallel solving
Middle ground 1: Divide cores among jobs (rigid)

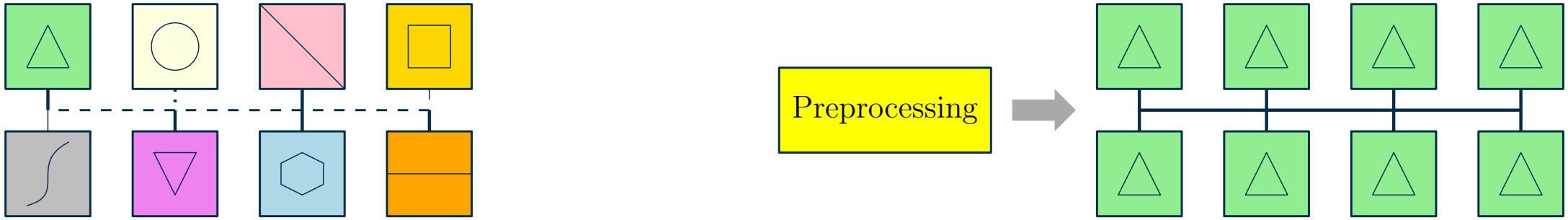
Middle ground 2: Divide cores dynamically among jobs

- Finishing jobs yield resources to remaining jobs
- Record number of solved instances using relatively little resources



Current Frontiers

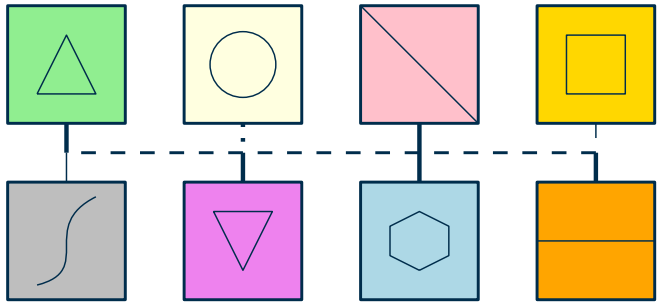
- Streamlined clause-sharing solver design, away from portfolios [22]

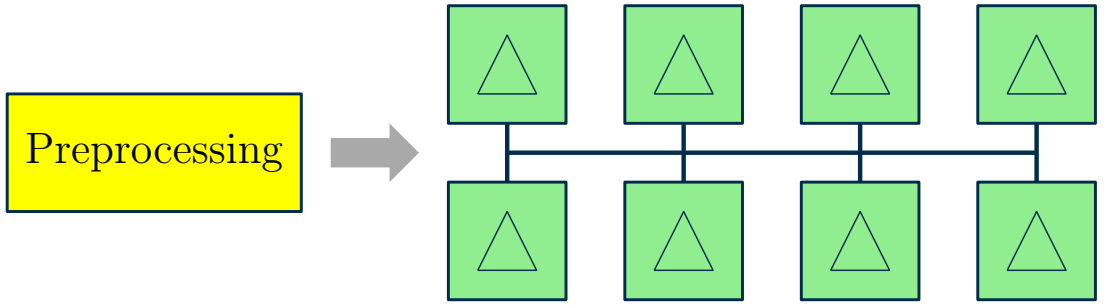


Clause sharing portfolio with “language barriers” **vs.** streamlined parallel CDCL

Current Frontiers

- Streamlined clause-sharing solver design, away from portfolios [22]



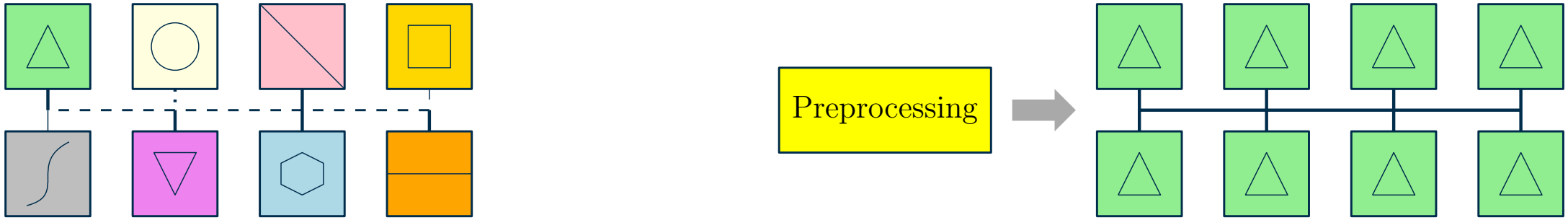


Clause sharing portfolio with “language barriers” **vs.** streamlined parallel CDCL

- Address high main memory requirements, especially for huge formulas (cf. [10])
- Preprocessing and Inprocessing in parallel SAT solving (cf. [16])

Current Frontiers

- Streamlined clause-sharing solver design, away from portfolios [22]

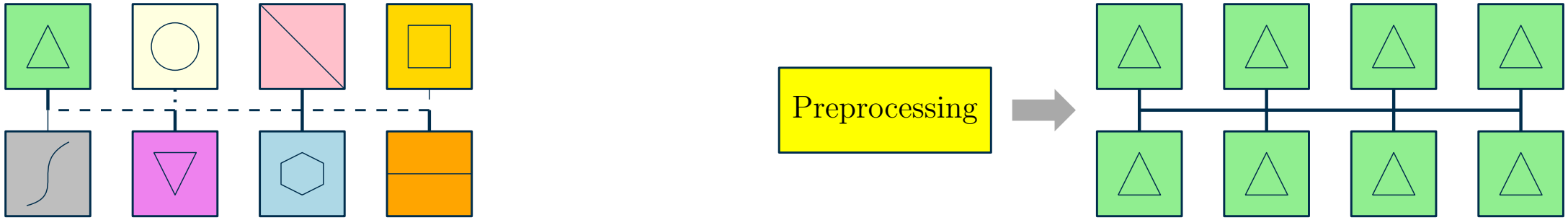


Clause sharing portfolio with “language barriers” **vs.** streamlined parallel CDCL

- Address high main memory requirements, especially for huge formulas (cf. [10])
- Preprocessing and Inprocessing in parallel SAT solving (cf. [16])
- Applying technology to related tools, applications
  - Verification (Bounded Model Checking, SMT Solving)
  - Optimization (MaxSAT Solving)
  - Crucial piece of technology: Distributed incremental SAT solving! [21]

Current Frontiers

- Streamlined clause-sharing solver design, away from portfolios [22]

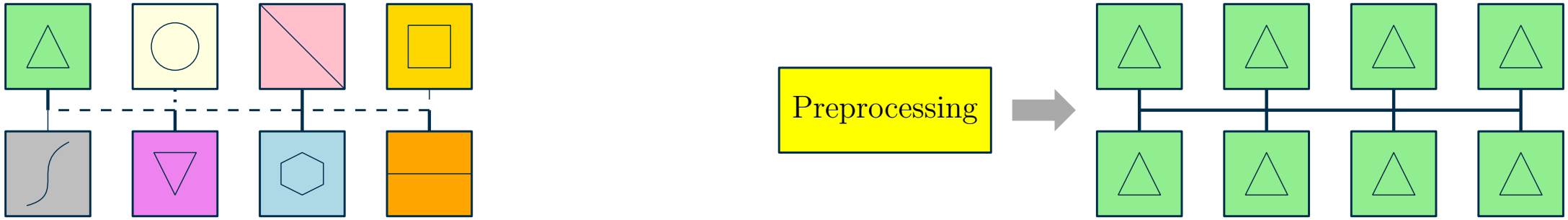


Clause sharing portfolio with “language barriers” **vs.** streamlined parallel CDCL

- Address high main memory requirements, especially for huge formulas (cf. [10])
- Preprocessing and Inprocessing in parallel SAT solving (cf. [16])
- Applying technology to related tools, applications
  – Verification (Bounded Model Checking, SMT Solving)
  – Optimization (MaxSAT Solving)
  – Crucial piece of technology: Distributed incremental SAT solving! [21]
- Uses of GPUs ?? (Stochastic Local Search [6], Inprocessing offloading [16], . . .)

Current Frontiers

- Streamlined clause-sharing solver design, away from portfolios [22]



Clause sharing portfolio with “language barriers” **vs.** streamlined parallel CDCL

- Address high main memory requirements, especially for huge formulas (cf. [10])
- Preprocessing and Inprocessing in parallel SAT solving (cf. [16])
- Applying technology to related tools, applications
  – Verification (Bounded Model Checking, SMT Solving)
  – Optimization (MaxSAT Solving)
  – Crucial piece of technology: Distributed incremental SAT solving! [21]
- Uses of GPUs ?? (Stochastic Local Search [6], Inprocessing offloading [16], . . .)
- Proofs of unsatisfiability (next lecture!)

Recap

Parallel SAT Solving

- 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
  — diversifies solvers effectively in and of itself
- Exploit embarrassingly parallel job processing for interactive solving & best efficiency

Next Up

- Proof pragmatics: Formats for different proof systems in the wild
- Bringing proof technology to parallel and distributed SAT solving

References I

[1] Tomáš Balyo, Peter Sanders und Carsten Sinz. „Hordesat: A massively parallel portfolio SAT solver“. In: *Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2015, S. 156–172. DOI: 10.1007/978-3-319-24318-4_12.

[2] Armin Biere. „Lingeling, Plingeling and Treengeling entering the SAT competition 2013“. In: *SAT Competition*. Bd. 2013. 2013, S. 1.

[3] Armin Biere. „Lingeling, Plingeling, Picosat and Precosat at SAT race 2010“. In: *SAT Competition*. 2010.

[4] Wolfgang Blochinger, Carsten Sinz und Wolfgang Küchlin. „Parallel propositional satisfiability checking with distributed dynamic learning“. In: *Parallel Computing* 29.7 (2003), S. 969–994. DOI: 10.1016/s0167-8191(03)00068-1.

[5] Max Böhm und Ewald Speckenmeyer. „A fast parallel SAT-solver – Efficient workload balancing“. In: *Annals of Mathematics and Artificial Intelligence* 17 (1996), S. 381–400. DOI: 10.1007/bf02127976.

[6] Yunuo Cen, Zhiwei Zhang und Xuanyao Fong. „Massively parallel continuous local search for hybrid SAT solving on GPUs“. In: *AAAI Conference*. Bd. 39. 11. 2025, S. 11140–11149. DOI: 10.1609/aaai.v39i11.33211.

[7] Thorsten Ehlers, Dirk Nowotka und Philipp Sieweck. „Communication in massively-parallel SAT solving“. In: *Int. Conf. on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2014, S. 709–716. DOI: 10.1109/ictai.2014.111.

[8] Yulik Feldman, Nachum Dershowitz und Ziyad Hanna. „Parallel multithreaded satisfiability solver: Design and implementation“. In: *Electronic Notes in Theoretical Computer Science* 128.3 (2005), S. 75–90. DOI: 10.1016/j.entcs.2004.10.020.

[9] Johannes K. Fichte u. a. „The Silent (R)evolution of SAT“. In: *Comm. ACM* 66.6 (2023), S. 64–72. DOI: 10.1145/3560469.

[10] Mathias Fleury und Armin Biere. „Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses“. In: *Pragmatics of SAT*. arxiv.org/pdf/2207.13577.pdf. 2022.

References II

[11] Youssef Hamadi, Said Jabbour und Lakhdar Sais. „ManySAT: a parallel SAT solver“. In: *Journal on Satisfiability, Boolean Modeling and Computation* 6.4 (2010), S. 245–262. DOI: 10.3233/sat190070.

[12] Marijn J. H. Heule. „Schur number five“. In: *AAAI Conference on Artificial Intelligence*. Bd. 32. 1. 2018. DOI: 10.1609/aaai.v32i1.12209.

[13] Marijn J. H. Heule, Oliver Kullmann und Victor Marek. „Solving and verifying the boolean pythagorean triples problem via cube-and-conquer“. In: *Proc. Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2016, S. 228–245. DOI: 10.1007/978-3-319-40970-2_15.

[14] Marijn J. H. Heule u. a. „Cube and conquer: Guiding CDCL SAT solvers by lookaheads“. In: *Proc. Haifa Verification Conference*. Springer. 2011, S. 50–65. DOI: 10.1007/978-3-642-34188-5_8.

[15] Bernard Jurkowiak, Chu Min Li und Gil Utard. „Parallelizing Satz using dynamic workload balancing“. In: *Electronic Notes in Discrete Mathematics* 9 (2001), S. 174–189. DOI: 10.1016/s1571-0653(04)00321-x.

[16] Muhammad Osama, Anton Wijs und Armin Biere. „Certified SAT solving with GPU accelerated inprocessing“. In: *Formal Methods in System Design* (2023), S. 1–40. DOI: 10.1007/s10703-023-00432-z.

[17] Alex Ozdemir, Haoze Wu und Clark Barrett. „SAT Solving in the Serverless Cloud“. In: *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2021, S. 241–245. DOI: 10.34727/2021/isbn.978-3-85448-046-4_33.

[18] Olivier Roussel. „Description of ppfolio (2011)“. In: *Proc. SAT Challenge*. 2012, S. 46.

[19] Peter Sanders und Dominik Schreiber. „Decentralized online scheduling of malleable NP-hard jobs“. In: *Parallel Processing (Euro-Par)*. 2022, S. 119–135. DOI: 10.1007/978-3-031-12597-3_8.

[20] Peter Sanders u. a. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019. DOI: 10.1007/978-3-030-25209-0.

References III

[21] Dominik Schreiber. *Distributed Incremental SAT Solving with Mallob: Report and Case Study with Hierarchical Planning*. 2025. arXiv: 2505.18836 [cs.DC]. URL: https://arxiv.org/abs/2505.18836.

[22] Dominik Schreiber, Niccolò Rigi-Luperti und Armin Biere. „Streamlining Distributed SAT Solver Design“. In: *Theory and Applications of Satisfiability Testing (SAT)*. 2025. DOI: 10.4230/LIPIcs.SAT.2025.23.

[23] Dominik Schreiber und Peter Sanders. „MallobSat: Scalable SAT Solving by Clause Sharing“. In: *Journal of Artificial Intelligence Research (JAIR)* 80 (2024), S. 1437–1495. DOI: 10.1613/jair.1.15827.

[24] Dominik Schreiber und Peter Sanders. „Scalable SAT Solving in the Cloud“. In: *Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2021, S. 518–534. DOI: 10.1007/978-3-030-80223-3_35.

[25] Sven Schulz und Wolfgang Blochinger. „Cooperate and compete! A hybrid solving strategy for task-parallel SAT solving on peer-to-peer desktop grids“. In: *Proc. Int. Conf. HPC & Simulation*. IEEE. 2010, S. 314–323.

[26] Hantao Zhang, Maria Paola Bonacina und Jieh Hsiang. „PSATO: a distributed propositional prover and its application to quasigroup problems“. In: *J. Symbolic Computation* 21.4-6 (1996), S. 543–560. DOI: 10.1006/jsco.1996.0030.