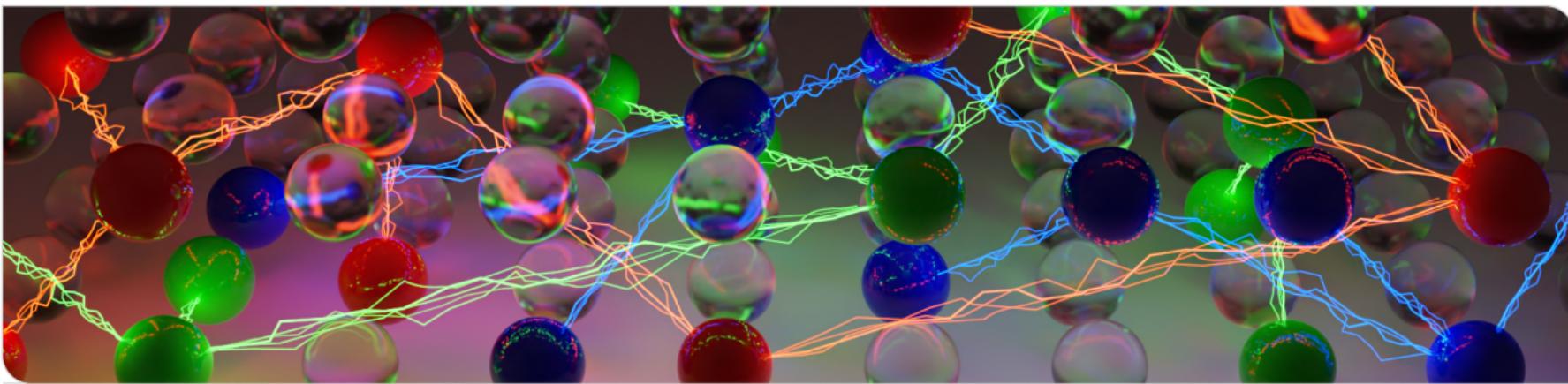


Practical SAT Solving

Lecture 13: Proofs: Pragmatics & Parallel Production

Markus Iser, Dominik Schreiber, Tomáš Balyo | July 15, 2024



Overview

Last lectures

- Background on [proof systems](#), proof complexity
- Connections to [preprocessing](#)

This lecture

- Common state-of-the-art [proof formats](#)
- Pragmatics of [proof production](#) and [checking](#)
- Producing proofs with [parallel + distributed solvers](#)
- Beyond proof files: on-the-fly checking

Back To Basics

- Solver on unsat. formula F produces sequence of clauses $P := \langle c_1, c_2, \dots, c_n \rangle$ with $c_n = \emptyset$
- Goal: **Justify** for $i = 1, \dots, n$ that $F \models c_i$, i.e., that c_i follows from F
 - actually (in practice): that $(F \cup \bigcup_{j=1}^{i-1} c_j) \models c_i$
- **Clausal Proof \mathcal{P}** : Expression of P with all information needed to justify all steps

Back To Basics

- Solver on unsat. formula F produces sequence of clauses $P := \langle c_1, c_2, \dots, c_n \rangle$ with $c_n = \emptyset$
- Goal: **Justify** for $i = 1, \dots, n$ that $F \models c_i$, i.e., that c_i follows from F
 - actually (in practice): that $(F \cup \bigcup_{j=1}^{i-1} c_j) \models c_i$
- **Clausal Proof \mathcal{P}** : Expression of P with all information needed to justify all steps **efficiently (?)**

Back To Basics

- Solver on unsat. formula F produces sequence of clauses $P := \langle c_1, c_2, \dots, c_n \rangle$ with $c_n = \emptyset$
- Goal: Justify for $i = 1, \dots, n$ that $F \models c_i$, i.e., that c_i follows from F
 - actually (in practice): that $(F \cup \bigcup_{j=1}^{i-1} c_j) \models c_i$
- **Clausal Proof \mathcal{P} :** Expression of P with all information needed to justify all steps efficiently (?)

Approach 1: Basic Clausal Proof

- **Solving:** Solver just logs each produced c_i to a file $\Rightarrow \mathcal{P} = P$
- **Checking:** Maintain clause database B initialized as $B := F$;
for each c_i , confirm that $B \models c_i$ and then $B := B \cup c_i$

Back To Basics

- Solver on unsat. formula F produces sequence of clauses $P := \langle c_1, c_2, \dots, c_n \rangle$ with $c_n = \emptyset$
- Goal: **Justify** for $i = 1, \dots, n$ that $F \models c_i$, i.e., that c_i follows from F
 - actually (in practice): that $(F \cup \bigcup_{j=1}^{i-1} c_j) \models c_i$
- **Clausal Proof \mathcal{P}** : Expression of P with all information needed to justify all steps **efficiently (?)**

Approach 1: Basic Clausal Proof

- **Solving**: Solver just logs each produced c_i to a file $\Rightarrow \mathcal{P} = P$
- **Checking**: Maintain clause database B initialized as $B := F$;
for each c_i , confirm that $B \models c_i$ and then $B := B \cup c_i$
- **How do we perform the confirmation step?**

Efficient Redundancy Checking [1]

The RUP Property

Given a clause set C and a clause c , we say that c has the Reverse Unit Propagation (RUP) property iff unit propagation on $(C \cup \{\bar{c}\})$, where $\bar{c} := \{\neg l : l \in c\}$, produces the empty clause.

- Is a clause c with RUP property w.r.t. a checker's clause set B a sound addition to B ?
 - Yes: $B \wedge \bigwedge_{l \in c} \neg l$ is unsatisfiable \rightarrow No way to satisfy B without satisfying $c \rightarrow B \models c$

Efficient Redundancy Checking [1]

The RUP Property

Given a clause set C and a clause c , we say that c has the Reverse Unit Propagation (RUP) property iff unit propagation on $(C \cup \{\bar{c}\})$, where $\bar{c} := \{\neg l : l \in c\}$, produces the **empty clause**.

- Is a clause c with RUP property w.r.t. a checker's clause set B a sound addition to B ?
 - Yes: $B \wedge \bigwedge_{l \in c} \neg l$ is unsatisfiable \rightarrow No way to satisfy B without satisfying $c \rightarrow B \models c$
- What kinds of clauses **have** the RUP property?
 - Conflict clauses from CDCL
 - Clauses arising from many pre- and inprocessing techniques
(variable elimination, subsumption, vivification, ...)
 - Actually, **all clauses** produced by out-of-the-box CADICAL

Efficient Redundancy Checking [1]

The RUP Property

Given a clause set C and a clause c , we say that c has the Reverse Unit Propagation (RUP) property iff unit propagation on $(C \cup \{\bar{c}\})$, where $\bar{c} := \{\neg l : l \in c\}$, produces the empty clause.

- Is a clause c with RUP property w.r.t. a checker's clause set B a sound addition to B ?
 - Yes: $B \wedge \bigwedge_{l \in c} \neg l$ is unsatisfiable \rightarrow No way to satisfy B without satisfying $c \rightarrow B \models c$
- What kinds of clauses have the RUP property?
 - Conflict clauses from CDCL
 - Clauses arising from many pre- and inprocessing techniques
(variable elimination, subsumption, vivification, ...)
 - Actually, all clauses produced by out-of-the-box CADICAL
- What kinds of clauses do not have the RUP property?
 - Extended Resolution steps
 - Propagation Redundancy (PR) clauses
 - ...

RUP Proof Checking

Approach 2: RUP Proof

Solving: Solver just logs each produced c_i to a file $\Rightarrow \mathcal{P} = P$.

Checking:

$B := F$

for $i = 1, \dots, n$:

 propagate $\neg l$ in B for each $l \in c_i$

if propagation in B does *not* yield the empty clause:

return **ERROR**

 undo propagations in B

$B := B \cup c_i$

return **VALIDATED**

RUP Proof Checking

Approach 2: RUP Proof

Solving: Solver just logs each produced c_i to a file $\Rightarrow \mathcal{P} = P$.

Checking:

```
B := F
for i = 1, ..., n:
    propagate  $\neg l$  in B for each  $l \in c_i$ 
    if propagation in B does not yield the empty clause:
        return ERROR
    undo propagations in B
    B := B  $\cup$   $c_i$ 
return VALIDATED
```

Checking complexity:

RUP Proof Checking

Approach 2: RUP Proof

Solving: Solver just logs each produced c_i to a file $\Rightarrow \mathcal{P} = P$.

Checking:

```
B := F
for i = 1, ..., n:
    propagate  $\neg l$  in B for each  $l \in c_i$ 
    if propagation in B does not yield the empty clause:
        return ERROR
    undo propagations in B
    B := B  $\cup$   $c_i$ 
return VALIDATED
```

Checking complexity: $\mathcal{O}(|B|)$ per step $\Rightarrow \mathcal{O}(|P|^2)$ for $|P| \gg |F|$

Checking space usage:

RUP Proof Checking

Approach 2: RUP Proof

Solving: Solver just logs each produced c_i to a file $\Rightarrow \mathcal{P} = P$.

Checking:

```
B := F
for i = 1, ..., n:
    propagate  $\neg l$  in B for each  $l \in c_i$ 
    if propagation in B does not yield the empty clause:
        return ERROR
    undo propagations in B
    B := B  $\cup$   $c_i$ 
return VALIDATED
```

Checking complexity: $\mathcal{O}(|B|)$ per step $\Rightarrow \mathcal{O}(|P|^2)$ for $|P| \gg |F|$

Checking space usage: $\mathcal{O}(|F| + |P|)$

How to improve on both?

From RUP to DRUP [2]

Actually, a solver also **deletes** clauses. \Rightarrow Put deletion information in the proof!

- $\mathcal{P} = (o_1, \dots, o_N)$ where $o_i = (op_i, c_i)$
 - $op_i \in \{\text{add}, \text{delete}\}$
 - delete: c_i is a clause added by some $o_j, j < i$, and **not deleted** by any $o_k, j < k < i$
 - **multi-set semantics** possible where a clause may be added (+ deleted) multiple times

From RUP to DRUP [2]

Actually, a solver also **deletes** clauses. \Rightarrow Put deletion information in the proof!

- $\mathcal{P} = (o_1, \dots, o_N)$ where $o_i = (op_i, c_i)$
- $op_i \in \{\text{add}, \text{delete}\}$
- delete: c_i is a clause added by some $o_j, j < i$, and **not deleted** by any $o_k, j < k < i$
- **multi-set semantics** possible where a clause may be added (+ deleted) multiple times

Formula:

$$\begin{aligned}
 & x_1 \vee \neg x_2 \\
 \wedge \quad & x_2 \vee \neg x_4 \\
 \wedge \quad & x_1 \vee x_2 \vee x_4 \\
 \wedge \quad & \neg x_1 \vee \neg x_3 \\
 \wedge \quad & x_1 \vee \neg x_3 \\
 \wedge \quad & \neg x_1 \vee x_3 \\
 \wedge \quad & x_1 \vee x_3 \vee \neg x_4 \\
 \wedge \quad & x_1 \vee x_3 \vee x_4
 \end{aligned}$$

Proof:

$$\begin{aligned}
 & \text{add } \neg x_3 \\
 & \text{add } x_1 \vee x_2 \\
 & \text{add } \neg x_1 \\
 & \text{del } \neg x_3 \\
 & \text{add } x_2 \vee x_3 \vee \neg x_4 \\
 & \text{add } x_1 \vee x_2 \vee x_3 \\
 & \text{add } \emptyset
 \end{aligned}$$

Approach 3: DRUP (Deletion RUP) Proof

```
B := F
for i = 1, ..., N:
    if opi = delete:
        B := B \ ci
        continue
    propagate  $\neg l$  in B for each l ∈ ci
    ...
    // continue as in RUP Proof
```

Correctness:

Approach 3: DRUP (Deletion RUP) Proof

```
B := F
for i = 1, ..., N:
    if opi = delete:
        B := B \ ci
        continue
    propagate  $\neg l$  in B for each l ∈ ci
    ...
    // continue as in RUP Proof
```

Correctness: deleting clauses only makes a clause set more satisfiable ✓

Complexity:

DRUP

Approach 3: DRUP (Deletion RUP) Proof

```
B := F
for i = 1, ..., N:
    if opi = delete:
        B := B \ ci
        continue
    propagate  $\neg l$  in B for each l ∈ ci
    ... // continue as in RUP Proof
```

Correctness: deleting clauses only makes a clause set more satisfiable ✓

Complexity: $\mathcal{O}(|P| \times M)$ where M is the max. volume of present clauses during solving

Space usage:

Approach 3: DRUP (Deletion RUP) Proof

```
B := F
for i = 1, ..., N:
    if opi = delete:
        B := B \ ci
        continue
    propagate  $\neg l$  in B for each l ∈ ci
    ... // continue as in RUP Proof
```

Correctness: deleting clauses only makes a clause set more satisfiable ✓

Complexity: $\mathcal{O}(|P| \times M)$ where M is the max. volume of present clauses during solving

Space usage: $\mathcal{O}(M)$ ⇒ “fits into RAM if solving fits into RAM”

Can we further improve running time?

From DRUP to LRUP [3]

Idea: Enrich proof to accelerate unit propagation (UP) of \bar{c}_i through B

- $\mathcal{P} = (o_1, \dots, o_N)$ where $o_i = (\text{add}, id_i, c_i, d_i)$ or $o_i = (\text{delete}, id_i)$
 - $id_i \in \mathbb{N}^+$, $d_i = \langle d_{i1}, \dots, d_{ik_i} \rangle$ where $d_{ij} \in \mathbb{N}^+$, $k_i \in \mathbb{N}^+$
- With d_i , solver references earlier clauses which UP needs to look at to arrive at the empty clause
 - for $1 \leq j < k_i$, clause # d_{ij} (i.e., the clause referred to by d_{ij}) must break down into a unit
 - clause # d_{ik_i} must break down into \emptyset

From DRUP to LRUP [3]

Idea: Enrich proof to accelerate unit propagation (UP) of \bar{c}_i through B

- $\mathcal{P} = (o_1, \dots, o_N)$ where $o_i = (\text{add}, id_i, c_i, d_i)$ or $o_i = (\text{delete}, id_i)$
 - $id_i \in \mathbb{N}^+$, $d_i = \langle d_{i1}, \dots, d_{ik_i} \rangle$ where $d_{ij} \in \mathbb{N}^+$, $k_i \in \mathbb{N}^+$
- With d_i , solver references earlier clauses which UP needs to look at to arrive at the empty clause
 - for $1 \leq j < k_i$, clause # d_{ij} (i.e., the clause referred to by d_{ij}) must break down into a unit
 - clause # d_{ik_i} must break down into \emptyset

Formula:

- (1) $x_1 \vee \neg x_2$
- (2) $x_2 \vee \neg x_4$
- (3) $x_1 \vee x_2 \vee x_4$
- (4) $\neg x_1 \vee \neg x_3$
- (5) $x_1 \vee \neg x_3$
- (6) $\neg x_1 \vee x_3$
- (7) $x_1 \vee x_3 \vee \neg x_4$
- (8) $x_1 \vee x_3 \vee x_4$

DRUP Proof:

- add $\neg x_3$
- add $x_1 \vee x_2$
- add $\neg x_1$
- del $\neg x_3$
- add $x_2 \vee x_3 \vee \neg x_4$
- add $x_1 \vee x_2 \vee x_3$
- add \emptyset

LRUP Proof:

- add (9) $\neg x_3$ (5, 4)
- add (10) $x_1 \vee x_2$ (3, 2)
- add (11) $\neg x_1$ (6, 9)
- del (9)
- add (12) $x_2 \vee x_3 \vee \neg x_4$ (7, 11)
- add (13) $x_1 \vee x_2 \vee x_3$ (8, 12)
- add (14) \emptyset (11, 10, 1)

LRUP

Approach 4: LRUP (Linear RUP) Proof Checking

```

 $B := F$ 
for  $i = 1, \dots, N$ :
  if  $op_i = \text{delete}$ :
     $B := B \setminus \{\# id_i\}$  // delete clause referred to by  $id_i$ 
    continue
   $U := \{\bar{l} : l \in c_i\}$ 
  for  $j = 1, \dots, k - 1$ :
    assert: clause #  $d_{ij}$  under  $U$  becomes a unit clause  $\{u\}$  // returns ERROR upon failure
     $U := U \cup \{u\}$ 
    assert: clause #  $d_{ik_j}$  under  $U$  becomes the empty clause // returns ERROR upon failure
   $B := B \cup \{c_i\}$  // confirmed:  $B \cup \{\bar{c}_i\} \models \emptyset$ 
return VALIDATED
  
```

Approach 4: LRUP (Linear RUP) Proof Checking

```

 $B := F$ 
for  $i = 1, \dots, N$ :
  if  $op_i = \text{delete}$ :
     $B := B \setminus \{\# id_i\}$  // delete clause referred to by  $id_i$ 
    continue
   $U := \{\bar{l} : l \in c_i\}$ 
  for  $j = 1, \dots, k - 1$ :
    assert: clause #  $d_{ij}$  under  $U$  becomes a unit clause  $\{u\}$  // returns ERROR upon failure
     $U := U \cup \{u\}$ 
    assert: clause #  $d_{ik_j}$  under  $U$  becomes the empty clause // returns ERROR upon failure
   $B := B \cup \{c_i\}$  // confirmed:  $B \cup \{\bar{c}_i\} \models \emptyset$ 
return VALIDATED
  
```

⇒ Larger proofs but much more efficient checking (often 10× or more)

⇒ Optional: Backward search from empty clause → prune all irrelevant proof lines

From (D|L)RUP to (D|L)RAT [4]

Resolution Asymmetric Tautology (recap)

Clause c has the *Resolution Asymmetric Tautology* (RAT) property in F w.r.t. literal $x \in c$ iff every resolvent $c' \in \{c \otimes_x \tilde{c} \mid \tilde{c} \in F_{\bar{x}}\}$ has the RUP property in F .

From (D|L)RUP to (D|L)RAT [4]

Resolution Asymmetric Tautology (recap)

Clause c has the *Resolution Asymmetric Tautology* (RAT) property in F w.r.t. literal $x \in c$ iff every resolvent $c' \in \{c \otimes_x \tilde{c} \mid \tilde{c} \in F_{\bar{x}}\}$ has the RUP property in F .

“Only” requiring each clause $c_i \in P$ to have the RAT property (rather than RUP) allows for stronger proofs!

- For RAT clause c , $F \cup c$ is satisfiability-preserving to F but may be not equivalent to F
- Allows to express satisfiability-preserving transformations like variable addition
- As powerful as Extended Resolution

From (D|L)RUP to (D|L)RAT [4]

Resolution Asymmetric Tautology (recap)

Clause c has the *Resolution Asymmetric Tautology* (RAT) property in F w.r.t. literal $x \in c$ iff every resolvent $c' \in \{c \otimes_x \tilde{c} \mid \tilde{c} \in F_{\bar{x}}\}$ has the RUP property in F .

“Only” requiring each clause $c_i \in P$ to have the RAT property (rather than RUP) allows for stronger proofs!

- For RAT clause c , $F \cup c$ is satisfiability-preserving to F but may be not equivalent to F
- Allows to express satisfiability-preserving transformations like variable addition
- As powerful as Extended Resolution

How to incorporate RAT into proof checking?

- DRUP \rightarrow DRAT: For each added clause c_i , find pivot literal $x \in c_i$ and confirm that c_i is RAT in B w.r.t. x
 - Convention: 1st literal of c_i must be valid pivot
 - Find all resolvents, check RUP for every one of them

From (D|L)RUP to (D|L)RAT [4]

Resolution Asymmetric Tautology (recap)

Clause c has the *Resolution Asymmetric Tautology* (RAT) property in F w.r.t. literal $x \in c$ iff every resolvent $c' \in \{c \otimes_x \tilde{c} \mid \tilde{c} \in F_{\bar{x}}\}$ has the RUP property in F .

“Only” requiring each clause $c_i \in P$ to have the RAT property (rather than RUP) allows for stronger proofs!

- For RAT clause c , $F \cup c$ is satisfiability-preserving to F but may be not equivalent to F
- Allows to express satisfiability-preserving transformations like variable addition
- As powerful as Extended Resolution

How to incorporate RAT into proof checking?

- DRUP → **DRAT**: For each added clause c_i , find pivot literal $x \in c_i$ and confirm that c_i is RAT in B w.r.t. x
 - Convention: 1st literal of c_i must be valid pivot
 - Find all resolvents, check RUP for every one of them
- LRUP → **LRAT**: Additions ($\text{add}, id_i, c_i, d_i, r_i$) with $r_i = \langle r_{i1}, \dots, r_{im_i} \rangle$, $m_i \in \mathbb{N}_0$
 - Each r_{ij} references a clause \tilde{c} and the required RUP steps for $c' = c_i \otimes_x \tilde{c}$ (like d_i for c_i in pure RUP)
 - Still need to internally maintain occurrences of each literal to check that all \tilde{c} are covered

Proof (File) Formats: DRAT and LRAT

DIMACS CNF

```
p cnf 4 8
1 -2 0
2 -4 0
1 2 4 0
-1 -3 0
1 -3 0
-1 3 0
1 3 -4 0
1 3 4 0
```

DRAT proof

-3	0		
1	2	0	
-1	0		
d	-3	0	
2	3	-4	0
1	2	3	0
			0

LRAT proof

9	-3	0	5	4	0		
10	1	2	0	3	2	0	
11	-1	0	6	9	0		
11	d	9	0				
12	2	3	-4	0	7	11	0
13	1	2	3	0	8	12	0
14	0	11	10	1			

These proofs only feature RUP additions. In an LRAT addition, each r_{ij} is written as the negated ID of \tilde{c} followed by IDs for the RUP steps of c' .

DRAT and LRAT: Pragmatics

DRAT-based solving and checking: Common tool chain

```
./solver input.cnf proof.drat    // solve, output DRAT proof
./drat-trim input.cnf proof.drat -L proof.lrat   // transform DRAT proof to LRAT
./cake-lpr input.cnf proof.lrat   // validate LRAT proof - trusted / verified
```

DRAT and LRAT: Pragmatics

DRAT-based solving and checking: Common tool chain

```
./solver input.cnf proof.drat    // solve, output DRAT proof  
./drat-trim input.cnf proof.drat -L proof.lrat   // transform DRAT proof to LRAT  
./cake-lpr input.cnf proof.lrat   // validate LRAT proof - trusted / verified
```

Compressed DRAT and LRAT proofs

- **Binary file** instead of text file
 - Numbers stored as integers instead of strings
 - Implicit separators
- **Variable byte length** encoding for each literal, clause ID
 - A byte's **first seven bits** denote its actual value
 - A byte's **last bit** indicates if **the number continues** at the next byte
 - Makes proof independent of underlying integer domain (32 vs. 64 bit), saves space for small values

Adding Proof Support to Solvers

I wrote my own CDCL SAT solver. How can I let it [produce UNSAT proofs](#)?

DRAT:

Adding Proof Support to Solvers

I wrote my own CDCL SAT solver. How can I let it [produce UNSAT proofs](#)?

DRAT: [super simple!](#)

- Create an (empty) proof file at upstart
- Log each new redundant clause (including the empty clause) into the proof file
- Log each discarded clause into the proof file, prepended with “d”

LRAT:

Adding Proof Support to Solvers

I wrote my own CDCL SAT solver. How can I let it [produce UNSAT proofs](#)?

DRAT: [super simple!](#)

- Create an (empty) proof file at upstart
- Log each new redundant clause (including the empty clause) into the proof file
- Log each discarded clause into the proof file, prepended with “d”

LRAT: Clause addition lines need to be enriched [with dependency information \(“hints”\)](#)

- CDCL conflict clauses: [simple](#) – use conflict’s [implication graph](#)
- [Additional effort](#) for each employed pre-/inprocessing technique

Adding Proof Support to Solvers

I wrote my own CDCL SAT solver. How can I let it [produce UNSAT proofs](#)?

DRAT: [super simple!](#)

- Create an (empty) proof file at upstart
- Log each new redundant clause (including the empty clause) into the proof file
- Log each discarded clause into the proof file, prepended with “d”

LRAT: Clause addition lines need to be enriched [with dependency information \(“hints”\)](#)

- CDCL conflict clauses: [simple](#) – use conflict’s [implication graph](#)
- [Additional effort](#) for each employed pre-/inprocessing technique

Other formats:

- [FRAT](#): Compromise between DRAT and FRAT at the developer’s discretion [5]
- [DPR](#), [LPR](#): Propagation Redundancy (PR) reasoning [6]
- [VeriPB](#): Pseudo-Boolean reasoning [6]

Note: [formally verified checkers](#) are available for [all these formats](#) (sometimes translation-based)

Proof Production: The Parallel Case

What about proofs from parallel solvers?

- Pure portfolios:

Proof Production: The Parallel Case

What about proofs from parallel solvers?

- Pure portfolios: [trivial](#) if each participant produces a proof
- Search space splitting solvers:

Proof Production: The Parallel Case

What about proofs from parallel solvers?

- Pure portfolios: [trivial](#) if each participant produces a proof
- Search space splitting solvers: straight forward to [stitch together proofs](#) for sub-problems (e.g., [7])
- Clause-sharing solvers:

Proof Production: The Parallel Case

What about proofs from parallel solvers?

- Pure portfolios: trivial if each participant produces a proof
- Search space splitting solvers: straight forward to stitch together proofs for sub-problems (e.g., [7])
- Clause-sharing solvers: more difficult due to cross-references between solvers' clauses [8]

Before 2023: Large gap of trustworthiness between best sequential and best parallel (clause-sharing) solvers

Proof Production: The Parallel Case

What about proofs from parallel solvers?

- Pure portfolios: trivial if each participant produces a proof
- Search space splitting solvers: straight forward to stitch together proofs for sub-problems (e.g., [7])
- Clause-sharing solvers: more difficult due to cross-references between solvers' clauses [8]

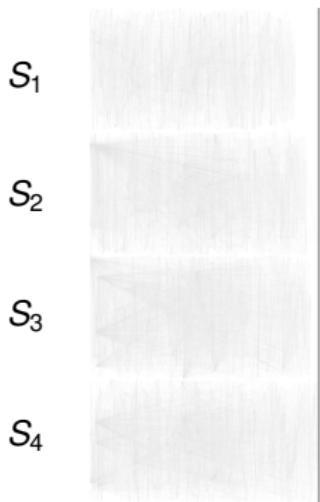
Before 2023: Large gap of trustworthiness between best sequential and best parallel (clause-sharing) solvers

2023: LRAT-based proofs from clause-sharing solvers [9]

- Globally unique clause IDs without communication
 - for o original clauses and p solver threads, the i -th thread assigns clause IDs $o + i + kp$ ($k \in \mathbb{N}_0$)
- After solving, rewind the procedure, using “hints” of LRAT to trace dependencies of empty clause
- Funnel all clauses marked as required into a single, ordered proof file

Distributed Proof Production (1/2) [9]

— Produced Clauses →

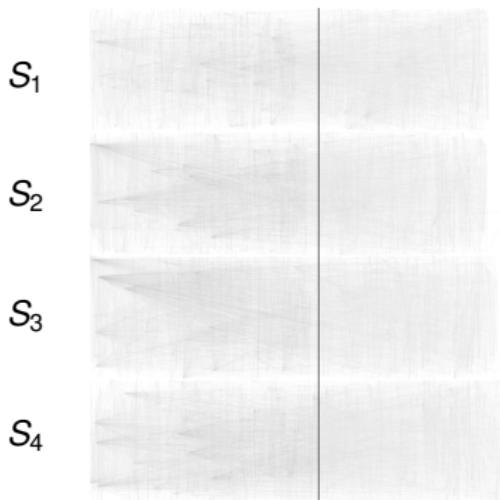


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Solving: Each thread **derives and shares** clauses, logs to **partial proof file**

Distributed Proof Production (1/2) [9]

— Produced Clauses →

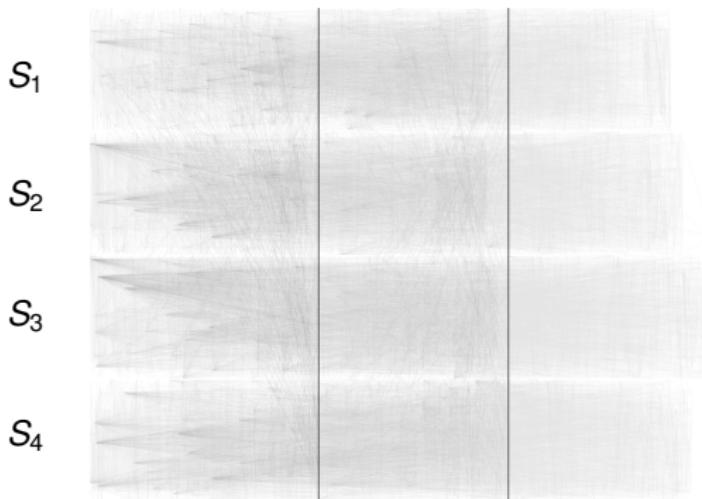


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Solving: Each thread **derives and shares** clauses, logs to **partial proof file**

Distributed Proof Production (1/2) [9]

— Produced Clauses →

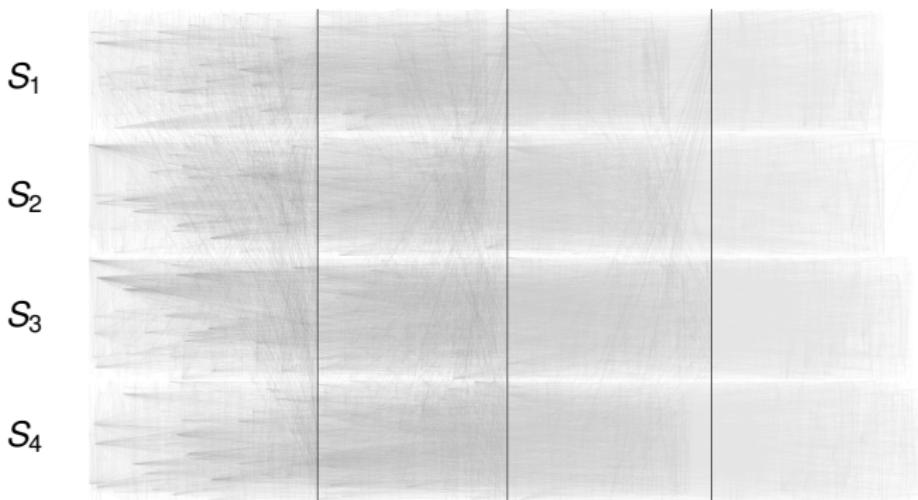


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Solving: Each thread **derives and shares** clauses, logs to **partial proof file**

Distributed Proof Production (1/2) [9]

— Produced Clauses →

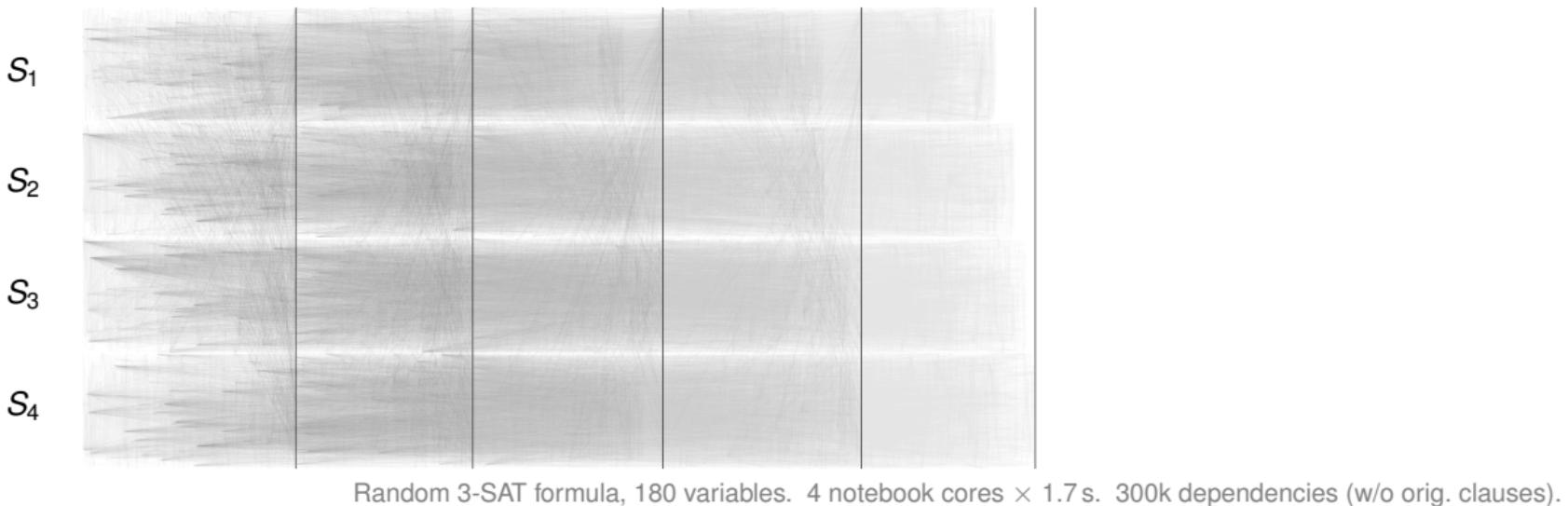


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Solving: Each thread **derives and shares** clauses, logs to **partial proof file**

Distributed Proof Production (1/2) [9]

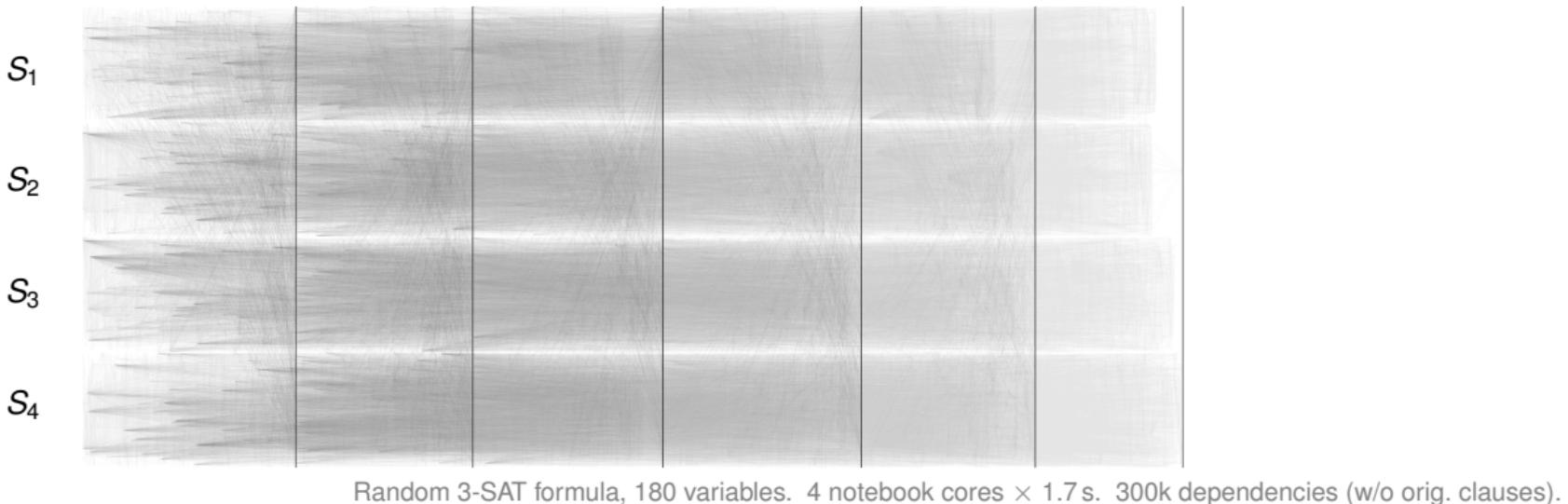
— Produced Clauses →



Solving: Each thread **derives and shares** clauses, logs to **partial proof file**

Distributed Proof Production (1/2) [9]

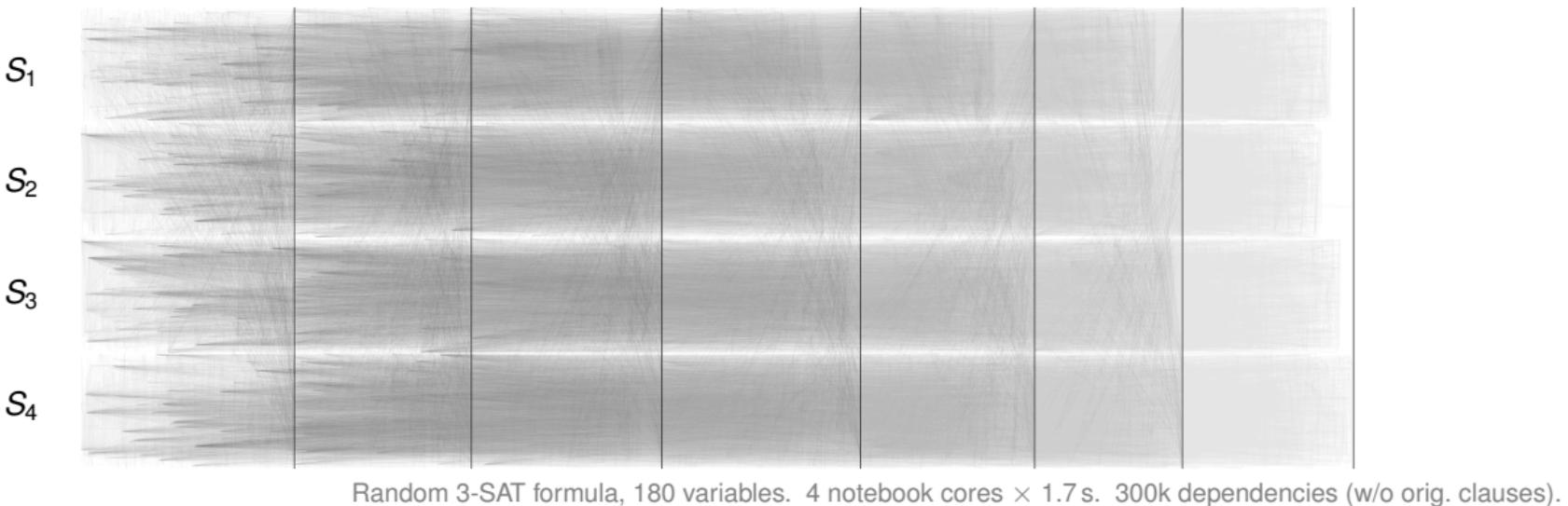
— Produced Clauses →



Solving: Each thread **derives and shares** clauses, logs to **partial proof file**

Distributed Proof Production (1/2) [9]

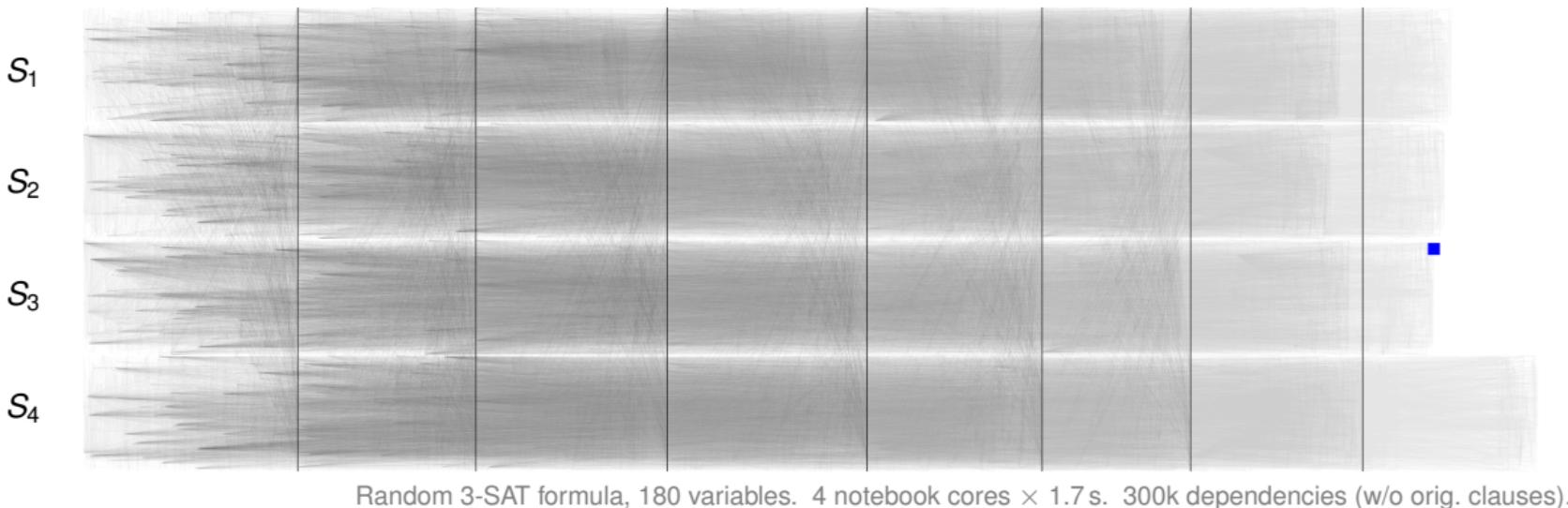
— Produced Clauses →



Solving: Each thread derives and shares clauses, logs to partial proof file

Distributed Proof Production (1/2) [9]

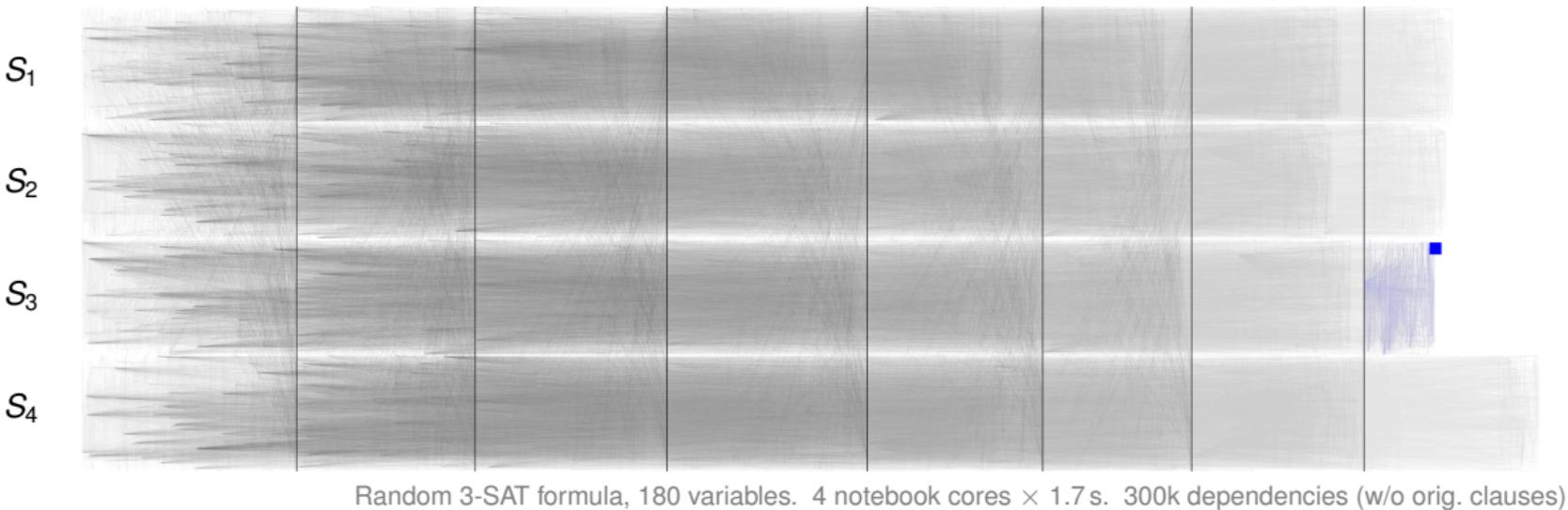
— Produced Clauses →



Solving: Each thread derives and shares clauses, logs to partial proof file

Distributed Proof Production (1/2) [9]

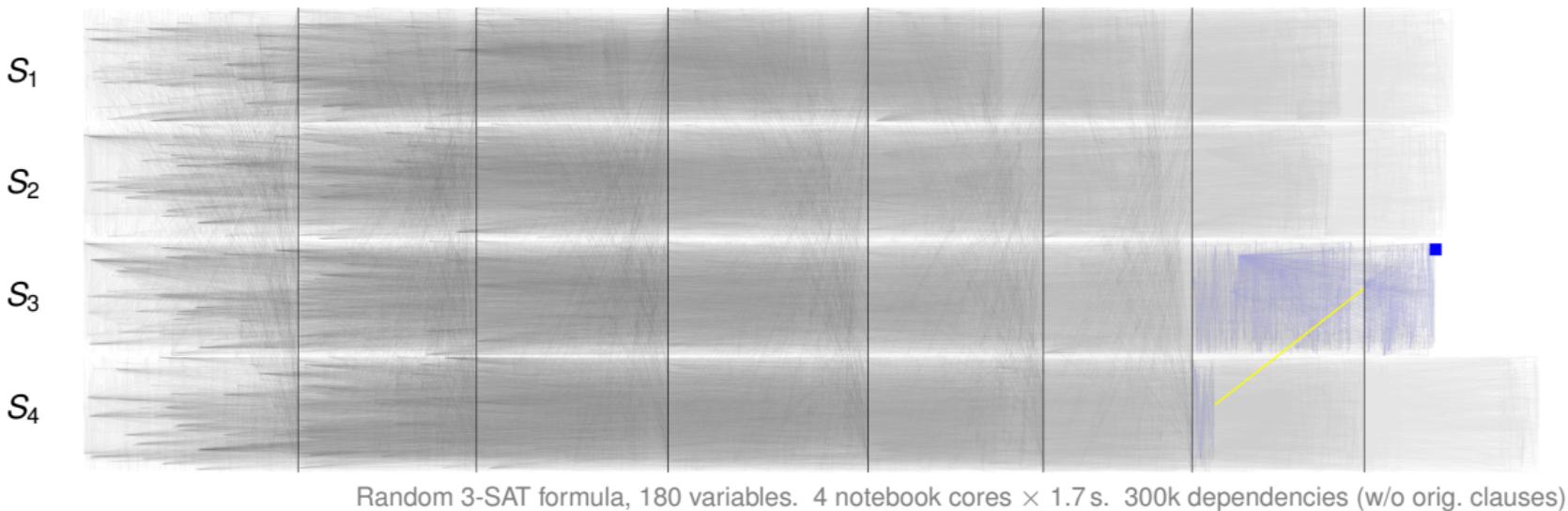
— Produced Clauses →



Reconstruction: Trace required clauses, revert each clause exchange

Distributed Proof Production (1/2) [9]

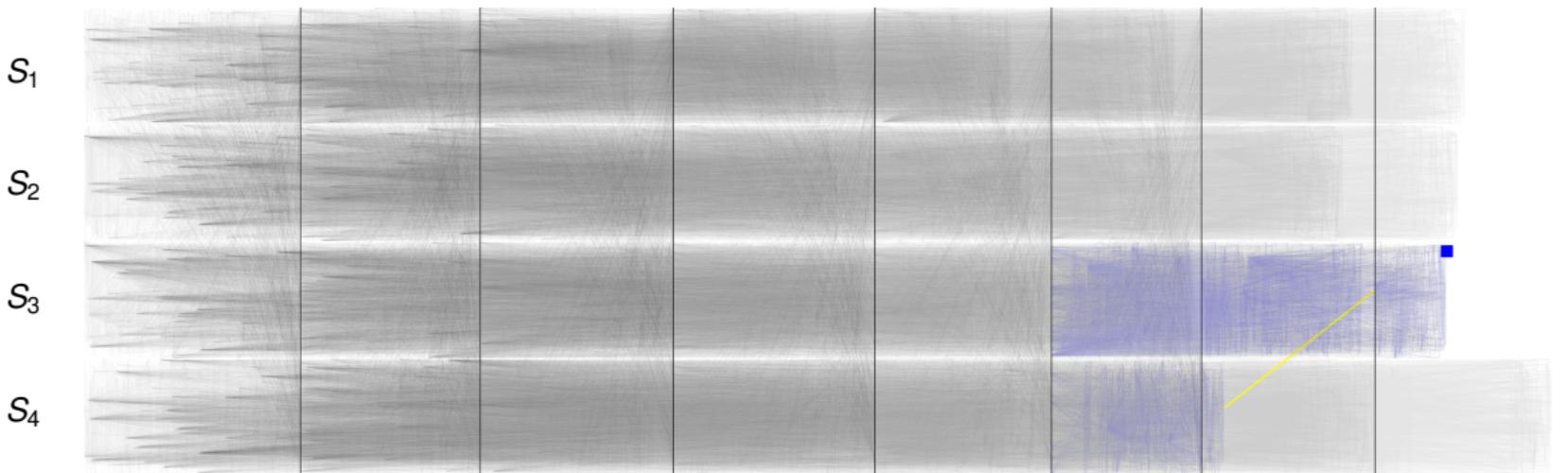
— Produced Clauses →



Reconstruction: Trace required clauses, revert each clause exchange

Distributed Proof Production (1/2) [9]

— Produced Clauses →

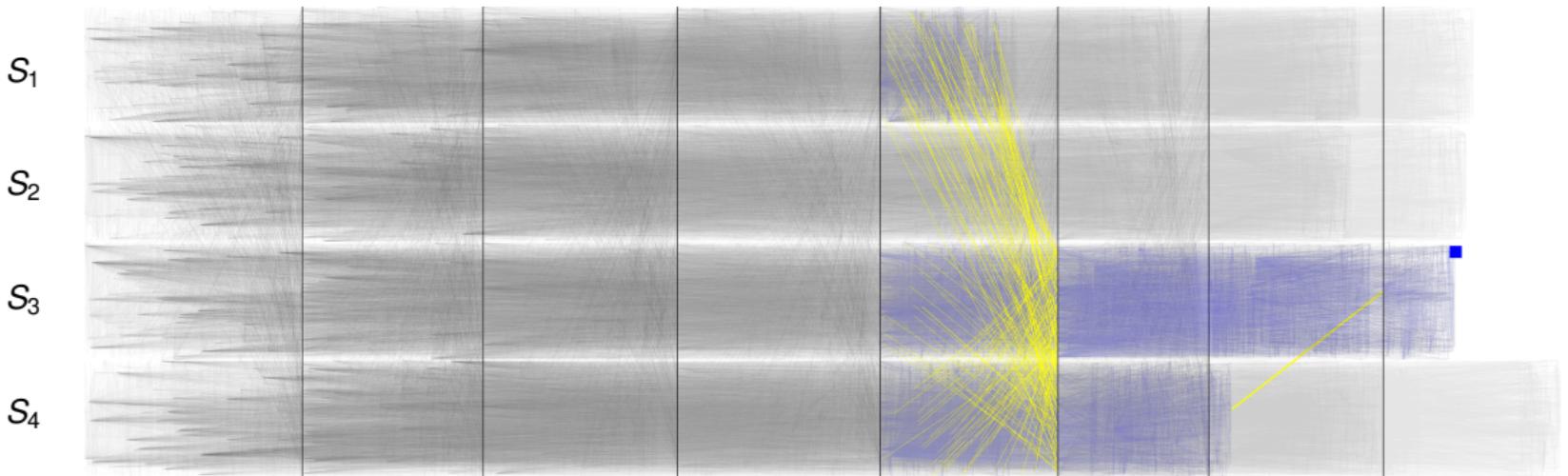


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Reconstruction: Trace required clauses, revert each clause exchange

Distributed Proof Production (1/2) [9]

— Produced Clauses →

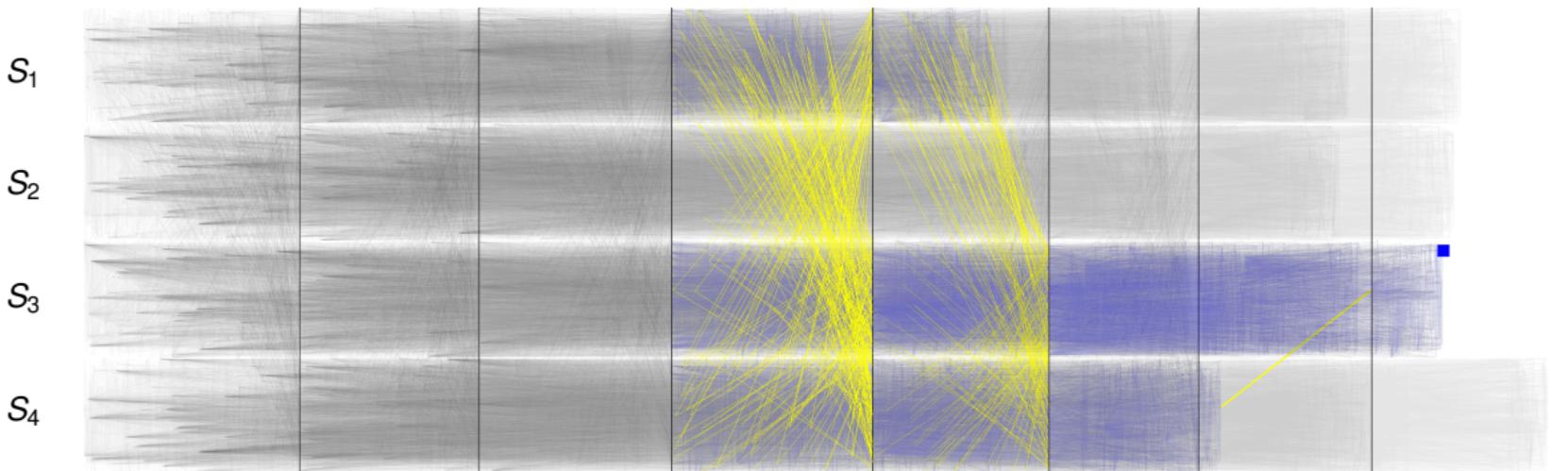


Random 3-SAT formula, 180 variables. 4 notebook cores × 1.7 s. 300k dependencies (w/o orig. clauses).

Reconstruction: Trace required clauses, **revert** each clause exchange

Distributed Proof Production (1/2) [9]

— Produced Clauses →

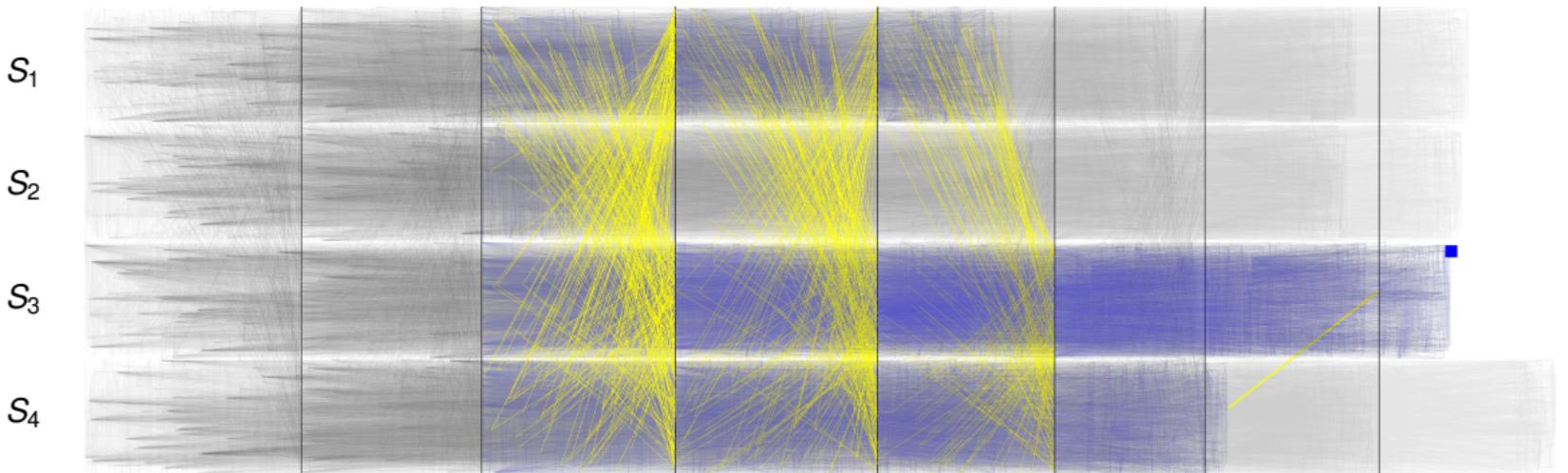


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Reconstruction: Trace required clauses, revert each clause exchange

Distributed Proof Production (1/2) [9]

— Produced Clauses →

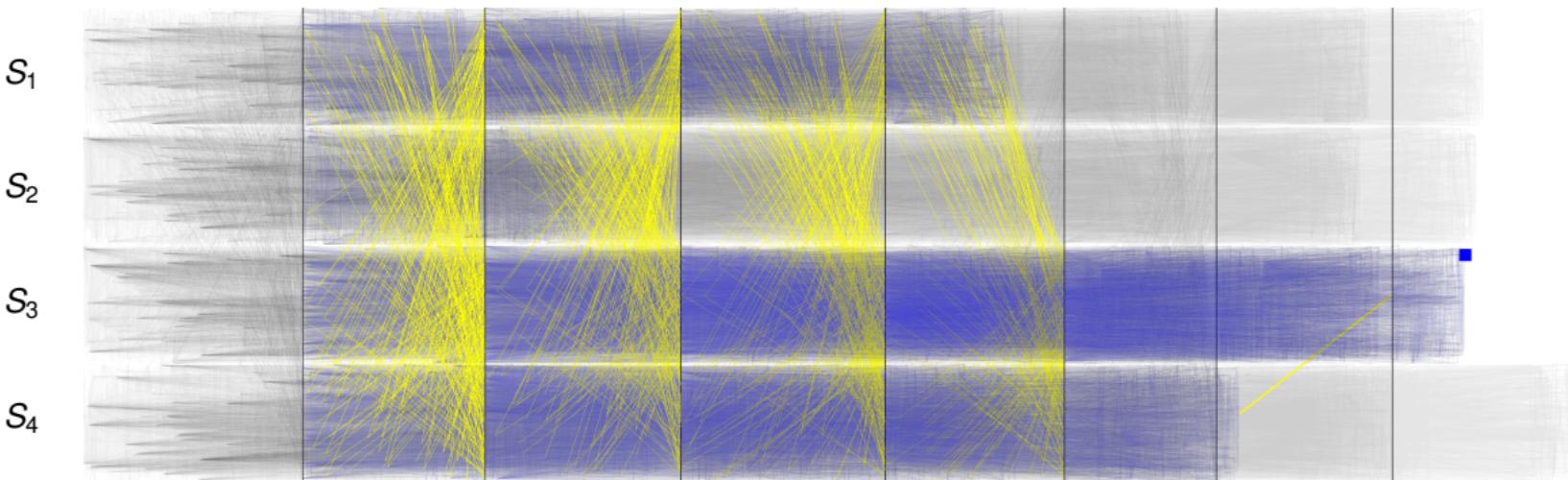


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Reconstruction: Trace required clauses, revert each clause exchange

Distributed Proof Production (1/2) [9]

— Produced Clauses →

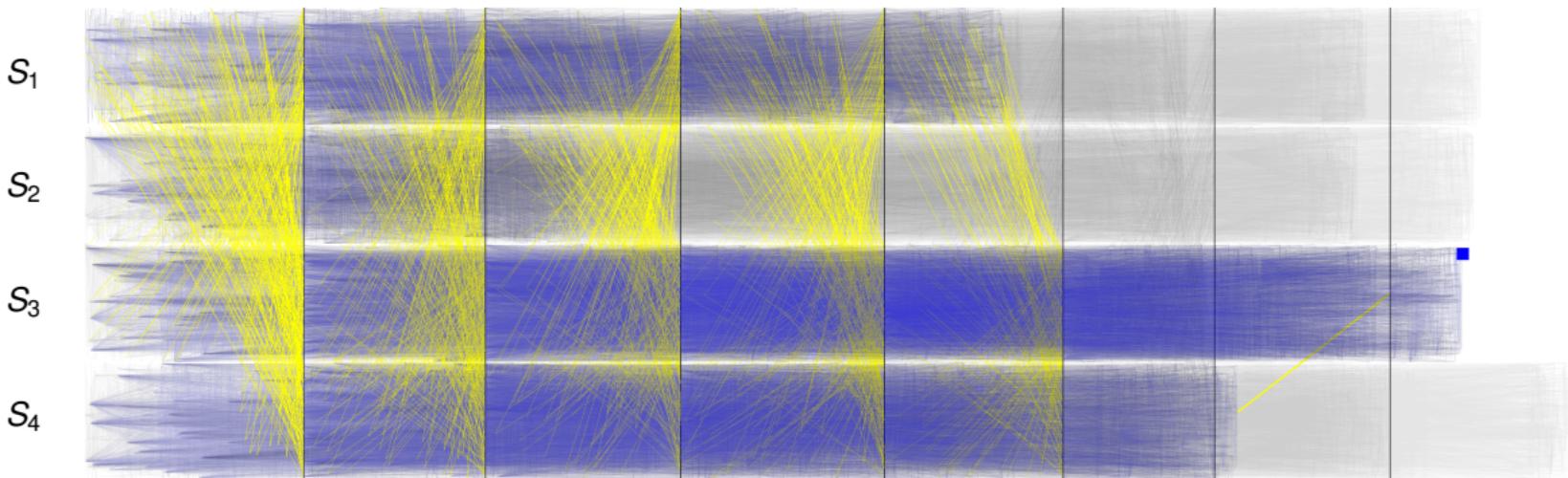


Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Reconstruction: Trace required clauses, revert each clause exchange

Distributed Proof Production (1/2) [9]

— Produced Clauses →



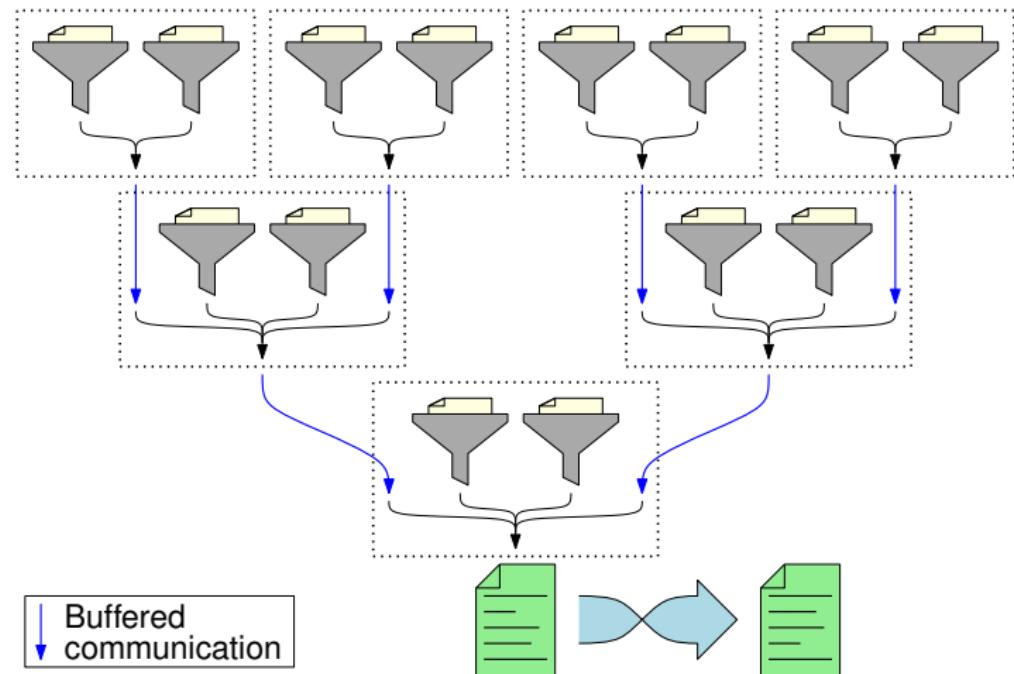
Random 3-SAT formula, 180 variables. 4 notebook cores \times 1.7 s. 300k dependencies (w/o orig. clauses).

Reconstruction: Trace required clauses, **revert** each clause exchange

Distributed Proof Production (2/2) [9]

Merging:

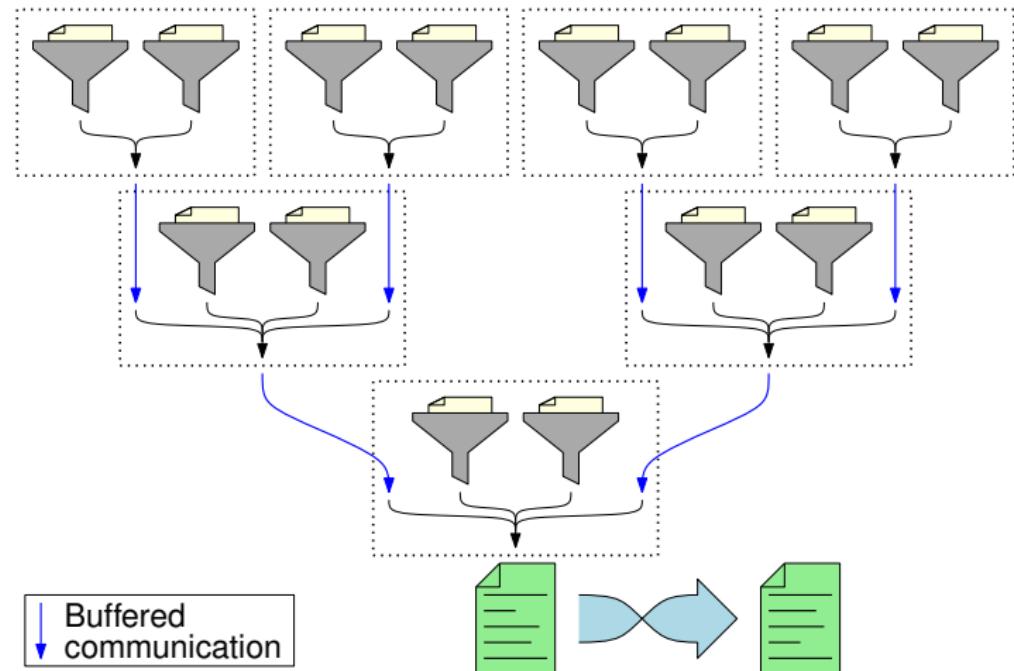
- Funnel required clause additions, **still in reverse order**, into singular proof file
- Hierarchical merging along tree



Distributed Proof Production (2/2) [9]

Merging:

- Funnel required clause additions, still in reverse order, into singular proof file
- Hierarchical merging along tree
- “Root process” writes output to file
 - Seeing an ID d_{ij} for the first time?
⇒ write deletion of d_{ij} before writing the current statement!
 - Finally: Invert lines of proof file



Distributed Proof Production: Discussion

Results:

(Latest setup, JAR'24 submission)

- Mean speedup of proof-emitting solver @ 1520 cores over sequential solver (solving times only): 17.5
 - Mean speedup of best MALLOB SAT @ 1520 cores over sequential solver: 26.9
- On average, assembling and checking a proof takes $\approx 3 \times$ solving time
 - Mean overhead of DRAT proof checking over sequential solving: $\approx 1 \times$
- Pruning irrelevant clause additions reduces proof size by $\approx 30\text{--}40 \times$
- LRAT proof size: median 3.1 GB, mean 11.6 GB, maximum 233.9 GB

Bottleneck:

Distributed Proof Production: Discussion

Results:

(Latest setup, JAR'24 submission)

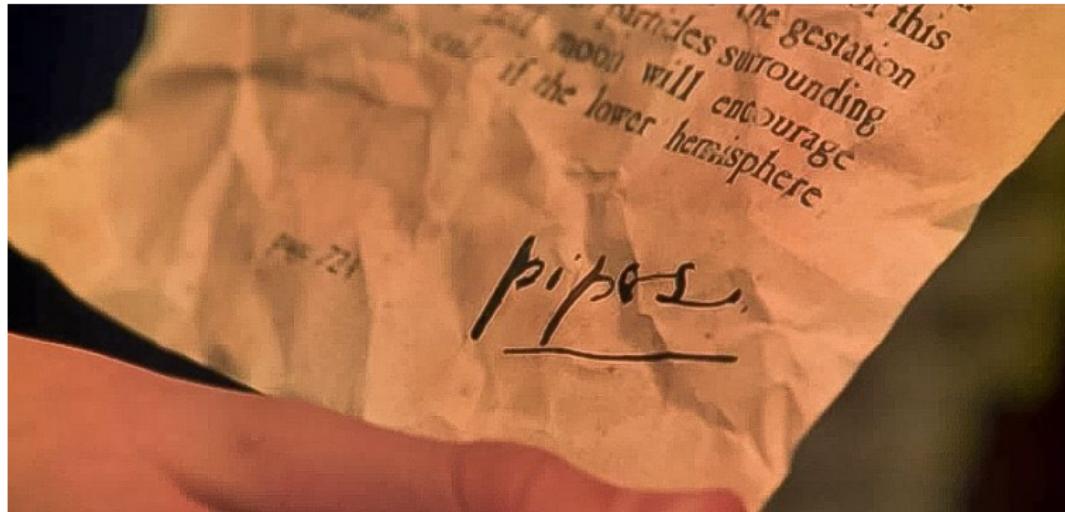
- Mean speedup of proof-emitting solver @ 1520 cores over sequential solver (solving times only): 17.5
 - Mean speedup of best MALLOB SAT @ 1520 cores over sequential solver: 26.9
- On average, assembling and checking a proof takes $\approx 3 \times$ solving time
 - Mean overhead of DRAT proof checking over sequential solving: $\approx 1 \times$
- Pruning irrelevant clause additions reduces proof size by $\approx 30\text{--}40 \times$
- LRAT proof size: median 3.1 GB, mean 11.6 GB, maximum 233.9 GB

Bottleneck: Assembly and validation of a monolithic proof

- Proof creation throttled by I/O bandwidth at final process
- Checking can take very long
- The assembled proof's "corridor" of active clauses may no longer fit into RAM

Can we do better?

Hermione's Answer to More Scalable Trusted Solving



<https://i.pinimg.com/originals/1b/3d/b6/1b3db639721eeafb188a3cc3060ff58b.jpg>

Beyond Monolithic Proof Files

```
mkfifo lratproof.pipe // create "pipe" file  
// Solve & check concurrently via pipe  
.solver input.cnf lratproof.pipe &  
./lrat-check input.cnf lratproof.pipe
```



Marijn Heule: Since LRUP checking is so efficient, we can feasibly do it [in realtime!](#)

- Solver streams proof output into a [pipe](#) (UNIX special file)
- Checker reads proof from pipe and checks it [on-the-fly](#)
 - [checking is done as soon as solving is done!](#)

Beyond Monolithic Proof Files

```
mkfifo lratproof.pipe // create "pipe" file  
// Solve & check concurrently via pipe  
.solver input.cnf lratproof.pipe &  
.lrat-check input.cnf lratproof.pipe
```

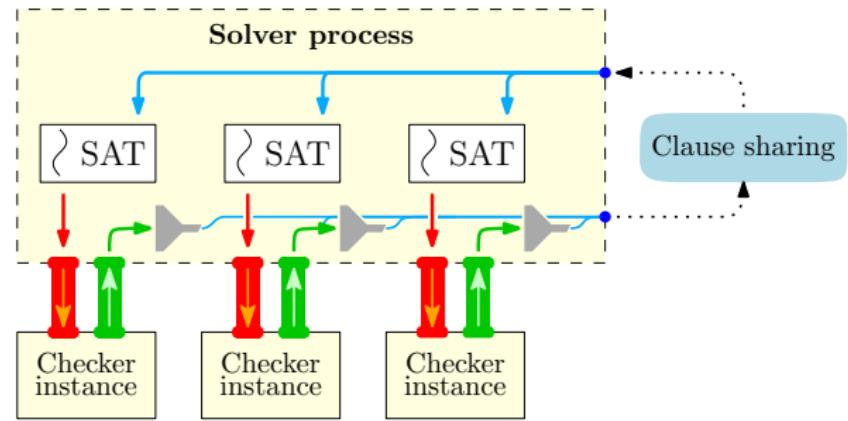


Marijn Heule: Since LRUP checking is so efficient, we can feasibly do it [in realtime!](#)

- Solver streams proof output into a [pipe](#) (UNIX special file)
- Checker reads proof from pipe and checks it [on-the-fly](#)
 - [checking is done as soon as solving is done!](#)
- [Almost no slowdown](#) when running solver and checker on [two hardware threads of the same core](#)
- [No disk I/O required](#), [same program code as with normal files](#) (execute `mkfifo` beforehand)
- [Does not yield a persistent artifact](#) to validate by independent parties

Clause-Sharing Solving with on-the-fly Checking [10]

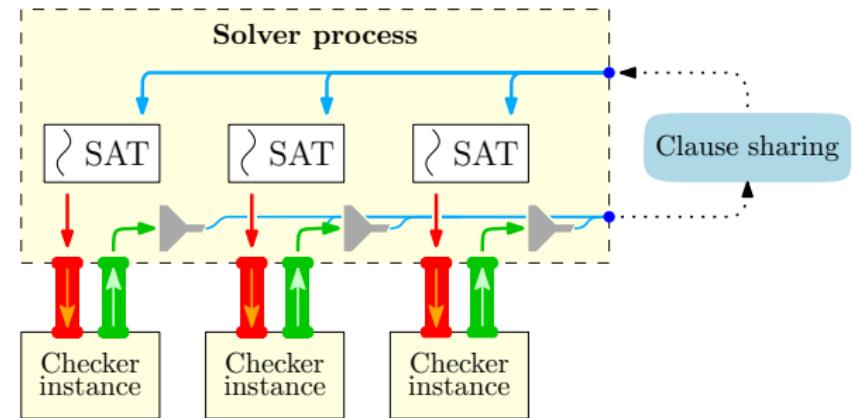
- One checker per solver thread
- Incoming shared clauses are forwarded to the checker **without (LRUP) checking**



github.com/domschrei/impcheck

Clause-Sharing Solving with on-the-fly Checking [10]

- One checker per solver thread
- Incoming shared clauses are forwarded to the checker **without (LRUP) checking**
- How do we account for **incorrect shared clauses**?



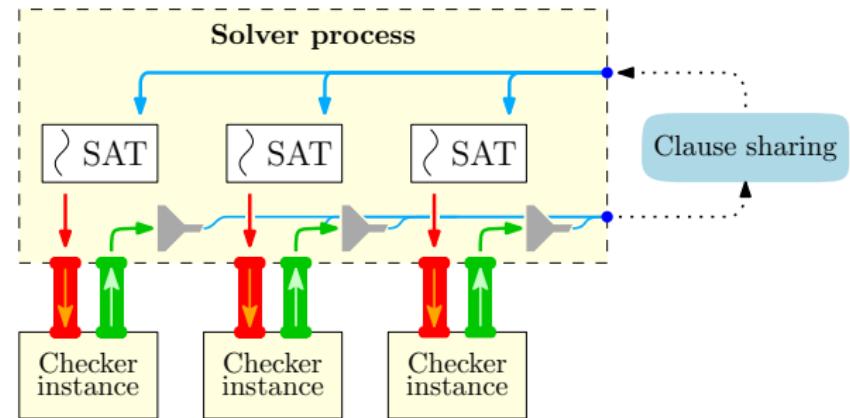
github.com/domschrei/impcheck

Clause-Sharing Solving with on-the-fly Checking [10]

- One checker per solver thread
- Incoming shared clauses are forwarded to the checker **without (LRUP) checking**
- Checkers **sign** each successfully checked clause c with **128-bit signature** based on **shared secret K** :

$$\mathcal{S}_K(c) = H_K(id(c) \parallel c \parallel \mathcal{S}_K(F))$$

H_K : **Message Authentication Code (MAC)**, specifically **SipHash**



github.com/domschrei/impcheck

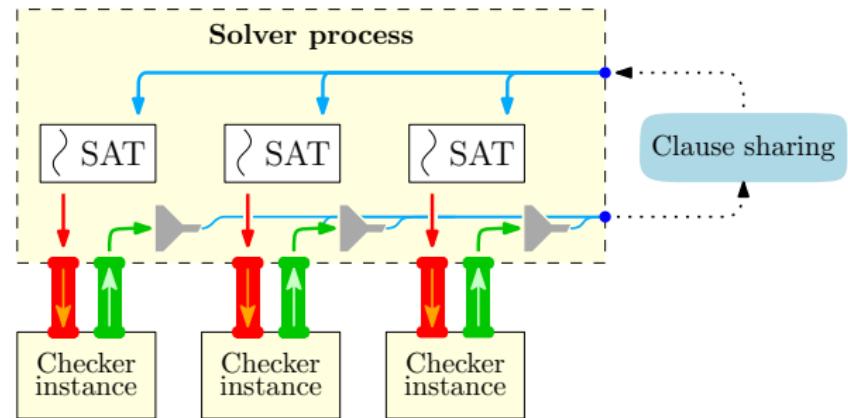
Clause-Sharing Solving with on-the-fly Checking [10]

- One checker per solver thread
- Incoming shared clauses are forwarded to the checker **without (LRUP) checking**
- Checkers **sign** each successfully checked clause c with **128-bit signature** based on **shared secret K** :

$$\mathcal{S}_K(c) = H_K(id(c) \parallel c \parallel \mathcal{S}_K(F))$$

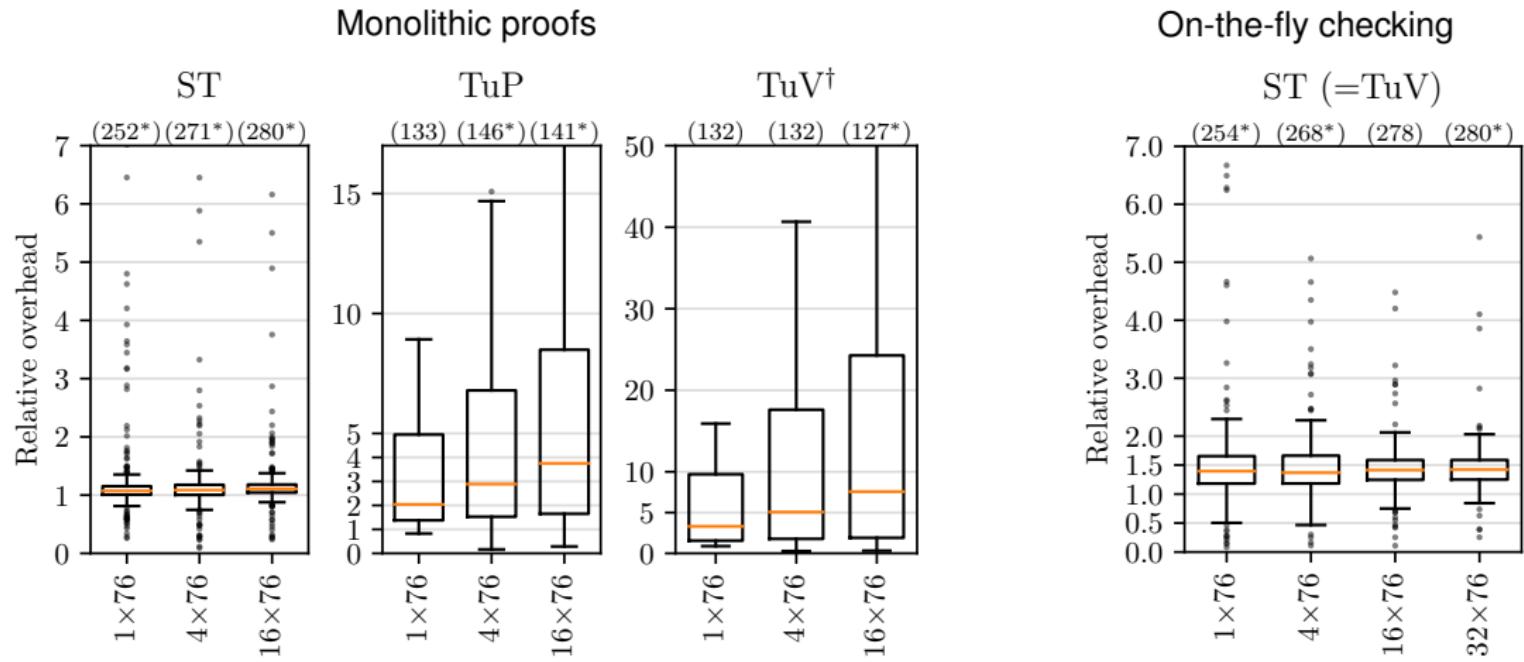
H_K : Message Authentication Code (MAC), specifically **SipHash**

- Incoming clause c is forwarded to checker **together with $\mathcal{S}_K(c)$**
 \Rightarrow Checker can validate that **another checker has signed the clause!**



github.com/domschrei/impcheck

Clause-Sharing Solving + Checking: Scalability [10]



Overhead relative to proof-free solving time · ST: Solving time · TuP: Time until Proof present · TuV: Time until Validation done
 76-core nodes · [†]Data extrapolated · *some data outside of displayed domain

Wrap-Up

- **Proofs:** Powerful and practical technology to ensure that a solver's result is correct
- **Proof formats:** Trade-off between expressivity, checking efficiency, and solver development effort

Wrap-Up

- **Proofs:** Powerful and practical technology to ensure that a solver's result is correct
- **Proof formats:** Trade-off between expressivity, checking efficiency, and solver development effort
- Highly efficient on-the-fly checking is possible if persistent proof artifact is expendable
 - Substantially more scalable than explicit proof production in distributed solving
 - Unclear if / how well this works for actual LRAT (not LRUP) derivations
 - especially for clause-sharing solving

Wrap-Up

- **Proofs:** Powerful and practical technology to ensure that a solver's result is correct
- **Proof formats:** Trade-off between expressivity, checking efficiency, and solver development effort
- Highly efficient on-the-fly checking is possible if persistent proof artifact is expendable
 - Substantially more scalable than explicit proof production in distributed solving
 - Unclear if / how well this works for actual LRAT (not LRUP) derivations
 - especially for clause-sharing solving
- **Right now:** Rise of new proof formats (PR, PB, ...) promising shorter proofs for some problems [6]
 - DRAT / LRAT is technically just as powerful but relies on variable addition for most powerful proofs
 - huge decision space, difficult to find a short proof
 - But: recent success in effective structured variable addition [11]

References

- [1] Allen Van Gelder. "Verifying RUP Proofs of Propositional Unsatisfiability". In: *ISAIM*. 2008.
- [2] Marijn J. H. Heule, Warren Hunt, and Nathan Wetzler. "Trimming while checking clausal proofs". In: *Proc. FMCAD*. IEEE. 2013, pp. 181–188. DOI: [10.1109/fmcad.2013.6679408](https://doi.org/10.1109/fmcad.2013.6679408).
- [3] Luis Cruz-Filipe et al. "Efficient Certified RAT Verification". In: *Proc. CADE*. Ed. by Leonardo de Moura. Vol. 10395. Lecture Notes in Computer Science. 2017, pp. 220–236. DOI: [10.1007/978-3-319-63046-5_14](https://doi.org/10.1007/978-3-319-63046-5_14).
- [4] Marijn J. H. Heule. "The DRAT format and DRAT-trim checker". In: *CoRR* abs/1610.06229 (2016). arXiv: [1610.06229](https://arxiv.org/abs/1610.06229).
- [5] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. "A flexible proof format for SAT solver-elaborator communication". In: *Logical Methods in Computer Science* 18 (2022).
- [6] Tomas Balyo et al., eds. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. English. Department of Computer Science Series of Publications B. Finland: Department of Computer Science, University of Helsinki, 2023.
- [7] Marijn J. H. Heule, Oliver Kullmann, and Victor Marek. "Solving and verifying the boolean pythagorean triples problem via cube-and-conquer". In: *Proc. SAT*. Springer. 2016, pp. 228–245. DOI: [10.1007/978-3-319-40970-2_15](https://doi.org/10.1007/978-3-319-40970-2_15).
- [8] Marijn J. H. Heule, Norbert Manthey, and Tobias Philipp. "Validating Unsatisfiability Results of Clause Sharing Parallel SAT Solvers". In: *Proc. Pragmatics of SAT*. 2014, pp. 12–25. DOI: [10.29007/6vwg](https://doi.org/10.29007/6vwg).
- [9] Dawn Michaelson et al. "Unsatisfiability proofs for distributed clause-sharing SAT solvers". In: *Proc. TACAS*. Springer. 2023, pp. 348–366. DOI: [10.1007/978-3-031-30823-9_18](https://doi.org/10.1007/978-3-031-30823-9_18).
- [10] Dominik Schreiber. "Trusted Scalable SAT Solving with on-the-fly LRAT Checking". Proc. SAT, in press. 2024. DOI: [10.4230/LIPIcs.SAT.2024.6](https://doi.org/10.4230/LIPIcs.SAT.2024.6). URL: <https://dominikschiereb.de/papers/2024-sat-trusted-pre.pdf>.
- [11] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. "Effective Auxiliary Variables via Structured Reencoding". In: *Proc. SAT*. 2023. DOI: [10.4230/LIPIcs.SAT.2023.11](https://doi.org/10.4230/LIPIcs.SAT.2023.11).