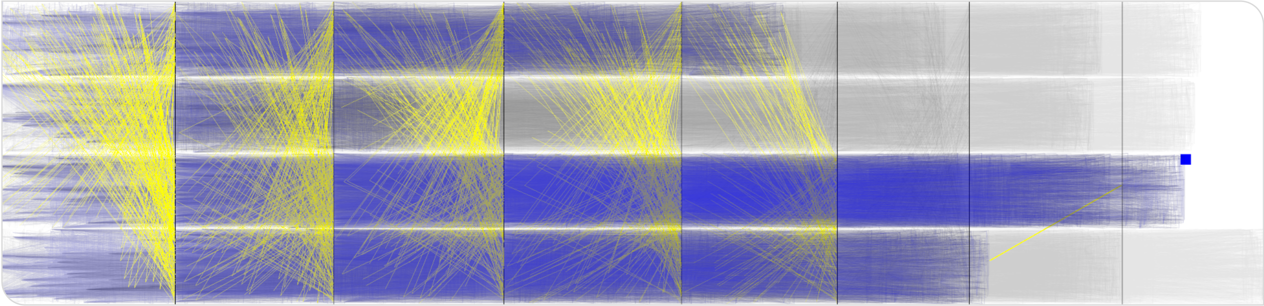
Unsatisfiability Proofs for Distributed Clause-Sharing SAT Solvers

TACAS 2023

Dawn Michaelson, Dominik Schreiber, Marijn J.H. Heule, Benjamin Kiesl-Reiter, Michael W. Whalen | April 24, 2023



Motivation: SAT Solving

SAT Problem

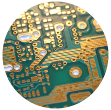
Given CNF formula $F := \bigwedge_{c \in C} \left(\bigvee_{\ell \in c} \ell \right)$, find satisfying variable assignment or report unsatisfiability.

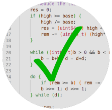
Motivation: SAT Solving

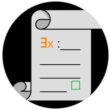
SAT Problem

Given CNF formula $F := \bigwedge_{c \in C} \left(\bigvee_{\ell \in c} \ell \right)$, find satisfying variable assignment or report unsatisfiability.

SAT solvers: Crucial building block for wide range of applications





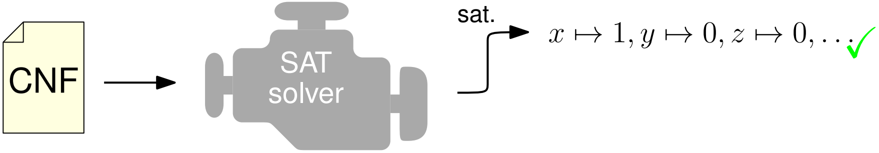


Motivation: SAT Solving and Trust

How can we trust the result of a SAT solver?

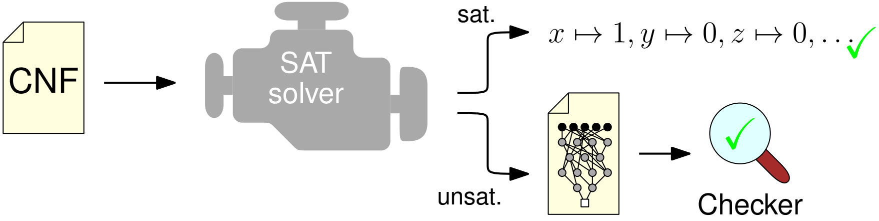
Motivation: SAT Solving and Trust

How can we trust the result of a SAT solver?

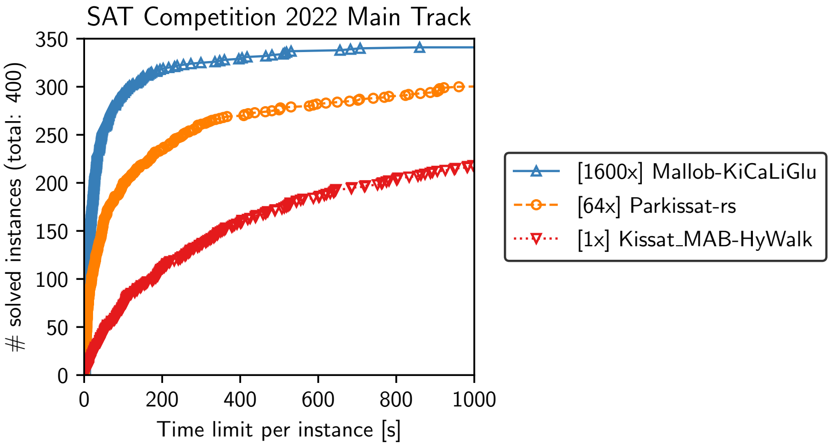


Motivation: SAT Solving and Trust

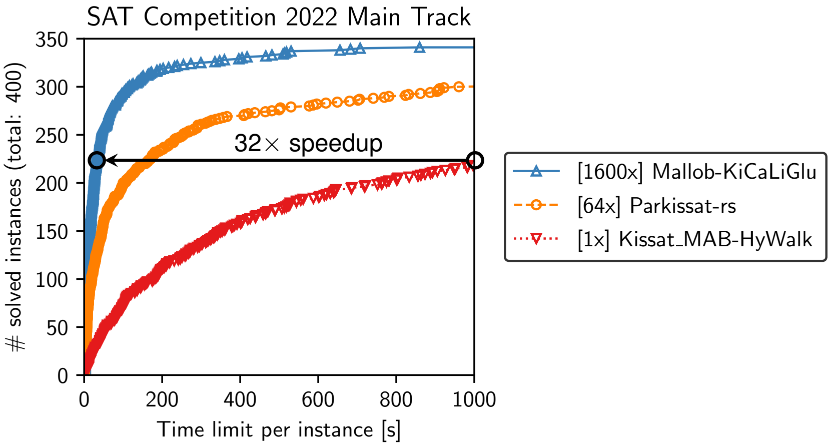
How can we trust the result of a SAT solver?



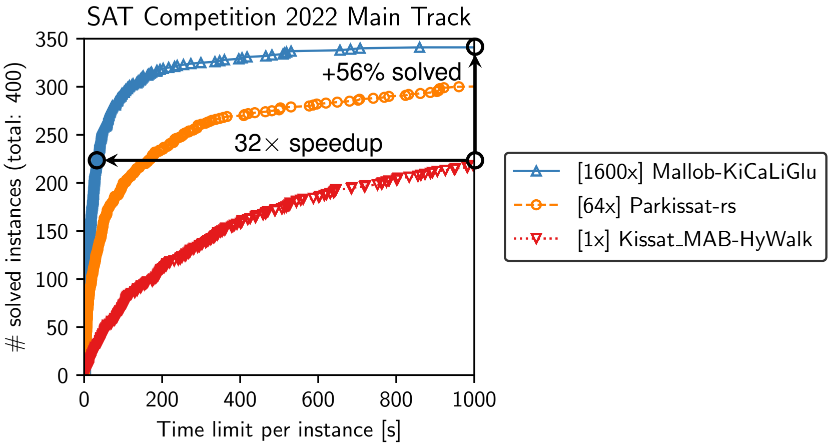
Motivation: Distributed SAT Solving



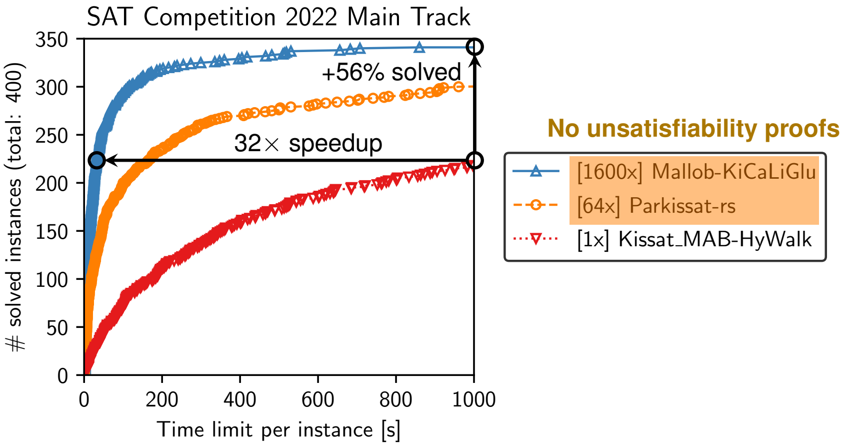
Motivation: Distributed SAT Solving



Motivation: Distributed SAT Solving



Motivation: Distributed SAT Solving



No unsatisfiability proofs!

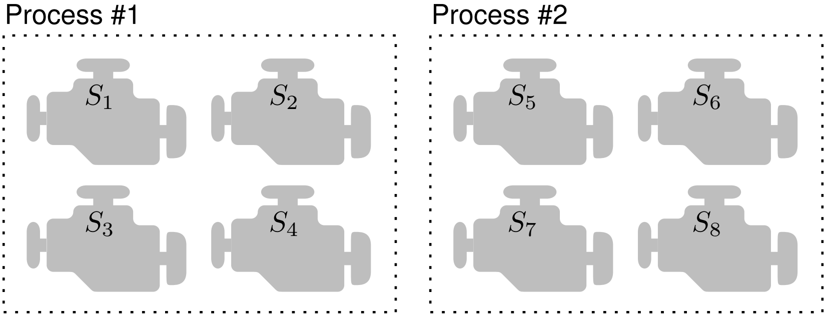
UNSAT Proofs for Distributed Solvers

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

Objective

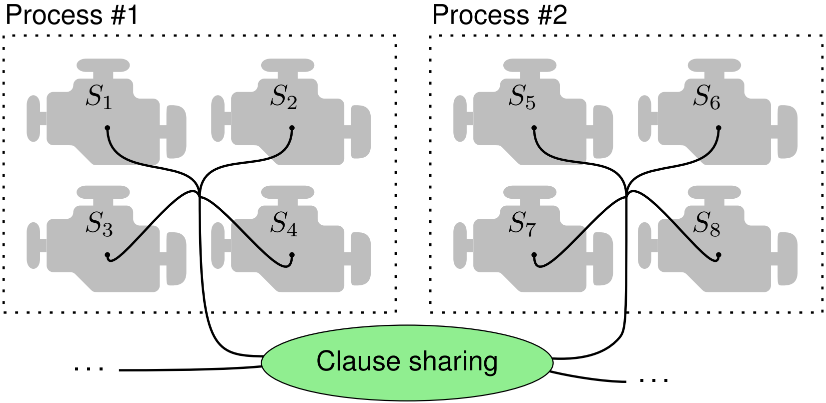
Introduce scalable production of unsatisfiability proofs for distributed clause-sharing SAT solvers, allowing to fully trust their results and exploit their power for critical applications.

Background: Distributed Clause-Sharing SAT Solving

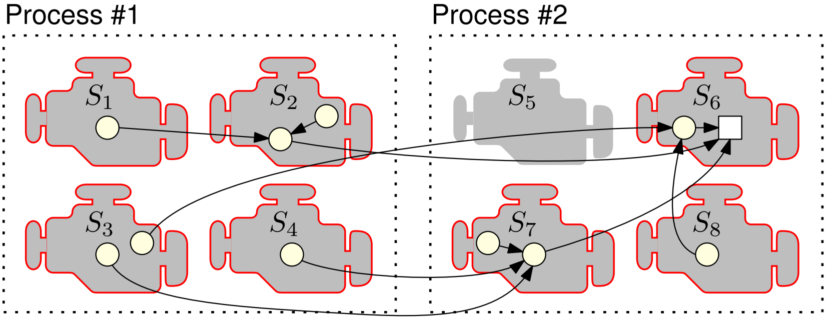


Portfolio of different CDCL solver configurations
$\approx$ producers of conflict clauses

Background: Distributed Clause-Sharing SAT Solving



Background: Distributed Clause-Sharing SAT Solving



Which Proof Format?

DRAT proof format

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

Which Proof Format?

DRAT proof format

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

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

Which Proof Format?

DRAT proof format

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

LRAT proof format

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

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

Which Proof Format?

DRAT proof format

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

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

LRAT proof format

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

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

Which Proof Format?

DRAT proof format

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

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

LRAT proof format

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

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

Unique LRAT IDs across solvers?

Which Proof Format?

DRAT proof format

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

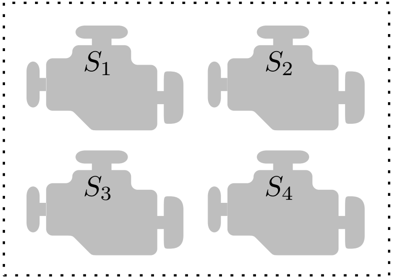
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

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

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

Unique LRAT IDs across solvers?



10 original clauses

Which Proof Format?

DRAT proof format

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

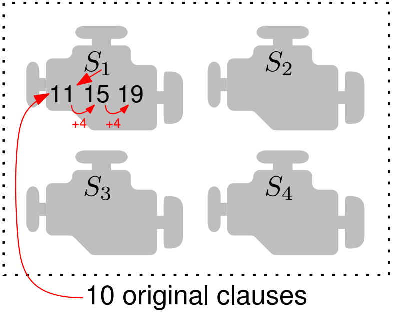
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

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

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

Unique LRAT IDs across solvers?



Which Proof Format?

DRAT proof format

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

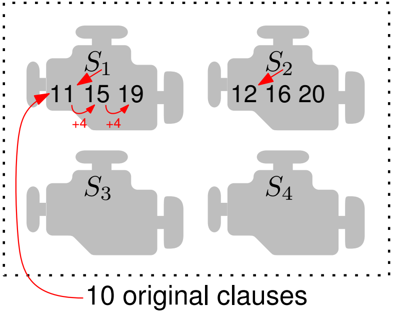
+ compact format
+ prevalent in solvers
- costly checking

LRAT proof format

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

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

Unique LRAT IDs across solvers?



Which Proof Format?

DRAT proof format

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

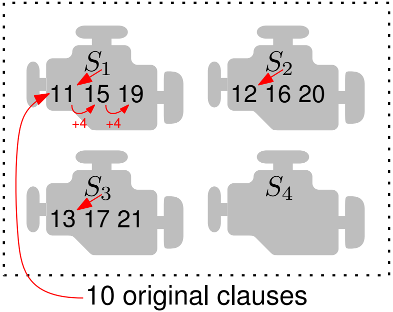
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

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

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

Unique LRAT IDs across solvers?



Which Proof Format?

DRAT proof format

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

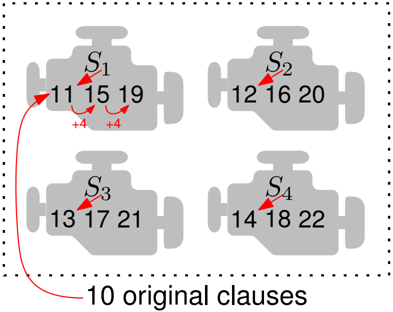
- + compact format
- + prevalent in solvers
- - costly checking

LRAT proof format

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

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

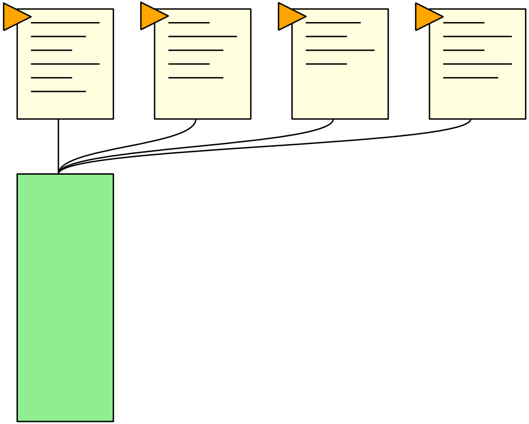
Unique LRAT IDs across solvers?



A Sequential Approach

1. Combination

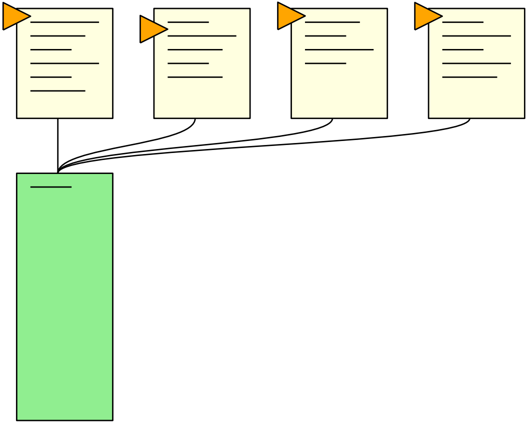
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies d output



A Sequential Approach

1. Combination

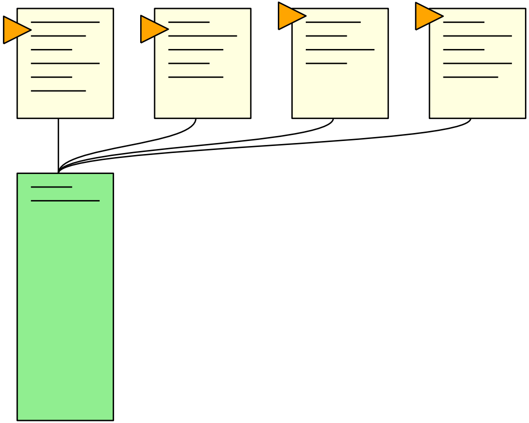
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

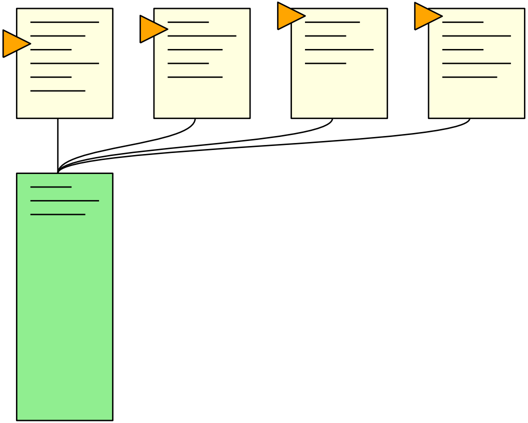
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies d output



A Sequential Approach

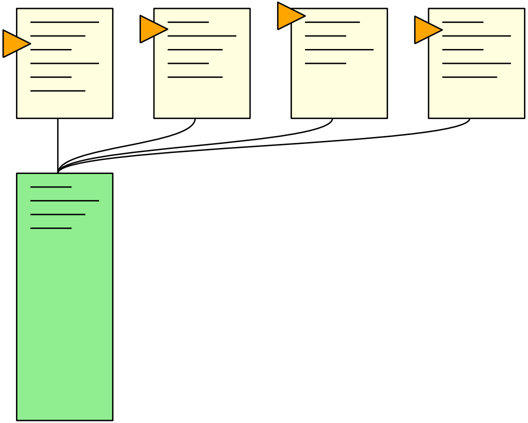
1. Combination

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



A Sequential Approach

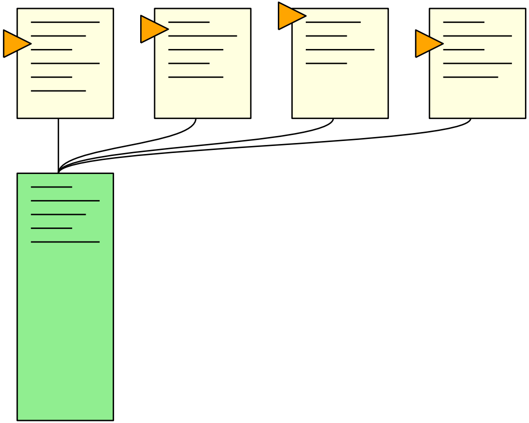
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies d output



A Sequential Approach

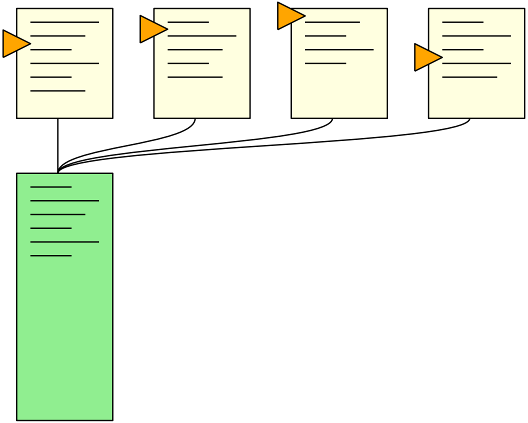
1. Combination

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



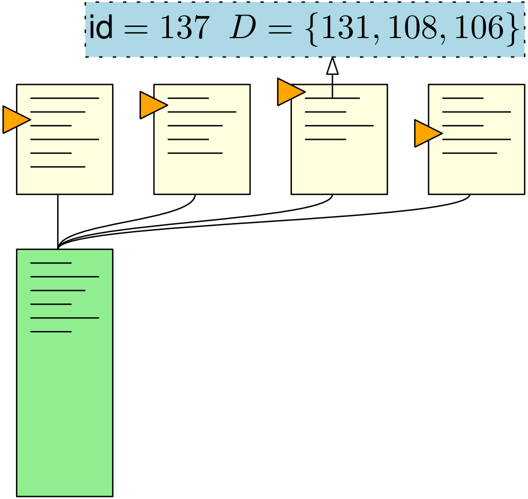
A Sequential Approach

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



A Sequential Approach

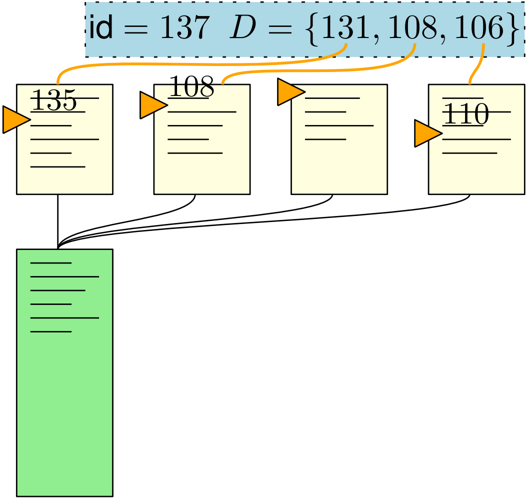
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

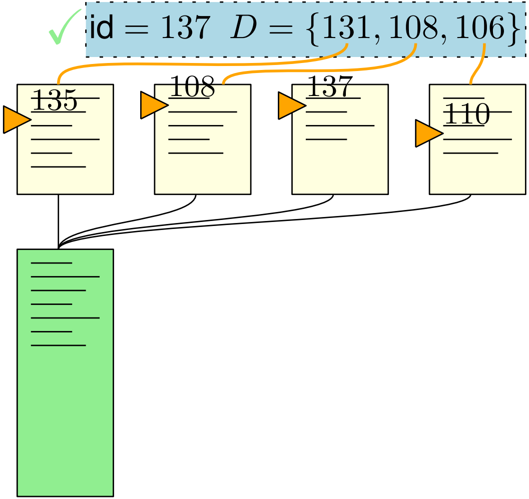
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. **Combination**

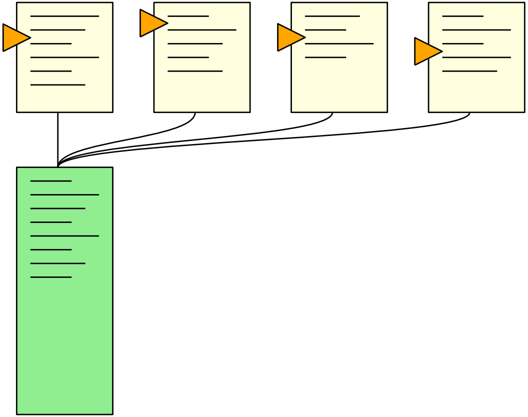
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

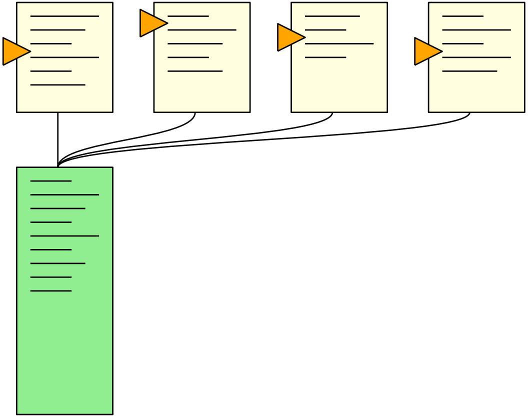
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

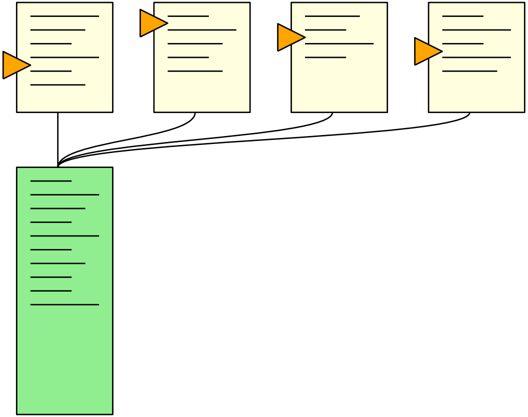
1. Combination

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



A Sequential Approach

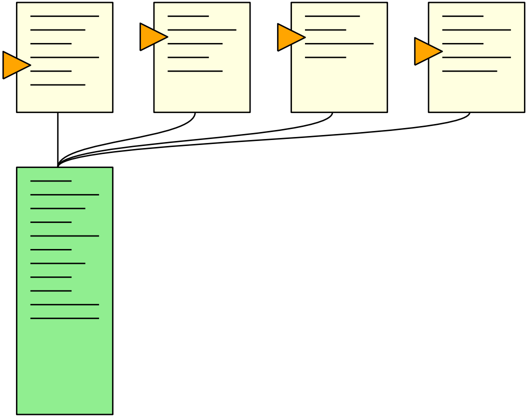
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

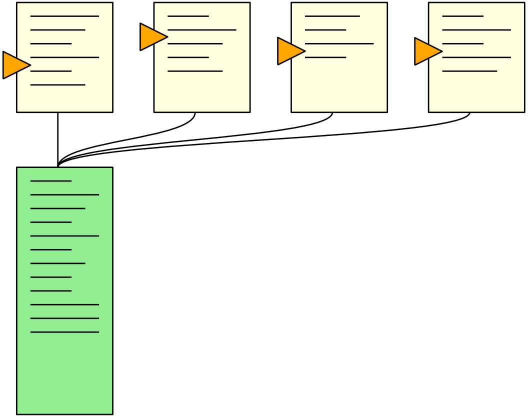
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

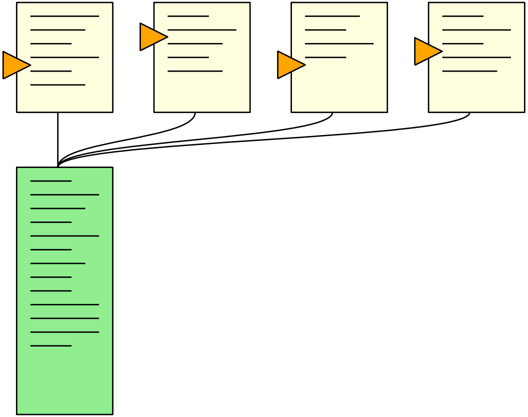
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

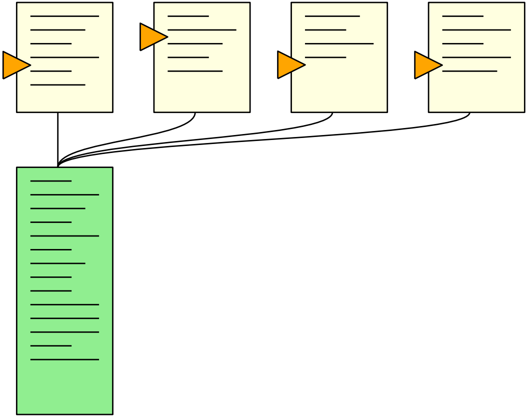
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

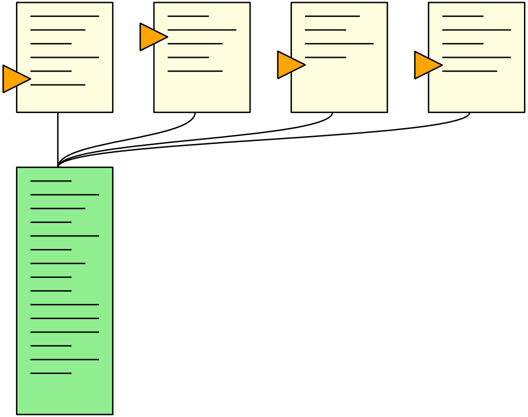
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

1. Combination

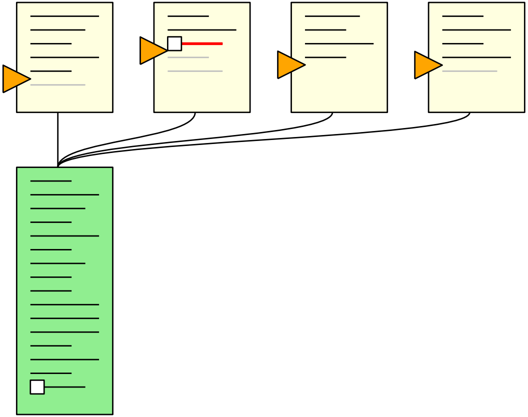
- Read all partial proofs simultaneously
- Output line ⇔ all dependencies *d* output



A Sequential Approach

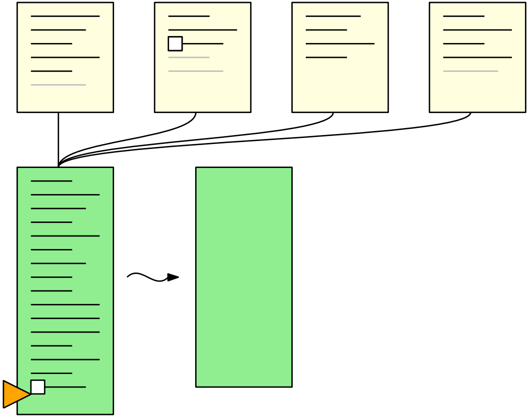
1. Combination

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



A Sequential Approach

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

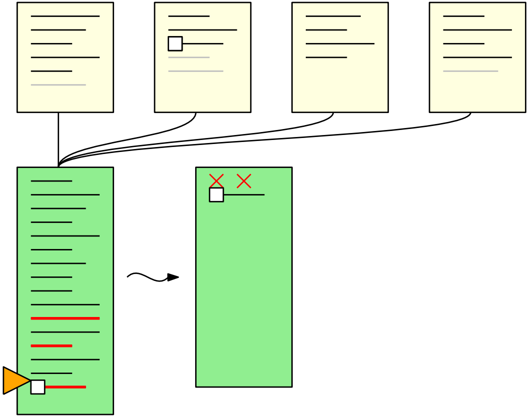


A Sequential Approach

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

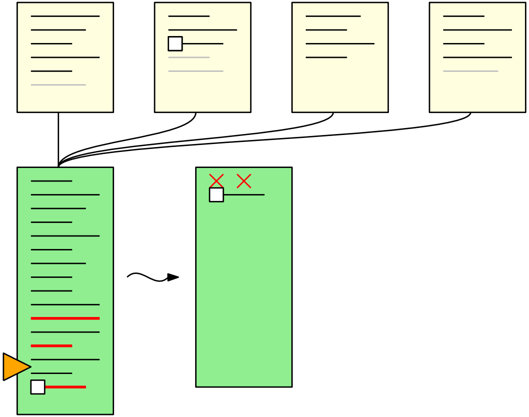
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c



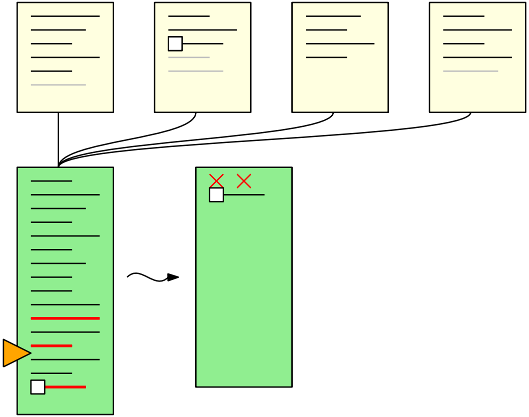
A Sequential Approach

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



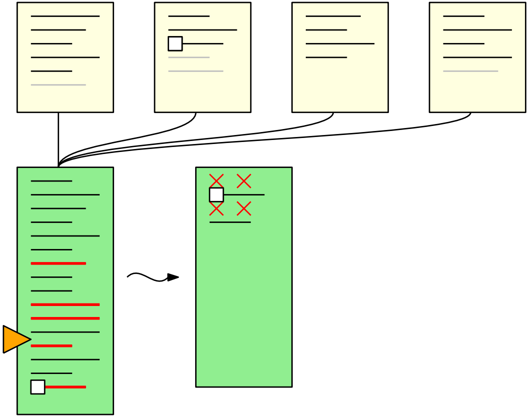
A Sequential Approach

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



A Sequential Approach

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

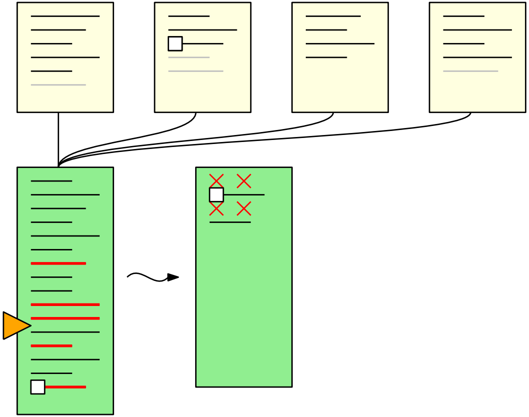


A Sequential Approach

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

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

     $\Rightarrow$ Output addition of c

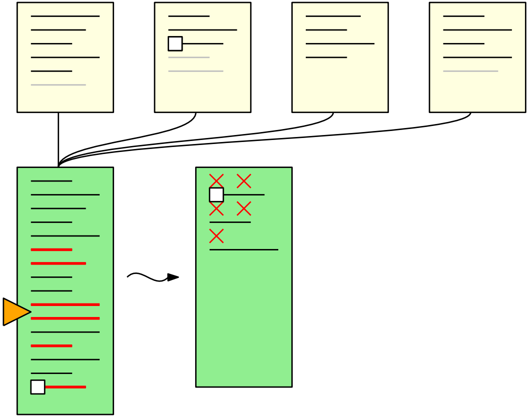


A Sequential Approach

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

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

     ⇒ Output addition of c

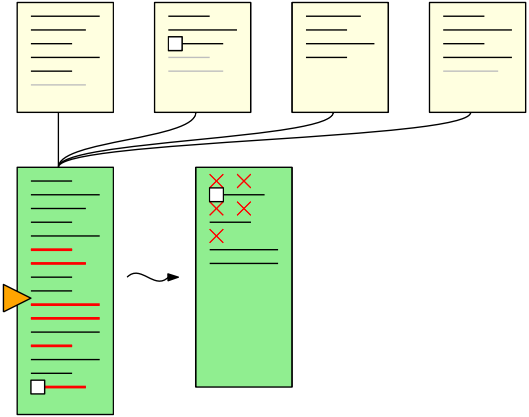


A Sequential Approach

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

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

     ⇒ Output addition of *c*

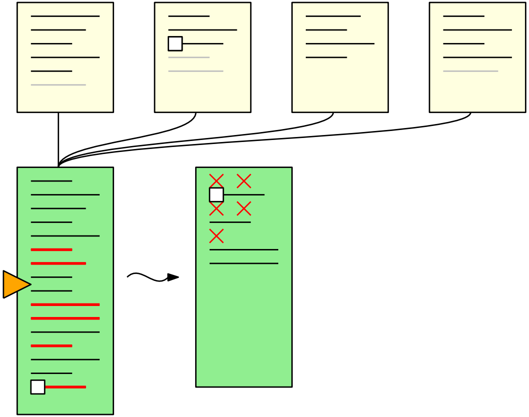


A Sequential Approach

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

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

     $\Rightarrow$ Output addition of c

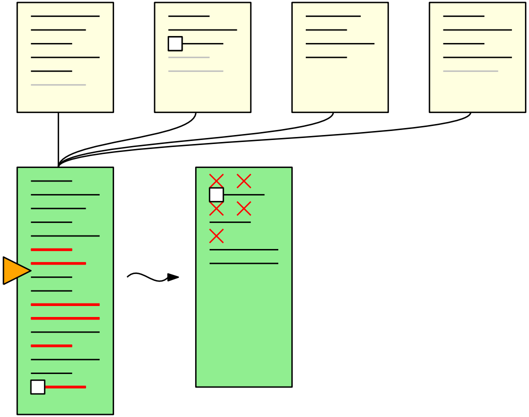


A Sequential Approach

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

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

     ⇒ Output addition of *c*

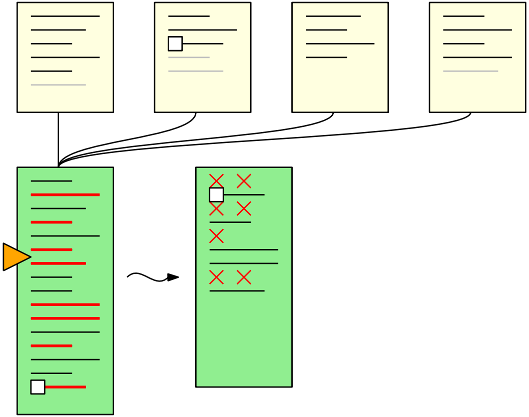


A Sequential Approach

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

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

     ⇒ Output addition of c



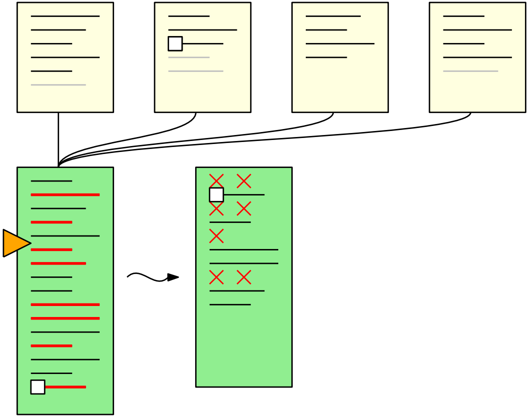
A Sequential Approach

1. Combination

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

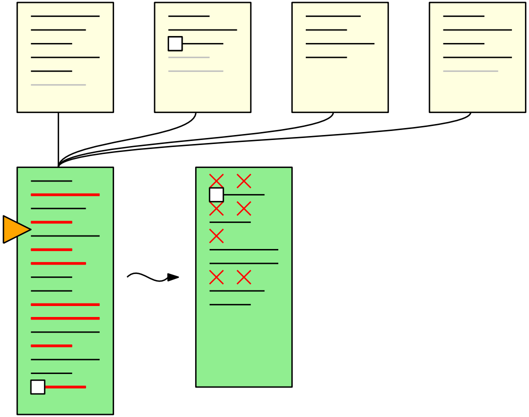
2. Pruning

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



A Sequential Approach

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

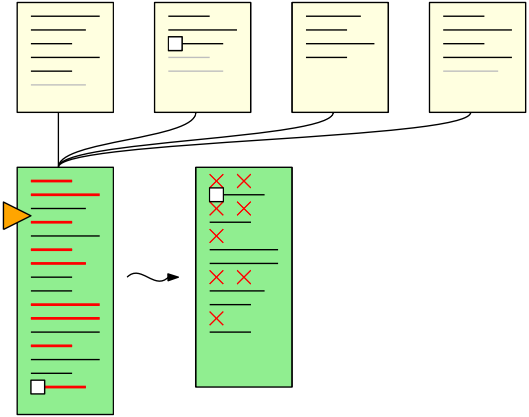


A Sequential Approach

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

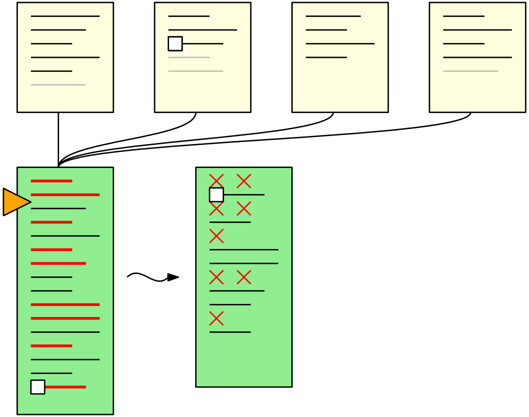
     $\Rightarrow$ For each dependency d of c, $d \notin R$:
     Output deletion of d, add d to R

     $\Rightarrow$ Output addition of c



A Sequential Approach

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

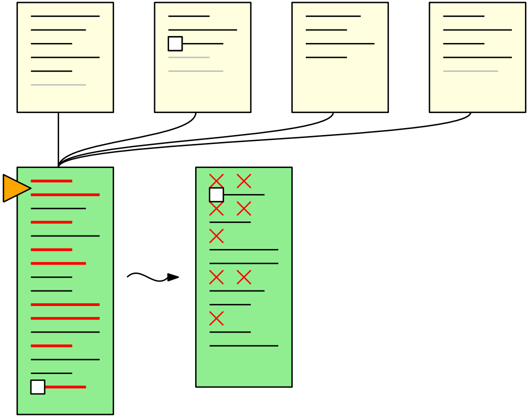


A Sequential Approach

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

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

     $\Rightarrow$ Output addition of c

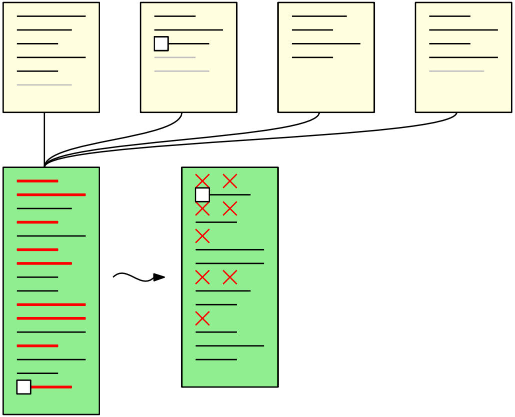


A Sequential Approach

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

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

     $\Rightarrow$ Output addition of *c*

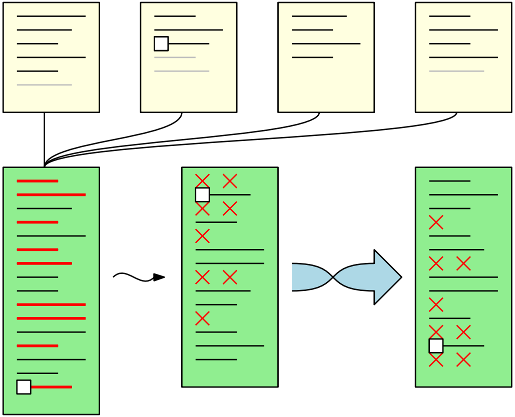


A Sequential Approach

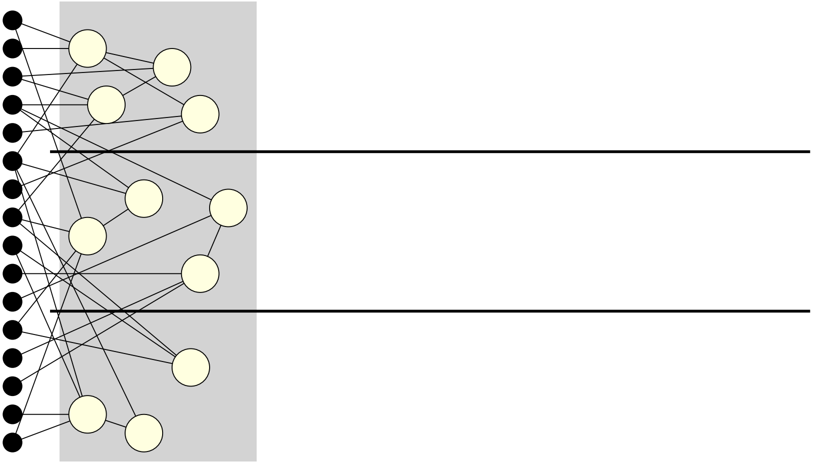
1. **Combination**
   - Read all partial proofs simultaneously
   - Output line ⇔ all dependencies *d* output
2. **Pruning**
   - Required clauses $R := \{id(\Box)\}$
   - Read combined proof from back to front
   - @ Clause *c*: $id(c) \in R$?

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

     ⇒ Output addition of *c*
3. **Reverse lines of pruned proof**

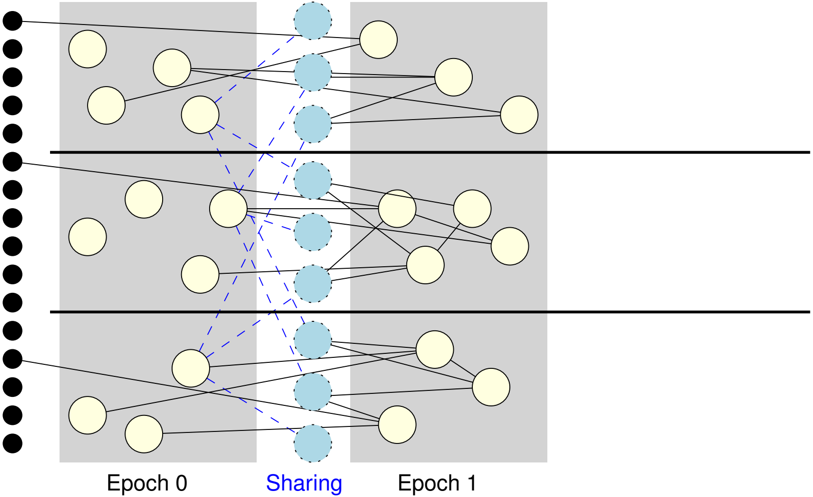


Distributed Pruning: Schematic Overview

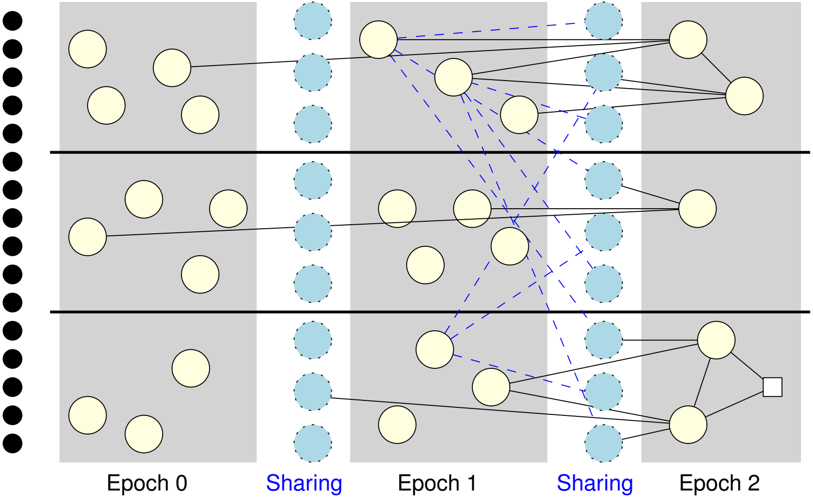


Epoch 0

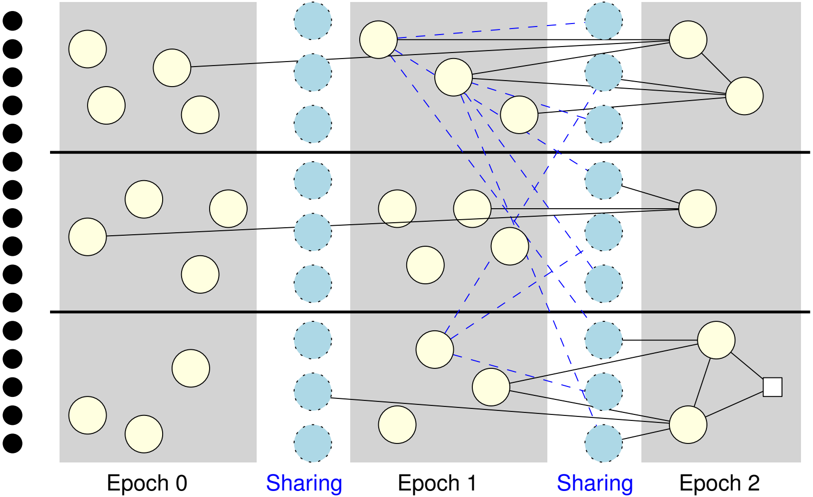
Distributed Pruning: Schematic Overview



Distributed Pruning: Schematic Overview

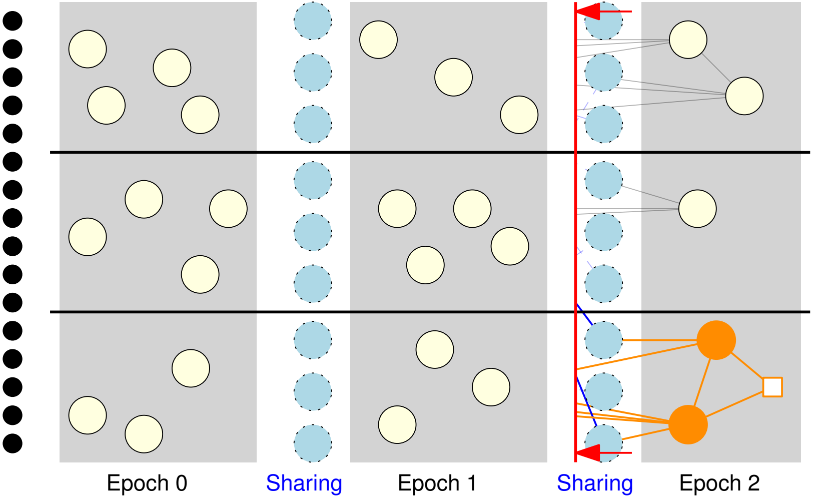


Distributed Pruning: Schematic Overview



First “prune”, then combine!

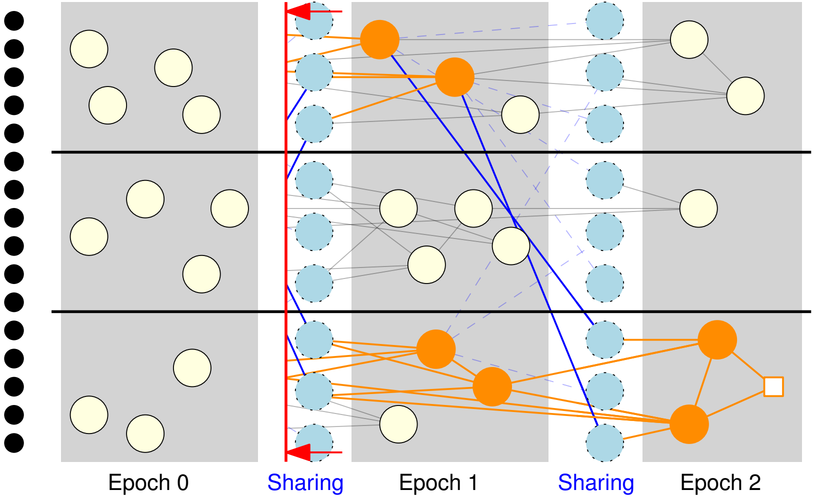
Distributed Pruning: Schematic Overview



First "prune",
then combine!

Trace dependencies
epoch by epoch

Distributed Pruning: Schematic Overview

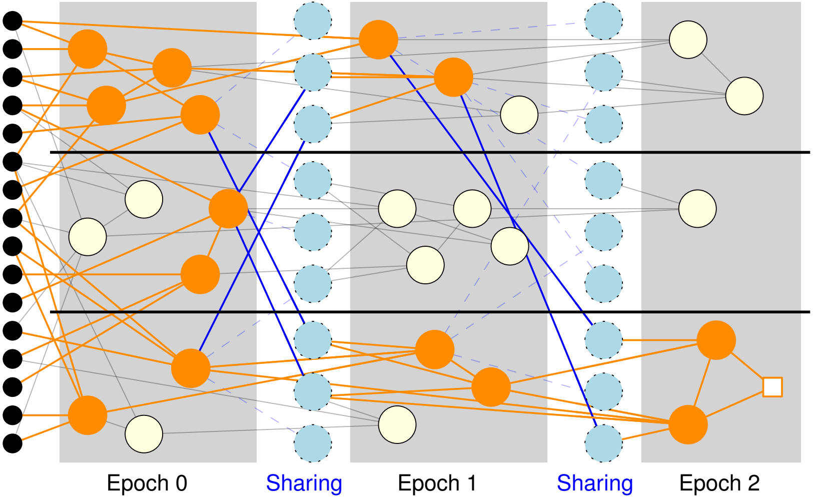


First "prune",
then combine!

Trace dependencies
epoch by epoch

Redistribute remote IDs
at epoch borders

Distributed Pruning: Schematic Overview



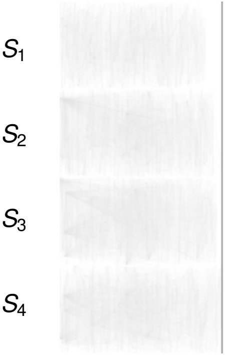
First “prune”,
then combine!

Trace dependencies
epoch by epoch

Redistribute remote IDs
at epoch borders

Distributed Pruning: Real Data

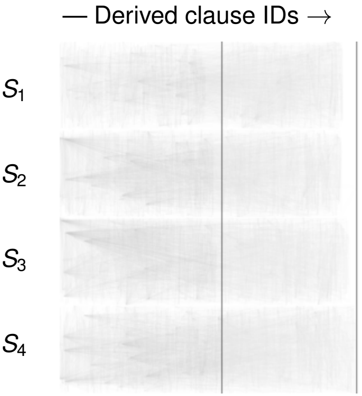
— Derived clause IDs →



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

Solving: Align clause IDs at each sharing epoch

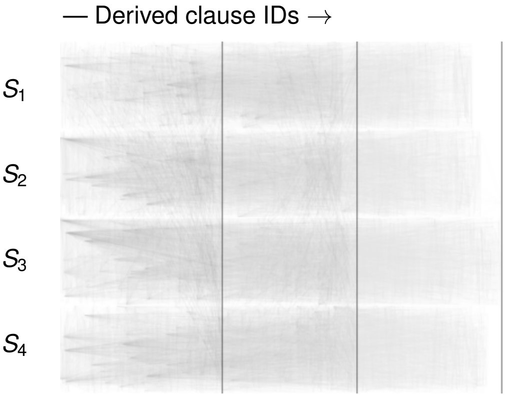
Distributed Pruning: Real Data



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

Solving: Align clause IDs at each sharing epoch

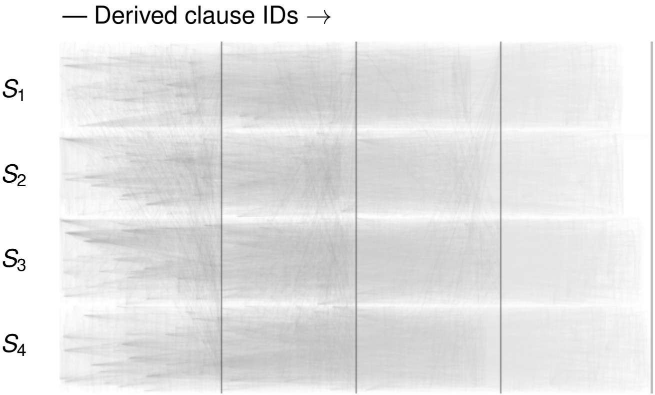
Distributed Pruning: Real Data



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

Solving: Align clause IDs at each sharing epoch

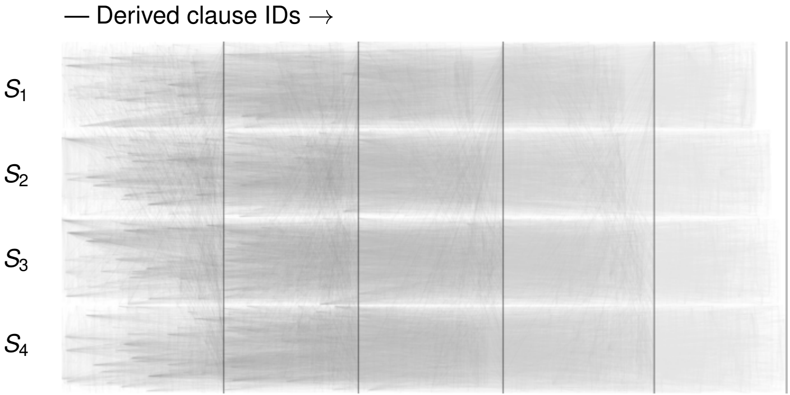
Distributed Pruning: Real Data



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

Solving: Align clause IDs at each sharing epoch

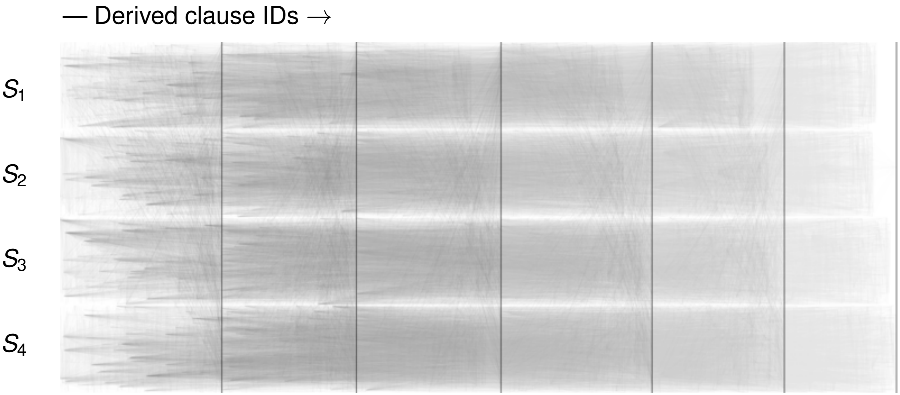
Distributed Pruning: Real Data



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

Solving: Align clause IDs at each sharing epoch

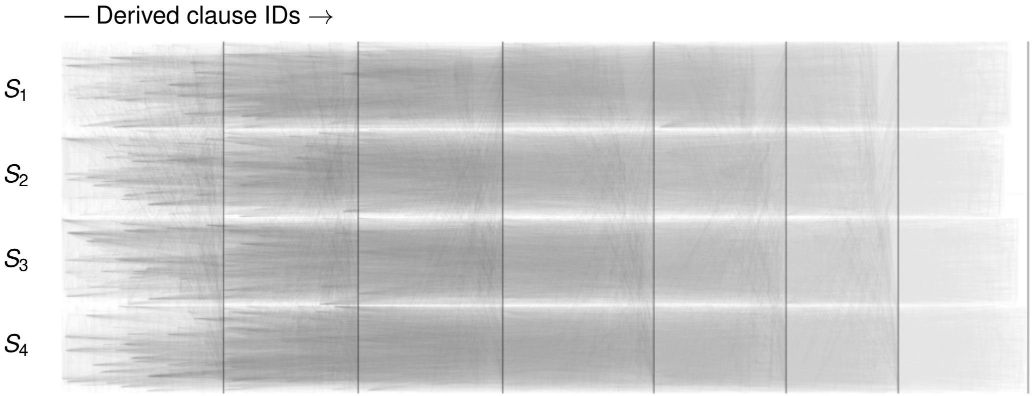
Distributed Pruning: Real Data



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

Solving: Align clause IDs at each sharing epoch

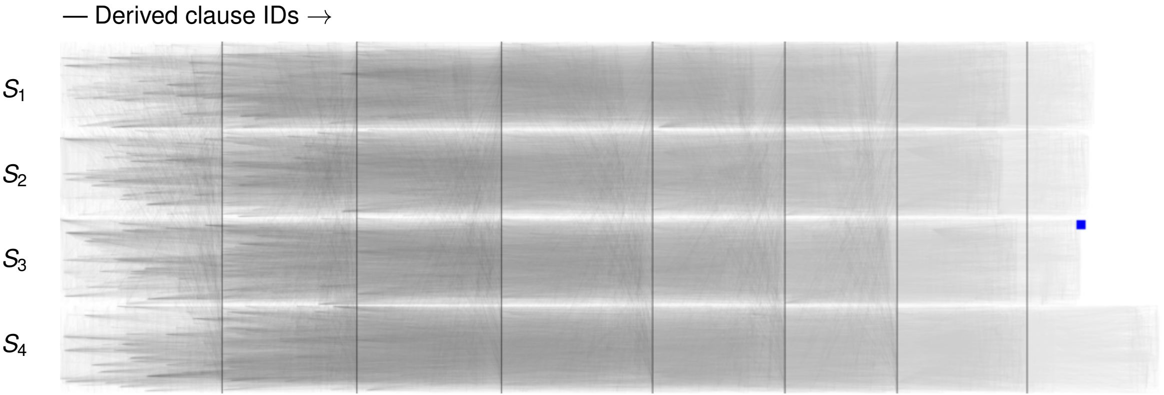
Distributed Pruning: Real Data



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

Solving: Align clause IDs at each sharing epoch

Distributed Pruning: Real Data

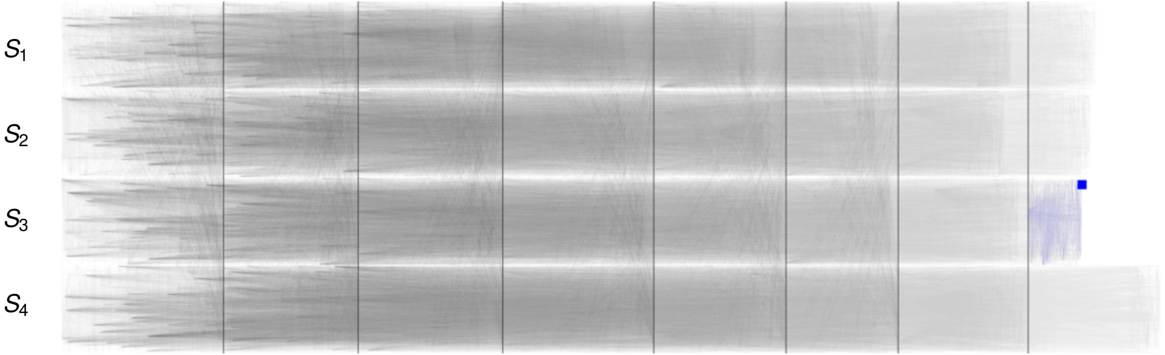


180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

Solving: Align clause IDs at each sharing epoch

Distributed Pruning: Real Data

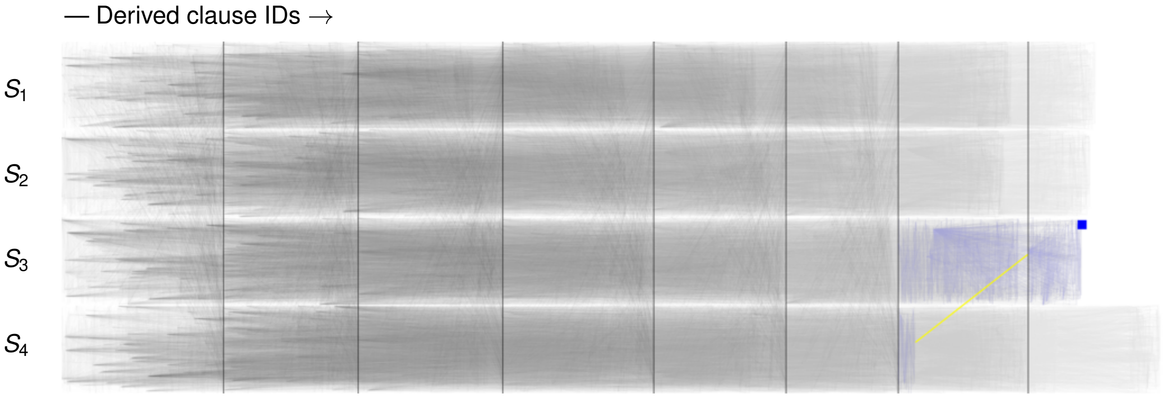
— Derived clause IDs →



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

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

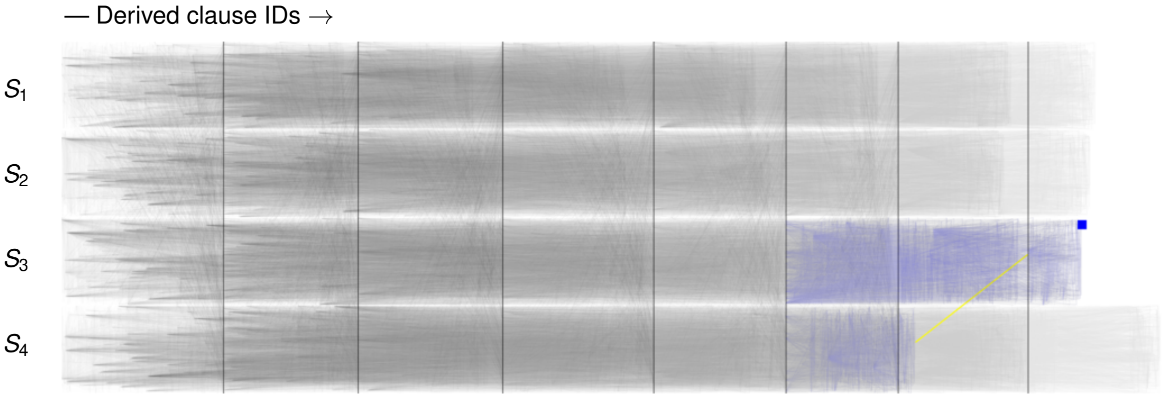
Distributed Pruning: Real Data



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

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

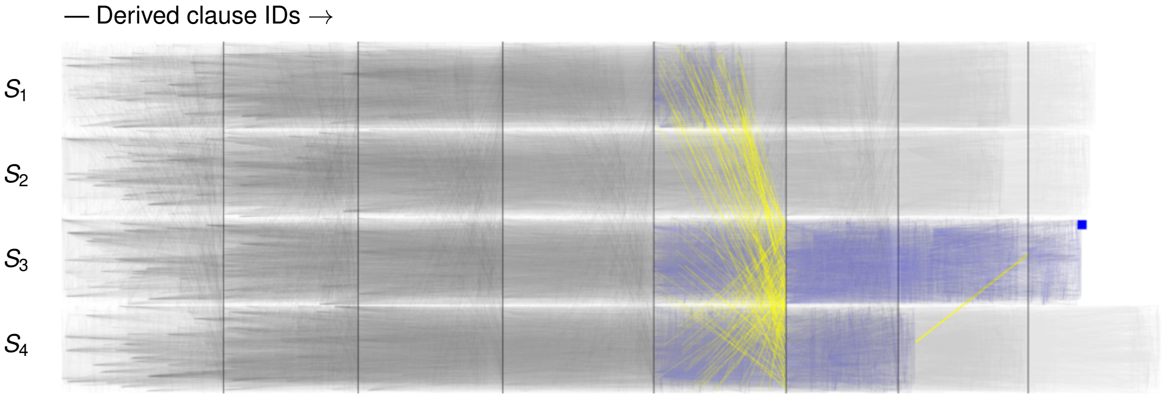
Distributed Pruning: Real Data



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

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

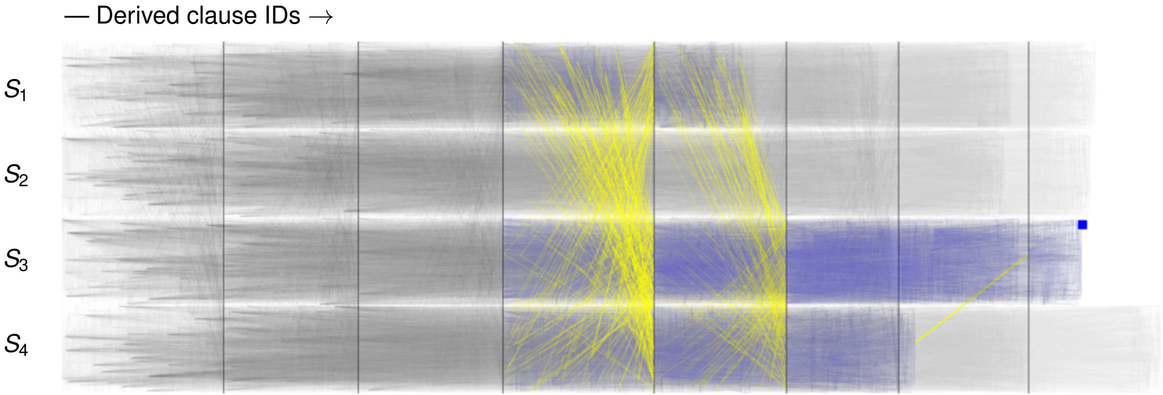
Distributed Pruning: Real Data



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

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

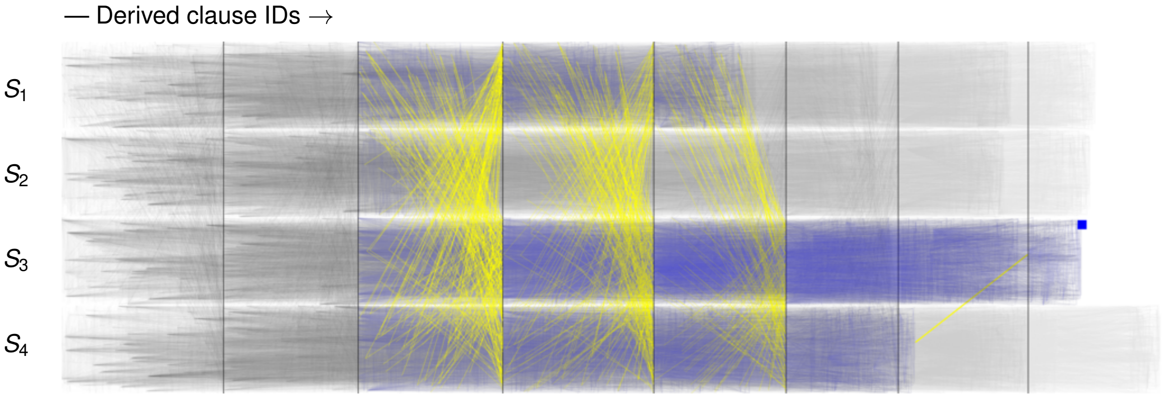
Distributed Pruning: Real Data



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

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

Distributed Pruning: Real Data

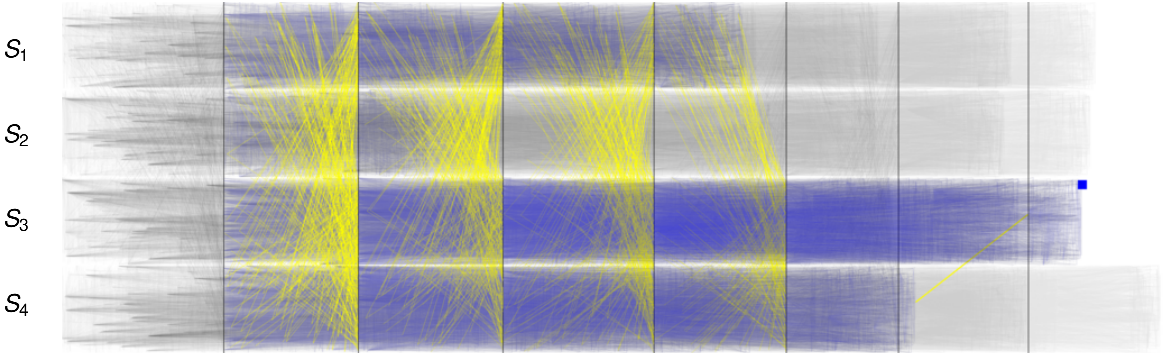


180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

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

Distributed Pruning: Real Data

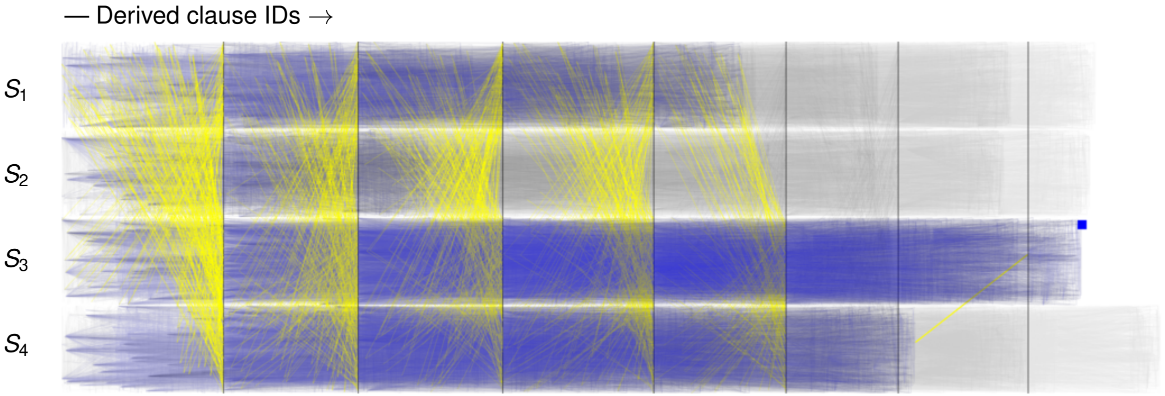
— Derived clause IDs →



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

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

Distributed Pruning: Real Data

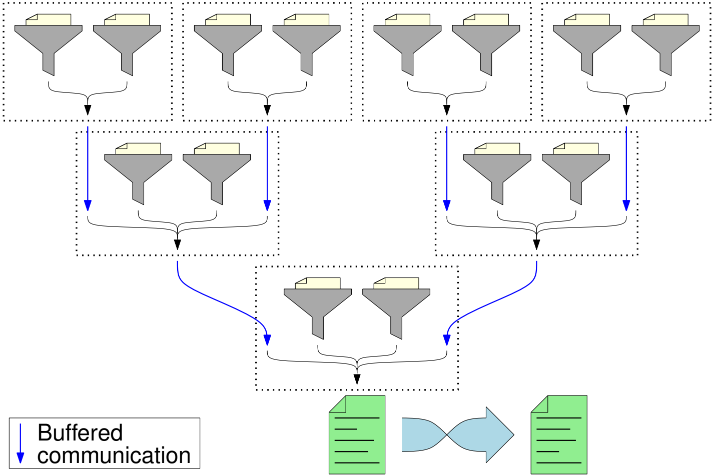


180-variable random 3-SAT formula. 4 notebook cores × 1.7 s. 300k dependencies (orig. clauses omitted).

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

Distributed Combination

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



Experimental Setup (1/2)

Technology

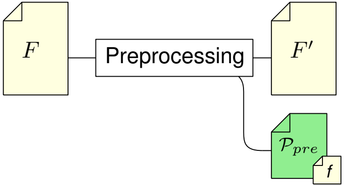
- Base SAT solver: CaDiCaL [Biere 2018] modified to output LRAT, restricted portfolio
- Distributed solver: Mallob [Schreiber+Sanders 2021] extended by clause IDs + proof production
- Proof checking: `lrat-check` from drat-trim tools (M. Heule)

Experimental Setup (1/2)

Technology

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

Pipeline

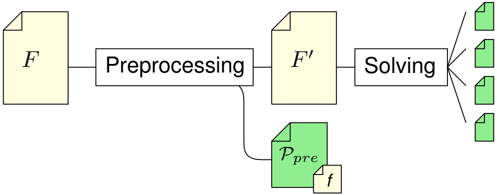


Experimental Setup (1/2)

Technology

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

Pipeline

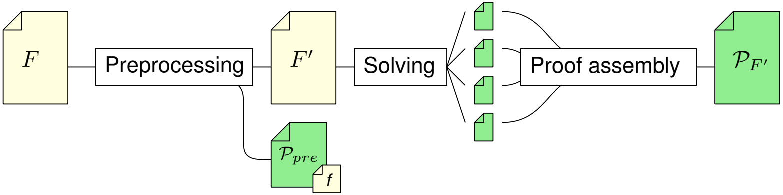


Experimental Setup (1/2)

Technology

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

Pipeline

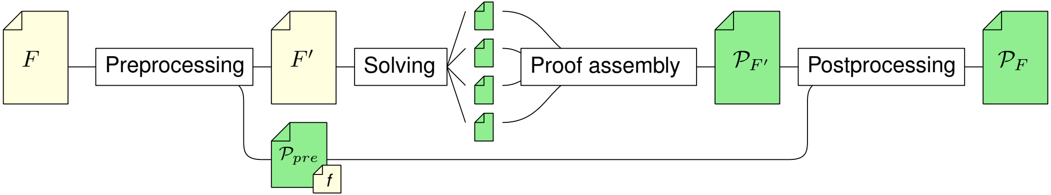


Experimental Setup (1/2)

Technology

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

Pipeline

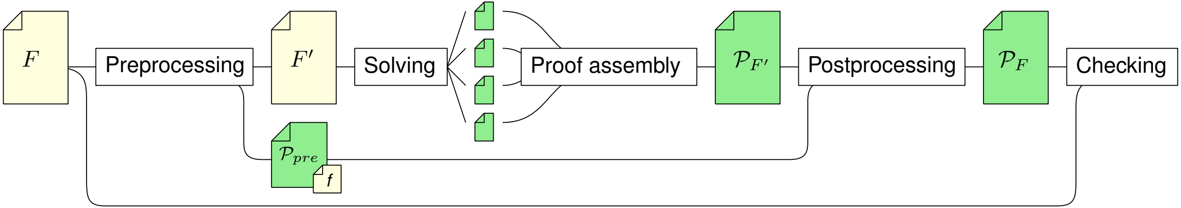


Experimental Setup (1/2)

Technology

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

Pipeline



Experimental Setup (2/2)

Comparison to prior work

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

Experimental Setup (2/2)

Comparison to prior work

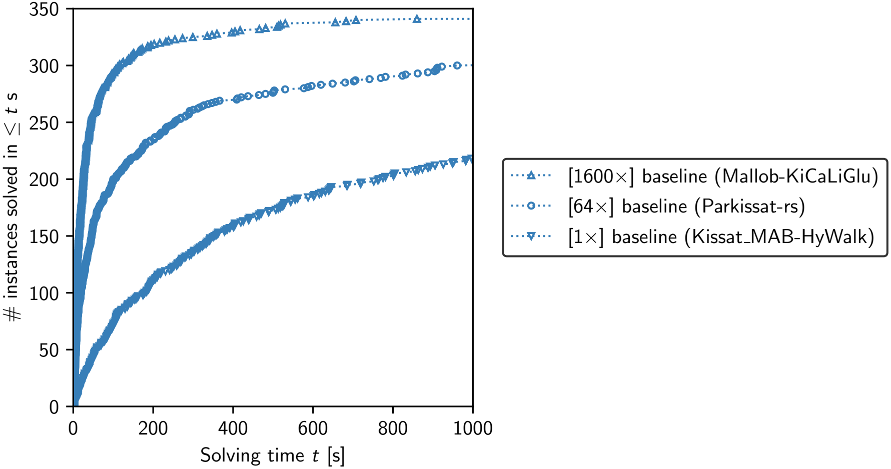
- Shared-memory clause-sharing portfolios: Heule, Manthey, Philipp @ POS'14
  - Synchronized, moderated logging into shared DRAT proof
  - Solver not competitive ⇒ Simulate proof output, compare checking times only
- Sequential SAT solving: `Kissat_MAB-HyWalk` @ SAT Comp. 2022

Resources

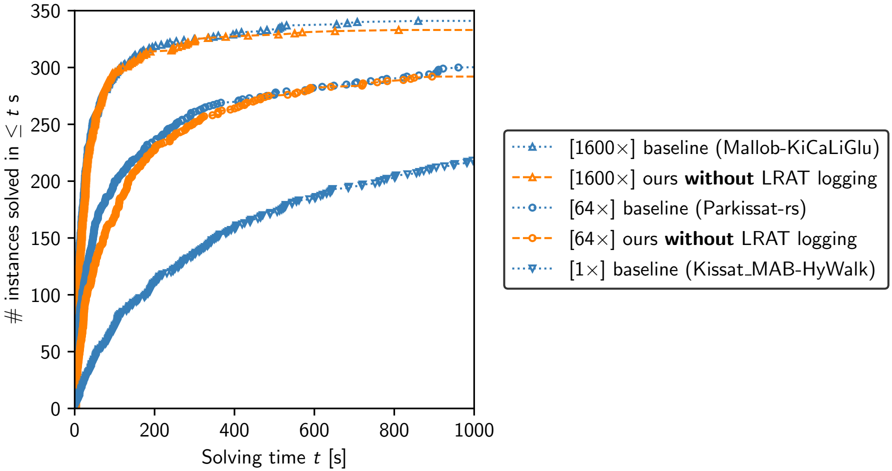
- 1600× setup: 100× `m6i.4xlarge` EC2 instances (16 hwthreads, 64 GB RAM)
- 64× setup: 1× `m6i.16xlarge` EC2 instance (64 hwthreads, 256 GB RAM)
- Sequential setup: One `m6i.4xlarge` EC2 instance

$\leq$ 1000 s solving
$\leq$ 4000 s proof prod.

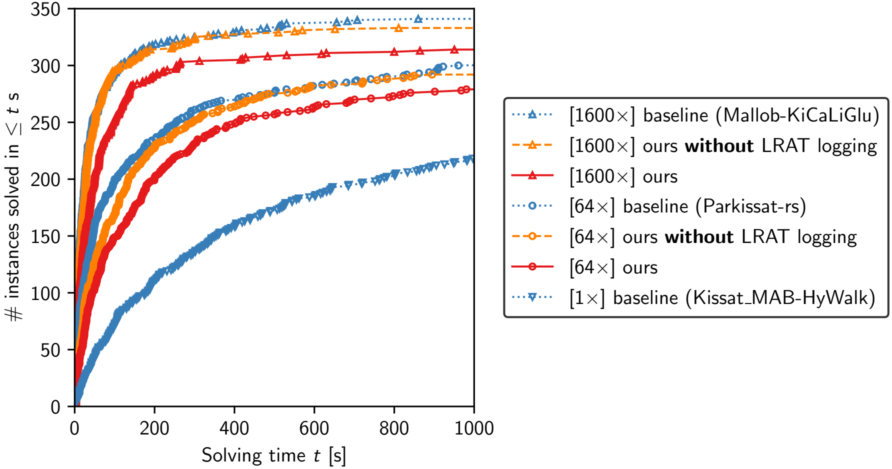
Evaluation: Solving Times



Evaluation: Solving Times

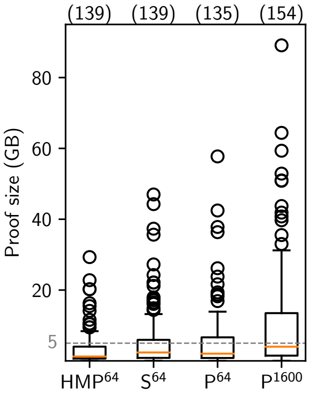


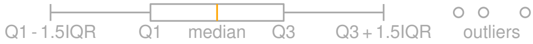
Evaluation: Solving Times



Evaluation: Proof Output

How large are the resulting proofs?

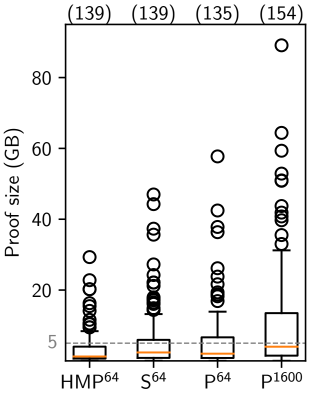




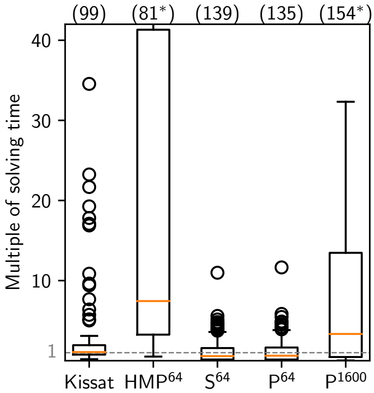
* Some data cut off

Evaluation: Proof Output

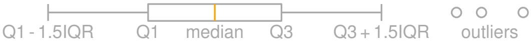
How large are the resulting proofs?



How fast can we check the proofs?

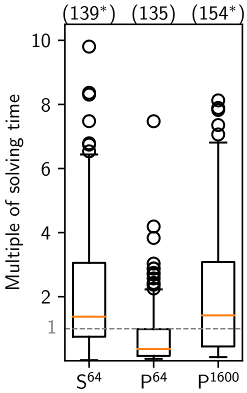


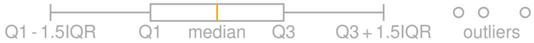
*Some data cut off



Evaluation: Overhead

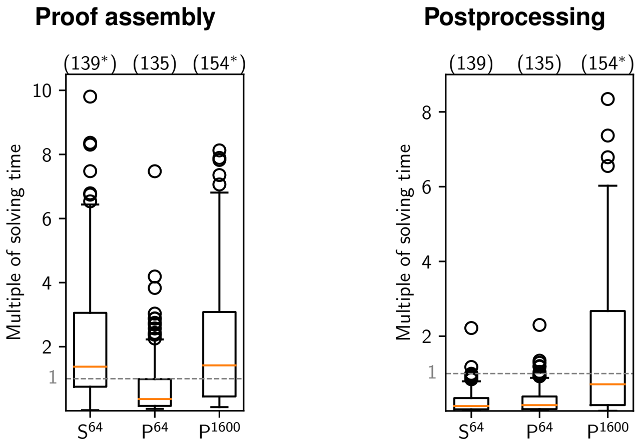
Proof assembly

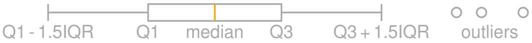




*Some data cut off

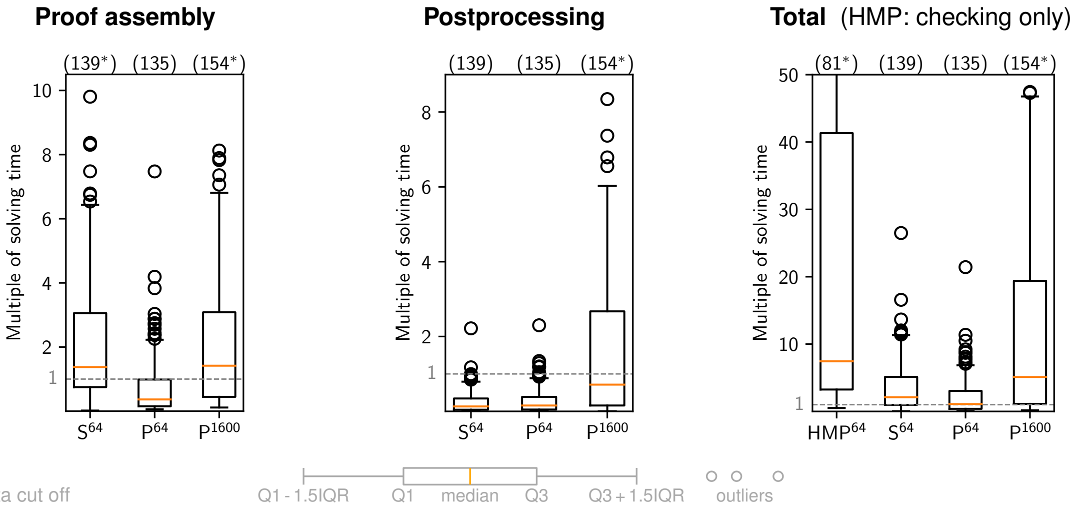
Evaluation: Overhead





*Some data cut off

Evaluation: Overhead



*Some data cut off

Conclusion

- First feasible approach to have distributed clause-sharing solvers produce UNSAT proofs
- Significantly outperform existing proof-producing solvers

Future work

- Reduce overhead — improve LRAT support in SAT backends!
- Proof production in Mallob's scheduled mode?

Distributed Approach

Parallel processing + distributed memory?

Distributed Approach

Parallel processing + distributed memory?

- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order

Distributed Approach

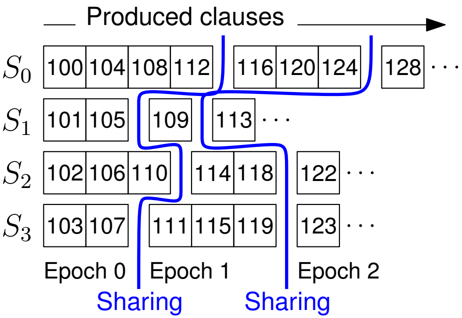
Parallel processing + distributed memory?

- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin

Distributed Approach

Parallel processing + distributed memory?

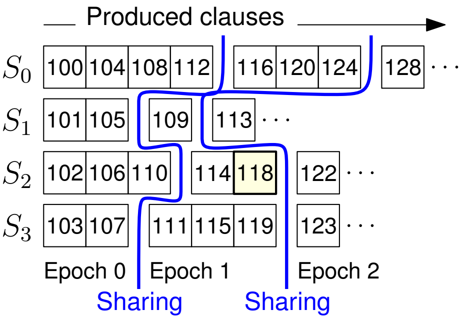
- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies



Distributed Approach

Parallel processing + distributed memory?

- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies



Produced clauses

S_0 100 104 108 112 116 120 124 128 · · ·

S_1 101 105 109 113 · · ·

S_2 102 106 110 114 118 122 · · ·

S_3 103 107 111 115 119 123 · · ·

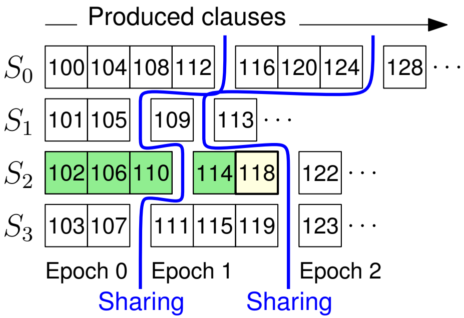
Epoch 0 Epoch 1 Epoch 2

Sharing Sharing

Distributed Approach

Parallel processing + distributed memory?

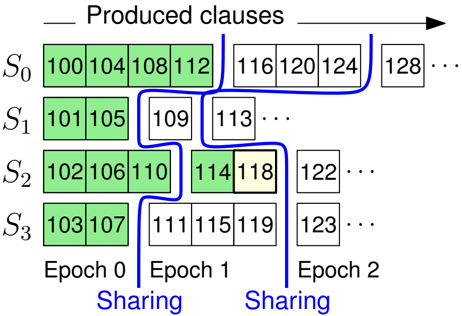
- Trace dependencies of "□" by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies



Distributed Approach

Parallel processing + distributed memory?

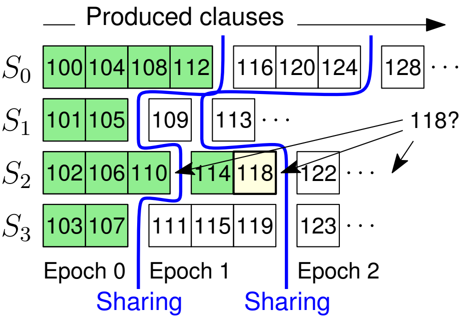
- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies



Distributed Approach

Parallel processing + distributed memory?

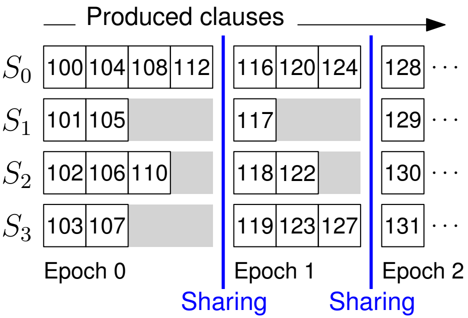
- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies



Distributed Approach

Parallel processing + distributed memory?

- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies
- Align clause IDs to find out when to redistribute them



Produced clauses

	Epoch 0	Epoch 1	Epoch 2
S_0	100 104 108 112	116 120 124	128 ···
S_1	101 105	117	129 ···
S_2	102 106 110	118 122	130 ···
S_3	103 107	119 123 127	131 ···

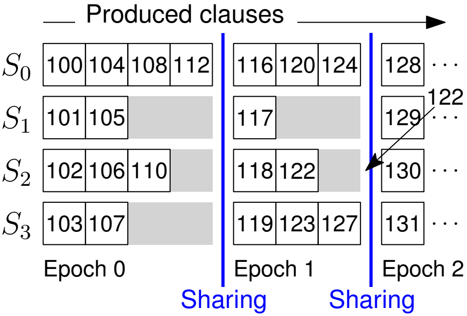
Sharing

Sharing

Distributed Approach

Parallel processing + distributed memory?

- Trace dependencies of “□” by scanning all partial proofs in reverse chronological order
- Redistribute remote required clause IDs to their origin
- Exploit structure of clause sharing to unroll dependencies
- Align clause IDs to find out when to redistribute them



Produced clauses

S_0 100 104 108 112 | 116 120 124 | 128 · · ·

S_1 101 105 | 117 | 129 · · ·

S_2 102 106 110 | 118 122 | 130 · · ·

S_3 103 107 | 119 123 127 | 131 · · ·

Epoch 0 Epoch 1 Epoch 2

Sharing Sharing

Distributed Approach

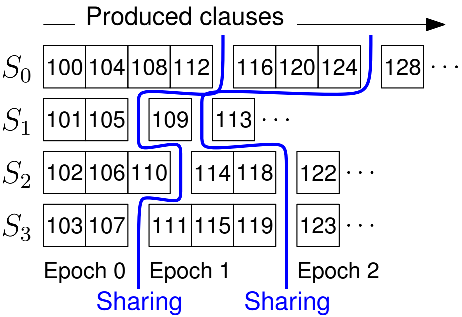
Parallel processing + distributed memory?

- Prune all partial proofs in parallel, **then** combine — read each partial proof only once!

Distributed Approach

Parallel processing + distributed memory?

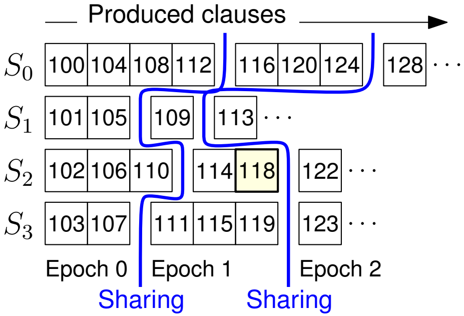
- Prune all partial proofs in parallel, **then** combine — read each partial proof only once!



Distributed Approach

Parallel processing + distributed memory?

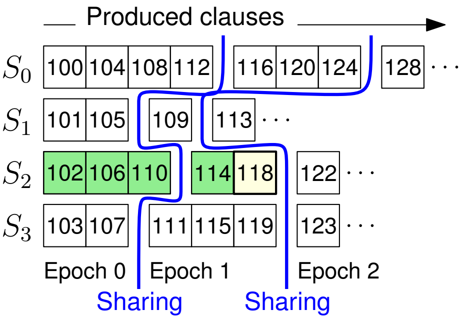
- Prune all partial proofs in parallel, **then** combine — read each partial proof only once!



Distributed Approach

Parallel processing + distributed memory?

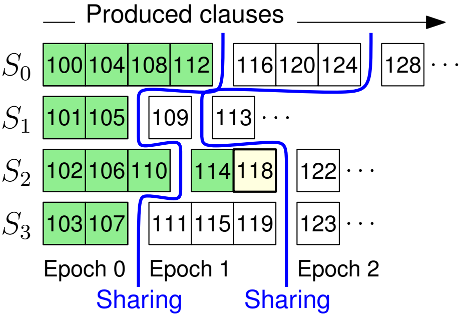
- Prune all partial proofs in parallel, **then** combine — read each partial proof only once!



Distributed Approach

Parallel processing + distributed memory?

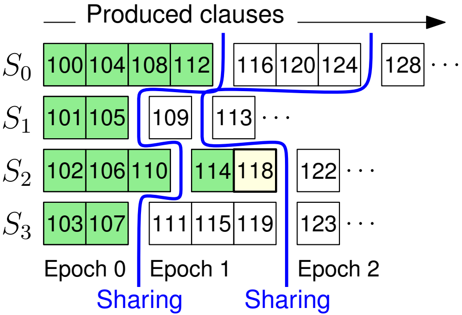
- Prune all partial proofs in parallel, **then** combine — read each partial proof only once!



Distributed Approach

Parallel processing + distributed memory?

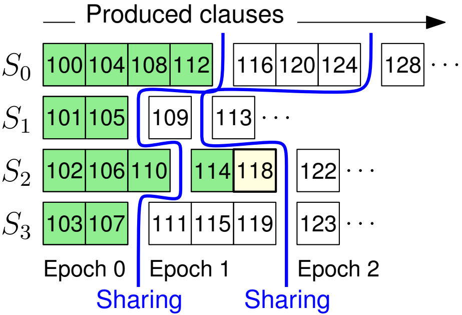
- Prune all partial proofs in parallel, **then** combine — read each partial proof only once!
- Unroll needed dependencies epoch by epoch in reverse chronological order



Distributed Approach

Parallel processing + distributed memory?

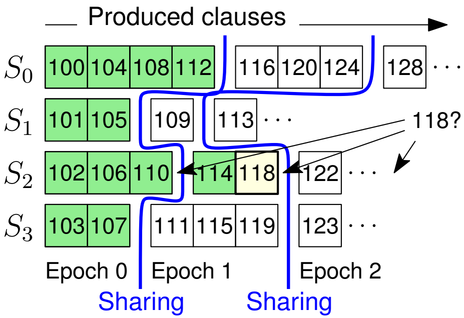
- Prune all partial proofs in parallel, **then** combine
  — read each partial proof only once!
- Unroll needed dependencies epoch by epoch
  in reverse chronological order
- Redistribute each required clause to its producer
  — redistribute each ID only once!
  — just before processing its originating epoch!



Distributed Approach

Parallel processing + distributed memory?

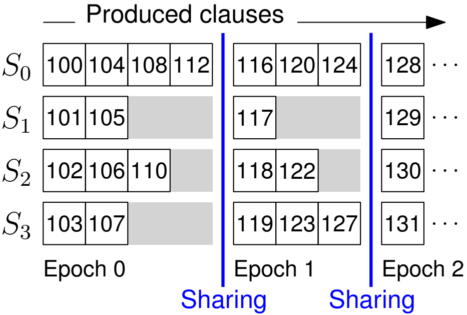
- Prune all partial proofs in parallel, **then** combine
  — read each partial proof only once!
- Unroll needed dependencies epoch by epoch
  in reverse chronological order
- Redistribute each required clause to its producer
  — redistribute each ID only once!
  — just before processing its originating epoch!



Distributed Approach

Parallel processing + distributed memory?

- Prune all partial proofs in parallel, **then** combine
  — read each partial proof only once!
- Unroll needed dependencies epoch by epoch
  in reverse chronological order
- Redistribute each required clause to its producer
  — redistribute each ID only once!
  — just before processing its originating epoch!
- Align clause IDs to efficiently find epoch of a clause

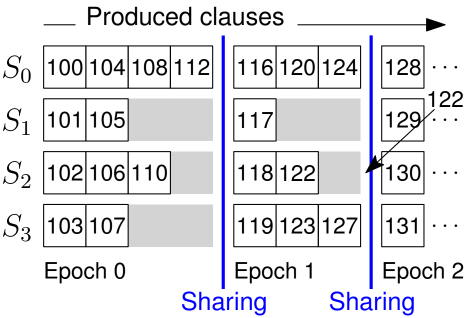


	Epoch 0	Epoch 1	Epoch 2
S_0	100 104 108 112	116 120 124	128 · · ·
S_1	101 105	117	129 · · ·
S_2	102 106 110	118 122	130 · · ·
S_3	103 107	119 123 127	131 · · ·

Distributed Approach

Parallel processing + distributed memory?

- Prune all partial proofs in parallel, **then** combine
  — read each partial proof only once!
- Unroll needed dependencies epoch by epoch
  in reverse chronological order
- Redistribute each required clause to its producer
  — redistribute each ID only once!
  — just before processing its originating epoch!
- Align clause IDs to efficiently find epoch of a clause



	Epoch 0	Epoch 1	Epoch 2
S_0	100 104 108 112	116 120 124	128 ···
S_1	101 105	117	129 ···
S_2	102 106 110	118 122	130 ···
S_3	103 107	119 123 127	131 ···

Distributed Pruning: Details

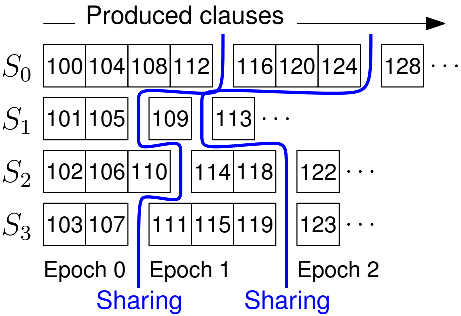
- Transfer each remote required clause ID once!

  — Defer redistribution until epoch of origin

  — Detect duplicates while aggregating clause IDs

Distributed Pruning: Details

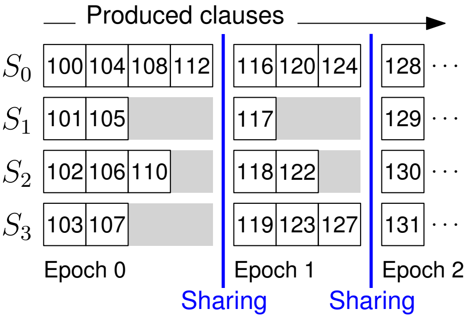
- Transfer each remote required clause ID once!
  - — Defer redistribution until epoch of origin
  - — Detect duplicates while aggregating clause IDs
- How to find epoch of origin of a remote clause ID?



Produced clauses
S_0 100 104 108 112 116 120 124 128 · · ·
S_1 101 105 109 113 · · ·
S_2 102 106 110 114 118 122 · · ·
S_3 103 107 111 115 119 123 · · ·
Epoch 0
Epoch 1
Epoch 2
Sharing
Sharing

Distributed Pruning: Details

- Transfer each remote required clause ID once!
  - — Defer redistribution until epoch of origin
  - — Detect duplicates while aggregating clause IDs
- How to find epoch of origin of a remote clause ID?
  - — Align clause IDs at epoch borders during solving

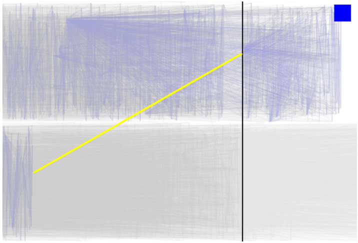


	Epoch 0	Epoch 1	Epoch 2
S_0	100 104 108 112	116 120 124	128 ···
S_1	101 105	117	129 ···
S_2	102 106 110	118 122	130 ···
S_3	103 107	119 123 127	131 ···

Rewind: Realization

Local Processing

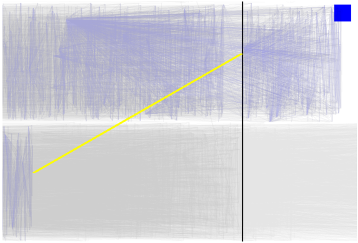
- For each S_i: Frontier F_i of req. clause IDs *produced by* S_i; Backlog B_i of *remote* req. clause IDs
  - External-memory priority queues partitioned by epoch
- Epoch e: Process proof parts from ep. e
- Clause c with $id(c) \in F_i$: Insert each $d \in deps(c)$ into F_i or B_i



Rewind: Realization

Local Processing

- For each S_i: Frontier F_i of req. clause IDs *produced by* S_i; Backlog B_i of *remote* req. clause IDs
  - External-memory priority queues partitioned by epoch
- Epoch e: Process proof parts from ep. e
- Clause c with $id(c) \in F_i$: Insert each $d \in deps(c)$ into F_i or B_i



Redistribution of Clause IDs

- After processing epoch e: Extract IDs from ep. $e - 1$ from all B_i
- All-reduction like Mallob's clause sharing, detecting duplicate IDs
- Strictly less communication than during solving

Experimental Setup (1/2)

Technology

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

Experimental Setup (1/2)

Technology

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

Pipeline

1. *Parallel* solving (→ partial proofs)
2. *Sequential or parallel* proof assembly
3. *Sequential* postprocessing of assembled proof
4. *Sequential* proof checking