Lecture Outline

- Preprocessing and Inprocessing
- Parallel SAT Solving
Preprocessing

- General idea: try to reduce the input formula by (polynomial time) simplification procedures.
- Assumption: smaller problems are easier to solve
Self-Subsuming Resolution

**Definition: Subsumption**

A clause $D$ subsumes a clause $C$ iff $D \subseteq C$. We also say that clause $C$ is subsumed by $D$.

To check satisfiability, subsumed clauses are irrelevant.

**Definition: Self-Subsuming Resolution**

Let $C$, $D$ be clauses and $\otimes_x$ the resolution operator on variable $x$. If $C \otimes D \subseteq C$ then $C$ is said to be self-subsumed by $D$ with respect to $x$.

Example: $\{\neg b, \neg e, f, \neg h\}$ is self-subsumed by $\{\neg b, \neg e, \neg f\}$ w.r.t. $f$. If $C$ is self-subsumed by $D$, $C$ can be replaced by $C \otimes D$. I.e. the learned clause in the 1UIP example can be strengthened to $\{\neg b, \neg e, \neg h\}$.
Variable Elimination

Elimination by Clause Distribution

For a CNF $F$, let $S_x$ and $S_{\overline{x}}$ be the sets of clauses containing $x$ resp. $\overline{x}$, and $S = S_x \cup S_{\overline{x}}$. Let $S_x \otimes S_{\overline{x}} = \{ C \otimes D \mid C \in S_x, D \in S_{\overline{x}} \}$. Replacing $S$ by $S_x \otimes S_{\overline{x}}$ in $F$ is called elimination by clause distribution.

The formulas $F$ and $F'$, where $S$ is replaced by $S' = S_x \otimes S_{\overline{x}}$ are satisfiability equivalent.

Variable elimination procedure:

- Apply “elimination by clause distribution” to all variables, but replace $S$ by $S'$ only if number of clauses decreases
- Do not count tautological resolvents

Variable elimination by clause distribution is also called “Bounded Variable Elimination” (BVE). It is the most important simplification technique today.
**Variable Elimination and Gates**

- Gates occur frequently when encoding hardware circuits into CNF.
- For example, the gate $x = \text{AND}(y, z)$ results in the clauses $\{\neg x, y\}, \{\neg x, z\}, \{x, \neg y, \neg z\}$.
- Resolving the clauses of a gate results in tautological clauses.
- Idea: detect a gate, split formula $F$ into $F = G \cup R$, where $G$ are the gate-clauses and $R$ the remaining clauses.
- Apply elimination by clause distribution: $S = (G_x \cup R_x) \cup (G_{\neg x} \cup R_{\neg x})$.
- Clause distribution results in $S' = (G_x \otimes R_{\neg x}) \cup (R_x \otimes G_{\neg x}) \cup (R_x \otimes R_{\neg x})$.
- Moreover, $(G_x \otimes R_{\neg x}) \cup (R_x \otimes G_{\neg x}) \models (R_x \otimes R_{\neg x})$. (Why?)
- Thus, we can replace $S$ by the satisfiability-equivalent $(R_x \otimes G_{\neg x}) \cup (R_{\neg x} \otimes G_x)$.
Failed Literal Elimination

- If applying unit propagation on $F \land \{l\}$ derives UNSAT ($F \land \{l\} \vdash_{UP} \bot$), replace $F$ by $F \land \{\neg l\}$.

- Generalization: If $(F \setminus \{C\}) \land \neg C \vdash_{UP} \bot$, remove $C$ from $F$. 
Blocked Clauses

**Definition**

A clause \((l \lor C)\) is blocked w.r.t. \(F\) by \(l\) if for every clause \((\neg l \lor D)\) in \(F\) the resolvent \((C \lor D)\) is a tautology.

Example: \(F = (a \lor b) \land (a \lor \neg b \lor \neg c) \land (\neg a \lor c)\).
First clause is not blocked, second is blocked by both \(a\) and \(\neg c\), third is blocked by \(c\).

Removal of an arbitrary blocked clause preserves satisfiability. Blocked clause elimination (BCE) has a unique fixpoint.
Autarkies

- Autarkies are a generalization of pure literals

**Autarky**

Given a partial assignment $\sigma$ and a formula $F$, a clause (of $F$) is *touched by* $\sigma$ if it contains the negation of a literal assigned in $\sigma$. A clause is *satisfied by* $\sigma$ if it contains a literal assigned to *True* by $\sigma$. If all touched clauses are satisfied then $\sigma$ is an *autarky*.

- All clauses touched by an autarky can be removed
- Example: $\{\neg a, b\}, \{\neg a, c\}, \{a, \neg b, \neg c\}, \{b, d\}, \ldots$ (more clauses without $a$ and $c$)
- Then $\sigma = \{\neg a, \neg c\}$ is an autarky.
Preprocessing Techniques that do not Reduce the Problem Size

- There are techniques, such as Bounded Variable Addition (BVA), that increase problem size.
- BVA is mainly based on the Extended Resolution rule.

**Extended Resolution**

Extended resolution adds a second rule to the resolution calculus, the Extension Rule. The idea is to introduce new variables as conjunction of existing literals, $x_{\text{new}} \leftrightarrow l_1 \land l_2$. As a rule for formulas in CNF:

$$
(\neg x_{\text{new}} \lor l_1) \land (\neg x_{\text{new}} \lor l_2) \land (x_{\text{new}} \lor \neg l_1 \lor \neg l_2)
$$

- There are proofs with exponential size by Resolution, but polynomial size by Extended Resolution, e.g. pigeonhole formulas
Inprocessing: Idea

- Idea: Interleave search and preprocessing
- Preprocessing can be extremely beneficial
  - Most solvers in the SAT competitions use variable elimination
  - Equivalence / XOR reasoning
  - Failed literal elimination
- Many preprocessing techniques, though polynomial, require considerable time
Inprocessing

- “Preempt” (interrupt) preprocessing techniques after some time
- Resume preprocessing between restarts
- Limit preprocessing time in relation to search
Three kinds of approaches

- Divide and Conquer – explicit search space partitioning
- Cube and Conquer – implicit load balancing
- Diversify and Conquer – portfolio search
1994 First parallel implementation of DPLL

- completely distributed (no master and slave roles)
- A list of partial assignment is generated
- Each processors receives the entire formula and a few partial assignments
- Each Processors consists of
  - Worker (solve or split the formula, use the partial assignments)
  - Balancer (estimate workload, communicate, stopping)

- If a worker has nothing to do (all its partial assignments lead to UNSAT) a balancing process is launched.
Centralized master-slave architecture
Communication only between master and slaves
Master assigns partial assignments based on the *Guiding Path*
  - Each node in the search tree is open or closed (closed means one branch is explored)
  - Master splits the open nodes and assigns job to slaves
All processors can get stuck on unpromising branches
Guiding Path Example

```
(1, 0, 0), (X3, 1, 0), (X4, 1, 1), (X5, 0, 0)
```

*** : explored branch
+ : current node
? : remaining subtree
The solver *Satz* improves PSATO the by adding *work stealing* for workload balancing:
- An idle slave requests work from the master.
- The master splits the work of the most loaded slave.
- The idle slave and most loaded slave get the parts.
1996 – Clause learning invented

I FREAKING LOVE

LEARNING!!!!
Clause Sharing Parallel Solvers

- 2001, Blochinger et al.: PaSAT – the first parallel DPLL with "intelligent backtracking" and clause sharing
  - Similar to PSATO and SATZ: master slave, guiding path, randomized work stealing
- 2004, Feldman et al. – the first shared memory parallel solver
  - Multi-core processors started to be popular
  - Uses same techniques as the previous solvers (guiding path etc.)
  - Bad performance explained by high number of cache misses (DPLL/CDCL is otherwise highly optimized for cache)
- ... and many many more similar solvers
Cube and Conquer

Basic Idea

Generate a large amount of partial assignments (millions) and then assign each to one of the slaves.

- it is unlikely that any of the slaves will run out tasks
- The partial assignments are usually generated using a look-ahead solver (breadth-first search up to a limited depth)
- Examples of such solvers
  - march (Heule) + iLingeling (Biere) introduced the idea in 2011
  - Treengeling (Biere) – still state of the art for combinatorial problems
  - This kind of solver was used in the 200TB proof
Pure Portfolios

Basic Idea
Each processor works on the entire problem (no partial assignment restrictions). Each processor uses a (slightly) different solver (different heuristics, random seeds, etc.) All processors stop when one solver solves the problem.

- PPfolio – winner of Parallel Track in the 2011 SAT Competition
- It is just a bash script that combines the best solvers from the 2010 Competition
- The author: “it’s probably the laziest and most stupid solver ever written, which does not even parse the CNF and knows nothing about the clauses”
- This kind of solvers is not allowed since then in SAT Competitions
Portfolios with Clause Learning

- Same as pure portfolio but clauses are shared
- Usually the same solver with different parameters is used for each processor
- 2009, Hamadi et al.\textsuperscript{1}: ManySAT – the first solver using this idea (based on MiniSat)

\textsuperscript{1}Microsoft® Research
Portfolios with Clause Learning

- Same as pure portfolio but clauses are shared
- Usually the same solver with different parameters is used for each processor
- 2009, Hamadi et al.\(^1\): ManySAT – the first solver using this idea (based on MiniSat)

This is most successful approach since then

\(^1\)Microsoft\textregistered\ Research
What Makes a Good Portfolio Solver

Two Pillars of Portfolios:

**Diversification**
- The search space of the solvers should not overlap too much
- Use different configuration values of heuristic parameters
- Partial assignment recommendations (no restrictions!)

**Clause Sharing**
- Which clauses to share?
- How many?
- How often?
- How to implement efficiently?
Experiments – Random Satisf. 3-SAT

Satisfiable Instances

Time in seconds

Problems

No Diversification, No Sharing
Only Sharing
Only Diversification
Diversification and Sharing
Advice for Satisfiable problems

DIVERSIFY

YOUR PORTFOLIO

memegenerator.net
Experiments – Random Unsat. 3-SAT

Unsatisfiable Instances

No Diversification, No Sharing
Only Sharing
Only Diversification
Diversification and Sharing

Problems

Time in seconds

Clause sharing is important for UNSAT
Clause sharing is important for UNSAT
A recent portfolio implementation

- HordeSAT – a Massively Parallel SAT Solver
- A scalable SAT solver for up to 2048 processors
HordeSAT Design Principles

- Modular Design
  - blackbox approach to SAT solvers
  - any solver implementing a simple interface can be used

- Decentralization
  - all nodes are equivalent, no central/master nodes

- Overlapping Search and Communication
  - search procedure (SAT solver) never waits for clause exchange
  - at the expense of losing some shared clauses

- Hierarchical Parallelization
  - running on clusters of multi-cpu nodes
  - shared memory inter-node clause sharing
  - message passing between nodes
Modular Design

Portfolio Solver Interface

```cpp
void addClause(vector<int> clause);
SatResult solve(); // SAT, UNSAT, UNKNOWN
void setSolverInterrupt();
void unsetSolverInterrupt();
void setPhase(int var, bool phase);
void diversify(int rank, int size);
void addLearnedClause(vector<int> clause);
void setLearnedClauseCallback(LCCallback* clb);
void increaseClauseProduction();
```

- Lingeling implementation with just glue code
- MiniSat implementation, small modification for learned clause stuff
Diversification

Setting Phases – "void setPhase(int var, bool phase)"
- Random – each variable random phase on each node
- Sparse – each variable random phase on exactly one node
- Sparse Random – each variable random phase with prob. $\frac{1}{\#\text{solvers}}$

Native Diversification – "void diversify(int rank, int size)"
- Each solver implements in its own way
- Example: random seed, restart/decision heuristic
- For lingeling we used plingeling diversification

Best is to use Sparse Random together with Native Diversification.
Clause Sharing

Regular (every 1 second) collective all-to-all clause exchange

Exporting Clauses
- Duplicate clauses filtered using Bloom filters
- Clause stored in a fixed buffer, when full clauses are discarded, when underfilled solvers are asked to produce more clauses
- Shorter clauses are preferred
- Concurrent Access – clauses are discarded

Importing Clauses
- Filtering duplicate clauses (Bloom filter)
  - Bloom filters are regularly cleared – the same clauses can be imported after some time
  - Useful since solvers seem to "forget" important clauses
Overall Algorithm

The Same Code for Each Process

SolveFormula(F, rank, size)
  for i = 1 to numThreads do
    s[i] = new PortfolioSolver(Lingeling);
    s[i].addClauses(F);
    diversify(s[i], rank, size);
    new Thread(s[i].solve());

forever do
  sleep(1) // 1 second
  if (anySolverFinished) break;
  exchangeLearnedClauses(s, rank, size);
Experiments – SAT 2011+2014

Lingeling
1x4x4
2x4x4
4x4x4
8x4x4
16x4x4
32x4x4
64x4x4
128x4x4

Time in seconds
Problems
Preprocessing
Inprocessing
Carsten Sinz, Tomáš Balyo – SAT Solving
July 10, 2019
Experiments – Speedups

Big Instance = solved after $10 \cdot (\#\text{threads})$ seconds by Lingeling

<table>
<thead>
<tr>
<th>Core Solvers</th>
<th>Parallel Solved</th>
<th>Both Solved</th>
<th>Speedup All</th>
<th>Speedup Big</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x4x4</td>
<td>385</td>
<td>363</td>
<td>303</td>
<td>25.01</td>
</tr>
<tr>
<td>2x4x4</td>
<td>421</td>
<td>392</td>
<td>310</td>
<td>30.38</td>
</tr>
<tr>
<td>4x4x4</td>
<td>447</td>
<td>405</td>
<td>323</td>
<td>41.30</td>
</tr>
<tr>
<td>8x4x4</td>
<td>466</td>
<td>420</td>
<td>317</td>
<td>50.48</td>
</tr>
<tr>
<td>16x4x4</td>
<td>480</td>
<td>425</td>
<td>330</td>
<td>65.27</td>
</tr>
<tr>
<td>32x4x4</td>
<td>481</td>
<td>427</td>
<td>399</td>
<td>83.68</td>
</tr>
<tr>
<td>64x4x4</td>
<td>476</td>
<td>421</td>
<td>377</td>
<td>104.01</td>
</tr>
<tr>
<td>128x4x4</td>
<td>476</td>
<td>421</td>
<td>407</td>
<td>109.34</td>
</tr>
</tbody>
</table>
Big Instance = solved after $10 \cdot (\# \text{threads})$ seconds by Lingeling.