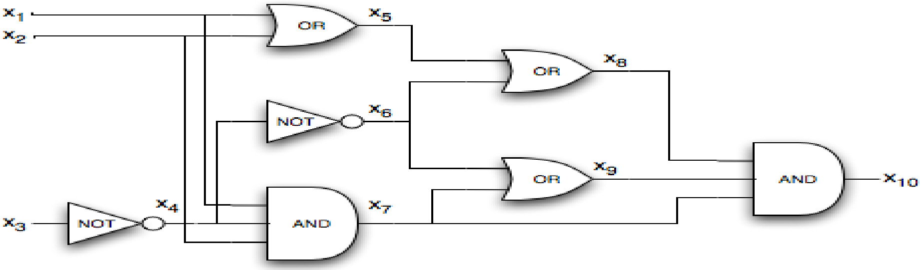
Karlsruhe Institute of Technology

Practical SAT Solving

Lecture 8

Carsten Sinz, Tomáš Balyo | July 10, 2019

INSTITUTE FOR THEORETICAL COMPUTER SCIENCE



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

Karlsruhe Institute of Technology

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

Karlsruhe Institute of Technology

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 $x = AND(y, z)$ 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: $S = (G_x \cup R_x) \cup (G_{\overline{x}} \cup R_{\overline{x}})$.
- Clause distribution results in $S' = (G_x \otimes R_{\overline{x}}) \cup (R_x \otimes G_{\overline{x}}) \cup (R_x \otimes R_{\overline{x}})$
- Moreover, $(G_x \otimes R_{\overline{x}}) \cup (R_x \otimes G_{\overline{x}}) \models (R_x \otimes R_{\overline{x}})$. (Why?)
- Thus, we can replace S by the satisfiability-equivalent $(R_x \otimes G_{\overline{x}}) \cup (R_{\overline{x}} \otimes G_x)$.

Failed Literal Elimination

- If applying unit propagation on $F \wedge \{l\}$ derives UNSAT ($F \wedge \{l\} \vdash_{UP} \bot$), replace F by $F \wedge \{\neg l\}$.
- Generalization: If $(F \setminus \{C\}) \wedge \neg C \vdash_{UP} \bot$, remove C from F.

Blocked Clauses

Definition

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

Example: $F = (a \vee b) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee c)$.
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 σ and a formula F, a clause (of F) is *touched by* σ if it contains the negation of a literal assigned in σ. A clause is *satisfied by* σ if it contains a literal assigned to *True* by σ. If all touched clauses are satisfied then σ is an *autarky*.

- All clauses touched by an autarky can be removed
- Example: $\{\neg a, b\}, \{\neg a, c\}, \{a, \neg b, \neg c\}, \{b, d\}, \dots$ (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, $x_{\mathrm{new}} \leftrightarrow l_1 \wedge l_2$. As a rule for formulas in CNF:

$$\frac{}{(\neg x_{\mathrm{new}} \vee l_1) \wedge (\neg x_{\mathrm{new}} \vee l_2) \wedge (x_{\mathrm{new}} \vee \neg l_1 \vee \neg l_2)}$$

- 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 reasoing
  - 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

Parallel SAT Solving

Three kinds of approaches

- Divide and Conquer – explicit search space paritioning
- Cube and Conquer – implicit load balancing
- Diversify and Conquer – portfolio search

Partitioning – Böhm and Speckenmeyer

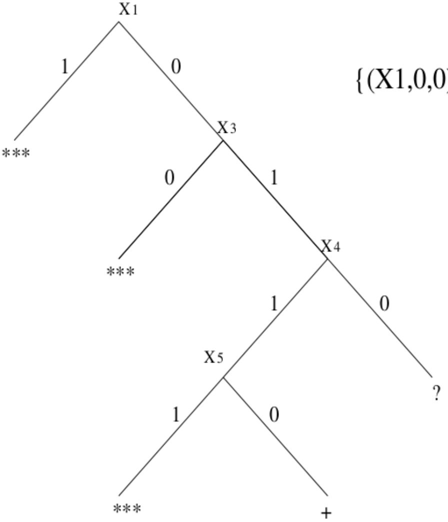
- 1994 First parallel implementation of DPLL
- completely distributed (no master and slave roles)
- A list of partial assignment is generated
- Each processors receives the entire formula and a few partial assignments
- Each Processors consists of
  - Worker (solve or split the formula, use the partial 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.

PSATO – Zhang et al. 1994

Karlsruhe Institute of Technology

- 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



guiding path

$\{(X1,0,0),(X3,1,0),(X4,1,1),(X5,0,0)\}$

*** : explored branch

+ : current node

? : remaining subtree

Satz – Jurkowiak et al. 2001

- The solver *Satz* improves PSATO the by adding *work stealing* for workload balancing
  - An idle slave request 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

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 processors 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.[1]: ManySAT – the first solver using this idea (based on MiniSat)

[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® Research

What Makes a Good Portfolio Solver

Two Pillars of Portfolios:

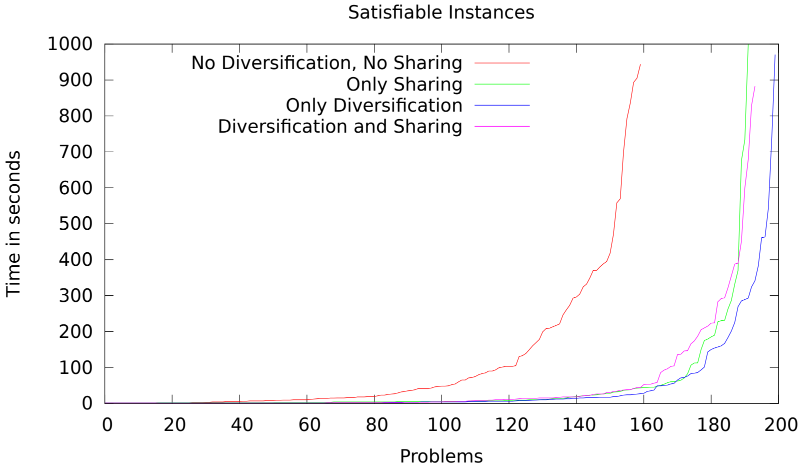
Diversification

- 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

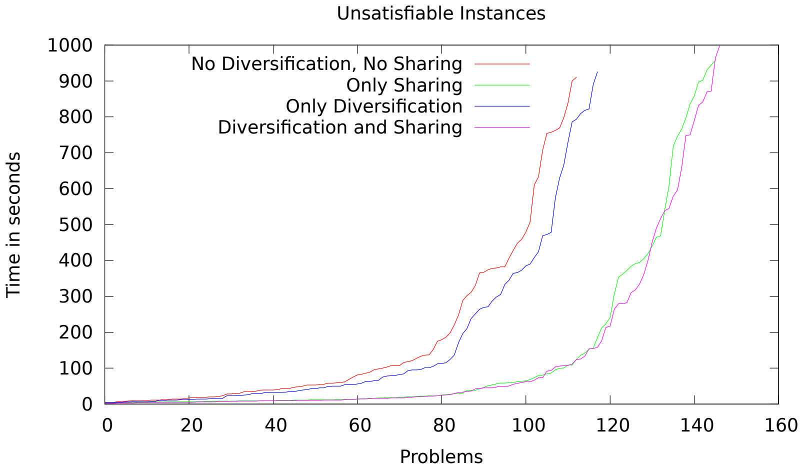
- Which clauses to share?
- How many?
- How often?
- How to implement efficiently?

Experiments – Random Satisf. 3-SAT

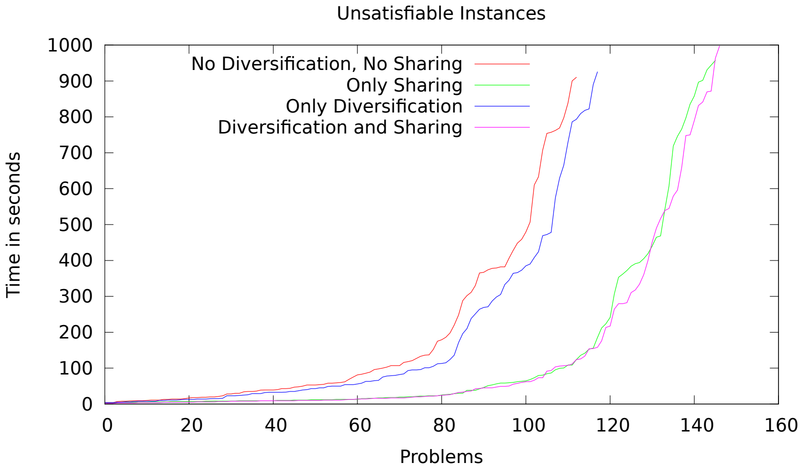


Advice for Satisfiable problems

Experiments – Random Unsat. 3-SAT



Experiments – Random Unsat. 3-SAT



- 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

```
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

Karlsruhe Institute of Technology

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. $\frac{1}{\#solvers}$

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

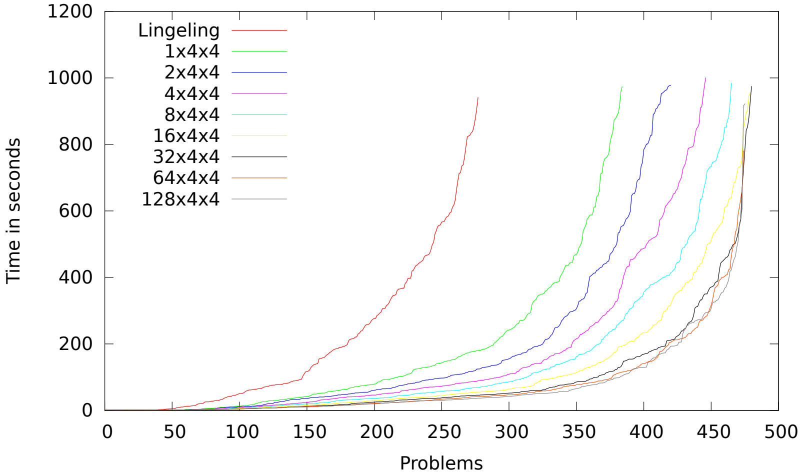
Overall Algorithm

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



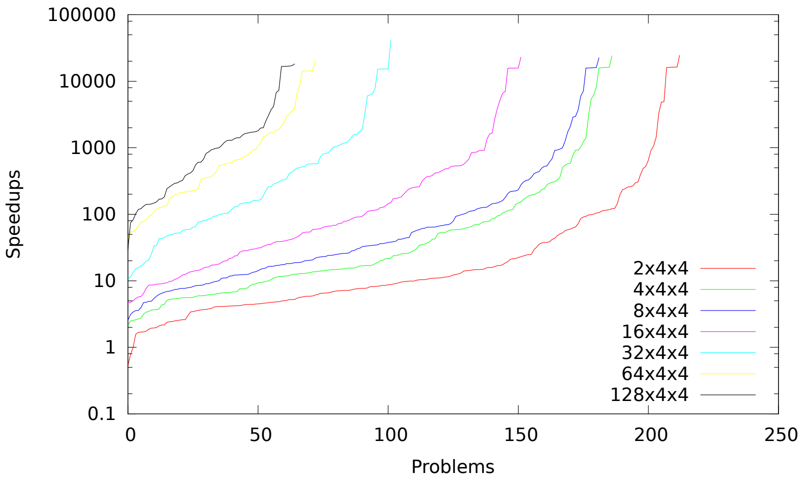
Experiments – Speedups

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

Core Solvers	Parallel Solved	Both Solved	Speedup All Avg.	Speedup All Tot.	Speedup All Med.	Speedup Big Avg.	Speedup Big Tot.	Speedup Big Med.
1x4x4	385	363	303	25.01	3.08	524	26.83	4.92
2x4x4	421	392	310	30.38	4.35	609	33.71	9.55
4x4x4	447	405	323	41.30	5.78	766	49.68	16.92
8x4x4	466	420	317	50.48	7.81	801	60.38	32.55
16x4x4	480	425	330	65.27	9.42	1006	85.23	63.75
32x4x4	481	427	399	83.68	11.45	1763	167.13	162.22
64x4x4	476	421	377	104.01	13.78	2138	295.76	540.89
128x4x4	476	421	407	109.34	13.05	2607	352.16	867.00

Experiments – Speedups on Big Inst.

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



References I