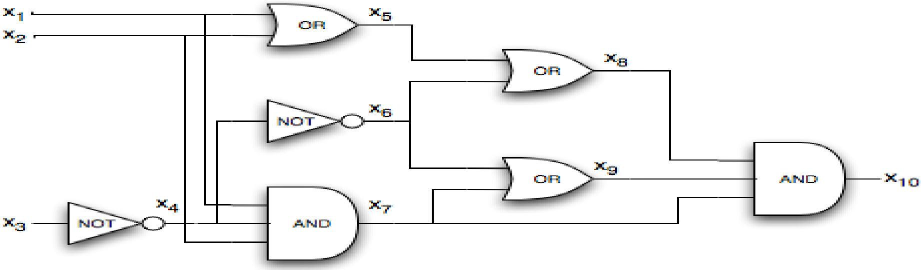
Karlsruhe Institute of Technology

Practical SAT Solving

Lecture 9

Carsten Sinz, Tomáš Balyo | May 23, 2016

INSTITUTE FOR THEORETICAL COMPUTER SCIENCE



Lecture Outline

- Parallel SAT Solving
- Incremental SAT Solving

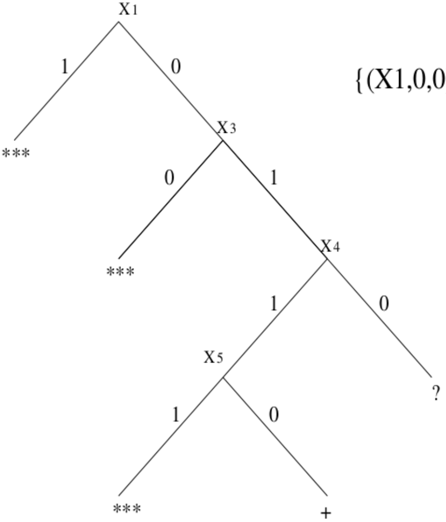
Partitioning – Böhm and Speckenmeyer

- 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

- 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

2001 – 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:

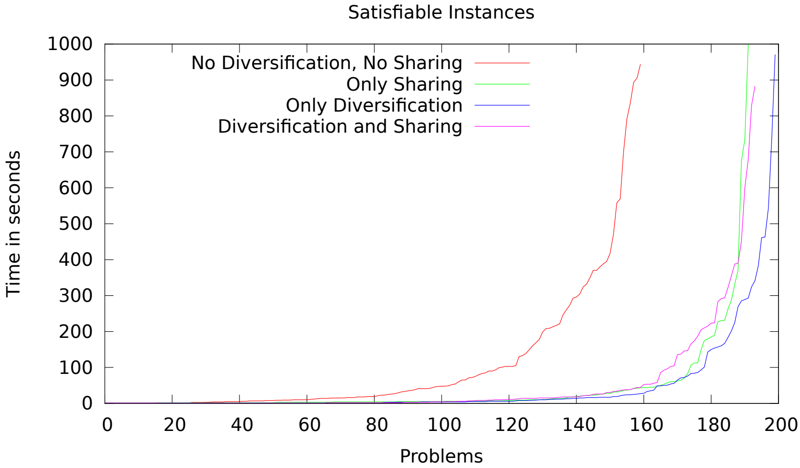
Diversification

- 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

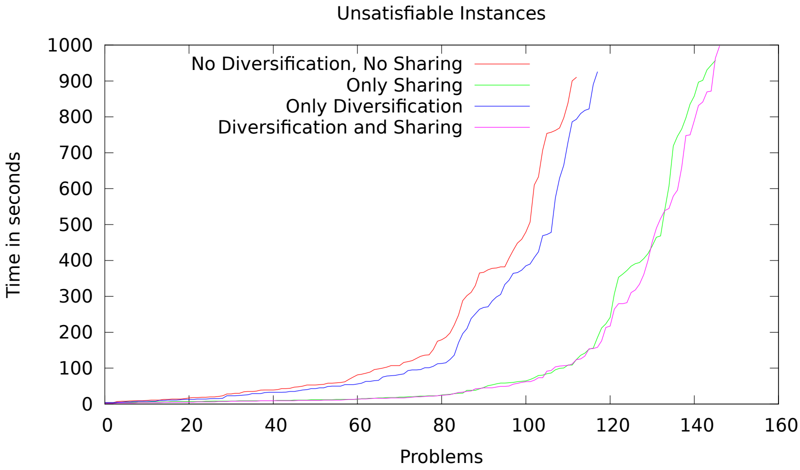
- Which clauses to share?
- How many?
- How often?
- How to implement efficiently?

Experiments – Random Satisf. 3-SAT

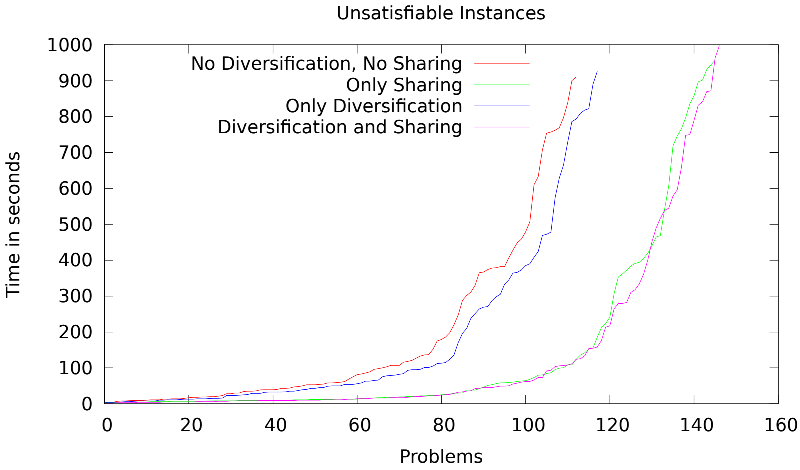


Advice for Satisfiable problems

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

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

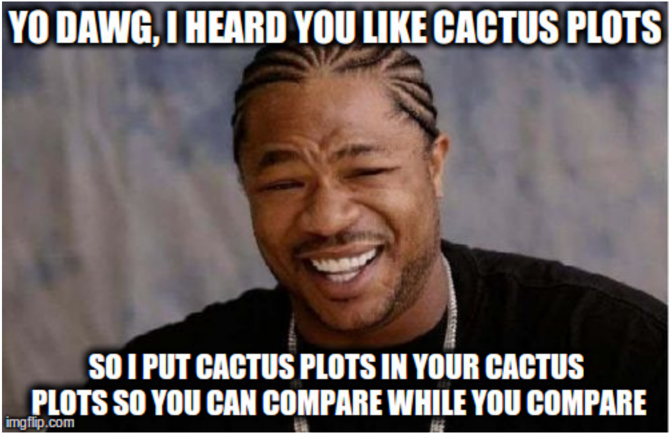
Overall Algorithm

The Same Code for Each Process

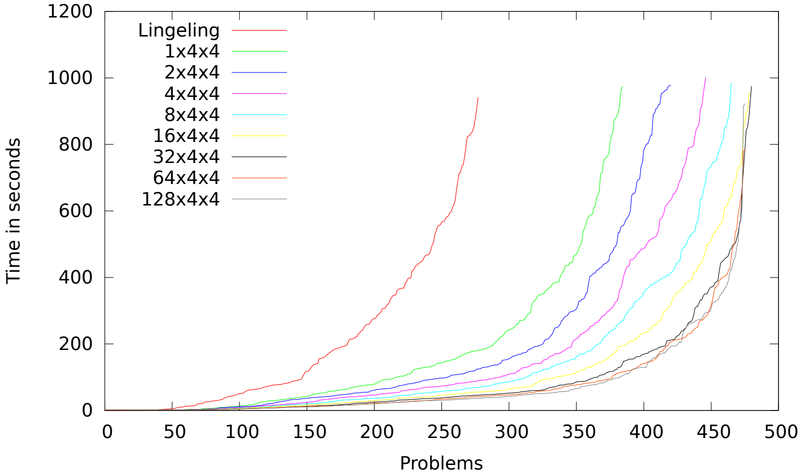
```
SolveFormula(F, rank, size)
    for i = 1 to numThreads do
        s[i] = new PortfolioSolver(Lingeling);
        s[i].addClauses(F);
        diversify(s[i], rank, size);
        new Thread(s[i].solve());

    forever do
        sleep(1) // 1 second
        if (anySolverFinished) break;
        exchangeLearnedClauses(s, rank, size);
```

HordeSAT Experimental Results



Experiments – SAT 2011+2014



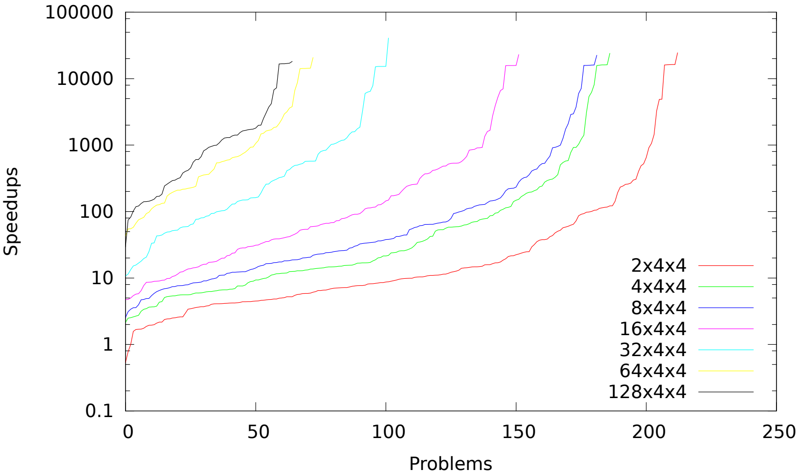
Experiments – Speedups

Big Instance = solved after 10 · (#*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 · (#*threads*) seconds by Lingeling



Incremental SAT Solving

- We often need to solve a sequence of similar SAT instances
  - for example planning as sat, sokoban, bounded model checking
  - the instances share most of the clauses with their neighbors
- Can we solve these sequences of instances more efficiently?

Incremental SAT Solving

- We often need to solve a sequence of similar SAT instances
  - for example planning as sat, sokoban, bounded model checking
  - the instances share most of the clauses with their neighbors
- Can we solve these sequences of instances more efficiently?
- What is incremental SAT solving?
  - Clauses can be added to and removed from the SAT solver
- Why not call the solver with the new formula every time?

Incremental SAT Solving

- We often need to solve a sequence of similar SAT instances
  - for example planning as sat, sokoban, bounded model checking
  - the instances share most of the clauses with their neighbors
- Can we solve these sequences of instances more efficiently?
- What is incremental SAT solving?
  - Clauses can be added to and removed from the SAT solver
- Why not call the solver with the new formula every time?
  - The solver can remember learned clauses and other stuff (variable scores required for heuristics)
  - (de)initialization overheads removed

Incremental Library Track – IPASIR

- Previously each SAT solver had a different incremental interface
- For the 2015 SAT Race a unified interface was defined – IPASIR
- IPASIR = Re-entrant Incremental Satisfiability Application Program Interface (acronym reversed)
- Currently around 10 SAT solvers are IPASIR compatible

IPASIR Overview

- Based on Lingeling incremental interface
- Clauses are added one literal at a time
  - To add $(x_1 \vee \overline{x_4})$ call `add(1); add(-4); add(0);`
- You can call a SAT solver with a set of assumptions
  - Assumptions are basically temporary unit clauses
  - Assumptions are cleared after each "solve" call
- Clause removal is done via activation literals and assumptions
  - You must know ahead which clauses you will maybe want to remove
  - Add the clause with an additional fresh variable (activation literal)
  - example: instead of $(x_1 \vee x_2)$ add $(x_1 \vee x_2 \vee a_1)$
  - solve with with assumption $\overline{a_1}$ to enforce $(x_1 \vee x_2)$
  - drop the assumption $\overline{a_1}$ to drop $(x_1 \vee x_2)$

Ipasir Interface

ipasir.h

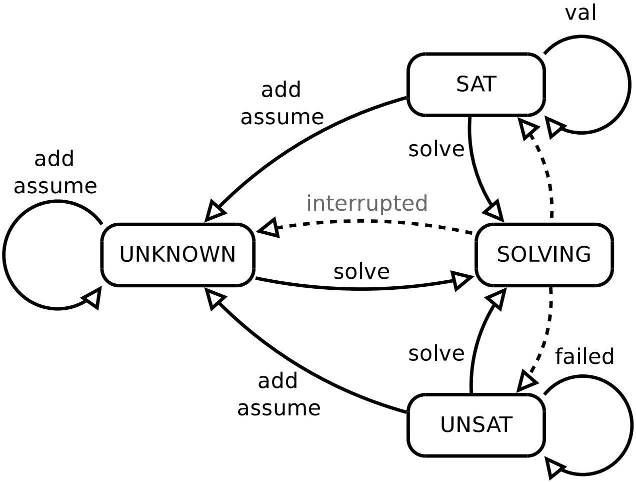
```
const char* ipasir_signature();
void* ipasir_init();
void ipasir_release(void* solver);
void ipasir_set_terminate(void* solver, void* state,
                          int (*terminate)(void* state));
void ipasir_add(void* solver, int lit_or_zero);
void ipasir_assume(void* solver, int lit);
int ipasir_solve(void* solver);
int ipasir_val(void* solver, int lit);
int ipasir_failed(void* solver, int lit);
```

For more details and examples of usage see `http://baldur.iti.kit.edu/sat-competition-2016/downloads/ipasir.zip`

IPASIR Functions

- signature – return the name and version of the solver
- init – initialize the solver, the pointer it returns is used for the rest of the functions
- add – add clauses one literal at a time
- assume – add an assumption, the assumptions are cleared after a "solve" call
- solve – solve the formula, return SAT, UNSAT or INTERRUPTED
- val – return the truth value of a variable (if solve returned SAT)
- failed – returns true if the given assumption was required for the unsatisfiability of the formula (if solver returned UNSAT)

IPASIR Solver States



Example – Essential Variables

- For a satisfiable formula F a variable x is essential if and only x is has to be assigned (True or False) in each satisfying assignment of F.
- Task: find all the essential variables of a given formula
- How to do it:
  - use Dual Rail Encoding – for each variable x add two new variables x_P and x_N, replace each positive (negative) occurrence of x with x_P (x_N), add a clause $(\overline{x_P} \vee \overline{x_N})$ (meaning x cannot be both true and false).
  - for each variable x solve the formula with the assumptions $\overline{x_P}$ and $\overline{x_N}$. If the formula is UNSAT then x is essential.

Example – Essential Variables – code

```
int pdr(int var) { return 2*var; }
int ndr(int var) { return 2*var - 1; }
int  dr(int lit) { return lit > 0 ? pdr(lit) : ndr(-lit); }

void Essentials(Formula f) {
  void* s = ipasir_init();
  for (int c = 0; c < f.clauses; c++) {
    for (int k = 0; k < f.clause[c].size; k++) {
      ipasir_add(s, dr(f.clause[c].lit[k]));
    }
    ipasir_add(s, 0);
  }
  for (int v = 1; v <= f.variables; v++) {
    ipasir_add(s, -pdr(v));
    ipasir_add(s, -ndr(v));
    ipasir_add(s, 0);
  }
  for (int v = 1; v <= f.variables; v++) {
    ipasir_assume(s, -pdr(v));
    ipasir_assume(s, -ndr(v));
    if (ipasir_solve(s) == 20) {
      printf("%d␣is␣Essential\n", v);
    } else {
      printf("%d␣is␣not␣Essential\n", v);
    }
  }
  ipasir_release(s);
}
```