Lecture Outline

- Parallel SAT Solving
- Incremental SAT Solving
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

guiding path

\{(X1,0,0),(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.
2001 – 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
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.\(^1\): 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:

<table>
<thead>
<tr>
<th>Diversification</th>
</tr>
</thead>
<tbody>
<tr>
<td>The search space of the solvers should not overlap too much</td>
</tr>
<tr>
<td>Use different configuration values of heuristic parameters</td>
</tr>
<tr>
<td>Partial assignment recommendations (no restrictions!)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clause Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which clauses to share?</td>
</tr>
<tr>
<td>How many?</td>
</tr>
<tr>
<td>How often?</td>
</tr>
<tr>
<td>How to implement efficiently?</td>
</tr>
</tbody>
</table>
Experiments – Random Satisf. 3-SAT

Satisfiable Instances

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

Time in seconds vs. Problems

Problems

Time in seconds
Advice for Satisfiable problems

DIVERSIFY
YOUR PORTFOLIO
Experiments – Random Unsat. 3-SAT

Unsatisfiable Instances

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

Time in seconds vs Problems

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
### Portfolio Solver Interface

```c
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
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

YO DAWG, I HEARD YOU LIKE CACTUS PLOTS

SO I PUT CACTUS PLOTS IN YOUR CACTUS PLOTS SO YOU CAN COMPARE WHILE YOU COMPARE

imgflip.com
Experiments – SAT 2011+2014

![Graph showing the time in seconds for different problem sizes solved by Lingeling. The x-axis represents problems, and the y-axis represents time in seconds. The graph compares the performance of Lingeling on different problem sizes: 1x4x4, 2x4x4, 4x4x4, 8x4x4, 16x4x4, 32x4x4, 64x4x4, and 128x4x4. Each line represents a different problem size, with the time increasing as the problem size increases.](image-url)
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>303</td>
<td>25.01</td>
<td>3.08</td>
<td>524</td>
</tr>
<tr>
<td>2x4x4</td>
<td>310</td>
<td>30.38</td>
<td>4.35</td>
<td>609</td>
</tr>
<tr>
<td>4x4x4</td>
<td>323</td>
<td>41.30</td>
<td>5.78</td>
<td>766</td>
</tr>
<tr>
<td>8x4x4</td>
<td>317</td>
<td>50.48</td>
<td>7.81</td>
<td>801</td>
</tr>
<tr>
<td>16x4x4</td>
<td>330</td>
<td>65.27</td>
<td>9.42</td>
<td>1006</td>
</tr>
<tr>
<td>32x4x4</td>
<td>399</td>
<td>83.68</td>
<td>11.45</td>
<td>1763</td>
</tr>
<tr>
<td>64x4x4</td>
<td>377</td>
<td>104.01</td>
<td>13.78</td>
<td>2138</td>
</tr>
<tr>
<td>128x4x4</td>
<td>407</td>
<td>109.34</td>
<td>13.05</td>
<td>2607</td>
</tr>
</tbody>
</table>
Experiments – Speedups on Big Inst.

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

![Graph showing speedups on Big Inst.](image-url)
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
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 \((x_1 \lor \overline{x_4})\) 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 \((x_1 \lor x_2)\) add \((x_1 \lor x_2 \lor a_1)\)
  - solve with with assumption \(\overline{a_1}\) to enforce \((x_1 \lor x_2)\)
  - drop the assumption \(\overline{a_1}\) to drop \((x_1 \lor x_2)\)
Ipasir Interface

**ipasir.h**

```c
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](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

- **UNKNOWN**
  - add assume

- **SOLVING**
  - solve

- **UNSAT**
  - add assume

- **SAT**
  - val
  - solve
  - interrupted

- **UNSAT**
  - failed
  - solve

- **UNKNOWN**
  - add assume
  - interrupted

Diagram represents the states and transitions in the IPASIR solver.
Example – Essential Variables

- For a satisfiable formula $F$ a variable $x$ is essential if and only if $x$ 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 $(\overline{x_P} \lor \overline{x_N})$ (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.
```c
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);
}
```