

From SAT to Parallel SAT Solving

Youssef Hamadi

Microsoft Research

Constraint Reasoning Group

Cambridge

Outline of this tutorial

1. The Propositional Satisfiability Problem (SAT)

- Theoretical importance
- Practical significance
 - Software verification

2. The state-of-the-art sequential algorithm

- Efficient BUP
- Conflict analysis
- Activity-based heuristics
- Restarts

3. Parallel SAT Solving

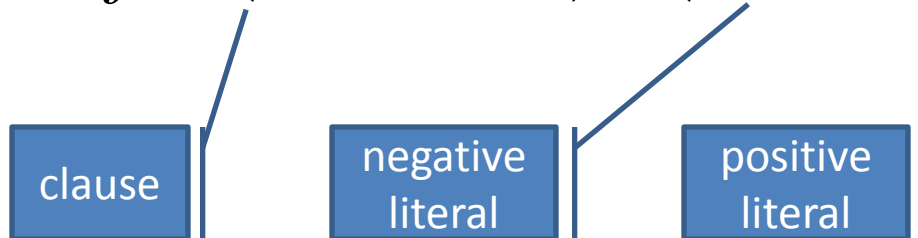
- Motivation
- Parallel Relaxation
- Parallel Search
 - Divide-and-conquer
 - Clause-sharing
 - Impact
 - Integration
 - Portfolio approach
 - Problems of static size clause sharing
 - Dynamic clause sharing

DEFINITION

The Propositional Satisfiability Problem (SAT)

- For a given Boolean formula f
 - Find an assignment of the variables which evaluates to true.
 - Prove that such assignment does not exist.
- Conjunctive Normal Form (CNF)
 - Example

$$f = (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_5)$$



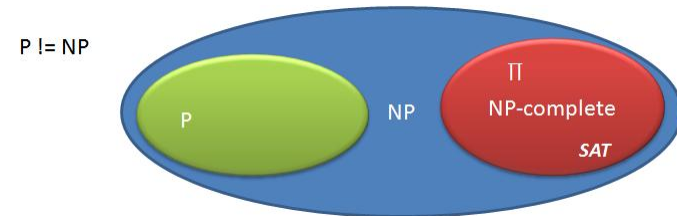
The Propositional Satisfiability Problem (SAT)

- A clause forbids a partial assignment: *nogood*
- Example:
 - Given a Boolean formula, $f(x_1, \dots, x_n)$
 - Clause $c: (\neg x_1 \vee \neg x_2 \vee \dots \vee \neg x_k) \setminus \text{in } f$
 - $c \approx \neg(x_1=\text{true} \wedge x_2=\text{true} \wedge \dots \wedge x_k=\text{true})$
 - c removes $2^{(n-k)}$ tuples

MOTIVATION

Theoretical Importance

- Established as the first NP-complete problem [Cook 1971].
- SAT used to prove NP-completeness of many important problems Π .
 1. Π in NP (Polynomial certificate)
 2. SAT 'efficiently-transformed-into' Π
 3. Π NP-complete, i.e., difficult ☹️ (assuming $P \neq NP$)
 4. Consider approximate solutions
e.g., greedy algorithms, dynamic-programming, etc.



Practical Importance

- Tremendous performance gains of sequential SAT-solvers in the last 10-15 years:
 - 3 orders of magnitude gains
 - 10 of thousands variables
 - Millions of clauses
- Black-box solving: no need to add domain-knowledge to be efficient.
- Π NP-complete \Rightarrow Π 'efficiently-transformed-into' SAT.
- New workflow:
 1. Π 'efficiently-transformed-into' SAT
 2. Run an out-of-the-box SAT solver on your encoded instance 😊

Example: k-colorability

- Graph k-colorability is NP-complete [Karp 72].
- Efficient SAT encoding of a particular instance
 - ‘Efficient’ polynomial in the size of the input graph
 - Example, 3-colorability:

Vertex A:

3 Boolean vars: A_1, A_2, A_3

“each vertex has one color”:

$$(A_1 \vee A_2 \vee A_3)$$

“one color at the time”:

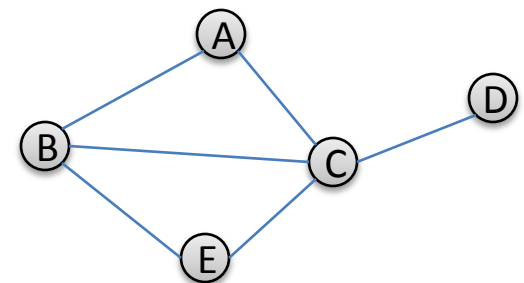
$$(\neg A_1 \vee \neg A_2) \wedge (\neg A_2 \vee \neg A_3) \wedge (\neg A_3 \vee \neg A_1)$$

“adjacent nodes have different colors”:

(A,B):

$$(\neg A_1 \vee \neg B_1) \wedge (\neg A_2 \vee \neg B_2) \wedge (\neg A_3 \vee \neg B_3)$$

Etc.



Practical Importance: Software Verification

- Fixing a bug costs:
 - \$1 on a programmer's desktop
 - \$100 once in a complete program
 - +\$1000 once released
- US National Institute of Standards and Technology (NIST), 2002 report:
 - 80% of software development costs spent on fixing defects (debugging).
 - Software errors cost the U.S. economy \$59.5 billion/year, i.e., 0.6% of GDP.

Software verification: example

- Microsoft's Spec# programming language:
 - Extension of C# (... similar to Java).
 - Design-by-contract methodology
 - Every public method has a precondition and a postcondition
 - Pre and postconditions are checked at compile time
 - Preconditions and postconditions are side effect free boolean-valued expressions - i.e. they evaluate to true/false
 - Static program verifier
 - Generates logical verification conditions from a Spec# program.
 - Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove the correctness of the program or find errors in it.

Spec# language: example

Source: [Rustan et al. 2008]

```
static void Swap(int[] a, int i, int j)
```

```
requires 0 <= i && i < a.Length;
```

```
requires 0 <= j && j < a.Length;
```

```
modifies a[i], a[j];
```

```
ensures a[i] == old(a[j]);
```

```
ensures a[j] == old(a[i]);
```

```
{
```

```
    int temp;
```

```
    temp = a[i];
```

```
    a[i] = a[j];
```

```
    a[j] = temp;
```

```
}
```

preconditions

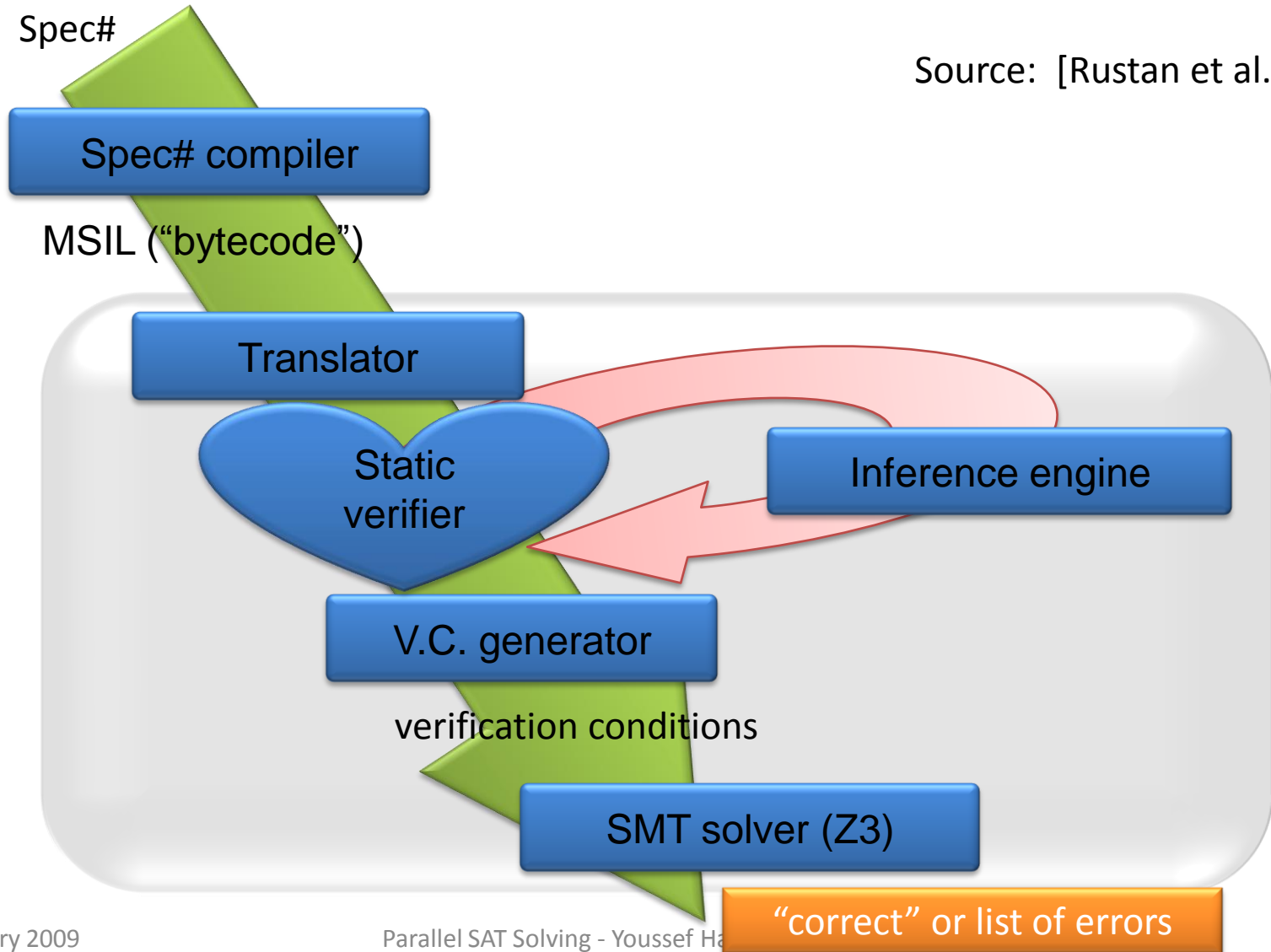
declares what a
method is allowed
to modify

old, value at the
entry of the
method

Spec# verifier: architecture

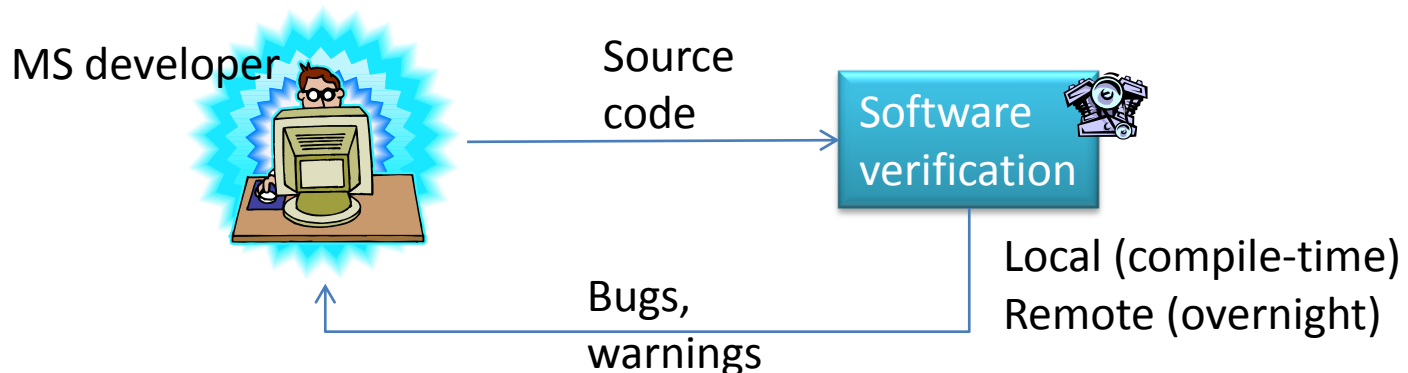
Spec#

Source: [Rustan et al. 2008]



Software verification in Microsoft

- Ratio (developer/tester) close to 1.
- Several teams of researchers / engineers, developing fundamental theory / practical tools for better software verification.
- **MS tools:** SAGE (CP-based, code coverage), Pex (static verification), SDV/SLAM (driver verification), Spec#, etc.



Practical Importance

- Demanding Application Domains:
 - Software verification
 - Electronic Design Automation
 - General theorem proving
 - Artificial Intelligence
 - Computational biology
 - Etc.
- Application pattern:

SAT as a basic building block of a more complex reasoning engine.

Outline of this tutorial

1. The Propositional Satisfiability Problem (SAT)

- Theoretical importance
- Practical significance
 - Software verification

2. The state-of-the-art sequential algorithm

- Efficient BUP
- Conflict analysis
- Activity-based heuristics
- Restarts

3. Parallel SAT Solving

- Motivation
- Parallel Relaxation
- Parallel Search
 - Divide-and-conquer
 - Clause-sharing
 - Impact
 - Integration
 - Portfolio approach
 - Problems of static size clause sharing
 - Dynamic clause sharing

THE STATE-OF-THE-ART SEQUENTIAL ALGORITHM

Basic mechanisms

- Decision
 - Assign a variable to some truth value.
- Implication
 - Decisions force the truth value of a variable.
 - *Unit-clause rule*: an unsatisfied clause is unit if it has exactly one unassigned literal.
 - Example: $f = (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_5)$

$$x_1 = \text{true} \Rightarrow x_2 = \text{false}$$

$$x_3 = \text{false} \Rightarrow x_4 = \text{true}$$

Basic mechanisms

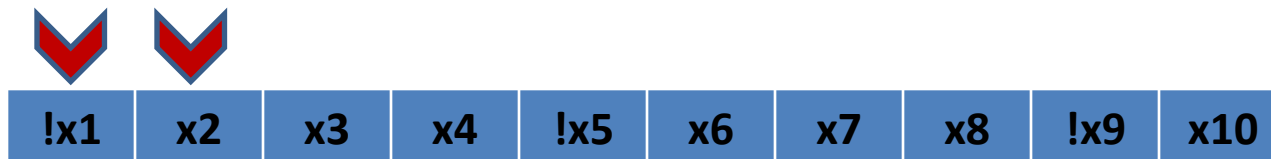
1. Boolean Unit Propagation (BUP)
 - Application of the unit-clause rule until
 - Fix-point.
 - **Conflict**: all literals in a clause are unsatisfied (wrong decisions).
2. Conflict analysis
 - *Learn* a conflict-clause
 - Avoid future occurrences of conflict
 - Correct the set of decisions
3. Activity-based heuristics
 - Order the decisions
 - Prioritize conflict-clauses
4. Restart policy
 - Give-up and restart (a more informed) search

Boolean Unit Propagation

- The problem: Determine as fast as possible the clauses that become unit,
 - i.e., all but one literals are false
 - >we must impose the remaining one

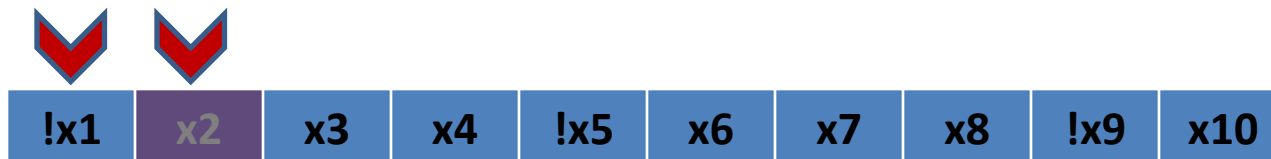
Boolean Unit Propagation

2-watch approach: zChaff [Moskewicz et al. 2001]



Boolean Unit Propagation

2-watch approach: zChaff [Moskewicz et al. 2001]



When a watched literal gets violated: we search for a free literal at a position different from the other pointer

Boolean Unit Propagation

2-watch approach: zChaff [Moskewicz et al. 2001]



Boolean Unit Propagation

2-watch approach: zChaff [Moskewicz et al. 2001]



When a watched literal gets violated: we search for a free literal at a position different from the other pointer

Boolean Unit Propagation

2-watch approach: zChaff [Moskewicz et al. 2001]



When the only free position is the other Pointer, the corresponding literal is unit (when all positions incorrect, fail)

Boolean Unit Propagation

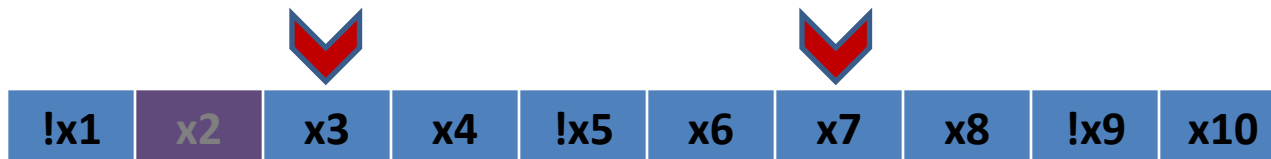
2-watch approach: zChaff [Moskewicz et al. 2001]



Remark: the pointers don't even need to be *backtrackable*.

Boolean Unit Propagation

2-watch approach: zChaff [Moskewicz et al. 2001]



Remark: the pointers don't even need to be *backtrackable*.

Conflict analysis

- GRASP [Marques-Silva et al. 1996]
- After a decision,
 - Boolean Unit Propagation until fix-point.
 - If conflict, **analysis** for subsequent pruning and current backjumping.

Conflict analysis

c1: $(\neg x_2 \vee \neg x_3 \vee x_4)$

c4: $(\neg x_5 \vee x_7)$

c2: $(\neg x_1 \vee \neg x_4 \vee x_6)$

c5: $(\neg x_6 \vee \neg x_7)$

c3: $(\neg x_4 \vee x_5)$

conflict

Conflict analysis

c1: $(\neg x_2 \vee \neg x_3 \vee x_4)$

c4: $(\neg x_5 \vee x_7)$

c2: $(\neg x_1 \vee \neg x_4 \vee x_6)$

c5: $(\neg x_6 \vee \neg x_7)$

c3: $(\neg x_4 \vee x_5)$

$x_1=1$ (1)

Decisions, BUP:

1. $x_1=1, \{\}$

conflict

Conflict analysis

$c1: (\neg x2 \vee \neg x3 \vee x4)$

$c4: (\neg x5 \vee x7)$

$c2: (\neg x1 \vee \neg x4 \vee x6)$

$c5: (\neg x6 \vee \neg x7)$

$c3: (\neg x4 \vee x5)$

$x1=1$ (1)

$x2=1$ (2)

Decisions, BUP:

1. $x1=1, \{\}$

2. $x2=1, \{\}$

conflict

Conflict analysis

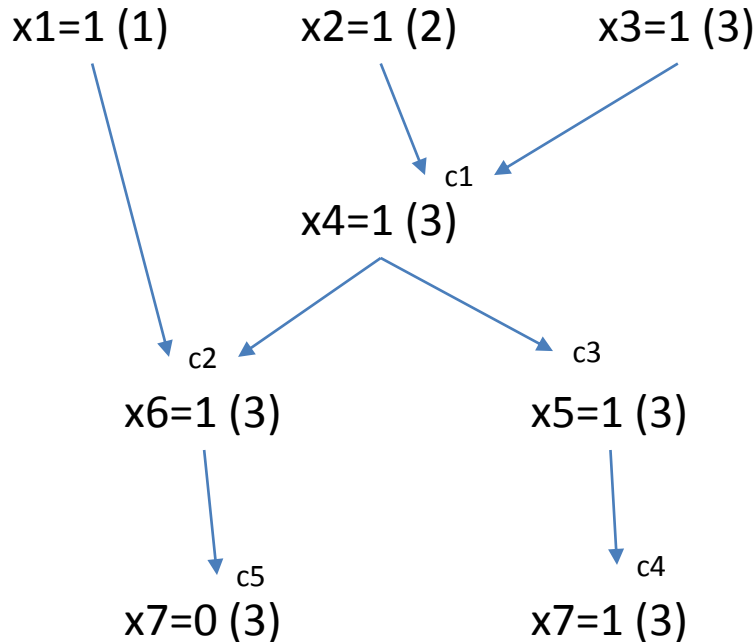
$$c1: (\neg x2 \vee \neg x3 \vee x4)$$

$$c2: (\neg x1 \vee \neg x4 \vee x6)$$

$$c3: (\neg x4 \vee x5)$$

$$c4: (\neg x5 \vee x7)$$

$$c5: (\neg x6 \vee \neg x7)$$



conflict

Decisions, BUP:

1. $x1=1, \{\}$
2. $x2=1, \{\}$
3. $x3=1, \{x4, x5, x6\}$

Conflicting-variable: $x7$

Conflicting-clause: $c5$

Conflict analysis

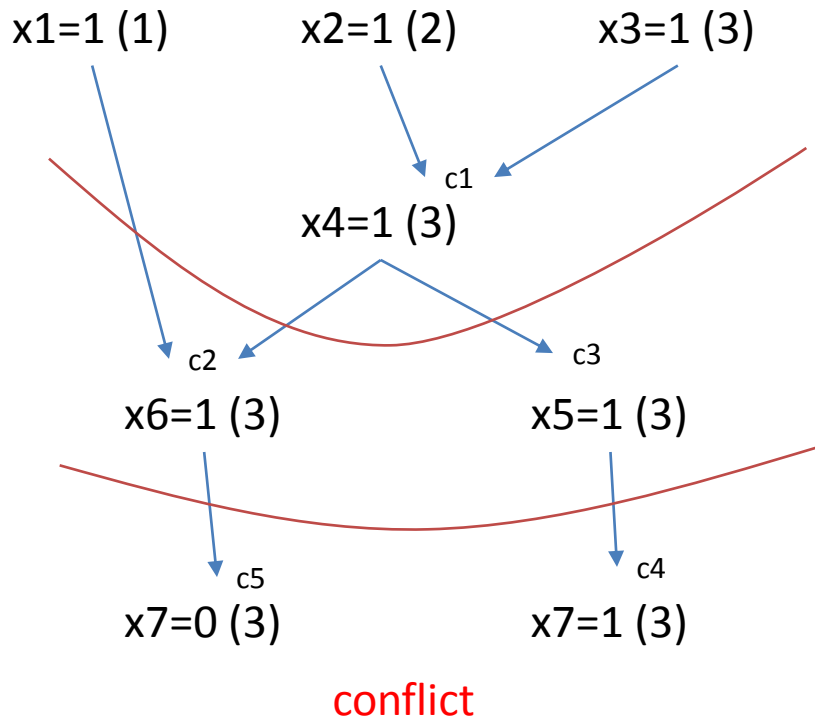
$$c1: (\neg x2 \vee \neg x3 \vee x4)$$

$$c2: (\neg x1 \vee \neg x4 \vee x6)$$

$$c3: (\neg x4 \vee x5)$$

$$c4: (\neg x5 \vee x7)$$

$$c5: (\neg x6 \vee \neg x7)$$



Decisions, BUP:

1. $x1=1, \{\}$
2. $x2=1, \{\}$
3. $x3=1, \{x4, x5, x6\}$

Conflicting-variable: $x7$

Conflicting-clause: $c5$

Candidate learnt clauses:

- any cut between the decisions and the conflict
- $(\neg x6 \vee \neg x5), (\neg x1 \vee \neg x4)$, etc.

Conflict analysis

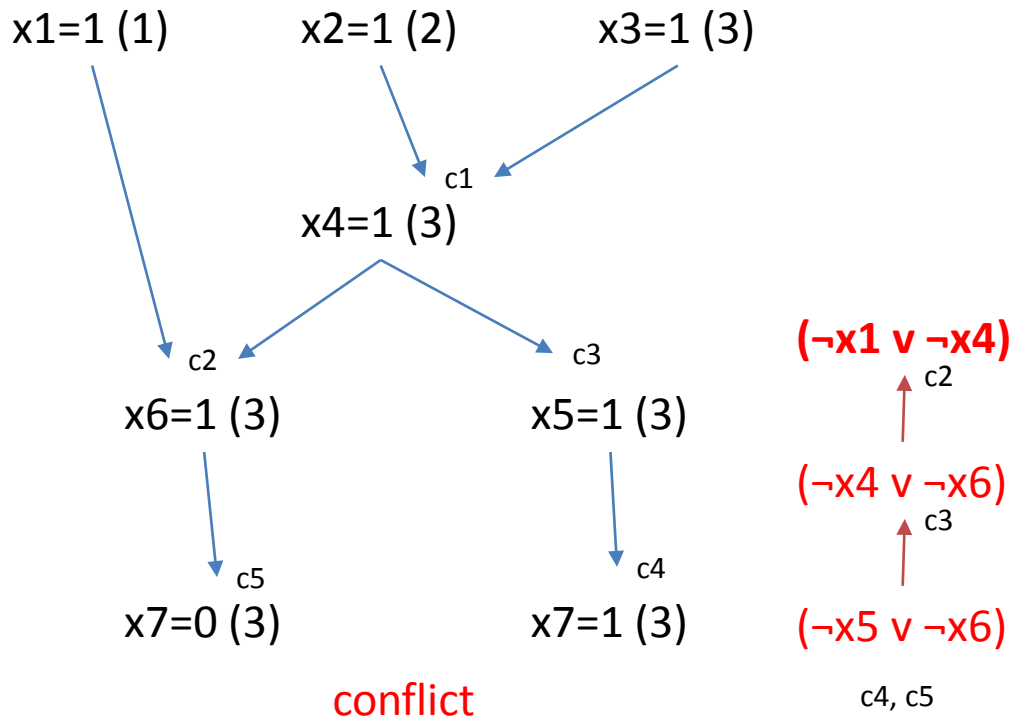
$$c1: (\neg x2 \vee \neg x3 \vee x4)$$

$$c4: (\neg x5 \vee x7)$$

$$c2: (\neg x1 \vee \neg x4 \vee x6)$$

$$c5: (\neg x6 \vee \neg x7)$$

$$c3: (\neg x4 \vee x5)$$



Decisions, BUP:

1. $x1=1, \{\}$
2. $x2=1, \{\}$
3. $x3=1, \{x4, x5, x6\}$

Conflicting-variable: $x7$

Conflicting-clause: $c5$

First-Unique-Implication point:

asserting clause: becomes unit-clause after backjumping. Only one literal of current decision level.

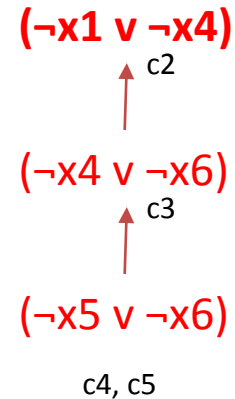
Backjumping: level 1.

1. $x1=1, \{\neg x4\}$

Activity-based heuristics

- [Moskewicz et al. 2001]
- **Variable selection**, *Variable State Independent Decaying Sum (VSIDS) decision heuristic*:
 - Each variable maintains a counter that stores the number of times it was part of a clause used during a backward resolution process.
 - Updated during conflict analysis.
 - Values of all counters are periodically divided by a small constant $c > 1$: preference given to recently deduced conflicts.
 - Select free variable x with maximum counter.

$c1: (\neg x2 \vee \neg x3 \vee x4)$, $c2: (\neg x1 \vee \neg x4 \vee x6)$
 $c3: (\neg x4 \vee x5)$, $c4: (\neg x5 \vee x7)$, $c5: (\neg x6 \vee \neg x7)$



x1	x2	x3	x4	x5	x6	x7
1	0	0	1	1	1	0

Activity-based heuristics

- [Moskewicz et al. 2001]
- **Clause deletion:**
 - Updated during conflict analysis.
 - Each clause maintains a counter that stores the number of times it was used during a backward resolution process.
 - Values of all counters are periodically divided by a small constant $c > 1$: preference given to recently deduced clauses.
 - From time to time, remove the 50% least active clauses.

Assuming learnt clauses:

$c1: (\neg x2 \vee \neg x3 \vee x4)$, $c2: (\neg x1 \vee \neg x4 \vee x6)$

$c3: (\neg x4 \vee x5)$, $c4: (\neg x5 \vee x7)$, $c5: (\neg x6 \vee \neg x7)$

$(\neg x1 \vee \neg x4)$

\uparrow
 $c2$

$(\neg x4 \vee \neg x6)$

\uparrow
 $c3$

$(\neg x5 \vee \neg x6)$

$c4, c5$

c1	c2	c3	c4	c5
0	1	1	1	1

Restart policy

[Gomes et al. 1998]

Motivation,

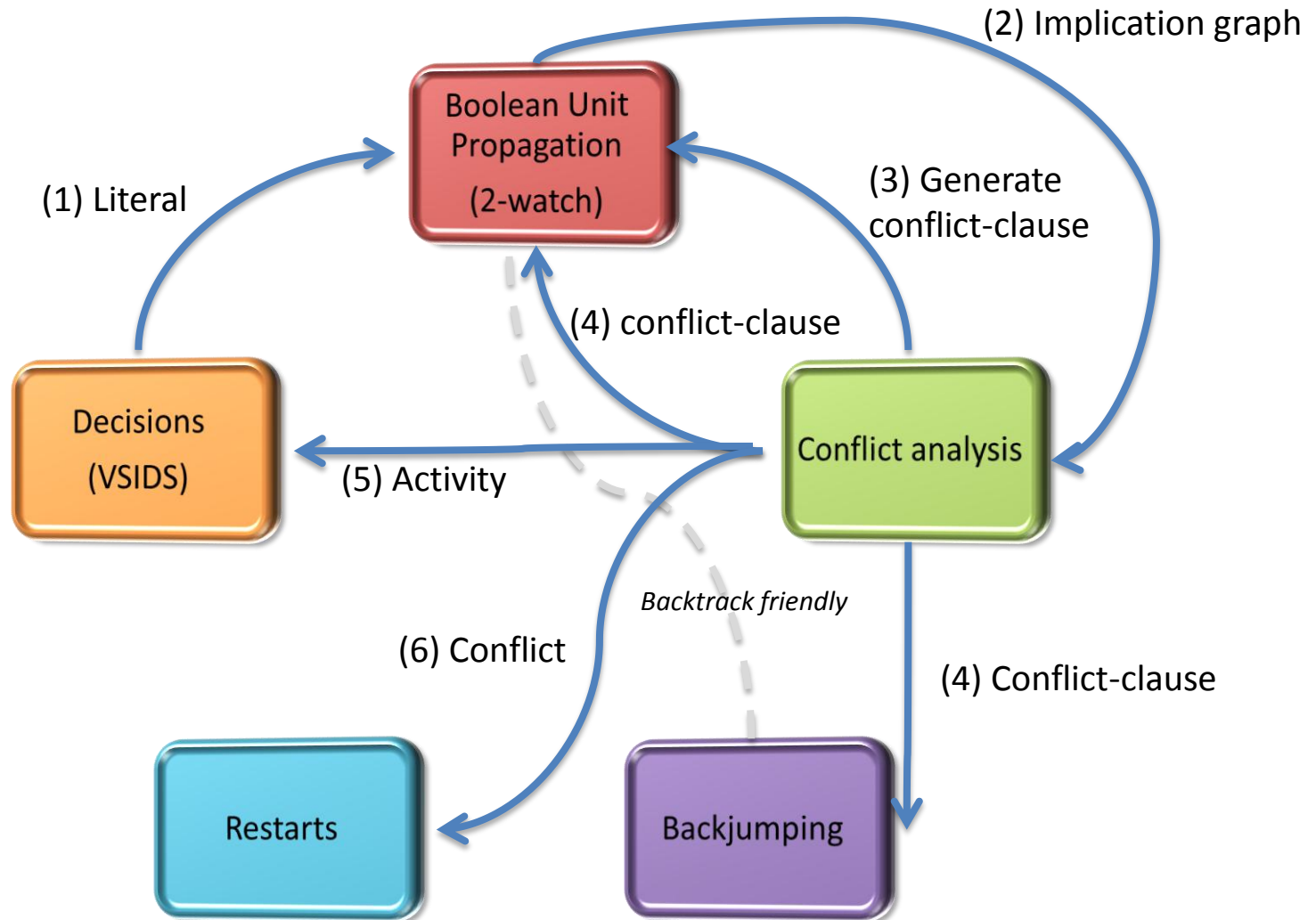
- In DFS, early wrong decisions are very expensive
heuristics are likely to be wrong there since they are poorly informed.

In modern SAT solvers,

- Knowledge is accumulated during conflict analysis: VSIDS, conflict-clauses.

-> Restart the search and focus on parts previously identified as important.

Architecture



Algorithm

```
DPLL () {  
    limit = c;  
    while (true) {  
        while (#conflicts < limit) {  
            lit = decide();  
            if (!lit) return SAT;  
            if (!BUP(lit)) {  
                cl = conflict-analysis();  
                if (!cl) return UNSAT;  
                #conflicts++;  
            }  
        }  
        undoDecisions();  
        increase(limit);  
    }  
}
```

Outline of this tutorial

1. The Propositional Satisfiability Problem (SAT)

- Theoretical importance
- Practical significance
 - Software verification

2. The state-of-the-art sequential algorithm

- Efficient BUP
- Conflict analysis
- Activity-based heuristics
- Restarts

3. Parallel SAT Solving

- Motivation
- Parallel Relaxation
- Parallel Search
 - Divide-and-conquer
 - Clause-sharing
 - Impact
 - Integration
 - Portfolio approach
 - Problems of static size clause sharing
 - Dynamic clause sharing

PARALLEL SAT SOLVING

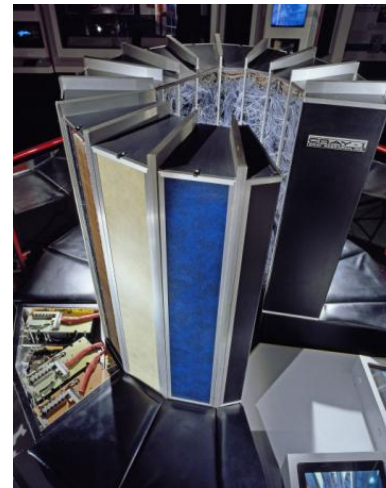
MOTIVATION

Why should we move to Parallel SAT?

- State of the art sequential algorithm looks difficult to improve (minor improvements, no orders of magnitude).
- SAT is applied to larger and more ambitious problems which cannot be solved in reasonable time.
 - ~90% of the industrial problems solved in less than 15min.
 - ~10% of the industrial problems solved in more than 1h.
- SAT is used in other formalisms,
 - Quantified Boolean Formulae
 - Satisfiability Modulo Theory

Why should we move to Parallel SAT?

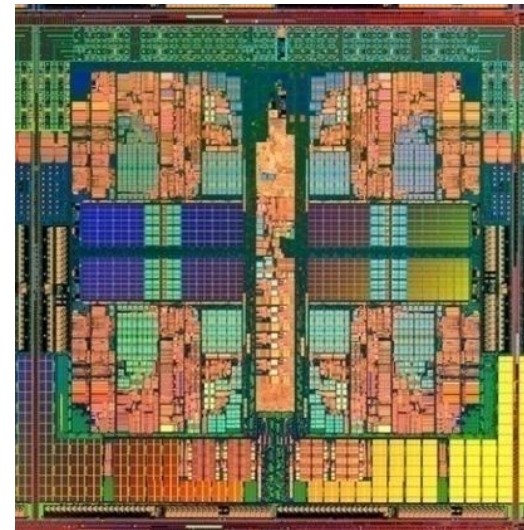
- Architectural shift:
 - Supercomputers
 - “Parallelism is the wave of the future and always will be...”
 - “Low cost” supercomputers:
 - Network of standards PCs, aka, beowulf machines



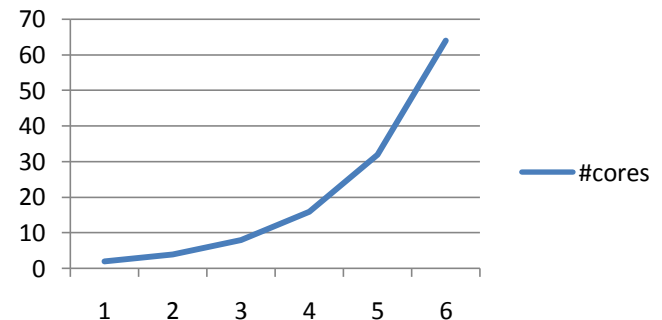
Why should we move to Parallel SAT?

- Physical laws:
 - Thermal wall: increases in processor clock frequency are slowing and in many cases frequencies are being decreased to reduce power consumption.
- Solution: multicore chips.
- Moore's law goes on:
 - Doubling the number of transistors on a chip about every two years or so.

Intel quadcore



#cores



Parallel computing concepts

- **Parallel system:** parallel algorithm + parallel architecture.
- **Scalability:** how well a parallel system takes advantage of increased computing resources.
- **Terms**
 - Sequential runtime T_s
 - Parallel runtime T_p (with p procs)
 - Parallel overhead $T_o = pT_p - T_s$
 - Speedup $S = T_s/T_p$
 - Efficiency $E = S/p$

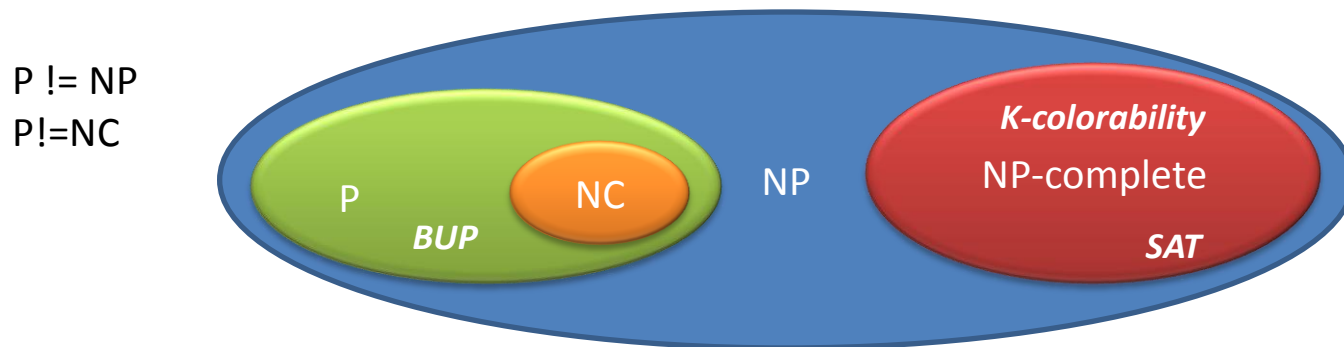
PARALLEL RELAXATION

Parallel Relaxation

- Binary Unit Propagation
 - *Unit-clause* rule: an unsatisfied clause is unit if it has exactly one unassigned literal.
 - Operates locally \implies obvious candidate for parallel algorithm??

Parallel Relaxation

- Def. class **NC** is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors.



- Theorem[Kasif 90]: Parallel Relaxation (Arc-consistency, Binary Unit Propagation) is log-space complete for P.
- Parallel algorithm (polynomial number of resources) is unlikely to improve the sequential algorithm by much.

Parallel Relaxation

- In the **worst case**, BUP is inherently sequential, i.e., Efficiency < 1.
- Example:

$$f = (x1 \vee x2) \wedge (x1 \vee \neg x2 \vee x3) \wedge (x1 \vee \neg x3 \vee x4) \wedge \dots$$

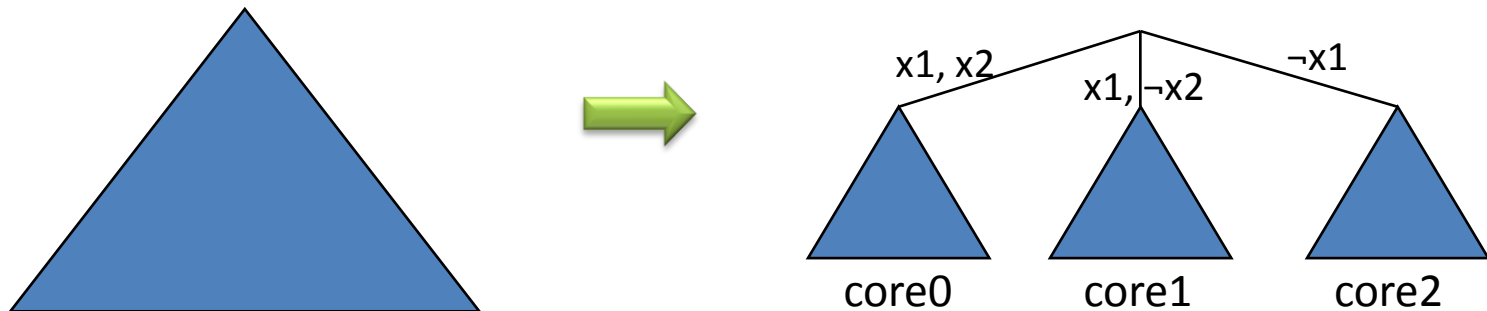
$$x1 = \text{false} \Rightarrow x2 = \text{true} \Rightarrow x3 = \text{true} \Rightarrow x4 = \text{true} \Rightarrow \dots$$

Chain of successive (sequential)
and unique implications.

PARALLEL SEARCH

Divide-and-conquer

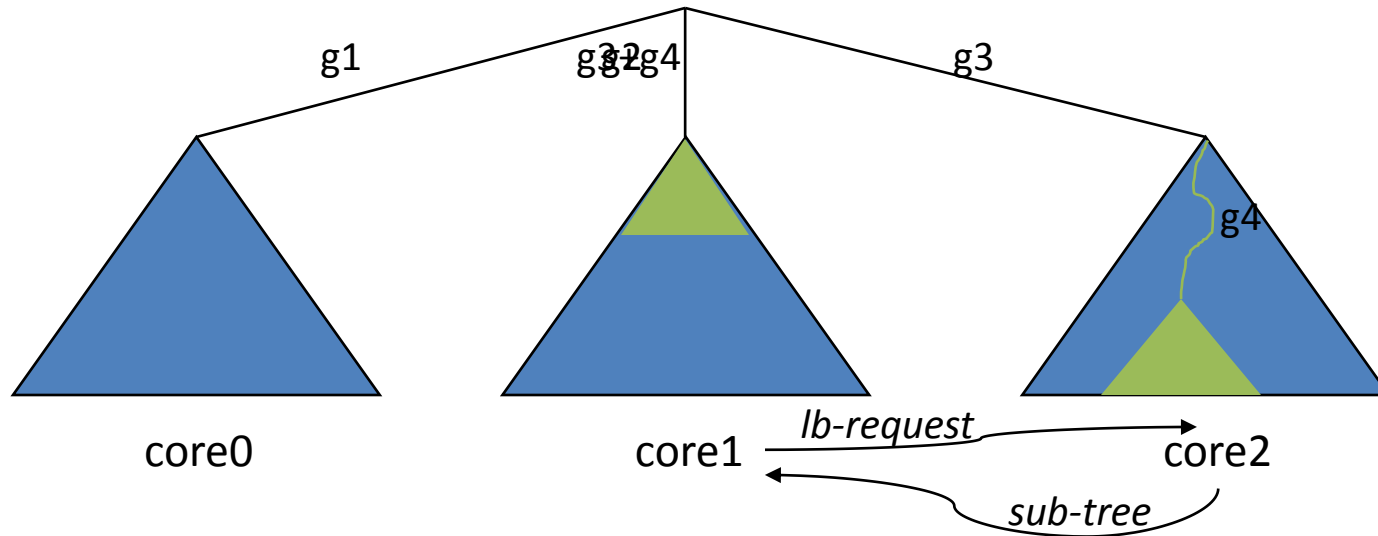
- Principle: allocate independent sub-trees to different resources.
 - Use a set of constraints (a.k.a. **guiding paths**) to split the original search space.



- Problem: load imbalance

Divide-and-conquer

- Load balancing:

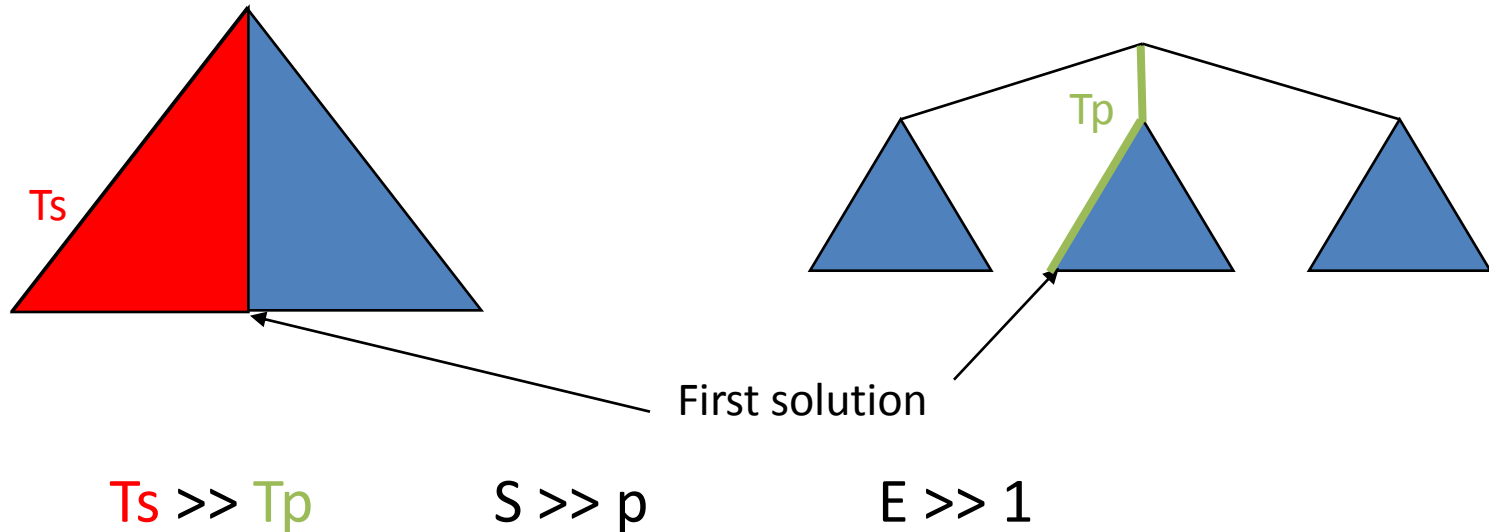


- Which possible performance?

Divide-and-conquer

- Typically in parallelism
 - Objective: divide the sequential runtime by the number of resources.
 - Linear *speed-up*, and scalability.
 - *Efficiency* very close to 1.

Speed-up anomalies in tree search



- First reported identification circa 1975 [Pruul 88]
- [Rao et al. 93]: “... sequential DFS is sub-optimal...”
 - > Interleaved DFS (sequential) [Meseguer 97]
 - > Interleaved Backtracking in Distributed Constraint Networks [Hamadi 01]

Divide-and-conquer in SAT

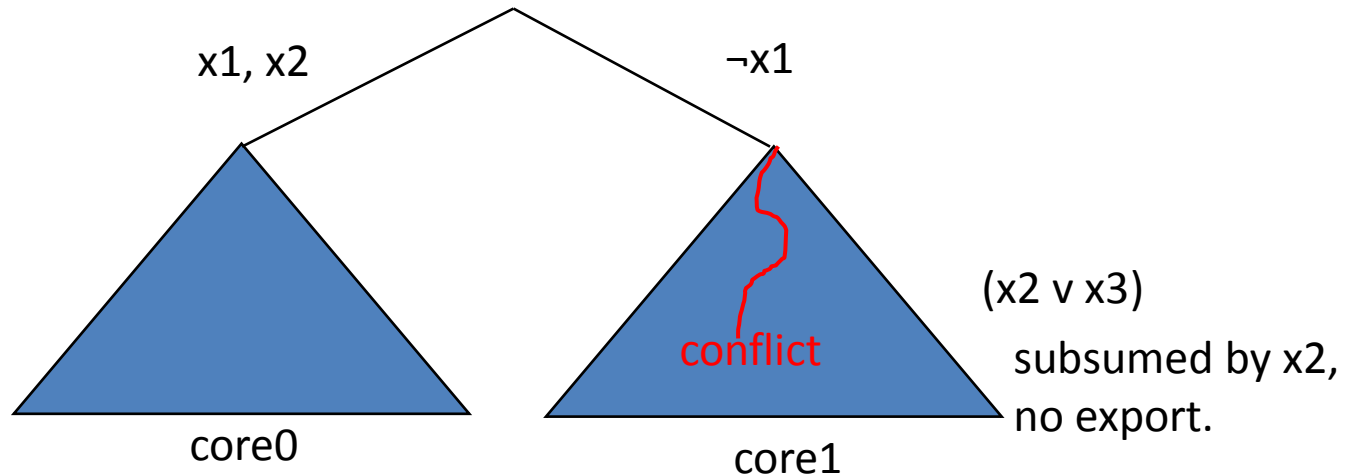
	Base algorithm	Parallel architecture	Knowledge sharing
Psato [Zhang et al. 1996]	Sato	workstations	Load-balancing
[Bohm et al. 1996]	ad-hoc	workstations	Load-balancing
Gradsat [Chrabakh et al. 2003]	zChaff	workstations	Load-balancing, clause sharing
[Blochinger et al. 2003]	zChaff	workstations	Load-balancing, restricted clause sharing
MiraXT [Lewis et al. 2007]	Minisat	multicore	Load-balancing, systematic clause sharing
Pminisat [Chu et al. 2008]	Minisat	multicore	Load-balancing, restricted clause sharing

Clause sharing

- Each core exchanges its conflict-clauses.
- Main problem: exponential number of clauses.
- Solution: export up to some **fixed size limit**.
 - > reduce the overhead
 - > moreover, the smaller the better
 - remember: $|c|=k$: removes $2^{(n-k)}$ tuples.

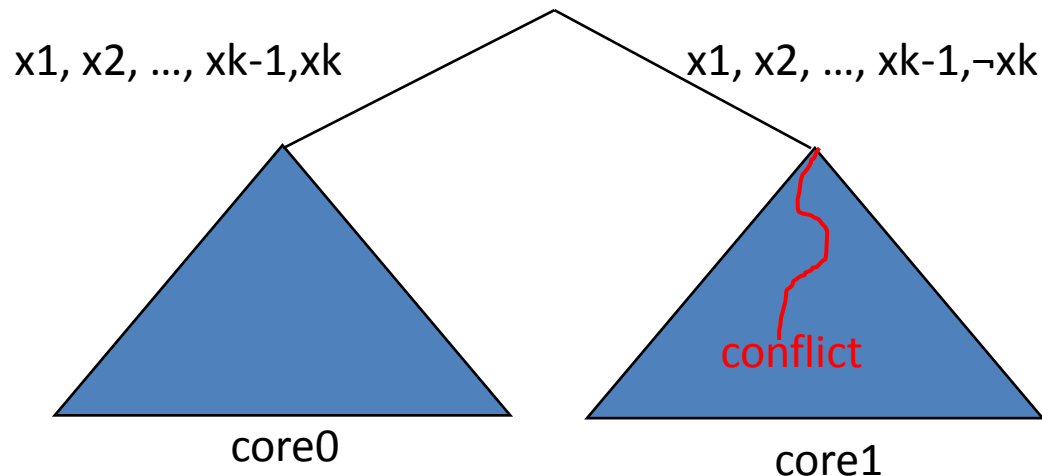
Clause sharing

- Take advantage of guiding-paths.
- [Blochinger et al. 2003]: do not export if subsumed by a guiding-path.
- Def: $c1$ *subsumes* $c2$ iff $c1 \setminus \text{in } c2$



Clause sharing

- Take advantage of guiding-paths.
- Pminisat [Chu et al. 2008]
- Size limit $k-1$



$|(\neg x_1 \vee \neg x_2 \vee \dots \vee \neg x_{k-1} \vee x_{k+1})| > k-1$, no export
 But equivalent to (x_{k+1}) in core0's context, export

Divide-and-conquer in SAT: algorithms

```

SlaveDPLL() {
1:get and enforce guiding-path;
  limit = c;
  while (!end) {
    <import foreign-clauses>;
    while (#conflicts < limit && !end) {
      <import foreign-clauses>;
      lit = decide();
      if (!lit)
        end = true;
        SAT = true;
      if (!BUP(lit)) {
        cl = conflict-analysis();
        if (!cl) goto 1;
        export cl;
        #conflicts++;
      }
    }
    undoDecisions();
    increase(limit);
  }
}

```

```

MasterDPLL() {
  produce initial guiding-paths;
  end = false;
  while (!end) {
    if (guiding-path-required())
      if (!guiding-path())
        end = true;
        SAT = false;
    <SlaveDPLL>
  }
}

```

end, SAT: shared memory variables.

Clause sharing

Integration of shared clauses: top level

- Straight forward
- Unit-clauses

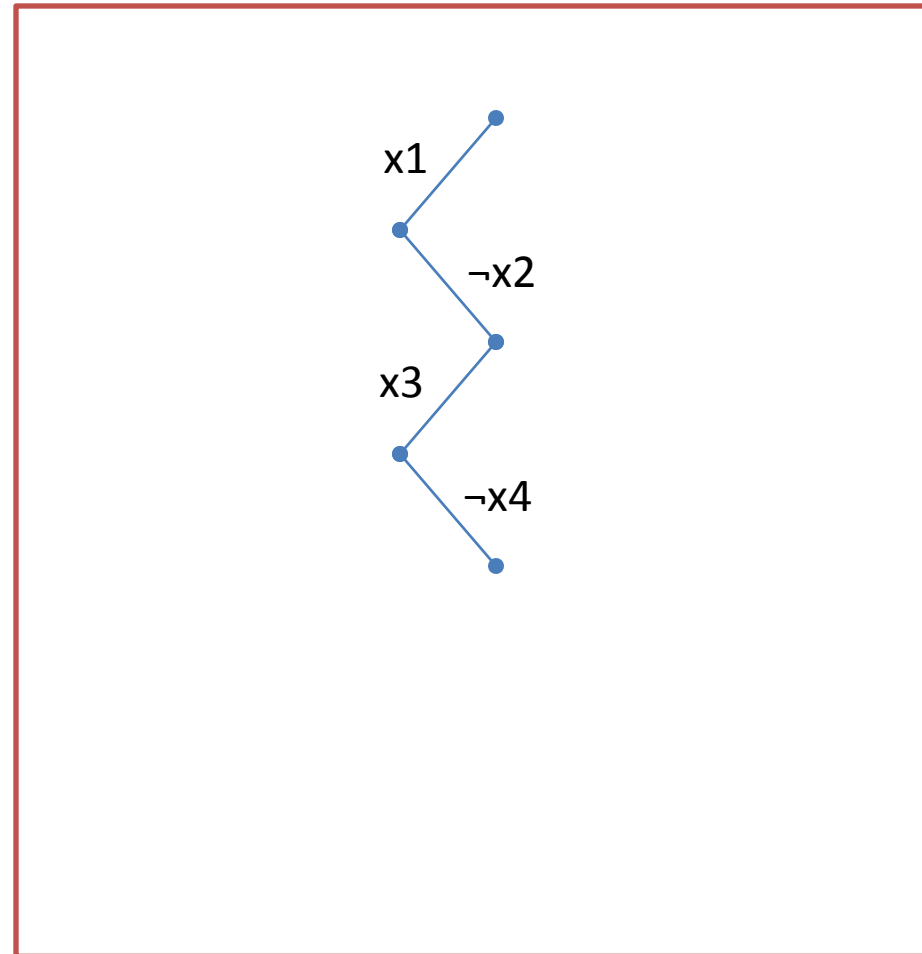
```

SlaveDPLL() {
  1: get and enforce guiding-path;
  limit = c;
  while (!end) {
    <import foreign-clauses>;
    while (#conflicts < limit && !end) {
      <import foreign-clauses>;
      lit = decide();
      if (!lit)
        end = true;
        SAT = true;
      if (!BUP(lit)) {
        cl = conflict-analysis();
        if (!cl) goto 1;
        export cl;
        #conflicts++;
      }
    }
    undoDecisions();
    increase(limit);
  }
}

```

Clause sharing

Integration of shared
clauses: on-the-fly

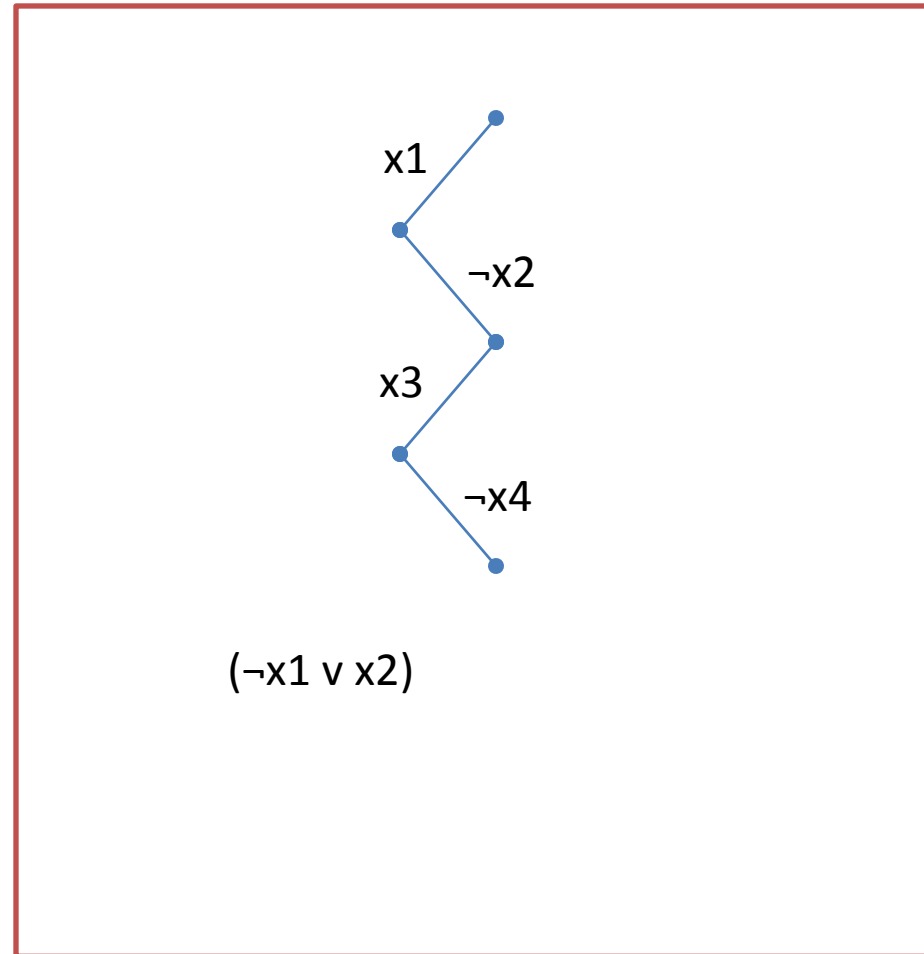


Clause sharing

Integration of shared clauses: on-the-fly

– False

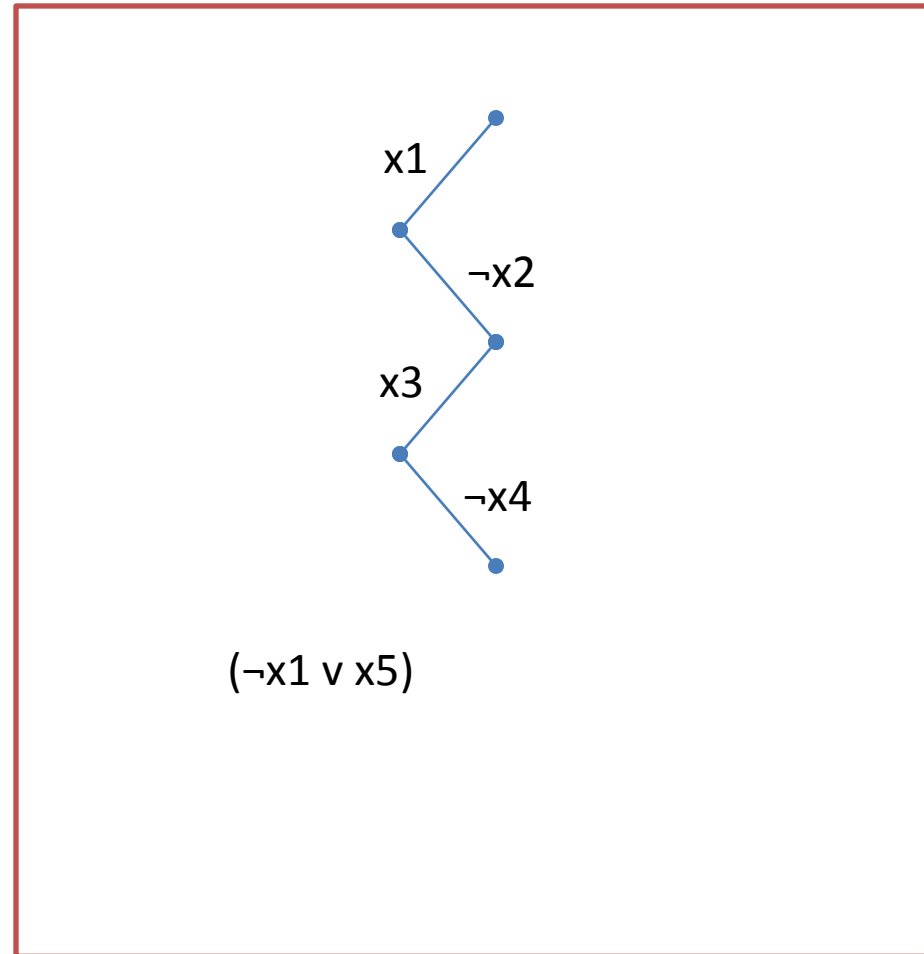
-> Conflict analysis



Clause sharing

Integration of shared clauses: on-the-fly

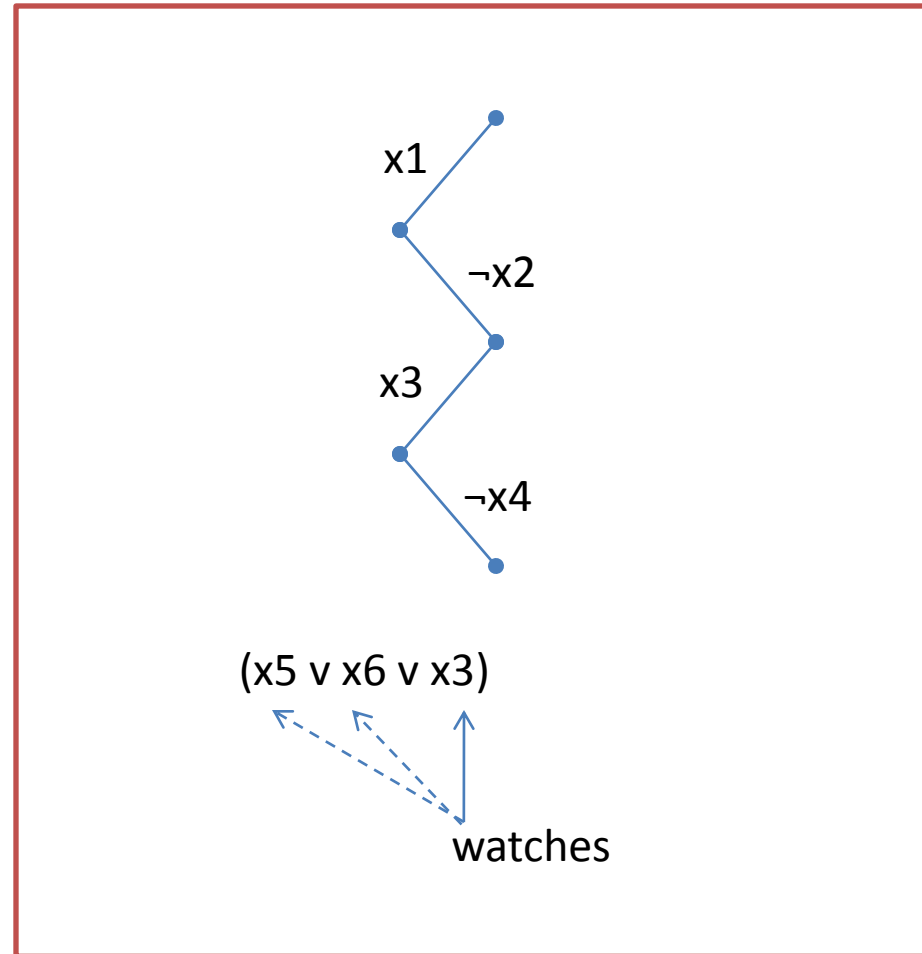
- False
 - > Conflict analysis
- Unit
 - > BUP



Clause sharing

Integration of shared clauses: on-the-fly

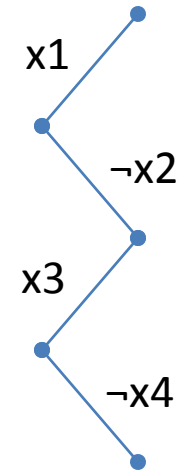
- False
 - > Conflict analysis
- Unit
 - > BUP
- Satisfied
 - > Watch the last satisfied



Clause sharing

Integration of shared clauses: on-the-fly

- False
 - > Conflict analysis
- Unit
 - > BUP
- Satisfied
 - > Watch the last satisfied
- Otherwise
 - Watch any pair of literals

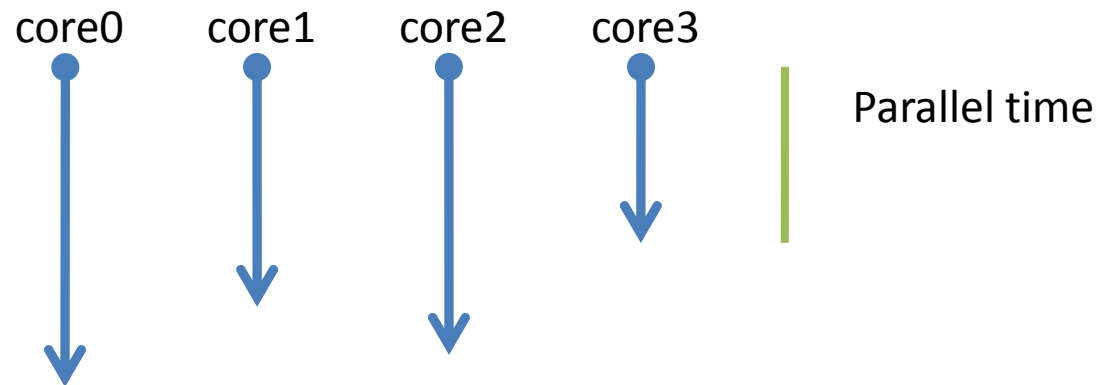


→ Divide-and-conquer: ManySAT

- [Hamadi, S. Jabbour, and L. Sais 2008]
- Observation:
 - Modern SAT solvers became highly stochastic.
 - They are sensible to their parameters, i.e., lack of robustness.
- ManySAT's principle: let several DPLLs **compete** and **cooperate** to be the first to solve a given instance.

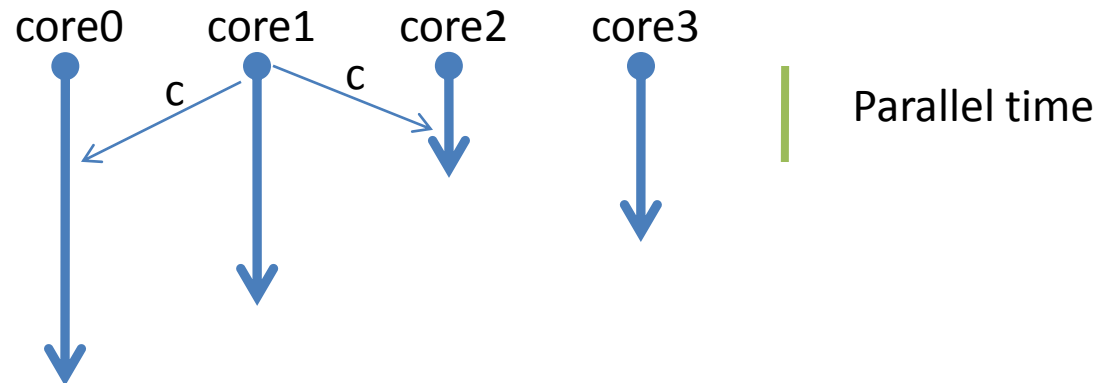
→ Divide-and-conquer: ManySAT

- No cooperation.
- Use a set of orthogonal strategies.
- Performance: **as good as the best.**



→ Divide-and-conquer: ManySAT

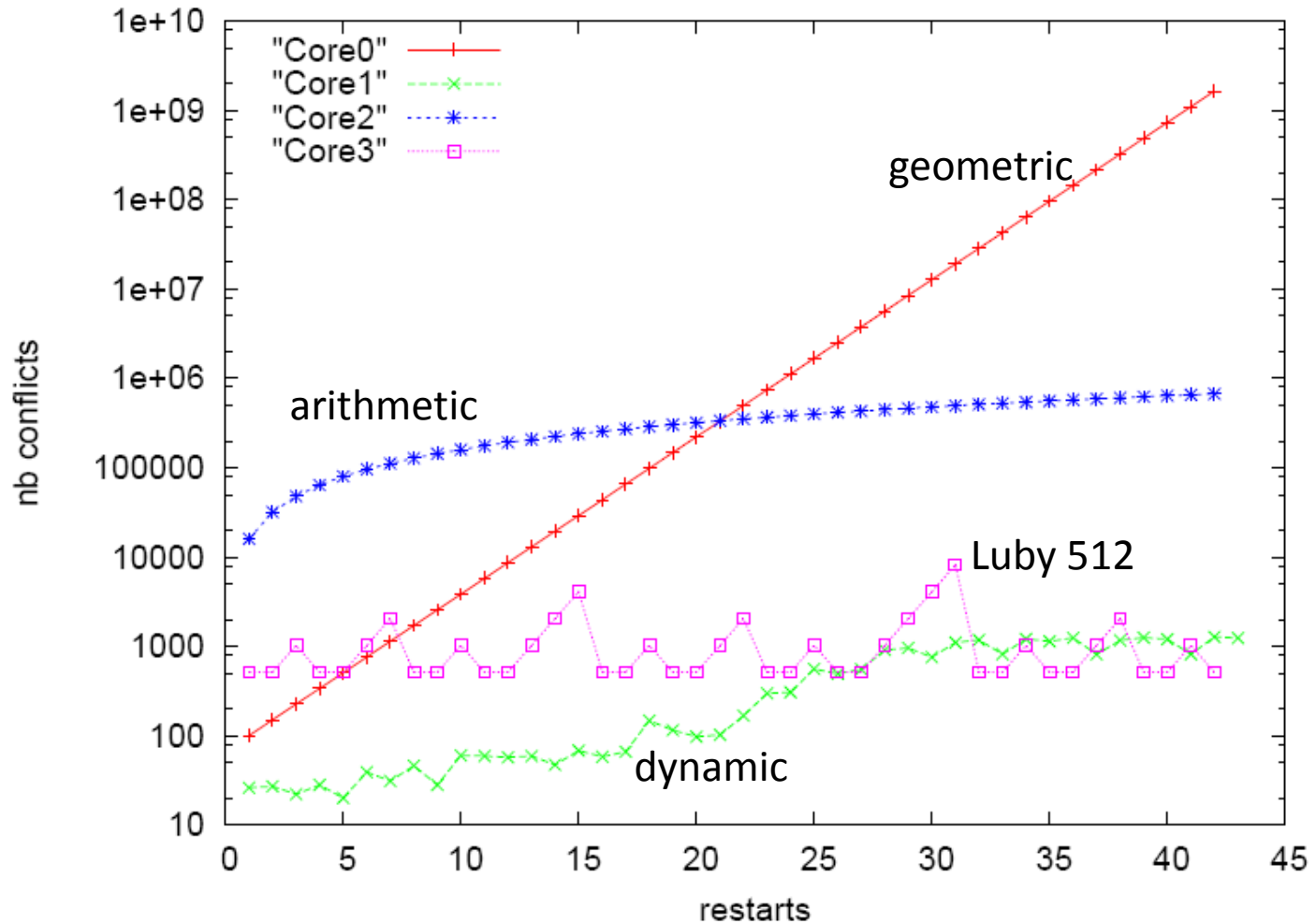
- Cooperation through clause-sharing.
- Use different but related strategies.
- Performance: **better than the best.**



ManySAT: diversification

- Restart policies
 - New: Dynamic (fast) restart policy
 - Exploits measures related to the hardness of the instance
- Polarity
 - New: Balances the number of positive and negative literals of any variable.
 - > more unit propagation
- Learning scheme
 - New: extended CDCL which exploits satisfied clauses
 - cf. SAT'08 paper

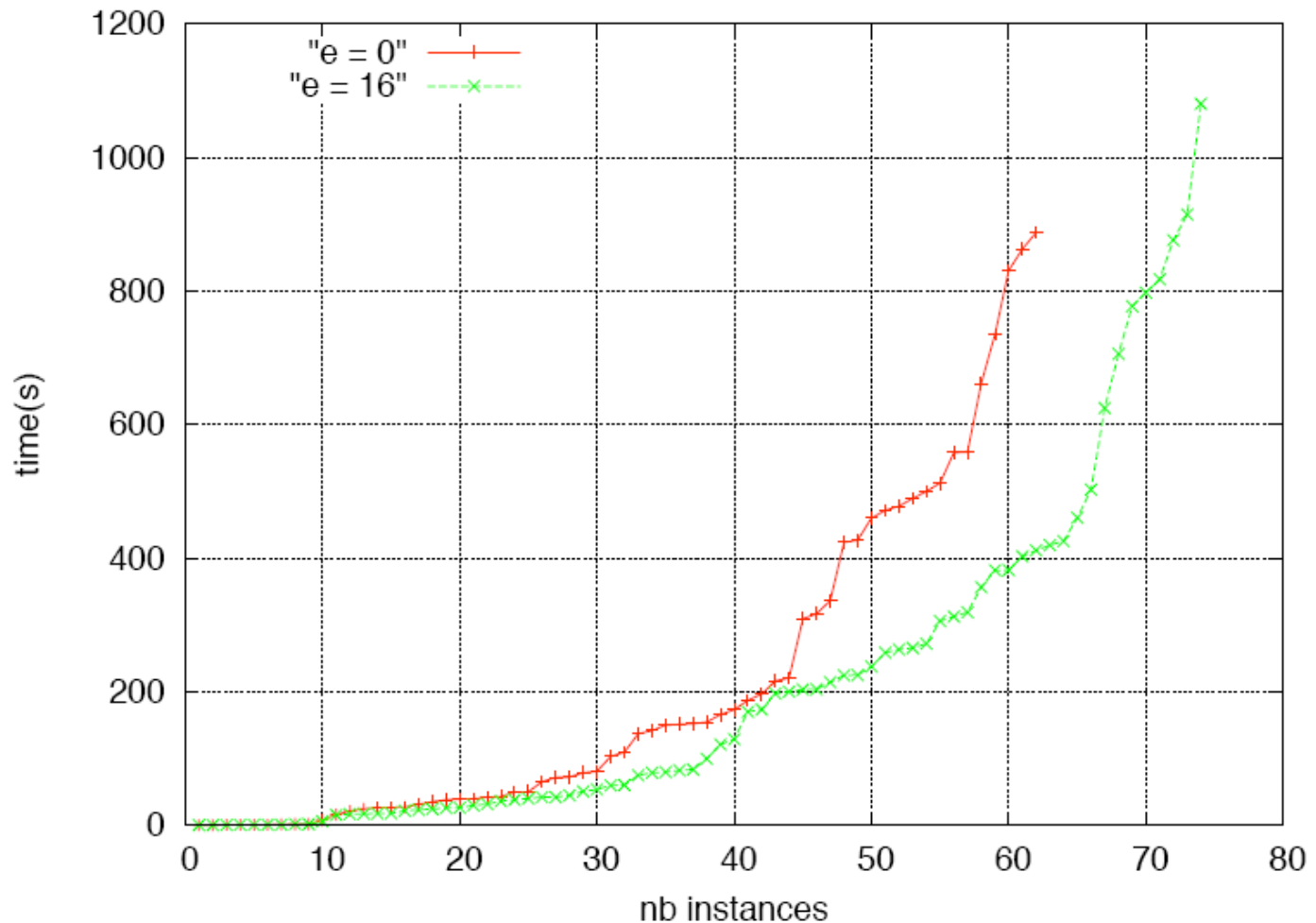
ManySAT: restart policies



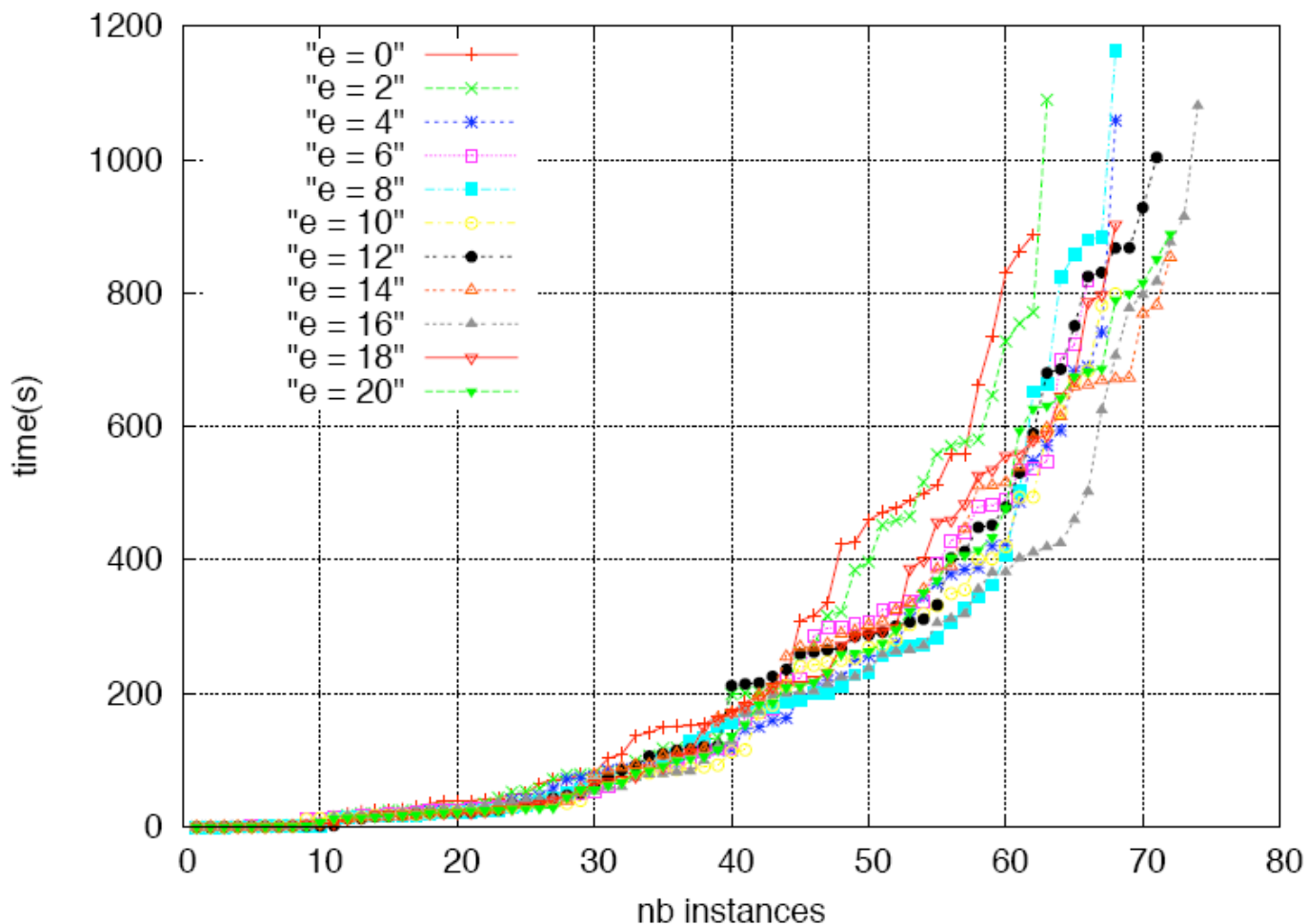
ManySAT: internals

Strategies	Core 0	Core 1	Core 2	Core 3
Restart	Geometric $x_1 = 100$ $x_i = 1.5 \times x_{i-1}$	Dynamic (Fast) $x_1 = 100, x_2 = 100$ $x_i = f(y_{i-1}, y_i), i > 2$ if $y_i > y_{i-1}$ $f(y_{i-1}, y_i) =$ $\frac{\alpha}{y_i} \times \cos(1 + \frac{y_{i-1}}{y_i}) $ else $f(y_{i-1}, y_i) =$ $\frac{\alpha}{y_i} \times \cos(1 + \frac{y_i}{y_{i-1}}) $ $\alpha = 1200$	Arithmetic $x_1 = 16000$ $x_i = x_{i-1} + 16000$	Luby 512
Heuristic	VSIDS (3% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)
Polarity	if $\#occ(l) > \#occ(\neg l)$ $l = true$ else $l = false$	Progress saving	false	Progress saving
Learning	CDCL (extended [1])	CDCL	CDCL	CDCL (extended [1])
Cl. sharing	size 8	size 8	size 8	size 8

ManySAT: performance with clause sharing



ManySAT: performance with clause sharing



ManySAT: comparative performance



- SAT-Race 2008
 - 100 industrial problems
 - 900 seconds timeout
 - 4 cores

	ManySAT	pMinisat	MiraXT
SAT	45	44	43
UNSAT	45	41	30

ManySAT: comparative performance



- SAT-Race 2008
 - vs. Minisat 2.1 (best sequential)

	ManySAT	pMinisat	MiraXT
Average speed-up	6.02	3.10	1.83
by SAT/UNSAT	8.84/3.14	4.00/2.18	1.85/1.81
Minimal speed-up	0.25	0.34	0.04
by SAT/UNSAT	0.25/0.76	0.34/0.46	0.04/0.74
Maximal speed-up	250.17	26.47	7.56
by SAT/UNSAT	250.17/4.74	26.47/10.57	7.56/4.26

ManySAT

- University of Munich, parity games.
- IBM Germany Research & Development GmbH in Boeblingen, Bounded Model Checking.
- Microsoft,
 - Z3 SMT Solver -> //Z3
 - Joint work with Leonardo De Moura and Christoph Wintersteiger.
 - ‘In production’ for software verification in coming weeks.

Recap

1. The Propositional Satisfiability Problem (SAT)

- Theoretical importance
- Practical significance
 - Software verification

2. The state-of-the-art sequential algorithm

- Efficient BUP
- Conflict analysis
- Activity-based heuristics
- Restarts

3. Parallel SAT Solving

- Motivation
- Parallel Relaxation
- Parallel Search
 - Divide-and-conquer
 - Clause-sharing
 - Impact
 - Integration
 - Portfolio approach
 - Problems of static size clause sharing
 - Dynamic clause sharing

Things to take home

- SAT is important for software verification which is crucial for modern life.
- Sequential SAT solvers have been heavily improved in the last decade.
- A few remaining paths to try, don't expect another 3 orders of magnitude gain.
- **Parallelism is happening now: multicore PCs**
 - Think 'parallel' for any new algorithm.
 - Number of cores is growing exp.
 - Importance of knowledge sharing: better than the best.

BIBLIOGRAPHY

Research

- Theory
 - [Cook 1971] Cook, Stephen "The complexity of theorem proving procedures". Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158.
 - [Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher editors, Complexity of Computer Computations, pages 85–103, New York, 1972. Plenum Press.
- Software testing
 - Automating Software Testing Using Program Analysis, Patrice Godefroid; Peli de Halleux; Michael Levin; Aditya Nori; Sriram K. Rajamani; Wolfram Schulte; Nikolai Tillmann, MSR-TR-2008-119
 - Building a better bug-trap, The Economist, Jun 19th 2003
 - The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST, Planning report 02-3
 - [Rustan et al.] Program Verification Using the Spec# Programming System ETAPS tutorial 2008.
- Sequential SAT
 - [Moskewicz et al. 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik: Chaff: Engineering an Efficient SAT Solver. DAC 2001, pp 530-535
 - [Marques Silva et al. 1996] João P. Marques Silva, Karem A. Sakallah: GRASP - a new search algorithm for satisfiability. ICCAD 1996: 220-227
 - [Gomes et al., 1998] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In Proc. AAAI'98, pages 431–437, 1998.
- Parallel relaxation
 - [Kasif 90] Simon Kasif: On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. Artif. Intell. 45(3): 275-286 (1990)
 - [Hamadi 99] Youssef Hamadi: Optimal Distributed Arc-Consistency. CP 1999: 219-233
- Parallel search
 - [Pruul 88] Branch-and-bound and parallel computation: A historical note – Pruul, Nemhauser – 1988
 - [Meseguer 97] Pedro Meseguer: Interleaved Depth-First Search. IJCAI 1997: 1382-1387
 - [Hamadi 01] Youssef Hamadi: Interleaved Backtracking in Distributed Constraint Networks. ICTAI 2001: 33-41
- Parallel SAT
 - [Zhang et al. 96] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation, 21:543-560, 1996.
 - [Bohm et al. 96] Max Bohm and Ewald Speckenmeyer. A fast parallel sat-solver with efficient workload balancing. Ann. Math. Artif. Intell., 17(3-4):381-400, 1996.
 - [Chrabakh et al. 03] Wahid Chrabakh and Rich Wolski. GrADSAT: A parallel sat solver for the grid. Technical report, UCSB Computer Science Technical Report Number 2003-05, 2003.
 - [Blochinger et al. 03] W. Blochinger, C. Sinz, and W. Kuchlin. Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing, 29(7):969-994, 2003.
 - [Lewis et al. 08] M. Lewis, T. Schubert, and B. Becker. Multithreaded sat solving. In 12th Asia and South Pacif Design Automation Conference, 2007. G. Chu and [Chu et al. 08] Geoffrey Chu and P. J. Stuckey. Pminisat: a parallelization of minisat 2.0. Technical report, Sat-race 2008, solver description, 2008.
 - [Hamadi et al., 2008] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: Solver description. In SAT race, 2008.

Thank you!

Slides will be online:

<http://www.intelligent-optimization.org/LION3/>