Improving Search Space Splitting for Parallel SAT Solving

Ruben Martins  Vasco Manquinho  Inês Lynce

INESC-ID/IST, Technical University of Lisbon, Portugal

28 October 2010
Outline

1. Sequential SAT Solving
2. Parallel SAT Solving: Portfolio vs Search Space Splitting
3. Improving Search Space Splitting
4. Experimental Results
5. Conclusions
Preliminaries

- Propositional Satisfiability (SAT):
  - A literal $l_i$ is either a Boolean variable $x_i$ or $\overline{x}_i$;
  - A clause $\omega = \bigvee_i l_i$:
    e.g. $\omega_1 = (x_1 \lor \overline{x}_2)$; $\omega_2 = (x_2 \lor x_3)$; $\omega_3 = (\overline{x}_2 \lor \overline{x}_3)$.
  - CNF formula $\varphi = \bigwedge_j \omega_j$:
    e.g. $\varphi = (\omega_1 \land \omega_2 \land \omega_3)$.
  - SAT problem is to decide if $\varphi$ is satisfiable:
    e.g. $\varphi$ is satisfied when $x_1 = 1$, $x_2 = 1$ and $x_3 = 0$. 
Preliminaries

- Since the mid 90s SAT solvers have shown remarkable improvements;
- Due to these improvements SAT solvers have been successfully applied to many practical applications:
  - Hardware and Software model checking;
  - Planning;
  - Cryptanalysis;
  - Computational Biology;
  - etc.
Sequential SAT Solving

**INPUT**: CNF formula $\varphi$;

**OUTPUT**: SAT if an assignment is found; UNSAT otherwise.
Sequential SAT Solving

**INPUT:** CNF formula $\varphi$;

**OUTPUT:** SAT if an assignment is found; UNSAT otherwise.

- VSIDS (Variable State Independent Decaying Sum) heuristic:
  - Each literal has an activity counter;
  - Each literal that occurs in a no-good has its activity increased;
  - At each call, the highest-value unassigned literal is chosen.
Sequential SAT Solving

**INPUT:** CNF formula $\varphi$;

**OUTPUT:** SAT if an assignment is found; UNSAT otherwise.

- **Unit Clause Rule:**
  - Given a unit clause, its only unassigned literal must be assigned value 1 for the clause to be satisfied.

- **Unit Propagation:**
  - Iterated application of the unit clause rule;
  - If an unsatisfied clause is identified it returns “Conflict”.

Assign Branching Variable

0

Unit Propagation

$X_5$
Sequential SAT Solving

**INPUT:** CNF formula $\varphi$;

**OUTPUT:** SAT if an assignment is found; UNSAT otherwise.

![Diagram of a SAT solving process]
Sequential SAT Solving

INPUT: CNF formula $\varphi$

OUTPUT: SAT if an assignment is found; UNSAT otherwise.

Conflict Analysis

Conflict
Sequential SAT Solving

**INPUT**: CNF formula $\phi$;

**OUTPUT**: SAT if an assignment is found; UNSAT otherwise.

![Diagram of SAT solving process]

**Conflict Analysis**

**Backtrack**
Sequential SAT Solving

- **INPUT**: CNF formula $\varphi$;
- **OUTPUT**: SAT if an assignment is found; UNSAT otherwise.
Parallel SAT Solving: Portfolio vs Search Space Splitting

Portfolio

\( \phi \)

Search Space

T1, T2, T3, T4

Different Heuristics
Same Search Space

Search Space Splitting

\( \phi \)

S1
S2
S3
S4

T1
T2
T3
T4

Same Heuristics
Different Search Spaces
Search Space Splitting

• The guiding paths describe the current state of the search process;
• The unused guiding paths are stored in the work queue;
• If a thread proves that its current subspace is unsatisfiable, it gets a new subspace from the work queue and continues searching;
• A dynamic work stealing procedure guarantees that work is available for all threads.
Search Space Splitting

- The guiding paths describe the current state of the search process;
- The unused guiding paths are stored in the work queue;
- If a thread proves that its current subspace is unsatisfiable, it gets a new subspace from the work queue and continues searching;
- A dynamic work stealing procedure guarantees that work is available for all threads.
Search Space Splitting

- The guiding paths describe the current state of the search process;
- The unused guiding paths are stored in the work queue;
- If a thread proves that its current subspace is unsatisfiable, it gets a new subspace from the work queue and continues searching;
- A dynamic work stealing procedure guarantees that work is available for all threads.
Search Space Splitting

- The guiding paths describe the current state of the search process;
- The unused guiding paths are stored in the work queue;
- If a thread proves that its current subspace is unsatisfiable, it gets a new subspace from the work queue and continues searching;
- A dynamic work stealing procedure guarantees that work is available for all threads.
Overview: Portfolio vs Search Space Splitting

Portfolio:

- Different strategies for each SAT solver cooperate when solving the same search space:
  - Covers the space of search strategies;
  - Each solver has a complete view of the formula;
  - For a large number of cores it can be hard to find diverse viewpoints that provide orthogonal performance;
  - State-of-the-art multicore SAT solvers use this approach.
Overview: Portfolio vs Search Space Splitting

Search Space Splitting:

- Each SAT solver solves a different search subspace until no more subspaces exist or satisfiability has been proved:
  - Each solver has only a partial view of the formula;
  - It is not clear how to effectively choose the partition variables;
  - Load balancing requires an overhead on the dynamic work stealing procedure;
Choosing the Partition Variables:

1. Collecting VSIDS information:
   - Each thread runs a sequential SAT algorithm until $k$ conflicts are reached;
   - After $k$ conflicts the VSIDS heuristic of each thread can be analyzed in order to determine the partition variables;

- Weak Portfolio:
  - The VSIDS information is increased by running each thread with a different initial order of the VSIDS heuristic.

- Advantages of the weak portfolio:
  - Some instances can be solved during this stage;
  - More information is collected about the variables.
Improving Search Space Splitting

Choosing the Partition Variables:

2. Using the VSIDS information for choosing the partition variables:

\[
\text{VSIDS}(T_1): \begin{array}{ccccc} x_1 & x_3 & x_2 & x_5 & x_4 \end{array} \\
\text{VSIDS}(T_2): \begin{array}{ccccc} x_3 & x_2 & x_1 & x_5 & x_4 \end{array} 
\]
Improving Search Space Splitting

Choosing the Partition Variables:

2. Using the VSIDS information for choosing the partition variables:

\[
\begin{align*}
\text{VSIDS}(T_1): & \quad x_1 & x_3 & x_2 & x_5 & x_4 \\
\text{Score:} & \quad 1 & 2 & 3 & 4 & 5 \\
\text{VSIDS}(T_2): & \quad x_3 & x_2 & x_1 & x_5 & x_4 \\
\text{Score:} & \quad 1 & 2 & 3 & 4 & 5
\end{align*}
\]

- For each variable is given a score from 1 to \(n\) according to its position;
Improving Search Space Splitting

Choosing the Partition Variables:

2 Using the VSIDS information for choosing the partition variables:

<table>
<thead>
<tr>
<th>VSIDS($T_1$):</th>
<th>$x_1$</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSIDS($T_2$):</th>
<th>$x_3$</th>
<th>$x_2$</th>
<th>$x_1$</th>
<th>$x_5$</th>
<th>$x_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VSIDS($T_1 + T_2$):</th>
<th>$x_3$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_5$</th>
<th>$x_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Score:</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

- The final score of each variable is the sum of its score in each thread;
- The first $n$ variables with lowest score are chosen as partition variables and are used to create the initial $2^n$ guiding paths.
Improving Search Space Splitting

Hybrid Heuristic:

1. Preventing load balance issues:
   a) If a thread $t$ is searching for more than $k$ conflicts;
   b) And at least half of the work pool has guiding paths that were created by $t$;

   • Then this means that the subspace of the thread $t$ is dominating the search and can cause load balance issues.

2. Increase diversification of the search:
   a) If a thread is searching for more than $z$ conflicts ($z >> k$);

   • Then this means that the search is going on for some time, and at this point a diversification of the search tends to lead to better results.
SAT4J/

- SAT4J is implemented in Java:
  - Even though SAT4J is not as efficient as other SAT solvers it is one of the most popular SAT solvers;
  - A parallel version of SAT4J can therefore be useful for many users.

- SAT4J/ is a parallelization of SAT4J 2.1 (sequential version):
  - Each thread maintains its own local clause database;
  - Clauses are not shared between threads.

- Although clause sharing increases the performance of a solver, without it we can have a better understanding of the impact of our heuristics.
Four versions of SAT4J were implemented:

- **no-Info:**
  - Search space splitting approach based on guiding paths;
  - Dynamic work stealing with a central queue of work which is topped up by the longer running thread;
  - The partition variables are chosen randomly.

- **Info:**
  - Uses a short initial stage of weak portfolio;
  - After this stage, the partition variables are chosen using the VSIDS information of all threads.
Four versions of SAT4J were implemented:

- **Pfolio**:
  - Uses a portfolio of SAT algorithms;
  - Each thread has a different combination of the following strategies:
    (1) restart, (2) polarity and (3) learning simplification.

- **Hybrid**:
  - Starts by using the search space splitting approach present in *Info*;
  - The hybrid heuristics switches from *Info* into *Pfolio*;
  - When switching to a portfolio mode:
    - All guiding paths are merged into a unique guiding path that has the literals that were common to all guiding paths;
    - The learnt clauses are kept from the search space splitting stage.
Experimental Results

• Benchmarks: 82 instances from the applications category of the SAT competition 2009 such that:
  • SAT4J 2.1 was able to solve in more than 180 seconds;
  • SAT4J 2.1 was unable to solve but MiniSAT 2.1 was able to solve within 1,200 seconds.

• The set of benchmarks is challenging for SAT4J and interesting for parallel testing;
Experimental Results

- Intel Core i7 CPU 930 (2.80 Ghz, 6GB) running Ubuntu 10.04 LTS;
- Timeout: 3,600 seconds (wall clock time);
- All versions of SAT4J// were run with 4 threads:
  - Each version of SAT4J// was run 3 times on each instance;
  - The runtimes shown in this section are the median of the successful runs for each instance;
  - An instance was considered solved if it could be solved in at least one run.
Experimental Results

- Number of instances solved by each approach:

<table>
<thead>
<tr>
<th></th>
<th># Inst</th>
<th>Seq</th>
<th>no-Info</th>
<th>Info</th>
<th>Pfolio</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>25</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>UNSAT</td>
<td>57</td>
<td>43</td>
<td>42</td>
<td>42</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>Total</td>
<td>82</td>
<td>59</td>
<td>59</td>
<td>61</td>
<td>64</td>
<td>65</td>
</tr>
</tbody>
</table>

- *Info* can solve more 2 instances than *no-Info*, showing the importance of the partition variables;

- *Hybrid* can solve more 1 instance than *Pfolio*, suggesting that a hybrid approach can outperform a pure portfolio approach.
Experimental Results

- Runtimes for *no-Info* and *Info*: 

![Graph showing runtimes for no-Info and Info]
Experimental Results

- Runtimes for *Pfolio* and *Hybrid*:

![Graph showing runtimes comparison between Pfolio and Hybrid]
Conclusions

• Portfolio approaches currently dominate multicore SAT solvers;

• Experimental results show that:
  • Heuristically choosing the partition variables leads to clear improvements in search space splitting;
  • A hybrid approach between search space splitting and portfolio can lead to better results than a pure portfolio approach.

• This provides a strong stimulus for further exploration of hybrid solutions.

• As future work, we propose to:
  • Extend the use of the VSIDS heuristic of all threads to guide the search during runtime;
  • Improve the hybrid heuristic.