SAT solver essentials, SAT modeling
Parallel SAT

Gilles Audemard

VTSA School - Liege - 2021

Thanks to N. Szczepanski and L. Simon
Parallel solving methods

- More and more difficult to improve SAT solvers
- Use multi-core / cloud computing is an issue to improve SAT solvers
- However, parallelization of SAT solvers is a difficult challenge
- Many attempts during last decade

<table>
<thead>
<tr>
<th>Portfolio</th>
<th>Multi-core</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PLINGELING</td>
<td>D-Syrup</td>
</tr>
<tr>
<td></td>
<td>CRYPTOMINISAT</td>
<td>HORDESAT</td>
</tr>
<tr>
<td></td>
<td>SYRUP</td>
<td></td>
</tr>
<tr>
<td>Divide and conquer</td>
<td>TREEGELING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PMINISAT</td>
<td>DOLIUS</td>
</tr>
<tr>
<td></td>
<td>PCASSO</td>
<td>AMPHAROS</td>
</tr>
</tbody>
</table>
Issues related to parallelism

Sequential algorithm

Parallel algorithm

Division of the global tasks in subtasks
Issues related to parallelism

**Sequential algorithms**

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

**Parallel Algorithm**

- $u_1$: 1, 6, 2, 3, 4
- $u_2$: 5, 6
- $u_3$: 9, 10, 7, 8, 11, 12

**Dependency problems**
Issues related to parallelism

Sequential algorithm

Parallel algorithm

Slowdown due to communications
Issues related to parallelism

Sequential algorithm

Parallel algorithm

Load balancing
Issues related to parallelism

Sequential algorithm

Parallel algorithm

Redundant work
Not so easy to parallelize SAT solvers (in a efficient way)
- Bad idea: parallelisation of the BCP engine
- 2 main approaches: portfolio and divide & Conquer
Portfolio approach

Each solver works on the same (original) formula

The first that successes is the winner

Allow to easily share clauses.

Clauses sharing is essential!
Divide and Conquer approach

- Split the search space
- Clauses sharing needs to use assumptions
- The end of the search depends on the status of the formula

```
Problem Initial
\[ \sum \]

Sub - Problem \[ \sum_1 \]
Solver

Sub - Problem \[ \sum_2 \]
Solver

Sub - Problem \[ \sum_3 \]
Solver

Sub - Problem \[ \sum_n \]
Solver

\[ \vdots \]

Clauses Sharing

\[ \vdots \]

SAT

UNSAT

UNSAT

UNSAT

SAT
```

SAT - Encodings

VTSA School - Liege - 2021
Divide and Conquer approach

- Split the search space
- Clauses sharing needs to use assumptions
- The end of the search depends on the status of the formula
Each thread has its own glucose solver

Each solver deals with the whole formula $\Sigma$ to solve

The first solver that produces an answer is the winner (the search is ended)
Syrup: Many glucose in parallel

- Each thread has its own glucose solver
- Each solver deals with the whole formula $\Sigma$ to solve
- The first solver that produces an answer is the winner (the search is ended)
- Communication: learnt clauses
- Dedicated strategy to import and export clauses
Many Cores, many more problems

Sharing clauses in parallel has many drawbacks

- Imported clauses can be bad (noise, wrong way, . . . )
- Imported clauses can be subsumed / useless
- Imported clauses can dominate learnt clauses
- Each thread has to manage many more clauses
- Many side effects on all core components

Parallel solvers limit the number of shared clauses (criteria: size, lbd...)
Many useless clauses even in single engine solvers

- x-axis: Number of conflicts
- y-axis: Useless clauses in final proof (UNSAT formulas)
How many times clauses are seen in conflicts

- **x-axis**: Number of conflicts
- **y-axis**: Number of clauses seen at least $Z$ times

---

![Graph showing the relationship between the number of conflicts and the number of clauses seen at least $Z$ times.](image)

- Seen $Z=1$ time
- L.R. Seen 1 time ($m=0.91$)
- Seen $Z=2$ times
- L.R. Seen 2 times ($m=0.34$)
- Seen $Z=3$ times
- L.R. Seen 3 times ($m=0.22$)
- Seen $Z=4$ times
- L.R. Seen 4 times ($m=0.17$)
Outcomes

A lot of useless clauses even in a single engine!

- In parallel, this situation will be even worse!
- Why sending a clause that is even not locally interesting?
- **We will consider clauses seen 2-times** (only 34% of learnt clauses)
  Already filter out the majority of clauses
- How to efficiently detect them?
  Is there a window of recent clauses to check?
  Can we check only recent learnt clauses (and save time)?
Lazy Exportation of Clauses

Clauses are sent during conflict analysis

- We only export clauses when seen 2 times in conflict analysis
- and
- Clauses with LBD $\leq$ median(LBD) and size $\leq$ average(SIZE)
- Limits updated at each clause database cleaning
- Unary clauses and very glue clauses are immediately sent

Lazy because we wait to have a good chance of local interest before considering sending it
Lazy Importation of Clauses

Problem of clauses importation
- can destroy the current search effort
- clauses can be redundant
- many clauses to manage (performance impact)

How to be sure an imported clause is interesting before considering it?
Idea: put it in probation

- Imported clauses are put in a 1-Watched scheme
- Will be promoted to a 2-Watched scheme only if found empty

Other Advantages

- Less efforts for propagations
- Can be seen as a dynamic Freezing/Reactivating strategy
- Clauses are imported with a local (and correct) LBD value
How many promotions?

- x-axis: Number of imported clauses (sum over 8 threads)
- y-axis: Number of promoted clauses (sum over 8 threads)
We develop a generic library for designing parallel solvers (C++)

https://github.com/crillab/pfactory

Powerful communication algorithm

allow to share more clauses without penalty
## Results

### SAT Competition

<table>
<thead>
<tr>
<th>Solver</th>
<th>2015 (100)</th>
<th>2018 (300)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SAT</td>
<td>UNSAT</td>
</tr>
<tr>
<td>GLUCOSE</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>Painless-GLUCOSE</td>
<td>42</td>
<td>32</td>
</tr>
<tr>
<td>Syrup</td>
<td>43</td>
<td>32</td>
</tr>
<tr>
<td>pFactory-GLUCOSE</td>
<td>49</td>
<td>32</td>
</tr>
</tbody>
</table>