SAT solver essentials, SAT modeling
Parallel SAT

Gilles Audemard

VTSA School - Liege - 2021

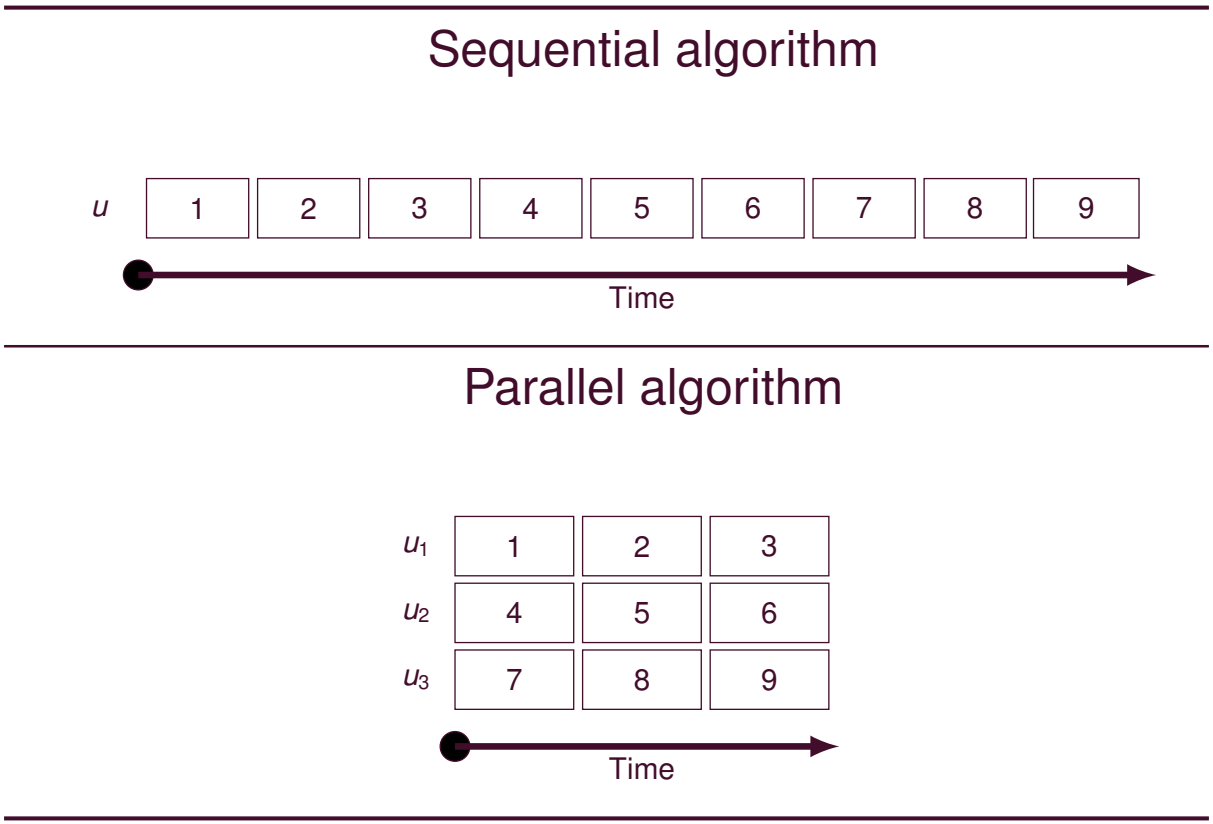
Thanks to N. Szczepanski and L. Simon

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

	Multi-core	Distributed
Portfolio	PLINGELING CRYPTOMINISAT SYRUP	D-Syrup HORDESAT
Divide and conquer	TREEGELING PMINISAT PCASSO	DOLIUS AMPHAROS

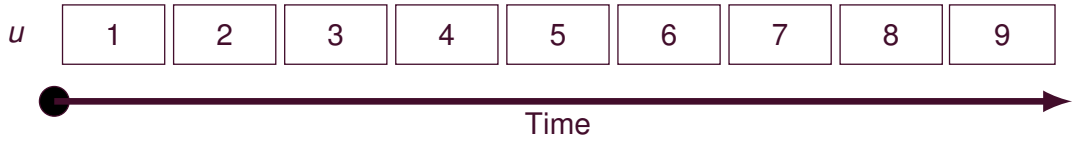
Issues related to parallelism



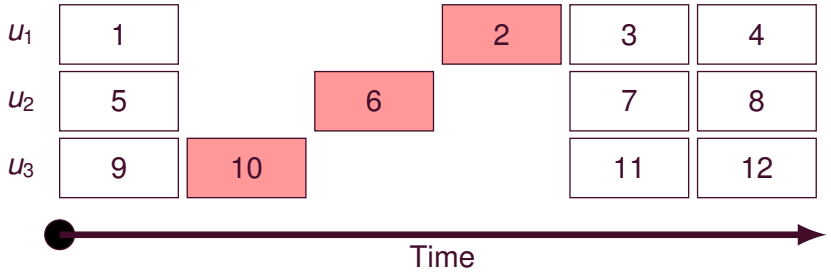
Division of the global tasks in subtasks

Issues related to parallelism

Sequential algorithms

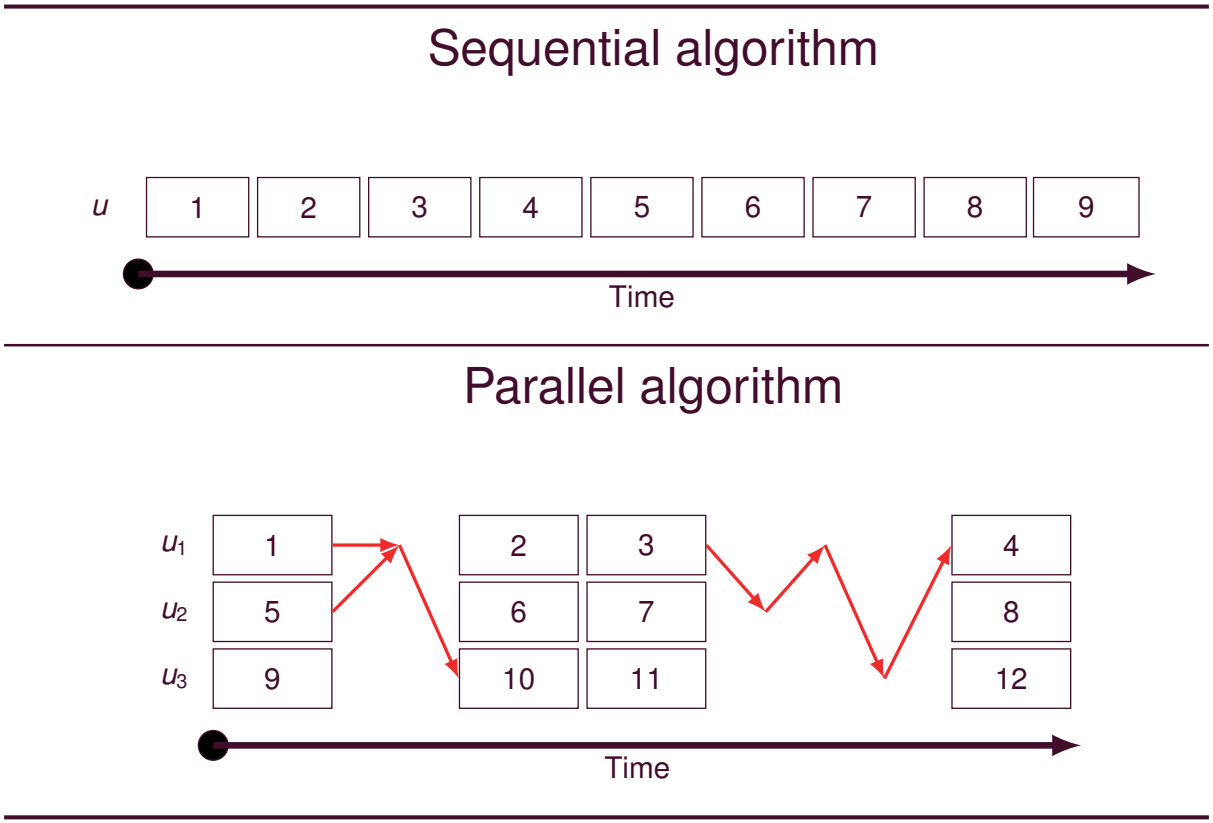


Parallel Algorithm



Dependency problems

Issues related to parallelism

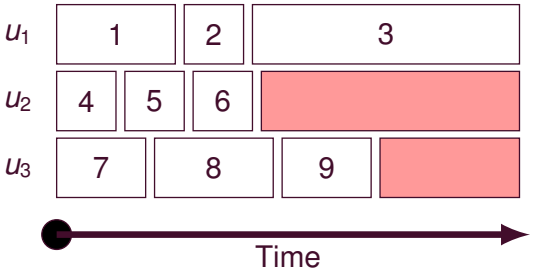


Slowdown due to communications

Issues related to parallelism

Sequential algorithm

Parallel algorithm

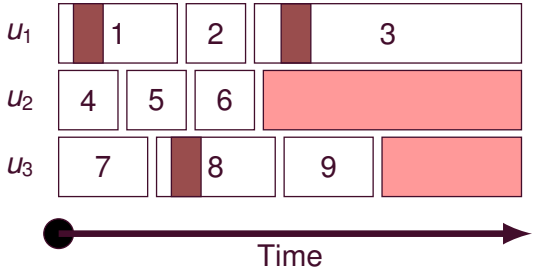


Load balancing

Issues related to parallelism

Sequential algorithm

Parallel algorithm

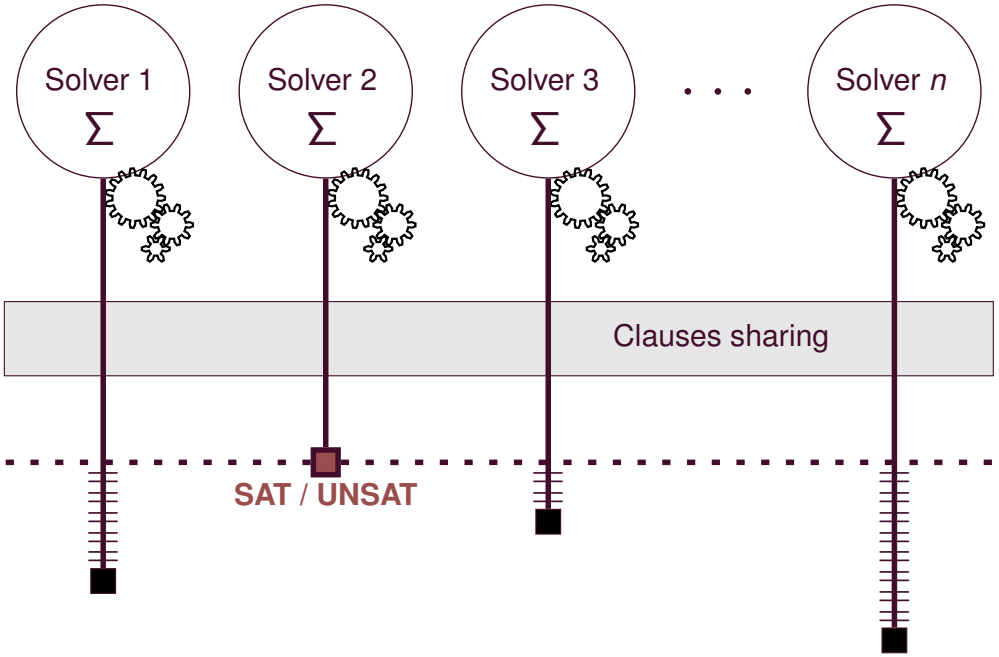


Redundant work

Solving SAT in parallel

- 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

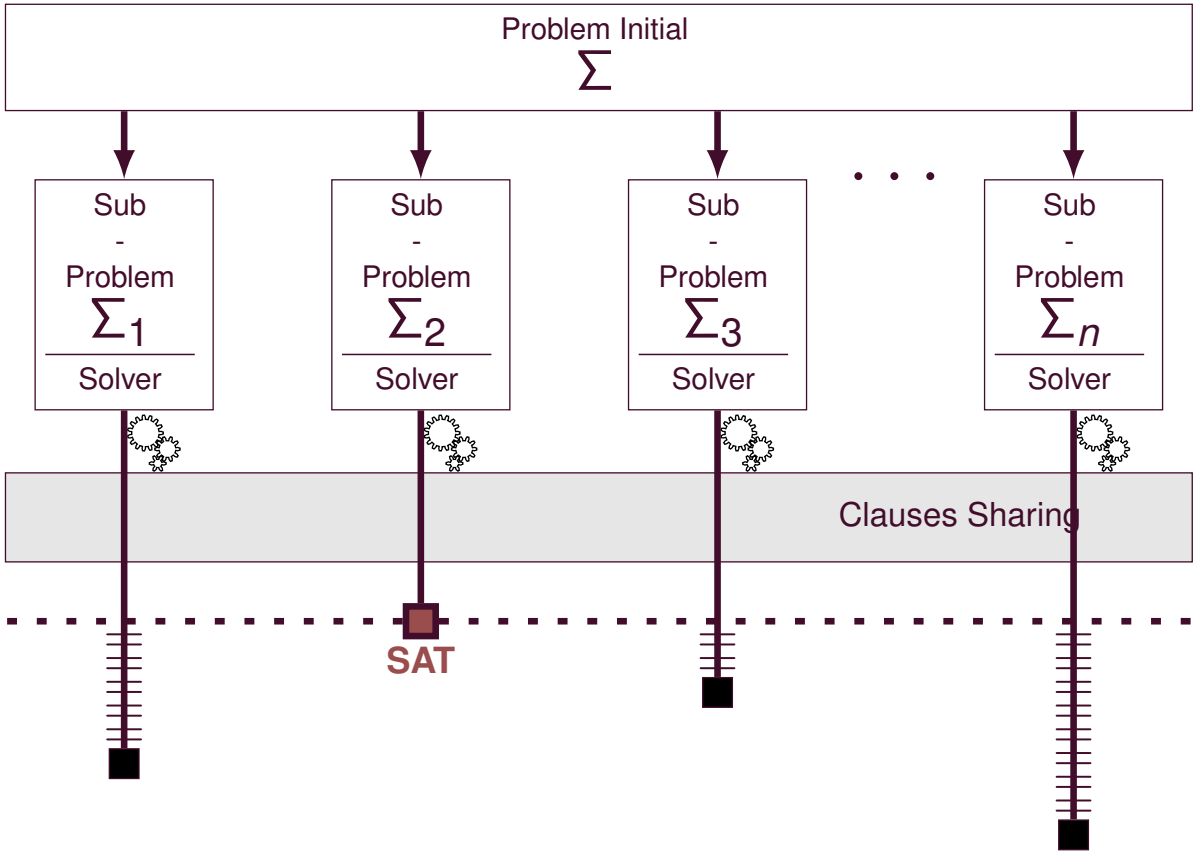


- Each solver works on the same (original) formula
- The first that successes is the winner
- Allow to easily share clauses.

Clauses sharing is essential !

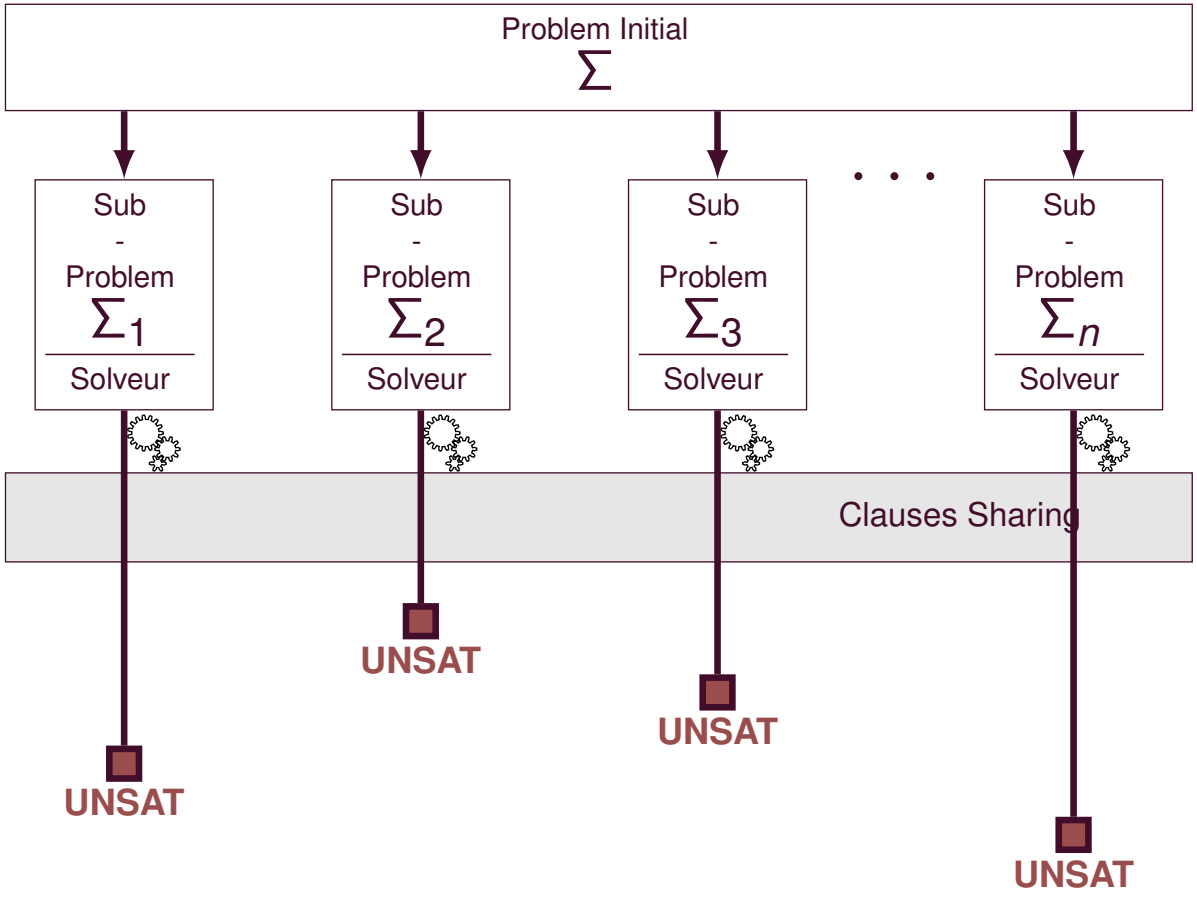
Divide and Conquer approach

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



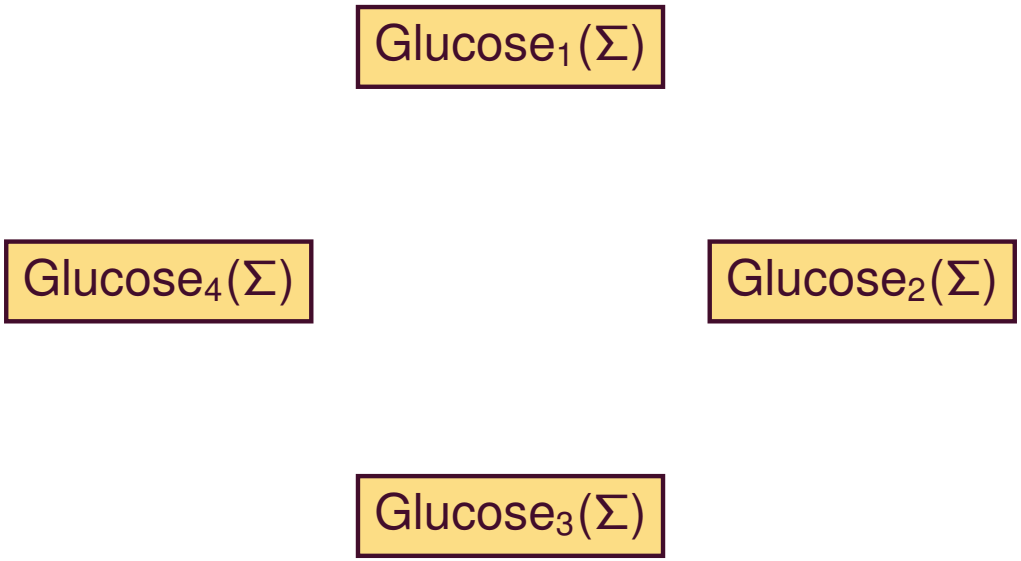
Divide and Conquer approach

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



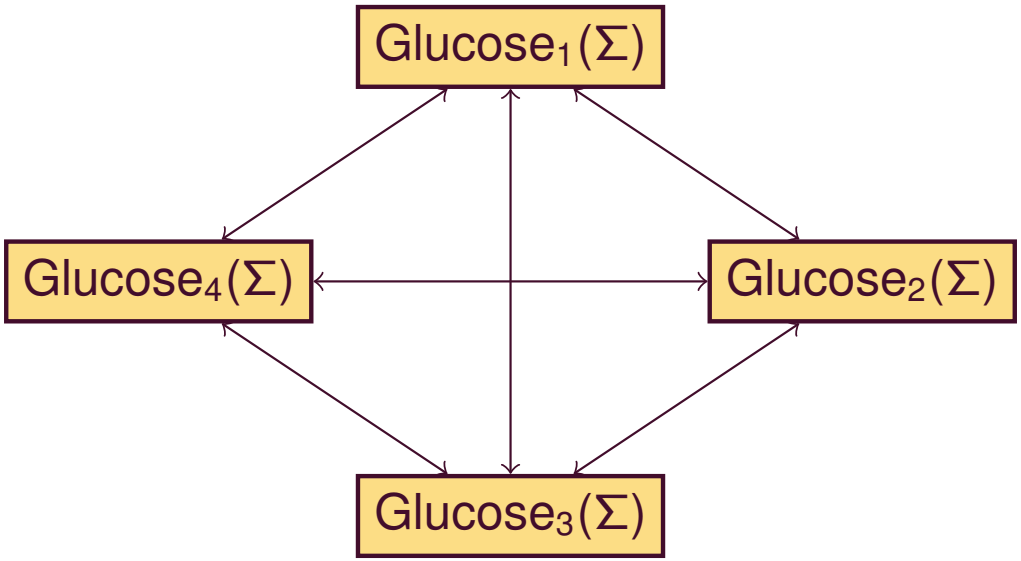
Syrup : Many glucose in parallel

- Each thread has its own glucose solver
- Each solver deals with the whole formula Σ 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 Σ 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



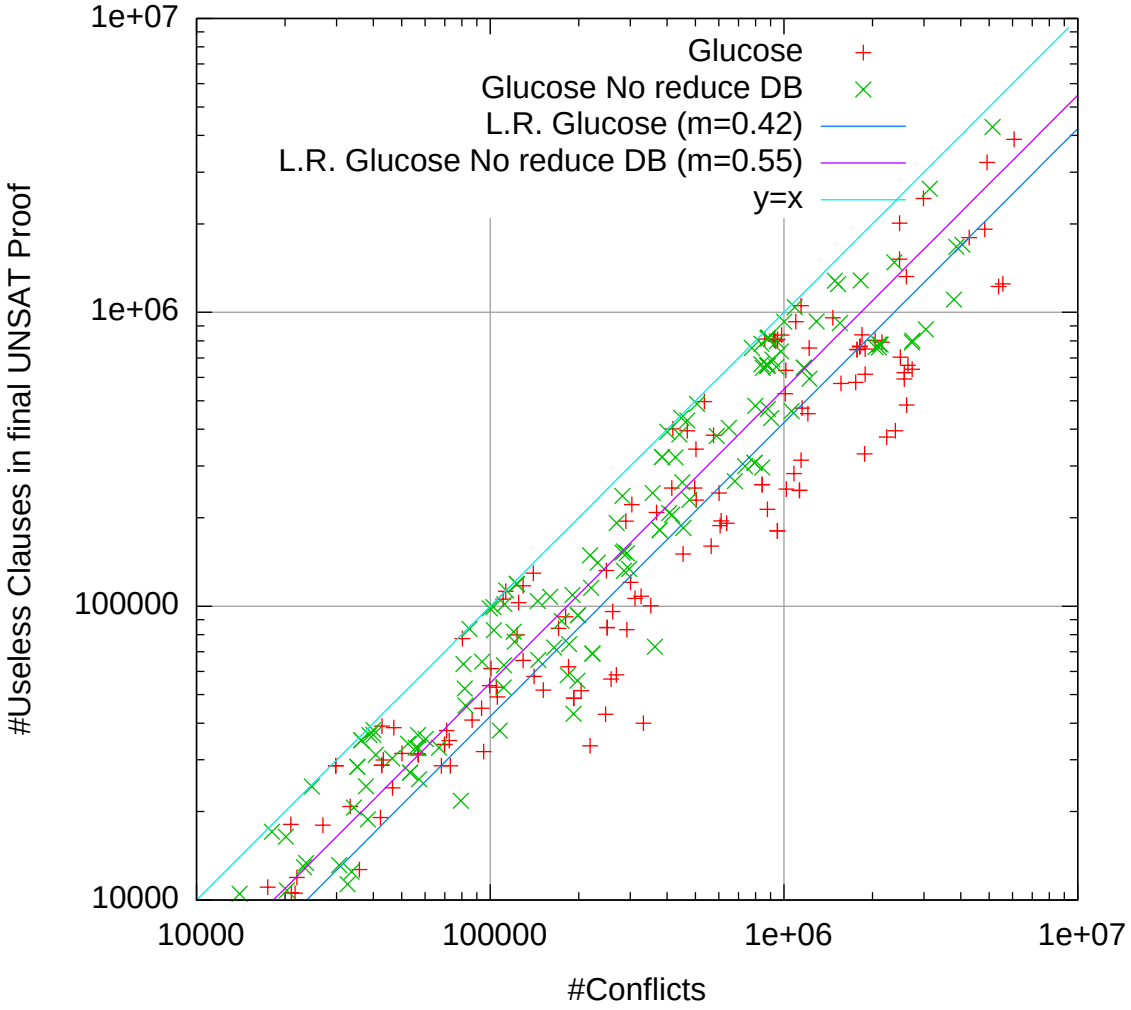
Many Cores, many more problems

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

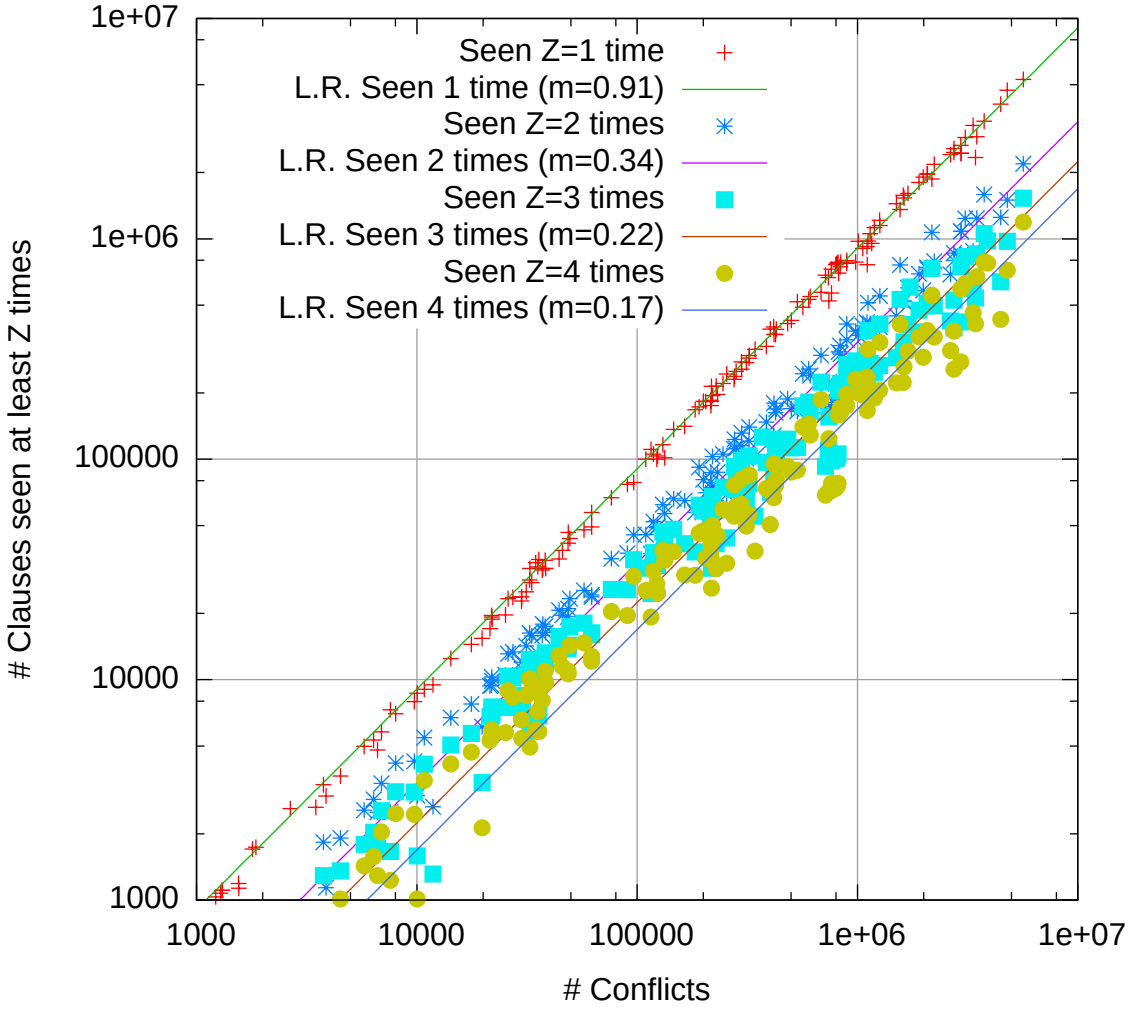
Parallel solvers limit the number of shared clauses (criterla : size, lbd...)

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

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

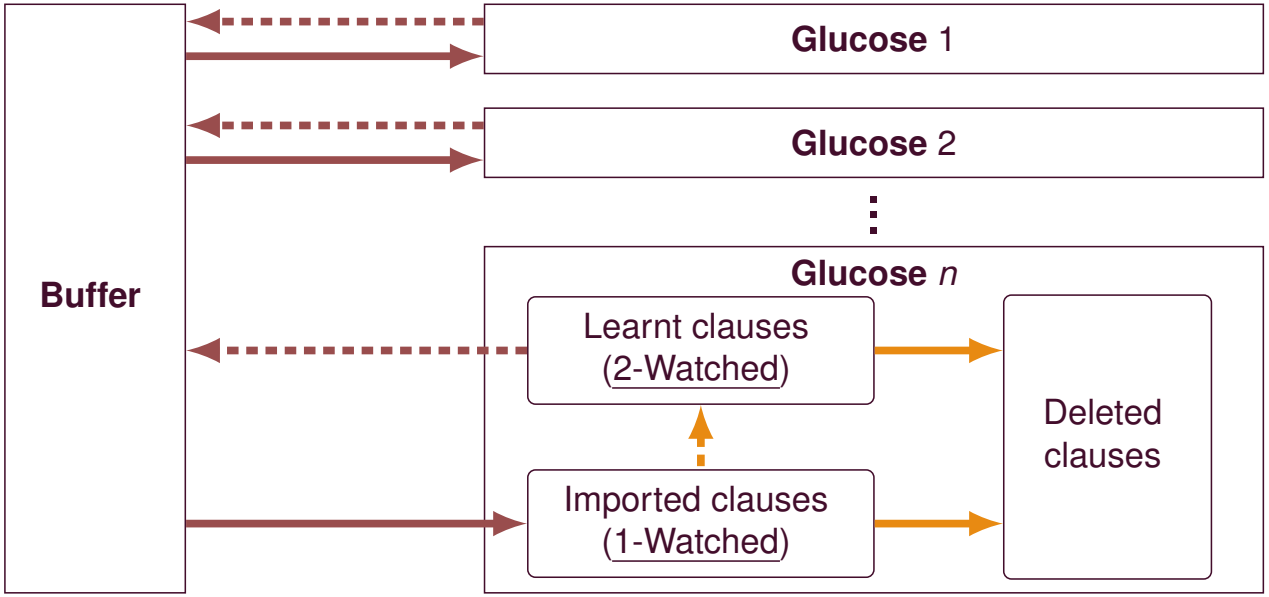
- can destroy the current search effort
- clauses can be redundant
- many clauses to manage (performance impact)

How to be sure an imported clause is interesting before considering it ?

Probation

Idea : put it in probation

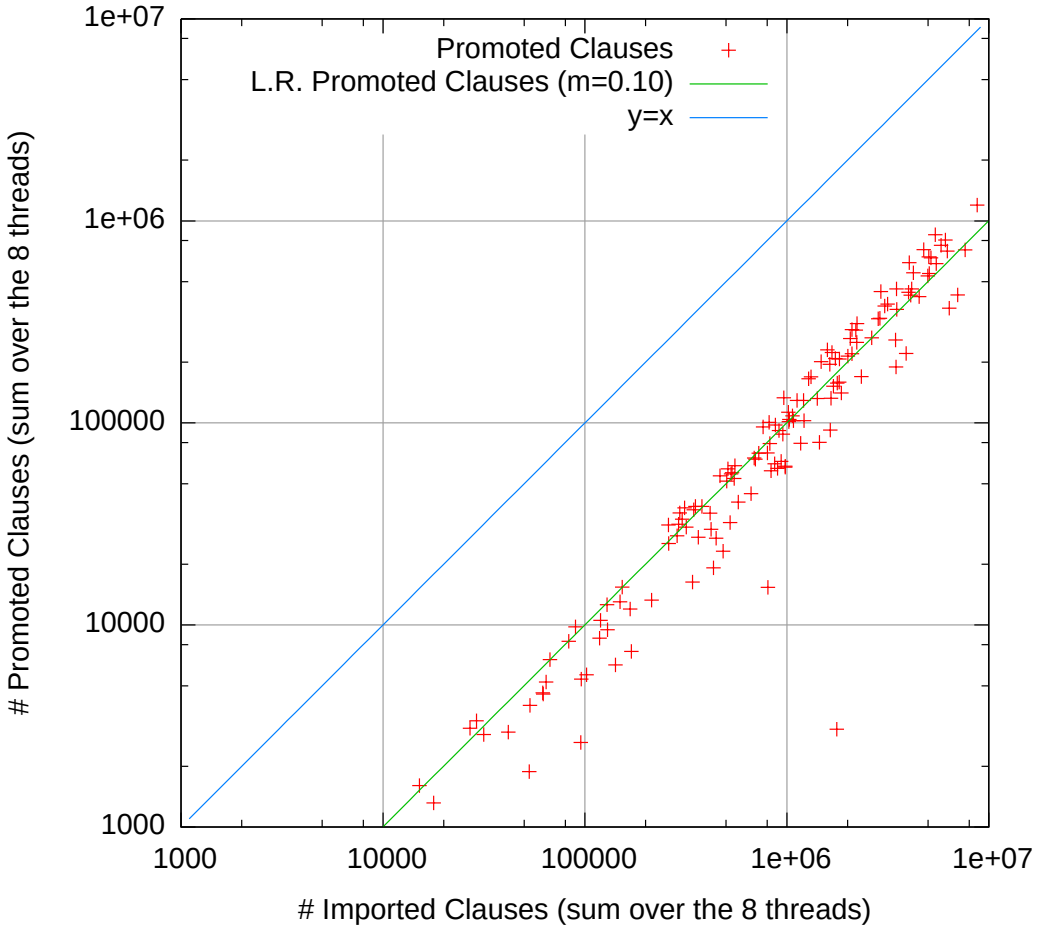
- Imported clauses are put in a 1-Watched scheme
- Will be promoted to a 2-Watched scheme only if found empty



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 ?

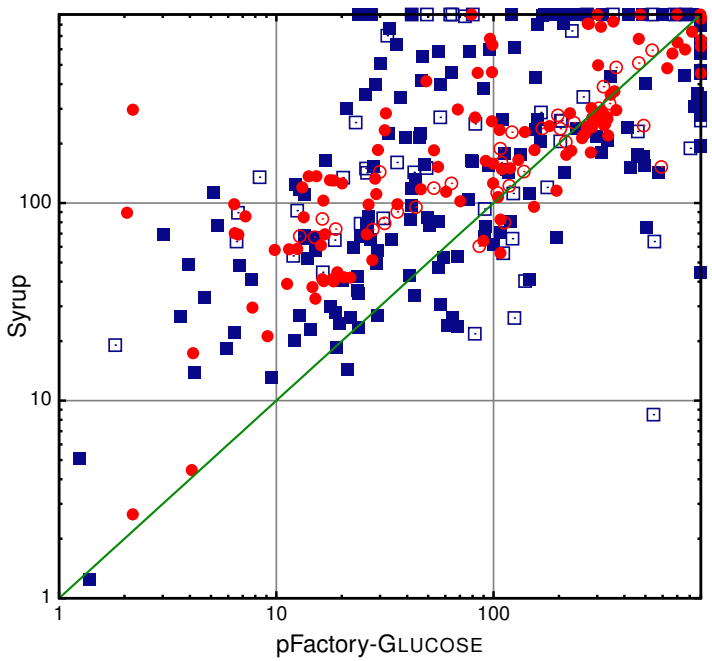


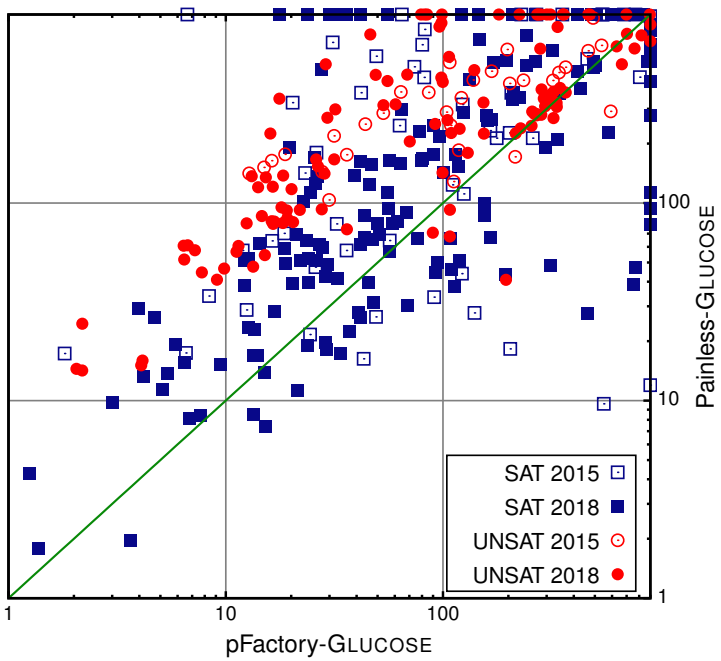
- x-axis : Number of imported clauses (sum over 8 threads)
- y-axis : Number of promoted clauses (sum over 8 threads)

pFactory

- 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	2015 (100)			2018 (300)		
Solver	**SAT**	**UNSAT**	**Total**	**SAT**	**UNSAT**	**Total**
GLUCOSE	9	6	15	67	66	133
Painless-GLUCOSE	42	32	74	141	99	240
Syrup	43	32	75	141	114	255
pFactory-GLUCOSE	49	32	81	149	114	263