Lookahead in Partitioning SMT

Antti E. J. Hyvärinen[1], Matteo Marescotti[2], and Natasha Sharygina[1]

1 Università della Svizzera italiana, Switzerland
2 Facebook, UK

Parallel SMT Solving

Increasingly relevant as a result of widely available parallel execution environments
(mainly Amazon EC2)

SMT-COMP 2021 -- the first time SMT solvers competed in a truly parallel setting
(64/1600 virtual CPU cores)

-> Parallel implementations of some of the best solvers (CVC5, OpenSMT, Par4, STP, Vampire)
-> Often the parallel solvers performed better than the best sequential solvers

See https://smt-comp.hyvarinen.ch/2021/parallel-and-cloud-tracks.html

The question on how to best achieve scalability is very relevant

Portfolio vs Divide-and-Conquer

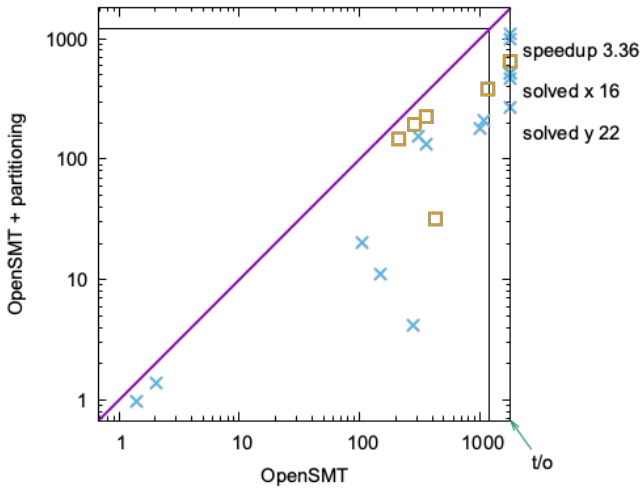
Algorithm Portfolio:

- Run a set of solvers in parallel on the same instance
- Terminate when one solver determines satisfiability
- Possibly share information between different solvers

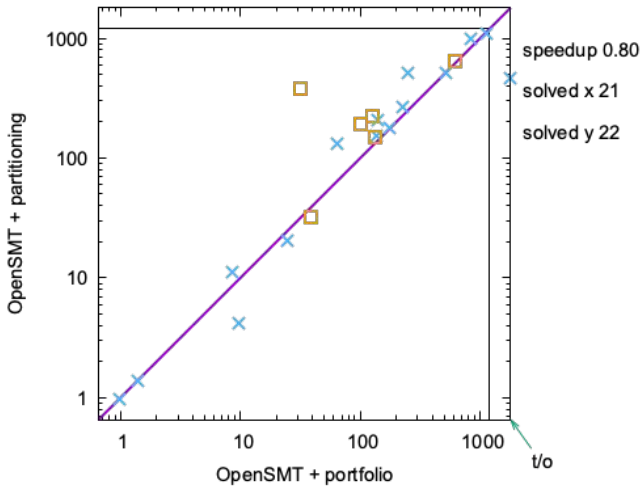
Divide-and-Conquer:

- Partition the search space of an instance to mutually disjoint parts
- Terminate when one solver has shown satisfiability or all partitions have been shown not to contain solutions
- Possibly share information, but make sure that the information is valid in the target partition

× -- satisfiable, □ -- unsatisfiable



Divide-and-conquer is much faster than the sequential version



There is no clear winner between divide-and-conquer and portfolio

SMT-COMP 2021 QF linear arithmetic divisions (QF_LinearIntArith, QF_LinearRealArith)

Understanding divide-and-conquer in SMT

Divide-and-conquer is complicated to implement. There are many potential reasons to explain a relative slowdown in SMT

- Communication delays
- Construction of partitions is costly
- Memory congestion in shared memory systems
- Suboptimal partitions
- Statistics-based slowdown in unsatisfiable instances [HJN11]
- Proof-theoretical reasons [JJ:CP07]
- Incomplete understanding of the real search space in SMT

[HJN11] Hyvärinen, Junttila, Niemelä: Partitioning Search Spaces of a Randomized Search. *Fundam. Informaticae* **107**(2-3): 289-311 (2011)

[JJ:CP07] Järvisalo, Junttila: Limitations of Restricted Branching in Clause Learning. *CP 2007*: 348-363

Understanding divide-and-conquer in SMT

Divide-and-conquer is complicated to implement. There are many potential reasons to explain a relative slowdown in SMT

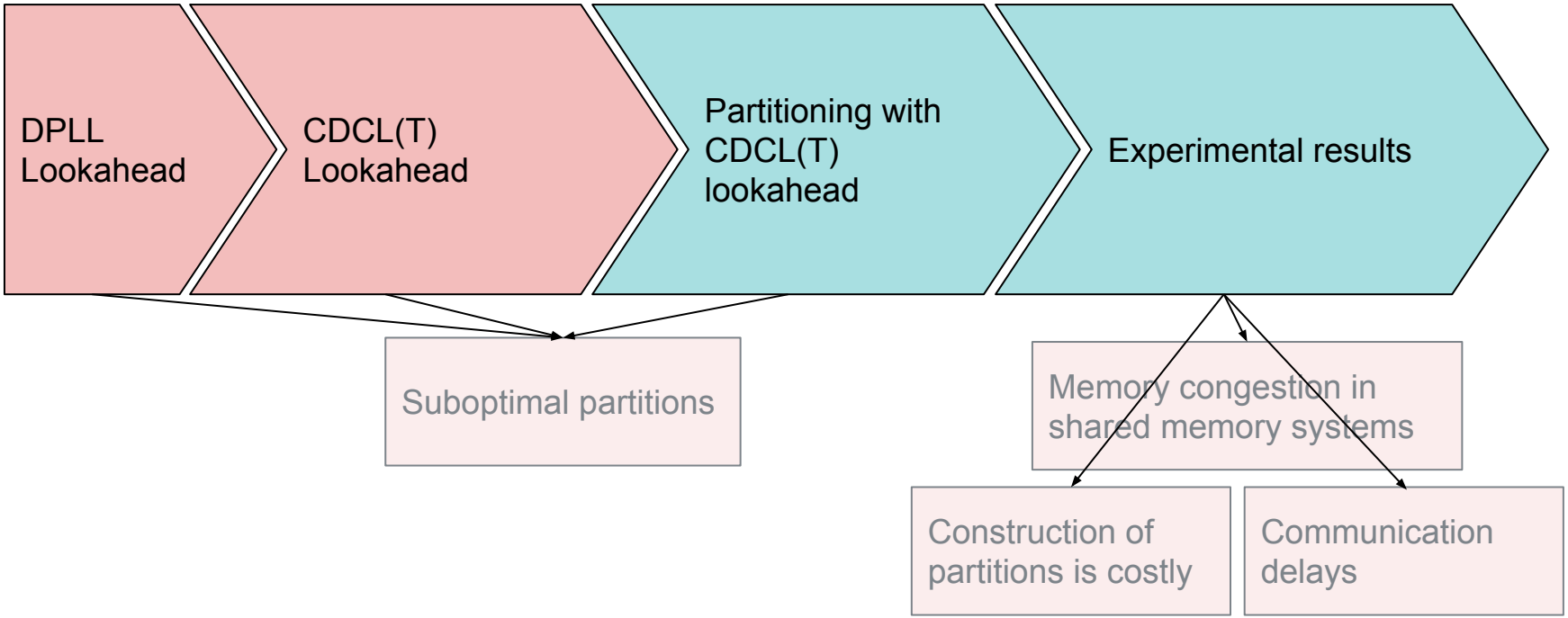
Communication delays	Construction of partitions is costly	Memory congestion in shared memory systems	Suboptimal partitions
Statistics-based slowdown in unsatisfiable instances [HJN11]	Proof-theoretical reasons [JJ:CP07]	Incomplete understanding of the real search space in SMT	

In this talk we attempt to exclude some of these reasons and still show a slowdown

[HJN11] Hyvärinen, Junttila, Niemelä: Partitioning Search Spaces of a Randomized Search. *Fundam. Informaticae* **107**(2-3): 289-311 (2011)

[JJ:CP07] Järvisalo, Junttila: Limitations of Restricted Branching in Clause Learning. *CP 2007*: 348-363

Presentation Outline



DPLL algorithm

SAT instance in Conjunctive Normal Form

$$\neg x \vee y$$

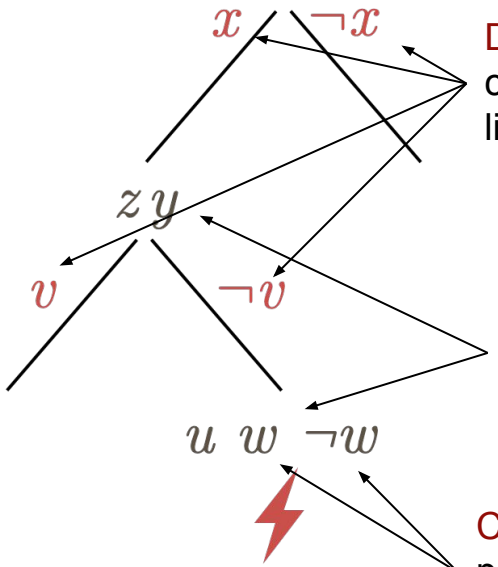
$$\neg x \vee z$$

$$\neg u \vee \neg z \vee w$$

$$v \vee \neg y \vee u$$

$$v \vee \neg u \vee \neg w$$

DPLL search tree



Decisions are heuristically chosen among unassigned literals

Propagation occurs when all but one literal in clause is assigned to false

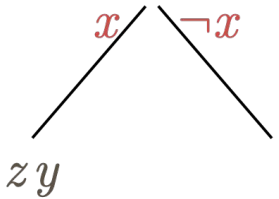
Conflict occurs when propagation forces a literal to be both true and false

Lookahead in DPLL

Before committing to a decision, see how many literals would propagate as a result
Commit to the decision that results in most propagations

$\mathrm{LA}(x) = |\{z, y\}| = 2$

$\mathrm{LA}(v) = 0$



$\mathrm{LA}(\neg x) = |\emptyset| = 0$

$\mathrm{LA}(\neg v) =$ 

$\neg x \vee y$

$\neg x \vee z$

$\neg u \vee \neg z \vee w$

$v \vee \neg y \vee u$

$v \vee \neg u \vee \neg w$

The DPLL algorithm changes a little because of the early conflict detection

Lifting Lookahead to CDCL(T)

Most SMT solvers are driven by a conflict-driven clause-learning SAT solver

The algorithm differs significantly from a DPLL style search

The algorithm learns clauses using resolution

SMT solvers in addition learn theory lemmas

There is no search tree in the sense of a DPLL search

CDCL: Search

Search is ordered as a stack of decisions and propagated variables

$\neg w$
w
u
$\neg v$
z
y
x

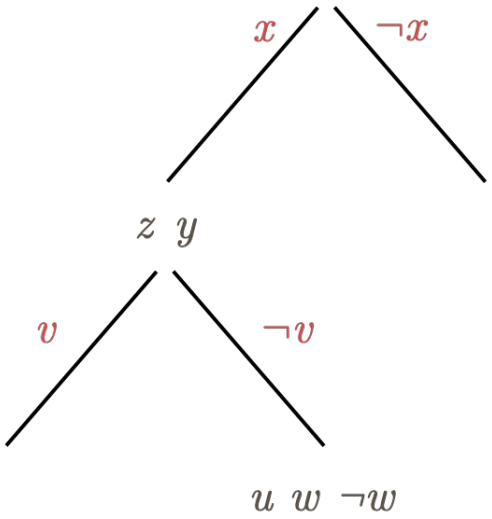
$$\neg x \vee y$$

$$\neg x \vee z$$

$$\neg u \vee \neg z \vee w$$

$$v \vee \neg y \vee u$$

$$v \vee \neg u \vee \neg w$$



CDCL: Clause Learning

If propagation results in conflict, solver constructs a conflict graph

$\neg w$
w
u
$\neg v$
z
y
x

$\neg x \vee y$

$\neg x \vee z$

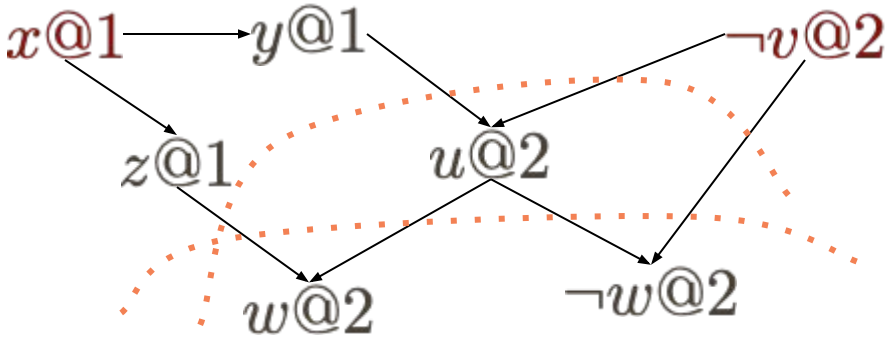
$\neg u \vee \neg z \vee w$

$v \vee \neg y \vee u$

$v \vee \neg u \vee \neg w$

$\neg z \vee \neg y \vee v$

A propagated literal has incoming edges from the propagating clause. Each literal is labelled with decision level based on number of decision literals in the stack.



A conflict graph guides resolution that derives a learned clause with a single literal at the highest decision level

CDCL: Clause Learning

If propagation results in conflict, solver constructs a conflict graph

$\neg w$
w
u
$\neg v$
z
y
x

$$\neg x \vee y$$

$$\neg x \vee z$$

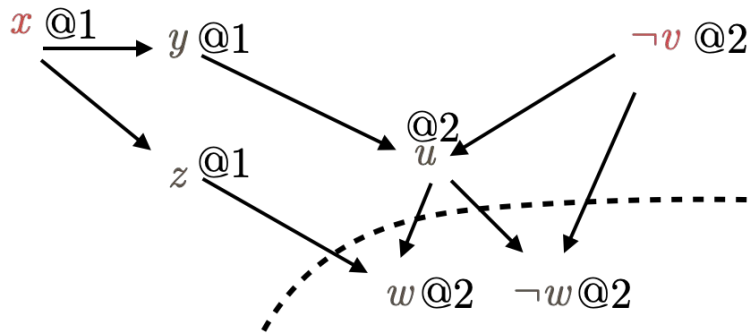
$$\neg u \vee \neg z \vee w$$

$$v \vee \neg y \vee u$$

$$v \vee \neg u \vee \neg w$$

$$\neg z \vee \neg u \vee v$$

Each literal is labelled with decision level based on number of decision literals in the stack



A conflict graph guides resolution and derives a learned clause with a single literal at the highest decision level

CDCL: Backtracking

The learned clause is added to the solver and used for backtracking

$\neg w$
w
u
~~v~~
z
y
x

$\neg x \vee y$

$\neg x \vee z$

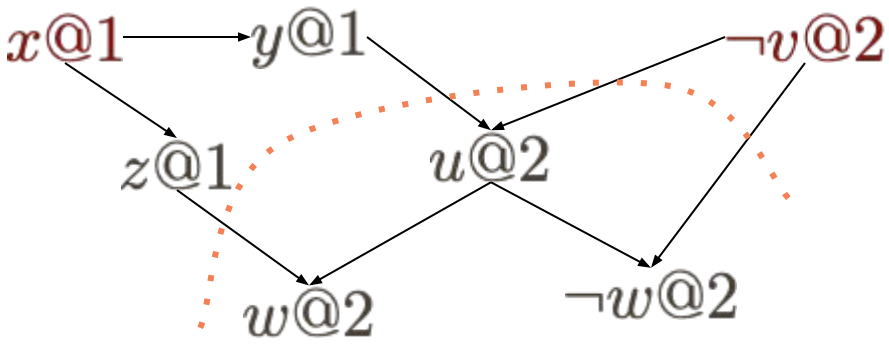
$\neg u \vee \neg z \vee w$

$v \vee \neg y \vee u$

$v \vee \neg u \vee \neg w$

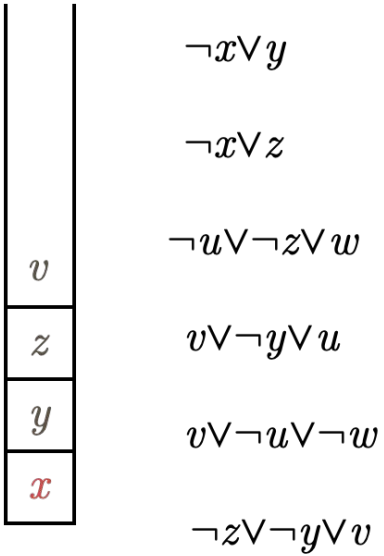
$\neg z \vee \neg y \vee v$

By construction the learned clause propagates a new literal

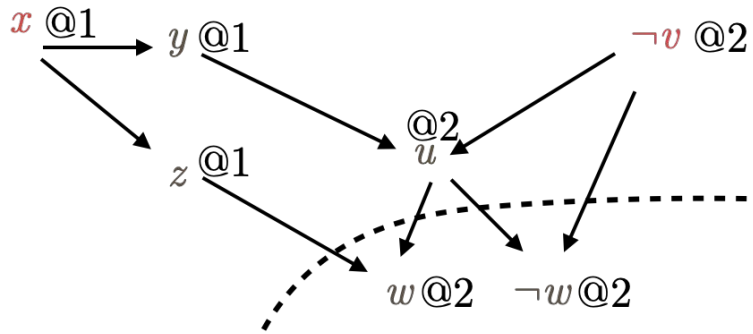


Backtracking

The learned clause is added to the solver and used for backtracking



By construction the learned clause propagates a new literal



A conflict graph guides resolution and derives a learned clause with a single literal at the highest decision level

Lookahead in CDCL-Based Solving

The idea is to build a DPLL search tree style object

We use a CDCL(T) solver to compute the lookahead values

The CDCL(T) state of the solver evolves

- Learned clauses and theory lemmas affect the lookahead scores
- Use the evolving state to prune partitions that are newly discovered unsatisfiable

CDLC(T) lookahead algorithm

Build the DPLL tree depth first

- Place the CDCL(T) solver to a given node
- Compute the lookahead score
  - Backtrack and learn clauses in case of conflicts
  - Choose the best literal according to LA

The CDCL(T) solver may encounter conflicts and needs to backtrack to resolve them, rebuilding parts of the tree

The solver state might change significantly in the lookahead phase

CDCL(T) lookahead in splitting

Construct a DPLL search tree depth-first

- Limit the depth of the tree to n where 2^n is the desired number of partitions

Use the CDCL(T) solver to guide the construction of the DPLL tree as in the CDCL(T) lookahead algorithm

This design results naturally in a splitting algorithm that either

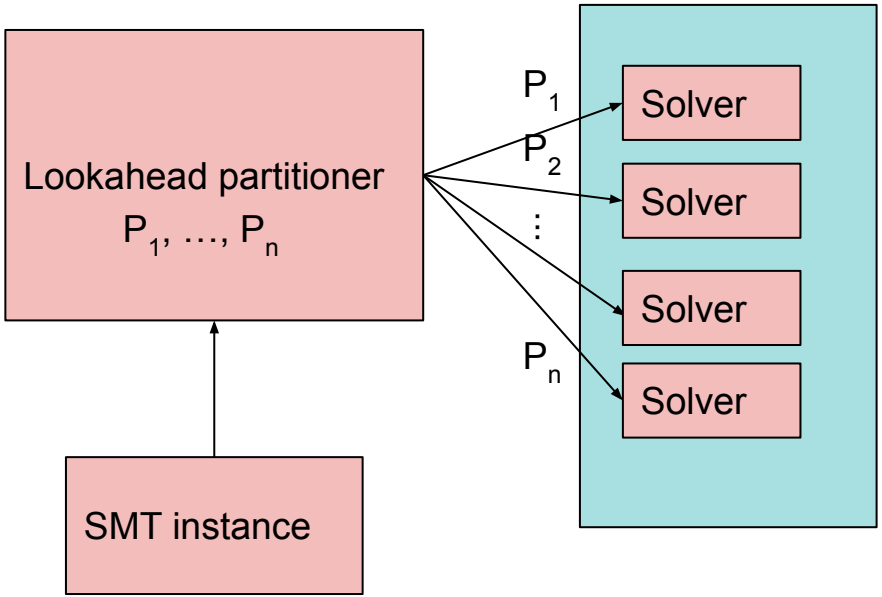
1. Produces exactly 2^n splits, or
2. Shows the instance unsatisfiable, or
3. Shows the instance satisfiable.

Experiments

Instances taken from SMT-LIB logics QF_LRA and QF_UF

- The logics exercise three algorithms that are widely used in practical SMT solving
  - CDCL algorithm (both QF_LRA and QF_UF)
  - Simplex (QF_LRA)
  - Congruence closure (QF_UF)
- We selected instances that are *hard*: OpenSMT needs 100 seconds to solve them
  - Recent improvements on OpenSMT result in some of these instances not being that hard, but they still require more than 10 seconds to solve.

Experimental setup



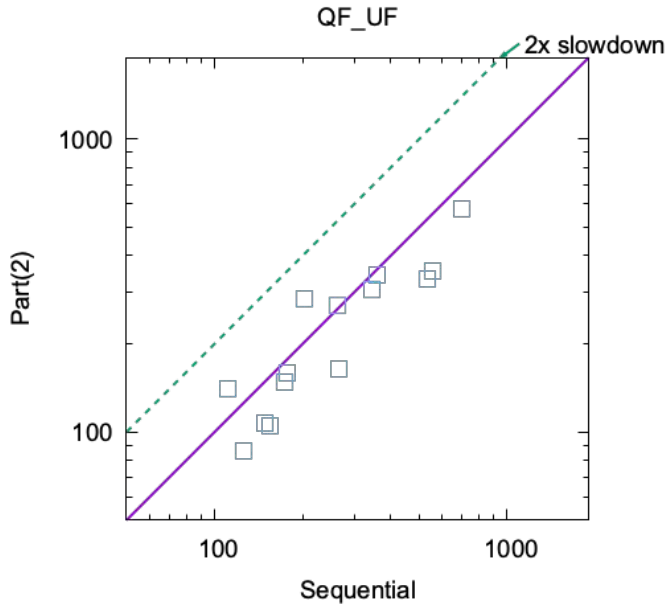
The run time of Part(n) is

- If instance is satisfiabile: the shortest run time of any Solver finding a solution
- If the instance is unsatisfiable: the longest run time of any Solver

Run time of Part(n) does not include communication delays or time to partition

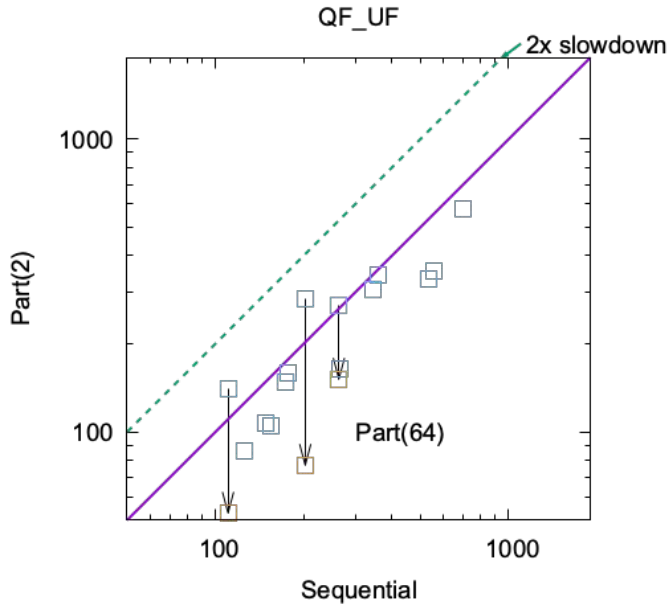
The partitions are inserted as-is to the SMT solver (no incrementality)

Partitioning in two parts



QF_UF: the approach works generally well, and slowdowns seem to be random noise.

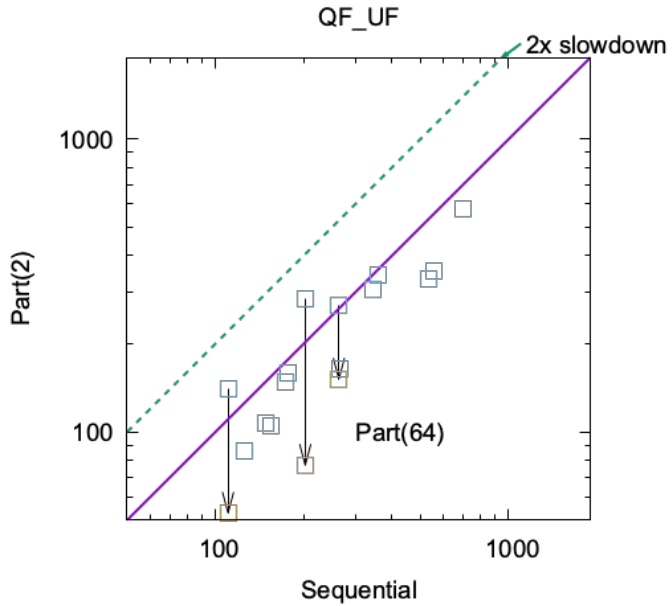
Partitioning in two parts

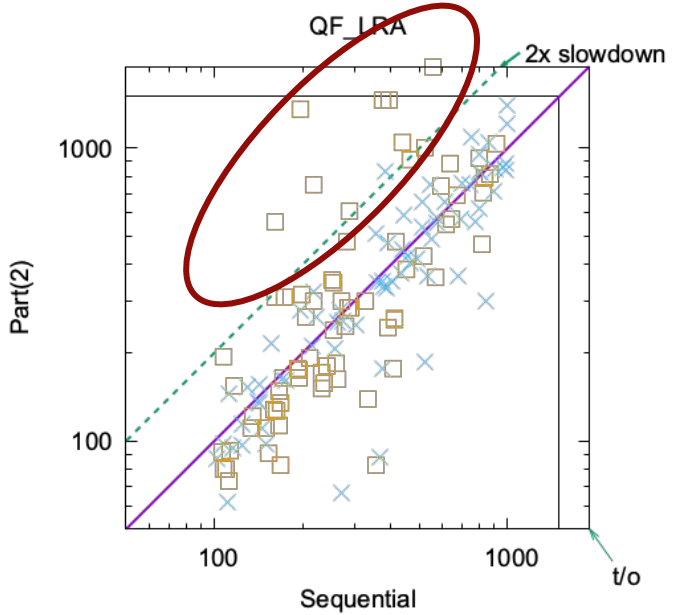


QF_UF: the approach works generally well, and slowdowns seem to be random noise.

The boxes pointed to by arrows are run times for partitioning into 64.

Partitioning in two parts



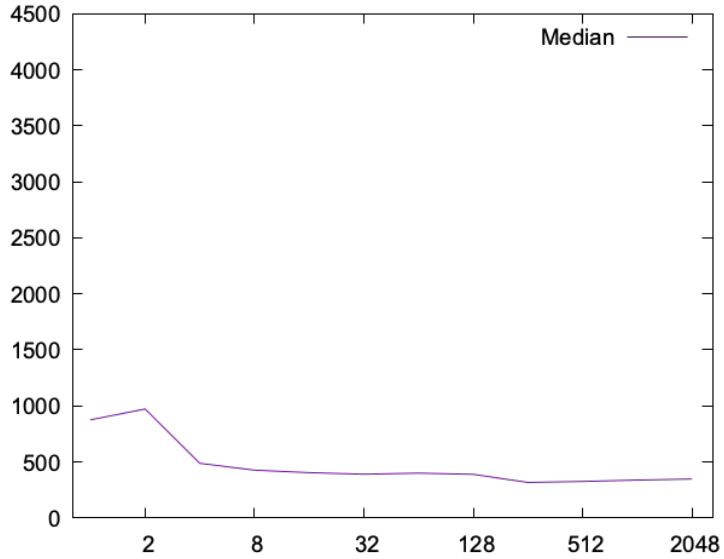


QF_UF: the approach works generally well, and slowdowns seem to be random noise.

The boxes pointed to by arrows are run times for partitioning into 64.

QF_LRA: the approach works generally well. There are, however, non-trivial slowdowns

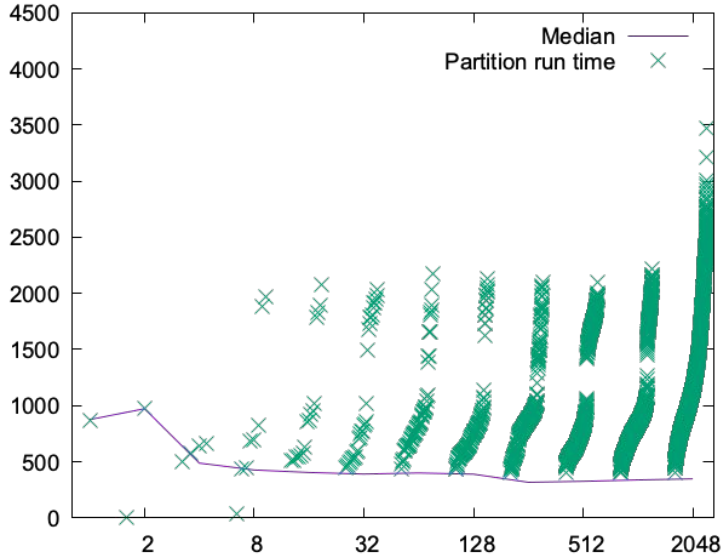
Slowdown anomaly: a satisfiable instance (QF_LRA)



Median solving time as a function of number of partitions

There is an increase before a speed-up at 4 partitions, that then stabilises

Slowdown anomaly: a satisfiable instance (QF_LRA)



Median solving time as a function of number of partitions

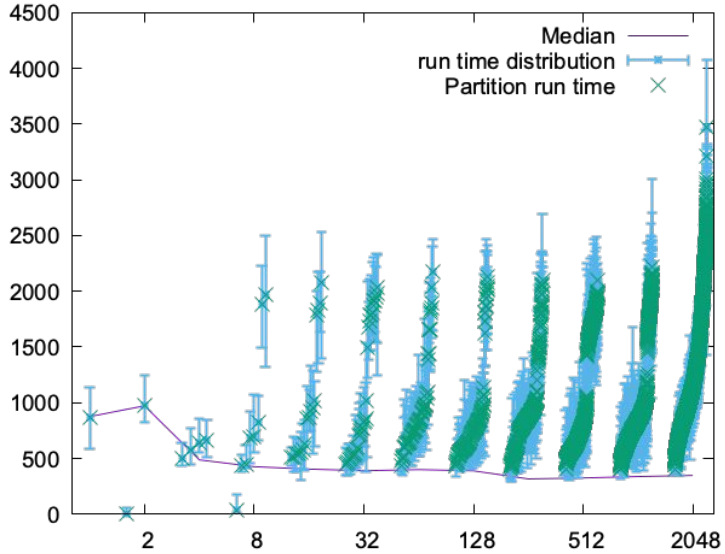
There is an increase before a speed-up at 4 partitions, that then stabilises

Run time for each partition. Partitions are grouped visually and sorted

When more partitions are created, the partitions split into two groups:

- Satisfiable (easier)
- Unsatisfiable (harder)

Slowdown anomalies: a satisfiable instance (QF_LRA)



Median solving time as a function of number of partitions

There is an increase before a speed-up at 4 partitions, that then stabilises

Run time for each partition. Partitions are grouped visually and sorted

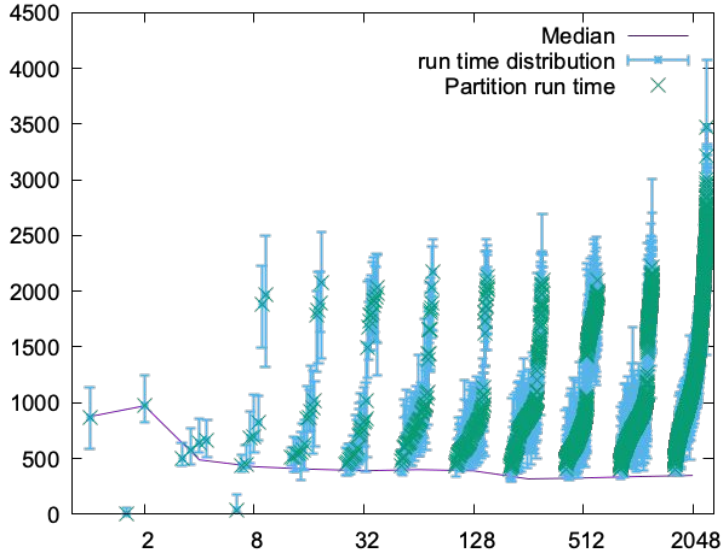
When more partitions are created, the partitions split into two groups:

- Satisfiable (easier)
- Unsatisfiable (harder)

Each partition was run several time to obtain a distribution. Overlapping partitions suggest good balance and result in speedup

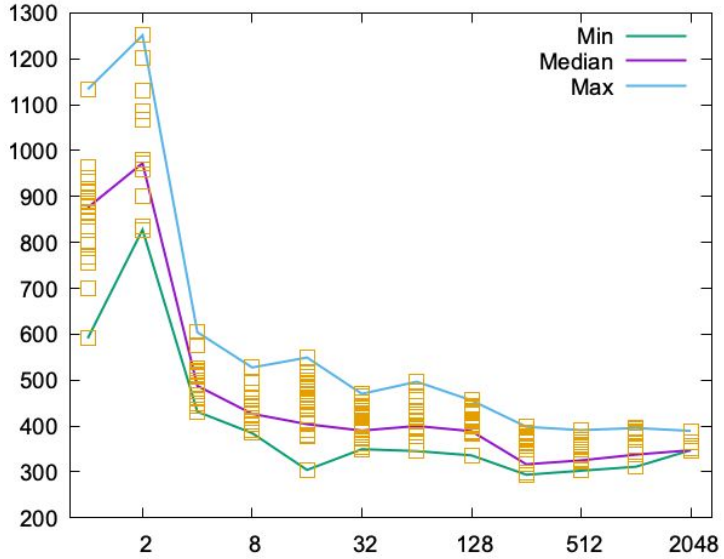
The increase in difficulty for Part(2) is quite clear

Slowdown anomalies: a satisfiable instance (QF_LRA)



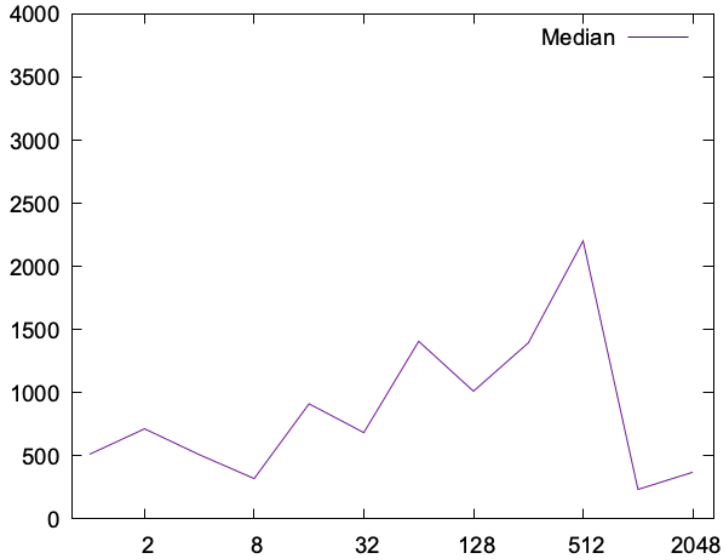
Median solving time as a function of number of partitions

There is an increase before a speed-up at 4 partitions, that then stabilises



Instance solving time statistics as a function of number of partitions: Part(2) is harder than the original instance and the spread diminishes with increasing number of partitions

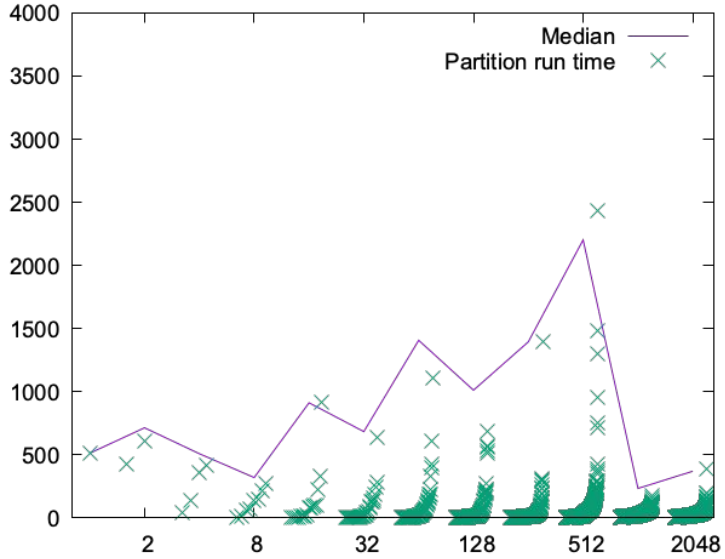
Slowdown anomalies: an unsatisfiable instance



Median solving time as a function of number of partitions

Runtime increase until a speed-up at 1024 partitions

Slowdown anomalies: an unsatisfiable instance

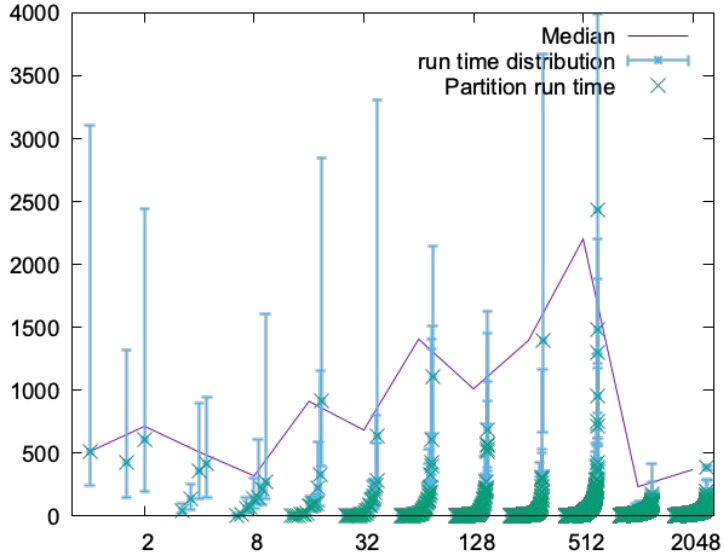


Median solving time as a function of number of partitions

Runtime increase until a speed-up at 1024 partitions

Often a handful of hard partitions, and the rest are relatively easy

Slowdown anomalies: an unsatisfiable instance



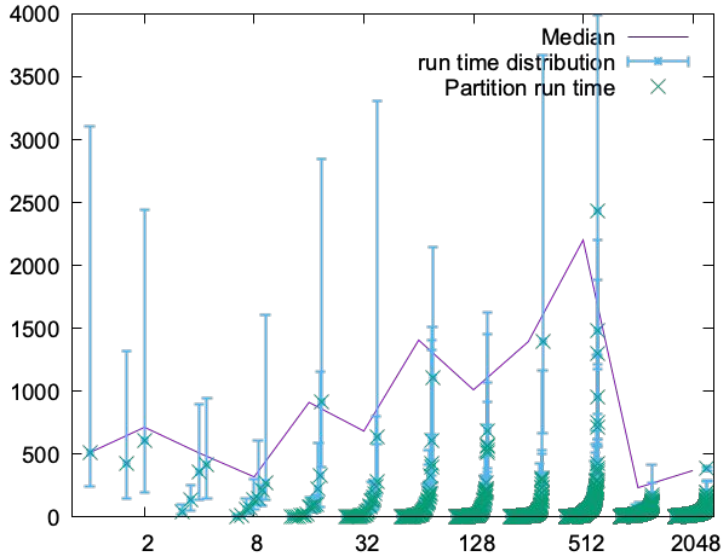
Median solving time as a function of number of partitions

Runtime increase until a speed-up at 1024 partitions

Often a handful of hard partitions, and the rest are relatively easy

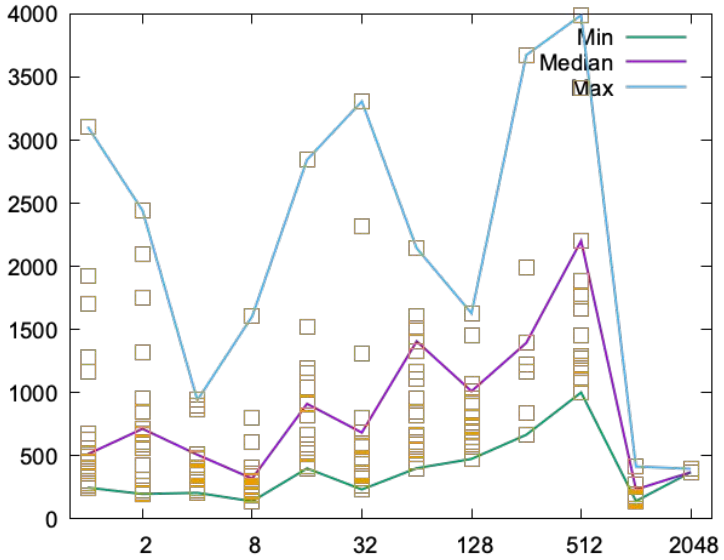
Increase and especially sudden drop in difficulty are visible in the distributions

Slowdown anomalies: an unsatisfiable instance



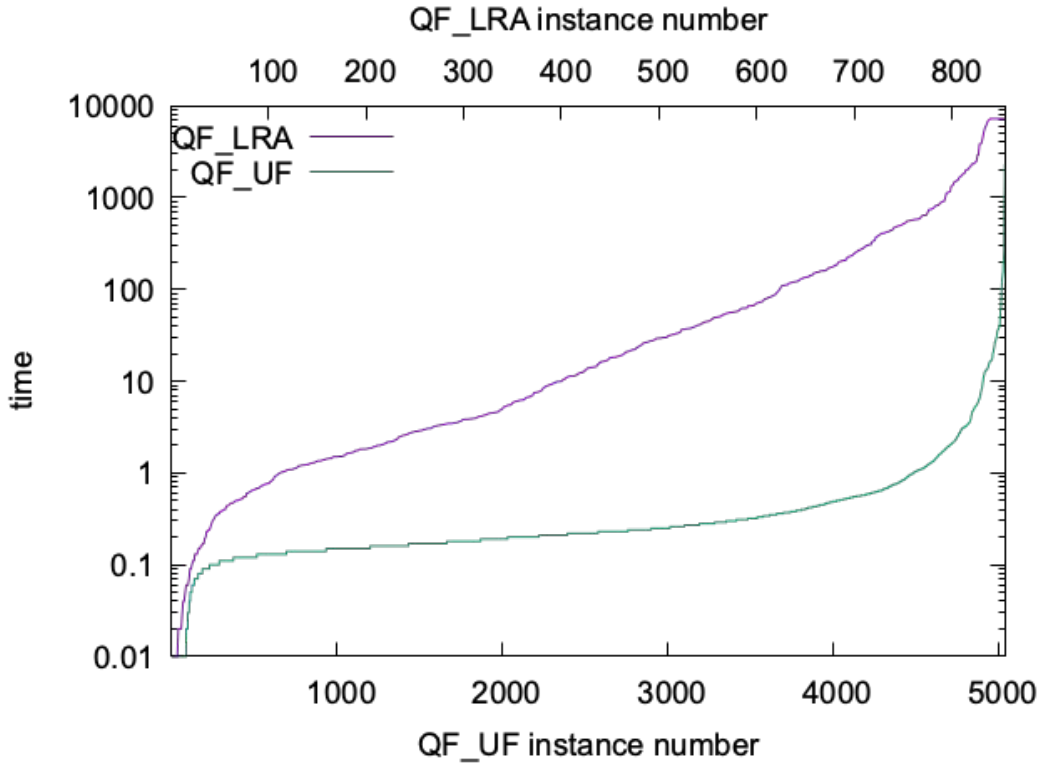
Median solving time as a function of number of partitions

Runtime increase until a speed-up at 1024 partitions



Runtime increase most visible in Min and least visible in Max

Computing the partitions with lookahead

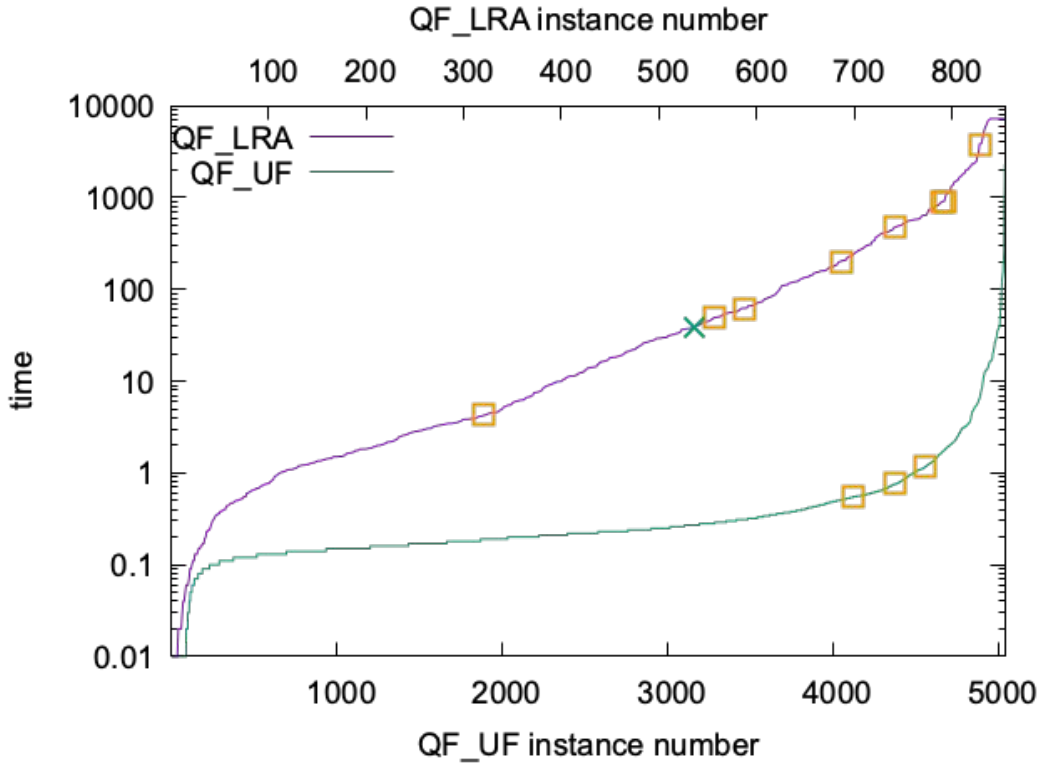


16 partitions

Producing instances with lookahead can be costly

The implementation is not particularly optimised.

Computing the partitions with lookahead



16 partitions

Producing instances with lookahead can be costly

The implementation is not particularly optimised.

Marked instances are the anomalous ones (✕ - sat, ☐ - unsat)

Conclusions

Using divide-and-conquer algorithms in parallel SMT solving is in general a good technique to obtain speedup

Even when using the good lookahead heuristic there may be surprising slowdowns especially with the Simplex-based QF_LRA algorithms

Partitioning to many parts not only makes solving often faster, but decreases the randomness in the run times

Lookahead-based partitioning in CDCL(T) is relatively expensive, but I’m optimistic that approximative algorithms and optimizations can make this approach practical.