Chapter 3
Parallel Satisfiability Modulo Theories

Antti E. J. Hyvärinen and Christoph M. Wintersteiger

Abstract Satisfiability Modulo Theories (SMT) is an extension of the propositional satisfiability problem (SAT) to other, well-chosen (first-order) theories such as integers, reals, and bit-vectors. This approach currently enjoys much popularity, especially in the field of software verification, where SMT solvers have become the de facto standard tool for the discharge of verification conditions. The development of *parallel* SMT solvers is still in its infancy, but the first general paradigms have been established. This chapter provides an overview of the recent advances in this area, specifically algorithm portfolio, search-space partitioning, and problem decomposition techniques, and how they relate to each other in theory and practice.

3.1 Introduction

Satisfiability Modulo Theories (SMT) [5] is an initiative in the area of automated deduction that aims to foster development of techniques for satisfiability checking that go beyond solving purely Boolean SAT problems. The scope of SMT is first-order logic with particular, and well-chosen, background theories of industrial or academic interest. In contrast to general first-order logic, SMT does not require background theories to be finitely axiomatizable or even decidable and still allows us to compute results that are of practical interest efficiently.

The traditional application field of SMT solvers is software verification, where a restriction to particular background theories enabled the development of specialized decision procedures that perform particularly well in determining satisfiability. Frequently the software is modeled so that satisfying solutions correspond to bugs or

Antti E. J. Hyvärinen
Università della Svizzera italiana, Lugano, Switzerland, e-mail: antti.hyvaerinen@usi.ch

Christoph M. Wintersteiger
Microsoft Research, e-mail: cwinter@microsoft.com

other undesirable program behavior. Today, SMT solvers are applied in an increasing number of applications outside of the traditional areas, including computational biology (e.g. [84, 6]), chemistry (e.g., [32]), and material science (e.g., [31]). Current research also attempts to lift the approach of providing a carefully crafted set of background theories to other domains, for instance model checking [34], optimization [77], planning [40], and probabilistic inference [76] – all modulo theories.

For some (combinations of) background theories, the computational cost of SMT solving can be very high and, in terms of computational complexity, often greatly exceeds the cost of NP-complete problems such as Boolean SAT. It is therefore of general interest to study ways of improving the problem-solving performance by addition of more parallel computing power to the SMT solver.

SMT solvers are conceptually based on SAT solvers and certain parallelization techniques can be applied in very similar ways in both approaches. Perhaps surprisingly the intricate interaction with the theory solvers results in certain techniques that work in a predictable way in SAT solvers not maintaining similar behavior in SMT solving. In particular the techniques for partitioning search space turn out to be significantly different in SMT, the technique being very efficient when applied in the presence of some background theories and even detrimental in the presence of other background theories. It seems that the background theories require a very different set of trade-offs to be considered (e.g., [83, 65]).

In this chapter we address the challenges in parallelizing SMT solvers using both multi-core and cloud-based computing environments and a variety of parallelization approaches.

3.2 General Preliminaries

We rely on the basic definitions of Boolean variables, literals, clauses, etc. from Chapter 1. Some terms used in the SMT research community and their publications may be confusing to the uninitiated reader, so we provide a brief description of the most important concepts here.

3.2.1 Theories

SMT (Satisfiability Modulo Theories) focuses on the satisfiability problem for first-order logic with particular, and well-chosen, background theories. Today, those theories are Booleans, arrays, bit-vectors, floating-point numbers, integers, real numbers, and uninterpreted symbols (equality and uninterpreted functions and sorts). From these theories, a number of fragments have been identified as academically and industrially important. Currently those fragments are as follows:

- ‘Core’ theory: Booleans and equalities,

- uninterpreted functions and sorts (UF),
- (infinite-size) arrays (including extensionality) (A),
- fixed-size bit-vectors (BV),
- floating-point numbers (FP),
- (non-)linear integer arithmetic (NIA, LIA),
- (non-)linear real arithmetic (NRA, LRA), and
- integer and real difference logic (IDL, RDL).

The abbreviations of those fragments are then combined to identify particular *logics*, where QF is used to indicate that a logic is quantifier-free. For instance, QF_AUFLIA is the quantifier-free theory of arrays, uninterpreted functions, and linear integer arithmetic. SMT solvers implement decision procedures for some or all of these logics and they are evaluated on a large set of community-contributed benchmarks available in the SMT library [5].

Note that SMT solvers do not necessarily implement specialized decision procedures for each logic. Instead, they employ *theory combination* strategies to craft decision procedures by combining more general *core theory solvers*. For instance, a general 'arithmetic' theory solver, usually implemented as a backtrackable variation of the Simplex algorithm [29], may be used for multiple logics involving some fragments of integer and real arithmetic.

3.2.2 The Underlying Conflict-Driven, Clause-Learning SAT Solver

The input to SMT solvers is usually a set of assertions (Boolean expressions, constraints), which may have a rich Boolean structure (the *skeleton*) that is not necessarily in any normal form. In practice it is often rewritten into Conjunctive Normal Form (CNF), such that existing SAT solver technology may be used to solve the skeleton. Any (partial or complete) model for the skeleton of the formula implies that some subset of theory literals needs to be solved.

Example 1. Consider the problem of solving

$$a = b \wedge (f(a) - f(b) = c) \wedge \neg(c \leq 0) \,,$$

where $a, b, c \in \mathbb{N}$ are integers and $f : \mathbb{N} \to \mathbb{N}$ is an uninterpreted function with integer range and domain. First, we introduce new Boolean variables x_1, x_2, x_3 to obtain

$$x_1 \wedge x_2 \wedge \neg x_3 \ \wedge \ x_1 \equiv (a = b) \wedge x_2 \equiv (f(a) - f(b) = c) \wedge x_3 \equiv (c \leq 0) \,,$$

where the first three conjuncts are purely Boolean. A model for this part of the formula is $x_1 = \mathit{true}, x_2 = \mathit{true}, x_3 = \mathit{false}$. This means that every theory solver will now have to solve a *conjunction of literals* instead of an arbitrary Boolean combination of literals. Here, these are

$$(a = b) \wedge (f(a) - f(b) = c) \wedge (c > 0)\,,$$

which are solved independently by a solver for the theory of (linear) integers and a solver for uninterpreted functions. The theory solvers will determine that these constraints are unsatisfiable, and will return a concise *explanation* $\neg(x_1 \wedge x_2 \wedge \neg x_3) \equiv (\neg x_1 \vee \neg x_2 \vee x_3)$ that the Boolean solver may learn and use for further guiding the search.

There are of course many more details that can make a great difference in runtime performance in practice, but here it is enough to remember that existing SAT technology such as DPLL and CDCL solvers (see Chapter 1) are immediately applicable to the Boolean skeleton of SMT formulas.

While SMT solvers learn skeleton clauses, they may also learn *lemmas* that involve theory-specific terms. These may be completely internal to a theory solver (e.g., in the form of caches), but they may also be exposed to the other theories involved. Conceptually, we can think of lemmas introducing new predicates (and thus new Boolean variables), or being new combinations of existing predicates or literals. Whether lemmas are theory-dependent or purely Boolean, heuristics similar to those in SAT solvers are employed to remove unnecessary clauses and lemmas periodically, and various simplification and minimization techniques are used to control memory usage.

3.2.3 Theory Combination

To combine theory solvers, the mechanism underlying many SMT solvers is the Nelson/Oppen theory combination framework [69]. The first step of this is to purify the formula into terms of single theories by introducing new variables for function application terms. As a result two theories only ever share uninterpreted symbols and constants. These sub-formulas are then solved independently and the theory solvers then exchange entailed equalities (syntactic and semantic).

Example 2. Suppose we need to solve the theory part of the two first conjuncts in Example 1:

$$(a = b) \wedge (f(a) - f(b) = 0)\,.$$

The aim is to employ separate theory solvers for subsets of the constraints. Before that, we purify the constraints by introduction of new interface variables. Here, this results in

$$\underbrace{(a = b) \wedge (a = e_1) \wedge (b = e_2)}_{Integers} \wedge \underbrace{(f(e_1) - f(e_2) = e_3) \wedge (e_3 = 0)}_{Uninterpreted\,functions}$$

so that e_1, e_2, and e_3 are the interfaces between theories. Basic theory combination as defined by Nelson and Oppen now exchanges *all equalities* over interface variables implied by the current constraints. Note that these equalities are

equalities between variables, not necessarily only between numerals. In this over-simplified example, because of $a = b$ we find the new equality $(e_1 = e_2)$, which the integer theory communicates to the UF theory solver. This can then derive $f(e_1) - f(e_2) \equiv f(e_1) - f(e_1)$ and thus $e_3 = 0$, satisfying all constraints.

Nelson and Oppen focused on equalities for the interface between theories, of which there may be an infinite number, and models themselves may be of infinite size, thus requiring theories to be 'stably infinite', and the theory solvers need to be able to perform case splits in non-convex theories. However, there have been extensions to deal with a larger set of theories since then, for instance by Tinelli and Zarba [80]. Variations with certain other desirable properties and better runtime performance include, for example, model-based theory combination [67], which, if possible, communicates fewer equalities, and delayed theory combination, which delays communication of some equalities to a later point in time [13, 17].

3.2.4 Interpolants

Let $v(\phi) = \{x_1, \ldots, x_n\}$ be the free variables of a first-order formula ϕ. Craig's interpolation theorem provides a way to characterize the relationship between two formulas when one implies the other.

Theorem 1 (Craig Interpolation [23]). *Let ϕ and ψ be first-order formulas. If $\phi \Rightarrow \psi$ then there exists an* Interpolant *I such that $\phi \Rightarrow I \wedge I \Rightarrow \psi$ and $v(I) \subseteq v(\phi) \cap v(\psi)$.*

Equivalently, there is an interpolant I such that $\phi \Rightarrow I \wedge I \Rightarrow \neg\psi$ whenever $\phi \wedge \psi$ is unsatisfiable, because $\phi \Rightarrow \neg\psi \equiv \neg(\phi \wedge \psi)$. Craig's theorem guarantees the existence of an interpolant, but does not provide an algorithm to compute one. Such algorithms are known for some logics, most importantly for propositional logic, and for some of them the relationship between multiple choices of interpolants are known. The most important interpolation algorithms for propositional logic can be expressed using the *labeled interpolation system* [27, 3], and includes the McMillan [66] and the Huang [48], Krajícek [61], Pudlák [73] (HKP) interpolants. It is, however, not necessary to understand the details of these algorithms in the remainder of this chapter.

It is worth noting that the Nelson/Oppen theory combination framework exchanges interface equalities, but their proof of soundness does in fact rely on the existence of Craig interpolants, which provides a more general view of theory combination, if all involved theories have interpolants and methods to compute them.

3.2.5 SMT Solvers

The software used for determining the satisfiability of formulas expressed in SMT are called SMT solvers. There are several SMT solver implementations with dif-

ferent strengths. The solvers offering most compelling support for the SMT language include CVC4 [4], Z3 [68], MathSAT [19], Yices [28]. Other solvers specializing in particular theories or problems include OpenSMT [53], MapleSTP, STP, Boolector [16], ABC [15], AProVE [35], iSAT [59], Minkeyrink, ProB [62], Q3B [58], raSAT [81], SMT-RAT [22], SMTInterpol [18], toysmt, Vampire [60], and veriT [12]. Many of these solvers take part in the annual SMT competition (for the latest edition see [21]).

3.3 Portfolios of SMT Solvers

Many branch-and-bound backtracking algorithms exhibit high variance in runtime when small alternations are introduced into the search process [79, 63, 72, 39]. Intuitively an 'unlucky' choice in the heuristic search can lead to a part of the branch-and-bound tree which is particularly difficult to solve. Sometimes such areas could have been avoided, had the search been performed in a slightly different order.

The SMT search can be seen as an instance of a branch-and-bound backtracking algorithm, and experiments confirm that the runtime of an SMT solver exhibits similar behavior. The range of the runtime depends on the instance being solved: for example the runtime of OpenSMT [53] for a fixed instance varies from twofold to several orders of magnitude. Two cumulative runtime distributions for benchmark instances from the SMT-LIB benchmark collection, showing the probability that an instance is solved in time less than a given t, are shown in Figure 3.1. The values are normalized to the minimum measured runtime of the respective instance to make the distributions comparable. Neither behavior is particularly unusual within the benchmark collection, but they represent very different behaviors. For one instance (purple, solid), the runtime ranges from 80 seconds to 400 seconds, resulting in roughly fivefold difference between the slowest and the fastest run. For the other (green, dashed), the runtime ranges from 0.2 seconds to 7.2 seconds, giving a much bigger, 36-fold difference. Both instances are unsatisfiable, and the difference between the two distributions means that the instances will benefit from parallelization approaches in very different ways.

The solving times of some SMT instances seem to obey a *heavy-tailed runtime distribution* [37]. Such distributions have a significant probability of producing 'outlier' samples, that is, a runtime which is far from the median. In practice, the distributions behave as if they had an infinite standard deviation or even an infinite mean. Since SMT solvers in particular in the quantifier-free cases reduce the problem to the satisfiability of a finite-sized propositional formula, the formulas have a finite search space. As the heuristic parameters are usually also finite, the distributions are, technically, finite as well. However, since the search space is in the worst case exponential in the size of the formula, the statistics can in practice be considered to be infinite for suitable formulas [38].

The small variations in the search can result, for example, from explanation generation inside the theory solvers or the process used for selecting decision literals.

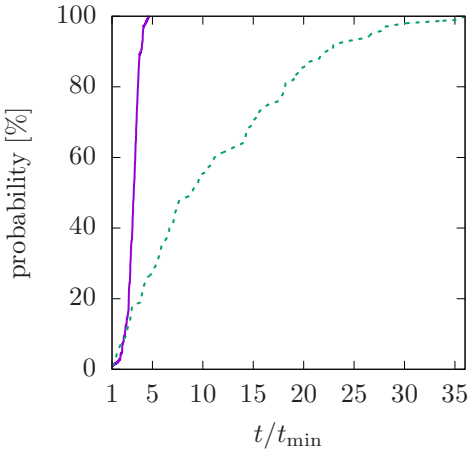

Fig. 3.1: Runtime distributions for two unsatisfiable formulas from the QF_UF category of SMT-LIB. The figure shows the cumulative solving probability with respect to the minimum measured solving time $t_{\min}$

Most heuristics employ randomization to break ties, and often implement a form of deliberate increase in the random behavior either by introducing a *heuristic equivalence parameter* [36] or by simply mixing the *random heuristic* together with a more context-dependent heuristic. Another source of randomness in runtimes can be obtained by using different algorithms for the core theory solvers, such as different variations of the Simplex algorithm, or using different pre-processing techniques that might be detrimental for some instances and very useful for others. Hence it is natural to express the runtime of a solver as a random variable and a related probability distribution.

Let T be the random variable describing the time required to solve a given formula ϕ with a (CDCL-based) SMT solver S randomized using, for example, some of the abovementioned approaches. The *cumulative runtime distribution* $q_T(t)$ gives the probability that $T \leq t$. We will use the cumulative distribution to express the expected time required to solve ϕ with S. By definition, this is

$$\mathbb{E}T = \int_0^\infty t q'_T(t) \mathrm{d}t \,, \tag{3.1}$$

where $q'_T(t)$ is the derivative of the cumulative distribution $q_T(t)$.

3.3.1 Parallel SMT Based on Algorithm Portfolios

The inherent randomness in SMT solver runtimes can be utilized in obtaining a natural parallelization approach. In such approaches the goal is to run in parallel several solvers with different heuristic parameters, such as restart and learning strategies, decision heuristics, or using different theory solvers, on the same formula and obtain the solution from the first solver determining the satisfiability. This *algorithm portfolio approach* [75, 49, 36] has been extensively studied in related areas [56, 57, 64, 71, 55, 33], and has recently proved surprisingly efficient in solving structured formulas [43, 44, 41, 7] in SAT as well as in SMT [83].

We first consider a simplified version of the problem where worker solvers communicate their success or failure in determining satisfiability to a master process. In Section 3.3.2 we extend this case by allowing the worker solvers to share clauses and lemmas with each other either through the master or directly. The plain algorithm portfolio approach is based on running several randomized SMT solvers in a distributed or parallel computing environment on a given formula, and obtaining the result from the first solver that finishes.

One effective approach is to simply introduce a small amount of randomness in the heuristic while keeping the search strategy of the solver otherwise unchanged. This provides an interesting setting for obtaining speed-up as it requires virtually no modification to the underlying solver. The results in, e.g., [83] also suggest that it compares favorably to many other portfolio-based approaches. In this case we are given a randomized solver and a formula such that the probability that the solver solves the instance within time t is $q_T(t)$. Assume now we are given n simultaneously running solvers. As the formula is solved if at least one of the solvers solves the formula within time t, the probability of solving within time t becomes

$$q_{T^n}(t) = 1 - (1 - q_T(t))^n \,. \tag{3.2}$$

Depending on the distribution $q_T(t)$, the expected runtime $\mathbb{E}T^n$ of the simple distribution approach can be be significantly lower than the expected runtime $\mathbb{E}T$ of a single solver.

3.3.2 Lemma Sharing in Portfolios

Since clauses and lemmas learned during solver execution are implied by the original problem, they may be shared freely between solvers, with the purpose of improving the performance of the receiving solver. The challenge with this approach is that the number of lemmas generated by an SMT solver is often very high and transferring all lemmas is too much overhead, so that it often has a detrimental effect on the overall performance. There are two approaches for avoiding this problem. One, taken in [83], is to place a strict limit on the number of literals in the transferred lemmas. The second, followed in [65], is to maintain a centralized database of lem-

mas from which the solvers receive a heuristically determined subset. The former allows a decentralized implementation, whereas the latter allows the use of more sophisticated heuristics. In both cases experiments show that the shared lemmas can improve solver performance significantly [83, 65].

3.3.3 Centralized Lemma Databases

Sharing of learned lemmas plays a central role in parallel SMT. The learned lemmas are transferred to a *lemma database*, where they are filtered using a *parallel lemma-sharing heuristic*, and then passed on to the running solvers. We first define some concepts that help to formalize the working of the database. Let S be a set of lemmas. The size of a lemma set $||S||$ is the total number of literals in S, that is, $||S|| = \sum_{C \in S} |C|$. Unit lemmas, i.e., lemmas consisting of a single literal, are handled specially in the process: they are always stored in the lemma database, and do not contribute to the size of the database.

Algorithm 1 shows a version of the CS-SDSMT algorithm and the related concepts. The lemma database, initialized on line 1, is denoted by LemmaDB, and is annotated with an index j to facilitate the representation of the results. The set U contains the unit lemmas that are already proven to be logical consequences of the input formula ϕ. The shorthand notation $\mathrm{UP}(\phi) = \mathrm{UP}(\phi, \emptyset)$ denotes computing the unit theory propagation closure of ϕ on the empty truth assignment (with no variables fixed to values).

The first part of the loop in lines 5–6 consists of submitting the formula, all unit lemmas, and a heuristically selected subset of LemmaDB of size at most *SubmSize* to the parallel computing environment so that the n computing resources are filled. The next phase is to receive the results in lines 8–14. The $Receive(i)$ function receives, from the resource i, a tuple consisting of the result of the computation, which can be Sat, Unsat, or Error (m/o), and a set L of learned lemmas. If the formula is found either satisfiable or unsatisfiable, the algorithm terminates. Otherwise the set of unit lemmas is updated using the learned lemmas on line 13 and the lemma database is updated on line 14, again using a heuristic function *Merge* and limiting the maximum size of the database to *MaxDBSize*.

The function *Merge* takes a central role in discussing lemma sharing. Firstly, the function acts as a heuristic for selecting learned lemmas, and secondly, it simplifies the learned lemmas using the set of literals U obtained by unit propagation. Two operations are involved in the simplification:

1. removing satisfied lemmas (lemmas C such that $C \cap U \neq \emptyset$), and
2. removing false literals $\neg l$ from lemmas so that for a given lemma C, the simplified lemma becomes $C' = \{l \in C \mid \neg l \notin U\}$.

Algorithm 1: The CS-SDSMT Algorithm

Input : Formula ϕ, n (number of computing elements), *MaxDBSize* (maximum database size), *SubmSize* (maximum submit size)
Output : Sat if ϕ is satisfiable, Unsat otherwise

```
1  LemmaDB^0 := ∅;
2  U := UP(φ);
3  j := 0;
4  while True do
5      for i := 1 to n do
6          Submit(φ ∪ U ∪ Choose(LemmaDB^j, SubmSize));
7          LemmaDB^{j+1} := LemmaDB^j;
8          for i := 1 to n do
9              (result, L) := Receive(i);
10             if result is in {Sat, Unsat} then
11                 return result;
12             else
13                 U := UP(φ ∪ U ∪ LemmaDB^{j+1} ∪ L);
14                 LemmaDB^{j+1} := Merge(U, LemmaDB^{j+1}, L, MaxDBSize);
15                 j := j + 1;
```

3.3.4 Experiments on the Algorithmic Framework

It is interesting to contemplate the different types of heuristics that can be implemented both for *Choose* and *Merge*. This section studies the following four possibilities:

- $Choose_{123}$ only considers lemmas of length 1, 2, or 3. If the size of the resulting database is greater than the limit, the shorter lemmas are preferred. This type of approach is used in many portfolio solvers. For example, [7] only transfers lemmas of length 1 to other solvers, and [44] only lemmas that have at most eight literals.
- $Choose_{\text{len}}$ returns the shortest lemmas. This approach is more general than $Choose_{123}$, as it always returns lemmas even if the argument set contains only lemmas longer than some limit.
- $Choose_{\text{freq}}$ returns the most common learned lemmas. As the parallel search is allowed to overlap, it is not unlikely that the same lemma can be learned many times in different solvers.
- $Choose_{\text{rand}}$ returns a randomly selected set of lemmas.

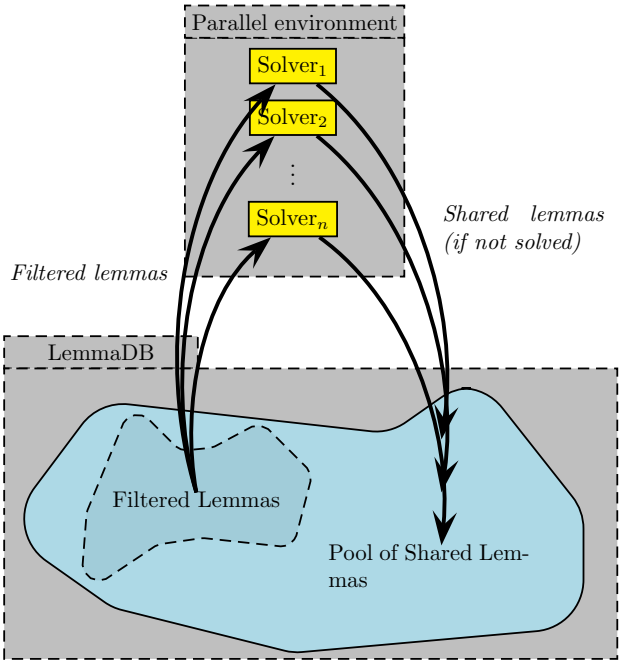

Fig. 3.2: The CS-SDSMT Process

3.3.5 Lemma Sharing and Partitioning

For certain instances partitioning of the search space and forcing the search to be performed on the partitions shows significant speed-up in the experiments. However, partitioning the search space of the formula makes lemma sharing more complicated because of the constraints that result in the SMT solver learning lemmas that might not be logical consequences of other partitions.

3.4 Search-Space Partitioning in SMT

The algorithm portfolio approach described in Section 3.3 does not force the solvers to explore different search spaces on the formula, but instead relies on randomization in the heuristic to produce speed-ups. The idea in portfolios is that it is unlikely that two randomized solvers would be searching for the solution in a similar fashion.

A complementary approach is to use the divide-and-conquer paradigm to force the search performed by the parallel computing units not to overlap with each other. This can be achieved by constraining the original problem into a set of indepen-

dently solvable *derived problems*, finding the solutions for them, and computing the final solution based on the results of the derived problems.

The constraints used for constructing the derived problems can be represented in SMT as conjunctions of clauses (lemmas). This section analyzes the effects of the partitioning approach on the expected time required to determine the satisfiability of a formula.

3.4.1 Plain Partitioning

Plain partitioning is the straightforward approach where an SMT formula ϕ is divided into n derived formulas $\phi_1, \ldots, \phi_n$ that are solved in parallel with an SMT solver S. The derived formulas are obtained with a *partitioning function* and satisfy the following conditions (see Definition 1):

1. $\phi \equiv \phi_1 \vee \ldots \vee \phi_n$, and
2. $\phi_i \wedge \phi_j$ is unsatisfiable if $i \neq j$.

The unsatisfiability of all the derived formulas implies the unsatisfiability of ϕ, whereas it suffices to show one of the derived formulas is satisfiable to prove satisfiability of ϕ. Of particular interest in this section is how much faster a given formula is solved with the plain-partitioning approach compared to solving the formula directly with the solver S.

In the discussion of this section we make an assumption that the partitioning of the instance is done only once and the number of derived formulas is fixed. This is in contrast to many implementations of plain partitioning where it is natural to use a form of load balancing where new derived formulas are constructed from formulas being solved as the satisfiability of previous formulas is determined. As a result, the number of derived formulas n is not fixed in these parallel solvers.

Despite such differences, an analysis of the plain-partitioning approach provides insight into practical parallel solving. The main result in this section is that the plain-partitioning approach is 'risky' in the following sense. Assume that for any cumulative probability distribution $q(t)$ there exists a formula ϕ_q such that the probability of solving ϕ_q with S in time less than or equal to t is $q(t)$. If the partitioning function is from a certain natural class described in Definition 1, and n is fixed and sufficiently large, there is always an unsatisfiable formula so that the expected runtime of the plain-partitioning approach will be higher than the expected runtime of the underlying solver S [52].

The approach is analyzed in a spirit similar to the analysis of the portfolio approach in Section 3.3. In particular, we will assume that given a formula, the time required to determine its satisfiability with a solver S is a random variable T with cumulative distribution $q_T(t)$. To simplify the discussion, we will assume for now that given a number $n \geq 2$, the partitioning function produces n derived instances which are all solved in parallel using n computing elements.

We will first introduce a model describing how a partitioning function affects the runtime distributions of the derived formulas. We assume that the solver S performs with the same probability a given search that takes time t_ϕ in the formula ϕ but, due to the partitioning constraints, a shorter time t_{ϕ_i} in the derived formulas ϕ_i. The efficiency $\varepsilon(n) = t_\phi / t_{\phi_i}$ of the partitioning function is assumed to depend only on the number n of derived formulas. This reasoning results in a model where, given a formula with the runtime distribution $q_T(t)$ on a solver S, the n derived formulas all have the distributions $q_T(\varepsilon(n)t)$.

The efficiency model that will be used in the proof is $\varepsilon(n) = n^\alpha$, where $0 \leq \alpha \leq 1$ is a constant depending on the partitioning function. This model can be motivated in two ways. Firstly, the efficiency satisfies the following natural properties:

1. $1 \leq \varepsilon(n) \leq n$,
2. $\varepsilon(n) \leq \varepsilon(n+1)$, and
3. $(\varepsilon(n))^p = \varepsilon(n^p)$ for all $p \in \mathbb{N}$,

The first condition states that the partitioning function should not make a particular search of S super-linearly faster or slow the search down. The second condition requires that the efficiency does not decrease as more derived formulas are created. The last condition states that if a partitioning function $P(\phi, n)$ is used to produce n^p derived formulas recursively, the resulting efficiency must equal the efficiency of $P(\phi, n^p)$ where the derived formulas are all generated at once.

Secondly, the model $\varepsilon(n) = n^\alpha$ can be derived from the following constructive application of partitioning. Assume there is a procedure for splitting the search space of an arbitrary formula ϕ following the runtime distribution $q_T(t)$ into a fixed number $n_0 \geq 2$ of derived formulas $\phi_1, \ldots, \phi_{n_0}$. Assume further that the derived formulas ϕ_i have runtime distributions $q_T(\beta t)$ where $1 \leq \beta \leq n_0$. Applying this procedure first to ϕ and then recursively to the derived formulas i times in total results in $n = n_0^i$ derived formulas with runtime distribution $q_T(\beta^i t)$. Hence the recursive application of the procedure results in a partitioning function $P(\phi, n)$ defined for values $n = n_0^i$ with efficiency β^i. Since $i = \log_{n_0} n$, we have

$$\beta^i = \beta^{\log_{n_0} n} = e^{\frac{\ln n}{\ln n_0} \ln \beta} = (e^{\ln n})^{\frac{\ln \beta}{\ln n_0}} = n^{\frac{\ln \beta}{\ln n_0}} = n^\alpha \,,$$

where $\alpha = \ln \beta / \ln n_0$. Alternative expressions for the efficiency include a linear model

$$\varepsilon'(n) = \max(\beta n, 1) \,,$$

where $0 \leq \beta \leq 1$ is a constant. However, condition 3 does not hold for $\varepsilon'(n)$. For example, setting $\beta = 0.9$, $n = 2$, and $p = 2$ results in $(\varepsilon'(2))^2 = 3.24$, while $\varepsilon'(4) = 3.6$. We are now ready to define the partitioning function more precisely.

Definition 1. Given a formula ϕ with runtime distribution $q_T(t)$ on solver S and a partitioning factor $n \geq 2$, a *partitioning function* $P : (\phi, n) \mapsto (\Pi_1, \ldots, \Pi_n)$ is a function mapping the formula ϕ to n *partitioning constraints* $\Pi_1, \ldots, \Pi_n$. The partitioning constraints produce n *derived formulas* $\phi_i = \phi \wedge \Pi_i, 1 \leq i \leq n$. The derived formulas then satisfy the following two properties:

1. $\phi \equiv \phi_1 \vee \ldots \vee \phi_n$, and
2. $\phi_i \wedge \phi_j$ is unsatisfiable for all $i \neq j$.

The runtime distribution of each of the derived formulas on solver S is described by the probability distribution $q_T(\varepsilon(n)t)$, where

$$\varepsilon(n) = n^\alpha, 0 \leq \alpha \leq 1 \tag{3.3}$$

describes the *efficiency* of the partitioning function.

We will denote by $\mathbb{E}T^n_{\text{plain}(\alpha)}$ the expected time required to determine the satisfiability of ϕ with the plain-partitioning approach using a partitioning function with efficiency $\varepsilon(n) = n^\alpha$. A partitioning function is called *void* if $\alpha = 0$ and hence $\varepsilon(n) = 1$. In this case all the derived instances are as difficult to solve as the original formula. A partitioning function is called *ideal* if $\alpha = 1$, that is, $\varepsilon(n) = n$.

With these definitions, we are now ready to show that for non-ideal partitioning functions there are distributions where solving with plain partitioning is slower than solving with the underlying solver.

Proposition 1. *Let $P(\phi, n)$ be a partitioning function as in Definition 1, where $0 \leq \alpha < 1$, and S a SAT solver. Then for every n and every α there exists a distribution $q_n(t)$ such that if the solving of an unsatisfiable instance follows $q_n(t)$ on S, then the expected runtime $\mathbb{E}T$ of S is lower than the expected runtime*

$$\mathbb{E}T^n_{\text{plain}(\alpha)}$$

of the plain-partitioning approach.

Proof. The family of distributions $q_n(t)$ we will use in the proof is

$$q_n(t) = \begin{cases} 0 & \text{if } t < t_1, \\ 1 - \frac{1}{n} & \text{if } t_1 \leq t < t_2, \textit{and} \\ 1 & \text{if } t \geq t_2, \end{cases} \tag{3.4}$$

where $t_1 < t_2$. Thus the probabilities that the formula is solved by S in exactly time t_1 is $1 - 1/n$ and in time t_2 is $1/n$. The expected runtime for a formula following the distribution $q_n(t)$ on S is

$$\mathbb{E}T = (1 - \frac{1}{n})t_1 + \frac{1}{n}t_2 \,. \tag{3.5}$$

The expected runtime of the plain-partitioning approach using the partition function $\varepsilon(n) = n^\alpha$ can be derived by noting that all derived formulas need to be solved before the result can be determined. This means that either all solvers are 'lucky', and determine the unsatisfiability in time t_1/n^α, or at least one of the solvers runs for time t_2/n^α, which will then become the runtime of the approach. This results in

$$\mathbb{E}T^n_{\text{plain}(\alpha)} = \left(1 - \frac{1}{n}\right)^n \frac{t_1}{n^\alpha} + \left(1 - (1 - \frac{1}{n})^n\right) \frac{t_2}{n^\alpha} \,. \tag{3.6}$$

We claim that for every α, there are values for n, t_1, and t_2 such that $\mathbb{E}T < \mathbb{E}T^n_{\text{plain}(\alpha)}$. Dividing both sides of the resulting inequality by t_2 and setting $k = t_1/t_2$ results in

$$(1-\frac{1}{n})k+\frac{1}{n} < \frac{(1-\frac{1}{n})^n}{n^\alpha}k+\frac{1-(1-\frac{1}{n})^n}{n^\alpha} ,$$

which we reorder to

$$k\left((1-\frac{1}{n})-\frac{(1-\frac{1}{n})^n}{n^\alpha}\right) < \frac{1-(1-\frac{1}{n})^n}{n^\alpha}-\frac{1}{n} .$$

Note that $(1-\frac{1}{n}) > (1-\frac{1}{n})^n/n^\alpha$ when $n \geq 2$, and therefore the left-hand side of the inequality is positive and can be made arbitrarily small by setting k small. It remains to show that the right-hand side of the inequality is positive for sufficiently large n, i.e.,

$$\frac{n-(1-\frac{1}{n})^n n-n^\alpha}{n^{\alpha+1}} > 0 .$$

Since $n^{\alpha+1}$ is always positive, we may simplify this and factor n from the denominator, resulting in

$$1-(1-\frac{1}{n})^n-n^{\alpha-1} > 0 . \tag{3.7}$$

Noting that $\lim_{n\to\infty}(1-\frac{1}{n})^n = \frac{1}{e} \approx 0.3$, and that $\lim_{n\to\infty} 1-n^{\alpha-1} = 1$ if $\alpha < 1$, we get the desired result, that is, for sufficiently large n, there are values t_1 and t_2 such that $t_1 < t_2$ and $\mathbb{E}T < \mathbb{E}T^n_{\text{plain}(\alpha)}$.

The following example illustrates the performance of the plain-partitioning approach for distributions of type Equation (3.4).

Example 3. Assume there is a formula following the distribution $q_{20}(t)$ such that $t_1 = 1$ and $t_2 = 1000$, and a partition function $\varepsilon(n) = n^{0.7}$ for this formula. The expected runtime of the solver S, given by Equation (3.5), is $\mathbb{E}T \approx 50.95$, while the expected runtime of the plain-partitioning algorithm, from Equation (3.6), is $\mathbb{E}T^{20}_{\text{plain}(0.7)} \approx 78.84$. The scalability of the expected runtime $\mathbb{E}T^n_{\text{plain}(\alpha)}$ of the plain-partitioning approach is shown for the distribution $q_{20}(t)$ for different values of α in Figure 3.3.

Note that the proof does not hold if the partitioning function is ideal, since the left-hand side of Inequality (3.7) is negative if $\alpha = 1$. In fact the condition that the partitioning function be ideal turns out to be sufficient to guarantee that the expected runtime of the plain-partitioning approach is never higher than the expected runtime of S, that is, $\mathbb{E}T \geq \mathbb{E}T^n_{\text{plain}(1)}$ for all n and T. To see this, we will first derive an expression for $\mathbb{E}T^n_{\text{plain}(\alpha)}$ for an arbitrary distribution $q_T(t)$ and an arbitrary partitioning function.

Let $q_T(t)$ be a runtime distribution of an unsatisfiable formula ϕ with a randomized SAT solver S, and $t_{\max}$ the maximum time required to solve ϕ with S (hence

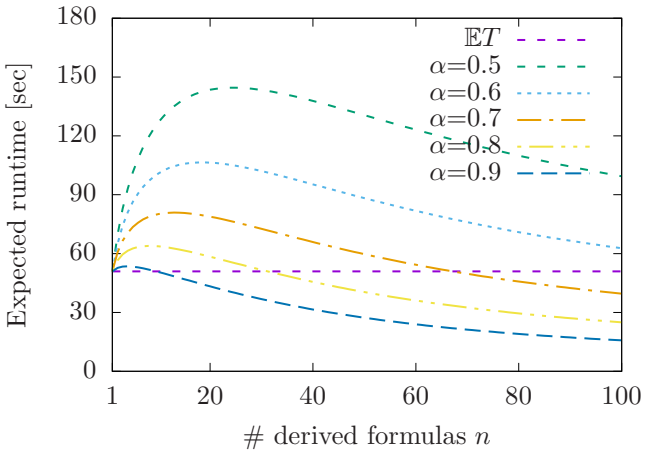

Fig. 3.3: The scalability of the plain-partitioning approach for the distribution $q_{20}(t)$ in Equation (3.4) where $t_1 = 1$ and $t_2 = 1,000$

$q_T(t) = 1$ if $t \geq t_{\max}$ and $q_T(t) < 1$ otherwise). The n partitions have runtime distributions $q_T(\varepsilon(n)t)$ and since they all need to be shown unsatisfiable, the runtime distribution of the plain-partitioning approach is $q_T(\varepsilon(n)t)^n$. Hence by Equation (3.1) the expected runtime of the plain-partitioning approach is given by

$$\mathbb{E}T^n_{\text{plain}(\alpha)} = \int_0^{t_{\max}} t \frac{\mathrm{d}}{\mathrm{d}t} q_T(\varepsilon(n)t)^n \mathrm{d}t,$$

where $\frac{\mathrm{d}}{\mathrm{d}t} q_T(\varepsilon(n)t)^n = n\varepsilon(n) q_T(\varepsilon(n)t)^{n-1} q'_T(\varepsilon(n)t)$ is the derivative of the distribution function. Substituting $\varepsilon(n)t = \tau$ above, the expected runtime can be written

$$\begin{aligned} \mathbb{E}T^n_{\text{plain}(\alpha)} &= \int_0^{t_{\max}} \tfrac{\tau}{\varepsilon(n)} n\varepsilon(n) q_T(\tau)^{n-1} q'_T(\tau) \tfrac{\mathrm{d}\tau}{\varepsilon(n)} \\ &= \int_0^{t_{\max}} \tfrac{n}{\varepsilon(n)} \tau q_T(\tau)^{n-1} q'_T(\tau) \mathrm{d}\tau \,. \end{aligned} \tag{3.8}$$

We can now state the following proposition that increasing the number of derived instances in ideal plain partitioning does not result in increased expected runtime.

Proposition 2. *Let $n \geq 1$, $\varepsilon(n) = n^1 = n$ be the efficiency of an ideal partitioning function, and $q_T(t)$ be the runtime distribution of an unsatisfiable formula with a randomized solver. Then $\mathbb{E}T^n_{\text{plain}(1)} \geq \mathbb{E}T^{n+1}_{\text{plain}(1)}$.*

Proof. Substituting $\varepsilon(n) = n$ in Equation (3.8) results in

$$\mathbb{E}T^n_{\text{plain}(1)} = \int_0^{t_{\max}} \tau q_T(\tau)^{n-1} q'_T(\tau) \mathrm{d}\tau \,.$$

Since $q_T(\tau) \leq 1$ when $0 \leq \tau \leq t_{\max}$, we immediately have the desired result $\mathbb{E}T^n_{\text{plain}(1)} \geq \mathbb{E}T^{n+1}_{\text{plain}(1)}$.

Finally from Propositions 1 and 2 we get the main result concerning unsatisfiable instances.

Proposition 3. *The expected runtime of the plain-partitioning approach,*

$$\mathbb{E}T^n_{\text{plain}(\alpha)} \ ,$$

is guaranteed not to be higher than the expected runtime $\mathbb{E}T$ of the underlying solver S if and only if the partitioning function is ideal, that is, $\alpha = 1$.

It is a strong requirement that the efficiency of a partitioning function must be ideal in order to never increase the time required to solve a formula, and it would be tempting to draw the conclusion that this requirement is never met. The practical implications of the above negative result are not as dramatic. However, it is not completely impossible that unsatisfiable formulas have such pathological distributions, even when the solvers employ restart strategies known to eliminate this type of behavior [37]. Furthermore, it is not impossible for the partitioning function to provide even super-linear speed-ups if, for example, the partitioning constraints are related to the *back door set* [82] of the formula.

3.5 Decomposition

In some cases where partitioning is not applicable and where portfolios require infeasible amounts of memory in practice, a different approach is required. For instance, suppose the input formula is too large to fit into the local memory. In this case the problem must be *decomposed* into a series of smaller problems, which, in contrast to a partitioning, do not compose disjunctively.

Definition 2 (Decomposition). Let ϕ be a first-order formula in conjunctive normal form, i.e., $\phi = \phi_1 \wedge \ldots \wedge \phi_n$. A *decomposition* of ϕ into k sub-formulas is a set of formulas $\{\psi_1, \ldots, \psi_k\}$ such that

- each $\psi_i \subseteq \bigcup_{j \in J} \phi_j$, for some selection of indices J, and
- each ϕ_i is included in at least one ψ_i.

Note that the original problem ϕ is unsatisfiable if at least one of the ψ_i in the decomposition is unsatisfiable, but the converse is not a sufficient criterion for satisfiability, i.e., each ψ_i being satisfiable does not imply that ϕ is satisfiable.

In theory, this type of decomposition is 'ultimately lazy' in that it does not require us to extract any other type of semantic information embedded in the input problem, apart from the number of clauses. This enables us to decompose very large input problems efficiently; for instance, we can simply send a random selection of $\frac{1}{n}$th of the clauses to each of n processors or nodes, without ever inspecting their content.

The cost that we pay for this laziness is then in the reconciliation of (partial) models: suppose that we obtain (partial) models μ_i from independent SMT solver queries, one for each corresponding ψ_i, and let $v(\psi_i)$ be the variables in ψ_i. Two models μ_i, μ_j are trivially 'compatible' if the corresponding ψ_i, ψ_j do not share variables, i.e., when $v(\psi_i) \cap v(\psi_j) = \emptyset$. Models may however not be reconcilable if they assign different values to variables that occur in both ψ_i and ψ_j. This criterion is easy to check for simple models that map theory variables to numerals, but we should keep in mind that for more complex background theories this may amount to function equivalence checks that are not polynomial-time decidable, as is the case, for instance, for some problems involving arrays or quantifiers.

The reconciliation process is perhaps best understood by conceptual introduction of one additional sub-formula σ (and perhaps an associated SMT solver and/or processor or node) which we use to track models for the shared variables *only*. Initially, we pick an arbitrary assignment μ_σ to those shared variables, which we propagate to all ψ_i, after which the resulting, modified ψ_i do not share any variables anymore, which means that all ψ_i can now be solved in parallel without communication between the computing elements, while we know that, if all sub-formulas are found to be satisfiable, the conjunction of the corresponding models μ_i are extensions of μ_σ and thus their conjunction is, trivially, a model for ϕ.

Usually however, there will be at least one ψ_i which is not satisfiable under μ_σ. Thus, we have $\psi_i \Rightarrow \neg\sigma$ for at least one ψ_i, and $v(\psi_i) \cap v(\sigma) \subseteq v(\phi)$, where we expect $v(\psi_i) \cap v(\sigma)$ to be a small set in practice. This is thus, essentially, an interpolation problem.

Lemma 1. *Let $\phi = \psi_1 \wedge \ldots \wedge \psi_k$, let μ_σ be a model for the set of shared variables $V = \bigcup_{(i,j)\in C} v(\psi_i) \cap v(\psi_j)$, where $C = \{(i,j) \mid 1 \leq i < j \leq k\}$. If I is an interpolant for $\psi_i \Rightarrow \neg\mu_\sigma$ over V, i.e., we have $\psi_i \Rightarrow I \wedge I \Rightarrow \neg\mu_\sigma$ and $v(I) \subseteq v(\psi_i) \cap v(\mu_\sigma)$, then $\phi \Rightarrow I$.*

Proof. We have $\psi_i \Rightarrow I \wedge I \Rightarrow \neg\mu_\sigma$ by Craig's interpolation lemma. By construction we also have $\phi \Rightarrow \psi_i$; thus it follows that $\phi \Rightarrow I$ (and $\phi \Rightarrow \neg\mu_\sigma$ as in traditional learned clauses and lemmas).

Thus, the interpolant I is implied by ϕ, which means we may, conceptually, (conjunctively) add I to ϕ, to σ, or to its corresponding ψ_i, while preserving satisfiability, just as we would keep a learned clause or lemma (in a local or a shared lemma database). This enables us to formulate a sound and (relatively) complete reconciliation algorithm for decompositions as presented in Algorithm 2.

3.5.1 Experimental Evidence

While decomposition is necessary for very large formulas, there are some indications that the concept itself may help to improve performance on small, hard problems as well. So far, this has been shown to be the case for some propositional

Algorithm 2: An interpolation-based reconciliation algorithm

```
   Input  : Formula φ
   Output : Sat if φ is satisfiable, Unsat otherwise
1  ψ_1, ..., ψ_k := decompose(φ);
2  σ := ∅;
3  flag := true;
4  while flag do
5      if σ is Unsat then
6          return Unsat;
7      else
8          Let μ_σ be a model for σ;
9      flag := false;
10     foreach i in 1...k do
11         if ψ_i ∧ μ_σ is Unsat then
               /* ¬(ψ_i ∧ μ_σ) ⇔ ψ_i ⇒ ¬μ_σ is valid */
12             Let I be an interpolant for ψ_i ⇒ ¬μ_σ (over v(ψ_i) ∩ v(μ_σ));
13             σ := σ ∧ I;
14             flag := true;
15 return satisfiable;
```

problems, for which multiple different interpolation techniques exist that can be compared. The experiments we present here are an example of this, on the small, but hard benchmark instances by Aloul et al. [2], which contain symmetries that can be broken by addition of symmetry-breaking predicates. Ideally, a good interpolation approach is able to detect such symmetries and it automatically breaks them by finding interpolants corresponding to symmetry-breaking predicates.

Figure 3.4 shows the runtime obtained for MiniSAT 1.14p [30] on each of Aloul et al.'s benchmarks, when decomposed into up to 50 sub-formulas (where '1' corresponds to no decomposition being performed). A time limit of 3,600 seconds is enforced and all averages are computed with memory-out problems counted as $36,000 = 10 \times$ time limit. For representation of interpolants, both McMillan's and HKP interpolants, Reduced Boolean Circuits [1] are used and they are computed along a resolution proof found by MiniSAT. Since they are general expressions and not in conjunctive normal form yet, they are then added to the formula via Tseitin transformation (including introduction of new variables). To compute the runtime of a decomposition, all iterations of Algorithm 2 are executed *sequentially*, i.e., the runtime improvements we see in Figure 3.4 describe a completely sequential algorithm, yet for decompositions into about 10 or more sub-formulas, the average runtime over all benchmark problems is smaller than the average runtime of the unmodified MiniSAT. Of course, for an effective parallelization of this algorithm, it is necessary to implement a proper load-balancing strategy. More details of this evaluation are provided by Hamadi et al. [45].

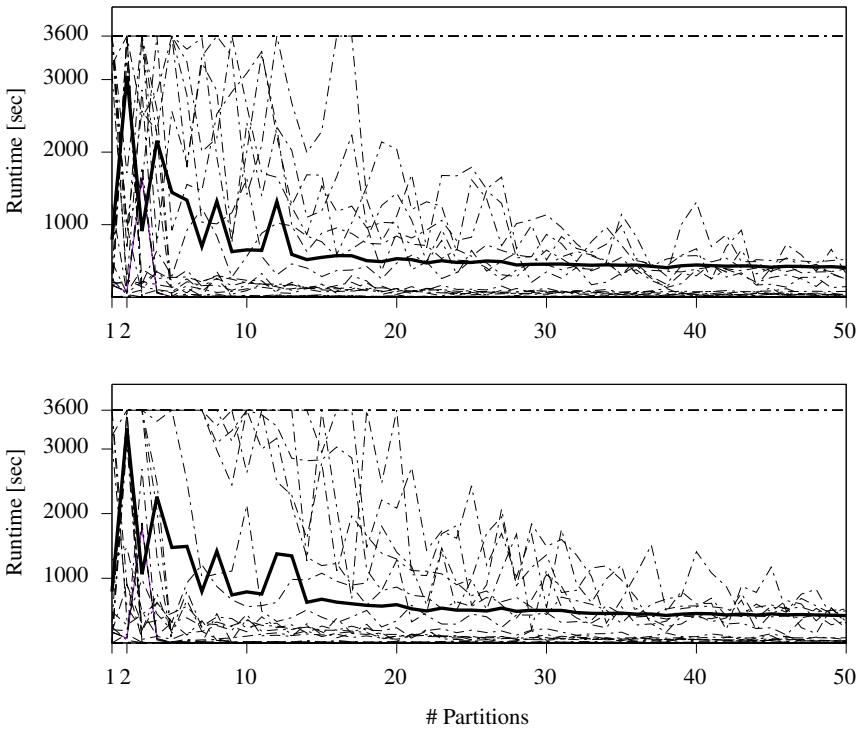

Fig. 3.4: Decompositions into up to 50 sub-formulas. Every line corresponds to one benchmark problem; their average is the bold line. Reconciliation via McMillan interpolants (top) and HKP interpolants (bottom)

3.5.2 Variations and Extensions

Clearly, keeping a single additional formula γ for assignments to shared variables is not ideal for all types of problems. For instance, there may be multiple non-overlapping sets of shared variables, such that γ itself may be decomposed into multiple parts trivially. Since all sub-formulas ψ_i as well as γ may grow (by addition of learned clauses and lemmas), a dynamic decomposition approach, which introduces new ψ_i on demand, may be a good choice in practice.

Furthermore, in any given decomposition, it is not strictly necessary for all sub-formulas (and sub-solvers) to communicate with all others. Solvers only need to communicate with other solvers if their sub-formulas do in fact share variables, in which case they communicate satisfying assignments and interpolants to each other. This means that all sub-solvers may solve their problems independently, sharing their satisfying assignments with those solvers that solve problems involving shared variables. In return, and completely asynchronously, the recipients of satisfying as-

signments may respond with an interpolant that excludes at least one particular variable assignment, at a later time. The process terminates when all interpolants have been received and all satisfying assignments have been consumed, which indicates satisfiability of the whole problem; unsatisfiability is detected by one of the interpolants becoming false. In this fashion, it is possible to construct a completely asynchronous system in which no solver needs to have access to all of the data at any given time, while termination of the algorithm is determined by a distributed consensus algorithm (or variation thereof).

It is worth noting that Nelson/Oppen theory combination is essentially also a decomposition in the sense defined here, with reconciliation taking place via a particular kind of interpolant. A decomposition, however, does not require us to separate sub-formulas into sets of constraints belonging to the same theory. Instead, we may have *multiple* theory solvers of the *same* conceptual theory. For instance, we may have two solvers for the theory of linear integers, each of them solving only a subset of all (purified) linear integer constraints, exchanging interpolants between them.

3.6 Combinations of Parallelization Algorithms

It is useful to analyze the differences between the three parallelization approaches, portfolio, partitioning, and decomposing, discussed in the preceding sections. For instance Bonacina [9] constructs a taxonomy of the approaches based on this distinction, while Grama and Kumar [39] provides an overview of the different parallelization approaches for constraint solving using essentially the same distinction. Attempts at understanding the differences of in particular the portfolio and partitioning approaches include Bordeaux, Hamadi, and Samulowitz [11], and Bonacina [9].

A complementary approach to understanding the differences between the approaches is to try to combine their strengths. Some work towards this has been done in Bonacina [10]; Segre et al. [78] on a parallel SAT-solving approach called *nagging*; Hyvärinen, Junttila, and Niemelä [50] on the parallelization approach based on SAT solving through *scattering*; Dequen, Vander-Swalmen, and Karajecki [25], which implements a similar approach; Ohmura and Ueda [70], which implements a *safe-partitioning* approach on computing clusters for SAT solving; and Gebser et al. [33], which implements a plain-partitioning approach strengthened with a dedicated solver solving the original, unpartitioned formula.

In this section we go a step further, giving a generic framework for combining partitioning and portfolio called *parallelization trees* [54], which represents the instances of parallel algorithms as and/or trees. We give a more in-depth analysis of three of the approaches that we find particularly interesting; parts of this have previously been presented in [52] and [51].

3.6.1 The Parallelization Tree

A way to represent the combination of partitioning and portfolio in a unified framework is the *parallelization tree* abstract algorithmic framework. The idea is to provide a unified way of presenting and comparing different parallelization algorithms. The parallelization tree consists of two types of nodes: *and-nodes* and *or-nodes*. The tree is constructed using the following simple rules:

- The root and the leaves of the parallelization tree are and-nodes.
- Each and-node is associated with an SMT instance and, with the possible exception of the root of the parallelization tree, with one or more SMT solvers.
- All children of an and-node are or-nodes,
- All children of an or-node are and-nodes.

In a more formal treatment we adapt the partitioning function of Definition 1 to the construction of the parallelization tree through the operator $\text{split}^k(n_1, \ldots, n_k, \phi)$.

Definition 3. The result of applying the operator split^k on an and-node ϕ is a tree rooted at the and-node ϕ with k children $o_1, \ldots, o_k$. Each child node o_i is an or-node and has as children the and-nodes $a^i_1, \ldots, a^i_{n_i}$. Finally, each and-node a^i_j is associated with the partition obtained by applying the (randomized) partitioning function P_{n_i} of Definition 1 on the formula ϕ.

The satisfiability of the instance is determined by the solvers and the tree structure as follows:

- The instance at the root of the parallelization tree is satisfiable if any instance among the and-nodes is shown satisfiable.
- A subtree rooted at an and-node is unsatisfiable if one of its children is unsatisfiable or at least one of the solvers associated with the and-node has shown the instance unsatisfiable.
- A tree rooted at an or-node is unsatisfiable if every tree rooted at its children is unsatisfiable.

We immediately obtain both the plain-partitioning and the portfolio approaches as instances of the parallelization tree approach:

- The *plain* partitioning approach $\text{plain}(n, \phi)$ corresponds to the parallelization tree $\text{split}^1(n, \phi)$ where each of the instances associated with the nodes $a^1_1, \ldots, a^1_n$ is solved with a single SMT solver.
- The *portfolio* approach $\text{portf}(k, \phi)$ corresponds to the parallelization tree consisting of the root associated with the instance ϕ and using k SMT solvers to solve the instance.

However, in addition to these two algorithms the parallelization tree approach allows us to easily define other, less trivial algorithms from the literature:

- The *safe*-partitioning approach $\text{safe}(n, s, \phi)$ corresponds to the parallelization tree $\text{split}^1(n, \phi)$ and solving each of the instances $a^1_1, \ldots, a^1_n$ with s SMT solvers.

- The *repeated*-partitioning approach $\mathrm{rep}(n,k,\phi)$ corresponds to the parallelization tree $\mathrm{split}^k(n,\ldots,n,\phi)$ where each instance associated with the nodes

$$a_1^1,\ldots,a_n^1,\ldots,a_1^k,\ldots,a_n^k$$

  is solved with one SMT solver.
- The *iterative*-partitioning approach $\mathrm{iter}(k,\phi)$ corresponds to the infinite parallelization tree where every instance associated with an and-node is solved with a single SMT solver and every and-node associated with an instance ϕ_a has the single or-child and and-grandchildren constructed by applying the operator $\mathrm{split}^1(n,\phi_a)$.

Figure 3.5 illustrates the corresponding parallelization trees and the solver assignments. When clear from the context, we omit the formula ϕ as well as the other parameters from the partitioning approach.

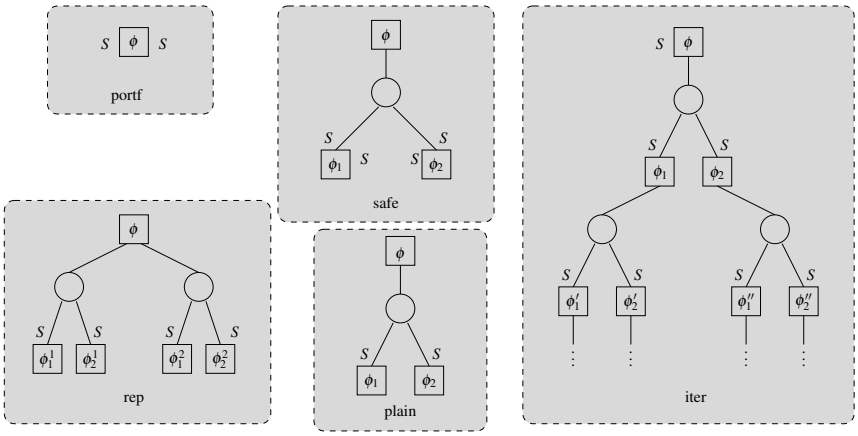

Fig. 3.5: Example parallelization trees (clockwise from the top left): $\mathrm{portf}(2,\phi)$, $\mathrm{safe}(2,2,\phi)$, $\mathrm{iter}(2,\phi)$, $\mathrm{plain}(2,\phi)$, and $\mathrm{rep}(2,2,\phi)$. The and-nodes are drawn with boxes, and the or-nodes with circles. The SMT solvers are indicated with the symbol *S* (Figure adapted from [54])

Concrete SMT instantiations of the parallelization tree include the CVC4 and Z3 SMT solvers, which implement a portfolio, and PBoolector [74], which implements an iterative-partitioning approach. The OpenSMT2 solver [53] implements the full parallelization tree framework.

3.6.2 Iterative Partitioning with Partition Trees

The result of Proposition 3, showing that the plain-partitioning approach is 'vulnerable' to certain distributions of unsatisfiable formulas, raises the question whether there are other solving techniques that use a partitioning function but are immune to the increased expected runtimes in all unsatisfiable cases. Given an unsatisfiable formula, the challenge in plain-partitioning is that the number of formulas needed to show unsatisfiability increases as more derived formulas are produced.

A trivial solution to this problem is to attempt to solve both the formula ϕ and the derived formulas using $n+1$ computing elements. This solution corresponds to solving the formula with the plain-partitioning approach and the underlying solver S in parallel, and guarantees that the expected runtime of the approach will be at most as high as the expected runtime of S. However, by Proposition 3, it is possible that the runtime of the plain-partitioning approach increases as more resources are used, which adversely affects the behavior of the proposed solution.

The *iterative-partitioning approach* [50], is based on a hierarchical partitioning of formulas into increasingly constrained derived formulas, which are organized as a tree. The satisfiability of the original formula is then determined by solving a sufficient number of the derived formulas independently with S. The intuition behind the approach is that the possible increase of the expected runtime due to Proposition 3 is avoided since every time a formula is partitioned, there is an added attempt to solve the unpartitioned formula directly.

This section gives a formalization and an analysis of the iterative-partitioning approach using the concept of a *partition tree* defined as follows.

Definition 4. A *partition tree* $\mathscr{T}_\phi$ of a formula ϕ is a finite n-ary tree rooted at v_0. The nodes v_i are associated with constraints: the constraints of the root consist of the formula ϕ and the constraints of the other nodes are obtained using a partitioning function on their parents. More precisely,

$$Constr(v_0) := \phi \ ,$$

and given a node v_i, its children $v_{i,1}, \ldots, v_{i,n}$, and a rooted path $v_0, \ldots, v_i$ in the partition tree, the partitioning constraints of the child nodes are

$$Constr(v_{i,k}) := \Pi_k \text{ where } \Pi_k \in P(Constr(v_0) \wedge \ldots \wedge Constr(v_i), n) \ .$$

Finally, each node v_i represents the derived formula

$$\phi_{v_i} := Constr(v_0) \wedge \ldots \wedge Constr(v_i) \ .$$

In the iterative-partitioning approach a partition tree $\mathscr{T}_\phi$ is constructed in breadth-first order and the solving of each derived formula ϕ_{v_i} is attempted in parallel with a solver S until the satisfiability of ϕ is determined. The satisfiability of a node v_i is determined either by solving ϕ_{v_i} with S, or determining the satisfiability of all the child nodes $v_{i,1}, \ldots, v_{i,n}$.

The iterative-partitioning approach guarantees that its expected runtime does not increase as more computing elements are introduced, even if the partitioning function is void. We will show this for partition trees $\mathscr{T}_\phi^k$, where all rooted paths to the leaves are of length k. As is conventional, we say that the height of $\mathscr{T}_\phi^k$ is k.

Proposition 4. *Let ϕ be an unsatisfiable formula, $\mathscr{T}_\phi^k$ and $\mathscr{T}_\phi^m$ be two partition trees of height k and m, respectively, constructed with a void partition function, and $k < m$. Then the expected runtime of the partition tree approach using $\mathscr{T}_\phi^m$ is less than or equal to the expected runtime of the partition tree approach using $\mathscr{T}_\phi^k$.*

Proof. We show by induction on the height of the partition tree that the probability that ϕ is solved within time t cannot decrease, from which the claim follows. Let $q(t)$ be the probability that ϕ is solved sequentially within time t, $q'(t)$ be its derivative at t, and let $q_i(t)$ denote the probability that ϕ is solved within time t using a partition tree $\mathscr{T}_\phi^i$ of height i. Then the probability $q_0(t) = q(t)$. The probability that the formula is solved within time t with the partition tree approach using a tree of height one is $q_1(t) = \int_0^t (q'(\tau) + (1-q(\tau))nq'(\tau)q(\tau)^{n-1})\mathrm{d}\tau$, that is, the integral of the sum of probability $q'(\tau)\mathrm{d}\tau$ that the formula is solved in the root of the tree at time τ, and the probability that the formula has not been solved in the root, has been solved by all children but one by time τ, and is solved at time τ in the last child. A direct calculation shows that $q_1(t) \geq q_0(t)$. Assume now that $q_k(t) \geq q_{k-1}(t)$ for all $t \geq 0$. As previously, $q_{k+1}(t) = \int_0^t (q'(\tau) + (1-q(\tau))nq_k'(\tau)q_k(\tau)^{n-1})\mathrm{d}\tau = q(t) + q_k(t)^n - \int_0^t q(\tau)nq_k'(\tau)q_k(\tau)^{n-1}\mathrm{d}\tau$. Integration by parts on the negative term results in $q_{k+1}(t) = q(t) + q_k(t)^n - q_k(t)^n q(t) + \int_0^t q_k(\tau)^n q'(\tau)\mathrm{d}\tau = q(t) + (1-q(t))q_k(t)^n + \int_0^t q_k(\tau)^n q'(\tau)\mathrm{d}\tau$. By the induction hypothesis $q_{k+1}(t) \geq q(t) + (1-q(t))q_{k-1}(t)^n + \int_0^t q_{k-1}(\tau)^n q'(\tau)\mathrm{d}\tau = q_k(t)$.

In practice the construction of the tree is not atomic, but the nodes of the tree are expanded at different times in breadth-first order. As the construction of the tree is not immediate, the tree expansion can use information obtained from earlier solving attempts. The straightforward way to use this information, as in Example 4 (and first published in [50]), is not to expand a subtree rooted at a formula shown unsatisfiable. This example further illustrates the use of iterative partitioning and the related partition tree.

Example 4. Figure 3.6 illustrates how the partition tree approach runs in an environment with $m = 8$ parallel resources. The left tree shows the initial setup, and the right tree shows how the solving has proceeded after one of the SAT solvers terminates in a memory out and three of the solvers return unsatisfiable for their respective formulas. In both trees the shaded area indicates the set of formulas currently being solved. The formulas shown unsatisfiable are labeled with Unsat and the formula that has exceeded its resource limit is labeled with Error (m/o) on the right-hand side tree. There is no need to solve $v_{0,1,1}$ once $v_{0,1,1,1}$ and $v_{0,1,1,2}$ are shown unsatisfiable.

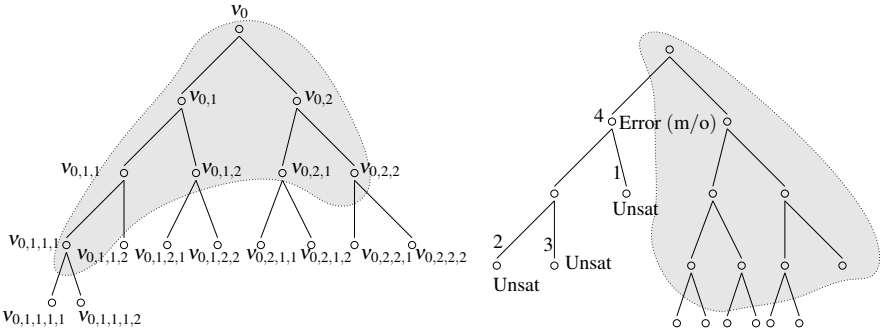

Fig. 3.6: Illustration of the partition tree approach. The shaded area represents jobs running simultaneously, the numbers indicate the order in which the jobs terminate, and the solid lines represent the edges of the tree

3.6.3 Safe and Repeated Partitioning

Another approach to avoiding the increase of expected runtime in solving unsatisfiable instances is to combine the plain-partitioning approach with randomization. This way the inherent randomness in runtimes of SAT and SMT solvers and the reduction in search space provided by the partitioning function can be used simultaneously to improve performance. We present two such composite approaches:

- *Safe partitioning* uses the partitioning function to derive formulas each of which is solved with the portfolio approach; and
- *Repeated partitioning* produces several sets of derived formulas with a partitioning function, and solves these sets in parallel using one solver per derived instance.

The use of safe partitioning has been suggested in [70, 33], whereas the repeated-partitioning approach is closely related to *hard restarts* in guiding-path-based approaches (e.g., [33]). Here, we analyze a setting where n^2 resources are used so that in safe partitioning the partitioning function results in n partitions that are solved using n solvers each. In repeated partitioning the partitioning function is repeated n times for the same formula, resulting again in n^2 formulas.

Safe partitioning applies a partitioning function $P(\phi,n) = (\Pi_1,\ldots,\Pi_n)$, and solves each derived formula $\phi \wedge \Pi_i$, $1 \leq i \leq n$, with a portfolio of n solvers. It suffices then to show each derived instance unsatisfiable with one solver. Intuitively this approach improves performance because derived formulas should be easier to solve than the original formula, and, assuming the solving times of the derived formulas obey a non-trivial random distribution, the portfolio approach results in lower runtimes for the derived formulas. The repeated-partitioning approach, on the other hand, consists of applying a family of partitioning functions $P^j(\phi,n) = (\Pi_1^j,\ldots,\Pi_n^j)$, $1 \leq j \leq n$, and solving each derived formula $\phi \wedge \Pi_i^j$,

$1 \leq i \leq n$, $1 \leq j \leq n$ with a solver S. To show a formula unsatisfiable it suffices to show unsatisfiable any set of derived formulas $\phi \wedge \Pi_1^k, \ldots, \phi \wedge \Pi_n^k$ for a fixed k. This approach is expected to provide speed-ups as the derived formulas are easier to solve than the original formula, but also because it is possible that one of the partitioning functions P^j performs better than some other partitioning function.

Based on the definition we can immediately give the runtime distributions of the two composite approaches using Equations (3.2) and (3.8) for simple distribution and plain-partitioning. The cumulative runtime distribution for safe partitioning of unsatisfiable formulas $q_{T_{\text{safe}}}(t)$ is given by substituting $q_T(t)$ in (3.2) by (3.8), yielding

$$q_{T_{\text{safe}}}(t) = (1-(1-q(\varepsilon(n)t))^n)^n \,, \tag{3.9}$$

and the repeated partitioning by substituting $q_T(t)$ in (3.8) by (3.2), resulting in

$$q_{T_{\text{rep}}}(t) = 1-(1-q(\varepsilon(n)t)^n)^n \,. \tag{3.10}$$

From Equations (3.9) and (3.10) it follows that the expected runtime of the repeated partitioning is always at least the expected runtime of the safe partitioning, independent of the partitioning function or number of computing elements (n).

Proposition 5. *Let $q_T(t)$ be the runtime distribution of an unsatisfiable formula. Then $\mathbb{E}T_{\text{safe}} \leq \mathbb{E}T_{\text{rep}}$.*

Proof. Since $\mathbb{E}T_{\text{safe}} \leq \mathbb{E}T_{\text{rep}}$ if $q_{T_{\text{safe}}}(t) \geq q_{T_{\text{rep}}}(t)$ for all $0 \leq t \leq t_{\max}$, it suffices to show that that $q_{T_{\text{rep}}}(t) = 1-(1-q(\varepsilon(n)t)^n)^n \leq q_{T_{\text{safe}}}(t) = (1-(1-q(\varepsilon(n)t))^n)^n$. Substituting $q(\varepsilon(n)t) = x$, this is equivalent to $1-(1-x^n)^n \leq (1-(1-x)^n)^n$ for $0 \leq x \leq 1$. We will show this by showing $f(x) = (1-(1-x)^n)^n - 1 + (1-x^n)^n \geq 0$ for $0 \leq x \leq 1$. First note that $f(x) = f(1-x)$ is symmetric with respect to $x = 1/2$. Since $f(0) = 0$, it suffices to show that $f(x)$ is increasing when $0 \leq x \leq 1/2$, that is, $\mathrm{d}/\mathrm{d}x\,(f(x)) \geq 0$, whenever $0 \leq x \leq 1/2$. By computing the derivative, we have

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x} f(x) &= n^2\,(1-(1-x)^n)^{n-1}\,(1-x)^{n-1} - n^2(1-x^n)^{n-1}x^{n-1} \\ &= n^2\left((1-(1-x)^n)^{n-1}\,(1-x)^{n-1} - (1-x^n)^{n-1}x^{n-1}\right) \,.\end{aligned}$$

Since n^2 is positive, it suffices to confirm that the parenthesized expression is positive. By rearranging the terms, we get

$$\begin{aligned}&\left((1-(1-x)^n)^{n-1}\,(1-x)^{n-1} - (1-x^n)^{n-1}x^{n-1}\right) = \\ &\left((1-x)-(1-x)^{n+1}\right)^{n-1} - \left(x-x^{n+1}\right)^{n-1} \,.\end{aligned}$$

The expression above is positive, since the distance between $(1-x)$ and $(1-x)^{n+1}$ is greater than or equal to the distance between x and x^{n+1} whenever $0 \leq x \leq 1/2$, as can be verified by confirming that the claim holds for $n = 1$ and noting that $\mathrm{d}/\mathrm{d}n((1-x)-(1-x)^{n+1}) = -(1-x)^{n+1}\log(1-x) \geq \mathrm{d}/\mathrm{d}n(x-x^{n+1}) =$

$-x^{n+1}\log x$. The latter can be shown using induction by noting that $-(1-x)^n\log(1-x) \geq -x^n \log x$ for $n=2$, assuming that the claim holds for $n=k$ and noting that in this case also $-(1-x)^{k+1}\log(1-x) \geq -x^{k+1}\log x$, since $(1-x) \geq x$ when $0 \leq x \leq 1/2$.

By Proposition 5, the cumulative runtime distribution of safe-partitioning is less than that of repeated partitioning for all t and unsatisfiable instances. We now show that, when the number of resources is fixed to N, there are distributions of unsatisfiable instances for which the expected runtime of the safe partitioning approach is greater than the expected runtime of a single solver. From this it follows that whenever partitioning is performed, it is possible that the expected performance of the approach is worse than that of a single solver. An example is a two-step distribution where the probability of solving the instance exactly at time t_1 is p and the probability of solving the instance exactly at time t_2 is $(1-p)$. The expected runtime of a single solver for this type of instance is $\mathbb{E}T = pt_1 + (1-p)t_2$. The safe partitioning approach with void partitioning function has the expected runtime $\mathbb{E}T^N_{\text{safe}(0)} = \left(1-(1-p)^N\right)t_1 + \left(1-\left(1-(1-p)^N\right)^N\right)t_2$. For example, if $p=0.01$, $t_1 = 1$, $t_2 = 1{,}000{,}000$ and $N=2$, the expected runtime of the safe-partitioning approach is approximately 1% higher than the expected runtime of a single solver.

Conversely, if the formula to be solved is satisfiable, we have the following.

Proposition 6. $\mathbb{E}T_{\text{safe}} = \mathbb{E}T_{\text{rep}}$ *for satisfiable instances.*

Proof. If instead r of the k partitions are satisfiable, the expected runtime of safe partitioning becomes

$$D^k_{\text{part}}(D^k_{\text{portf}}(q(t))) = 1-(1-(1-(1-q(\varepsilon(k)t))^k))^r = 1-(1-q(\varepsilon(k)t))^{kr}\,,$$

and the expected runtime for repeated partitioning equally becomes

$$D^k_{\text{portf}}(D^k_{\text{part}}(q(t))) = 1-(1-(1-(1-q(\varepsilon(k)t))^r))^k = 1-(1-q(\varepsilon(k)t))^{rk}\,.$$

These types of step probability functions turn out to be interesting to compare the working of different partitioning approaches on extreme cases. In Figure 3.7 we show the behavior of some of the partitioning algorithms discussed in this chapter when the number of parallel computing elements is increased. The distribution used in the simulation is defined by

$$q_T(t) = \begin{cases} 0 & \text{for } 0 \leq t < 1, \\ 0.8 & \text{for } 1 \leq t < 1{,}000{,}000, \text{ and} \\ 1 & \text{for } t \geq 1{,}000{,}000. \end{cases} \tag{3.11}$$

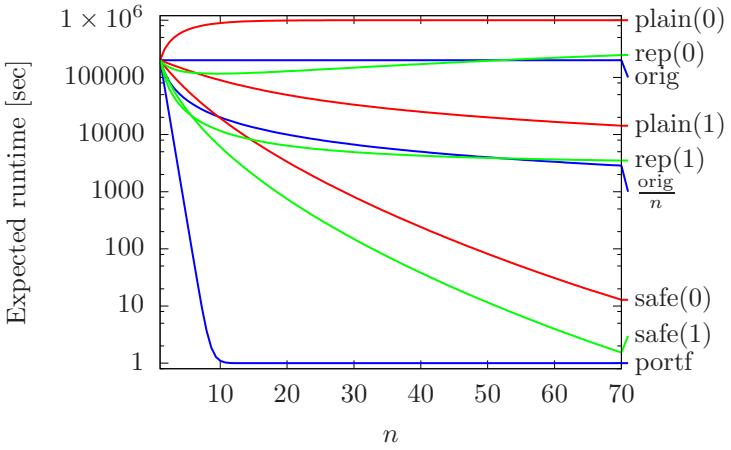

Fig. 3.7: The expected runtimes of different approaches on an (artificial) unsatisfiable instance having the distribution described in Equation (3.11).

3.6.4 Constructing Partitions

As seen from the preceding analytical discussion, the quality of the partitioning function is critical for performance, and, in case of plain, safe, and repeated partitioning, avoiding an increase in expected runtime is too. The partitioning functions considered here introduce new constraints, represented as clauses, to a formula. We consider two types of partitioning functions: the *DPLL-based partitioning* producing only unit clauses and the *scattering-based partitioning*, which produces longer clauses. Heuristics for constructing the constraints are used for increasing the likelihood of obtaining partitions that result in low runtime. All implementations of the partitioning functions are built on a CDCL SAT solver underlying the SMT solver.

The first partitioning function discussed here uses the unit propagation lookahead (see, e.g., [46]). The goal is to use as decision literals the literals that result in the highest number of unit propagations.

Computing the full lookahead for a formula ϕ is worst-case quadratic in the number of variables in ϕ. To guarantee scalability the implementations only study a subset of promising literals of ϕ and use several optimizations in the computation. The *lookahead DPLL* partitioning function implements such optimizations to produce evenly sized derived formulas and uses both theory and Boolean propagation to determine the heuristic value of the variables. Given a formula ϕ, promising literals l are studied by computing the number of literals in the unit propagation closure $\mathrm{UP}(\phi, l)$ and $\mathrm{UP}(\phi, \neg l)$. As the number of literals in $\mathrm{UP}(\phi, l)$ might differ dramatically compared to $\mathrm{UP}(\phi, \neg l)$, the implementation scores literals based on the minimum of these two numbers. Once a heuristically good literal has been selected, the corresponding two derived formulas $\phi \wedge \mathrm{UP}(\phi, l)$ and $\phi \wedge \mathrm{UP}(\phi, \neg l)$ are recursively

handled in a similar way. The binary tree up to the depth n constructed this way can be interpreted as consisting of 2^n derived formulas covering all potential satisfying truth assignments of ϕ, and the idea in DPLL-based partitioning is to return exactly these formulas as the derived formulas.

It is interesting to study partitioning functions producing more general constraints. The derived formulas in DPLL-based partitioning are of the form $\phi \wedge l_1 \wedge \ldots \wedge l_n$, but there is no need to limit partitioning functions to producing only constraints of unit clauses. Scattering-based partitioning produces both unit and longer clauses as the constraints. The idea is to first run the SMT solver for a fixed time to tune the heuristic of the solver. If the satisfiability of the formula is not determined in this time, the solver restarts, and starts to produce derived formulas. The first derived formula is produced by making the decisions $l_1^1, \ldots, l_{d_1}^1$, and producing the formula $\phi \wedge l_1^1 \wedge \ldots \wedge l_{d_1}^1$ as in DPLL-based partitioning. Then, instead of selecting the next branch of the search tree, the negation of the literals is added as a clause to ϕ. The solver restarts again, makes new decisions $l_1^2, \ldots, l_{d_2}^2$, and produces the formula $\phi \wedge (\neg l_1^1 \vee \ldots \vee \neg l_{d_1}^n) \wedge l_1^2 \wedge \ldots \wedge l_{d_2}^2$. The process is continued until a sufficient number of derived formulas are produced. The idea leads to a partitioning function producing the derived formula ϕ_i such that

$$\phi_i = \begin{cases} \phi \wedge (l_1^1) \wedge \ldots \wedge (l_{d_1}^1) & \text{if } i = 1, \\ \phi \wedge (\neg l_1^1 \vee \ldots \vee \neg l_{d_1}^1) \wedge \\ \quad \wedge \ldots \wedge (\neg l_1^{i-1} \vee \ldots \vee \neg l_{d_{i-1}}^{i-1}) \wedge \\ \quad (l_1^i) \wedge \ldots \wedge (l_{d_i}^i) & \text{if } 1 < i < n, \\ \phi \wedge (\neg l_1^1 \vee \ldots \vee \neg l_{d_1}^1) \wedge \ldots \wedge \\ \quad \wedge (\neg l_1^{n-1} \vee \ldots \vee \neg l_{d_{n-1}}^{n-1}) & \text{if } i = n. \end{cases} \tag{3.12}$$

Essentially the derived formulas consist of the original formula ϕ, a conjunction of unit clauses $(l_1) \wedge \ldots \wedge (l_d)$, and clauses representing negations of the previously selected unit clauses. In order for the derived formulas to be of roughly equal size, the number of new unit clauses, denoted by d_i, should not in general be the same in all derived formulas. The selection of the number d_i is motivated so that the expected runtime of each derived formula should be t/n, where t is the expected runtime of the original formula and n is the total number of derived instances produced by the partitioning function. Hence the goal fraction r_i of the runtime for the derived formula ϕ_i can be obtained from the equality

$$\frac{t}{n} = (t - (i-1)\frac{t}{n}) r_i \,,$$

where $(i-1)\frac{t}{n}$ is the runtime already contributed to the derived formulas $\phi_1, \ldots, \phi_{i-1}$. Solving the above for r_i results in

$$r_i = \frac{1}{n-i+1} \,. \tag{3.13}$$

Here, we assume that conjoining a literal with a formula halves the expected runtime of the formula, and therefore the number d_i is chosen to be the integer minimizing the difference

$$\Delta = |r_i - 2^{-d_i}| \,. \tag{3.14}$$

Example 5. Let ϕ be a propositional formula and P a partitioning function producing three partitions. From Equation (3.13), the first fraction of the search space should be $r_1 = 1/3$. The value $d_1 = 2$ minimizes Δ in Equation (3.14); the first derived formula becomes, by Equation (3.12), $\phi_1 = \phi \wedge (l_1^1) \wedge (l_2^1)$. Similarly, $r_2 = 1/2$ and the value $d_2 = 1$ minimizes Δ; the second derived formula becomes $\phi_2 = \phi \wedge (\neg l_1^1 \vee \neg l_2^1) \wedge (l_1^2)$. The final derived formula then becomes $\phi_3 = \phi \wedge (\neg l_1^1 \vee \neg l_2^1) \wedge (\neg l_1^2)$.

The approach for choosing values for d_i using the model in Equation (3.13) is not the only choice we have. The following example illustrates how the scattering approach can 'simulate' a DPLL-based partitioning.

Example 6. Let ϕ be a formula. Our target will be to build a partitioning function producing four derived formulas. Let the first derived formula be $\phi_1 = \phi \wedge (l_1) \wedge (l_2)$. Setting $d_2 = 1$ we may choose $\phi_2 = \phi \wedge (\neg l_1 \vee \neg l_2) \wedge (l_1)$ as the second derived formula. Since $\mathrm{UP}((\neg l_1 \vee \neg l_2) \wedge (l_1)) = \{l_1, \neg l_2\}$, the solving of ϕ_2 will proceed exactly as if the second derived formula had been $\phi_2 = \phi \wedge (l_1) \wedge (\neg l_2)$, corresponding to the DPLL-based partitioning. Similarly it is possible to choose $d_3 = 1$ in Equation (3.12) and $\phi_3 = \phi \wedge (\neg l_1 \vee \neg l_2) \wedge (\neg l_1) \wedge (l_3)$ resulting in the search corresponding to the formula $\phi \wedge \neg l_1 \wedge l_3$, derived from the DPLL-based-partitioning, and finally $\phi_4 = \phi \wedge (\neg l_1 \vee \neg l_2) \wedge (\neg l_1) \wedge (\neg l_3)$.

The approach presented in the above example generalizes to higher numbers of derived formulas. Let $S_n = (d_1, \ldots, d_n)$ denote the sequence producing n derived instances as in Example 6. Let $S_i = (d_1, \ldots, d_i)$ and $T_j = (e_1, \ldots, e_j)$ be two such sequences. We denote by $S_n + 1$ the sequence $(d_1 + 1, \ldots, d_n + 1)$ and by $(S_i) \cdot (T_j)$ the concatenation of the two sequences $(d_1, \ldots, d_i, e_1, \ldots, e_j)$. The scattering-based partitioning function simulates the DPLL-based partitioning function producing $n = 2^k, k \geq 0$ derived instances by using a fixed variable ordering and the sequence S_n defined recursively as $S_1 = S_{k^0} = (0)$ and $S_{2^k} = (S_{2^{k-1}} + 1) \cdot (S_{2^{k-1}})$.

3.7 Further topics

Theory Solver Parallelization: Two core algorithms of an SMT theory solver are the congruence closure algorithm based on the *E-graph* data structure [26] and an incremental implementation of the Simplex algorithm [29]. For most non-incremental SMT problems most of the runtime of an SMT solver is spent on the theory solvers. This presents an interesting challenge for parallelization of the theory solvers, since while the total time spent in theories is high, each individual call to the solver is

usually very short. However, the performance of a theory solver parallelization is of course highly dependent on the type and behavior of the theory that it decides. While this type of parallelization is often used for very specific problems in many areas, their application in SMT with multiple theories being combined is not very common yet. However, this area is just starting to be explored, for instance by Hadarean et al. [42] who use lazy and eager bit-vector solvers in parallel.

Parallelization of Incremental SMT: Software and hardware verification using symbolic model-checking is undeniably the most important application driving the development of SMT solvers today. Many successful symbolic model-checking approaches reduce the verification problem to repeated, related queries to an SMT solver [14]. This frequently results in a very large set of (relatively) simple queries. Thus, the problem of parallelization across multiple, but related, problems is of very high importance as well.

References

1. Abdulla, P.A., Bjesse, P., Eén, N.: Symbolic reachability analysis based on SAT-solvers. In: S. Graf, M.I. Schwartzbach (eds.) Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings, *Lecture Notes in Computer Science*, vol. 1785, pp. 411–425. Springer (2000)
2. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult SAT instances in the presence of symmetry. In: Proceedings of the 39th Design Automation Conference, DAC 2002, New Orleans, LA, USA, June 10-14, 2002, pp. 731–736. ACM (2002)
3. Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: A proof-sensitive approach for small propositional interpolants. In: A. Gurfinkel, S.A. Seshia (eds.) Verified Software: Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 9593, pp. 1–18. Springer (2015)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: G. Gopalakrishnan, S. Qadeer (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6806, pp. 171–177. Springer (2011)
5. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
6. Benque, D., Bourton, S., Cockerton, C., Cook, B., Fisher, J., Ishtiaq, S., Piterman, N., Taylor, A.S., Vardi, M.Y.: BMA: Visual tool for modeling and analyzing biological networks. In: P. Madhusudan, S.A. Seshia (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, *Lecture Notes in Computer Science*, vol. 7358, pp. 686–692. Springer (2012)
7. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University (2010)
8. Biere, A., Bloem, R. (eds.): Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8559. Springer (2014)
9. Bonacina, M.P.: A taxonomy of parallel strategies for deduction. Annals of Mathematics and Artificial Intelligence **29**(1–4), 223–257 (2000)

10. Bonacina, M.P.: Combination of distributed search and multi-search in Peers-mcd.d. In: Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001), *Lecture Notes in Artificial Intelligence*, vol. 2083, pp. 448–452. Springer (2001)
11. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 443–448 (2009)
12. Bouton, T., Oliveira, D.C.B.D., Déharbe, D., Fontaine, P.: verit: An open, trustable and efficient smt-solver. In: R.A. Schmidt (ed.) Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5663, pp. 151–156. Springer (2009)
13. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., Ranise, S., van Rossum, P., Sebastiani, R.: Efficient satisfiability modulo theories via delayed theory combination. In: K. Etessami, S.K. Rajamani (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings, *Lecture Notes in Computer Science*, vol. 3576, pp. 335–349. Springer (2005)
14. Bradley, A.R.: SAT-based model checking without unrolling. In: R. Jhala, D.A. Schmidt (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6538, pp. 70–87. Springer (2011)
15. Brayton, R.K., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: T. Touili, B. Cook, P.B. Jackson (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6174, pp. 24–40. Springer (2010)
16. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: S. Kowalewski, A. Philippou (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5505, pp. 174–177. Springer (2009)
17. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: a comparative analysis. Ann. Math. Artif. Intell. **55**(1-2), 63–99 (2009)
18. Christ, J., Hoenicke, J., Nutz, A.: Smtinterpol: An interpolating SMT solver. In: A.F. Donaldson, D. Parker (eds.) Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7385, pp. 248–254. Springer (2012)
19. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: N. Piterman, S.A. Smolka (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7795, pp. 93–107. Springer (2013)
20. Cimatti, A., Sebastiani, R. (eds.): Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7317. Springer (2012)
21. Conchon, S., Déharbe, D., Heizmann, M., Weber., T.: 11th International Satisfiability Modulo Theories Competition (SMT-COMP 2016). `http://smtcomp.sourceforge.net/2016/` (2016)
22. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Heule and Weaver [47], pp. 360–368
23. Craig, W.: Linear reasoning. a new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957)
24. Creignou, N., Berre, D.L. (eds.): Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9710. Springer (2016)

25. Dequen, G., Vander-Swalmen, P., Krajecki, M.: Toward easy parallel SAT solving. In: Proceedings of the 21st IEEE International Concerence on Tools with Artificial Intelligence (ICTAI 2009), pp. 425–432. IEEE Press (2009)
26. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM **52**(3), 365–473 (2005)
27. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: G. Barthe, M.V. Hermenegildo (eds.) Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 5944, pp. 129–145. Springer (2010)
28. Dutertre, B.: Yices 2.2. In: Biere and Bloem [8], pp. 737–744
29. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: T. Ball, R.B. Jones (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, *Lecture Notes in Computer Science*, vol. 4144, pp. 81–94. Springer (2006)
30. Eén, N., Sörensson, N.: An extensible SAT-solver. In: E. Giunchiglia, A. Tacchella (eds.) Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers, *Lecture Notes in Computer Science*, vol. 2919, pp. 502–518. Springer (2003)
31. Ermon, S., LeBras, R., Gomes, C.P., Selman, B., van Dover, R.B.: SMT-aided combinatorial materials discovery. In: Cimatti and Sebastiani [20], pp. 172–185
32. Fagerberg, R., Flamm, C., Merkle, D., Peters, P.: Exploring chemistry using SMT. In: M. Milano (ed.) Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7514, pp. 900–915. Springer (2012)
33. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schnor, B.: Cluster-based ASP solving with Claspar. In: Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), *Lecture Notes in Computer Science*, vol. 6645, pp. 364–369. Springer (2011)
34. Ghilardi, S., Ranise, S.: MCMT: A model checker modulo theories. In: J. Giesl, R. Hähnle (eds.) Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6173, pp. 22–29. Springer (2010)
35. Giesl, J., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Proving termination of programs automatically with aprove. In: S. Demri, D. Kapur, C. Weidenbach (eds.) Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings, *Lecture Notes in Computer Science*, vol. 8562, pp. 184–191. Springer (2014)
36. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1–2), 43–62 (2001)
37. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24**(1/2), 67–100 (2000)
38. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998), pp. 431–437. AAAI Press (1998)
39. Grama, A., Kumar, V.: State of the art in parallel search techniques for discrete optimization problems. IEEE Transactions on Knowledge and Data Engineering **11**(1), 28–34 (1999)
40. Gregory, P., Long, D., Fox, M., Beck, J.C.: Planning modulo theories: Extending the planning paradigm. In: L. McCluskey, B.C. Williams, J.R. Silva, B. Bonet (eds.) Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012. AAAI (2012)
41. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and intensification in parallel SAT solving. In: 16th International Conference on Principles and Practice of Constraint Programming (CP 2010), *Lecture Notes in Computer Science*, vol. 6308, pp. 252 – 265. Springer (2010)

42. Hadarean, L., Bansal, K., Jovanovic, D., Barrett, C., Tinelli, C.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: Biere and Bloem [8], pp. 680–695
43. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel SAT solving. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 499–504 (2009)
44. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **6**(4), 245 – 262 (2009)
45. Hamadi, Y., Marques-Silva, J., Wintersteiger, C.M.: Lazy decomposition for distributed decision procedures. In: J. Barnat, K. Heljanko (eds.) Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011., *EPTCS*, vol. 72, pp. 43–54 (2011)
46. Heule, M., van Maaren, H.: Look-ahead based SAT solvers. In: A. Biere, M. Heule, H. van Maaren, T. Walsh (eds.) Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 155–184. IOS Press (2009)
47. Heule, M., Weaver, S. (eds.): Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9340. Springer (2015)
48. Huang, G.: Constructing craig interpolation formulas. In: D. Du, M. Li (eds.) Computing and Combinatorics, First Annual International Conference, COCOON '95, Xi'an, China, August 24-26, 1995, Proceedings, *Lecture Notes in Computer Science*, vol. 959, pp. 181–190. Springer (1995)
49. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science **275**(5296), 51–54 (1997)
50. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006), *Lecture Notes in Computer Science*, vol. 4121, pp. 430–435. Springer (2006)
51. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. Fundam. Inform. **107**(2-3), 289–311 (2011)
52. Hyvärinen, A.E.J., Manthey, N.: Designing scalable parallel SAT solvers. In: Cimatti and Sebastiani [20], pp. 214–227
53. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou and Berre [24], pp. 547–553
54. Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: Search-space partitioning for parallelizing SMT solvers. In: Heule and Weaver [47], pp. 369–386
55. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154**(16), 2291–2306 (2006)
56. Janakiram, V.K., Agrawal, D.P., Mehrotra, R.: A randomized parallel backtracking algorithm. IEEE Transactions on Computers **37**(12), 1665–1676 (1988)
57. Janakiram, V.K., Gehringer, E.F., Agrawal, D.P., Mehrotra, R.: A randomized parallel branch-and-bound algorithm. International Journal of Parallel Programming **17**(3), 277 – 301 (1988)
58. Jonás, M., Strejcek, J.: Solving quantified bit-vector formulas using binary decision diagrams. In: Creignou and Berre [24], pp. 267–283
59. Kalinnik, N., Ábrahám, E., Schubert, T., Wimmer, R., Becker, B.: Exploiting different strategies for the parallelization of an SMT solver. In: M. Dietrich (ed.) Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Dresden, Germany, February 22-24, 2010, pp. 97–106. Fraunhofer Verlag (2010)
60. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 8044, pp. 1–35. Springer (2013)
61. Krajícek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symb. Log. **62**(2), 457–486 (1997)

62. Krings, S., Bendisposto, J., Leuschel, M.: From failure to proof: The prob disprover for B and event-b. In: R. Calinescu, B. Rumpe (eds.) Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings, *Lecture Notes in Computer Science*, vol. 9276, pp. 199–214. Springer (2015)
63. Li, G.J., Wah, B.W.: Computational efficiency of parallel combinatorial OR-Tree searches. IEEE Transactions on Software Engineering **16**(1), 13–31 (1990)
64. Luby, M., Ertel, W.: Optimal parallelization of Las Vegas algorithms. In: Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1994), *Lecture Notes in Computer Science*, vol. 775, pp. 463–474. Springer (1994)
65. Marescotti, M., Hyvärinen, A.E.J., Sharygina, N.: Clause sharing and partitioning for cloud-based SMT solving. In: C. Artho, A. Legay, D. Peled (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9938, pp. 428–443 (2016)
66. McMillan, K.L.: Interpolation and SAT-based model checking. In: W.A.H. Jr., F. Somenzi (eds.) Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings, *Lecture Notes in Computer Science*, vol. 2725, pp. 1–13. Springer (2003)
67. de Moura, L.M., Bjørner, N.: Model-based theory combination. Electr. Notes Theor. Comput. Sci. **198**(2), 37–49 (2008)
68. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
69. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. **1**(2), 245–257 (1979)
70. Ohmura, K., Ueda, K.: c-sat: A parallel SAT solver for clusters. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009), *Lecture Notes in Computer Science*, vol. 5584, pp. 524–537. Springer (2009)
71. Petrik, M., Zilberstein, S.: Learning parallel portfolios of algorithms. Annals of Mathematics and Artificial Intelligence **48**(1-2), 85–106 (2006)
72. Prestwich, S., Mudambi, S.: Improved branch and bound in constraint logic programming. In: Proceedings of the 1st International Conference on Principles and Practie of Constraint Programming (CP 1995), *Lecture Notes in Computer Science*, vol. 976, pp. 534–548. Springer (1995)
73. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. **62**(3), 981–998 (1997)
74. Reisenberger, C.: PBoolector: a parallel SMT solver for QF_BV by combining bit-blasting with look-ahead. Master's thesis, Johannes Kepler Univesität Linz, Linz, Austria (2014)
75. Rice, J.R.: The algorithm selection problem. Advances in Computers **15**, 65 – 118 (1976)
76. de Salvo Braz, R., O'Reilly, C., Gogate, V., Dechter, R.: Probabilistic inference modulo theories. In: S. Kambhampati (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016, pp. 3591–3599. IJCAI/AAAI Press (2016)
77. Sebastiani, R., Trentin, P.: OptiMathSAT: A tool for optimization modulo theories. In: D. Kroening, C.S. Pasareanu (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 9206, pp. 447–454. Springer (2015)
78. Segre, A.M., Forman, S.L., Resta, G., Wildenberg, A.: Nagging: A scalable fault-tolerant paradigm for distributed search. Artificial Intelligence **140**(1/2), 71–106 (2002)
79. Speckenmeyer, E., Monien, B., Vornberger, O.: Superlinear speedup for parallel backtracking. In: Proceedings of the 1st international conference on Supercomputing (SC 1987), *Lecture Notes in Computer Science*, vol. 297, pp. 985–993. Springer (1988)

80. Tinelli, C., Zarba, C.G.: Combining nonstably infinite theories. J. Autom. Reasoning **34**(3), 209–238 (2005)
81. Tung, V.X., Khanh, T.V., Ogawa, M.: raSAT: An SMT solver for polynomial constraints. In: N. Olivetti, A. Tiwari (eds.) Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 9706, pp. 228–237. Springer (2016)
82. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 1173–1178. Morgan Kaufmann (2003)
83. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. In: A. Bouajjani, O. Maler (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5643, pp. 715–720. Springer (2009)
84. Yordanov, B., Wintersteiger, C.M., Hamadi, Y., Kugler, H.: SMT-based analysis of biological computation. In: G. Brat, N. Rungta, A. Venet (eds.) NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7871, pp. 78–92. Springer (2013)