Visualizing SMT-Based Parallel Constraint Solving

Jelena Budakovic, Matteo Marescotti,
Antti E. J. Hyvärinen, and Natasha Sharygina

Università della Svizzera italiana, Switzerland

Abstract

Problem instances arising from a multitude of applications can be naturally reduced to the search for a solution satisfying a set of constraints. Characteristically such applications, that include model checking and the satisfiability of propositional formulas (SAT) and SAT modulo theories, are notorious for the computational complexity of the underlying decision problem. A central approach for tackling the hardness of these problems is through the use of parallel computing, with methods such as divide-and-conquer, algorithm portfolios, and their combinations. However, such algorithms are often complicated and it is difficult to assess whether their executions have anomalies. This paper presents a user-friendly interface to analyze the partitioning tree parallelization approach that generalizes both divide-and-conquer and algorithm portfolios. We use the interface to analyze parallel executions of both the IC3/PDR algorithm and an SMT solver, demonstrating its usefulness in visualizing the otherwise complicated executions of diverse algorithms in a parallel environment. Based on the initial results we believe that the visualization of these executions will further encourage the adoption of parallel computing techniques in these domains.

1 Introduction

Representing practical computational questions as instances of decision problems over a set of constraints is increasingly important as solving methodologies such as Satisfiability Modulo Theories (SMT) and propositional satisfiability (SAT) are being adopted by domain specialists in particular in model checking and verification. Problems emerging from these application domains can be extremely challenging and therefore there is a constant pressure for developing techniques for applying parallel computing to automatically speed up their solving. Parallel constraint solving is, however, often a very complicated process, and in particular identifying and locating problems related to correctness and performance can be an overwhelming task.

Different approaches for constraint solving share two basic features that are useful in obtaining speed-up in parallel computing. First, solving approaches perform often vastly differently in seemingly similar instances, motivating the use of algorithm portfolios; and second, the structure of the instances allows an automatic and often efficient divide-and-conquer approach to be used. In our experience visualizing these features can help in understanding parallel executions of a wide range of algorithms. This paper presents a web-service-based visualization interface for analyzing the parallel executions of constraint-solving algorithms that base on algorithm portfolios and divide-and-conquer. The goal of the tool is to provide a uniform, clear understanding of the execution of an algorithm either on-line while the algorithm is running, or off-line once the algorithm has already terminated. As the underlying abstraction of the parallel algorithms we use the rich framework of algorithms provided by *parallelization trees* [7] supporting arbitrary combinations of divide-and-conquer and algorithm portfolios.

The interface assumes that the execution of a parallelization tree algorithm is stored in a database, and is therefore made independent of the details of the algorithm. The information is used for visualizing a snapshot of the parallelization tree in a given time as a schematic picture,

together with descriptions of the running processes. In particular the database describes how the divide-and-conquer approach is being used to partition a given instance, and what kind of portfolios are being run on each of the instance or to construct the partitions.

We use the interface to analyze executions of a parallel implementation of the IC3/PDR algorithm for model-checking safety properties [2] as well as executions of a general-purpose SMT solver OpenSMT2 [7]. Based on our experience in applying the tool to both SMT solving and model checking with the IC3/PDR algorithm, we feel confident that the tool is versatile and the feedback that it provides from an execution can be used to help understanding and locating performance problems in parallel algorithms based on parallelization trees.

This paper provides an overview of the problem for both SMT and PDR solving, describes our solution to the visualization of the executions, and gives a description of the API we use for communicating with the visualizer. In addition we provide a short tutorial to the visualizer, and give some implementation-level details.

Related work. Some work exists in visualizing the structure of constraint problems and the executions of related, sequential search algorithms for propositional satisfiability [11] and ASP [9]. In both of these approaches the information is visualized by rendering a graph that shows the relationship of variables and constraints where they appear, thereby providing insights into the structure of an instances. We believe that our approach is the first made for visualizing the execution of parallel algorithms and integrates the time component into the system.

A verification oriented web service allowing users to browse runs of a large number of verification instances is presented in [1]. We follow the same ease of use approach by providing a web service, but, rather than visualizing the results of verification runs, we visualize the execution of one verification or constraint solving run.

Visualization of parallel executions of constraint logic programs is studied in [3]. Similar to us, the tool allows inspecting the execution at different time points and provides a tree view of the execution. However, the tool is not directly applicable to our setting which uses the parallelization tree approach.

2 Background

Due to the difficulty of the problems underlying many constraint problems, such as propositional satisfiability, satisfiability modulo theories, or symbolic model checking, the constraint solvers use heuristics to guide their search for a solution. Often these heuristics are very fragile in the sense that small changes in the heuristic can result in significant and often unpredictable differences in run times.

Algorithm portfolios take advantage of this seemingly random behaviour by executing in parallel several solvers, each initialized with a different heuristic. This approach naïvely increases search-space covering having each execution taking different path, therefore increasing the probability of guessing a good heuristic. However, in particular for unsatisfiable problems there often exists a theoretical or experimental minimum number of steps that the algorithm needs to take to prove the absence of models, and the use of a portfolio cannot therefore help in over-passing such a limitation. One way of to overcome the limitation is by applying a divide-and-conquer approach where the problem is partitioned into several independently solvable sub-problems, and the solution to the original problem can be obtained by solving the sub-problems. However, as a result of the partitioning, the solver now has to solve several problem instances instead of a single one, and the efficiency of the approach depends on the ability to construct sub-problems that are easier than the original problem.

Algorithm Portfolios and Divide-and-Conquer for SMT and PDR. In this work we study two different approaches for solving constraint problems.

The *Satisfiability Modulo Theories* (SMT) solvers accept as input formulas in propositional logic, where some of the Boolean variables are interpreted as equalities in first-order theories. The task of the solver is then to find an assignment to the propositional variables so that the corresponding equalities and inequalities are consistent in the first-order theories, or prove the absence of such assignments.

The *Property-Driven Reachability* (IC3/PDR) solvers take as input a transition system and a safety property, describing respectively a program and a correctness condition. The task of the solver is then to find for the transition system an inductive invariant that is valid with respect to the safety property. This is done by maintaining a sequence of formulas $F_1, \ldots, F_n$ over-approximating the states reachable by the transition system from the initial state in i steps, and either finding a concrete execution that ends in a state breaking the safety property, or finding two consecutive formulas F_i, F_{i+1} that are equal.

The former acts as a counterexample for the correctness of the program and the latter as the safe inductive invariant. For simplicity of the discussion we call the input formula of a PDR solver *satisfiable* if the program is incorrect and *unsatisfiable* if the program is correct.

Both in SMT and PDR we can apply algorithm portfolios and divide-and-conquer. In SMT we implement the portfolios by randomization, that is, by allowing the branching function used during the search to occasionally make random choices that go against the heuristic. In PDR we use, in addition to randomization, different methods for constructing the over-approximations F_i.

Our earlier work in applying divide-and-conquer in SAT [4] and SMT [7] is based on partitioning the original instance into sub-problems that share no model and, when disjointed, the resulting formula is equisatisfiable with the original instance. A similar idea can be applied in PDR, by applying the transition function backwards from unsafe states, and asking the PDR solvers to find solutions to such states.

Parallelization Trees. Both algorithm portfolios and divide-and-conquer have their downsides and interestingly one seems to perform well exactly when the other performs badly [5]. Therefore it is natural to ask if there is an approach that can performs well in both cases. The *Parallelization Trees* are a concept used to formalize the combination of portfolio and partitioning. The parallelization trees were initially introduced in [7], but for self-containedness we proved here a short description.

The idea is to describe in a uniform way algorithms where an instance can not only be at the same time solved by several solvers as in algorithm portfolios, and partitioned into several sub-problems as in divide-and-conquer, but also partitioned simultaneously in different ways to sub-problems.

A parallelization tree contains two types of nodes: *and-nodes* and *or-nodes*, alternating at each level. The root of a parallelization tree is an and-node representing the input instance. Each and-node is associated with a constraint problem and one or more constraint solvers. The input instance is satisfiable if the at least one of the instances in the and-nodes is shown satisfiable. A subtree rooted at an and-node (including the full tree rooted at the input instance) is unsatisfiable if one of its children is unsatisfiable or at least one of the constraint solvers working on the instance on the and-node has shown the instance unsatisfiable. Finally, a tree rooted at an or-node is unsatisfiable if every tree rooted at its children is unsatisfiable.

We refer the reader to [10, 7] for further detail about the parallelization tree.

The SMTService Framework. SMTService is a framework and tool for parallel and distributed constraint solving available at https://scm.ti-edu.ch/projects/smts. While the visualization tool discussed in his paper is independent from SMTService, we provide here a short description to help understanding the further discussion.

SMTService is used in [10] for distributed SMT solving based on the SMT solver OpenSMT2 [6], and we have recent updated the system to also support parallel execution of the PDR algorithm using the model checker Spacer [8]. SMTService supports the parallelization tree approach for SMT and to some extent PDR, and in addition supports clause sharing for SMT and lemma sharing (i.e., sharing the formulas F_i) for PDR. The SMTService framework is meant to be easily integrable with different solvers in order to make the study of parallel techniques more affordable.

Figure 1 offers an overview of SMTService. The *solving server* acts as the central point of contact for the visualization tool. It contains the *control socket*, which is the default interface for interacting with the solving server. In the current implementation a user may provide the solving server with instances through the terminal access to the control socket. In addition an external tool requesting the solving of constraint problems may connect to the solving server for this purpose through the API provided by the control socket. The parallelization algorithm is provided to the solving server as a partitioning tree described in the configuration.

The *solvers* connect directly to the solving server using a persistent TCP/IP connection. Solvers' failures and connections of new solvers are handled in a gracefully way by the solving server. The solving server asks each solver to solve a partition following the partitioning tree provided in the configuration file. The *scheduler* keeps track of and arranges all solvers' commitments based on the parallelization tree. In addition the solvers may share lemmas using a push and pull mechanism provided by the *lemma database*.

The process is completely transparent to the solvers, making the system more easy to adapt for other solver implementations. SMTService provides a set of APIs, and new solvers can be supported with little effort by implementing these APIs. This has the added benefit that different solvers can be dynamically added during solving in a transparent way.

Finally, the *events database* (Events DB) acts as a bridge between SMTService and SMTViewer. This database is constantly updated during execution by the Solving Server with information regarding the solving task performed by the solvers in the cluster. The information stored in the database are retrieved by SMTViewer in order to provide a visual analysis of the execution to the solver.

3 SMTViewer Architecture

In this section we present SMTViewer, a graphical user interface for analyzing executions of parallel algorithms based on the parallelization tree. An overview of how SMTViewer interacts with SMTService is given in Figure 1. We identify the entities involved in this a architecture as the user, the client web app and the web server. The graphical user interface offered by the client web app aims at helping the user to analyze a parallel solving execution done either with SMTService or any other tool.

Due to the implementation of the communication through a database containing the events, the user has the possibility to analyze the parallel solving of SMTService either in real time or over past executions logs.

Real time analysis mode enables a live interaction with the internal solving server's components by interfacing with the control socket. Analyzing past executions may be very helpful for debugging and identifying performance bottlenecks. For this reason, the user is able to

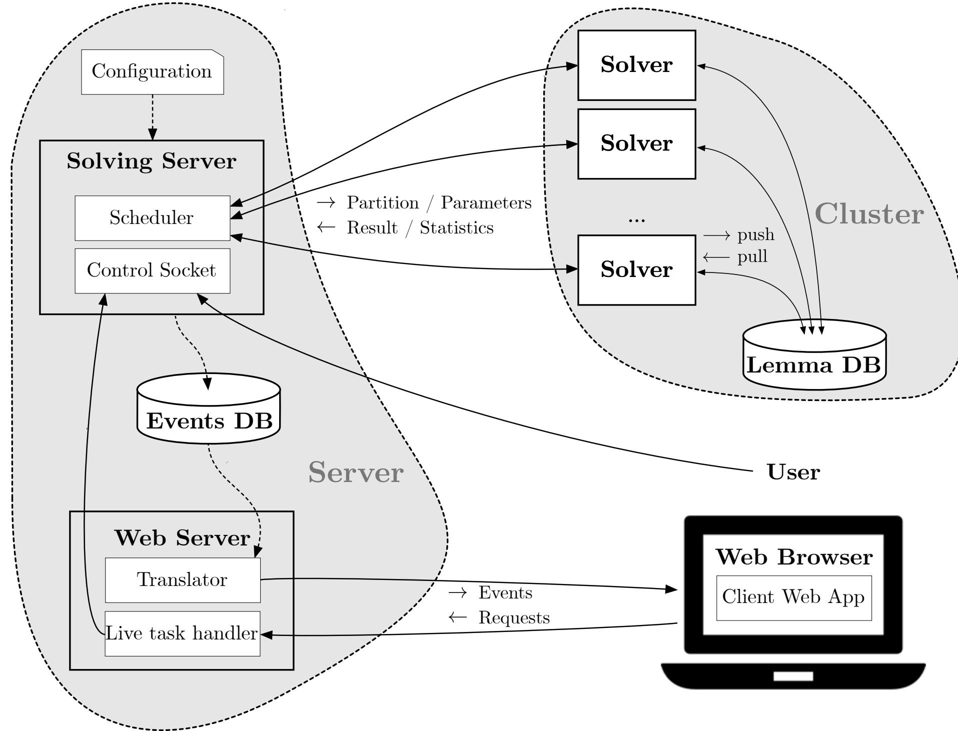

Figure 1: SMTService framework overview. SMTViewer is represented by the web server retrieving solving events and the client web app displaying them. Solid lines represent connections over TCP/IP, while dashed lines represent disk I/O.

construct the tree representing each problem as it was at any given moment in the past. The analysis of the past executions allows visualizing parallel solving information retrieved from any other tool. This feature is exploitable by providing SMTViewer with a suitable events database containing the parallel solving information gathered from any external tool execution. The web server is responsible for retrieving and translating the events, while the client web app provides the graphical interface for such events to the user.

In the following we provide details of SMTViewer components. In particular Section 3.1 focuses on SMTViewer API, Sections 4 and 4.1 give details about the web server and the client web app respectively, while Section 5 provides implementation details.

3.1 The SMTViewer API

This section presents in brief the API of the visualization component, serving as a guide for implementing the analysis of parallelization-tree-based algorithms.

The input of SMTViewer is a path to an SQLite3 database containing solving events and acting as an API for SMTViewer. The database must contain a table called SolvingHistory, where each record represents an event involving a solver and a node of the parallelization tree of the instance being solved.

Table 1 shows the required format. Each row of the table represents an event described by a unique identifier, its Unix timestamp, the solver working on a node of parallelization tree of

Table 1: Format of the table SolvingHistory.

Column	Type	Description
id	Integer	Unique event id number.
ts	Integer	The Unix timestamp of the event.
name	String	Instance name.
node	JsonArray<Integer>	Path from the root to the node.
event	String	Type of the event.
solver	String	Solver identifier.
data	JsonObject<String, String>	Data associated with the event.

the given instance, the event type, and finally a generic data field. The data field has different meaning accordingly to the event type and the solver. A list of all event types with the expected content of the data field is given below.

- `+` : `solver` is assigned to work on `node`
- `-` : `solver` working on `node` is requested to stop.
- `OR` : an OR node is created. The `data` field contains partitioning heuristic details.
- `AND` : an AND node (i.e. a sub-problem) is created. The `data` field represents the constraints of this partition and must contain a `node` entry with the path of the new node.
- `STATUS` : the solver proved the satisfiability or unsatisfiability of the `node` it was working on. The `data` field contains all the statistics of the entire solving task done by the `solver`.
- `SOLVED` : the satisfiability of the root was determined, and therefore the instance `name` is solved. The `data` field provides statistics about the entire parallel solving.

4 The Tool Usage

This section serves as a usage manual for the visualization tool. When the user analyses a past execution the client web app asks the web server for the content of the database and arranges its views according to the user request. Instead, when SMTViewer is configured for real-time analysis, the client web app constantly asks the web server for new events available in the database and re-arranges its views accordingly. In this way the graphical user interface is always updated providing a real-time overview of the solving tasks going on inside the cluster.

The terminal access to the control socket provides a wide range of functionalities for interfacing with the internal behaviour of the solving server, involved only in real-time analysis. The client web app provides a restricted set of the functionalities offered by the control socket. Currently three kinds of requests are supported: uploading a new instance for the solving server to be solved; setting a different timeout for a solver; and stopping the solving of the current instance. Once the user sends one of these requests through the client web app, the *live task handler* is responsible for creating the proper request to the control socket and thus for interacting with the solving server.

The *translator* component of the web server is responsible for translating database events to the form needed by the client web app. This modular approach prevents a propagation of code maintenance otherwise needed in case the events database format is changed.

In the case of real time analysis, the events database is periodically checked for changes on a predefined time interval. In this way, a user monitoring the execution in real-time using the client web app is immediately updated with new informations.

4.1 Client Web App

All the necessary information required to build the graphical interface is gathered by the web server and sent to the client web app. The mechanisms involved in the construction of the tree view and the management of the data are done at the client side by the web browser while executing the client web app.

The data received from the web server is displayed throughout six different views. In this way the user can interactively analyze the execution of SMTService, retrieving statistics, parameters and past events according to his needs. We provide a brief explanation for each interface view. Figure 2 shows SMT Viewer client web app with its six views.

Instance view. A list of all problems belonging to the current execution is shown here. The user may select a particular instance from this view for the analysis.

Events view. The view is composed of two interactive components: a table showing all the events related to the selected instance, and a time-line displaying how this events are distributed over time. When the user selects an event or a point in the time-line, all the other views are updated accordingly to the ongoing status at that time. This allows the user to "rewind" the solving execution and better analyze every event involved during solving. Browsing the past is the typical way for finding performance problems and other anomalies, and we found the interactiveness to be useful in this task.

Tree view. The parallelization tree is represented here as an interactive tree. Each node is associated with an integer indicating the number of solvers working on the related instance. Clicking on the node results in all the node-specific information being updated in data view. The colour of the nodes changes accordingly to the type (and- or or-nodes) and status (satisfiable, unsatisfiable or unknown)

Solvers view. This view shows each solver together with its assigned node at the time of the selected event in the events view. The user can then analyze how many solvers were present and to which node they were assigned at each phase of the solving process. When a node in the tree is selected, the solvers working on it at that time are highlighted in this view in order to be easily discovered.

Data view. This view is for visualizing the data field related to the selected event. All the data related to the tree nodes, the solvers, and the events are displayed here when each of them is clicked.

Control view. This view changes accordingly to the selected mode. On real time analysis mode, the view allows the live interaction with the solving server through the SMTService control socket. We examining past executions, the user can upload different databases to be analyzed.

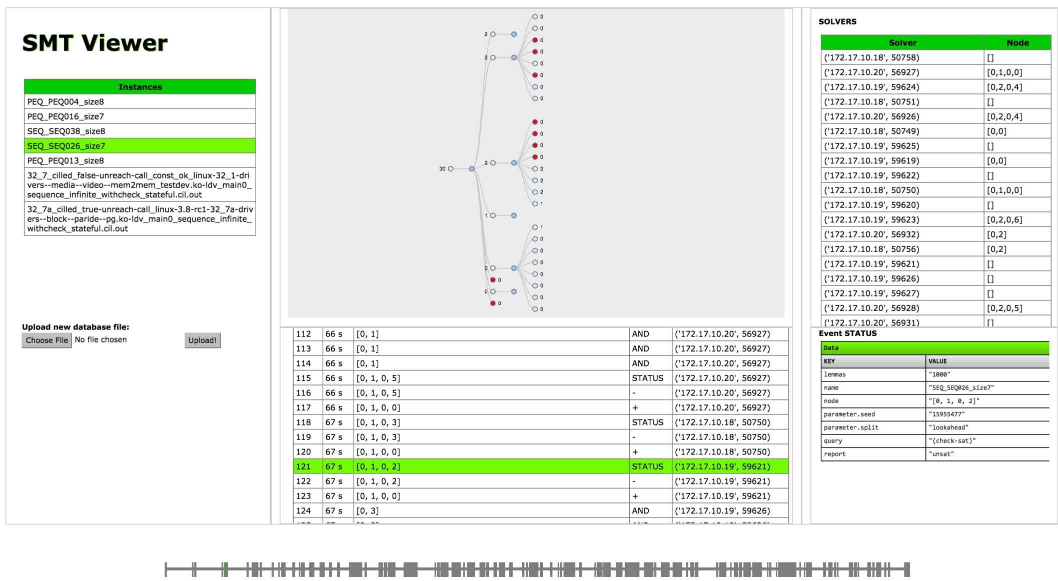

Figure 2: Client web app overview.

5 Implementation Details

In order to have an easily maintainable and modern style implementation we use several recent frameworks and tools available for web development, in addition to some very well established techniques.

The web server is built using NodeJS and Express frameworks. The SQLite3 module for NodeJS is used for opening and reading the events database. The web server translator uses pure Javascript for JSON event formatting. The live task handler uses a TCP socket to communicate with the SMTService control socket.

The web browser is required to support HTML5, Javascript and CSS3. We use the frameworks AngularJS for user events handling and W3.CSS for page layout.

The Typescript language is used for the entire class hierarchy necessary for event handling and arrangement. The graphic library D3 is responsible for visualizing the events in a tree view. The time-line present in event view is constructed using HTML5 features.

6 Conclusion

The increasing popularity of constraint solvers ensures a steady flow of increasingly complex problem instances for such solvers. This phenomenon is visible in particular in domains such as SMT solving and model checking. Although parallelization is known to speed up the solving

of such problems, many researchers remain skeptical of the technique due to its complicated nature and difficulties in implementation.

The tool presented in this paper aims at simplifying the study of parallel techniques by providing a user-friendly graphical interface. In particular we focus on helping users to understand and analyze the partitioning tree parallelization approach. The user is able to analyze the solving process by retrieving all the necessary information at any time of an execution. We believe that this new service will help users to be more confident with parallel computing techniques, encouraging the community to develop new efficient decision procedures for constraint solving.

Acknowledgements. This work is supported by the Swiss National Science Foundation (SNSF) grants 153402 and 166288.

References

[1] D. Beyer and M. Dangl. Verification-aided debugging: An interactive web-service for exploring error witnesses. In *Proc. CAV 2016*, volume 9780 of *LNCS*, pages 502–509. Springer, 2016.

[2] A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87, 2011.

[3] M. Carro and M. V. Hermenegildo. Tools for search-tree visualisation: The APT tool. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, volume 1870 of *LNCS*, pages 237–252. Springer, 2000.

[4] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. A distribution method for solving SAT in grids. In *Proc. SAT 2006*, volume 4121 of *LNCS*, pages 430–435. Springer, 2006.

[5] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning search spaces of a randomized search. *Fundamenta Informaticae*, 107(2-3):289–311, 2011.

[6] A. E. J. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In *Proc. SAT 2016*, number 9710 in LNCS, pages 547 – 553. Springer, 2016.

[7] A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina. Search-space partitioning for parallelizing SMT solvers. In *Proc. SAT 2015*, volume 9340 of *LNCS*, pages 369–386. Springer, 2015.

[8] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[9] A. König and T. Schaub. Monitoring and visualizing answer set solving. *TPLP*, 13(4-5-Online-Supplement), 2013.

[10] M. Marescotti, A. E. J. Hyvärinen, and N. Sharygina. Clause sharing and partitioning for cloud-based SMT solving. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 428–443, 2016.

[11] C. Sinz. Visualizing SAT instances and runs of the DPLL algorithm. *J. Autom. Reasoning*, 39(2):219–243, 2007.