Scalable SAT Solving and its Application

Zur Erlangung des akademischen Grades eines

Dr. rer. nat.

von der KIT-Fakultät für Informatik des

Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Dominik Pascal Schreiber

M.Sc., geb. 31. Juli 1994 in Stuttgart

Gemäß der Promotionsordnung vom 12. Januar 2017

Tag der mündlichen Prüfung: 17. November 2023

1. Referent:	Prof. Dr. Peter Sanders Karlsruher Institut für Technologie Deutschland
2. Referent:	Prof. Dr. Armin Biere Albert-Ludwigs-Universität Freiburg Deutschland

To my family

Abstract

The problem of propositional satisfiability (SAT) is to find a variable assignment for a given propositional formula such that the formula evaluates to true or, if no such assignment exists, to report that the formula is unsatisfiable. SAT solving has attracted a great deal of attention due to its theoretical significance, its generic nature, and its broad applicability to a wide range of problems. In this work, we target a number of scalability challenges in the realm of applied SAT solving, guided by three research questions: How can we efficiently exploit modern distributed computing environments for SAT solving? How can we render SAT solving systems in such environments trustworthy for critical applications? And: How can complex applications make more efficient use of SAT solvers in order to handle previously infeasible inputs? We present contributions which address these questions with the methodology of algorithm engineering, combining theoretic considerations with practical engineering.

First, we investigate the efficient scheduling of SAT tasks and other tasks with unknown execution time in large distributed environments. Especially in on-demand computing such as high performance computing (HPC) or cloud services, low scheduling latencies and response times are desirable as well as resource efficiency and fairness. We derive a decentralized scheduling approach which exploits malleability, i.e., the ability of a task to handle a fluctuating amount of computational resources during its execution. We derive fully scalable algorithms for our scheduling model and implement practically efficient variations of them. In experiments on up to 6144 cores, our approach results in scheduling latencies in the range of milliseconds and achieves near-optimal system utilization.

Secondly, we explore the efficient parallelization of SAT solving itself. In particular, we design a compact all-to-all clause sharing scheme which scales to thousands of solvers. In experiments on up to 3072 cores, our approach combined with state-of-the-art solver backends more than doubles the previously highest reported speedups in general-purpose distributed SAT solving. Our approach has dominated the cloud track (1600 hardware threads) of the renowned International SAT Competition four years in a row while also proving highly competitive on a moderately parallel scale (64 hardware threads). Within our job scheduling framework, we show that the combination of parallel job processing and malleable SAT solving can achieve appealing speedups while retaining good resource efficiency.

Thirdly, we address a major shortcoming of most parallel and distributed solvers, namely their inability to produce certificates of unsatisfiability which limits their trustworthiness. We propose the first feasible and scalable approach to generating proof certificates in massively parallel SAT solving. Our distributed algorithm essentially rewinds the solving procedure and tracks the origin of all relevant shared clauses. With reasonable overhead compared to our non proof producing solver, this approach is able to efficiently generate proofs holding many gigabytes of compressed information.

Last but not least, we conduct a major case study on applied SAT solving. Specifically, we present a SAT-based approach to Totally Ordered Hierarchical Task Network (TOHTN) planning—a popular branch of automated planning which provides a rich framework to model complex hierarchical tasks. We present the first SAT encoding of TOHTN problems which preserves a lifted, i.e., parametrized, problem description and therefore avoids the combinatorial blowup which other SAT-based approaches introduce. With an integrated approach that alternates between encoding and incremental SAT solving, our approach generates formulas which are oftentimes smaller by one to two orders of magnitude compared to prior SAT-based approaches. We also present a way to process hierarchical planning problems on parallel hardware using our job scheduling and malleable SAT solving framework. For this means, we enable our system to support incremental SAT solving, rendering it the first large-scale incremental SAT solver.

Put together, our work has led to substantial advances in scalable SAT solving and its efficient application. We thus push the frontier of automated reasoning problems that are feasible to solve in modern computing environments.

Deutsche Zusammenfassung

Das Problem der aussagenlogischen Erfüllbarkeit (*Propositional Satisfiability*, SAT) besteht darin, eine Variablenbelegung für eine gegebene aussagenlogische Formel zu finden, sodass die Formel “wahr” ergibt, oder, wenn keine solche Belegung existiert, die Unerfüllbarkeit der Formel zu attestieren. Das Lösen derartiger Probleme, genannt *SAT Solving*, hat aufgrund seiner theoretischen Bedeutung, seiner generischen Natur und seiner breiten Anwendbarkeit auf eine Vielzahl von Problemen viel Beachtung erlangt. In dieser Arbeit widmen wir uns einer Reihe von Herausforderungen zur Skalierbarkeit von angewandtem SAT Solving, geleitet von drei zentralen Forschungsfragen: Wie können wir moderne verteilte Rechenumgebungen effizient für SAT Solving nutzen? Wie können wir Systeme für SAT Solving in derartigen Umgebungen vollständig vertrauenswürdig machen, um deren Einsatz für kritische Anwendungen zu ermöglichen? Und: Wie können komplexe Anwendungen effizienteren Gebrauch von SAT-Solvern machen, damit zuvor unlösbare Probleme bewältigt werden können? Wir präsentieren Beiträge, die diese Fragen mit der Methodik des *Algorithm Engineering* angehen und dabei theoretische Überlegungen mit praktischer Entwicklungsarbeit verbinden.

Zunächst untersuchen wir die effiziente Verteilung von SAT-Problemen und von ähnlichen Aufgaben mit unbekannter Ausführungszeit auf große verteilte Umgebungen. Insbesondere bei On-Demand-Computing wie *High Performance Computing* (HPC) oder Cloud-Diensten sind niedrige Latenzen und Reaktionszeiten ebenso erwünscht wie Ressourceneffizienz und Fairness. Wir stellen einen dezentralen Scheduling-Ansatz vor, der *Verformbarkeit* (*malleability*) ausnutzt, d.h. die Fähigkeit einer Berechnung, während ihrer Ausführung mit einer fluktuierenden Anzahl von Rechenressourcen umzugehen. Wir präsentieren vollständig skalierbare Algorithmen für unser Scheduling-Modell und implementieren praktisch effiziente Varianten dieser Algorithmen. In Experimenten mit bis zu 6144 Rechenkernen führt unser Ansatz zu Scheduling-Latenzen im Bereich von Millisekunden und erreicht eine nahezu optimale Systemauslastung.

Zweitens untersuchen wir die effiziente Parallelisierung von SAT Solving selbst. Insbesondere entwerfen wir einen kompakten Ansatz für globalen Klauselaustausch, der für Tausende von Solvern geeignet ist. In Experimenten mit bis zu 3072 Kernen hat unser Ansatz in Kombination mit modernen sequentiellen Solvern die bisher höchsten berichteten Speedups von verteiltem anwendungsunabhängigem SAT Solving mehr als verdoppelt. Unser Ansatz hat die Cloud-Kategorie (1600 logische Kerne) der renommierten International SAT Competition vier Jahre in Folge dominiert und sich auch auf moderat paralleler Hardware (64 logische Kerne) als äußerst konkurrenzfähig erwiesen. Innerhalb unseres Scheduling-Systems zeigen wir, dass die Kombination aus paralleler Jobverarbeitung und verformbarem SAT Solving ansprechende Speedups bei gleichzeitig guter Ressourceneffizienz erzielen kann.

Drittens befassen wir uns mit einer großen Einschränkung der meisten parallelen und verteilten SAT-Solver, nämlich ihrer Unfähigkeit, Beweise für Unerfüllbarkeit zu erzeugen, was ihre Vertrauenswürdigkeit einschränkt. Wir stellen den ersten praktikablen und skalierbaren Ansatz zur Erzeugung solcher Zertifikate bei massiv parallelem SAT Solving vor. Gewissermaßen spult unser verteilter Algorithmus den Lösungsvorgang in der Zeit zurück und verfolgt den Ursprung aller relevanten geteilten Klauseln. Mit einem vertretbaren Zusatzaufwand im Vergleich zu unserem Solver ohne Beweis-Erzeugung ist dieser Ansatz in der Lage, effizient Beweise auszugeben, die viele Gigabytes an komprimierter Information enthalten.

Zu guter Letzt berichten wir von einer umfangreichen Fallstudie zu angewandtem SAT Solving. Konkret stellen wir einen SAT-basierten Ansatz für die Planung von hierarchischen Aufgabennetzwerken mit totaler Ordnung (*Totally Ordered Hierarchical Task Networks*, TOHTN) vor — ein beliebter Zweig der automatisierten Planung, der reichhaltige Möglichkeiten für die Modellierung komplexer hierarchischer Planungsaufgaben bereit hält. Wir präsentieren die erste SAT-Kodierung von TOHTN-Problemen, die eine geliftete, d.h. parametrisierte Problembeschreibung beibehält und daher die kombinatorische Vergrößerung der Eingabe vermeidet, die andere SAT-basierte Ansätze mit sich bringen. Mit einem integrierten Ansatz, der zwischen Kodierung und inkrementellem SAT Solving alterniert, erzeugt unser Ansatz Formeln, die im Vergleich zu vorherigen SAT-basierten Ansätzen oft um ein bis zwei Größenordnungen kleiner sind. Wir stellen auch eine Möglichkeit vor, hierarchische Planungsprobleme auf paralleler Hardware zu verarbeiten, indem wir unser Scheduling- und SAT-Solving-System verwenden. Im Rahmen dieser Bemühung haben wir unser System um die Unterstützung von inkrementellem SAT Solving erweitert, was es zum ersten inkrementellen SAT-Solver für große Systeme macht.

Insgesamt hat unsere Arbeit zu erheblichen Fortschritten bei skalierbarem SAT Solving und dessen effizienter Anwendung geführt. Damit erweitern wir den Horizont der logischen Probleme, die in modernen Rechenumgebungen praktikabel lösbar sind.

Acknowledgments

After five years of work,[1] I am thrilled to release this dissertation. I sincerely hope that its readers will find its contents satisfying. With this project coming to an end, quite a few acknowledgments are in order.

First of all, I wish to thank the many people on whose shoulders I've been able to stand, in particular the authors of all the great, openly available SAT solving systems from the past and the present. Many thanks to Markus Iser for providing GBD—a benchmark database which makes instance-specific analyses in SAT research possible for a newcomer without despairing—and many thanks to all the organizers of the past years' SAT Competition, who helped my research and its visibility tremendously by providing a public platform for distributed SAT solver performance. In particular, I wish to thank Markus Iser and Tomáš Balyo for encouraging and motivating me to submit my very preliminary HordeSat *rework to the 2020 Competition.*

During my research, I have been able to get in touch with several different (although related) scientific communities, which I am very grateful for. Many thanks to my dear co-authors Tomáš Balyo, Damien Pellier, Humbert Fiorino, Michael W. Whalen, Marijn Heule, Benjamin Kiesl-Reiter, Dawn Michaelson, and of course Peter Sanders for the pleasant cooperation. Among many other people I've been connecting with, I would especially like to thank Gregor Behnke, Pascal Bercher, Martina Seidl, Max Heisinger, and Armin Biere for fruitful and helpful exchanges.

I want to thank Peter Sanders for the great discussions, his ideas and encouragements, and for the great freedom he entrusted me with to pursue not only our shared research interests but also the ones I came up with myself. It is in large parts due to him that I experienced my doctoral research not at all as a burdensome struggle—which, unfortunately, is the case for way too many PhD students in Germany and around the globe—but as a pleasant and fulfilling time. I also want to thank Armin Biere for agreeing to be the second reviewer of my dissertation and for joining us in Karlsruhe the day of my defense.

I wish to thank the many people who have kindly read and examined drafts of this thesis and provided very helpful feedback: Lukas Hübner, Nikolai Mass, Tim Niklas Uhl, Tobias Heuer, Tomáš Balyo, Matthias Schimek, Moritz Laupichler, Marvin Williams, Markus Iser, Niko Wilhelm, as well as my dad Ekkehard, my mom Karin, my sister Franziska, and my wife Isabel.

Many thanks to all of my colleagues, past and present, for their help and support, many fruitful discussions, and for all the fun we had over the past years.

[1] *Three* years, if subtracting the customary two years due to distributed programming.

I also wish to thank all of my students—participants of the lectures Automated Planning & Scheduling *and* Practical SAT Solving *as well as participants of our SAT solving seminars and the students who I supervised in the context of a qualification thesis, research project or programming project. All of them have put in hard work for our common projects, and it showed!*

Many thanks to my friends in Karlsruhe, Backnang, and in the DMZ for always having such a good time together and for a very enjoyable balance between earnest discussions and utter nonsense.

Last but certainly not least, I wish to thank all of my family for their love and incredible support. You are phenomenal! My final expression of thanks goes to my wife Isabel, who went above and beyond to support me on the home stretch to finish this thesis. Thank you so much!

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

Some of this work was performed on the supercomputer ForHLR funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Some of this work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

The author gratefully acknowledges the Gauss Centre for Supercomputing e.V. (`www.gauss-centre.eu`) for funding this project by providing computing time on the GCS Supercomputer SuperMUC-NG at Leibniz Supercomputing Centre (`www.lrz.de`).

Table of Contents

Chapter 1

Introduction

1

The thesis at hand addresses a number of scalability challenges in the realm of propositional satisfiability (SAT) solving and its application. This chapter serves as an introduction to this thesis. We explain the context of our research, its motivation, and its necessity. We outline our general methodology and provide a problem statement in the shape of three research questions. We then provide a brief overview of each subsequent chapter, describing our approaches and mentioning central results obtained. Lastly, we give some advice for how to read and traverse this thesis.

1.1 Motivation

For over two millennia, philosophers and mathematicians have been concerned with how to formalize and analyze logical reasoning [Smi22]. To arrive in today's age of ubiquitous computing, a fundamental cornerstone of this journey was George Boole's *Mathematical Analysis of Logic*, which introduced a formalization that we know today as *Boolean algebra* [Boo47]. Variables in this algebra may only take one of two values, `true` or `false`. Junctors such as NOT ($\neg$), AND ($\wedge$), and OR ($\vee$) allow to build compound expressions. For example, the statement "*If it does not rain and my bike is intact, I will go swimming.*" can be written as the following Boolean expression:

$$\mathit{Rain} \;\vee\; \neg \mathit{BikeIntact} \;\vee\; \mathit{Swim}$$

We call such a disjunction ($\vee$) of Boolean variables or their negation a *clause.* The above clause is always `true` except for a single case: It does not rain, the person's bike is intact and yet they do not go swimming.

Boolean algebra has become an essential tool in modern sciences, in particular as a crucial foundation for the inner workings, design, and programming of computers [ORe21]. Likewise, the problem of trying to *satisfy* a given Boolean expression, i.e., to assign a value to each variable such that the expression evaluates to `true`, has become a fundamental problem of computer science. This *Propositional Satisfiability* problem, abbreviated SAT, has attracted a great amount of attention not only due to its theoretical significance [Coo71], being the "canonical" NP-complete problem and therefore playing a central role in the famous P vs. NP problem [Coo00]. SAT solving also means to reason over one of the most fundamental and generic forms of knowledge representation and, therefore, can be applied to a wide range of problems.

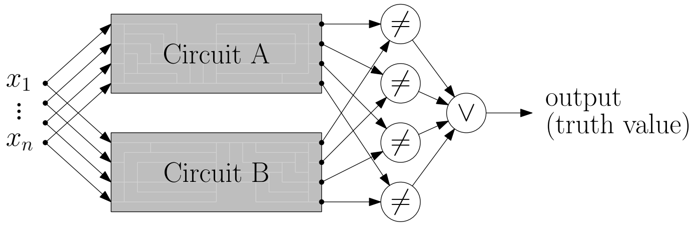


Figure 1.1: Encoding the (in-)equivalence of two circuits A and B as a SAT instance [MG99]. The input bits x_1 to x_n ($n = 4$ in this example) are the Boolean variables relative to which the circuits A and B are encoded. Further constraints enforce that the entire expression is `true` if and only if the circuits' outputs differ in some bit. With some additional variables, all circuits and constraints can be expressed as a conjunction ($\wedge$) of disjunctions ($\vee$), i.e., clauses.

Over the last decades, increasingly efficient SAT solving approaches [SLM92; MS99; Mos+01; ES04; AS09; Lia+16a; Cai+22] have enabled applications to use SAT as a blackbox engine for disciplines as diverse as domain-independent automated planning [KS92] and scheduling [Gro+12], electronic design [MS00], verification [Cla+01], cryptography [SNC09], theorem proving [HKM16], as well as knowledge compilation and explainable AI [Dar20]. These applications, in turn, boosted attention and research towards more efficient SAT solvers [Fro+21; Fic+23]. This positive feedback loop has culminated in highly efficient and openly available software solutions (e.g., [Bie+20a]) which are capable of solving many challenging problems almost interactively.

Let us consider this year's *International SAT Competition* (ISC) [Bal+23a] as an example. One of the submitted solvers, KISSAT_MAB_PROP-NO_SYM [Gao23], was able to solve 84 out of 400 diverse benchmark problems in under ten seconds each. 21 of these problems feature more than 100 000 clauses. One such problem's purpose is to check the logical equivalence of two arithmetic circuits [Bie16a]—a natural application of SAT solving and an important building block for electronic design automation [MG99]. Fig. 1.1 illustrates how such a problem can be encoded as a SAT instance. The concerned instance[1] has over 260 000 variables and over 845 000 clauses and is represented by 15.7 MB of ASCII data. KISSAT_MAB_PROP-NO_SYM solved this instance in 1.33 s, reporting that it is satisfiable. Specifically, the solver found a combination of input bits for which the compared circuits result in different outputs, implying that they are not equivalent.

The above example illustrates the impressive best-case performance of today's sequential SAT solvers. In some other cases, however, the exponential complexity of SAT solving algorithms becomes apparent. Continuing our example, the ISC 2023 also featured some logical equivalence checking problems which *none* of the 42 submitted sequential solvers was able to solve within 5000 s. One of these problems[2]

[1] `g2-ak128astepbg2asisc.cnf`
[2] `multiplier_14bits__miter_14.cnf`

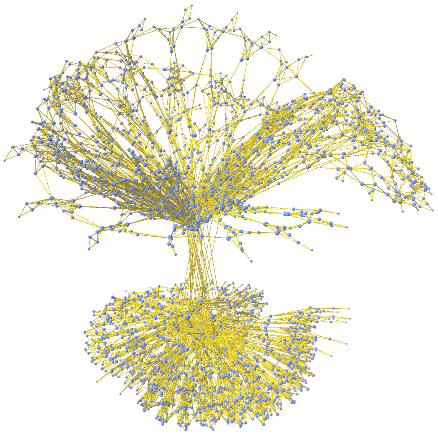


Figure 1.2: Simplified visualization of `multiplier_14bits__miter_14.cnf` via `3dVis` [Sin07]. Each vertex corresponds to a variable. Two vertices are connected whenever the two variables occur together in some clause.

only features 4376 variables and 13 018 clauses. Fig. 1.2 shows a simplified visualization of this problem. Since the encoded circuits are in fact equivalent, this problem is *unsatisfiable*: The solver's task is to construct a proof expressing that there is not a single combination of input bits for which the circuits result in different outputs. This task proves to be substantially more difficult than finding a single input with different outputs in the earlier, much larger instance.

As it seems, no matter how efficient SAT solvers have become, there are always relevant and reasonable problems which are infeasible to solve with the current state of the art. One way to approach this gap is to consider an application's perspective. Orthogonal to advancing SAT solving itself, an equally active research topic in many disciplines is to pursue new and innovative ways to *encode* application instances into SAT problems—aiming to make optimal use of solvers and to push the frontier of problems that are feasible to solve [TM12; ZK17; Gan+19; Sur+22; HB22].

Another way to render SAT solving more powerful is to exploit the increasing parallel processing capabilities of modern computing environments [Lei+20]. First approaches to parallel SAT solving emerged more than 20 years ago, explicitly subdividing the work at hand by partitioning the search space of variable assignments [ZBH96; CW03; LSB07]. While today's successors to this parallelization paradigm have achieved some famous results, such as solving some long-standing open problems of mathematics [HKM16; Heu18; SH23], they do not perform convincingly on other important application problems [BS18]. We want to consider parallel SAT solving that is *general purpose*, i.e., suitable for diverse and previously unseen application instances. Today's best approaches for general-purpose parallel SAT solving are *clause-sharing portfolios* [HJS10; Bie10], where a diverse set of sequential solvers process the entire problem in parallel and frequently exchange useful insights to the problem. Some of these approaches show good practical performance for modest degrees of parallelism, e.g., a few dozen cores [Bie16b; BF22b]. By contrast, modern computing environments increasingly feature distributed computing with thousands of cores.

Two important examples for this are High-Performance Computing (HPC) clusters, often referred to as *supercomputers* [SBA18], and cloud computing environments [Fos+08]. Prior SAT solving systems designed for such environments [ENS14; BSS15; EN19; NCT19; HFB20] are quite limited in terms of scalability and for the most part cannot make efficient use of a few hundred cores or more for average inputs.

In addition, it is still largely unclear how clause-sharing portfolios can be made fully trustworthy. Modern sequential solvers are trustworthy in the sense that they not only claim that a problem is unsatisfiable but provide an actual proof that can be verified independently [Heu16]. Efficient parallel SAT solvers are still missing this crucial feature (*cf.* [HMP14]), which impacts their viability for critical applications. For these reasons put together, the prior state of the art in parallel and distributed SAT solving is unsatisfactory in terms of its suitability for real-world applications.

1.2 Problem Statement

In this work, we target a number of scalability challenges in the realm of applied SAT solving. We understand *scalability* as the degree to which an approach remains effective and efficient when increasing the amount of computational resources and/or the input size. In the context of our research, *input size* can also refer to the *difficulty* of the problem we attempt to solve.

Specifically, our work is guided by the following three research questions:

(i) How can we efficiently exploit modern distributed computing environments for SAT solving?

(ii) How can we render SAT solving systems in such environments trustworthy for critical applications?

(iii) How can complex applications make more efficient use of SAT solvers in order to handle previously infeasible inputs?

1.3 Methodology

Our approaches throughout this thesis are aligned with the methodology of *Algorithm Engineering* (AE) [San09]. As Fig. 1.3 illustrates, AE can be seen as a cyclic process which features the stages of modeling and design, theoretical analysis, implementation, and experiments. From an epistemological point of view, AE is closely related to Popper's *Scientific method* [Pop34]: Traversing the AE cycle means to propose and test *falsifiable hypotheses*, which can be disproven or supported (but not proven) by empirical means—implementation and experiments. A central aspect of AE is that applications of the researched algorithms are, quite literally, *in the loop*. Applications should be considered for finding realistic models and can provide real-world inputs for experiments. In turn, an algorithm's implementation should yield practical and reusable software modules (e.g., software libraries) whereas its theoretical analysis

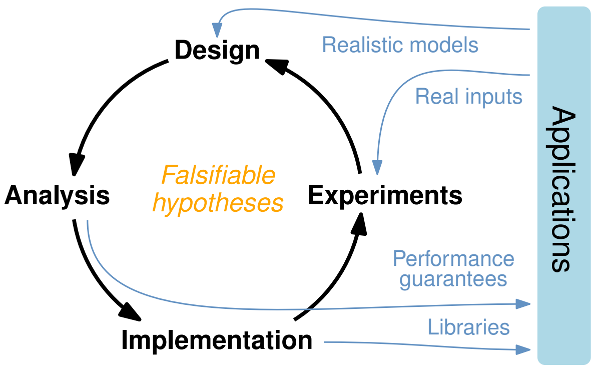


Figure 1.3: The methodology of *Algorithm Engineering*—adapted and simplified from Sanders [San09].

yields performance guarantees. With these principles, AE aims to bridge the gap between classical algorithm theory and pragmatic engineering [San09].

1.4 Contributions

We now outline our contributions throughout this thesis.

Our first line of work is motivated by a simple insight: Scheduling and processing many independent tasks in parallel can be much more efficient than using the same resources to process a single task at a time, especially in cases where the used parallelization does not scale linearly. Consequently, we investigate the efficient online scheduling of SAT tasks and similar kinds of tasks with unknown execution time in large distributed environments. Especially in on-demand computing such as HPC or cloud services, low scheduling latencies and response times are as crucial as resource efficiency and fairness [Fos+08]. We derive a scheduling approach which exploits *malleability*, i.e., the ability of a task to support a fluctuating amount of processing units during its execution. Each job in the system is represented as a binary tree of processing units which can grow and shrink whenever necessary. Our approach is fully decentralized: All processing units participate in a two-stage scheduling protocol whenever the system state changes. First, the fair amount of resources for each job is negotiated. Secondly, a mapping between jobs and processing units is found. We show how this protocol can be realized with fully scalable algorithms and implement practically efficient variations of these algorithms. In experiments on up to 6144 cores, our approach results in scheduling latencies in the range of milliseconds and achieves near-optimal system utilization whenever sufficient demand is present.

Secondly, we explore the efficient parallelization of SAT solving itself. As a point of departure, we consider the massively parallel solver HordeSat [BSS15] which executes hundreds of sequential SAT solvers and enables *clause sharing* among them.

We propose a compact approach to clause sharing which maximizes the density and utility of information that is exchanged across all solvers. We also present an exact filtering approach for clauses shared repeatedly, take measures to reduce the memory requirements of our system, and integrate state-of-the-art sequential solvers in our portfolio. In experiments on up to 3072 cores (64 machines) our approach consistently doubles the speedups which were previously the highest reported speedups for general-purpose SAT solving [BSS15]. The implementation of our approach has been awarded a total of 18 medals in four iterations of the International SAT Competition—dominating the massively parallel ("cloud") track on 1600 hardware threads and also proving highly competitive on a moderately parallel scale (64 hardware threads). Within our decentralized scheduling framework, we show in an experiment on 6400 cores that malleable scheduling of SAT tasks can result in an appealing combination of significant speedups, great resource efficiency, and full utilization of resources.

Thirdly, we address the question of trust in large-scale SAT solving. Whereas sequential SAT solvers have been able for years to generate verifiable proofs for a formula's unsatisfiability [Heu21b], the most scalable parallel and distributed solvers are still missing this crucial feature. This presents a pressing problem which reduces the trustworthiness of modern distributed solvers. We propose the first feasible and scalable approach to generating proof certificates in massively parallel SAT solving. We introduce a distributed algorithm which essentially rewinds the solving procedure and tracks the origin of each required clause with external-memory data structures. Our implementation, based on our award-winning SAT solving engine, takes on average five times its own solving time to assemble and check a proof for a formula's unsatisfiability.

Fourthly, we present a major case study on applied SAT solving. Specifically, we investigate the SAT-based resolution of hierarchical planning problems. Totally Ordered Hierarchical Task Network (TOHTN) planning is a popular branch of automated planning which provides a rich framework to model complex hierarchical tasks [BAH19]. We are the first to propose a SAT encoding for TOHTN problems which keeps a *lifted*, i.e., parametrized, problem description and therefore avoids the combinatorial blowup that is usually introduced by generating a ground representation of the problem. With an integrated approach of instantiating, encoding, and solving the problem hierarchy layer by layer, our approach generates formulas which are oftentimes smaller by one to two orders of magnitude compared to prior SAT-based approaches. Our implementation Lilotane scored the second place in the *Total Order* track of the International Planning Competition (IPC) 2020. As such, we demonstrate how applications of SAT can be made more scalable by considering radical new encodings and by carefully engineering the encoding and solving approach as a whole.

Lastly, we present a way to process hierarchical planning problems on parallel hardware with a combination of our contributions on job scheduling, SAT solving, and SAT-based TOHTN planning. For this means, we enable our system to support incremental SAT solving, rendering it the (to our knowledge) first distributed incremental SAT solver. In an experiment on 2348 cores, we connect hundreds of concurrent planner processes to our scheduling and SAT solving platform, achieving significant (albeit highly domain-dependent) speedups.

Put together, our contributions have substantially advanced the state of the art in distributed SAT solving and its efficient application. As such, our work pushes the frontier of automated reasoning problems that are feasible to solve in modern computing environments. In turn, we also anticipate our work to enable advances in fields which build upon SAT—such as Satisfiability Modulo Theories (SMT) solving, Quantified Boolean Formula (QBF) solving, model checking, or automated planning.

1.5 Chapter Overview

The thesis at hand is structured as follows. **Chapter 2** provides the necessary foundations for our research, most notably parallel processing and SAT solving, and features a unified discussion of related work. In **Chapter 3**, based on a *Euro-Par* conference publication [SS22a] with added content, we present our work on decentralized scheduling of malleable unpredictable jobs such as SAT tasks. **Chapter 4** features our work on rendering distributed SAT solving itself more scalable. This chapter is partly based on a *SAT* conference publication [SS21b] and on technical reports to the International SAT Competition 2020–2023 [Sch20; Sch21e; Sch22; Sch23]. It also features a significant amount of original content. In **Chapter 5**, based on a *TACAS* conference publication [Mic+23], we describe our efforts on producing proofs of unsatisfiability with our solving system. In **Chapter 6**, based on a *JAIR* article [Sch21d], we present a case study on efficient applied SAT solving in the context of hierarchical planning. **Chapter 7** extends this case study by connecting our planning approach with our job scheduling and SAT solving system. In **Chapter 8** we conclude the thesis, estimate the impact of our work, and discuss directions for future work.

1.6 Reading This Thesis

We give some brief advice for how to read and traverse the thesis at hand.

The chapters are ordered in a way to minimize forward dependencies. For non-chronological reading, individual chapters are as self-contained as possible but do contain back references in particular to Chapter 2.

Names of major (software) systems are written in small capitals (e.g., HordeSat or HyperTensioN) whereas smaller software tools are written in monotype format (e.g., `drat-trim`). In the digital version of this document, relevant acronyms and abbreviations (e.g., SAT or TOHTN) can be clicked and refer the reader to a glossary. Similarly, citations link to the corresponding bibliography entry. In both the glossary and the bibliography, each entry lists all pages where this entry was referenced, which allows to backtrack to the prior position of reading.

Author's Notes. *At the beginning of most chapters, I include a small section providing some context from my personal perspective. For the sake of transparency and good scientific practice, I describe which parts of the respective chapter, if any, do not originate solely from myself.*

Chapter 2

Preliminaries and Related Work

2

This chapter establishes important foundations for the subsequent chapters. First, we provide a brief overview of parallel and distributed processing in the scope of this work. We then discuss foundations of (sequential) SAT solving and outline some important applications and extensions before proceeding with a discussion of parallel and distributed SAT solving. We close with discussing some pragmatics of SAT solving research and practice.

2.1 Parallel Processing

In this section, we provide an introduction to parallel hardware and to distributed computing. We also describe important notions to assess parallel algorithms.

2.1.1 Parallel Hardware

We now outline a model for the hardware which we consider throughout this work.

A single machine can feature multiple *hardware threads* which run in parallel. These hardware threads are distributed over one or several *sockets*. Each socket can feature several *cores*, which commonly feature one or two hardware threads each. Exploiting two hardware threads of a single core at the same time can achieve some concurrency, e.g., by hiding I/O latencies, but generally does not yield the same degree of parallelization as using two distinct cores [Tau+02]. Fig. 2.1 (left) shows a schematic example for such a machine with two sockets and two hardware threads per core.

Machines feature a *memory hierarchy*, where the time required to access memory increases by several orders of magnitude for each further level in the hierarchy [San+19, Sect. 2.2]. First of all, each core has exclusive access to a small amount of fast *cache memory*. Secondly, all hardware threads have shared access to a larger amount of *main memory* (or RAM). Multiple threads may read a location of main memory concurrently but only a single thread may write to a certain location of main memory at a time. Main memory can be used to synchronize and cooperate, e.g., using semaphores or special fields which can be queried and modified atomically. Thirdly, all threads may access *disk memory*—for our purposes an unlimited amount of persistent storage. On machines with several sockets, regions of main memory may be associated with certain sockets in a way that accessing another socket's memory region incurs additional latencies. This intricacy is known as *Non Uniform Memory Access* (NUMA) [Li+13].

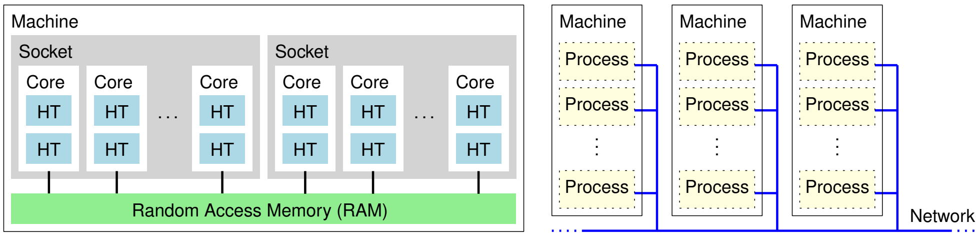


Figure 2.1: Example schematics for a single machine (left) and for distributed machines (right). Cache memory is omitted for the sake of simplicity.

2.1.2 Distributed Computing

In addition to the described model for a single machine, we utilize multiple machines in parallel. These machines do not share any memory.[1] We logically subdivide each machine into a number of *processes*, which are mapped in some way to the machine's cores. In particular, each process can be multi-threaded itself.

The processes communicate with each another with so-called *message passing*. Specifically, we design our algorithms on top of the standardized *Message Passing Interface* (MPI) of which several independent implementations exist (e.g., OpenMPI [Gra+06], MPICH [GL96], and Intel MPI [Int23]). A *message* in the context of this standard is a bundle of data addressed to a specific process and sent over a network. Depending on the used computing environment, message passing may take place via differing network interfaces, from ultra-rapid interconnects such as InfiniBand [Pfi01] or OmniPath [Bir+15] with microsecond latencies to TCP/IP connections with larger latencies by an order of magnitude [Lar+09]. Fig. 2.1 (right) provides a schematic illustration of our communication model.

2.1.2.a Distributed Programming

For consistent exchange of messages, we address processes with unique *ranks* $0, \ldots, m-1$ where m is the number of processes. Note that we occasionally use the term *rank* as a synonym for the process with this rank. Message passing applications are commonly implemented using the concept of *Single Instruction, Multiple Data* (SIMD): Each process runs the same binary program but with different input data. In particular, the processes' ranks are used to diversify the program's execution.

The concept of point-to-point message passing is commonly extended by abstractions named *collective operations* [San+19]. A collective operation is a distributed computation enabled by message passing across a certain subset of processes.

[1] In some distributed systems, one or multiple shared (network) file systems are present which can be used to exchange or accumulate data but are not suited for low-level communication or synchronization across machines. Moreover, there are distributed computation models with shared main memory on an abstract level (e.g., [Zhe+14]) which we do not cover in our work.

We now list some collective operations which we use in subsequent chapters. Assume that P denotes the set of process ranks participating in a collective operation.

- `Broadcast`: A particular rank emits some data D. All ranks in P receive D.
- `Reduce`, `All-reduce`: Each rank $i \in P$ contributes a value x_i. Given an associative operation $\otimes$, the result $X := \bigotimes_{i=0}^{m-1} x_i$ is computed. In a *reduction*, one particular rank receives result X. In an *all-reduction*, all ranks in P receive X. Note that an all-reduction is logically equivalent to a reduction onto a certain rank followed by a broadcast to all other ranks.
- `Gather`, `all-gather`: Each participating rank i contributes a buffer b_i. In a *gather* operation, one particular rank receives the concatenation $B := b_0 \circ b_1 \circ \ldots \circ b_{m-1}$. In an *all-gather* operation, all ranks in P receive this result B.
- `Prefix sum`: Given an associative operation $\otimes$ and an element x_i on each rank $i \in P$, a prefix sum yields the result $\bigotimes_{j=0}^{i} x_j$ (*inclusive prefix sum*) or $\bigotimes_{j=0}^{i-1} x_j$ (*exclusive prefix sum*) on rank i. As such, a prefix sum has as many results as there are participating processes, and each result represents the aggregation of a prefix of the sequence $\langle x_0, \ldots, x_{m-1} \rangle$.

2.1.2.b HPC and Cloud Computing

A computing environment (or *cluster*) which allows to perform distributed computations on thousands, hundreds of thousands, or even millions of cores is commonly called a *supercomputer*. Performing distributed computations on such a supercomputer is referred to as *High Performance Computing* (HPC) [SBA18]. Providers of supercomputers optimize their systems for high-bandwidth, low-latency communication and schedule user jobs in a way that utilization is maximized [YZ13]. Consequently, the *scheduling latency* for a submitted user job can range from seconds to weeks depending on the job's scale and duration and the cluster's utilization [Reu+18].

A related computation model which emerged in the last decades is called *cloud computing*. Compute clouds are mostly commercially provided environments where computational power is made available on demand and, from a user perspective, in a nearly unlimited amount (within the user's financial constraints) [MG11]. As a result, user applications can be deployed and rescaled rapidly. In contrast to most HPC environments, where users receive exclusive "bare-metal" access to physical machines, application code run in a cloud environment is usually virtualized [MG11] which incurs overhead in terms of startup time [XFJ16] and application performance [Exp+13]. In addition, communication between (virtual) compute nodes can be considerably slower than with the high-speed interconnects of HPC [Ost+10]. As such, transferring HPC applications to clouds bears considerable challenges [Ost+10; Exp+13].

2.1.3 Assessing Parallel Algorithms

To analyze parallel and distributed algorithms from an asymptotic point of view, we use the notions of *work* and *span* [Ble96]. The *work* of a distributed algorithm is defined as the total complexity of local operations performed by all processes.

Next, consider a dependency graph induced by the data dependencies between all operations across all processes. The *span*, or *depth*, of the algorithm is defined as the length of a critical path through this graph from the input to the output data.

For example, consider a reduction of n numbers on n processes along a binary tree of processes, i.e., each inner node receives partial results from its children and sends its own partial result to its parent. The work is $\mathcal{O}(n)$ since each process performs a constant number of operations. The span is $\mathcal{O}(\log n)$ since the computation's critical path is to traverse the tree from the bottom up. Similarly, a prefix sum can be implemented with $\mathcal{O}(n)$ work and $\mathcal{O}(\log n)$ span, while broadcasting data of length $\mathcal{O}(n)$ requires $\mathcal{O}(n^2)$ work and $\mathcal{O}(n \log n)$ span.

To assess our algorithms empirically, we use the notions of *speedup* and *efficiency* [San+19, p.62]. If a parallel algorithm A_{par} on p cores has running time $T_{\text{par}}(\mathcal{I}, p)$ on an input $\mathcal{I}$ and the *best available* sequential algorithm A_{seq} for the same problem has running time $T_{\text{seq}}(\mathcal{I})$, we define the *speedup* $s(\mathcal{I}, p) := T_{\text{seq}}(\mathcal{I})/T_{\text{par}}(\mathcal{I}, p)$. If A_{seq} is substituted with the execution of A_{par} on a single processing unit, we use the term *self-speedup* instead. The *efficiency* of A_{par} is given as $E(\mathcal{I}, p) := s(\mathcal{I}, p)/p$. A parallel algorithm has *linear scaling* if $s(\mathcal{I}, p) \in \Theta(p)$ and scales *perfectly* if $s(\mathcal{I}, p) = p$ (i.e., $E(\mathcal{I}, p) = 1$). *Superlinear speedups* $s(\mathcal{I}, p) > p$ are possible if the parallel algorithm profits from additional resources, e.g., having access to more main memory or cache memory in total [San+19, p. 62]. Otherwise, scheduling the parallel execution threads sequentially may in fact constitute a better sequential algorithm than A_{seq}.

A parallel algorithm is commonly evaluated for different values of p to observe how its speedup and efficiency develop. If the work to be performed remains fixed, we use the term *strong scaling*. If the work to be performed is increased proportional to p, we use the term *weak scaling* (see [Gus88]). Intuitively, weak scaling accounts for the fact that investing large amounts of resources is commonly tied to accordingly large inputs.

2.2 SAT Fundamentals

In the following, we aim to provide a mostly self-contained overview of the Boolean Satisfiability problem, denoted SAT. We discuss essential sequential SAT solving approaches, outline the concept of unsatisfiability certificates, and touch on some applications and extensions of the SAT problem. For a comprehensive overview of SAT solving, we refer to the *Handbook of Satisfiability* [Bie+21].

2.2.1 The SAT Problem

A *Boolean variable* is a variable which can only be `true` or `false`. A *literal* is a Boolean variable or its negation. For each literal l we write the negation of l as $\bar{l}$ or $\neg l$. A *clause* is a disjunction of literals, i.e., a logical expression which evaluates to `true` if and only if at least one of the literals in the clause is `true`. A *Conjunctive Normal Form* (CNF) formula is a conjunction of clauses, i.e., a logical expression which evaluates to `true` if and only if each of the clauses evaluates to `true`.

An *assignment* $\mathcal{A}$ for a logical expression F assigns values to some of the variables occurring in F. $\mathcal{A}$ is *partial* if some variables in F are left unassigned, and $\mathcal{A}$ is *total* otherwise. If $\mathcal{A}$ is total and F evaluates to `true` under $\mathcal{A}$, then we write $\mathcal{A} \models F$ and say that *$\mathcal{A}$ is a model for F* or that *$\mathcal{A}$ satisfies F*. We refer to such an assignment as a *satisfying assignment (for F)*. A CNF formula F is *satisfiable* if and only if a satisfying assignment to F exists. Otherwise, F is *unsatisfiable.*

An instance of the *SAT decision problem* is given as a CNF formula F.[2] The task is to decide whether F is satisfiable. We define the *constructive SAT problem* as an extension of the SAT decision problem which additionally requires to output a satisfying assignment $\mathcal{A}$ if F was found to be satisfiable. Likewise, we define the *certified SAT problem* as an extension of the constructive SAT problem which additionally requires to output an *unsatisfiability certificate* $\mathcal{C}$ if F was found to be unsatisfiable. Intuitively, an unsatisfiability certificate is a chain of logical reasoning which the solver used to derive unsatisfiability and which others can verify independently. We will introduce unsatisfiability certificates more precisely in Section 2.2.4.

2.2.2 Complexity

One of the most famous results of theoretical computer science, Cook showed in 1971 that the SAT decision problem is NP-complete [Coo71]. This theorem consists of two parts: The first part is that SAT is in the problem class NP, i.e., SAT can be solved in polynomial time with a *non-deterministic* Turing machine. The second and more intricate part of the theorem is that every other problem in NP can be reduced to SAT in polynomial time: Intuitively, every problem in NP is at most as hard as SAT, up to a polynomial-time transformation. Cook's result implied that a feasible (polynomial-time) algorithm processing SAT problems would also represent a feasible algorithm for a rich class of other problems. The question whether such an algorithm exists is equivalent to the P vs. NP problem, arguably the most famous open problem of computer science [Coo00].

2.2.3 Sequential Algorithms

On the one hand, Cook's result proved that SAT is a problem of fundamental and generic nature to which a rich set of difficult problems can be reduced. On the other hand, the result also implied that it would be impossible to solve SAT "efficiently", i.e., in polynomial time, unless P=NP.

Over the last decades, while $\mathsf{P} \overset{?}{=} \mathsf{NP}$ remained unsolved, research on SAT and its solving approaches soared, leading to two encouraging insights. First, the applicability of SAT to other problems is not only of theoretical interest but can be a practical approach to process a variety of combinatorial problems, such as planning, scheduling, verification, cryptography, or design tasks (see Chapter 2.2.5).

[2]Any Boolean expression with common junctors ($\wedge$, $\vee$, implication, equivalence, XOR, etc.) of size n can be transformed into a CNF formula of size $\mathcal{O}(n)$ by introducing helper variables [Tse83].

Secondly, it showed that SAT solving algorithms can be exponential in the worst case and yet efficient on practical inputs [Fic+23]. In the following, we will touch on the most crucial sequential SAT solving algorithms and techniques.

2.2.3.a DP and DPLL

The *Davis-Putnam* (DP) procedure [DP60] and its successor, the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [DLL62], form the basis for today's most efficient complete SAT solvers [DP21].

The DP procedure is an iterative procedure which, at its core, uses the inference rule of *resolution* [Rob65]: Given two clauses c, c' such that $l \in c$ and $\overline{l} \in c'$ for some literal l, the resolution rule states that the clause $\hat{c} := \{c \setminus l\} \cup \{c' \setminus \overline{l}\}$ can be inferred. This clause $\hat{c}$ is called a *resolvent.* Given a CNF formula F, the DP procedure repeatedly performs the following steps:[3]

- *Unit propagation*, also called unit resolution or Boolean Constraint Propagation [ZM88], extends a partial variable assignment $\mathcal{A}$ for F using a special case of the resolution rule: In a clause c where one literal is unassigned and all other literals are assigned to be `false`, the unassigned literal must be set to `true` to satisfy c. Extending $\mathcal{A}$ this way can trigger additional clauses to satisfy this condition. The rule is thus applied repeatedly until no eligible clauses remain.
- *Pure literal elimination* extends $\mathcal{A}$ by $\mathcal{A}(l) :=$ `true` for each literal l for which $\overline{l}$ does not occur in F.
- *Termination criteria*: If $\mathcal{A}$ is total, i.e., if all variables are assigned, then F is reported to be satisfiable with $\mathcal{A}$ as a witness. If a clause only features `false` literals and no unassigned literals, then F is reported to be unsatisfiable. We call the derivation of such an *empty clause* a *conflict.*
- *Resolution* for $l := v$ and $\overline{l} = \neg v$ is applied to all eligible pairs of clauses, where variable v is picked in some way at each resolution step. This step can add a quadratic number of new resolvent clauses to F.

The most problematic aspect of the DP procedure is its space usage: Each time another variable v is picked for resolution, the set of clauses may multiply in size, which implies exponential space requirements.

DPLL [DLL62] addressed this problem of DP. In contrast to the iterative reasoning procedure DP, the DPLL algorithm performs a *backtracking search* over the space of partial variable assignments. Instead of explicitly generating all resolvent clauses for a picked variable v, DPLL *branches* on v, i.e., the search is continued recursively for $\mathcal{A}(v) :=$ `true` and for $\mathcal{A}(v) :=$ `false` respectively. Like DP, DPLL also incorporates unit propagation and pure literal elimination in each iteration. The search terminates if $\mathcal{A}$ is total, and the search *backtracks* on its last decision if a conflict arises. This behavior corresponds to a kind of depth-first search which only requires additional space in the depth of the search tree, i.e., in the number of variables. Unit propagation and branching put together are sufficient to render DPLL complete both for satisfiable

[3]For the sake of consistent notation, we express DP and DPLL in terms of manipulating a partial assignment $\mathcal{A}$. The original formulations successively edited and simplified clauses.

problems (eventually reaching a satisfying assignment) and for unsatisfiable problems (eventually having searched the entire tree).

2.2.3.b Conflict-Driven Clause Learning

The DPLL search algorithm was substantially refined by a number of different techniques. The successor algorithm named Conflict-Driven Clause Learning (CDCL), pioneered by the solver GRASP in the late 90s [MS96; MS99], is distinguished from DPLL mainly based on two major interrelated advancements: *clause learning* and *non-chronological backtracking* [MLM21]. We discuss some further important features of today's CDCL solvers, such as heuristics and restarts, thereafter in Section 2.2.3.c.

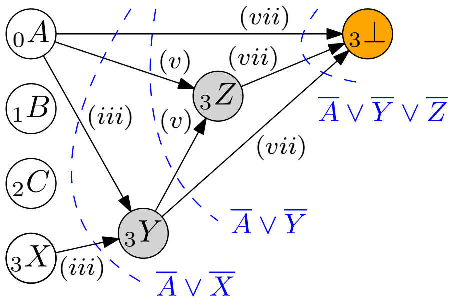
DPLL performs exhaustive search on the tree of variable decisions if F is unsatisfiable. Depending on the variable decision order, a lot of time can be spent searching all branches of a large subtree and backtracking repeatedly even if the decision at a subtree's root already introduces a hidden contradiction to the problem [MS96]. CDCL incorporates the insights gained from conflicts to shortcut this search.

Each assignment to a variable v in our search is either due to a *decision* (a branching point in our search tree) or due to subsequent (unit) *propagation*. We associate each assignment to v with the number of decisions made so far, the *decision level* $d \in \mathbb{N}_0$. For assignments due to propagation, we remember which assignment caused this assignment to v. When encountering a conflict, we use this information to build a causal network of assignments which caused this conflict. Such a network is a directed acyclic graph (DAG) and is called an *implication graph* [DP21].

Fig. 2.2 shows an example for such an implication graph. We begin at decision level 0 and make decisions $A \coloneqq \texttt{true}$, $B \coloneqq \texttt{true}$, $C \coloneqq \texttt{true}$. Our decision at level 3, $X \coloneqq \texttt{true}$, implies $Y \coloneqq \texttt{true}$ due to clause (*iii*), then $Z \coloneqq \texttt{true}$ due to clause (*v*), and finally a conflict due to clause (*vii*) becoming empty.

We can infer two valuable pieces of information from such an implication graph.

First, consider a *cut* of G that separates the conflict from the decisions which caused it. The set of assignments with a cut outgoing edge imply a conflict and, therefore, mark an unsatisfiable subspace of variable assignments. A clause which consists of these literals in negated form is a constraint forbidding this subspace for the subsequent search. We can add this *conflict clause* to the set of problem clauses, not unlike the addition of resolvents in the DP procedure. Fig. 2.2 shows three such cuts between conflict $\{\bot\}$ and its causes $\{A, X\}$. While any of the resulting conflict clauses can be added to the problem, the most prominent strategy to choose a clause is the *1-UIP learning scheme* [Zha+01], where we cut the outgoing edges of the first assignment (seen from the conflict side) that lies on *all* paths from the conflict to the last decision. In our example, this strategy chooses conflict clause $\overline{A} \vee \overline{Y}$ since assignment $Y \coloneqq \texttt{true}$ lies on all such paths and is closer to the conflict than the last decision $X \coloneqq \texttt{true}$ itself. There are many extensions and variants of clause learning. For example, clauses can be minimized after learning [SB09], and the alternative *All-UIP* learning scheme was recently found to reduce the size of learned clauses even further [FB21] and is today used by the state-of-the-art solvers CaDiCaL and Kissat [BFH21].


$_0A$
$_1B$
$_2C$
$_3X$
$_3Y$
$_3Z$
$_3\bot$
(vii)
(v)
(vii)
(iii)
(v)
(vii)
(iii)
$\overline{A} \vee \overline{Y} \vee \overline{Z}$
$\overline{A} \vee \overline{Y}$
$\overline{A} \vee \overline{X}$

(i)	$A \vee B$
(ii)	$B \vee C$
(iii)	$\overline{A} \vee \overline{X} \vee Y$
(iv)	$\overline{A} \vee X \vee Z$
(v)	$\overline{A} \vee \overline{Y} \vee Z$
(vi)	$\overline{A} \vee X \vee \overline{Z}$
(vii)	$\overline{A} \vee \overline{Y} \vee \overline{Z}$

Figure 2.2: Example adapted from Darwiche and Pipatsrisawat [DP21] of an implication graph during CDCL conflict analysis. The problem is defined by clauses (i)–(vii) on the right. Each circle represents a variable assignment and contains the literal set to `true` and the decision level. Decisions are white, propagated assignments are gray, and the conflict is orange. Each edge, i.e., implication is labeled with the clause responsible for the implication. Three cuts of the implication graph and resulting conflict clauses are shown, colored in blue.

Secondly, an implication graph and a resulting conflict clause c indicate the decision levels where the decisions resulting in the conflict were made. *Non-chronological backtracking* is the method of backtracking directly to the decision level where c becomes *asserting*, i.e., a unit clause [DP21]. In Fig. 2.2, non-chronological backtracking lets us backtrack all the way to decision level 0 (last decision $A :=$ `true`) where conflict clause $\overline{A} \vee \overline{Y}$ asserts $\overline{Y}$ due to unit propagation. While clause learning ensures that the subspace identified as contradictory will not be visited again, non-chronological backtracking ensures that search will immediately leave this subspace and continue in a more promising branch. Non-chronological backtracking has been the undisputed backbone of CDCL until Nadel and Ryvchin [NR18] challenged this view by combining non-chronological and (a form of) chronological backtracking with promising results [MB19]. CaDiCaL [Bie18] and Kissat [Bie+20a] now use such a combination.

Beame et al. found that clause-learning solvers, under certain assumptions, can find exponentially shorter proofs than DP or DPLL [BKS04]. Such considerations have led some to view CDCL as a resolution engine instead of a search procedure. For instance, Audemard and Simon remark that "*[m]odern SAT solvers themselves share fewer and fewer properties with their ancestor, the classical backtrack search DPLL procedure*" [AS12], and Huang notes how "*clause learning SAT solvers [...] are now understood as performing a form of resolution, something fundamentally different from search (in the sense of backtracking search for satisfying assignments)*" [Hua07].

2.2.3.c Advanced Techniques

Today's CDCL SAT solvers are remarkably complex pieces of software which feature a plethora of heuristics, techniques, and subprocedures in order to perform well on diverse problems. In the following, we will highlight some of these advanced techniques since this will also help understand how today's solvers can be *diversified* effectively (Section 2.3.2). For the most part, we will put a focus on the techniques which are featured in KISSAT [Bie+20a] and CADICAL [Bie17], arguably the best performing sequential SAT solvers to date.

Efficient propagation. DPLL and CDCL solvers tend to spend the majority of time on propagation [Mos+01; HMS10], i.e., applying a new variable assignment to the entire formula and identifying clauses which become unit clauses. Therefore, in order to obtain efficient solvers, propagation has been a popular subject of optimization efforts. In particular, Moskewicz et al. [Mos+01] introduced a scheme where only two literals are being *watched* per clause. For each literal a *watcher list* contains the clauses in which this literal is being watched [ES04]. During propagation, only these clauses need to be visited, and backtracking of decisions becomes essentially free with this scheme. This technique has been refined over the last decades to achieve highly efficient propagation for all types of clauses [Rya04; Bie10; Bie+20a].

Managing learned clauses. An important trait of CDCL solvers is to periodically reduce the database of learned clauses. While learned clauses are crucial for CDCL, too many of them can slow down propagation significantly and may eventually require too much memory. For instance, Audemard and Simon proposed an aggressive deletion strategy based on the *Literal Block Distance* (LBD) metric [AS09]. Given a conflict clause c, the LBD or *glue value* of c is the number of distinct decision levels from which the conflicting variable assignments originate. Audemard and Simon argued and experimentally showed that clauses with lower LBD are much more likely to help the subsequent solving procedure, in particular those with an LBD value of 2 (termed *glue clauses*). Their solver GLUCOSE consequently deletes the "worse" half of all learned clauses every $20\,000 + 500i$ conflicts (incrementing i each time) and was shown to be highly competitive at the time [Le +09]. Following this success, most CDCL solvers adopted the LBD metric for their solving strategy (e.g., Biere's solvers since 2010 [Bie10], CRYPTOMINISAT [SNC09] and the MAPLESAT family [Lia+16a]). Many of them now follow a *three-tier* clause database management, where very low-LBD clauses are kept indefinitely in the highest tier and other clauses are promoted and demoted across tiers (and potentially removed) based on their participation in recent conflicts [Oh15; Bie+20a]. The LBD metric is also used in further contexts such as selecting clauses to be minimized [Luo+17], scheduling restarts [AS12], and deciding which clauses to exchange across solvers in parallel SAT solving (see Section 2.3.2.b).

Variable decision heuristics. The choice of which literal to branch on during search is crucial for SAT solving performance [Mos+01]. Since this choice is mostly treated as two separate decisions—first deciding on a variable and then committing to a *phase* (another term for the variable's polarity)—we begin outlining crucial work on variable decision heuristics and then continue with the selection of variable phases.

In order to select a variable to branch on, one of the most widely used heuristics is the *Variable State-Independent Decaying Sum* (VSIDS) heuristic [Mos+01].

Moskewicz et al. introduced VSIDS with the solver Chaff [Mos+01]. VSIDS prioritizes literals which occurred in recently learned conflict clauses. This is achieved by “bumping”, i.e., incrementing the scores of a learned clause’s literals and periodically dividing all scores by some constant. A more efficient refinement of VSIDS was proposed via Minisat [ES04] and termed *Exponential VSIDS* (EVSIDS) in subsequent research [BF15]. Another popular decision heuristic is the *Variable Move-To-Front* (VMTF) heuristic introduced with the solver Siege [Rya04] (see also [BF15]). The idea of VMTF is to keep all variables in a list sorted by score in descending order and to bump (a subset of) the variables occurring in a conflict clause by moving them to the front of the list. More recently, Liang et al. [Lia+16a] proposed *Learning Rate Branching* (LRB), a branching heuristic designed to maximize the *learning rate* of the chosen variable, i.e., its potential to generate learned clauses, with the use of Multi-Armed Bandit models. This study created the MapleSAT solver family [Lia+16a]. The solvers CaDiCaL and Kissat employ both EVSIDS and VMTF, switching between them based on the solving mode (see below at “Restarts”).

Variable phases. To decide on a variable’s phase the first time it is selected for a decision, *static* heuristics can be used, which are computed once and remain constant throughout the solving progress. A popular example is the One-Sided Jeroslow-Wang heuristic [JW90] which rates each literal according to the number and size of clauses it occurs in. When branching over a variable several times (with backtracking in between), the technique of *phase saving* has been established [PD07], where each assignment is cached before backtracking and then later re-used as the variable’s default phase. As such, a solver can remember partial solutions despite frequent non-chronological backtracking, hence avoiding redundant work.

With the CaDiCaL solver in 2018, Biere introduced *rephasing* [Bie18] where periodically (with increasing period) the stored phases of variables are manipulated in different ways, including resetting each to its initial phase or flipping it. In addition, in 2019 Biere introduced the concept of *target phases* [Bie19]: During backtracking, if there is a large prefix in the solver’s trail (i.e., the sequence of decisions) which did not cause any conflicts yet, then these assignments are stored as the target phases of the respective variables, which are then used preferably during *stable* solving (see below at “Restarts”). Biere considered target phases one of the most crucial improvements to CaDiCaL [Bie19] and confirmed it to be an essential part of Kissat’s outstanding performance on satisfiable instances [BF20].

Restarts. Gomes et al. [Gom+00] demonstrated that performing n runs of a randomized backtracking search procedure (such as DPLL) results in a running time distribution which exhibits *heavy-tailed* characteristics. In particular, the mean running time does not stabilize with an increasing number of runs n but tends to diverge. For each run there is a non-negligible probability to exceed the mean by multiple orders of magnitude. As a countermeasure, Gomes et al. suggested to introduce periodic *restarts* to these procedures. For the case of SAT solving, a restart means that all decisions the solver made are reset, while certain gathered information such as heuristic scores may be kept. Restarts have been integrated in CDCL solvers, where the database of learned clauses is preserved across restarts [Hua07]. MINISAT implemented restarts following a geometric series [ES04], first restarting after 100 conflicts and then multiplying this interval by 1.5 each restart. Huang identified the *Luby sequence* [LSZ93] as promising for scheduling restarts [Hua07]. The GLUCOSE solver, by contrast, introduced aggressive and rapid restarts which contributed to its outstanding performance on unsatisfiable instances [AS12]. Essentially, GLUCOSE restarts if the LBD of the X most recently learned clauses is considerably higher than average (for $X = 100$ in version 2.0 and $X = 50$ in version 2.1). Since the resulting frequent restarts can deteriorate performance on satisfiable instances, Audemard et al. proposed to delay restarts if unusually many variables are assigned [AS12].

Oh [Oh15] further examined the different behavior of SAT solvers on satisfiable versus unsatisfiable instances and proposed specialized strategies for each case, in particular regarding restart scheduling. Oh suggested that overall solver performance could be improved by interleaving these specialized strategies. Biere [Bie18; Bie19] followed this suggestion and implemented in CADICAL and KISSAT what he later referred to as *stable* and *unstable solving mode* [Bie+20a]. In stable solving mode, the solver is specialized to satisfiable instances; restarts are done rarely and variable scores are updated smoothly using a EVSIDS-based heap. In unstable solving mode, the solver is specialized to unsatisfiable instances; it restarts rapidly and updates variable scores more aggressively with a VMTF-based queue of variables.

Pre- and inprocessing. Modern SAT solvers feature an ever increasing set of processing techniques which promise to boost performance by applying transformations to the input or the set of learned clauses. Such a technique is called *preprocessing* if it is performed once prior to solving, and is often called *inprocessing* if it is interleaved with the solving process [JHB12]. We only provide a few crucial examples for such techniques and refer to the *Handbook of Satisfiability* [BJK21] for an overview of the rich diversity of pre– and inprocessing in SAT solving.

Bounded variable elimination (BVE) [EB05] is a technique which, as Biere stated in 2019, "*remains the most important pre- and inprocessing technique*" [Bie19]. A variable x can be eliminated by applying the resolution rule for $(x, \bar{x})$ exhaustively and replacing the resolved clauses with the resolvents. Sometimes many resolvents can be identified to be redundant, allowing to drop them instead [EB05]. Variable elimination is *bounded* if it is enforced not to increase the volume of the formula [BJK21].

An central method related to simplifying the set of clauses is *subsumption*: A clause C *subsumes* another clause C' if (interpreted as sets of literals) $C \subset C'$. In this case, C' can be replaced by C. Subsumption strategies can be categorized in *forward subsumption*, checking if a given clause is subsumed by F, and *backward subsumption*, checking if F features clauses which are subsumed by a given clause [BJK21].

Another approach to improve the quality of learned clauses is termed *vivification* [Luo+17; Li+20], where SAT solvers attempt to subsume certain clauses (e.g., with a low LBD score [Luo+17]) by testing unit propagation on each clause's literals in a certain order [PHS08]. A closely related concept is that of *clause strengthening*, which was proposed to be performed concurrently to the solving process [WH13b].

2.2.3.d Local Search

Contrary to CDCL solvers which explore the space of partial assignments, the idea of *local search* solvers is to begin with a total (usually not satisfying) variable assignment and to perform small modifications ("flips") on the assignment such that the number of unsatisfied clauses is minimized [HS00]. While local search approaches can solve satisfiable instances this way, they cannot find unsatisfiability and are therefore considered *incomplete* for the task of SAT solving [KSS21].

Since local search solvers work very differently from CDCL solvers, it can be beneficial to schedule local search into CDCL solvers and either find satisfiability this way or otherwise reuse the best found assignment as saved variable phases (and exploit further statistics from local search) for the CDCL procedure [Cai+22].

For a recent and comprehensive overview of local search and other incomplete SAT solving algorithms, we refer to the *Handbook of Satisfiability* [KSS21].

2.2.3.e Look-Ahead Solvers

A third kind of SAT solving algorithm is called *look-ahead solving*. The basic idea of look-ahead solvers is to spend more effort on a single branching decision than CDCL solvers do in order to arrive at a more informed decision [HvM21]. Specifically, a potential branching variable is propagated through the formula to evaluate the impact of the decision before actually committing to its assignment.

While they do not play a crucial role in our work, look-ahead solvers are an important backbone for a particular parallelization approach to SAT solving (*Cube&Conquer*, Section 2.3.1) since they can be used to recursively split a formula into many, similarly difficult sub-formulas. For more details on look-ahead solving we refer to the corresponding book chapter in the *Handbook of Satisfiability* [HvM21].

2.2.4 Certified Unsatisfiability

In Section 2.2.1, we defined the *certified SAT problem* where we extend the pure decision problem by also requiring a justification for the found result. For the satisfiable case, this is easy to achieve. All common SAT solving approaches conclude the satisfiability of a formula by constructing a satisfying assignment. This assignment is suitable as a justification since it can be verified in linear time by evaluating the formula on the assignment. Many applications of SAT require the solver to report a satisfying assignment since it can be decoded to a solution to the original problem at hand.

For the unsatisfiable case, the chain of logical reasoning which the solver followed to arrive at the empty clause, showing unsatisfiability, needs to be considered as a justification. In contrast to a satisfying assignment, this chain is not necessarily linear in the problem input, and in fact, the resolution calculus used by common SAT solvers can require an exponentially sized proof for certain inputs [Hak85; Heu21b].

Consider a formula F and a sequence of clauses $C := \langle c_1, c_2, \dots, c_n \rangle$ learned by a CDCL solver S while processing F. c_n is the empty clause, i.e., the solver has derived unsatisfiability for F. In order to verify that the result is correct, we can check for each $i \in \{1, \dots, n\}$ if c_i is indeed a logical implication of the prior formula:

$$\big(F \cup \bigcup_{j=1}^{i-1} c_j\big) \overset{?}{\Rightarrow} c_i$$

Established clause learning as well as many clause-producing preprocessing techniques have a convenient property named the *Reverse Unit Propagation* (RUP) property [Van08]. For any clause found with RUP property, the above check can always be achieved by means of unit propagation: We assert each literal of c_i to be `false` and then check if unit propagation leads to a direct conflict. In this case, we showed that $F \wedge \neg c_i$ is unsatisfiable, hence c_i is a logical consequence of F and S was correct to derive it. If no conflict arises from unit propagation, c_i is not a sound RUP clause and we therefore reject the proof. Performing this check step by step for the entire sequence C is a means of verifying the result of S. Later proof formats use refinements of RUP to allow for more powerful reasoning and more efficient checking [Heu16; Cru+17].

Propagating each clause in C through F can be expensive, and for large derivations we cannot keep the entirety of $F \cup C$ in memory. Therefore, a popular extension of proof formats is to support the *deletion* of clauses [Heu16]. Whenever S deletes a clause, it logs this deletion just like it logs learned clauses. This deletion can then be mirrored by the proof checker traversing the proof. A proof certificate now takes the shape $\mathcal{C} := \langle a_1, a_2, \dots, a_{n'} \rangle$ where $a_i = (op, c_i)$, $op \in \{add, delete\}$, and c_i is a clause. Outputting such a certificate $\mathcal{C}$ when reporting unsatisfiability solves a *Certified SAT problem instance* according to our definition (Section 2.2.1).

2.2.5 Applications

The academic and industrial applications of SAT solving are numerous and diverse. For example, SAT solving has been used successfully for automated planning [KS92] and scheduling [Met+05], combinatorial design theory [Zha96], test pattern generation [SS97; Bie21], formal verification [Cla+01], electronic design automation [MS00], cryptography [SNC09], theorem proving [HKM16], puzzle solving [Web05], social choice theory [BGP17], design of data structures [WH20], knowledge compilation [Dar20], and explainable AI [SS21a]. In the following we discuss two major disciplines, namely formal verification and automated planning & scheduling, in some more detail.

Automated planning and scheduling. Automated planning is among the oldest and most well-established branches of Artificial Intelligence (AI) and was one of the earliest major applications of SAT solving [KS96]. In its classical formulation, a planning problem consists of a set of world state features, a set of operators which have certain preconditions and effects with respect to the current world state, as well as an initial world state and a set of goals to reach. The task is to find a sequence of operators which successively transform the initial state to a state where all goals hold [GNT04]. Since automated planning is PSPACE-complete [Byl94], a planning problem presumably cannot be encoded into a single polynomial sized SAT encoding.

Instead, the problem is iteratively re-encoded with an increasing number of steps until the resulting formula is satisfiable [KS92; Rin14]. In addition to classical planning, SAT solving has found application in non-deterministic planning [FG00], contingent planning under uncertainty [ML03], multi-agent path finding [Sur12; Sur+22], temporal planning [RG15], and hierarchical planning [MK98; Sch+19b]. We provide a more in-depth introduction to (hierarchical) automated planning in Chapter 6.

Related to automated planning, *scheduling problems* commonly feature a number of resources (e.g., machines) and a number of tasks which need to be executed on the machines [Gra+79]. For many (mostly NP-hard) scheduling formalisms, SAT solving is one of the most popular approaches to efficiently obtain schedules which satisfy all problem constraints. It has been applied to open job shop problems [Kos+10], distributed real-time systems [Met+05], project scheduling [CV11], sport schedules [Zha02; HBB12], and periodic event scheduling problems [Gro+12].

Formal verification. Generally speaking, formal verification aims to show the correctness of a certain program or model with respect to a certain specification. An important branch of this discipline is called *model checking*, where a finite-state transition system is analyzed for whether it behaves correctly [VWM15]. Planning and model checking share structural similarities [GT99; KBS19]: Whereas classical planning aims to find a goal in the space of world states, model checking aims to find a *counter-example* to a model's correctness in the space of model states. As in automated planning, many SAT-based approaches to model checking impose a bound on the depth up to which the transition system is explored [Bie+99]. This so-called *bounded model checking* [Cla+01] has become one of the most essential techniques for formal hardware [VWM15] and software [DKW08] verification. SAT-based methods are also prevalent in *unbounded* model checking [McM03], information flow analysis [KMM13], bug detection [XA05], and many other verification tasks. For further reading we refer to Prasad et al. [PBG05], D'Silva et al. [DKW08], and Vizel et al. [VWM15].

2.2.6 Extensions

Due to the generic nature and wide applicability of SAT solving, a plethora of extensions and generalizations of the SAT problem exist. For example, *MaxSAT* is the optimization problem to SAT where the amount of (certain) unsatisfied clauses is to be minimized [LM21]. Another important example are *Satisfiability Modulo Theories* (SMT) which provide a powerful framework to generalize SAT to mathematical formulas or even entire computer programs [Mon16]. In the following, we describe the concept of *incremental SAT solving* and briefly touch on *QBF solving.*

2.2.6.a Incremental SAT Solving

Incremental SAT solving provides a popular interface for applications which make internal use of SAT solving [AHL08; Liu+16; GB17; KBS19; Sch+19b; Sur+22]. Initially proposed by Een and Sörensson [ES03], today's most widely used incremental SAT solving model extends conventional SAT solving in two aspects. First, instead of a single SAT solving call, a sequence of calls can be made to the same solver instance, with the possibility to add new clauses in between solve calls. Each clause added once is permanent and cannot be removed. Secondly, each solve call may be accompanied by one or several *assumption literals*. An assumption (literal) is a unit literal which the solver will consider to hold for the current solve call *only*. As such, while the set of considered clauses is monotonic in the number of solve calls, assumptions can be used to effectively activate or deactivate some previous clauses [ALS13].

This simple interface enables many useful interactions between an application and a solver. For example, automated planning problems are commonly re-encoded iteratively with an increasing number of permitted actions until a satisfiable assignment can be found (Section 2.2.5). Incremental SAT solving allows to maintain and expand a single formula, which renders the solving process more efficient [GB17; Sch+19a]. Many other applications can profit from this technique, e.g., model checking [ES03; KBS19], multi-agent path finding [Sur+22], and formal grammar analysis [AHL08]. Other use cases of incremental SAT include searching for optimal solutions with respect to some cost metric [Sch+19b] as well as interactively refining an under-defined (relaxed) problem until a solution to the actual problem is found [Glo+19; FBS19c].

Compared to conventional SAT solving, an incremental solver is able to preserve its knowledge base and, in particular, reuse conflict clauses from earlier solving iterations [ES03]. In addition, incremental solving helps to avoid the repeated parsing and preprocessing of similar sets of clauses [FBS19a]. On the other hand, assumption-based incremental SAT solving can require more complex inprocessing techniques [FBS19a] or limit their use, which can lead to worse performance in some cases [NR12].

On a technical level, the standard interface used by most applications of incremental SAT solving is called IPASIR, a reverse acronym for *Re-entrant Incremental Satisfiability Application Program Interface* [Bal+16]. IPASIR features a small set of functions which an application can call to interact with a SAT solver, such as adding literals/clauses to the problem, performing a solving attempt with a number of assumptions, and querying the found satisfying assignment to a certain variable. The SAT solver code is directly compiled into or linked with the application code.

Note that there is very few literature on parallel incremental SAT solving to our knowledge. Wieringa and Heljanko [WH13a] presented asynchronous processing of several increments for certain use cases on up to eight threads.

2.2.6.b QBF Solving

A *Quantified Boolean Formula* (QBF) is a CNF formula in which all Boolean variables are *quantified* either universally ($\forall$) or existentially ($\exists$) in a certain order [Bey+21]. As in SAT solving, the QBF problem is to decide whether the formula is satisfiable. Constructing and reporting a satisfying assignment for a QBF is significantly more complex than for a usual CNF since the assignment to each variable needs to be expressed relative to the values of its preceding variables. QBF solving is the canonical PSPACE-complete problem [GJ79] and has uses, among others, for automated planning [CFG13; SvdP22], explainable AI [DM21], and bounded model checking [DHK05].

For further information on QBF solving we refer to the corresponding chapter in the *Handbook of Satisfiability* [Bey+21].

2.3 Parallel SAT Solving

We now turn to the parallelization of SAT solving. We focus on two orthogonal parallelization approaches which together subsume the vast majority of parallel SAT solvers, namely explicit search space partitioning approaches and portfolio approaches, both with and without clause sharing across workers. After discussing the general parallelization paradigms, we discuss SAT solving approaches specifically designed for large distributed systems. Lastly, we discuss parallel *certified* SAT solving.

For a broad overview of early parallel SAT solvers with a finer taxonomy of approaches than ours, we refer to Martins et al. [MML12]. A more recent but less comprehensive overview of parallel SAT solving was given by Balyo and Sinz [BS18].

2.3.1 Explicit Search Space Partitioning

Since DPLL and CDCL are fundamentally based on the notion of searching the space of (partial) variable assignments, a natural approach to parallel SAT solving appears to be to parallelize this search itself. For many years, most parallel SAT solvers followed this paradigm [BS18]. We now highlight a number of important works on this subject.

One of the first parallel SAT solvers in literature, named PSATO [ZBH96], introduced the notion of *guiding paths*. A guiding path represents a path of decision variables along the decision tree of a parallel DPLL procedure and tracks which nodes along the path still need to be explored in the opposite direction. Multiple workers can work independently on different guiding path prefixes, which is equivalent to assigning a different *partial assignment*, i.e., a small number of assigned variables, to each worker. Deriving the satisfiability of a subtree is equivalent to deriving the satisfiability of the original problem via appending the partial assignment which corresponds to the subtree. A subtree is pruned if it proves to be unsatisfiable, and pruning the root of the tree implies that the input formula is unsatisfiable. Interestingly, this very early work on parallel SAT solving already considered distributed computation, preemption of solvers, and fault tolerance: Guiding paths are distributed by a leader node to all workers,[4] and bookkeeping the guiding paths the workers followed allows to resume an interrupted or cancelled solving procedure later on [ZBH96].

Similar to PSATO, Böhm & Speckenmeyer introduced a parallel solver [BS96] which assigns a certain subspace of assignments to each worker. However, their solver performs a rebalancing of work whenever the estimated workload of a worker goes below a certain limit. The workload is estimated based on the number of unassigned variables. Rebalancing itself is performed based on prefix sums. Böhm & Speckenmeyer

[4]Note that most of the discussed work used the *master/slave* terminology, which we adjusted to *leader/worker* to follow today's naming conventions.

reported near-linear speedups on particular kinds of formulas (random k-SAT and Tseitin graph instances) for up to 256 processors.

Attention to parallel SAT solving soared in the 2000s. As an important cornerstone, guiding path based solver PaSAT [SBK01] introduced *clause sharing* (then called *lemma exchange*) to parallel SAT solving as an additional means to improve performance. Intuitively, if a solver finds a conflict clause, then it may share this clause with the other solvers in order to reduce redundant work. We discuss clause sharing in more detail in the context of solver portfolios in Section 2.3.2.b.

The basic ideas of dynamic load balancing and clause sharing were adopted and refined in subsequent works [CW03; BSK03; JLU05; LSB07]. Chrabakh and Wolski [CW03] proposed to recursively split a subproblem once solving time surpasses a certain threshold, and Blochinger et al. [BSK03] introduced load balancing via *randomized work stealing* [BL99] to parallel SAT solving: If a worker runs out of work, it will attempt to "steal" a subproblem from a different worker. The latter work uses the notion of *mobile agents* to share clauses among workers: Each worker is associated with a mobile agent which visits remote workers one by one and collects knowledge on its way until it returns to its home worker. MiraXT [LSB07] uses a single shared-memory clause database across all workers with concurrent read access.

In later years, the term *Cube&Conquer* has been coined for an extreme variant of search space partitioning where a large number of partial assignments—named *cubes*—are generated using look-ahead solvers (Section 2.2.3.e) and then distributed over all available workers [Heu+11]. In principle, this static load balancing can result in uneven load across workers. Still, by using good branching heuristics, generating many cubes and distributing them randomly, the expectation is that good load balancing is achieved in practice. Cube&Conquer is a simple and effective parallelization technique after all cubes have been generated, and unsatisfiability proofs for all individual cubes can be transformed into an unsatisfiability proof for the entire formula [HS18]. C&C solving in large distributed systems managed to solve long-standing open problems of mathematics, such as the Pythagorean Triples problem [HKM16] or Schur number five [Heu18]. On shared-memory hardware, the Lingeling-based C&C solver Treengeling [Bie12; Bie14] was among the top performing parallel solvers in some iterations of the International SAT Competition (ISC), especially on hard combinatorial problems [Bel+14]. There has been work towards finding a middle ground between C&C and conventional search space partitioning, for instance letting C&C solvers split and generate cubes dynamically based on perceived difficulty [Aud+16; HFB20; Sch21a], not unlike the much older GrADSAT [CW03].

Overall, search space partitioning solvers promise speedups by splitting the problem into several subproblems. The main drawback of these approaches is that the heuristics used to split problems (or generate cubes) are crucial for overall performance and ideally fine-tuned to the application at hand [HKM16]. A bad choice of a branching variable may not divide the work at hand into two halves but rather *multiply* it, yielding two problems that are as hard as the original problem—a phenomenom Schulz and Blochinger referred to as "*bogus splits*" [SB10]. Another possible problem is that some of the split problems may turn out to be trivial ("*oblique splits*" [SB10]).

If such splits happen repeatedly, a *ping-pong phenomenom* occurs where more time is spent on communication and waiting than on solving [JLU01]. In terms of SAT solving that is *general purpose*, i.e., practically efficient for diverse application instances, today's search space partitioning solvers tend to be outperformed [BS18] by another parallelization paradigm which we describe in the following.

2.3.2 Solver Portfolios

Consider a large assembly of puzzle experts which are given a single difficult puzzle, e.g., a (generalized) Sudoku problem [Web05]. Their task is to solve the puzzle as quickly as possible. Each of the experts is rather anti-social and achieves the highest output if left undisturbed. Since each expert has their own strategies, strengths, and weaknesses, it can be an effective approach to have all experts work independently on the entire puzzle. Only one of the experts needs to arrive at a solution.

The SAT solving approach which corresponds to the described "assembly of experts" is termed the *(pure) portfolio* approach [GS03]: A number of sufficiently diverse solvers are executed in parallel and "compete" for solving the same problem. The optimistic view on this simple kind of parallelization is that it effectively emulates a *perfect oracle*—a theoretical device which has access to a set of approaches and, given a particular problem, always selects the approach which solves this problem the fastest.[5] Such a device is also called the *Virtual Best Solver*, VBS in short [Xu+12]. A more pessimistic (or perhaps realistic) view is that a pure portfolio, while it may be effective for many problems, is not *efficient* (see Section 2.1.3). With p solvers, only a single solver contributes to the final solution, hence the efficiency is $1/p$ relative to the "winning" solver. If we only consider a single particular instance, the definition of a (parallel) speedup even suggests that there is no speedup at all since we are comparing our portfolio to the *best available* sequential algorithm.

2.3.2.a Diversification

A crucial aspect of pure portfolios is the *diversification* of solvers. Diversification is any kind of mechanism which lets the participants in a portfolio explore different parts of search space [HJS10; BSS15]. Ideally, small differences made to a solver cause a "butterfly effect" with respect to the solver's internal state and therefore its decision making, resulting in different search space exploration. In the following, we provide some common examples for diversification strategies.

- *Supplying different random seeds to solvers.* Arguably the simplest way to diversify multiple instances of the same solver, this lets solvers take different branches when taking random decisions [BSS15].
- *Setting initial variable phases.* When a SAT solver selects a variable to branch on, its *phase* decides which value it should be assigned first (Section 2.2.3.c,

[5]This "oracle" view neglects potential slowdowns incurred by running solvers in parallel due to shared resources [Aig+13].

Variable phases). If some initial variable phases are set differently for each solver, the solvers may explore search space in a different manner [HJS10].
- *Using different solver parameters.* State-of-the-art SAT solvers commonly have a large set of configuration options which may be set at runtime. Among many other options, this can include restart intervals [HJS10], pre- and inprocessing options [Bie10] or clause database management [AS17a].
- *Using different SAT solvers.* Perhaps the strongest diversification technique, employing wholly different SAT solving algorithms and/or implementations tends to lead to very different search behavior across the portfolio members [Xu+12]. However, this diversification technique is very limited with respect to the portfolio size, since there is only a limited number of somewhat competitive SAT solvers to integrate in a portfolio solver.[6]

Some literature [Guo+10] describes diversification as one of two poles in parallel SAT solving, *intensification* being the other, which need to be balanced. Some go as far as referring to this as the "*intensification/diversification dilemma*" [Aud+16]. Intensification refers to the effort of focusing several solvers to a certain subspace of search space (yet still with individual diversification) if that subspace seems promising.

2.3.2.b Clause Sharing

Going back to our puzzle analogy, consider now that the puzzle experts begin to hold brief meetings in periodic intervals. In such a meeting, each of the experts has the opportunity to share the insights they gained since the last meeting, e.g., a partial solution to the puzzle. All experts then continue to work independently, free to include the other experts' insights into their own solving process as they deem fit.

The described "meetings" in our assembly of experts correspond to what we know as *clause sharing* in parallel portfolio SAT solving. While clause sharing for parallel SAT solving has been introduced first in the context of search space partitioning solvers [SBK01] (see Section 2.3.1), it is parallel portfolio solvers with appropriate clause sharing which tend to perform the best in terms of general-purpose parallel SAT solving [Bal+16; BHJ17; Fro+21]. Intuitively, each shared clause has the potential to prune the search space, and communicating a promising pruning opportunity found by a single solver to many other solvers can lead to significant acceleration [HJS10; BSS15]. This can reduce the redundancy of work performed in solvers and, in practice, lead to considerable speedups. That being said, clause sharing also adds overhead in terms of computation, synchronization and/or communication. For this reason, there is a trade-off to consider between sharing many potentially useful clauses and keeping the solver threads as undisturbed as possible [HJS12; ENS14; Aud+17].

The number of conflict clauses produced by a set of solvers is linear in the number of solvers (disregarding duplicates). In distributed setups with hundreds of solvers, it is not believed feasible to share all learned clauses across all solvers [ENS14].

[6] Bach et al. [BIB22] computed optimal pure portfolios of k solvers based on data from the ISC 2020–2021. Their results indicate stagnating performance beyond $k \approx 20$.

As such, parallel solvers need to prioritize which clauses to share [HJS10; Bie13; ENS14]. Most commonly, the following two metrics for clause quality are considered:

- *Clause length.* This very simple metric is based on a fundamental property of CDCL: The fewer variables are part of a conflict clause, the larger the subspace of assignments which the conflict clause constrains. As a consequence, shorter clauses are more likely to be useful to another solver. Using clause length to prioritize clauses to share ranges back to PaSAT [SBK01]. In the first portfolio solver ManySAT [HJS10] only clauses of length eight or less were shared.
- *Literal block distance* [AS09]. This metric, also called LBD or *glue* value, indicates how many distinct decision levels a conflict features (see Section 2.2.3.c, *Managing learned clauses*). Clauses with a low LBD value are more likely to be useful again in the search procedure [AS09]. In contrast to clause length, LBD is a metric that depends on the particular solver state in which the clause was derived and may also be updated during subsequent search [AS09].

In terms of clause quality metrics beyond length and LBD, Audemard et al. proposed to have receiving solvers assess the subjective merit of incoming clauses based on how well they fit to the solver's current variable phases [Aud+12], and Vallade et al. proposed a metric for clauses based on the formula's community structure [Val+20a].

Since the distribution over the produced clauses' quality may vary based on factors such as execution time or formula structure, imposing fixed thresholds for shared clause quality (e.g., [HJS10; Bie10]) can result in too many or too few shared clauses [HJS12]. Therefore, some adaptive quality limits have been proposed. For instance, Hamadi et al. [HJS12] introduced an adaptive control scheme where the quality threshold for clause sharing evolves over time between each pair of solvers, and HordeSat [BSS15] features an initially very strict LBD limit for each solver that is increased successively until the targeted output volume is reached.

Clause sharing can be implemented in a distributed and decentralized manner, which allows to deploy clause-sharing portfolios on large distributed systems without a bottleneck on a single machine [BSS15]. Recently, some parallel solvers [Val+20b; Ehl+20] additionally employ concurrent clause strengthening [WH13b] in separate threads or even on dedicated GPUs [PSM21] to increase clause quality.

2.3.2.c An Overview of Clause-Sharing Portfolio Systems

Hamadi et al. introduced parallel portfolio SAT solving with ManySAT [HJS10]. It featured two crucial ingredients adopted by most later portfolio solvers: *Diversification* of individual solvers and *clause sharing* across solvers. While ManySAT was fine-tuned to four cores only and restricted to shared-memory parallelism, it initiated what can be considered a paradigm shift in parallel SAT solving. The portfolio paradigm was adopted by subsequent solvers such as Plingeling [Bie10], SArTagnan [KK11], and PeneLoPe [Aud+12]. Since 2010, portfolio solvers have also dominated the International SAT Competition [BS18]. This included some extreme cases like ppfolio [Rou12]—a script which just executed several winners from a past competition in parallel—and a similar approach with slightly more tuning [Wot+12].

For many years Plingeling was considered the best performing parallel SAT solver [BS18]. It runs configurations of the sequential SAT solver Lingeling [Bie10] and, in later versions, the local search solver YalSAT [Bie14]. Lingeling instances are diversified in terms of restart scheduling, inprocessing options, and initial variable phases. While the initial version of Plingeling only shared unit clauses, later versions also share detected literal equivalences as well as learned clauses up to length 40 and LBD 8 [Bie13]. Plingeling decides how many YalSAT instances to run based on how closely the input resembles a uniform random formula [Bie14].

Apart from Plingeling, the Glucose-based portfolio Syrup showed competitive performance [AS14; AS17a]. Two central ideas in Syrup are (a) a cautious kind of clause exchange which only considers a clause for sharing after it has been locally encountered twice, and (b) putting each incoming clause in a *probation* where only one of its literals is watched. Only if this literal is falsified at some point, the clause is *promoted* to be handled normally. This careful approach to clause sharing results in a substantial reduction of the set of exchanged and imported clauses [AS14] and has been (partially) adopted by several later systems [Aud+16; Aud+17; EN19].

Balyo et al. introduced a generic and modular framework for clause-sharing portfolios with HordeSat [BSS15]. Since HordeSat is specifically designed for massively parallel hardware, we discuss it separately in Chapter 2.3.3.a. Many other modern portfolios are built on top of the PaInLeSS framework [Le +17b]. PaInLeSS offers an even more modular architecture than HordeSat in order to "painlessly" develop parallel SAT solvers and explore new techniques. It offers parallelization via portfolios, clause sharing, and search space partitioning. PaInLeSS adopted many of HordeSat's features and parameters, such as diversification based on sparsely and randomly setting variable phases, periodic all-to-all clause exchange of 1500 literals per solver unit per sharing, and an adaptive LBD limit for exporting clauses based on the volume of produced clauses. PaInLeSS portfolios using MCOMSPS[7] as a sequential backend solver [Le +17a; Val+20a; Val+21] were some of the best performing parallel approaches in more recent competitions [Fro+21]. The most recent versions of P-MCOMSPS also feature concurrent clause strengthening in dedicated threads [Val+20a; Val+21]. While PaInLeSS as a framework is designed for shared-memory parallelism, there have been recent efforts to develop distributed solvers by orchestrating multiple PaInLeSS instances [Val+21].

A shared-memory solver performing very well in 2022, ParkissatRS [ZCC22], is also built on top of the PaInLeSS framework but uses a Kissat [Bie+20a] variant as a solver backend. Diversification is achieved by randomly shuffling the branching order of decision variables. The successor to this system, named PRS [ZCC23], enhances this style of diversification and also features a distributed setup. In this setup, PRS deploys four different groups of processes (SAT, UNSAT, DEFAULT, and MAPLE), the latter of which uses MCOMSPS rather than Kissat. Clause sharing across processes is performed only within a unidirectional ring communication structure.

[7] MapleCOMSPS [Lia+16b], or MCOMSPS, is a solver based on CoMinisat-PS [Oh16] with the LRB heuristic from MapleSAT [Lia+16a] and some further changes.

Since the running times of clause-sharing portfolios can vary significantly across different runs, Nabeshima and Inoue [NI20] proposed a framework for *deterministic* parallel SAT solving. By synchronizing all solvers at each sharing based on reproducible metrics (e.g., the number of decisions or conflicts), the parallel solving procedure results in exactly the same data flow and thus identical behavior for repeated runs.

Fleury and Biere recently presented Gimsatul [FB22], a system with a remarkably different architecture compared to most portfolio solvers. Gimsatul performs true, physical sharing of clauses without copying them across different solver threads. Consequently, Gimsatul is written from scratch instead of reusing existing sequential solvers. This approach led to a decreased memory footprint as well as to near-linear self-speedups for up to 16 threads on a 16-core machine. In the International SAT Competition, Gimsatul [BF22b] did not yet perform competitively [Bal+23b]. Gimsatul's architecture is restricted to shared memory parallelism by design. That being said, orchestrating Gimsatul instances on multiple machines with distributed clause sharing across them appears to be a possibility for future work.

2.3.2.d Scalability Limits

There are some theoretical arguments for intrinsic limits to the scalability of parallel clause-sharing solvers. On a very fine grained level, the problem of unit propagation is known to be P-complete and, therefore, difficult to parallelize [HW13]. While this limitation in principle holds for any algorithm running CDCL in parallel, today's solvers are parallelized on a much more coarse grained level.

Katsirelos et al. [Kat+13] used the structure of proofs found by sequential solvers to argue that the efficiency of parallel clause-sharing solvers is intrinsically limited. Specifically, the considered proofs often resemble a deep network of clausal dependencies. All clauses along a critical path through this network, up until the empty clause, need to be derived sequentially. In particular, Katsirelos et al. argue that typical proofs feature several clauses which are necessarily part of *all* critical paths and which therefore present bottlenecks for a parallel derivation.

While this study does indicate that there are (UN)SAT instances which are intrinsically hard to parallelize with clause sharing, it is difficult to estimate its practical consequences for today's clause-sharing solvers. Most importantly, Katsirelos et al. assume that a parallel solver needs to reproduce the same proof which the sequential solver found. In reality, many application instances may allow parallel solvers to find another proof of similar volume but with fewer (perceived) bottlenecks (*cf.* [FS18]). It is also worth considering that the majority of clauses learned by a sequential solver do not actively contribute to its final proof [Sim14], indicating that the resolution steps taken by sequential CDCL solvers are suboptimal as well. Ehlers and Nowotka [EN19] argue that strict scalability limits for individual instances are of limited concern if the main objective is to achieve good performance in terms of *weak scaling*, where the difficulty of problems increases together with the degree of parallelism.

2.3.3 SAT Solving in Distributed Systems

The majority of works we discussed so far have been focusing on shared-memory parallelism. To efficiently exploit massively parallel and distributed systems, we differentiate two strategies: On the one hand, isolated formulas can be solved with a massively parallel solver that uses all available resources. On the other hand, multiple SAT tasks can be scheduled and resolved in parallel. We will now shed light on both these strategies and their combination in previous literature.

2.3.3.a Massively Parallel SAT Solving

The oldest massively parallel SAT solvers performed explicit search space partitioning, such as PSATO [ZBH96] and the solver by Böhm & Speckenmeyer [BS96] (see Section 2.3.1). A somewhat recent distributed search-space partitioning solver is SATUZK-DDC [Gri17], which was evaluated on 160 cores and achieved a median speedup of 5.7 over sequential solver SATUZK-SEQ on the ISC 2017 benchmark set [Gri17]. (We discuss our view on speedup metrics later in Section 2.4.2.b).

The most famous results achieved by parallel and distributed SAT solving have been due to C&C setups [HKM16; Heu18; SH23]. All cubes are first generated on a single processing element, followed by a decentralized solve phase with independent SAT solver tasks on many compute nodes [Heu+11]. Practical advantages of this strategy are that the cubing and solving phases can be performed independently and that no communication across solving tasks is required. However, in order to achieve the mentioned results, several steps of manual labor were required, such as identifying good splitting strategies for the problem at hand. By contrast, some integrated distributed systems for C&C SAT solving have been proposed, such as DOLIUS [Aud+14] and, more recently, PARACOOBA [HFB20] which Heisinger et al. showed to achieve linear speedups on a specific kind of instance [HFB20]. In terms of general-purpose distributed SAT solving, C&C strategies are currently not on par with clause-sharing portfolio solvers [BS18; Fro+21; Bal+23a].

HORDESAT [BSS15] is a system for massively parallel clause-sharing portfolio SAT solving. Its modular solver interface allows to orchestrate different SAT solvers without changing their internal workings. Each process in HORDESAT runs multiple solver threads (four in the original evaluation). Dedicated communication threads are used for communication across processes—solvers never need to be synchronized. HORDESAT performs periodic all-to-all clause exchange via a collective operation (*all-gather*, see Section 2.1.2.a). Each process contributes a fixed volume of the best (shortest) clauses produced by its local solvers. Initially only clauses with LBD score ≤ 2 are considered for export. A process lifts this restriction successively whenever it is not able to contribute the expected volume of clauses. HORDESAT performs approximate filtering of previously seen incoming clauses using Bloom filters [Blo70]. Balyo et al. evaluated HORDESAT on up to 2048 cores [BSS15], which to our knowledge is the largest evaluated scale of a portfolio solver (prior to our work). HORDESAT reached a median speedup of 13.8 at 1024 cores (13.1 at 2048 cores) on ISC application benchmarks [BSS15].

We do not necessarily agree with the claim that HordeSat achieves "*superlinear average speedup for difficult instances*" [BSS15] since the underlying speedup metric is statistically not meaningful—see Section 2.4.2.b.

Ehlers and Nowotka explored massively parallel SAT solving with a system named TopoSAT [ENS14; EN19]. TopoSAT is an integrated system which orchestrates a number of modified Glucose [AS09] instances. Instead of performing all-to-all clause sharing in fixed time intervals, TopoSAT threads export clauses autonomously either after some time since the last sharing passed *or* when their internal export buffer runs full. Only clauses with LBD ≤ 4 are considered for export. TopoSAT can also follow a lazy clause exchange policy which only exports clauses after at least four process-local solvers deemed this clause relevant. At import, each incoming clause is attributed an LBD value equal to the clause's length, a conservative upper bound, whereas HordeSat reuses the LBD from the exporting thread in each importing thread. The initial version of TopoSAT realizes clause sharing with point-to-point messaging along communication graphs [ENS14]. To our understanding, TopoSAT2 as submitted to ISC 2020 [Ehl+20] has each solver process send each batch of clauses to every other solver process (naïve *all-to-all*), implying a quadratic number of messages.

Audemard et al. proposed a distributed version [Aud+17] of their shared-memory solver Syrup [AS17a]. They emphasized the benefit of combining multiple solvers per (MPI) process with multiple processes across machines. In addition, Audemard et al. observed degrading performance the more often all-to-all clause sharing is performed. Instead, their system D-Syrup has solvers individually export clauses over the network, presumably to all other processes. As in Syrup, clause export and import is done carefully by only allowing clauses encountered twice to be exported and by putting imported clauses in a certain probation. In experiments on up to 256 cores, their system D-Syrup outperformed the initial versions of HordeSat and TopoSAT [Aud+17]. Audemard et al. did not report speedup measures.

Burgess et al. recently proposed the distributed system Dagster [Bur+22] which incorporates user knowledge on how to partition a given formula. The interrelated subproblems are then solved with leader-worker load balancing and search space partitioning. Since Dagster expects a DAG of subproblems as an input, it is not obvious how to exploit the system for general-purpose SAT solving.

Last but not least, there have been some efforts to parallelize local search SAT solving (see Section 2.2.3.d) on up to 256 cores [AC12].

2.3.3.b Processing SAT Tasks in Parallel

In general, processing many SAT formulas in parallel constitutes a more efficient use of computational resources than using all resources for a single formula at a time. The argument supporting this claim is simple: Since mean speedups reported by general-purpose distributed SAT solvers are strongly sublinear (see Chapter 4), solving k formulas with p/k processing elements each leads to higher efficiencies than solving one formula at a time with p processing elements. The most basic way to achieve this is by running p sequential solvers concurrently to process p formulas (*cf.* [Aig+13]).

Ngoko et al. presented a distributed system for cloud environments [NTC17; NCT19]: A centralized scheduler uses running time predictions to compute an offline schedule that features stages of portfolio solving without any clause sharing, with the reasoning that "*such solutions [for exchange of knowledge] are not necessarily suitable for distributed clouds in which the communication time could be important*" [NTC17].

Some work on scheduling SAT tasks in distributed systems concentrates on explicit search space partitioning. The C&C solver Paracooba [HFB20] supports parallel processing of multiple jobs and malleable load balancing: The partitioning of the problem at hand into a large number of cubes allows to dynamically redistribute work if new compute nodes register or if leaving nodes unregister from a computation. Another C&C platform has been proposed for "serverless cloud" setups [OWB21b].

Recently, Biere et al. proposed a mechanism to migrate the internal state of a (sequential) SAT solver from one machine to another [Bie+22] which can allow the preemption and/or rescheduling of SAT tasks.

2.3.4 Parallel Certified SAT Solving

In the following, we describe the state of the art for providing certificates of unsatisfiability (Section 2.2.4) in parallel SAT solving.

Achieving certified unsatisfiability is trivial for *pure* portfolio solvers if each of the employed solvers is able to output a proof itself. Certified SAT solving is also possible with search space partitioning solvers by appropriately joining the proofs for all subproblems [HKM16; Nai+22]. For clause-sharing portfolios, on the other hand, the derivation of the empty clause can depend on a clause produced by another solver, which may again depend on clauses from other solvers, and so on. The full chain of reasoning for a formula's unsatisfiability can thus be a dense and interleaved network that features conflict clauses from many or even all participating solvers.

Prior work on generating proofs from clause-sharing portfolio solvers is limited to shared-memory parallelism and cannot be generalized to distributed memory in any obvious manner. The recent shared-memory solver Gimsatul [BF22b] is designed specifically to support outputting proofs. In terms of generic clause-sharing portfolios, Heule et al. [HMP14] attempted to generate proofs by having the solver threads emit proof lines concurrently into a single proof. Clause deletion statements can be added to the proof only after *all* solvers have reported deletion of the clause. Heule et al. obtained mixed results and for the most part were not able to arrive at proofs that are feasible to check, mostly due to the sheer size of the output and the large circumference of clauses which the checker is required to keep in memory.

2.4 Pragmatics of SAT Solving

In the following, we consider some pragmatic aspects of the development, use, and evaluation of SAT solving systems.

2.4.1 File Formats and Standards

The DIMACS CNF format (`.cnf` or `.dimacs` ending), originally proposed for a DIMACS implementation challenge in 1993 [JT96], is the standard format for SAT formulas. It is a simple text-based format that features a header line followed by one line for each clause. The header line provides the number of variables $|V|$ and clauses $|C|$ in the formula. Each clause line features a number of whitespace-separated literals terminated by a zero. Each literal is represented by a decimal number from 1 to $|V|$, or from -1 to $-|V|$ if the literal is negated.

A SAT solver takes a DIMACS CNF file and, before terminating, outputs a text line indicating whether the problem is `SATISFIABLE`, `UNSATISFIABLE`, or of `UNKNOWN` status. In case of satisfiability, the solver includes the found satisfying assignment in its output, represented by a sequence of integers. In case of unsatisfiability, if the solver is configured accordingly and supports certified SAT solving, a proof is written to a separate file and can then be verified by an independent checker. Such checker applications take the DIMACS CNF file and the proof file and output whether the proof is a correct certificate for the formula (e.g., [THM23]).

2.4.2 Evaluating SAT Solving Performance

We now discuss how the performance of sequential and parallel SAT solvers is commonly assessed and add some of our own considerations.

2.4.2.a Performance metrics

The performance of a SAT solver is usually examined by running it on a fixed set of diverse benchmark problems at a fixed (CPU or wallclock) timeout per instance. The data from such a run is then evaluated based on various metrics.

One of the most simple and popular metrics is the number of solved instances, sometimes referred to as *solved count.* Since solvers behave and perform differently on satisfiable vs. unsatisfiable problems [Oh15], it is often worthwhile to separate this metric into the number of solved satisfiable and unsatisfiable instances respectively.

Another commonly used metric is called *Penalized Average Runtime* (PAR). For any integer X, the PAR-X score of a run is defined as its arithmetic mean running time over all considered instances, where each timeout is attributed a running time of X times the time limit (e.g., a timeout at 5 000 s incurs a PAR-2 penalty of 10 000 s) [Fro+21]. The value of X weighs the solved count versus the average running time on solved instances. For instance, PAR-1 assumes that all unsolved instances are solved at the time limit, and PAR-X for $X \to \infty$ ranks solvers according to their solved count. Other commonly used PAR metrics are PAR-10 (e.g., [AC12; Mal+13; Khu+16]) and PAR-2 (e.g., [Sco+21; Fro+21; BIB22]).

PAR scores are suited to aggregate a solver's performance on many instances to a single value and thus to compare several different solver systems with each another, as is done in the International SAT Competition [Fro+21]. However, if differences between competitors are small, PAR scores can be vulnerable to noise introduced by variances in the solvers' running times. The underlying issue is that PAR-X for $X > 1$ features significant discontinuities: a minor variation in the running time on some instance can result in hitting the time limit, which incurs a penalty and therefore leads to a jump in the PAR score. This effect is amplified if a small time limit is used. In our research, we noticed this effect mostly for satisfiable instances, which are well-known to result in much larger running time variation compared to unsatisfiable instances (e.g., [OU09; Sim14; Oh15]). Averaging multiple runs of each configuration can be useful for reducing variances but can also be costly depending on the setup.

In cases where PAR scores and the sets of solved instances are sufficiently similar, we will consider an additional metric which we refer to as *Commonly Solved Average Runtime* (CSAR). Given a set of solver configurations, we identify the set of instances which *all* configurations were able to solve and then compute each configuration's arithmetic mean running time on those instances. While CSAR neglects additional solved instances, we believe that it complements PAR in a useful manner since it features significantly less noise. Note that CSAR scores, in contrast to PAR scores, cannot be compared across different sets of experiments.

2.4.2.b Speedup Metrics

For a single input $\mathcal{I}$ and p cores, the speedup of a parallel algorithm A_{par} with running time $T_{\text{par}}(\mathcal{I}, p)$ over a sequential algorithm A_{seq} with running time $T_{\text{seq}}(\mathcal{I})$ is simply computed as $s(\mathcal{I}, p) = T_{\text{seq}}(\mathcal{I})/T_{\text{par}}(\mathcal{I}, p)$ (see Section 2.1.3). The accumulation of several speedups on a given benchmark set B is more challenging. An initial idea might be to compute the *arithmetic mean of speedups* $\frac{1}{|B|}\sum_{\mathcal{I}\in B} s(\mathcal{I}, p)$ [BSS15; BS18]. This, however, is not an adequate measure. In general, arithmetic means over normalized values or ratios are meaningless [FW86]. As an example, consider two instances $\mathcal{I}_1$, $\mathcal{I}_2$ and two algorithms A, B. On instance $\mathcal{I}_1$ algorithm A is $10\times$ faster than B whereas on instance $\mathcal{I}_2$ algorithm B is $10\times$ faster than A. The arithmetic mean speedup of A over B and of B over A are exactly the same, namely $(10 + 0.1)/2 = 5.05$. The evidently incorrect conclusion is that both A and B are on average more than five times faster than the other. The statistically sound measure to compute a mean value of individual speedups is the *geometric mean* $s_{geom} := (\prod_{\mathcal{I}\in B} s(\mathcal{I}, p))^{1/n}$ [FW86]. To our knowledge, this measure is used rarely in SAT solving research.

The *median speedup* s_{med} is defined as the $\frac{|B|}{2}$-th speedup from a sorted list of speedups $s(\mathcal{I}, p), \mathcal{I} \in B$. As Balyo et al. noted, this value can be unsatisfactory if B contains many easy instances where parallelization does not pay off [BS18]. Since median speedups on such benchmark sets can also indicate the amount of overhead a parallel solver incurs, we do consider median speedups an important metric.

The *total speedup* $s_{tot} := \sum_{\mathcal{I}\in B} T_{\text{seq}}(\mathcal{I}) / \sum_{\mathcal{I}\in B} T_{\text{par}}(\mathcal{I}, p)$ is the speedup with respect to the total time spent by A_{par} vs. A_{seq} on the entire benchmark set. While this is a sensible metric with a direct and intuitive meaning, total speedups put a large emphasis on the instances which took a long time. Therefore, it should not be misinterpreted as a kind of "average speedup" or expected speedup for a single instance.

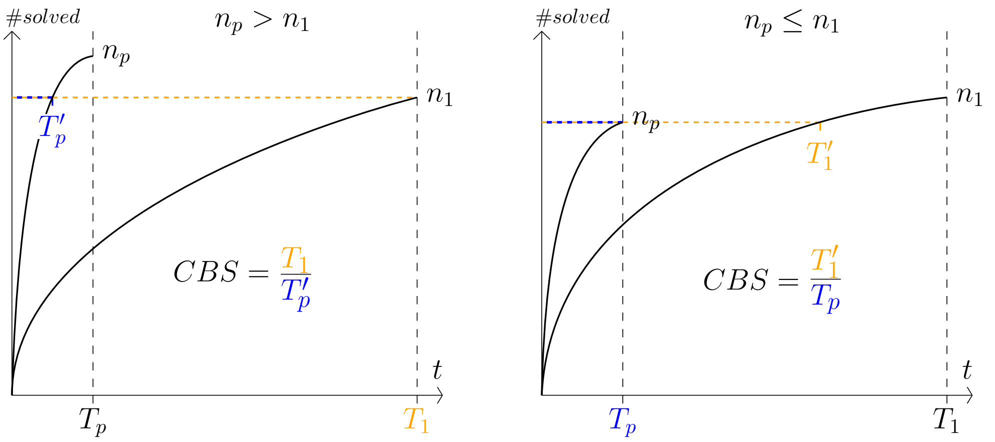


Figure 2.3: Illustration of the Count-Based Speedup (CBS) definition according to Balyo et al. [BSS15]. Assume that a parallel solver solved n_p instances within T_p seconds and a sequential baseline solved n_1 instances within T_1 seconds. The definition is split into two cases, $n_p > n_1$ (left) and $n_p \leq n_1$ (right).

The last speedup metric we discuss is the *Count Based Speedup* (CBS) by Balyo et al. [BSS15]. As illustrated in Fig. 2.3, we take the per-instance time limit T_1 (T_p) of the approach which solved fewer instances and compute the ratio between this time limit and the time limit T'_p (T'_1) required by the other approach to solve the same number of instances. The ratio is always computed in such a way that the sequential solver's time limit is divided by the parallel solver's time limit. CBS does not require to handle one-sided timeouts separately (see Section 2.4.2.c) and can be used to estimate the acceleration *any* improved approach brings over a baseline, regardless of the degree of parallelism involved. Since the set of solved instances may differ between the solvers, CBS can indirectly account for additional solved instances. A drawback of CBS is its sensitivity to outliers. As a pathological example, assume that the stronger solver solved 100 instances within 1 s each and that the weaker solver solved 99 instances within 1 s each and one instance within 1000 s. The CBS then evaluates to 1000 even though the stronger solver achieved a true speedup only on a single instance.

2.4.2.c Handling One-sided Timeouts

Orthogonal to the choice of how speedups are aggregated, another question is how to handle timeouts when calculating speedups or other instance-specific metrics. Oftentimes, a parallel solver solves more instances than a sequential solver, leading to incomplete data when trying to accumulate speedups over a given benchmark set.

One option is to only consider the instances which both the sequential solver and the parallel solver were able to solve. This neglects parts of the merit of the parallel solver but results in accurate and truthful speedup measures for the considered set.

Another option is to “generously” reinterpret each of the sequential solver’s timeouts as instances solved exactly within the time limit [BSS15]. This approach can be combined with running the sequential solver for a significantly longer (wallclock) time than the parallel solver to avoid under-approximating the sequential running times by too much. In particular, setting the sequential time limit to p times the time limit of the parallel solver at p cores results in equal CPU time budgets for both approaches.

We suspect that there are cases where some instances can be solved realistically only by the parallel solver but not by the sequential solver. This may be, e.g., due to a crucial technique in one of the portfolio’s solvers which the sequential solver is missing or because the empty clause’s derivation requires a huge set of learned clauses which a single solver cannot realistically maintain in its single database. Reporting such instances as solved after an impractically high time limit may inflate the reported speedups or render them difficult to interpret. For this reason, our preferred approach is to keep speedups and additional solved instances separate.

2.4.2.d Weak Scaling

Balyo and Sinz [BS18] suggest to transfer the idea of weak scaling (Section 2.1.3) to parallel SAT solving in the following way: If A_{par} was run with c cores, then only consider the instances which took A_{seq} at least $k \cdot c$ seconds to solve for some constant k. Evaluating the resulting speedups for different scales can give an impression on how well a solver scales to larger inputs. Large k can result in a very small number of considered instances, especially if only commonly solved instances are considered, which can add noise and amplify few large speedups (*cf.* [BSS15]). We thus suggest to consider different values of k for more robust results.

2.4.3 International SAT Competition

The *International SAT Competition* (ISC) is a research-oriented competition [Fro+21] whose first iteration took place in 1992 [BK92]. For at least 17 years, organizers have used the following statement to express the objective of the ISC:
“*The area of SAT Solving has seen tremendous progress over the last years. Many problems (e.g. in hardware and software verification) that seemed to be completely out of reach a decade ago can now be handled routinely. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for this success. To keep up the driving force in improving SAT solvers, we want to motivate implementors to present their work to a broader audience and to compare it with that of others.*”[8]

Indeed, the solvers and datasets gathered throughout the history of the ISC indicate that the mentioned “*tremendous progress*” first noted decades ago is still ongoing. Over its 25 iterations at the time of writing, the ISC tracked significant improvements in terms of SAT solving performance achieved through new algorithms, techniques, and engineering. For sequential solving, this progress is shown in Fig. 2.4 with a plot type that is popular in SAT literature and will occur several times in later chapters.

[8]See `http://fmv.jku.at/sat-race-2006` as well as `https://satcompetition.github.io/2023`

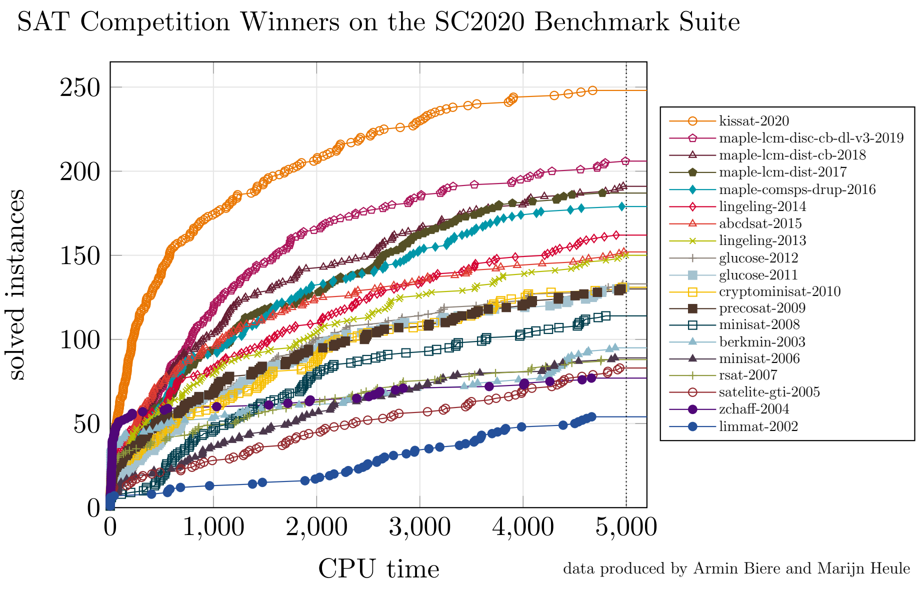


Figure 2.4: Direct comparison of winning SAT solvers from ISC 2002-2020 [Bie22]. Biere and Heule conducted the experiments on modern hardware and used the ISC 2020 benchmark set. Taken from: `http://fmv.jku.at/kissat/`

The displayed kind of plot is sometimes referred to as *inverted cactus plot* since the same plot with flipped axes is commonly dubbed *cactus plot*—allegedly due to the visual image of cactus-like arms extending from left to right.[9] Another term for inverted cactus plots is *CDF plot*: normalizing the y axis to the number of considered instances leads to a cumulative distribution function (CDF) for the empirical probability that an instance is solved within a certain time limit.

For instance, Fig. 2.4 visualizes how the solver from 2019 solved 58% more instances than the 2009 solver and was able to solve 130 instances at a per-instance time limit of 1177 s whereas the 2009 solver required a time limit of 5000 s to solve the same number of instances. Since the considered solvers' performance strongly correlates with their age and since identical hardware was used for all runs, it is evident that SAT solving has become increasingly viable not (only) due to improved hardware but rather due to algorithmic improvements and better engineering (*cf.* [FHS20]).

Initial iterations of the ISC were restricted to sequential SAT solving. As far as we are aware, the first ISC featuring a parallel track was the 2010 iteration where each parallel solver was executed on eight cores. Ten years later, distributed SAT solving

[9]We have not found any original source for this term. Cactus plots have been used by the SAT community at least since 2002 [SLH05]. In 2009 Audemard and Simon referenced "*the classical 'cactus' plot used in SAT competitions*" [AS09].

was introduced to the ISC in the form of a so-called *cloud track* sponsored by Amazon Web Services [Fro+21]. Each submission to this track is executed on 100 machines at once with 16 hardware threads each at a timeout of 1000 s per formula. Later iterations of the ISC in 2021 and 2022 maintained this track as well as the parallel track that is evaluated on a single machine with 64 hardware threads at a timeout of 5000 s per formula. Across all tracks, the PAR-2 metric (see Section 2.4.2.a) is used to rank solver submissions [Fro+21].

The set of ISC benchmark instances are carefully selected each year based on new submissions from participants as well as a representative mix of instances from previous years [Fro+21]. The ISC benchmark sets have therefore become a crucial resource for researching and evaluating SAT techniques [AS09; BSS15; Lia+16a; NR18; Val+20a]. That being said, it is also important to acknowledge that these benchmarks are not without bias: ISC instances are filtered[10] and selected [Bal+15; Fro+21] based on how well certain prior solvers perform on them, and solvers are in turn tuned on ISC instances. This co-evolution bears a risk of overfitting (*cf.* [BH19]).

[10]For example, Manthey states in a 2023 benchmark description: "*to not move further into a* KISSAT *solver mono culture, we filtered the generated formulas and dropped the ones that could be solved easily with* KISSAT" [Man23].

Chapter 3

3

Decentralized Scheduling of Malleable NP-hard Tasks

In this chapter, we address an online job scheduling problem in a large distributed computing environment. Each job has a priority and a demand of resources, takes an unknown amount of time, and is malleable, *i.e., the number of allotted workers can fluctuate during its execution. We subdivide the problem into (a) determining a fair amount of resources for each job and (b) assigning each job to an according number of processing elements. Our approach is fully decentralized, uses lightweight communication, and arranges each job as a binary tree of workers which can grow and shrink as necessary. With SAT solving as an application, we experimentally show on up to 128 machines (6144 cores) that our approach leads to near-optimal utilization, imposes minimal computational overhead, and performs fair scheduling of incoming jobs within a few milliseconds.*

Author's Notes. *This chapter is based on "*Decentralized Online Scheduling of Malleable NP-hard Jobs*" [SS22a], a publication by Peter Sanders and myself. Large parts of this chapter are copied verbatim or with minor changes from that publication. Peter Sanders and I have devised the proposed algorithms together. In particular, the concept and initial wording of the logarithmic-span volume calculation (refining my earlier and more naïve algorithm) are due to Peter Sanders. I, in turn, refined Peter Sanders' algorithm and authored the vast majority of the remaining content featured in this chapter, with editing by Peter Sanders. This chapter features some added content, in particular experiments on prefix sum based request matching (Section 3.6.4.b) and a more in-depth presentation of our system (Section 3.5). The latter section also features an overview of the case studies we performed (Section 3.5.3), briefly summarizing several qualification theses I (co-)supervised [Dör22; Sch21a; Sön21; Wil22].*

3.1 Introduction

A parallel task is called *malleable* if it can handle a fluctuating number of workers during its execution [Fei97]. Malleability has long been recognized as a powerful paradigm which opens up vast possibilities for fair and flexible scheduling and load balancing in parallel and distributed systems [Hun04; Gup+14]. Most previous research on malleable job scheduling focuses on iterative data-driven applications with a regular kind of parallelization and with near-linear scaling behavior [DEV07; SS12; Gup+14].

In this work, we consider malleability in a different context, namely for NP-hard tasks with unknown processing times—such as instances of the SAT problem (Section 2.2).

Inflexible scheduling policies, which assign a fixed amount of resources to each incoming job, have limited means to react to unpredictable changes in the system such as a job finishing earlier or taking much longer than anticipated. For SAT tasks, where even predicting *sequential* running times is a challenging machine learning problem [Hut+14], inflexible scheduling of *parallel* tasks likely results in suboptimal utilization of resources and in poor load balancing. We consider malleable scheduling of parallel SAT tasks much more promising: The description of a job can be relatively small even for very difficult problems, and the successful *portfolio approach* where many orthogonal search strategies compete in parallel (Section 2.3.2) can be made malleable easily without any redistribution of data. Moreover, the limited scalability of parallel SAT solving calls for carefully distributing the computational resources at hand. To this end, processing several jobs in parallel can help to make more efficient use of a large amount of computational power. Specifically, we believe that an on-demand service platform for NP-hard problems has the potential to drastically improve efficiency and productivity for many organizations and environments. Using malleable job scheduling, we can schedule new jobs within few milliseconds, resolve trivial jobs in a fraction of second, and rapidly resize more difficult jobs to a fair share of all resources—as far as the job can make efficient use of these resources.

To meet these objectives, we propose a fully decentralized scheduling approach which guarantees fast, fair, and bottleneck-free scheduling of resources without prior knowledge on processing times. We address two subproblems. The first problem is to let m workers compute a fair number of workers v_j for each active job j, accounting for its priority and maximum demand, while optimizing system utilization. The second problem is to assign v_j workers to each job j while keeping the assignment as stable as possible over time. For both problems, we outline fully scalable algorithms with $\mathcal{O}(\log m)$ span which consistently result in optimal utilization. In particular, we represent each parallel job as a binary tree of v_j processes which can grow and shrink at its leaf level on demand. Furthermore, we introduce measures to have these job trees preferably re-grow in such a way that they reuse existing (suspended) workers for this job rather than initializing new workers.

We present our decentralized scheduling platform Mallob which features practical implementations of our scheduling approaches and several application case studies. Experiments on up to 128 nodes (6144 cores), using SAT as an application problem, show that our system leads to near-optimal utilization and schedules jobs with a fair share of resources within tens of milliseconds. We consider our theoretical as well as practical results to be promising contributions towards processing SAT and other malleable NP-hard tasks in a more scalable and resource-efficient manner.

This chapter is structured as follows. We begin with some foundations in Section 3.2 and a problem statement in Section 3.3. We then describe our algorithmic approach in Section 3.4. We provide an overview of our scheduling system Mallob, including the applications it supports, in Section 3.5 and evaluate our work in Section 3.6. Section 3.7 concludes the chapter and outlines future work.

3.2 Foundations

We now discuss foundations and related work in terms of (malleable) job scheduling in the context of our work. For a comprehensive introduction to job scheduling, we refer to the *Handbook on Scheduling* [Bla+19].

We use the following definitions by Feitelson [Fei97]: A *rigid* task requires a fixed number of workers. A *moldable* task can be scaled to a number of workers at the time of its scheduling but then remains rigid. Finally, a *malleable* task is able to adapt to a fluctuating number of workers *during* its execution. Many parallel algorithms are moldable (e.g., most parallel SAT solvers described in Section 2.3) while relatively few parallel algorithms are malleable [Gup+14].

Malleability can be a highly desirable property of tasks because it allows to balance tasks continuously to warrant fair and optimal utilization of the system at hand [Hun04]. For example, if a small job arrives in a fully utilized system with many large jobs, malleable scheduling allows to shrink active jobs in order to schedule the new job immediately, substantially decreasing its response time. Due to the appeal of malleable job scheduling, there has been ongoing research to exploit malleability, from shared-memory systems [Gup+14] to HPC environments [Bui+07; DEV07], also as a means to improve energy efficiency [SS12].

Prior malleable job scheduling approaches mostly rely on prior knowledge on the processing times of jobs and on an accurate model for their execution time relative to the degree of parallelism [Bla+04; Bla+06; SS11]. In many cases, these approaches consider an *offline* scheduling problem where the number, arrival times, and properties of jobs are known in advance [Bla+04; SS12]. By contrast, we address an *online* scheduling problem, where the system has to react to previously unknown jobs arriving at unknown times. Our approach also has no prior knowledge of a job's execution time and it only assumes that the execution time of a job j generally decreases when scaling up j, up to some job-specific resource limit d_j.

Furthermore, while most approaches employ a centralized scheduler [Hun04; Bui+07; DEV07], which implies a potential bottleneck and a single point of failure, our approach is fully decentralized and uses a small part of each processes' CPU time to perform distributed scheduling, which also opens up the possibility to add fault tolerance to our work in the future. For instance, this may include continuing to schedule and process jobs even in case of network-partitioning faults [Alq+18], i.e., failures where sub-networks in the computing environment are disconnected from each another.

The effort required to transform a moldable (or rigid) algorithm into a malleable algorithm depends on the application at hand. For iterative data-driven applications, redistribution of data is necessary if a task is expanded or shrunk [DEV07]. In contrast, basic malleability is simple to achieve if the parallel algorithm is composed of independent search strategies which compete for finding a solution: The abrupt suspension or termination of individual workers can imply the loss of progress, but preserves completeness. Moreover, if workers exchange knowledge—as is the case for clause-sharing solvers (Section 2.3.2.b)—the progress made on a worker may benefit the job even after the worker has been removed from the distributed computation.

We thus do not need to migrate application processes across machines (*cf.* [HLK03; DEV07]) to make clause-sharing portfolio SAT solving malleable. In order to account for applications beyond SAT, we do allow a malleable computation to react in any way to workers being added or removed, such as redistributing data or re-negotiating the task's work subdivision. In future work, we may also combine our approach with the recently investigated migration of SAT solver states across machines [Bie+22].

3.3 Problem Statement

As described in Section 2.1, we consider a distributed environment with M interconnected, identical machines running a total of $m \geq M$ processes. Each process has a *rank* $x \in \{0, \dots, m-1\}$ and runs exclusively on $c \geq 1$ cores of its local machine. Processes exchange information via message passing.

Jobs are introduced over an interface connecting to some of the processes. Each job j has a job description, a *priority* $p_j \in \mathbb{R}^+$, a *demand* $d_j \in \mathbb{N}^+$, and a *budget* b_j (in terms of wallclock time or CPU time). If a process participates in processing a job j, it runs an execution environment of j named a *worker*. A job's demand d_j indicates the maximum number of parallel workers it can currently employ: d_j is initialized to 1 and can then be adjusted by the job after an initial worker has been scheduled. A job's priority p_j may be set, e.g., depending on who submitted j and on how important they deem j relative to an average job of theirs. In a simple setting where all jobs are equally important, assume $p_j = 1 \; \forall j$. A job is cancelled if it spends its budget b_j before finishing. We assume for the active jobs J in the system that the number $n = |J|$ of active jobs is no higher than m and that each process employs at most one worker at any given time. However, a process can preempt its current worker, run a worker of another job, and possibly resume the former worker at a later point.

Let T_j be the set of active workers of $j \in J$. We call $v_j := |T_j|$ the *volume* of j. Our aim is to continuously assign each $j \in J$ to a set T_j of processes subject to:

(C1) (*Optimal utilization*) Either all job demands are fully met or all m processes are occupied: $(\forall j \in J : v_j = d_j) \;\vee\; \sum_{j \in J} v_j = m$.

(C2) (*Individual job constraints*) Each job must have at least one worker and is limited to d_j workers: $\forall j \in J : 1 \leq v_j \leq d_j$.

(C3) (*Fairness*) Resources allotted to each job j scale proportionally with p_j except if prevented by C2:
For each $j, j' \in J$ with $p_j \geq p_{j'}$, there are *fair assignments* $\omega, \omega' \in \mathbb{R}^+$ with $\omega/\omega' = p_j/p_{j'}$ and some $0 \leq \varepsilon \leq 1$ such that $v_j = \min(d_j, \max(1, \lfloor \omega + \varepsilon \rfloor))$ and $v_{j'} = \min(d_{j'}, \max(1, \lfloor \omega' \rfloor))$.

Due to rounding, in C3 we allow for job volumes to deviate by a single unit (see $\varepsilon \leq 1$) from a fair distribution as long as the job of higher priority is favored.

3.4 Approach

In the following, we present algorithms which address two distinct subproblems: First, find *fair volumes* v_j for all currently active jobs $j \in J$ subject to C1–C3. Secondly, identify pairwise disjoint sets of processes T_j with $|T_j| = v_j$ for each $j \in J$.

3.4.1 Calculation of Fair Volumes

Given jobs J with individual priorities and demands, we want to find a fair volume v_j for each job j such that constraints C1–C3 are met. Volumes are recomputed periodically taking into account new jobs, departing jobs, and changed demands. In the following, assume that each job has a single worker which represents this (and only this) job. We elaborate on these representants in Section 3.4.2.

First, we show that algorithms for our volume calculation problem may in principle scale arbitrarily well. We then present a practically efficient algorithm for our purposes.

3.4.1.a Asymptotic Upper Bound

We now prove that the volume calculation problem can be solved with $\mathcal{O}(\log m)$ span, which is asymptotically as expensive as a single collective operation such as, e.g., an all-reduction of $\mathcal{O}(1)$ data.

Theorem 3.1

Given $n \leq m$ jobs J, volumes v_j for each job $j \in J$ which meet constraints C1–C3 can be computed on m processes in $\mathcal{O}(\log m)$ span and $\mathcal{O}(m \log m)$ work.

Proof. The sum of available demands, $\sum_{j \in J} d_j$ can be obtained easily with a single all-reduction. We assume that this collective demand exceeds the available resources, i.e., $\sum_{j \in J} d_j > m$, since otherwise we can trivially set $v_j = d_j$ for all jobs j. Assuming real-valued job volumes for now, we can observe that for any parameter $\alpha \geq 0$, constraints C2–C3 are fulfilled if we set $v_j = v_j(\alpha) := \max(1, \min(d_j, \alpha p_j))$. By appropriately choosing α, we can also meet the utilization constraint C1: Consider the function

$$\xi(\alpha) \quad := \quad m - \sum_{j \in J} v_j(\alpha) \quad = \quad m - \sum_{j \in J} \max(1, \min(d_j, \alpha p_j)) \tag{3.1}$$

which expresses the resources left unused for a particular value of α. Function ξ is continuous, monotonically decreasing, and piece-wise linear —see Fig. 3.1 for an example. Moreover, $\xi(0) = m - n \geq 0$ and $\xi(\max_{j \in J} d_j / p_j) = m - \sum_{j \in J} d_j < 0$. Hence $\xi(\alpha) = 0$ has a solution α_0 which represents the desired choice of α that exploits all resources and thus satisfies constraint C1.

Assuming that we know α_0, we only need to round each $v_j(\alpha_0)$ to an integer. Due to C1 and C3, we propose to round down all volumes and then increment the volume of the $k := m - \sum_j \lfloor v_j(\alpha_0) \rfloor$ jobs of highest priority. To this end, we sort all jobs with some remaining demand, $J' := \{j \in J : v_j(\alpha_0) < d_j\}$, by job priority and compute k with an all-reduction. We can then select the first k jobs among the sorted jobs.

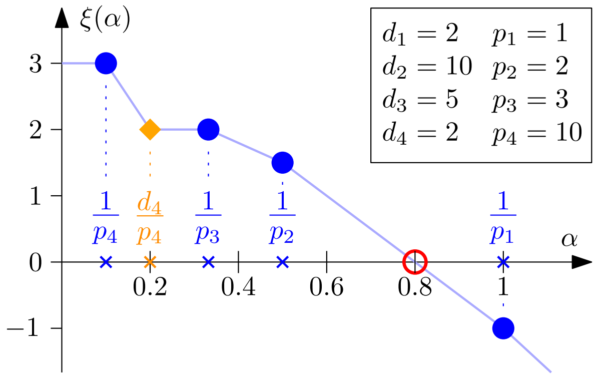


Figure 3.1: Volume calculation example with four jobs and $m = 7$. Five of the eight points where $\xi(\alpha)$ is evaluated are depicted, three more (d_3/p_3, d_1/p_1, and d_2/p_2) are omitted. The red circle marks $\alpha_0 = 0.8$.

We now outline how to find α_0 in $\mathcal{O}(\log m)$ span. We exploit that ξ', the derivative of ξ, changes at no more than $2n$ values of α, namely when $\alpha p_j = 1$ (blue spheres in Fig. 3.1) or when $\alpha p_j = d_j$ (orange diamond in Fig. 3.1) for some $j \in J$. Since we have $m \geq n$ processes, we can evaluate all these $\mathcal{O}(n)$ values of $\xi(\alpha)$ in parallel. We then find the two points with smallest positive value and largest negative value using another all-reduction. Lastly, we interpolate ξ between these points to find α_0.

In the example in Fig. 3.1, we find $\alpha_0 = 0.8$ in the interval $[1/p_2, 1/p_1] = [0.5, 1]$. Since $\alpha_0 > d_4/p_4$ we cap job 4 at its demand ($v_4 = 2$), and since $\alpha_0 < 1/p_1$ we raise the volume of job 1 to $v_1 = 1$. The real-valued shares $\alpha_0 p_2 = 1.6$ and $\alpha_0 p_3 = 2.4$ are rounded to $v_2 = 1$ and $v_3 = 3$ as job 3 has the higher priority.

The parallel evaluation of ξ is still nontrivial since a naïve implementation would incur quadratic work—$\mathcal{O}(n)$ for each value of α. We now explain how to accelerate the evaluation of ξ. For this, we rewrite $\xi(\alpha) = m - \sum_{j \in J} v_j(\alpha)$ as:

$$\xi(\alpha) = m - \underbrace{\left(\sum_{j : \alpha p_j < 1} 1 + \sum_{j : \alpha p_j > d_j} d_j \right)}_{R} - \alpha \underbrace{\sum_{j : 1 \leq \alpha p_j \leq d_j} p_j}_{P} \tag{3.2}$$

Intuitively, R sums up all resources which are assigned due to raising a job volume to 1 (if $\alpha p_j < 1$) and due to capping a job volume at d_j (if $\alpha p_j > d_j$); and αP sums up all resources assigned as $v_j = \alpha p_j$ (if $1 \leq \alpha p_j \leq d_j$).

This new representation only features two unknown variables, R and P, which can be computed efficiently. At $\alpha = 0$, we have $R = n$ and $P = 0$ since all job volumes are raised to one. If we then successively increase α, we pass $2n$ *events* where R and P are modified, namely whenever $\alpha p_j = 1$ or $\alpha p_j = d_j$ for some job j. Since each such event modifies R and P by a fixed amount, we can use a single prefix sum calculation to obtain *all* intermediate values of R and P.

We define an event as a triple $e = (\alpha_e, r_e, p_e)$ which represents that e occurs at point α_e and adds r_e to R and p_e to P. Each job j causes two events: $\underline{e}_j = (1/p_j, -1, p_j)$ for the point $\alpha p_j = 1$ where v_j stops being raised to 1, and $\overline{e}_j = (d_j/p_j, d_j, -p_j)$ for the point $\alpha p_j = d_j$ where v_j begins to be capped at d_j. We sort all events by α_e and then compute a prefix sum over r_e and p_e: $(R_e, P_e) = (\sum_{e' \leq e} r_{e'}, \sum_{e' \leq e} p_{e'})$, where "$<$" denotes the ordering of events after sorting. We can now compute $\xi(\alpha_e) = m - (n + R_e) - \alpha_e P_e$ at each event e.[1] The value of n can be obtained with an all-reduction.

Computing all-reductions and prefix sums of $\mathcal{O}(1)$ data is possible in logarithmic time. We also need to sort $2n$ job events and later, for correct rounding, $\mathcal{O}(n)$ jobs with remaining demand, which is possible in logarithmic time as well using $m \geq n$ processes.[2] This results in $\mathcal{O}(\log m)$ span and $\mathcal{O}(m \log m)$ work. □

3.4.1.b Practical Implementation

For our practical implementation, we compute job volumes similar to the algorithm outlined in the previous proof. However, each process computes the desired change of root α_0 of ξ locally. All events in the system (job arrivals, departures, and changes in demands) are aggregated and broadcast periodically such that each process can maintain a local image of all active jobs' demands and priorities. The local search for α_0 is then done via bisection over the domain of ξ. This approach requires a single all-reduction of worst-case message length $\mathcal{O}(n)$ followed by a computation of complexity $\mathcal{O}(n \log n)$ on each worker. Therefore, in the worst case our approach incurs $\mathcal{O}(m \cdot n \log n) = \mathcal{O}(m^2 \log m)$ work and has span $\mathcal{O}(n \log m + n \log n) = \mathcal{O}(m \log m)$. Compared to our theoretical fully scalable algorithm, we cut down the required communication to a single sparse all-reduction at the cost of requiring more complex local computations. We expect this to be a worthwhile trade-off, especially at the targeted scale of our current implementation ($n \leq 10^3$ and $m \leq 10^4$) and for the case of $n \ll m$. When targeting much larger configurations in the future, it may be beneficial to implement a variant of our theoretical algorithm instead.

3.4.2 Assignment of Jobs to Processes

We now describe how the fair volumes computed as in the previous section translate to an actual assignment of jobs to processes. First, we outline how each malleable job is structured in our distributed system. Then we propose three different approaches to match requests for new workers with idle processes. Finally, we discuss adjustments to our strategies which result in an increased reuse of suspended workers.

[1] If there are multiple events at the same α, their prefix sum results can differ but will still result in the same $\xi(\alpha)$. This is due to the continuous nature of ξ: Note how each event modifies the gradient $\xi'(\alpha)$ but preserves the value of $\xi(\alpha)$.

[2] Asymptotically optimal sorting on communication networks [AKS83] is of mostly theoretical value due to the large constant values involved. However there are quite practical algorithms when $n \in \mathcal{O}(\sqrt{m})$ or when spending $\mathcal{O}(\log^2 n)$ time is acceptable [AS17b].

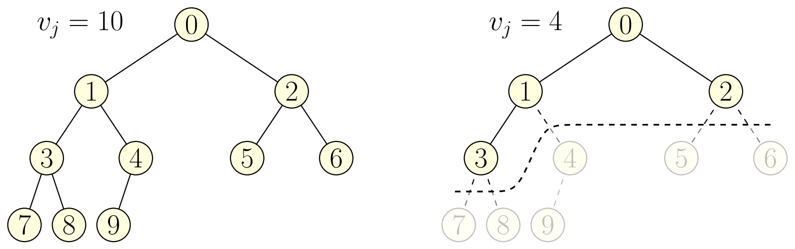


Figure 3.2: Left: Job tree T_j features ten workers $\{w_j^0, w_j^1, \ldots, w_j^9\}$ due to the volume $v_j = 10$ assigned to j. Right: Volume update $v_j = 4$ arrives. Consequently, all workers with index ≥ 4 are suspended and the corresponding processes can run other workers instead.

3.4.2.a Distributed Structure of Jobs

For each job j, we address the k current workers in T_j as $w_j^0, w_j^1, \ldots, w_j^{k-1}$. These workers can be scattered throughout the system—their *job indices* $0, \ldots, k-1$ within T_j are not to be confused with their process ranks. The k workers form a communication structure in the shape of a binary tree (Fig. 3.2) which can also be used conveniently for job-internal communication—such as the periodic exchange of clauses for the case of SAT solving. Worker w_j^0 is the root of this tree and represents j for the calculation of its volume (Section 3.4.1). Workers w_j^{2i+1} and w_j^{2i+2} are the left and right children of w_j^i. Jobs are made malleable by letting T_j grow and shrink dynamically. Specifically, we enforce that T_j consists of exactly $k = v_j$ workers. If v_j is updated, all workers w_j^i for which $i \geq v_j$ are suspended and the corresponding processes turn idle. Likewise, workers without a left (right) child for which $2i + 1 < v_j$ ($2i + 2 < v_j$) attempt to find a child worker w_j^{2i+1} (w_j^{2i+2}). We find new workers for a job via *request messages*: A request message $r_j^i = (j, i, x)$ holds index i of the requested worker w_j^i as well as rank x of the requesting worker. If a *new* job is introduced at some process, then this process emits a request for the root node w_j^0 of T_j. All requests for w_j^i, $i > 0$ are emitted by the designated parent node $w_j^{\lfloor (i-1)/2 \rfloor}$ of the desired worker.

3.4.2.b Matching through Random Walks

We now outline a first approach to route request messages to idle workers. Consider a directed graph $G = (V, E)$ where V is the set of processes in the system and E is a set of communication channels between them. If a request message is spawned, it performs a random walk through G and is resolved as soon as it hits an idle process. We choose E in such a way that G is a *regular graph*, where each vertex has k outgoing edges for a constant k. Specifically, we generate E in a communication-free manner on the basis of k pseudo-random permutations of $\{1, \ldots, m\}$. As a special case, the very first hop of a request r_j^i is not bound to G but can be any process in the system: It is determined by the i-th item of a pseudo-random permutation π_j of $\{1, \ldots, m\}$.

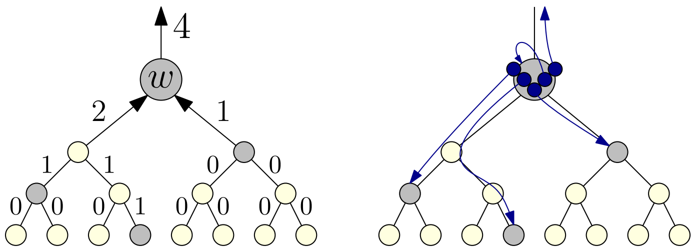


Figure 3.3: Routing tree matching strategy. Left: Each subtree of processes aggregates the number of idle processes (gray) to its root. Process w receives $2 + 1$ idle counts and is idle itself, hence it reports $2 + 1 + 1 = 4$ to its parent. Right: w has received five request messages. It matches one of them with itself and sends two requests to its left subtree and one request to its right subtree. The remaining request is routed upwards to be handled recursively.

π_j is generated using the ID j as a seed. Deciding on each request's first destination in this special manner has two consequences: First, each request begins its random walk in an environment of G that is independent from its parent process. Secondly, if a single job enters an empty system, then each request immediately arrives at an idle process without any need for random walks.

Our rationale for this approach based on random walks is that a logarithmic number of hops will result in a roughly uniform probability distribution over the location which the request is currently visiting [CF05]. The work needed to let a request perform a hop is constant and no synchronization is required. However, some requests can require a large number of hops until they are matched. If we assume as a simplification that G is fully connected ($k = m - 1$) and if only a small share ε of workers is idle, then each hop of a request corresponds to a Bernoulli process with success probability ε, and a request takes an expected $1/\varepsilon$ hops until an idle process is hit. In particular, this results in expected linear work to match a single request with a single idle process in an otherwise busy system—a realistic scenario when scheduling a new job. In order to mitigate this worst-case latency, we can configure our volume calculation to target a system utilization smaller than 1 such that a small ratio of workers is kept idle.

3.4.2.c Matching through Routing Tree

In contrast to our basic approach, our next algorithm does not depend on suboptimal utilization. After sending each request r_j^i to its natural point of departure $\pi_j[i]$, we deliberately route each surviving request *towards* idle processes. To achieve this, we use a k-ary *routing tree* R of processes (for a small constant k) to route requests as well as information on idle processes. If a process becomes idle or occupied, it reports this change to its parent in R. Each inner node in R maintains the *idle count*, i.e., the number of idle processes, in each of its subtrees (see Fig. 3.3 left). A process receiving a request performs the first possible action among the following options:

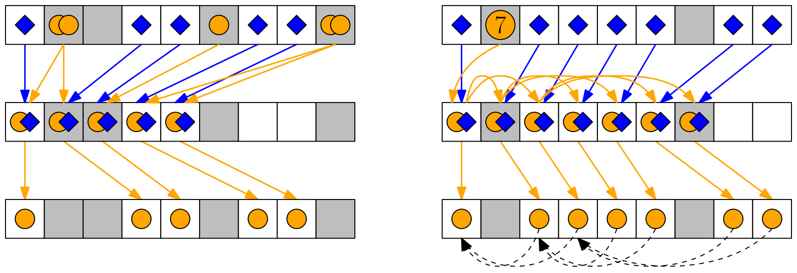


Figure 3.4: Examples for matching requests and idle processes. White (gray) squares represent idle (busy) processes, spheres (diamonds) represent requests (idle tokens). Left: A prefix sum (not depicted) numbers all requests and idle tokens, and each request (token) of index i is sent to rank i. Each process with a matching pair sends the request to the idle process. Right: A job j grows by multiple layers of T_j. Requests are sent along a tree structure and child-parent relationships of T_j are encoded into the distributed requests.

1. *Match the request with yourself (if you are idle).*
2. *Send the request to a subtree with non-zero idle count.*
3. *Send the request to your parent.*

Fig. 3.3 (right) shows an example where all of these actions are performed. Intuitively, most requests will be resolved in a small local environment of the starting point. The number of requests versus the number of idle processes may still be imbalanced across neighboring subtrees, in which case the imbalanced requests need to be routed over their common parent process. In particular, the traffic along the root r of R to offset this imbalance may constitute a bottleneck if a large number of requests are emitted at once. In the worst case, where one of the k subtrees of r has no idle processes but $m - m/k$ requests for *all* other processes, all these $\mathcal{O}(m)$ requests need to be transferred along r. In a simplified probabilistic framework, we can consider the k subtrees as *bins* and the number x of requests as *balls* for a *balls-into-bins* analysis, which tells us for $x \gg k$ that the most heavily loaded subtree has $x/k + \Theta(\sqrt{x \log k/k})$ requests with high probability [RS98; Ber+00]. Since that subtree has an expected number of x/k idle processes, it needs to route an expected $\mathcal{O}(\sqrt{x})$ requests over r.

3.4.2.d Matching through Prefix Sums

Our third and final algorithm is not randomized but matches processes and requests in a regular manner using prefix sums. Fig. 3.4 illustrates this algorithm. We regard our system as an array of processes where each process is either busy or idle and where each process can emit a number of requests. We first assume a synchronized procedure where all requests are emitted at the same point in time and need to be matched with the processes that are currently idle. Afterwards we relax this assumption and outline a fully asynchronous variant of the procedure.

In a first phase, our algorithm computes two prefix sums with one collective operation: the number of requests q_i being emitted by processes of rank $< i$, and the number o_i of idle processes of rank $< i$. We also broadcast the total sums, q_m and o_m, to all processes. The q_i and o_i provide an implicit global numbering of all requests and all idle processes. In a second phase, the i-th request and the i-th token are both sent to rank i. In the third and final phase, each process which received both a request and an idle token sends the request to the idle process referenced by the token.

If we assume that each request for a worker $w_j^i, i > 0$ originates from its designated parent worker $w_j^{\lfloor (i-1)/2 \rfloor}$, then our algorithm so far may need to be repeated $\mathcal{O}(\log m)$ times to account for a single volume update: In repetition l workers are initialized which need to wait for repetition $l + 1$ to emit requests themselves. As such, each repetition only allows to grow a job tree by a single layer. Alternatively, we can let a worker emit requests not only for its direct children, but for all *transitive* children it desires. Each worker w_j^i can compute the number k of desired transitive children from v_j and i. The worker's process then contributes value k to the prefix sum q_i. In the second phase, the k requests can be distributed communication-efficiently to a range of ranks $\{x, \ldots, x + k - 1\}$: w_j^i sends requests for workers w_j^{2i+1} and w_j^{2i+2} to ranks x and $x + 1$, which send requests for corresponding child workers to ranks $x + 2$ through $x + 5$, and so on, until worker index $v_j - 1$ is reached. To enable this distribution, we append to each request the values x, v_j, and the rank of the process where the respective parent worker will be initialized. As such, each child knows its parent within T_j (Fig. 3.4) for job-internal communication.

We now outline a fully asynchronous version of our algorithm. We compute prefix sums within an *In-Order* binary tree of processes [San+19, Sect. 13.3], i.e., all children in the left subtree of rank i have a rank $< i$ and all children in the right subtree have a rank $> i$. This computation can be made sparse and asynchronous: Only non-zero contributions to a prefix sum are sent upwards, and there is a minimum delay in between sending contributions to a parent. We extend our prefix sums to also include *inclusive* prefix sums q_i', o_i' which denote the number of requests (tokens) at processes of rank $\leq i$. Every process can see from the difference $q_i' - q_i$ $(o_i' - o_i)$ how many of its local requests (tokens) took part in the prefix sum. The number of tokens and the number of requests may not always match—a process which receives either a request or an idle token (but not both) knows of this imbalance due to the total sums q_m, o_m. The unmatched message then can re-participate in the next iteration.

Our matching algorithm has $\mathcal{O}(\log m)$ span and takes $\mathcal{O}(m)$ local work. The maximum local work of any given process is in $\mathcal{O}(\log m)$ (to compute the above k), which is amortized by other processes because at most m requests are emitted.

3.4.3 Reuse of Suspended Workers

Each process remembers a small constant of most recently used workers before deleting them, since a suspended worker may be resumed at a later time. Our scheduling so far is unaware of suspended workers, which are therefore only reused in lucky circumstances. We now outline how we can increase the reuse of suspended workers.

3.4.3.a Basic Approach

In our first approach, each worker remembers a limited number of ranks of its past (direct) children. A worker which desires a child queries them for reactivation one after the other until success or until all past children have been queried unsuccessfully. In the latter case, a normal job request is emitted. A process on the receiving side may also respond that the concerned worker is not present any more, which prompts the requesting worker to forget this past child.

3.4.3.b Improved Approach

We made two improvements to our basic strategy. First, we remember past workers in a distributed fashion. More precisely, whenever a worker joins or leaves T_j, we distribute information along T_j to maintain the following invariant: Each current leaf w_j^i in T_j remembers the past workers which were located in a subtree below index i. As such, past workers can be remembered and reused even if T_j shrinks by multiple layers and needs to re-grow differently at some point.

Secondly, we adjust our scheduling to actively prioritize the reuse of existing workers over the initialization of new workers. In our implementation, each idle process can infer from its local volume calculation (Section 3.4.1) which of its local suspended workers w_j^i are *eligible for reuse*, i.e., $v_j > i$ in the current volume assignment. If a process has such a suspended worker w_j^i, the process will reject any job requests until it received a message regarding w_j^i. This message is either a query to resume w_j^i or a notification that w_j^i will not be reused. On the opposite side, a worker which desires a child begins to query past children according to a "most recently used" strategy. If a query succeeds, all remaining past children are notified that they will not be reused. If all queries failed, a normal job request is emitted.

3.5 The Mallob System

In the following, we outline the design and implementation of our platform named Mallob, short for **Mal**leable **Lo**ad **B**alancer. We also describe the different applications which we explored to be scheduled within Mallob.

3.5.1 Overview

Mallob is an application written in C++ 17. It has been developed over the course of four years and currently features roughly 30 000 lines of code.[3] Using the Message Passing Interface (MPI) [GLS99], Mallob can be deployed on a single machine or on many interconnected machines at the same time. It features an API which applications can be connected to and then allows to schedule and process tasks from

[3]As of September 2023; excluding comments, blank lines, and external libraries/tools. Mallob's core accounts for 16.5 kLOC and our SAT solving engine MallobSat accounts for 13.3 kLOC.

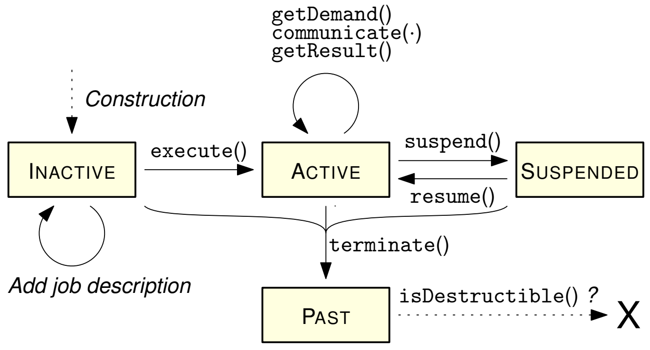


Figure 3.5: Simplified overview of the life cycle of a worker.

these applications on demand. There are no dedicated processes or machines for the scheduling of jobs—every process can be configured to take job submissions from an external source and to participate in decentralized scheduling negotiations.

Within each (MPI) process of MALLOB, the program flow is structured as follows. A single *main thread* frequently loops through a number of tasks, namely (i) polling for incoming messages and dispatching them to appropriate modules; (ii) checking progress in the active worker (if present), the scheduling and balancing components, and any open job interfaces; and (iii) sending messages which these different actors wish to emit. As such, only the main thread of an MPI process issues MPI calls, which adheres to the most widely supported mode of operation for multithreaded MPI programs [Vai+15]. As we aim for scheduling latencies in the range of milliseconds, each process must frequently check for incoming messages. For instance, if a main thread copies a large job description, this can cause a prohibitively long period where no messages are processed. For this reason, we use a separate thread pool for all tasks which involve a risk of taking a long time.

The application-specific workers running on each process are defined via an API with a small set of methods. These methods define a worker's behavior if it is started, suspended, resumed, or terminated, and allow it to send and receive application-specific messages. An important contract with applications in MALLOB is that all of these interface methods return quickly and that none of them blocks the calling main thread for a significant period (i.e., a few milliseconds). Instead, a worker is required to use separate threads or subprocesses for the actual job processing and any potentially costly initialization or destruction procedures.

Fig. 3.5 shows an overview of a worker's life cycle. Initially, a worker is INACTIVE and devoid of any job-specific information. After adding a job description to the worker, `execute()` transitions the worker into an ACTIVE state. The process main thread now frequently queries the worker whether a result is present and whether it would like to communicate. If the worker represents the job's root node, it is also queried periodically for how many workers the job currently desires (`getDemand()`).

At any time, an active worker may be suspended and a suspended worker may be resumed, prompting the application to suspend or resume its worker threads respectively. After a worker is terminated, the process main thread repeatedly queries if the worker can be cleaned up (`isDestructible()`). If this is not the case, e.g., because it still needs to forward some information to another worker, then the process main thread also progresses the worker's communication while performing this check.

3.5.2 Communication

We now elaborate on the implementation of communication protocols in Mallob. Our system exclusively features asynchronous communication, i.e., a process will never block when sending or receiving messages. As a result, our protocols are designed without explicit synchronization (barriers or similar) and only use point-to-point message passing (`MPI_Isend` and `MPI_Irecv`), with few exceptions.

Each message in MPI is associated with a certain (integer) *message tag* which indicates the meaning of the message and allows its matching with an open *request handle* on the receiver side. Mallob abstracts away the maintenance of such request handles via a simplified *subscription system*: Any module of Mallob may subscribe to a specific message tag by providing a callback function which the main thread calls whenever a message of this tag arrives. This abstraction greatly simplifies asynchronous message passing compared to a direct use of MPI functions. Furthermore, Mallob splits large messages into batches of smaller messages, e.g., when transferring large job descriptions to new workers, in order not to occupy the main thread for too long with handling a single message. The message is re-assembled concurrently at the receiving process and then digested normally.

As an example for our communication protocols, we provide a flow diagram for a new worker joining a job in Fig. 3.6. If a job request arrives (via a message of tag `MSG_REQUEST_WORKER`), the receiving process will check whether it is eligible for adopting the request. Reasons against this may be an active worker at this process or an ongoing commitment with respect to another request. If the process is able to adopt the request, it *commits* to the request and offers its adoption to the requesting process. While a process is in a committed state, any other requests are rejected until the commitment is resolved one way or another. In the displayed case, the requesting process finds that its request is still relevant and answers the adoption offer affirmatively, prompting the accepting process to query the required job description and finally begin to execute the worker. There are some alternative cases which Fig. 3.6 does not cover for the sake of simplicity. For instance, the requesting process may find that the incoming adoption offer has become obsolete because the job's allowed number of workers decreased in the meantime due to a volume update. In this case, it returns a message which prompts the accepting process to withdraw its commitment, allowing it to adopt other requests instead. Another example is that the accepting process may find that it already has the required job description, e.g., due to a suspended worker, and therefore does not need to query the requesting process.

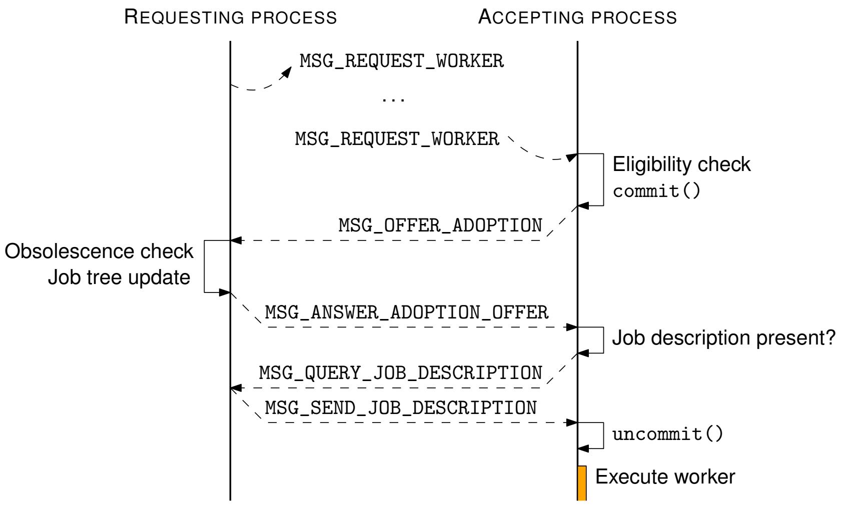


Figure 3.6: Flow diagram for a common scenario in Mallob where a process adopts an incoming job request and joins an active job as a new worker.

3.5.3 Applications

We designed Mallob as a generic scheduling framework for malleable tasks with unknown execution time. Consequently, while SAT solving was the central motivation for Mallob, a natural extension to our work is to consider applications beyond SAT. We now outline the applications we have explored with Mallob, namely SAT solving and two further case studies: hierarchical planning and k-means clustering.

3.5.3.a SAT Solving

Mallob's SAT solving engine, which we denote as MallobSat throughout this work, is described in detail in Chapter 4. Parallelization is achieved by employing a clause-sharing portfolio of solvers (see Section 2.3.2). Clauses to be shared are aggregated and broadcast along T_j. The 2021 version of our engine served as the application code executed in the evaluations in this chapter.

In addition to MallobSat, we have investigated alternative parallel SAT solving approaches within Mallob. Most notably, the master thesis of Maximilian Schick [Sch21a] explored a Cube&Conquer-inspired parallelization (see Section 2.3.1) and its implementation in Mallob, and the master thesis of Malte Sönnichsen [Sön21] investigated an asynchronous peer-to-peer clause exchange scheme. Overall, these more explorative studies did not surpass the performance of our primary engine.

We have experimented with setting specific job demands for SAT jobs, especially with initializing each demand to $d_j = 1$ and then increasing the demand geometrically ($d_j = 1, 3, 7, 15, \ldots$) in periodic intervals until the maximum demand $d_j = m$ is reached.

This "careful expansion" of jobs is supposed to account for the many easy SAT jobs where parallelization does not pay off. However, in preliminary experiments this strategy led to slightly worse performance compared to immediately setting $d_j = m$. Due to our efficient and low-latency scheduling approach, expanding jobs aggressively turns out to be inexpensive and makes best use of the available resources. For this reason, we set SAT job demands to $d_j = m$ in most cases. That being said, we did find the described geometric increases of job demands beneficial in distributed *incremental* SAT solving (Chapter 7) where we reset job demands for each new increment and where extremely low response times for individual increments proved to be crucial.

3.5.3.b Hierarchical Planning

The master thesis of Niko Wilhelm [Wil22] integrated an engine for processing Totally Ordered Hierarchical Task Network (TOHTN) planning problems in Mallob. We refer to Chapter 6 for an introduction to this hierarchical variant of automated planning. Since TOHTN planning is a 2-EXPTIME-hard problem [ABA15], it features extremely irregular search spaces. Therefore, we found TOHTN tasks to be even more unpredictable than SAT tasks, which merits the use of malleable scheduling. Rather than a SAT-based approach (see Chapter 6), we investigated a direct, search-based approach in this case study.

Our parallel TOHTN planning approach employs a distributed tree search which dynamically splits the search space at hand using randomized work stealing [San02]. Adding new workers to an existing planning task is simple—arriving workers will simply steal work from established, busy workers. Removing workers turned out to be more challenging. Discarding a leaving worker's open nodes leads to an incomplete search, and returning its open nodes to another node can be costly since a worker's fringe of open nodes may be very large. In our case study we found a middle ground in only returning the root node common to all local open nodes, which discards some work performed but preserves completeness. In addition, the workers in our distributed planner periodically share visited search nodes in order to avoid duplicate work. Visited nodes are shared in an approximate manner using an Approximate Membership Query (AMQ) data structure. To mitigate completeness issues arising from false positives in the AMQ, search is restarted globally in increasing intervals.

In experiments with moderate degrees of parallelism (up to 64 cores), the described TOHTN planning approach proved to be feasible and showed good scaling behavior for some planning domains. However, we observed deteriorating performance under fluctuating resources of a TOHTN task. In terms of absolute performance, our approach is still outperformed by the state of the art in sequential TOHTN planning.

All in all, we were able to show that irregular search problems with randomized work stealing can be integrated in Mallob. Still, it remains an open problem for future work to design and implement a well performing protocol for leaving workers and to evaluate such an approach on a larger, distributed scale.

3.5.3.c k-means Clustering

Given a set P of n d-dimensional points, *k-means clustering* aims to find k d-dimensional cluster centers which best fit the observed distributions of points in P [HW79]. In contrast to SAT solving and planning, k-means clustering is a numerical problem which can be parallelized in a regular manner [Zha+11]. Since finding an optimal solution is NP-hard [Alo+09], the k-means algorithm approximates a solution: Beginning with k initial cluster centers, each point $p \in P$ is assigned to its nearest center and then each center is updated to best fit its points [HW79]. This is repeated until a termination condition is met (e.g., the change per iteration becomes negligible).

The bachelor thesis of Michael Dörr [Dör22] presented a malleable engine for k-means clustering within Mallob. Parallelization is achieved by partitioning P into $|T_j|$ equally sized parts $P_1, \dots, P_{|T_j|}$ and updating this partitioning whenever T_j is updated.[4] Each worker assigns points and computes updated cluster centers for its personal chunk of P, and these partial results can be all-reduced along T_j to obtain the global new cluster centers for the next iteration. Special handling is required for workers leaving a job during such an all-reduction. Our approach has each leaving worker send a special message to its (transitive) *active* parent, which then adopts the work that should have been performed by its children.

In experiments with up to 128 cores, our approach showed good scaling behavior (e.g., a self-speedup of 31 for 127 workers) up to a certain point. Eventually, communication dominates running times and no further speedups are possible by adding workers. Based on this observation, we introduced a simple predictor $f(n, d, k)$ for the maximum resources a given task is able to make efficient use of, and we used this predictor to set job demands. Using Mallob's feature of fairly scheduling jobs with such constraints, we were able to observe improved performance compared to a scheduling scenario with default demands ($d_j = m$). Overall, while the absolute performance of our parallel clustering is not yet on par with established and optimized approaches, we consider the results encouraging for future use of Mallob for numerical optimization problems.

3.6 Evaluation

We now present our experimental evaluation. Our software and experimental data are available online (see Appendix A).

3.6.1 Setup

All experiments have been conducted on the supercomputer **SuperMUC-NG**. This system at the *Leibniz Rechenzentrum* features a total of "*311 040 compute cores with a main memory of 719 TB and a peak performance of 26.9 PetaFlop/s*" [Rec23].

[4] P is replicated on all participating workers. While keeping only the relevant chunk P_i at each worker w_j^i would be more memory-efficient, such an approach requires redistribution of data whenever a job is resized. Our simple approach is quite practical for modestly sized jobs if job volumes fluctuate frequently. See also Section 8.3.

	Throughput		Efficiency		
n_{par}	θ	θ_{opt}	$\frac{\theta}{\theta_{opt}}$	η	u
3	0.159	0.16	0.991	0.990	0.981
6	0.318	0.32	0.994	0.990	0.983
12	0.636	0.64	0.993	0.991	0.984
24	1.271	1.28	0.993	0.992	0.985
48	2.543	2.56	0.993	0.993	0.985
96	5.071	5.12	0.990	0.993	0.986
192	10.141	10.24	0.990	0.995	0.985
384	20.114	20.48	0.982	0.995	0.983
768	39.972	40.96	0.976	0.992	0.980

Table 3.1: Scheduling uniform jobs on 1536 processes (6144 cores) compared to a hypothetical optimal rigid scheduler. From left to right: Max. number n_{par} of parallel jobs, max. measured throughput θ, optimal throughput θ_{opt} (in jobs per second), throughput efficiency θ/θ_{opt}, work efficiency η, mean measured CPU utilization u of worker threads.

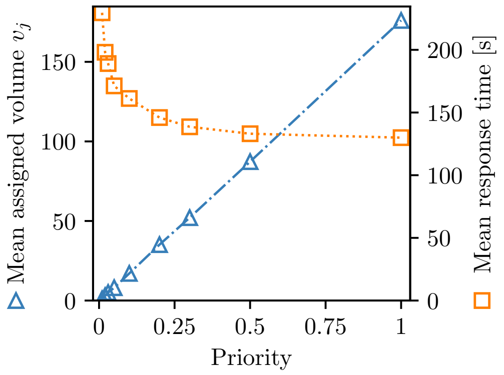
In particular, SuperMUC-NG features 6 336 "thin" compute nodes each with a two-socket Intel Skylake Xeon Platinum 8174 processor clocked at 2.7 GHz with 48 physical cores (96 hardware threads) and 96 GB of main memory. Nodes are interconnected via OmniPath [Bir+15] and run SUSE Linux Enterprise Service (SLES).

If not specified otherwise, we used 128 "thin" compute nodes. We launch twelve processes per machine, assign eight hardware threads to each process, and let an active worker run four threads. Our system can use the four remaining hardware threads on each process in order to keep disturbance of the actual computation at a minimum. We compiled Mallob with GCC 9 and with Intel MPI [Int23] 2019.

3.6.2 Uniform Jobs

In a first set of experiments, we analyze the base performance of our system by introducing a stream of jobs in such a way that exactly n_{par} jobs are in the system at any time. We limit each job j to a CPU time budget of $B = 1920/n_{\text{par}}$ core-minutes, thus ranging from 640 core-min for $n_{\text{par}} = 3$ to 2.5 core-min for $n_{\text{par}} = 768$ while the duration of jobs remains fixed. Each job corresponds to a difficult SAT formula which cannot be solved within the given budget. As such, we emulate jobs of fixed size.

We chose m and the values of n_{par} in such a way that $m/n_{\text{par}} \in \mathbb{N}$ for all runs. We compare our runs against a hypothetical rigid scheduler which functions as follows: Exactly m/n_{par} processes are allotted for each job, starting with the first n_{par} jobs at $t = 0$. At periodic points in time, all jobs finish and each set of processes instantly receives the next job. This leads to perfect utilization and maximizes throughput. We further pretend that this strategy is completely free of any kind of overhead.



p_j	$\tilde{v}_j$	#	RT [s]
0.01	1.0	27	229.6
0.02	3.0	36	198.0
0.03	5.0	37	189.1
0.05	8.1	45	171.3
0.10	17.1	50	161.2
0.20	35.2	51	146.0
0.30	52.4	54	138.6
0.50	87.5	56	133.1
1.00	176.6	58	130.0

Figure 3.7: Impact of job priority on mean assigned volume (left axis, blue triangles) and response time (right axis, orange squares). The table shows each used priority p_j with the corresponding mean assigned volume $\tilde{v}_j$, number of solved instances ("#"), and mean response time in seconds.

For a modest number of parallel jobs n_{par} in the system ($n_{\text{par}} \le 192$), our scheduler reaches 99% of the optimal rigid scheduler's throughput (Table 3.1). This efficiency decreases to 97.6% for the largest n_{par} where $v_j = 2$ for each job. As the CPU time of each job is calculated in terms of its assigned volume and as the allocation of workers takes some time, each job uses slightly less CPU time than advertised: Dividing the time for which each job's workers have been active by its advertised CPU time, we obtained a *work efficiency* of $\eta \ge 99\%$. Lastly, we measured the CPU utilization of all worker threads as reported by the operating system, which averages at 98% or more. In terms of overall work efficiency $\eta \times u$, we observed an optimum of 98% at $n_{\text{par}} = 192$, a point where neither n_{par} nor the size of individual job trees is close to m.

3.6.3 Impact of Priorities

In the following we evaluate the impact of job priorities. We use 32 nodes (1536 cores, 384 processes) and introduce nine streams of jobs, each stream with a different job priority $p \in [0.01, 1]$ (see Fig. 3.7 right) and with a wallclock limit of 300 s per job. As such, the system processes nine jobs with nine different priorities at a time. Each stream is a permutation of 80 diverse SAT instances from the ISC 2020 (see [SS21b]).

As expected, Fig. 3.7 indicates a proportional relationship between priority and assigned volume, with small variations due to rounding. Response times appear to decrease exponentially, which is in line with the NP-hardness of SAT and the limited scalability of parallel SAT solving (Chapter 4). The number of solved instances increases consistently and more than doubles going from the lowest to the highest priority job stream. As such, we can clearly observe the desired impact of job priorities.

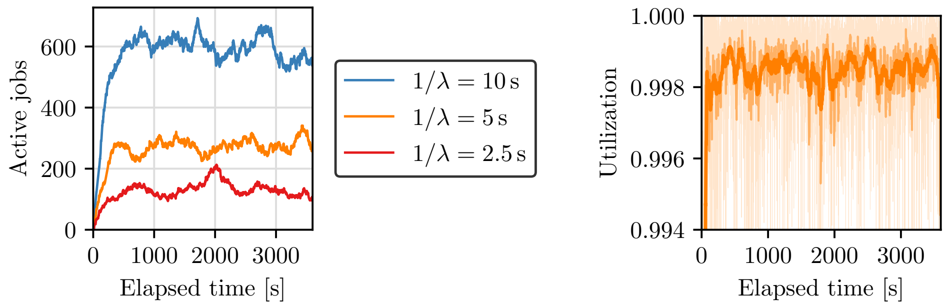


Figure 3.8: Left: Number of active jobs for different interarrival times $1/\lambda$. Right: Utilization (i.e., ratio of busy processes) for $1/\lambda = 5\,\mathrm{s}$ at a sliding average of window size 1 s, 15 s, and 60 s respectively.

3.6.4 Realistic Job Arrivals

In the next set of experiments, we analyze the properties of our system in a more realistic scenario. Four processes introduce batches of jobs, with each batch consisting of one to eight jobs. Arrival times of individual batches are drawn from an exponential distribution with rate parameter $\lambda \in \{0.4/\mathrm{s}, 0.2/\mathrm{s}, 0.1/\mathrm{s}\}$, equivalent to expected interarrival time $1/\lambda \in \{2.5\,\mathrm{s}, 5\,\mathrm{s}, 10\,\mathrm{s}\}$. This way, the arrivals of batches follow a Poisson distribution [KR68]. Our intention is to simulate users which arrive independently and submit several jobs at once, where the choice of λ controls the overall load. We also sample a priority $p_j \in [0.01, 1]$, a maximum demand $d_j \in \{1, \dots, p\}$, and a wallclock limit $b_j \in [1\,\mathrm{s}, 600\,\mathrm{s}]$ for each job.

3.6.4.a Overview

Fig. 3.8 (left) shows the number of active jobs in the system over time for our default configuration—featuring our improved worker reuse strategy (Section 3.4.3.b) and matching requests along a routing tree (Section 3.4.2.c). For all tested interarrival times, considerable changes in the system load can be observed during a job's average life time which justify the employment of a malleable scheduling strategy. Fig. 3.8 (right) illustrates for $1/\lambda = 5\,s$ that system utilization is at around 99.8% on average and almost always above 99.5%. We also measured the ratio of time for which each process is idle: The median process is busy 99.08% of all time for the least frequent job arrivals ($1/\lambda = 10\,\mathrm{s}$), 99.77% for $1/\lambda = 5\,\mathrm{s}$, and 99.85% for $1/\lambda = 2.5\,\mathrm{s}$. Also note that $\sum_j d_j < m$ for the first seconds of each run, hence not all processes can be utilized immediately. The latency of our volume calculation, i.e., the latency until a process receives an updated volume for an updated job, reaches a median of 1 ms and a maximum of 34 ms for our default configuration (not shown).

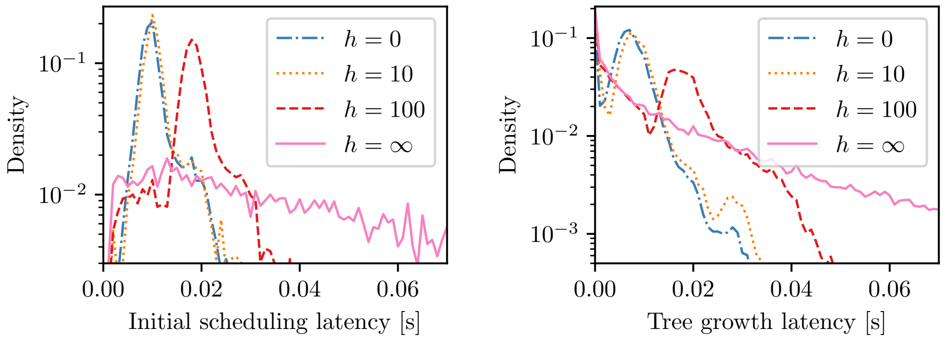


Figure 3.9: Distribution over measured latency for the initial scheduling of a job (left) and finding a requested worker (right), for inter arrival rate $1/\lambda = 5\,s$, for a varying number h of random hops until a request message is routed along R.

3.6.4.b Request Matching Strategies

In the following, we evaluate our request matching strategies. We first compare our routing tree based request matching (denoted T; see Section 3.4.2.c) with our “random walks” approach (denoted W; see Section 3.4.2.b). For different values of h, each request message performs up to h random hops before switching to strategy T.

We first examine latencies for the initial scheduling of an arriving job. Fig. 3.9 (left) shows that the lowest latencies were achieved by executing strategy T only ($h = 0$). For increasing values of h, the variance of latencies increases and high latencies become increasingly likely. Note that jobs usually enter a fully utilized system and begin with a demand of $d_j = 1$. Therefore, some balancing updates render only a single process idle, which heavily disfavors W (see Section 3.4.2.b).

The second kind of latency we examine is the *tree growth latency*, measuring how long it takes until an emitted request results in a ready worker reporting back. Fig. 3.9 (right) illustrates that while most requests are resolved very quickly with W, some requests take a very large number (thousands) of hops to succeed, resulting in high latencies (> 50 ms). Again, T results in much lower and more predictable latencies.

In addition to our original publication [SS22a], we have implemented a third request matching approach based on asynchronous prefix sums (Section 3.4.2.d). We therefore present another set of experiments with 96, 384, and 1536 processes (384, 1536, 6144 cores) to compare all three approaches on different scales.[5] We configured $m/24$ processes to introduce jobs with Poisson-distributed inter arrival rates.

Fig. 3.10 shows results. As in the previous experiments, random walks (W) match some requests very rapidly but also result in a significant ratio of high latencies.

[5]We performed this set of experiments with a newer version of our codebase (March 2023) compared to all other experiments in this chapter (February 2022).

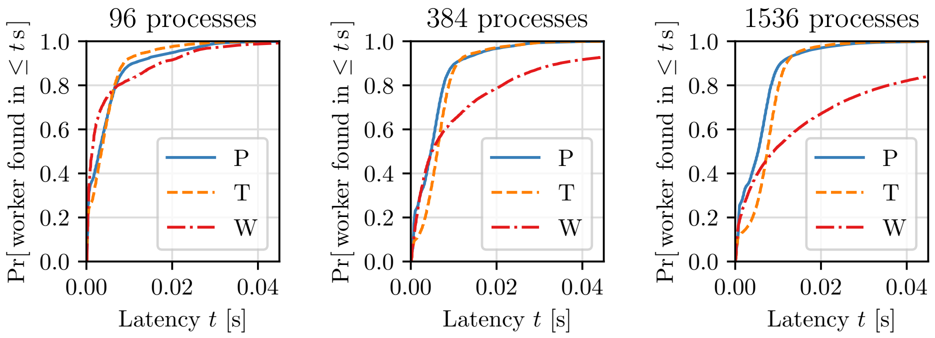


Figure 3.10: Comparison of three request matching strategies (W: random walks with $h = \infty$; T: routing trees; P: prefix sums) executed with 96 (left), 384 (center), and 1536 (right) processes. Each plot shows cumulative distribution functions for the probability that a requested worker is found within a given time.

We can also see that latencies of W scale the worst with an increasing number of processes. Our approaches with prefix sums (P) and routing trees (T) scale substantially better. P is more likely to result in very low latencies (around 5 ms and lower) compared to T—an effect that becomes more pronounced when scaling up to a higher number of processes. We suspect this is due to the fact that P is able to expand job trees by multiple layers at once, therefore avoiding a logarithmic factor on many latencies. On the other hand, higher latencies (around 10 ms and higher) are marginally more frequent for P than for T: Since changes in the set of active jobs (i.e., volume updates or jobs finishing) can invalidate some of the process-request pairs matched by a prefix sum, individual requests may need to undergo several rounds of prefix sums until they are matched successfully. T, on the other hand, can react to changes in the system state more quickly and dynamically. This drawback of P becomes less pronounced for a high number of processes.

To conclude, we found request matching both with routing trees and with prefix sums to be robust strategies, with an indication that prefix sums scale better to large systems. For the following experiments, we used request matching via routing trees.

3.6.4.c Worker Reuse Strategies

We compare three different suspended worker reuse strategies (Section 3.4.3): No deliberate reuse at all, the basic approach, and our improved approach. As an evaluation metric, we count how often a new worker was created for job j and divide this total by the job's maximum assigned volume v_j. This *Worker Creation Ratio* (WCR) is ideally 1 and becomes larger the more often a worker is suspended and then re-created at a different process. We report the WCR for each job and in total: As Tab. 3.2 shows, our latest approach reduces a WCR of 2.14 down to 1.8 (-15.9%).

	WCR			CS				Pr [WC ≤ ·]				
	med.	max.	total	med.	mean	Pr.	RT	1	2	5	10	25
None	1.43	33.0	2.14	136	138.2	5923	153.40	0.87	0.90	0.94	0.97	0.992
Basic	1.40	31.5	2.07	134	135.3	5921	153.89	0.87	0.90	0.94	0.97	0.993
Ours	1.25	24.5	1.80	130	131.8	5939	152.33	0.89	0.91	0.94	0.97	0.993

Table 3.2: Worker reuse strategies in terms of worker creation ratio (WCR, per job—median, maximum—and in total), context switches (CS, median per process and mean), jobs processed within 1 h (Pr.), their mean response time (RT), and the fraction of workers created on at most $\{1, 2, 5, 10, 25\}$ distinct processes.

Context switches (i.e., how many times a process changed its affiliation) and average response times are improved marginally compared to the naïve approach. Last but not least, we counted on how many distinct processes each w_j^i has been created: Our latest strategy initializes 89% of all workers only once, and 94% of workers have been created at most five times. We conclude that most jobs only feature a small number of workers, at the leaf level of their job tree, which are rescheduled frequently.

3.7 Conclusion

We have presented a decentralized and highly scalable approach to online job scheduling of malleable NP-hard jobs with unknown processing times. We split our problem into two subproblems, namely the computation of fair job volumes and the assignment of jobs to processes, and proposed scalable distributed algorithms with $\mathcal{O}(\log m)$ span for both of them. We presented a practical implementation and, with SAT solving as an application, experimentally showed that our system schedules incoming jobs within tens of milliseconds, distributes resources proportional to each job's priority, and leads to near-optimal utilization of resources. The application studies we performed indicate that our approach may also offer an appealing scheduling environment for disciplines beyond SAT solving in the future.

In terms of future work, we consider to generalize our approach to heterogeneous computing environments and add fault tolerance to our distributed algorithms. Furthermore, we intend to explore more realistic and sophisticated models for job priorities. In particular, we envision the use of decentralized auctioning schemes for determining job volumes that are based on the submitters' expressed utility for different job sizes. Such a mechanism may not only increase resource-efficiency but would also allow our approach to be used in the context of service-based computing with explicit accounting.

Chapter 4

4 Scalable Distributed SAT Solving

SAT solving in large distributed environments has previously led to some famous results and to linear or even super-linear speedups for particular inputs. However, in terms of general-purpose SAT solving, existing approaches still cannot make efficient use of a large number of processors. We aim to address this issue with a complete and systematic overhaul of the prior solving system HordeSat *with a focus on its algorithmic building blocks. In particular, we propose a communication-efficient approach to clause sharing, careful buffering and filtering of produced clauses, and a new portfolio consisting of state-of-the-art solver backends. In our evaluations, our approach significantly outperforms* HordeSat*, doubling its mean speedup. In the last four iterations of the International SAT Competition (2020–2023) our system was consistently the best performing massively parallel solver (1600 hardware threads) and one of the best performing shared-memory parallel solvers (64 hardware threads). Last but not least, our SAT solving engine is malleable, i.e., designed to be deployed in a flexible manner on a fluctuating set of resources. In experiments with up to 6400 cores we show that our system combines parallel job processing and parallel SAT solving to resolve a record number of instances from a SAT competition in a highly resource-efficient manner.*

Author's Notes. *This chapter can be considered a substantial revision and extension of "*Scalable SAT Solving in the Cloud*" [SS21b], a publication by Peter Sanders and myself. Some parts of this chapter are copied verbatim or with minor changes from that publication. The publication features a preliminary introduction to our scheduling environment (see Chapter 3) which is omitted in this chapter. I contributed the vast majority of the publication's remaining content, with editing by Peter Sanders. This chapter also includes content from my solver descriptions to the International SAT Competition [Sch20; Sch21e; Sch22; Sch23] (non-peer-reviewed technical reports).*

4.1 Introduction

In today's applied SAT solving, researchers and industrial users increasingly strive to exploit modern distributed environments with hundreds to thousands of cores [NTC17; HFB20; Fro+21; Coo21; Bur+22] in order to reduce processing times and to tackle difficult problems which are infeasible to solve with the sequential state of the art.

Prior massively parallel approaches which are used successfully to solve open mathematical problems [HKM16; Heu18] are usually fine-tuned to the particular problem at hand. In terms of SAT solving that is *general purpose*, i.e., that works efficiently on application problems never seen before, mean speedups of existing distributed solvers still leave much to be desired. Since the difficulty of individual propositional formulas can range from trivial to extremely difficult and is usually not known beforehand, the linear or even superlinear speedups which have been reported for few individual instances [BSS15] must be set in relation with the total work invested in every single formula to achieve such peak speedups. The prior state of the art in massively parallel SAT solving, HordeSat, reportedly achieved a median speedup of 13 on 2048 cores on industrial benchmarks [BSS15], implying a median efficiency of only 0.6%.

We argue that two distinct challenges must be met to improve on the practical merit of distributed SAT solving. First and foremost, distributed SAT solving itself needs to become more efficient and more scalable for realistic and diverse inputs. Secondly, the deployment of SAT solving tasks in distributed environments must be performed in a more careful manner—ideally processing several tasks at a time and allotting a large amount of computational resources only to sufficiently difficult problems.

In this work, we aim to address both challenges with a new distributed SAT solving engine, denoted MallobSat throughout this work. We revisit the popular massively parallel SAT solver HordeSat [BSS15] and carefully re-engineer its algorithmic building blocks. Most notably, we introduce a novel approach to periodic all-to-all clause sharing. We aggregate the globally most useful clauses (w.r.t. clause length primarily and LBD secondarily) and we limit the overall sharing volume to a function that is sublinear in the number of participating solvers, therefore ensuring a communication-efficient operation and low overhead even with thousands of cores. We also introduce a distributed filtering approach for recently shared clauses. To further improve the practicality of our approach, we propose different measures to make more careful use of main memory, such as the manipulation of shared clauses' LBD values, and we extend and update the used sequential SAT solver backends. Last but not least, MallobSat is designed to be *malleable*, i.e., it supports a fluctuating number of workers during its execution. This allow us to integrate MallobSat in the job scheduling platform Mallob we presented in Chapter 3.

In comprehensive evaluations, we investigate the benefits of our different techniques and configurations. Among other results, we find that our clause sharing acts as an effective kind of work subdivision even if solver diversification is reduced to a minimum. Our scaling results show that our solver doubles the speedups of an updated version of HordeSat. With our system we observed improved performance whenever increasing the computational resources, up to 3072 cores (64 nodes). We analyze the results of four iterations of the International SAT Competition (ISC), which together clearly indicate that our system represents the state of the art in (massively) parallel SAT solving. We also demonstrate that MallobSat's malleability is an important feature which can result in an appealing combination of "embarrassingly parallel" job processing and the speedups obtained by parallel SAT solving: In an experiment on 6400 cores, Mallob with flexible deployment of MallobSat tasks is able to resolve a record number of instances from ISC 2021 within merely two hours of total running time.

This chapter is structured as follows. In Section 4.2 we provide an overview of our design decisions. We then present our novel approaches on clause sharing in Section 4.3 and on diversification in Section 4.4. We describe further technical contributions such as memory awareness in Section 4.5. Section 4.6 features our own experimental evaluation and Section 4.7 contains a discussion of our system in the International SAT Competition. We conclude this chapter in Section 4.8.

4.2 Overview

We now discuss our high-level design decisions compared to HordeSat [BSS15]. For a general introduction to HordeSat we refer to Section 2.3.3.a.

Some of our technical design decisions are directly adopted from HordeSat. First, we follow a modular portfolio solver design with a generic and compact interface to which different sequential solver backends can be connected. This "blackbox approach" allows a distributed solver to profit from a large pool of diversification and can also help to transfer progress in sequential SAT solving to distributed solving with little effort, in contrast to systems which have a tighter integration with a specific sequential solver (e.g., TopoSAT [ENS14] or D-Syrup [Aud+17] with Glucose).

Secondly, we follow a two-level parallelization model where multiple (distributed) processes communicate via message passing and multiple solvers run concurrently within each process. In this point we will go further than HordeSat, which was tuned to spawn a process for each set of four cores, by preparing our system for process layouts that are faithful for the hardware at hand. In other words, we aim to group all cores of a socket into a single process, which requires careful use of concurrent data structures given that a socket can feature dozens of cores.

Thirdly, we overlap communication and SAT solving: Distributed clause sharing is performed in a dedicated communication thread and solvers do not need to be interrupted. Again, we will go further than HordeSat by performing purely *asynchronous communication*. As a consequence, even the communication thread of a process can perform other tasks while a collective operation is being performed.

Compared to HordeSat, we make some deviating assumptions to our execution environment. First, we do not assume any kind of shared memory, disk or RAM, across processes. For this reason, in our system a single process parses the input formula and then distributes it to all processes via message passing. Secondly, we require our computation to be *malleable*, i.e., certain processing elements may be added or removed during its execution. In particular, we use the job deployment model of Mallob (Chapter 3). On an abstract level, we can imagine a particular SAT solving task as a list of processes where, at any time during the computation, any (strict) suffix may be removed or appended. More specifically, each solving task is deployed as a binary tree of processes $T = \langle w^0, w^1, \ldots, w^{|T|-1} \rangle$ which can grow or shrink at the leaf level at any time. Any communication we perform has to respect these fluctuations.

4.3 Clause Sharing

In the following, we describe the overall design decisions for our approach to clause sharing and then describe three crucial aspects of its implementation: the collective operation itself, the appropriate buffering of clauses, and the filtering of undesirable clauses. We also touch on a compensation mechanism for unused sharing volume and on the appropriate handling of LBD values.

4.3.1 Design Decisions

We now present our view on clause sharing and our subsequent design decisions.

On an abstract level, we consider a distributed solving procedure a collaborative effort of diverse experts (Section 2.3.2.b) where clause sharing acts as a kind of search space pruning: If a single solver is able to find a crucial conflict c which has not yet been found by any other solver, broadcasting c prevents the other solvers from exploring the subspace pruned by c. Following this intuition, we explore all-to-all clause sharing rather than limiting the potential receivers of a given clause producer to a subset of solvers (*cf.* [ENS14]). Prior all-to-all clause sharing [BSS15; Aud+17; EN19] does suffer from problems in terms of scalability and/or quality, which we intend to overcome with algorithmic improvements. Furthermore, in line with prior observations,[1] we assume that distributed search space pruning via clause sharing works best if the mean *clause turnaround time*, i.e., the time it takes for a produced clause to be imported by another solver, is as low as possible. Intuitively, the longer it takes for a clause to be shared and imported, the higher the probability that other solvers redundantly performed the same work. We thus aim to reduce clause turnaround times.

On a technical level, in large parts we adopt the general information flow of HordeSat's clause sharing as illustrated in Fig. 4.1. The solver threads within each process write some of their produced clauses into a shared, process-local *export buffer*. Periodically, all processes extract clauses from their export buffers and contribute them to a collective sharing operation. Each process then receives the aggregated set of shared clauses and forwards them to its solver threads for import. At several stages of this procedure, clauses may be filtered or selected based on various criteria.

A central design consideration of clause sharing is which and how many clauses to share (Section 2.3.2.b). When sharing more and more clauses in a distributed solver, we assume that the merit of clause sharing eventually levels off whereas communication and computational overhead continue to increase [ENS14; BSS15; EN19]. For this reason, we aim to limit clause sharing to a point where its merit is nearly maximized while avoiding any unnecessary overhead. We believe that fixing the volume of the *globally best distinct* clauses in a careful all-to-all sharing operation is a robust and effective means to control the degree of clause sharing. In addition, this global view allows clause sharing to seamlessly adapt to the instance at hand and to the current

[1]"*sharing clauses, as soon as possible, among search units provides better results*" [Aud+17]; "*it appears [...] important to exchange learnt clauses fast*" [EN19]

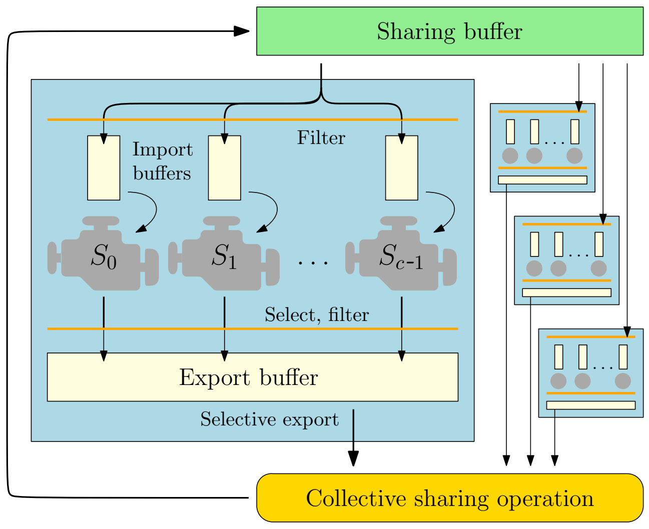


Figure 4.1: Information flow in the clause sharing of HORDESAT and also our system. Each blue rectangle corresponds to a solver process. One of these processes is depicted in more detail with c solvers, clause buffering and filtering.

state of solvers—effectively enforcing dynamic quality criteria (*cf.* [HJS12]) rather than fixed clause length or LBD thresholds.

Under the view of clause sharing as distributed search space pruning, we believe that clause length is a sensible metric for clause quality. Without any further assumptions, shorter clauses in general prune larger parts of search space and therefore provide the most valuable and most densely encoded information to other solvers (see Section 2.3.2.b). LBD values, which are a crucial clause quality metric in sequential SAT solving, may not necessarily translate to a (massively) parallel setting in the same direct manner. Since the LBD value of a clause depends on the producing solver's internal state, the metric is not necessarily as meaningful on a global scale, i.e., for all receiving solvers with different internal states. As such, we design our clause sharing to prefer short clauses first and merely break ties using LBD values.

4.3.2 Clause Exchange Operation

We now discuss the collective (communication) operation which forms the basis for clause sharing, beginning with HORDESAT and then presenting our own approaches.

4.3.2.a HordeSat

HORDESAT uses synchronous MPI calls to periodically perform an all-to-all clause exchange. Specifically, a collective operation named *all-gather* (Section 2.1.2) is used.

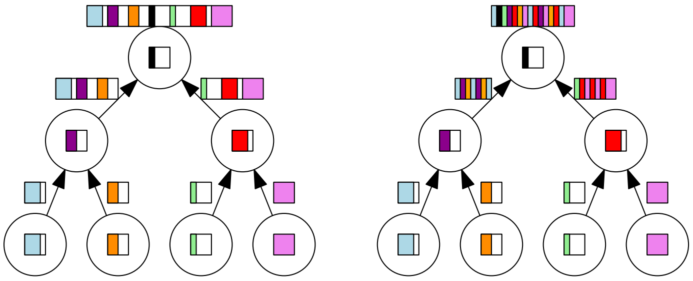


Figure 4.2: Exemplary flow of information in the first half of HordeSat's all-gather operation (left) and in our aggregation within a job tree (right). Each circle is a process; each box in a circle represents the process's buffer b_i.

Each process i contributes a buffer b_i of fixed size β. Each b_i contains a serialization of a set of clauses which the process's solver threads produced and then stored in the process-local export buffer. The concatenation of all b_i, the *sharing buffer* $B := \langle b_1, \ldots, b_p \rangle$, is broadcast to all processes. Then each process can hand (some of) the shared clauses to its solvers. Fig. 4.2 (left) illustrates the process of aggregating B.

The above clause exchange mechanism has various shortcomings. First, whenever some process i does not completely fill b_i, parts of the sharing buffer B remain unused and carry no information [Aud+17]—see Fig. 4.2 (left) for an illustration. HordeSat attempts to remedy this with an initially very strict LBD requirement for exported clauses which under-producing processes can then successively lift for their solvers. This, however, does not work reliably in all cases and also implies an initial "warmup phase" where many decent clauses may be discarded. A second problem is that B can contain a significant portion of redundant clauses. In particular at the beginning of SAT solving when a formula is simplified and preprocessed, we noticed that this can lead to highly redundant sets of clauses in B. This effect is especially pronounced for unit clauses which are never checked for redundancy (see below). Thirdly, the targeted volume of shared clauses is strictly proportional to the number of involved processes. For sufficiently large setups, this may constitute a bottleneck both in communication volume and in the local work necessary to digest and process every clause received.

4.3.2.b Compact Clause Exchange

We assume a distributed binary tree T of processes as a communication structure (see Section 4.2). At the beginning of each clause exchange, each leaf node in T sends its exported clauses to its parent node in T. When an inner node has received as many buffers as it has children, it exports its own clauses and then performs a two- or three-way *merge* of the present buffers: All input buffers are read simultaneously from left to right and aggregated into a single sorted output buffer, similar to the merge of sorted sequences in textbook Mergesort [San+19, p. 160]. We keep the clauses in each buffer ordered by length, then by LBD and finally in lexicographic order.

We sort the literals within each exported clause, which brings them into a canonical form and helps to easily recognize and filter duplicates.

In the merge operation at a node of T that is the root of a subtree with u nodes, we *limit* the size of the output buffer with a function $b(u)$. Any remaining clauses in the input buffers which exceed the limit $b(u)$ are inserted back into the local export buffer (see Section 4.3.3.b) where they may be re-exported at a later point in time. As the clauses in each merged buffer are ordered by quality, we aggregate some of the globally most valuable information while imposing a strict limit on the overall communication volume. We also ensure high density of useful information in the transferred data because each sent buffer is of compact shape and contains no duplicates.

For $m = |T|$ solver processes and any monotonically increasing function $b(m)$, our clause sharing operation has span $\mathcal{O}(b(m)\log m)$ and incurs $\mathcal{O}(m \cdot b(m))$ internal work. Due to the binary tree structure of our aggregation and broadcast, exactly $2(m-1)$ messages of size $\mathcal{O}(b(m))$ are sent.

Malleable implementation. Our compact clause exchange is straight forward to make malleable as described in Section 4.2. In our implementation, a clause exchange is initiated by the root of T broadcasting a signal through T. The "snapshot" $\tilde{T}$ of all processes receiving this signal defines the aggregation operation. Inner nodes in $\tilde{T}$ expect a contribution of clauses from each of their children in $\tilde{T}$, and leaves in $\tilde{T}$ prepare to export and send clauses to their parent. If a process leaves the computation before it can send clauses, it sends an empty buffer to their parent in $\tilde{T}$ to ensure that the aggregation progresses. The broadcast of the sharing buffer, by design, reaches all processes which belong to T at *that* point in time.

Buffer limit scaling. Initially, we chose the limit $b(u)$ on the buffer size at each node of T as follows: Bundled with each buffer payload we communicate the number u of buffers aggregated so far. For each aggregation step, i.e., for each further level of T_j that is reached, we want to discount the maximum buffer size by a factor of α. As a consequence, we compute the buffer size limit $b(u) := \lceil u \cdot \alpha^{\log_2(u)} \cdot \beta \rceil$, where β is the base sharing volume per process (see Section 4.3.2.a). This limit on the shared clause literals can be steered by a parameter $\alpha \in [\frac{1}{2}, 1]$, the *discount factor* at each buffer aggregation. For $\alpha = \frac{1}{2}$, we can see that the clause buffer size converges to β; for $\alpha > \frac{1}{2}$, the clause buffer size diverges. For $\alpha = 1$, our approach emulates HordeSat's shared clause buffer which grows proportionally without any discount.

After some experiments with MallobSat, we have noticed a few shortcomings of the above function $b(u)$. First, $b(u)$, while sublinear in u, is still unbounded, such that the sharing volume may eventually overburden the solvers. Secondly, the appropriate parametrization of $b(u)$ proved to be challenging. For instance, $b(u)$ depends only on the number u of processes involved in sharing, not on the number of solver threads. Therefore, deploying MallobSat with varying process layouts changes the scaling of buffer sizes in an undesired manner. As an example, running MallobSat with 960 solvers at four solvers per process and $\beta = 1500$, the buffer limit for $\alpha = 0.9$ evaluates to $b(240) = 156\,496$, whereas running MallobSat at the same scale at 24 solvers per process and consequently with $\beta = 24/4 \cdot 1500 = 9000$ yields $b(40) = 205\,487$.

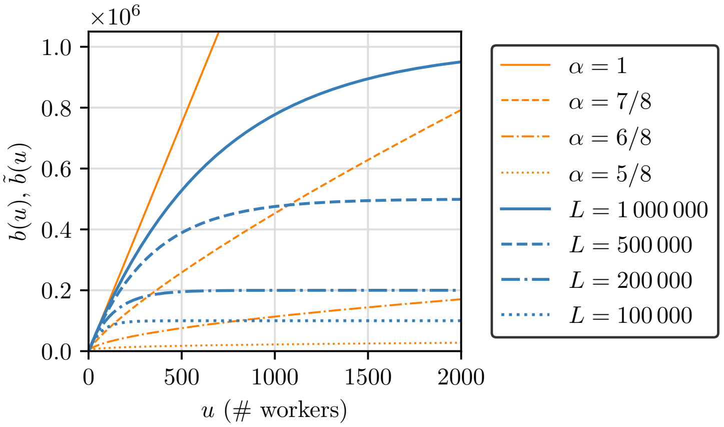


Figure 4.3: Different parametrization of $b(u)$ (fine orange lines, based on α) and $\tilde{b}(u)$ (thick blue lines, based on L) for $\beta = 1500$.

For our second approach, we remodel buffer limit $\tilde{b}(u)$ based on three simple constraints. For the smallest of setups, the buffer size should be proportional to the number of workers. Therefore, the value and the derivative of $\tilde{b}(u)$ at $u = 1$ should both be equal to β: (i) $\tilde{b}(1) = \beta$ and (ii) $\tilde{b}'(1) = \beta$. For sufficiently large setups, the buffer limit should converge to an upper bound L. Therefore, (iii) $\tilde{b}(u) \to L$ for $u \to \infty$. A simple model which satisfies constraints (i) and (iii) is the function

$$\tilde{b}(u) = L - (L - \beta) \cdot e^{-k(u-1)}$$

for $k \in \mathbb{R}^+$. To satisfy constraint (ii), we set $k := \frac{\beta}{L-\beta}$ and therefore[2] arrive at

$$\tilde{b}(u) = L - (L - \beta) \cdot e^{\frac{\beta}{\beta - L}(u-1)}.$$

Fig. 4.3 illustrates how $b(u)$ and $\tilde{b}(u)$ scale differently relative to u. At large scales where $\tilde{b}(u)$ approaches L and stops increasing, note that we may still profit from adding solvers to the computation—not in the sense of sharing more clauses, but rather in the sense of sharing *better* clauses. Whereas $b(u)$ is based on the notion of reducing the buffer growth by a certain factor at each further aggregation level in T, $\tilde{b}(u)$ depends on the limit L on each sharing's volume. We consider the latter parameter to be more natural and ergonomic than the former. Last but not least, changing the number of solver threads per process while adjusting β by the same factor now leads to very similar behavior relative to u, except for small differences

[2]The derivative of $\tilde{b}(u)$ is $\tilde{b}'(u) = k(L - \beta)e^{-k(u-1)}$. Assuming $\tilde{b}'(1) = \beta$ yields $k(L - \beta) = \beta$ and therefore $k = \beta/(L - \beta)$.

due to the offset of u by 1 in the exponent.[3] Alternatively, we can reinterpret u as the number of solver threads and β as the buffer base size *per solver* to render $\tilde{b}(u)$ completely agnostic to how solvers are deployed.

4.3.3 Clause Buffering

In distributed clause sharing, the journey of a produced clause from its original solver to the clause database of another solver features several steps of buffering where the clause may be deferred or discarded for various reasons. We describe this journey for HordeSat and then propose our improvements. Our overarching aim is to discard a clause c only if the volume of shareable clauses of equal or better quality than c already exhausts the budget of clauses to share.

4.3.3.a HordeSat

In HordeSat, clauses of sufficient quality exported by local solvers are written into an *export buffer* structure $\mathcal{B}$ which features *buckets*, one for each admissible clause length. Each such bucket is a stack of fixed capacity that stores each clause together with its individual LBD value. If a bucket is full, any further clauses of this length will be discarded until the next export of clauses. At each export, the buffers are flushed by decreasing clause quality until the local export volume is met or no more clauses remain. A consequence of this simple strategy is that some buckets running full may result in losing potentially useful clauses. Another consequence is that if full buckets are not flushed for an extended period, the first (oldest) clauses which have been produced are preserved whereas more recent clauses are discarded.

Regarding the import of incoming clauses, the main thread of a HordeSat process copies all admitted clauses from clause sharing into an *import buffer* $\mathcal{B}_S$ for each solver S and increases its size as necessary. S can then import the clauses in $\mathcal{B}_S$ at its own discretion. $\mathcal{B}_S$ is guarded by a mutex which is locked by the solver thread before reading clauses and by the main thread before writing clauses. If S cannot acquire this lock, it retries at a later point in time. If S does not import clauses sufficiently fast or often enough, $\mathcal{B}_S$ may increase in size indefinitely. For instance, we noticed that certain solvers can spend minutes in expensive preprocessing routines without retrieving any shared clauses, which leads to very large import buffers at times.

4.3.3.b Adaptive Clause Buffering

We observed that the statistical distribution over the length of clauses exported by a solver depends on many variables, such as the input formula, the type and configuration of the solver, and the point in time during solving. This indicates that a clause buffering structure with fixed-size buckets for each clause length such as HordeSat's is suboptimal for portfolio solving. Instead it may be beneficial to adapt the size of individual buckets to the observed distribution of clause lengths.

[3]In the earlier example we get $\tilde{b}(240) = 97\,413$ for $\beta = 1500$ and $\tilde{b}(40) = 98\,077$ for $\beta = 9000$.

For example, if a clause of length three arrives yet the bucket for such clauses is full, then we may increase that bucket's capacity at the cost of shrinking the bucket for clauses of length four. Likewise, if an import buffer $\mathcal{B}_S$ runs full, we should not discard the oldest or latest clauses but rather the clauses of worst quality.

Following this intuition, we aimed at a more dynamic allocation of space across the buckets in the export buffer $\mathcal{B}$. We again implement $\mathcal{B}$ as an array of buckets, one bucket for each clause length $l \geq 1$. Each bucket corresponds to a stack of clauses which can be added to or removed from. A single *budget* integer shared by all buckets represents the remaining number of literals which can still be inserted until $\mathcal{B}$ is full. If this budget is insufficient for inserting a given clause c of length l, an attempt is made to discard clauses from a bucket $l' > l$ in order to "steal" space for c. If this is unsuccessful for all admissible l', c is discarded. With this flexible buffering structure, we balance the available space dynamically among the different clause quality levels and therefore discard any clauses below a certain quality threshold. This threshold is determined indirectly by the distribution over produced clause lengths.

We do not insert produced clauses beyond length 60 in $\mathcal{B}$. Note that long clauses with tens of literals are admitted (and thus considered for sharing) only if the mean produced clause length is accordingly high, which we did observe on some instances. In terms of LBD scores, we dropped HordeSat's method to only admit low-LBD clauses to $\mathcal{B}$ and to successively lift this limit for underproducing solvers. Steering the export volume per process is not as crucial for our approach, and in fact, MallobSat achieved slightly better performance without this method in early experiments [SS21b]. We do, however, use separate buckets for each length-LBD combination up to a certain clause length, allowing us to break ties via LBD values during export.

We noticed that solvers occasionally produce huge amounts of unit clauses (tens of thousands) in a single burst, which overburdens $\mathcal{B}$ and results in discarding most produced clauses. For this reason, we now allow the buffers to store an unlimited number of unit clauses while keeping the shared budget for all other slots. We expect that even keeping all derivable unit clauses of the problem in main memory is not a problem since the representation of F itself is strictly larger.

We use the same data structure as $\mathcal{B}$ for each import buffer $\mathcal{B}_S$. This way, the buffering of incoming clauses is robust towards solvers which may not import clauses for a long period of time and therefore necessitate dropping some buffered clauses: Only the clauses of worst quality will be dropped in such cases.

4.3.4 Clause Filtering

Tied to clause sharing, the *clause filtering problem* is to decide for a shared clause c and a solver S whether S has received or produced c before and should therefore not receive c (*cf.* [BSS15]). A variant of this problem is to block such clauses not indefinitely but to rather re-allow their sharing after some amount of time or some number of sharing operations have passed. The reasoning behind such temporally limited filtering is that solvers forget most redundant clauses over time and may in some cases benefit from re-learning crucial clauses [AS14; BSS15].

4.3.4.a HordeSat

HordeSat's clause filtering is realized with approximate membership query (AMQ) data structures [BSS15], specifically Bloom filters [Blo70]. Each process employs one *node filter* and *t solver filters* (one for each solver thread). At clause export, each clause is registered in its solver filter and then tested against the node filter. At clause import, each clause is tested against the node filter and then against each solver filter. The usage of AMQs implies that false positives may occur, leading to the rejection of some potentially useful clauses which have in fact not been shared before. This risk of false positives was the main motivation for HordeSat to admit all produced unit clauses without any filtering due to their importance [BSS15]. This can be problematic because particular unit clauses can be produced redundantly by many solvers and can therefore waste considerable amounts of space in HordeSat's sharing buffers.

4.3.4.b Base Approach

In our first approach, we adjusted HordeSat's clause filtering to align it with our new clause sharing operation. We omitted node filters because their main use is to check for duplicate clauses *across* processes, what is already done during the aggregation of buffers in our case. We complemented the solver filters with an additional filtering of unit clauses, using an exact set instead of an AMQ data structure. This way we do not get any false positives for unit clauses and make sure that each such clause is being shared once. We also implemented a mechanism similar to restarts into the clause filters: Every X seconds, half of all clauses (chosen randomly) in each clause filter are forgotten and therefore can be shared again. However, the probabilistic forgetting in our implementation can result in a "degenerating" AMQ and empirically did not perform convincingly compared to keeping filter information indefinitely [SS21b]. For this reason, we omit this technique from our evaluations in this chapter.

4.3.4.c Distributed Filter

Our base approach still features Bloom filters at each solver process which occasionally result in erroneous rejection of unseen clauses. The probability for such false positives grows with the number of clauses registered in the filters, which may become noticeable if millions of clauses are being shared.

For any $e \geq 0$, we define *epoch* e as the time interval that begins with the e-th sharing operation (or, for $e = 0$, with the start of solving) and ends with the $e + 1$-th sharing operation. The clause filtering mechanism we describe in the following is exact in the sense that a shared clause c is *admitted for import* in epoch e if and only if c has *not* been shared and admitted for import in epochs $e - z, \ldots, e - 1$, where $z \geq 0$ is the user-defined *resharing period*. The insight enabling our approach is that it is very much possible to remember all (successfully) shared clauses within a horizon z if we exploit the available distributed memory effectively. In our approach, each process is responsible for remembering the clauses which it contributed itself. We can use a periodic garbage collection to remove clauses older than z epochs from the filter.

As such, the memory requirements at each process are limited to the volume of clauses it produced within the last $\mathcal{O}(z)$ epochs.

On each process, we use a hash table H of clauses which maps a produced clause c to $H[c] := (p(c), e_{\text{prod}}(c), e_{\text{sh}}(c))$, where $p(c)$ is a bitset representing which local solvers produced c, $e_{\text{prod}}(c) \in \mathbb{N}$ indicates the last epoch where a local solver produced c, and $e_{\text{sh}}(c) \in \mathbb{N} \cup \{-z\}$ indicates the last epoch where c was shared and admitted for import by our filter. If c was never shared before, then $e_{\text{sh}}(c) = -z$.

Each produced clause c which meets a basic quality criterion (w.r.t. clause length, see Section 4.3.3.b) is looked up in H. If $c \notin H$, we try to insert c into export buffer $\mathcal{B}$. On success, c is inserted into H as well, initializing $p(c)$ with the producing solver and $e_{\text{prod}}(c)$ with the current epoch. Note that the attempted insertion in $\mathcal{B}$ presents an additional quality-dependent barrier for sharing c and may on success delete "worse" clauses in $\mathcal{B}$. If $c \in H$ but the insertion in $\mathcal{B}$ was not performed (successfully), we still add the calling solver to $p(c)$ and update $e_{\text{prod}}(c)$.

Clause sharing and filtering in epoch e works as follows:

- **Export:** Each process flushes clauses from $\mathcal{B}$ up to a certain total length l. The sharing buffer B of globally best clauses is aggregated and then broadcast to all processes as described in Section 4.3.2.
- Each process iterates over each clause $c_i \in B$, $i \geq 0$ and constructs a bit vector $\tilde{v}$ where $\tilde{v}[i] = q_{c_i} := [c_i \in H \land e \leq e_{\text{sh}}(c_i) + z] \in \{0, 1\}$. As such, the i-th bit of $\tilde{v}$ represents whether the process remembers that c_i was shared and admitted for import in one of the last z epochs.
- All local bit vectors $\tilde{v}$ are reduced to a single *filter vector* v via bitwise OR operations. v is aggregated and then shared among all processes just like B.
- **Import:** Each process iterates over B and v and only admits clause c_i for import if $v[i] = 0$. Each admitted clause c is inserted into the import buffer $\mathcal{B}_{S_j}$ of each local solver S_j if $(c \notin H \lor j \notin p(c))$. If $c \in H$ for an admitted clause c, then c is *marked as shared*: $e_{\text{sh}}(c) := e$ and we reset $p(c) := 0$ after its use.

Our filter has two purposes: filtering clauses shared recently (*distributed filtering*) and preventing clauses from being mirrored back to their producers (*by-solver filtering*). For the former purpose, we establish our filter's correctness with the following theorem:

Theorem 4.1 (Correctness of distributed filtering)
The described approach admits a shared clause c for import in epoch e if and only if c has not been admitted for import in epochs $e - z, \ldots, e - 1$.

Proof. Assume that some process shares c in epoch e. The following chain of reasoning holds in both directions:

c is admitted for import in epoch e
$\Leftrightarrow$ at epoch e, $q_c = 0$ on all processes
$\Leftrightarrow$ at epoch e, $c \notin H \lor e > e_{\text{sh}}(c) + z$ on all processes
$\Leftrightarrow$ at epoch e, $e_{\text{sh}}(c) < e - z$ on each process with $c \in H$
$\Leftrightarrow$ $\forall\, e' \in \{e - z, \ldots, e - 1\}$, no process has marked c as shared
$\Leftrightarrow$ $\forall\, e' \in \{e - z, \ldots, e - 1\}$, c has not been admitted for import.

The last equivalence holds due to the following argument: A shared clause c always has at least one process of origin where $c \in H$. Since each such process with $c \in H$ marks c as shared if and only if c is admitted for import, it follows that a clause c is admitted for import if and only if there is a process which marks c as shared. □

In terms of by-solver filtering, an exact approach may guarantee the following: If a solver S produces clause c, then c will not be handed to S for exactly z epochs. We relax two aspects of this guarantee to allow for a more efficient implementation.

First, if z epochs expire without c being shared and admitted, then c may still be blocked *once* from being handed to S in a later epoch. This is because the epoch of production $e_{\text{prod}}(c)$ is a field shared between all local solvers. Other solvers producing c update $e_{\text{prod}}(c)$ as well, which can prolong the filtering status of c. This inaccuracy can be eliminated by replacing $p(c)$ and $e_{\text{prod}}(c)$ with a vector of production epochs, one for each local solver, at the cost of storing additional data for each clause in H.

Secondly, we only employ by-solver filtering for clauses which are successfully inserted or updated in H at the time of production. A produced clause c which does not meet the requirements to fit in $\mathcal{B}$ is not inserted in H but may still be shared successfully by another process at a later point in time. This is possible if, at the time of production of c, $\mathcal{B}$ is full and some of the later incoming clauses are of equal or worse quality than *all* clauses in $\mathcal{B}$. Since we configure $\mathcal{B}$ to hold x times the export volume β per epoch ($x = 10$ in our implementation), this scenario is unlikely except if the distribution over produced clause quality changes suddenly, e.g., if solvers produce batches of particularly short clauses during some pre– or inprocessing. We could eliminate this inaccuracy completely if we inserted every produced clause in H no matter its quality, which would increase the filter's memory footprint.

The communication cost of our clause filtering approach is a second all-reduction whose span, work, and communication volume are dominated by the corresponding clause exchange operation. Each process now needs to iterate over the shared clause buffer twice—once for filtering and once for importing clauses. On each process, H requires memory linear in the volume of locally produced clauses which were inserted in $\mathcal{B}$ in the last $\mathcal{O}(z)$ epochs. Such an insertion, in turn, is done only if the volume of clauses in $\mathcal{B}$ of equal or higher quality does not exceed the capacity of $\mathcal{B}$.

4.3.5 Compensating for Unused Sharing Volume

The volume of clauses successfully shared across processes can often stay behind the sharing volume which we target with our scaled sharing buffer limit (Section 4.3.2.b). There are three causes for this effect: (i) the processes do not produce enough clauses that are admissible for sharing; (ii) duplicate clauses are detected and consequently eliminated during aggregation; (iii) our filtering mechanism leads to the rejection of some transmitted clauses. While (i) is a normal occurrence, e.g., at the beginning of a large formula's processing, we wish to compensate for the sharing volume that remained unused for algorithmic reasons, i.e., due to causes (ii) and (iii).

For this purpose, we multiply the buffer limit for epoch e with a *compensation factor* $\kappa^{(e)}$ which is calculated based on statistics of recent sharings.

Let X_{target} be the targeted number and X_{actual} the actual number of successfully shared literals so far. Let $\tilde{x}_{\text{in}}$ be an estimate for the number x_{in} of *incoming literals* next sharing (i.e., all literals contributed by all processes *before* aggregation and filtering) and let $\tilde{x}_{\text{out}}$ be an estimate for the number x_{out} of successfully shared literals. We keep $\tilde{x}_{\text{in}}$ and $\tilde{x}_{\text{out}}$ normalized by κ. Since we want the next sharing of expected effective size $\kappa\tilde{x}_{\text{out}}$ to not only meet the anticipated sharing volume $\tilde{x}_{\text{in}}$ but to also compensate for the discrepancy $X_{\text{target}} - X_{\text{actual}}$, we target $\kappa\tilde{x}_{\text{out}} = X_{\text{target}} - X_{\text{actual}} + \tilde{x}_{\text{in}}$. Therefore we set $\kappa = \min\{\kappa_{\max}, (X_{\text{target}} - X_{\text{actual}} + \tilde{x}_{\text{in}})/\tilde{x}_{\text{out}}\}$, where $\kappa_{\max}$ is a small constant that presents an upper bound for κ ($\kappa_{\max} = 5$ in our configuration). We obtain estimates $\tilde{x}_{\text{in}}$ and $\tilde{x}_{\text{out}}$ using "elastic" updates with update factor δ:

$$\tilde{x}_{\text{in}}{}^{(e+1)} := \delta\tilde{x}_{\text{in}}{}^{(e)} + (1-\delta)\frac{x_{\text{in}}{}^{(e)}}{\kappa^{(e)}}$$

$$\tilde{x}_{\text{out}}{}^{(e+1)} := \delta\tilde{x}_{\text{out}}{}^{(e)} + (1-\delta)\frac{x_{\text{out}}{}^{(e)}}{\kappa^{(e)}}$$

These estimates allow our sharing to react to changes in the distribution over produced, filtered, and duplicate clauses. Instead of aggregating X_{target} and X_{actual} exactly over the entire execution time, we decay these values over time. As such, missed out sharing volume is either compensated for in a timely fashion or otherwise forgotten over time. Limiting κ to $\kappa_{\max}$ helps to keep communication manageable at all times and also aims to distribute larger amounts of compensation over multiple epochs rather than performing a single huge sharing with comparably bad clauses.

4.3.6 Handling LBD Values

Each clause c a sequential SAT solver produces is associated with a particular LBD value. We refer to Section 2.2.3.c for a general introduction to LBD values and Section 2.3.2.b for a discussion in the context of parallel SAT solving. We reiterate how prior systems handle LBD values and then describe our own approach.

HordeSat considers a clause's LBD value a fixed part of the clause during export, aggregation, and import and has each solver import each clause together with its original LBD value. Since LBD is an essential metric for clause quality in sequential SAT solving, many SAT solvers keep clauses with an LBD value of at most 2 *indefinitely* and never consider to delete them [AS09; Bie+20a]. These solvers, however, are usually tuned to expect a single solver's worth of clauses. In massively parallel systems, a statistical argument can be made that the overall volume of such "very good" clauses can become very large compared to a sequential execution [EN19]—especially if we prefer sharing low-LBD clauses. As such, the solvers' clause databases may grow in size significantly, leading to running time overhead and increased memory footprints.

By contrast, in the system TopoSAT 2 each LBD value is reset to $|c|$, i.e., an upper bound on possible LBD values, before c is imported [EN19]. This lets a solver prioritize the clauses it produced itself over external clauses and presumably leads to many shared clauses being deleted quickly after a brief probation period [AS14].

In our system, we have implemented both LBD handling approaches and propose a third alternative: At the import of a clause c, we *increment* its LBD value. With this measure, we preserve the LBD-based prioritization of (most) shared clauses while also ensuring that a solver prefers local clauses and retains authority over which of its clauses are kept indefinitely. As such, we expect that this strategy keeps clause databases more manageable than with HordeSat-style LBD handling and still allows the solvers to make good use of the shared clauses' original LBD values.

4.4 Achieving Diversity

Intuitively, diversification of solvers has two important merits. First, if different solvers explore different subspaces, the probability is higher for some solver to arrive at a satisfying assignment. Secondly, starting off the solvers along different directions leads to different learned clauses, which results in more diverse sets of shared clauses.

As in HordeSat, our approach relies on different sources of diversification. First, our solver takes a certain *portfolio policy* π as an input, e.g., $\pi = \langle k, k, c, l, g \rangle$ for Kissat-Kissat-CaDiCaL-Lingeling-Glucose, and then maps the i-th solver thread in the computation ($i \geq 0$) to the solver backend $\pi[i \mod |\pi|]$. Specifically, if each process runs c solvers, then the j-th process of MallobSat ($j \geq 0$) launches solver threads with $i \in \{j \cdot c, \ldots, (j+1) \cdot c - 1\}$. Secondly, each solver thread has a *diversification index* x, which indicates that it is the x-th thread running this particular solver backend. x is used to cycle through solver-specific diversification options. Lastly, a distinct *diversification seed* is computed for each thread based on i, which is used for random decisions within the solver and for additional random diversification (Section 4.4.2).

4.4.1 Solver Portfolio

We have integrated several different sequential SAT solvers in our system. In addition to an updated version of HordeSat's Lingeling [Bie10] backend, we support the popular and widely used solver Glucose [AS09] (a fork of Minisat [ES04]) as well as the two state-of-the-art solvers Kissat and CaDiCaL [Bie18; Bie+20a].

We provide all used configurations in Tab. 9.1 (Appendix B). We picked most configurations and their ordering by measuring their individual 32-core performance on the ISC 2020 benchmark set and then greedily adding the next best configuration to a virtual pure portfolio until improvements become negligible (*cf.* [BIB22]). While we found this approach to result in good performance, dedicated machine learning approaches may be able to further improve our setup (*cf.* [Xu+12; Bie15; BIB22]).

For the initial version of MallobSat [Sch20; SS21b], we focused on **Lingeling** as an efficient and reliable SAT solver with well-performing diversification options from Plingeling [Bie14]. Note that Lingeling features some non-standard reasoning, such as Gaussian elimination and cardinality constraint reasoning [Bie12; Bie13].

Due to these techniques, Lingeling has a crucial advantage on some unsatisfiable instances compared to many other solvers [IMM17]. While these techniques cannot be expressed in terms of general resolution and therefore have to be disabled for certified SAT solving [Bie13], we are able to exploit them for (uncertified) parallel SAT solving. We use the most recent version of Lingeling in terms of sequential solving features [Bie18]. Similarly, we use most CDCL diversification options from the latest Plingeling [Bie18]. Every eleventh solver thread uses local search solver **YalSAT** (integrated in Lingeling), alternatingly with and without preprocessing.

For the **Glucose** [AS09] interface, we used some of the sources of Syrup [AS14] to import and export clauses asynchronously. This includes the technique of exporting clauses only after they have been encountered for the second time. We adopted and adjusted the diversification of Syrup, which includes different scheduling strategies for clause deletion and restarts, toggling simplification techniques, and a dynamic adaption of some parameters after the first x conflicts. In addition, the first search descent and initial variable activities are randomized.

For **CaDiCaL** [Bie17], we used the existing clause export interface and implemented a clause import interface—a first version of this interface was provided by Maximilian Schick [Sch21a]. We diversify CaDiCaL instances via its `sat` and `unsat` presets, randomizing restart intervals, and toggling individual options such as random walks, bounded variable elimination, and inprocessing in general. For **Kissat** [Bie+20a], we implemented our own clause import and export mechanism based on our CaDiCaL variant and employ diversification similar to CaDiCaL. In the latest versions of our solver interfaces, we let the solvers try to import arrived clauses whenever at decision level 0—in earlier versions we had Kissat find a minimum of 500 conflicts between attempted clause imports, which can increase clause turnaround times.

Note that certain inprocessing, such as bounded variable elimination, has a peculiar impact on clause sharing: Incoming clauses which feature a literal that has been eliminated in the importing solver cannot be imported but must be discarded [Bie13]. We performed measurements with our Kissat backend and observed that this can, in some cases, lead to around 90% of incoming clauses being discarded by individual solvers. Disabling these inprocessing techniques in some solvers can therefore allow them to import much richer sets of clauses. We also experimented with completely disabling such inprocessing for many or even all solvers but were not able to observe improved performance. We believe that enabling full clause sharing despite extensive pre- and inprocessing is an important line of future work, especially considering the recent emergence of new powerful preprocessing techniques [HGH23; RB23].

4.4.2 Diversification Techniques

In the following, we present the diversification techniques we employ beyond the hand-crafted sets of solver configurations.

Sparse random variable phases. A *variable phase* in a solver decides which value to assign to the variable if it is chosen as a decision variable (see Section 2.2.3.c).

HordeSat features a diversification technique where in a run with p solvers, each variable's initial phase in a solver is overridden with probability $1/p$ [BSS15]. The variable phase is then determined by a coin flip. We adopted this technique in MallobSat; it is enabled after configuration-based diversification is exhausted. For Kissat, we implemented an option to provide a vector of initial variable phases since this was not part of its original interface.

Input permutation. There is no formal notion of order among the clauses in a CNF formula F—any permutation of a given sequence of clauses is logically the same input to a SAT solver. In practice however, the order in which clauses are given to a SAT solver can make a difference in terms of the internal data structures and the decisions made by the solver, therefore leading to considerable variation in running time [BH19]. For this reason, permuting the input clauses before handing them to a solver can be exploited as an additional source of diversification. In our implementation, all but the first ten solver threads have a 50% chance to perform input permutation.

A formula arrives at each of our SAT solving processes in the form of a flat array of integers. For very large formulas,[4] it can take tens of seconds for each solver thread to import all clauses. Permuting this input by explicitly reordering the data for each solver is unacceptable in terms of running time and/or additional memory usage. Creating a pointer to each clause and then permuting these pointers is more viable in practice but leads to highly irregular memory accesses and thus cache-unfriendly behavior while importing the permuted formula.

We select up to $k = 128$ clauses to which we store a pointer. The first clause in the input is always selected while the remaining $k - 1$ clauses are selected at random. As such, each of the k pointers represents a chunk of the input beginning at the referenced clause and ending at the next pointer's address or at the end of the input data. These k pointers are then permuted and the input chunks are read in the corresponding order. This procedure remains cache-friendly for large inputs. It cannot yield all possible $n!$ clause permutations, but for any pair of clauses (c_1, c_2) in the input there is a non-zero probability that the order of c_1 and c_2 is reversed. Overall, the approach allows for $\binom{n}{k} \cdot k!$ different permutations of the input since there are $\binom{n}{k}$ possible selections of pointers and $k!$ ways to permute them.

Noisy numerical parameters. To further diversify solvers, we suggest to add a small amount of random noise to certain numerical parameters. For each such parameter, we sample a number from a Gaussian distribution centered at zero and then add the number to the parameter's default. We have chosen restart intervals and variable score decays as promising candidates for this kind of randomization.

[4]The largest instance of ISC 2022, `SAT_MS_sat_nurikabe_p16.pddl_166.cnf`, has 213 million clauses. It results in a serialization with more than 712 million integers (including clause separators) and, therefore, around 2.65 GB of raw data.

4.5 Technical Improvements

In addition to the presented main ingredients to our system—clause sharing and diversification—we present a number of improvements on a technical level which contribute to the efficiency and viability of our system.

4.5.1 Memory Awareness

The memory consumption of parallel portfolios is a known issue [IBS19; FB22]: As each solver commonly maintains its own clause database, the increase in memory requirements is proportional to the number of employed solvers. As such, our system executed on large formulas can cause nodes to run out of main memory. To counteract this issue, we introduce a simple but effective step of precaution: For a given threshold $\hat{s}$, if a given serialized formula description has size $s > \hat{s}$, then only $t' = \max\{1, \lfloor t \cdot \hat{s}/s \rfloor\}$ threads will be spawned for each process. The choice of $\hat{s}$ depends on the amount of available main memory per process. The system we used only features 2 GB of RAM per solver if all physical cores are used. Based on monitoring the memory usage for different large formulas in our system, we use $\hat{s} := 50 \cdot 10^6$. As t' only depends on s, the t' threads can be started immediately without any further inspection of the formula.

The above step of precaution can be effective for some inputs, but it does not address all issues. Most significantly, memory usage which is initially acceptable but then grows to unsustainable levels is not accounted for. We thus introduce an additional measure to counteract excessive memory usage. At program start, we create a communication group for the MPI processes at each physical machine. In other words, we identify groups of processes with a shared RAM budget. Each group periodically checks the current memory usage of its machine and exchanges certain diagnostics for each process. If a certain memory limit is exceeded ($> 90\%$ of RAM used), one or multiple processes are chosen to trigger a *memory panic* (*cf.* [AS14]). The heuristic which decides on the particular process(es) considers the memory used by each process as well as the importance of its role in the portfolio—processes closer to the job tree's root are considered more important. A memory panic at a particular process triggers the termination and clean up of some of the process' solver threads.

4.5.2 Preemption of Solvers

To support malleability, it is essential that a process's management thread can suspend, resume, and terminate each job node at will. We noticed that we cannot rely on each solver thread periodically calling an according callback function because a solver can sometimes get stuck in expensive preprocessing and inprocessing [Bie16b] for several minutes. To still enable quick preemption, we enabled our solver's workers to be run as a separate *subprocess*. This incurs some overhead as a new process is forked, a shared memory segment for efficient inter-process communication (IPC) is set up, and the subprocess runs an additional management thread. However, suspension and termination of a process is supported on the OS level in a safe manner through *signals*.

As solver threads may be unresponsive when the subprocess catches a termination signal, they are interrupted and cleaned up forcefully. We can also adjust each subprocess in such a way that it is killed first by the operating system if a machine runs out of main memory [Sch23]. In both cases, this leaves the main process and the distributed computation in a valid state. Last but not least, performing SAT solving in a subprocess allows for a kind of fault tolerance: If a solver crashes,[5] the concerned subprocess can be restarted without disrupting the distributed solving effort.

4.6 Evaluation

We now turn to the evaluation of our work. After explaining the experimental setup, we first evaluate MallobSat on individual inputs at fixed scales. Then we evaluate MallobSat within our malleable job scheduler Mallob (Chapter 3), scheduling multiple distributed solving procedures in parallel. Our software and experimental data are available online (see Appendix A).

4.6.1 Experimental Setup

We implemented MallobSat in C++17 as an application engine within Mallob. For implementation details on Mallob we refer to Section 3.5.

We updated HordeSat to also use the latest Lingeling+YalSAT [Bie18] backend. Furthermore, we fixed a performance bug in HordeSat: Lingeling frequently queries the time elapsed since its initialization. In the original code, no callback providing this elapsed time was given to Lingeling which caused each solver thread to fall back to expensive system calls. Depending on the configuration this led to each solver spending more than 10% of its time in kernel mode.

We performed our experiments on SuperMUC-NG (see Section 3.6.1) where each machine features two 24-core sockets, i.e., two sockets with 48 hardware threads each. We allocated up to 134 machines ($= 134 \times 2 \times 24 = 6432$ cores) at a time, mapping each MPI process either to four cores as in HordeSat or to an entire socket of 24 cores. We compiled our software with GCC 11.2 and Intel MPI 2019.12.

We tested the parallel solvers on a comparably low wallclock time limit of 300 seconds because we are interested in solving SAT instances as rapidly as possible in a costly large-scale distributed environment, where we invest more resources than with sequential solving by multiple orders of magnitude.[6]

Performance metrics. We rate solver runs based on solved count and PAR-2 score (see Section 2.4.2.a). To compute PAR-2 scores restricted to satisfiable and to unsatisfiable instances, we assume that there are equally many satisfiable and unsatisfiable instances.

[5]For instance, we experienced occasional crashes of Lingeling on an instance named `lang28.cnf.gz.CP3-cnfmiter.cnf` where Lingeling's *simple probing* mechanism [Bie12] results in vast amounts of irredundant clause literals which eventually exceed a hard limit in the code.

[6]In the ISC, the by-instance CPU timeout in the cloud track (1000 s $\times$ 1600 hardware threads) exceeds that of the sequential main track (5000 s $\times$ 1 hardware thread) by a factor of 320.

	Mean RAM	#	PAR-2	CSAR
$16 \times 12 \times 4$	342.0	**323**	147.1	36.1
$16 \times 2 \times 24$	**286.0**	320	**142.3**	**27.9**

Table 4.1: Impact of process layout, written as (# machines) × (# processes per machine) × (# solvers per process). The better result per column is highlighted.

In addition, we use CSAR scores (Section 2.4.2.a) in cases where PAR-2 scores and solved instances are similar.

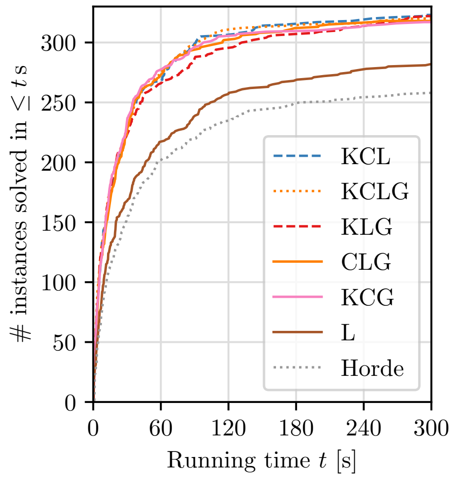
Selection of benchmarks. For the unbiased evaluation of SAT solvers, a benchmark set consisting of many formulas from diverse origins is necessary. For the evaluation of the modules and parameters of our system we used the 349 benchmark instances from ISC 2022 [Bal+22a] which *some* solver was able to solve (across all tracks). This may lead to slight underestimation of a run's performance but drastically reduces running times and, consequently, resource usage. For instance, our baseline configuration spent 4.9 h on the 349 instances whereas it would have spent up to 9.2 h on all 400 instances with unchanged results, assuming it to be unsuccessful on the instances which remained unsolved in the competition. To evaluate the scaling behavior and speedups of our system, we use a different benchmark set in order to reduce overfitting effects. We use the ISC 2021 benchmark set [Bal+21a] consisting of 400 instances.

4.6.2 SAT Solving Configuration

We begin our experiments with a pretuned configuration of MallobSat obtained by a series of explorative tests on a random selection of 125 instances from the ISC 2022 Anniversary benchmark set. This configuration features our latest techniques and data structures; two sharings per second with compensation for unused sharing volume;[7] distributed filtering with a resharing period of 30 epochs (15 s); incrementing each LBD score before import; a clause buffer limit based on limited growth ($L = 100\,000$); and a process setup faithful to the hardware used, i.e., with one MPI process for each socket of 24 cores. In the following, we adjust individual components and/or parameters of our system and analyze the respective differences. If not indicated otherwise, we use a setup with 768 cores (16 machines). We chose this scale as a compromise between making responsible use of resources and still being able to observe effects that are specific to distributed solving (such as fully exhausted diversification in terms of distinct solver configurations).

Process layout. We begin with the observation that our setup faithful to the hardware at hand is indeed beneficial for performance and memory usage (Tab. 4.1). In a direct

[7]The "Process Layout" and "Portfolio" runs in this section and the runs in Sections 4.6.5 and 4.6.6 updated X_{target} in a way that erroneously led to more clause sharing than anticipated—in total around 2× the volume intended with $L = 100\,000$. All other experiments use corrected updates of X_{target} and, as discussed later, switch to $L = 250\,000$ at some point.



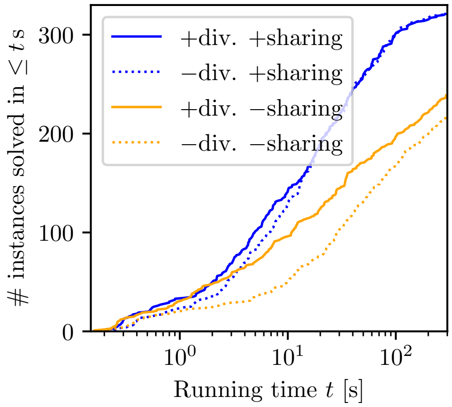
	overall		SAT		UNSAT	
Portfolio	#	PAR	#	PAR	#	PAR
KCL \ G	**323**	**140.1**	**158**	141.9	**165**	138.4
KCLG	320	142.3	156	145.4	164	139.2
KLG \ C	322	145.9	**158**	148.9	164	143.0
CLG \ K	318	146.2	154	154.2	164	**138.1**
KCG \ L	317	146.7	**158**	**140.4**	159	153.0
L	283	206.6	126	238.8	157	174.4
L (Horde)	253	243.8	121	254.3	132	233.2

Figure 4.4: Performance of MallobSat with different solver portfolios and of HordeSat. MallobSat employs combinations of Kissat (K), CaDiCaL (C), Lingeling (L), and Glucose (G).

comparison of our most recent "16 × 2 × 24" setup with the earlier HordeSat-style "16 × 12 × 4" setup, we arrived at a PAR-2 score of 142.3 for the former and 147.1 for the latter, although the latter setup was barely able to solve three additional instances close to the time limit. Our new setup's advantage in terms of CSAR scores is even larger. More importantly, our new setup reduced mean RAM usage[8] by more than 16% (286 GB down from 342 GB), mostly due to the fact that fewer copies of the formula are present on each machine.

Portfolio. To assess the impact of individual solver backends, we performed cross-checking by omitting one solver backend at a time from our portfolio. We also included a Lingeling-only portfolio and a run of HordeSat (also with Lingeling). Fig. 4.4 shows results. First, MallobSat significantly outperforms HordeSat if both use the same solver backend, solving 30 more instances overall. Secondly, all diverse portfolios of MallobSat significantly outperform MallobSat's Lingeling-only portfolio. Across the portfolios with three to four solver backends each, we only observed modest differences in performance. Kissat appears to be most important for satisfiable instances: Its omission degraded performance on satisfiable instances whereas the three portfolios with the highest Kissat ratio (one out of three) performed the best. By contrast, Lingeling seems to be the most important solver for unsatisfiable instances, presumably due to its advanced reasoning techniques (Section 4.4.1).

[8]RAM usage is measured by aggregating the global Resident Set Size (RSS) main memory usage of all Mallob processes, including SAT subprocesses, every second.



		overall		SAT		UNSAT	
div.	sh.	#	PAR	#	PAR	#	PAR
+	+	321	143.4	155	149.1	166	137.6
−	+	321	143.5	156	146.9	165	140.0
+	−	239	271.7	152	162.4	87	381.0
−	−	217	308.7	138	215.6	79	401.9

Figure 4.5: Impact of diversification (seeds, phases, permutation, noise) and clause sharing, one by one and together. Note the logarithmic scale in x direction.

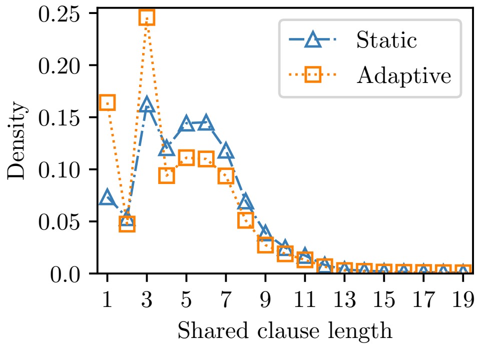
Our CaDiCaL backend appears to be a solid all-rounder since its omission results in increased running times for both satisfiable and unsatisfiable instances. Glucose is the only solver whose omission results in slightly improved overall performance. Removing Glucose has two further merits: Glucose tends to spend the most amount of memory, and Glucose is the only part of our system with a non-free license.[9] For these reasons, we continue our evaluation with a portfolio of Lingeling, CaDiCaL, and Kissat (at equal parts).

Diversification. Since we were not able to isolate significant performance differences between enabling and disabling individual diversification techniques, we show results for all of our diversification put together—random seeds, initial phases, input permutation, and noisy parameters—and for all of it disabled. The results (Fig. 4.5) were surprising to us: Disabling all such diversification, which leaves less than 40 distinct solver procedures across 768 cores, led to almost unchanged performance. We repeated the same experiments with clause sharing disabled and found that our diversification does have a significant merit in that case. The conclusion we draw is that our clause sharing effectively *subsumes* many common forms of diversification, resulting in its own kind of work subdivision. Note how in the first few seconds of solving, diversification is in fact beneficial for our clause-sharing solver: Without our diversification, the exact same SAT solving code is executed many times redundantly which results in bad performance. After a brief initial period, however, solvers begin to differ due to shared clauses arriving at slightly different points in time, and quickly these differences become significant enough such that clause sharing is able to function as a kind of distributed search space pruning.

[9]The use of Glucose in any competitive event without the authors' explicit permission is prohibited.

	Mean RAM	Median RAM	#	PAR-2	CSAR
Original LBD	269.0	99.1	321	143.6	31.3
Reset LBD	269.0	**92.5**	**322**	**143.2**	32.2
Increment LBD	**267.1**	93.5	321	143.4	**30.0**

Table 4.2: Impact of LBD handling in terms of RAM usage (in GB) and scores.

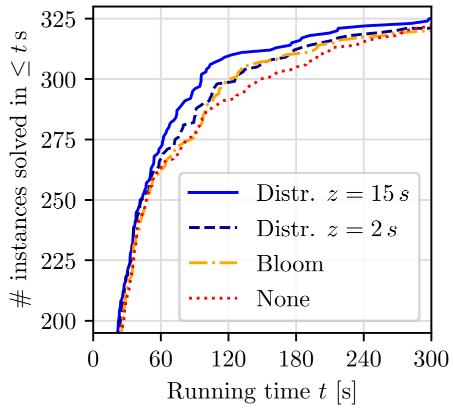


	#	PAR-2	CSAR
Both adaptive	**321**	143.4	27.9
Static export	**321**	**141.9**	**26.5**
Static import	320	146.5	31.9

Figure 4.6: Impact of different clause buffers. The left figure shows a histogram over successfully shared clauses' lengths for adaptive and static export buffers.

Handling LBD values. We compare our strategy of incrementing each incoming LBD value with the established approaches of using each LBD as is [BSS15] and resetting each LBD at import [EN19]. As Tab. 4.2 shows, editing LBD value appears to make a difference regarding (median) memory usage. While the differences in PAR-2 performance are insignificant, we found our strategy to result in a slightly lower CSAR score than both other strategies. As such, we consider our strategy an appealing combination of good performance with reduced memory requirements.

Clause buffering. Next we evaluate our clause buffering techniques. Compared to HordeSat-style export buffers (with bucket sizes adjusted to our export buffer size), our adaptive export buffers reduce the mean shared clause length from 6.5 to 5.7. As shown in Fig. 4.6 (left), unit and ternary clauses are much more likely to be shared whereas longer clauses become less likely. Binary clauses are produced less frequently than unit or ternary clauses, hence both buffer types can handle their modest volume well. Despite sharing shorter clauses, adaptive export buffers resulted in no improvement but rather a moderate decline in performance (Fig. 4.6 right). A possible explanation is that the unlimited buffering of unit clauses may in fact be detrimental for some instances where huge amounts of unit clauses arise and prevent other clauses from being shared. Our adaptive import buffering, on the other hand, outperformed the earlier approach based on lock-free ring buffers [SS21b].



	#	PAR	CSAR
No filtering	322	148.4	—
Bloom filters	321	147.5	—
Distr., $z = 2\,s$	321	145.6	31.78
Distr., $z = 4\,s$	324	141.4	30.49
Distr., $z = 7.5\,s$	**325**	139.6	29.38
Distr., $z = 15\,s$	**325**	**138.8**	**28.47**
Distr., $z = 30\,s$	322	141.9	28.48
Distr., $z = \infty$	322	141.8	28.55
Distr., $z = 7.5\,s$, NC	322	145.5	—

Figure 4.7: Left: CDFs for the best and worst performing distributed filter configuration as well as for Bloom filtering and no filtering (note the offset of the y axis). Right: Performance of filter strategies, also including a run with no compensation ("NC") for unused sharing volume.

Clause filtering. In terms of clause filtering, we compared our distributed filter at different resharing periods z with HordeSat-style filtering (Section 4.3.4.b) and no filtering at all. MallobSat counts the resharing period in terms of sharing epochs; for the sake of simplicity we display z in terms of seconds. As MallobSat performs two sharings per second, $z = 7.5\,\text{s}$ is equivalent to filtering clauses for 15 epochs.

We provide results in Fig. 4.7. Disabling clause filtering completely resulted in worst performance. All distributed filtering runs resulted in improved PAR-2 scores compared to Bloom filtering. Distributed filtering at blocking period $z = 15\,\text{s}$ performed the best in terms of PAR-2 scores and performed similar to higher z in terms of CSAR scores. It appears to be particularly beneficial to filter clauses which are re-shared after a brief period. Clauses which re-occur after an extended period are less important to filter and might even be useful to admit for sharing again. Removing clauses from the filter after some time also keeps the memory footprint more manageable for higher time limits. The total ratio of clauses which were admitted ranges from 43.8% ($z = \infty$) to 75.1% ($z = 2\,\text{s}$). A resharing period of $z = 15\,\text{s}$ is sufficiently high to block more than half of all clauses (47.8% of clauses admitted) at this sharing volume.

Upon closer inspection, we found that the instances which profit the most from our clause filtering (speedup ≥ 2) are unsatisfiable model checking tasks. These include nine of "*some of the hardest SAT problems generated by the bounded model checker CBMC [CKL04] when verifying open-source code at AWS*" [Fof+22], two string safety problems also encoded with CBMC [OW21], and one unsatisfiable automated planning problem with a bounded number of steps [Fro20b]. Classical automated planning tasks and bounded model checking tasks are structurally similar (see Section 2.2.5).

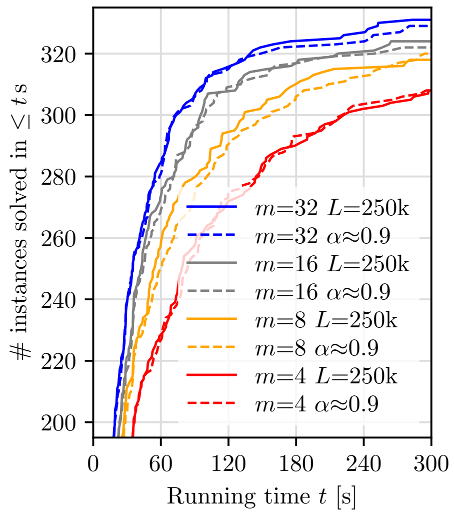
	#	PAR-2	CSAR
3/s	323	140.9	**30.0**
2/s	**325**	**138.8**	30.1
1/s	321	144.5	32.4

Table 4.3: Impact of clause sharing frequency.

Unused sharing volume compensation. To assess how well our compensation technique (Section 4.3.5) makes up for the significant ratio of filtered clauses, we also performed a run at resharing period $z = 7.5\,\mathrm{s}$ without such compensation—neither for filtered clauses nor for clauses identified as duplicates during aggregation. We observed worse results, underperforming all other distributed filtering runs except for $z = 2\,\mathrm{s}$ (see Fig. 4.7 left). In total, the run without compensation exchanged 80.5% and admitted for sharing 87.5% of the literals which the corresponding run with compensation exchanged/admitted. MallobSat with compensation reaches 88.6% of its total targeted sharing volume while MallobSat without this technique only reaches 72.3%. The sharing volume that remains unused despite our compensation is, in large parts, due to points in time where insufficient amounts of distinct clauses are available for sharing in the first place. Overall, our compensation mechanism proved to be a relevant contribution to the performance of our clause sharing approach.

Frequency of clause sharing. We ran some experiments to assess the impact of the frequency at which sharing is performed. We tried frequencies of 1/s, 2/s, and 3/s and resized the base buffer size per process by a factor of 1, 1/2, and 1/3 respectively to keep the overall sharing volume fixed. The PAR-2 and CSAR scores given in Table 4.3 indicate that performing clause sharing more than once a second is indeed beneficial for performance. Note that this result contrasts earlier work [Aud+17] where more frequent all-to-all sharing resulted in worse performance. Clearly, our clause sharing implementation and its embedding in the solver processes is sufficiently scalable to profit from reduced clause turnaround times while the added overhead is negligible. Two sharings per second performed the best in terms of PAR-2 scores while three sharings per second performed similarly well in terms of CSAR scores.

Buffer limit scaling. To assess our buffer limit scaling strategies, we first identified a sharing volume with which our system performs well on a relatively large scale of 32 nodes (1536 cores). The adjusted parametrization we use is a limit of $L = 250\,000$ literals per sharing and, equivalent in terms of sharing volume at this scale, a discount factor of $\alpha = 0.903121$. We then used these parameters on successively lower scales to see how well each function is suited to be used across all scalings with a fixed parametrization. Fig. 4.8 shows that our function based on recursive discounts results in similar performance at 4 nodes and in worse performance at 8 nodes and above. Indeed, our more recent approach appears to adapt better to the different scales. For the following experiments, we consequently use $L = 250\,000$.



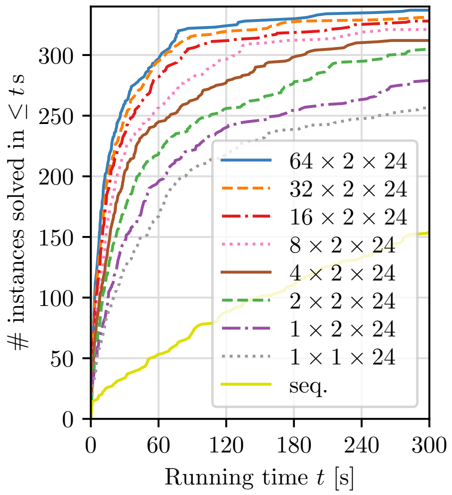
m	Variant	Lits/s	#	PAR	CSAR
32	$L = 250$k	345 274	331	127.4	27.5
32	$\alpha \approx 0.9$	345 274	329	130.2	28.5
16	$L = 250$k	221 835	324	139.2	29.6
16	$\alpha \approx 0.9$	190 728	322	141.7	30.5
8	$L = 250$k	127 031	318	151.7	36.0
8	$\alpha \approx 0.9$	105 132	320	153.2	39.3
4	$L = 250$k	68 126	308	173.6	45.6
4	$\alpha \approx 0.9$	57 718	308	173.7	45.3

Figure 4.8: Impact of buffer scaling strategies for a common, fixed sharing volume at $m = 32$ nodes. Note the offset of the y axis in the left figure. CSAR scores are computed separately for each pair of competing configurations.

4.6.3 Scaling and Speedups

In order to assess the scalability of our tuned system, we use the set of 400 benchmarks from ISC 2021. We run MallobSat on 1, 2, 4, ..., 128 24-core sockets (i.e., up to 3072 cores on 64 machines) with one solver per physical core. As a sequential baseline we run KissatMABHyWalk [Zhe+22], the winning sequential solver from ISC 2022, without proof logging. We set the sequential solver's time limit per instance to a very high 115 200 s (32 hours). In terms of CPU time, this limit is equivalent to the maximum CPU time per instance for our run with 384 cores. We limited the main memory for each sequential run to 12 GB.

Fig. 4.9 shows the scaling behavior of our MallobSat configuration. Performance increases whenever computational resources are doubled, including the largest scale of 3072 cores. Improvements do diminish noticeably beyond 16 nodes (768 cores). On satisfiable instances, the number of solved instances only increases marginally from 96 cores onward whereas average running times improve consistently up to 3072 cores. On unsatisfiable instances, we observe more pronounced scaling up to 3072 cores both in terms of instances solved and in terms of PAR-2 scores. At the largest scale tested, our system solves 19 more unsatisfiable instances than satisfiable instances. While this may partly be a result of the considered benchmark set, we later show that clause sharing especially benefits unsatisfiable instances (Section 4.6.4), which may indicate that more work is required to achieve similar benefits for satisfiable instances.



		overall		SAT		UNSAT	
m	cores	#	PAR	#	PAR	#	PAR
1	24	257	251.3	126	256.1	131	246.5
1	48	279	221.2	139	222.6	140	219.7
2	96	305	184.0	150	187.3	155	180.7
4	192	312	164.2	153	169.3	159	159.1
8	384	321	147.2	156	159.1	165	135.3
16	768	328	132.4	156	151.1	172	113.8
32	1536	331	124.4	157	145.3	174	103.5
64	3072	337	115.1	159	137.2	178	92.9
seq.		154	414.0	87	389.1	67	438.9

Figure 4.9: MallobSat scaling results for an increasing number of machines m. We also include sequential baseline KissatMABHyWalk ("seq.") with the same wallclock timeout of 300 s per instance.

Tab. 4.4 lists the speedups of our system compared to the currently best sequential solver. In an effort to provide as clean and authentic measures as possible, at each scale we only consider the set of instances which both the sequential approach and the parallel approach were able to solve (see Section 2.4.2.c). We provide median and geometric mean speedups as well as total speedups (see Section 2.4.2.b).

KissatMABHyWalk was able to solve 331 instances. As such, our system at 1536 cores is able to solve the same number of instances within five minutes per instance as a state-of-the-art sequential solver can solve within 32 hours per instance. That being said, the parallel approach at this scale was allowed to use 4× the CPU time of the sequential baseline. In contrast to previously reported behavior of HordeSat [BSS15], our speedup measures never decline but consistently increase whenever more resources are used. At the largest scale of 3072 cores we observed a median speedup of 41 and a total speedup of 159. To put these measures in perspective, prior literature on massively parallel general-purpose SAT solving reported median speedups of up to 13.8 (at 1024 cores) and total speedups of up to 109 (at 2048 cores)—both with HordeSat [BSS15].[10] Our work achieves a median speedup of 19 with only 384 cores and a total speedup of 138.7 with only 1536 cores. The geometric speedup of MallobSat at 3072 cores is 43.7 over all commonly solved instances and is slightly larger for unsatisfiable instances compared to satisfiable instances (45.3 vs. 41.9).

[10]Note that we explicitly disregard the "arithmetic mean speedups" reported by Balyo et al. [BSS15] since these measures are statistically not meaningful (see Section 2.4.2.b for a discussion).

		overall				SAT				UNSAT			
m	cores	#	med.	geom.	total	#	med.	geom.	total	#	med.	geom.	total
1	24	254	6.51	7.20	12.13	126	5.12	6.30	8.91	128	7.48	8.21	14.92
1	48	275	8.32	9.42	17.46	138	6.73	7.92	11.18	137	9.76	11.21	23.52
2	96	302	10.88	12.62	21.54	150	9.36	10.87	15.70	152	12.77	14.63	26.29
4	192	309	15.52	17.71	37.09	153	13.17	16.13	30.16	156	17.87	19.41	42.52
8	384	316	19.00	22.63	63.20	154	14.96	18.80	35.06	162	25.33	27.00	86.15
16	768	322	25.14	31.75	105.01	154	19.86	28.27	51.05	168	35.86	35.32	137.13
32	1536	323	32.92	37.40	138.70	154	30.76	33.30	63.58	169	38.02	41.57	182.29
64	3072	324	40.99	43.67	159.00	154	42.09	41.93	78.29	170	38.90	45.30	200.54

Table 4.4: Speedups of MallobSat over KissatMABHyWalk on 24 to 3072 cores. Each section shows the number of commonly solved instances and the median, geometric mean, and total speedup for these instances.

At 384 cores, where the parallel and sequential approach received equal resources, MallobSat achieves an efficiency of 63.20/384 ≈ 16.5% in terms of total speedup.

Fig. 4.10 shows per-instance speedups of MallobSat at the smallest and largest tested scale. Speedups clearly correlate with sequential running times, which confirms the increasing merit of parallel solving as instances become more difficult. At 3072 cores, all 14 instances which resulted in a slowdown (speedup < 1) took the sequential solver less than 1.4 seconds. On the more difficult half of instances, i.e., those which took the sequential solver more than 349 s to solve, MallobSat at 3072 cores achieves a geometric mean speedup of 104.5 (median 81.9). On the 40 instances which took the sequential solver more than an hour to solve, the geometric mean speedup reaches 418.7 (median 370.5) and thus an efficiency of 418.7/3072 = 13.6%. Increasing the sequential and parallel running times may further extend the observed speedups.

While our 24-core run achieved superlinear speedups on 45 instances, the 3072-core run achieved only four such speedups. Our solver executes many different solver implementations and configurations which excel on different instances, whereas the sequential baseline only runs a single configuration. We believe this discrepancy to be the main reason for the observed superlinear speedups. At 3072 cores, this effect is watered down by using far more cores than there are distinct configurations. Note however that the 3072-core run solved 31% more instances than the 24-core run, which clearly indicates *weak scaling* (Section 2.4.2.d). Fig. 9.1 (Appendix B) provides a full graphical overview of the weak scaling of MallobSat.

Using the SAT benchmark database GBD [IS19], we traced the observed speedups to specific benchmark families. Among the instances at the lower edge of the cone of speedups in Fig. 4.10, where MallobSat at 3072 cores scales the worst, are unsatisfiable string safety verification tasks [OW21] (speedups of 13–25 at sequential running times of 800–1800 s) and satisfiable Hamiltonian Cycle encodings [Heu21a] (speedups of 17–36 for seven instances with sequential times of 700–2300 s). Conversely, Tab. 4.5 shows some of the *best* consistent speedups MallobSat achieves.

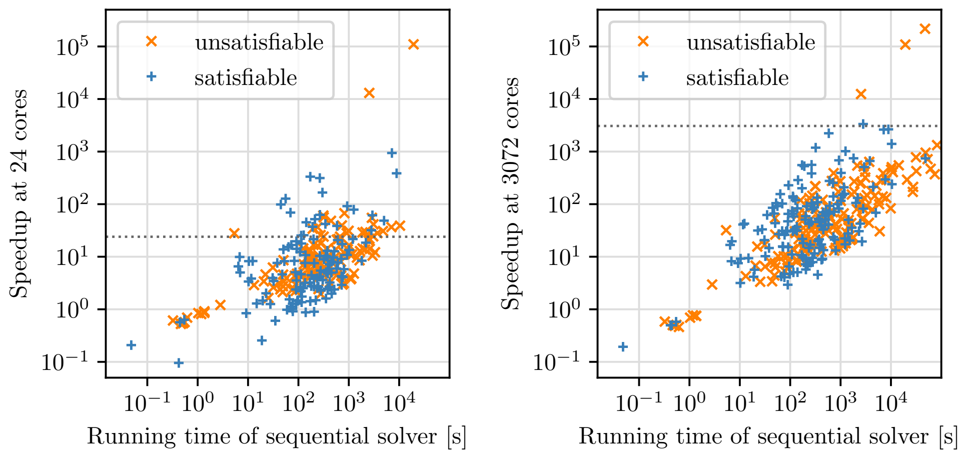


Figure 4.10: Per-instance log-scale speedups of MallobSat relative to the sequential solver's running time, on 24 cores (left) and on 3072 cores (right). The horizontal dotted lines separate sublinear from superlinear speedups.

		Speedup on x cores							
Instance	Seq. (s)	24	48	92	192	384	768	1536	3072
ctl_4201_555_unsat.cnf	1873.9	14.4	20.1	32.8	49.0	75.3	105.7	138.7	171.2
edit_distance031_283.cnf	2465.9	20.3	36.3	59.8	85.9	122.7	184.2	230.8	247.1
edit_distance031_284.cnf	2561.6	22.0	34.3	59.4	87.7	130.4	185.7	230.3	256.8
mp1-bsat201-707.cnf	3087.9	41.4	67.7	125.3	165.5	234.1	260.7	287.5	321.3
mp1-bsat210-739.cnf	7011.8	36.6	60.3	109.1	150.8	253.0	294.4	324.3	387.5
mp1-klieber2017s-2000-022-eq.cnf	908.0	28.1	49.0	65.8	81.6	92.7	110.6	149.7	172.7
randomG-B-Mix-n16-d05.cnf	3053.2	24.4	41.1	63.5	106.5	163.4	238.5	332.6	413.3
rphp4_065_shuffled.cnf	671.5	24.1	42.3	66.0	94.0	129.0	179.4	231.5	276.8
rphp4_080_shuffled.cnf	2145.0	29.7	53.1	85.3	161.1	229.6	295.1	416.3	542.1
rphp4_090_shuffled.cnf	3709.9	28.2	51.3	91.7	138.9	261.7	379.4	503.2	638.1
satch2ways14u.cnf	1868.9	29.0	45.9	60.8	87.7	120.7	183.0	218.1	270.9
satch2ways15.cnf	10383.0	39.0	59.6	78.5	106.8	180.0	265.9	369.4	506.1
sp4-33-one-stri-tree-noid.cnf	727.5	39.4	55.6	95.6	133.4	198.0	266.6	320.0	390.0

Table 4.5: Instances where the speedup of MallobSat increases consistently (by $\geq 5\%$ per step) and exceeds 5% efficiency (i.e., speedup ≥ 153.6) at 3072 cores.

			overall				SAT				UNSAT			
Sys.	m	cores	#	med.	geom.	total	#	med.	geom.	total	#	med.	geom.	total
H	1	24	184	3.80	3.26	7.95	82	3.03	2.56	5.62	102	4.74	3.95	10.17
H	1	48	214	3.67	3.87	8.02	98	2.96	2.94	5.01	116	5.27	4.89	11.51
H	2	96	241	6.00	6.00	19.35	105	5.35	5.10	15.29	136	7.34	6.80	22.11
H	4	192	267	8.54	8.13	26.72	117	7.54	6.86	17.14	150	9.77	9.27	32.70
H	8	384	284	10.93	11.23	34.35	127	9.61	10.21	17.75	157	11.52	12.14	43.69
H	16	768	293	13.72	13.83	43.31	133	12.16	12.46	26.50	160	15.06	15.08	54.95
H	32	1536	294	17.52	15.92	49.73	132	16.26	14.63	31.65	162	18.11	17.07	60.97
M'L	32	1536	311	20.37	21.14	59.51	147	17.46	17.81	31.71	164	25.33	24.63	83.24

Table 4.6: Speedups of HordeSat (H) and of MallobSat with a Lingeling-only portfolio (M'L), with the same metrics as in Tab. 4.4.

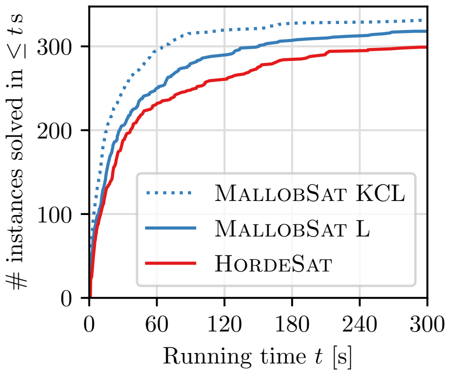
All of the displayed instances are unsatisfiable since speedups on satisfiable instances are more erratic. The benchmark families featured more than once encode cluster graph editing distance (`edit_distance*`) [Men21], balanced random SAT problems (`mp1-bsat*`) [Spe17], relativized Pidgeon Hole Principle (PHP) problems (`rphp4*`) [EN16], and automated test configuration (`satch*`) [Bie21].

We ran HordeSat on up to 1536 cores and computed speedups in the same manner as for MallobSat. As Tab. 4.6 shows, speedups are slightly larger than reported for original HordeSat—now achieving a median speedup of 17.5 on 1536 cores—although we computed them based on the current sequential state of the art and disregard sequential timeouts. The most likely causes for this improvement are (a) our updates to HordeSat's sources and solver backends, (b) differing benchmark sets, and (c) deviating time limits. Tab. 4.6 also includes a run of our system where we replaced our final, mixed portfolio with a Lingeling-only portfolio as HordeSat's. With the same solver backend, MallobSat improves on HordeSat's geometric mean speedup by 32.8% while also solving 17 more instances. With our mixed portfolio, MallobSat more than doubles the speedups of HordeSat at all scales (see Tab. 4.4).

4.6.4 Performance Insights

To conclude the evaluation of MallobSat, we provide some further insights based on the data of our scaling experiments.

Fig. 4.11 provides a direct comparison of HordeSat vs. MallobSat on the ISC 2021 benchmarks at 1536 cores. We again show the performance of MallobSat both with Lingeling only as well as with our final mixed portfolio consisting of equal parts of Kissat, CaDiCaL, and Lingeling. Clearly, MallobSat only reaches its full potential with our new portfolio. Since we designed, tested, and tuned our techniques based on this new portfolio, this does not imply that HordeSat would improve by the same margin if it employed this portfolio as well. The CBS (Count Based Speedup, Section 2.4.2.b) of MallobSat over HordeSat is 1.98 if both systems use Lingeling only and 4.54 if MallobSat uses our mixed portfolio.



	overall		SAT		UNSAT	
	#	PAR	#	PAR	#	PAR
Mal' KCL	331	122.8	158	142.4	173	103.2
Mal' L	318	153.8	149	183.9	169	125.1
Horde	299	184.8	133	226.6	166	143.1

Figure 4.11: Comparison of our updated HordeSat vs. MallobSat on ISC 2021 benchmarks with 1536 cores.

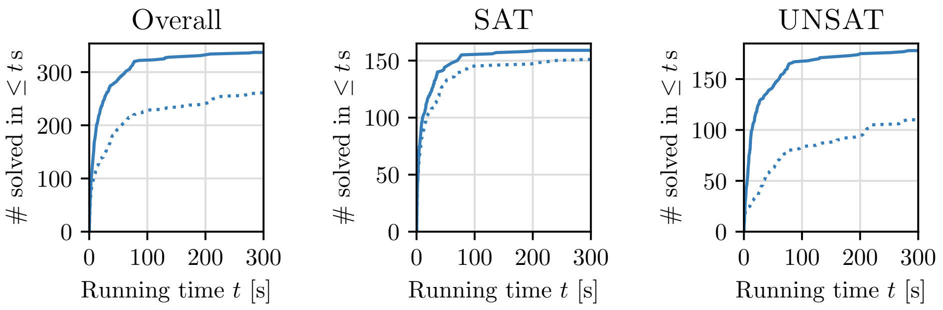


Figure 4.12: Performance of MallobSat at 3072 cores with (straight line) and without (dotted line) clause sharing.

Next, we analyze the impact of MallobSat's clause sharing at the largest tested scale of 3072 cores. We re-ran MallobSat at this scale *without* clause sharing, rendering it a pure portfolio solver without any exchange of information. Visual results are given in Fig. 4.12. Separating the benchmarks into satisfiable and unsatisfiable instances, we can confirm earlier findings that clause sharing is crucial for unsatisfiable instances, much moreso than for satisfiable instances [BSS15]. Balyo et al. did not find evidence that clause sharing benefits satisfiable instances at all—even finding clause sharing to deteriorate performance for random instances. By contrast, we do observe improved performance on satisfiable instances. This supports the intuition that shared clauses are central for eventually deriving the empty clause of unsatisfiable problems but can also be useful as a means of pruning the search for a satisfying assignment.

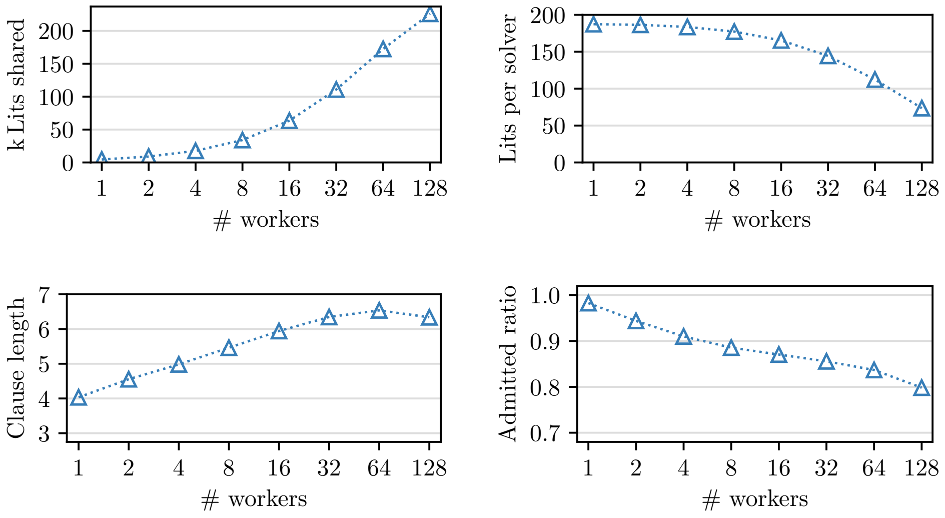


Figure 4.13: Clause sharing behavior for 1–128 workers (1–3072 cores) in terms of the median successfully shared literals (top left), the same measure divided by the number of solvers (top right), the mean length of successfully shared clauses (bottom left), and the mean ratio of exchanged clauses admitted by the distributed filter (bottom right). Note the y axis offsets in the bottom figures.

The CBS of the clause sharing run over the sharing-less run is 4.08 for satisfiable and 15.59 for unsatisfiable instances. Without sharing, the PAR-2 score is 166.1 on satisfiable instances and 309.7 on unsatisfiable instances. We can compare these scores to the PAR-2 scores MallobSat achieves at different scales with sharing enabled (Fig. 4.9): On satisfiable instances, the sharing-less run on satisfiable instances performs worse than the clause sharing run at 384 cores (PAR-2 159.1) and more. On unsatisfiable instances, the sharing-less run performs worse than *all* clause sharing runs—even the run at only 24 cores (PAR-2 246.5). With these results, we provide clear evidence that clause sharing is crucial for the performance of our system and causes it to substantially surpass the performance of a pure portfolio.

Last but not least, we consider some clause sharing statistics. Fig. 4.13 (top) illustrates the impact of the sharing buffer's sublinear scaling: The number of successfully shared literals per solver and per sharing decreases as the number of solvers increases. In these measures the clauses blocked by distributed filtering (see Fig. 4.13 bottom right) have already been compensated for by correspondingly larger sharing buffers.

The average length of successfully shared clauses (Fig. 4.13 bottom left) mostly increases with the number of involved solvers. Whenever the number of solvers is doubled, significantly more literals are allowed to be shared. Since the set of distinct short clauses which the solvers produce is limited, this generally leads to an increase in the mean length of successfully shared clauses. This is no longer the case at the largest scale of 128 workers (3072 solvers), where the total sharing volume increases only by about 31% compared to 64 workers. We rather see a slight reduction in the mean shared clause length (7.5 to 7.3) at this scale—showing that more selective sharing relative to the global volume of produced clauses can indeed improve shared clause quality. We conjecture that the mean clause quality may continue to improve when further increasing the number of solvers.

The ratio of clauses admitted by the distributed filter (Fig. 4.13 bottom right) indicates the increasing relevance of filtering the more workers are involved. Adding workers increases the probability that a clause is exported redundantly at several workers. At the largest scale, another effect can be observed: As the sharing budget per solver decreases, stricter quality criteria are enforced which results in a smaller interval of admissible clause lengths and, therefore, an increased ratio of duplicates.

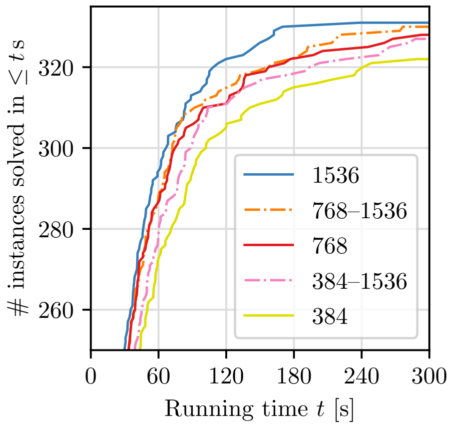
4.6.5 Malleable SAT Solving

In the following, we evaluate how the performance of MallobSat is impacted by malleability, i.e., by repeatedly modifying the number of associated workers. We use a 32-machine setup (1536 cores) of our scheduler Mallob with two streams of jobs: a *benchmark stream* and a *disturbance stream*. The benchmark stream sequentially introduces 400 jobs corresponding to the ISC 2021 benchmarks at a time limit of 300 s per instance. The disturbance stream is configured differently: Every 20 s, a formula with a wallclock timeout of 10 s is introduced. The formula is chosen in such a way that it is realistically not solvable within these constraints.[11] We assign a certain priority p to each such disturbance job while we keep default priority 1 for each benchmark job. As such, each benchmark job is forced to yield $100 \cdot \frac{p}{p+1}\%$ of its resources 50% of the time (on average). We run the experiment once with $p = 1$ and once with $p = 3$, which corresponds to periodically removing one and two layers from the job tree respectively. We compare the performance of our disturbed benchmark stream with the performance observed in our scaling results.

Fig. 4.14 compares the performance of each benchmark stream to our earlier scaling runs without any disturbances. The disturbed stream oscillating between 1536 and 384 cores is not able to reach or surpass the performance of an undisturbed 768-core stream although its mean resources are at 960 cores. Still, it clearly surpasses the performance of a 384-core run which confirms that the fluctuating resources do have some merit. On unsatisfiable instances, disturbed performance is quite close to 768-core performance. On satisfiable instances, the disturbed run happens to solve one instance less than the 384-core run while very slightly improving on its running times.[12] The disturbed stream oscillating between 1536 and 768 cores behaves similarly—showing good performance on unsatisfiable instances just between the 768-core and the 1536-core run while dropping to the 768-core run's performance on satisfiable instances.

[11]Since the formula encodes the existence of a period-19 pattern in *Conway's Game of Life* [Gar70] in a 20×20 grid, we would welcome to be proven wrong.

[12]At 200 instances and a 300 s timeout, an unsolved instances incurs $2 \cdot 300/200 = 3$ PAR-2 points.



	overall		SAT		UNSAT	
cores	#	PAR	#	PAR	#	PAR
384	322	141.7	157	150.0	165	133.3
384–1536	327	133.9	156	150.5	171	117.4
768	328	130.1	157	146.5	171	113.6
768–1536	330	127.4	157	146.4	173	108.4
1536	331	122.8	158	142.4	173	103.2

Figure 4.14: SAT performance with fluctuating resources—note the y axis offset.

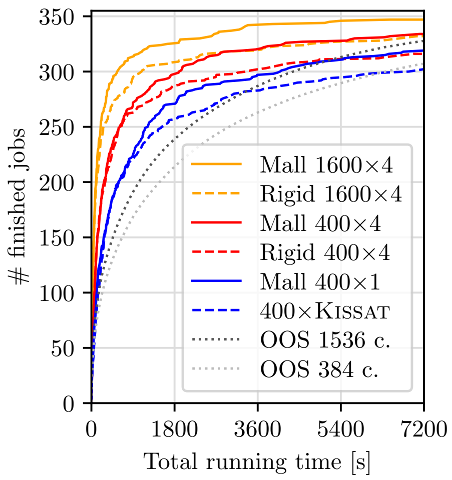
We conclude that under continuous and significant fluctuations of a job's resources, our simple malleability model allows to maintain the performance achieved by the (undisturbed) base resources and to profit from the fluctuating resources to some degree. However, we were not able to demonstrate the latter effect for satisfiable instances. A possible reason is that unsatisfiable instances profit substantially from shared clauses, which a solver contributes whenever it is not suspended. In this sense, the progress made within a particular solver may still benefit the overall procedure even at times where it is suspended. On satisfiable instances, which do not benefit from shared clauses to the same extent, a solver which is only active half of the time is unlikely to contribute in a truly significant manner, i.e., by being the first to find a solution. As such, further work is required to achieve similar positive effects when solving satisfiable instances. We believe that this may potentially be achieved by more aggressive and dynamic (re-)diversification of solvers, especially of those which are resumed after they have been suspended for some time.

4.6.6 Massively Parallel Processing of SAT Jobs

In a last set of experiments, we evaluate MALLOB as a malleable scheduler and processor of many concurrent SAT tasks with MALLOBSAT as an application engine. We allocate a certain set of computational resources for two hours of wallclock time and attempt to complete as many tasks as possible among 400 SAT tasks, which correspond to the benchmark set of ISC 2021 and which are all available at the beginning of the time frame. We use our malleable scheduling system as follows: We randomly partition the 400 formulas into 16 equally sized sets. Among all worker processes, we configure 16 processes to additionally parse and introduce one of the job sets at system start.

	m	Cores	Processes	Cores/job
1600×4	136	6400	1600	16
400×4	36	1600	400	4
400×1	9	400	400	1

Table 4.7: Scales used in experiments on massively parallel job processing.



	Cores	#	PAR
Mall 1600×4	6400	347	2220.8
Rigid 1600×4	6400	332	2825.9
Mall 400×4	1600	334	2909.9
Rigid 400×4	1600	316	3508.3
Mall 400×1	400	319	3635.3
400×Kissat	400	303	5204.5
OOS	1536	328	3695.0
OOS	384	308	4467.4

Figure 4.15: Performance of different (real-world or virtual) scheduling strategies for the ISC 2021 benchmark set on 384–6400 cores.

We compare our malleable scheduling with some other, simpler scheduling strategies. First, we use our scheduling system while capping the maximum demand of each task to 1/400 of the overall computational resources. This approach thus runs 400 parallel solvers, all with fixed and equal amounts of resources, at the same time. We thus refer to this approach as *rigid scheduling*. Secondly, we consider the isolated processing times of our massively parallel solver at a comparable scale and sort the 400 instances by processing time in ascending order. This yields what we refer to as an *Optimal Offline Schedule* (OOS)—the best possible way to *sequentially* schedule runs of our parallel solver if we had perfect knowledge on the processing time of each job.

As detailed in Tab. 4.7, we run our scheduling at three scales which initially run each job on 16, four, and a single core respectively. At the lowest scale of 400 cores, instead of running Mallob with rigid scheduling we provide the results of KissatMABHyWalk, pretending that all 400 runs are done in parallel on 400 cores.

Fig. 4.15 shows results in terms of finished jobs over time. The OOS are shown at 1536 cores and at 384 cores—slightly different scales compared to the other schedulers.

Still, the OOS can provide a good intuition on how such schedules would perform. Given sufficient time, running each job at the maximum available scale will presumably solve the most instances. However, the OOS are clearly not resource efficient: Over long time periods, they stay behind the response times achieved by running 400×Kissat. The rigid strategies, including 400×Kissat, achieve better resource efficiency since each job is executed at a comparably low degree of parallelism and in turn all jobs can be processed in parallel. We observed significant speedups for increasing the resources of each job from one to four cores and from four to 16 cores respectively.

Enabling our scheduling's full malleability further improves performance by significant margins. We especially consider the pronounced performance improvements of our 400-core malleable scheduler over 400×Kissat to be a strong result since it shows how our system is able to seamlessly shift from a multi-instance sequential solver to a (small-scale) parallel solver, both with state-of-the-art performance. This is achieved while demonstrably incurring low overhead, even though we run an MPI process for every single solver thread, and without requiring any additional resources except for making use of both hardware threads on each core. Furthermore, the 400-core malleable scheduler reaches the performance of the 1600-core rigid scheduler after around 1.5 hours and then even solves a few more instances. At this point in time, less than 100 jobs are remaining for the malleable approach and therefore each job runs on at least four cores—the resources per job of the rigid approach. An equivalent effect can be seen for the 1600-core malleable scheduler which begins to outperform the 6400-core rigid scheduler after around 40 minutes since each job then receives at least 16 cores. Evidently, MallobSat is able to make good use of these added resources, which we consider a confirmation that our malleable solving approach is useful and effective in such scenarios. At the largest scale, our malleable approach is able to solve 347 instances—by far more than any system in the ISC 2021 or in our prior evaluations—while only spending 12 800 ch (core hours).[13]

4.7 MallobSat in the International SAT Competition

At the time of writing, our system participated in four iterations of the ISC (2020–2023). For a general introduction to the ISC, see Section 2.4.3. We now discuss the submitted configurations of MallobSat and their performance in these four years. In short, our submissions competed with a total of thirteen other submissions in four iterations of the cloud track, achieving the best performance each time, and competed with a total of 30 other submissions in three iterations of the (shared-memory) parallel track, achieving a top-three rank each time. Overall, our submissions have been awarded with 18 medals (11 gold, 5 silver, 2 bronze). These results confirm that MallobSat (i) performs well as a general-purpose SAT solver on previously unseen inputs, (ii) can be efficiently deployed in a cloud environment, and (iii) compares favorably to other parallel SAT solvers, from shared-memory parallelism to distributed computing.

[13]In our latest scaling experiments MallobSat solved 331 instances within 12 129 ch (at 1536 cores) or 335 instances within 22 919 ch (at 3072 cores).

4.7.1 Setup

Since 2020, the parallel and cloud tracks of the ISC are evaluated in Amazon Web Services (AWS) [Bal+20a]. In this cloud environment (see Section 2.1.2.b), solvers are deployed in Docker containers. In the parallel track, a single machine of instance type `m4.16xlarge` is used. This instance type features *Intel Broadwell Xeon E5-2686 v4* processors and is advertised to feature 64 "virtual CPUs" and 256 GB of RAM.[14] We assume that the machine is in fact two-socket hardware with 18 cores (36 hardware threads) each, totalling 72 hardware threads 64 of which are accessible for user tasks. In the cloud track, 100 machines of instance type `m4.4xlarge` are used in parallel. This type is advertised to feature 16 virtual CPUs and 64 GB of RAM.[15]

The setup suggested by the organizers to set up communication via MPI is the following: Each worker node reports its IP to a single leader node (via `SSH` and the leader node's file system). The leader node then executes the solver system via MPI on the assembled list of IPs. Latencies and overhead of this TCP/IP based communication setup are significantly higher than with the rapid interconnects on modern HPC clusters such as Infiniband or OmniPath [Gra+06]. This may lead to a relative disadvantage for systems with high communication bandwidth (such as TopoSAT 2 in 2020 with naïve all-to-all sharing). Since our system performs careful communication with only a few collective operations per second and performs this communication strictly asynchronously, it is able to handle these increased costs well.

4.7.2 By-year Discussion

We now discuss our submissions and the results achieved in each year. Table 4.8 provides an accompanying overview.

4.7.2.a 2020

The first time MallobSat participated in the ISC was in 2020 [Sch20]. Our work back then was focused on preparing a malleable SAT solving engine based on HordeSat with reasonable efficiency, with the aim of deploying it within Mallob. Our configuration performed very cautious clause sharing (up to clause length five) and employed an updated Lingeling portfolio including some YalSAT instances. We did not consider submitting MallobSat to the parallel track that year.

Our SAT solving engine proved to be highly competitive even though its main feature, malleability, remained unused. MallobSat outperformed the second best performing cloud solver TopoSAT 2 by a decent margin [Bal+20b] and was able to solve two satisfiable ("`Steiner`") instances exclusively across all main, parallel, and cloud track solvers. However, since the organizers did not provide performance data of HordeSat, which had been set up as a baseline example solver, the degree to which MallobSat advanced the state of the art remained unclear.

[14]`https://aws.amazon.com/ec2/dedicated-hosts/pricing`
[15]`https://aws.amazon.com/de/ec2/instance-types/`

Year, ref.	Track	Submission name	#	PAR	Rank
2020 [Bal+20b]	Cloud	mallob-mono	299	583	1.
		TopoSAT 2	278	706	2.
2021 [Bal+21b]	Cloud	Mallob-HC*	337	373	—
		Mallob	316	480	1.
		MergeHordeSat	260	858	2.
	Parallel	P-MCOMSPS	320	2386	1.
		Mallob-parallel	318	2411	2.
2022 [Bal+22b]	Cloud	Mallob-KiCaLiGlu	341	345	1.
		Paracooba	221	795	2.
	Parallel	ParkissatRS	326	2105	1.
		Mallob-Ki	292	2988	3.
	Anniv. Cloud	Mallob-KiCaLiGlu	4687	279	1.
		Paracooba	3619	725	2.
	Anniv. Parallel	Mallob-Ki	4400	1992	1.
		MergeSAT-AWS	4076	2690	2.
2023 [Bal+23a]	Cloud	Mallob1600	328	426	1.
		PRS-distributed	305	531	2.
	Parallel	PRS-parallel	320	2272	1.
		Mallob64	301	2746	2.
		Mallob32	293	2941	—

Table 4.8: Overview of MallobSat's performance in ISC 2020–2023, including all of our submissions as well as the best ranking competitor for each track. The 2022 Anniversary tracks featured 5355 benchmark instances; all other tracks featured 400 instances. *Mallob-HC was not ranked officially since the mixed portfolio it employed was disallowed in the competition.

4.7.2.b 2021

Following the success of MallobSat in 2020, we submitted a configuration both to the parallel and the cloud track [Sch21e]. Among that year's improvements, MallobSat now shared clauses up to length 30 and also targeted a significantly higher sharing volume, with an increased discount factor α from 0.75 in 2020 up to 0.9 (cloud track) and 1.0 (parallel track) respectively. We also introduced a rudimentary and probabilistic clearing of our (Bloom) clause filters at a half life of 300 s. Motivated by huge formulas in the prior year's benchmark set, we added basic memory awareness to MallobSat (spawning fewer solver threads based on formula size).

MallobSat in 2021 dominated the cloud track with a substantial margin to all other competitors, solving 56 instances more than the submission placed second. MallobSat also proved to be a highly competitive system in the parallel track, scoring the second place overall and being the only solver with a top-3 rank on both satisfiable and unsatisfiable instances. While mixed portfolios featuring multiple different solver backends were still disallowed, the organizers kindly ran such a submission of MallobSat *hors concours* as in the cloud track. This version, Mallob-HC, significantly outperformed our official submission and was able to solve 42 out of 400 instances *exclusively* across the three tracks (listed online, see Appendix A)—demonstrating the power of our mixed portfolio with clause sharing. Following this iteration of the competition, the unprecedented performance of our system was acknowledged in an Amazon Science blog post on automated reasoning, commenting that it "*is now, by a **wide** margin, the most powerful SAT solver on the planet*" [Coo21].

4.7.2.c 2022

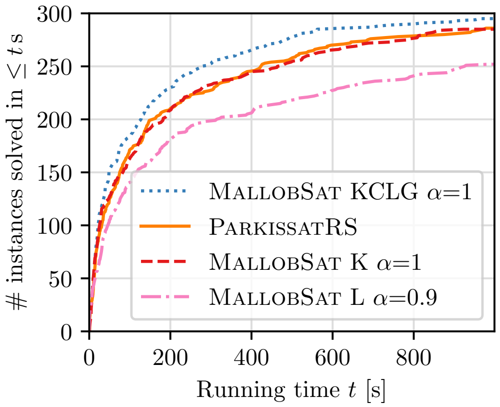
2022 marked the first iteration of the ISC where authors were allowed to submit *mixed portfolios* consisting of several different sequential solvers to the cloud track [Bal+22b]. We submitted MallobSat with a portfolio of Kissat, CaDiCaL, Lingeling, and Glucose. For the first time, MallobSat used subprocessing and an early version of distributed filtering. In the parallel track (where mixed portfolios remain forbidden) we intended to submit a Kissat-only portfolio [Sch22] but in fact submitted a Lingeling-only portfolio due to a misconfiguration. The same mistake also led to a deviating buffer limit discount $\alpha = 0.9$ instead of the intended value $\alpha = 1$ in the parallel track.

The cloud track, unfortunately, only saw two qualified participants (with a third one disqualified), the other submission being Paracooba [Hei22]. In the parallel track, our unintentional submission of a Lingeling-based portfolio proved to be a blessing in disguise on unsatisfiable instances: MallobSat scored the top rank in all unsatisfiability sub-tracks it was eligible for. Overall our parallel solver scored the 3rd place. 16 instances among the 400 benchmark instances were solved exclusively by some MallobSat configuration(s) (listed online, see Appendix A).

We ran a follow-up evaluation on our own hardware (64 hardware threads) to assess the impact of our misconfiguration in the parallel track. Fig. 4.16 shows results, indicating that our intended configuration performs substantially better than our submitted configuration and is on par with the winning parallel solver ParkissatRS [ZCC22] at a timeout of 1000 s. A mixed portfolio further improves performance by a considerable margin but is disallowed in the parallel track.

Since 2022 marked the 25th iteration of the ISC, it featured three further tracks on the *Anniversary benchmark set*—the largest set of SAT instances yet, with a total of 5355 formulas. MallobSat won both the cloud and the parallel Anniversary track. Since this benchmark set features many instances on which solvers have been tuned for years, these results are not necessarily as meaningful as those in the usual tracks.[16] We identified 266 instances which *only* MallobSat (parallel or cloud) was able to solve (see Appendix A). We also provide a comparison of MallobSat with Paracooba based on the 2022 Anniversary cloud track in Fig. 9.2 (Appendix B).

[16]Biere et al. suggest to use a benchmark set for up to three years after its publication [Bie+20b].



	#	PAR-2
MallobSat KCLG α=1	295	623.4
ParkissatRS	286	687.0
MallobSat K α=1	285	692.5
MallobSat L α=0.9	252	858.6

Figure 4.16: Performance at 64 hardware threads of ParkissatRS (2022) and of MallobSat with the submitted 2022 configuration ("L $\alpha = 0.9$"), the intended configuration ("K $\alpha = 1$"), and a mix of all solvers supported by MallobSat ("KCLG $\alpha = 1$", a configuration which would be disallowed in the parallel track).

4.7.2.d 2023

In the most recent iteration of the ISC at the time of writing, we submitted our system in a configuration similar to our final tuned version in this chapter with some notable differences [Sch23]. Our submission shares clauses *three* times a second and still uses the older sublinear buffer limit function. It also contains a bug in the clause filtering which leads to suboptimal by-solver filtering and only features a preliminary compensation technique for unused sharing volume. We submitted two variants to the parallel track: One which employs 32 solver threads and one which employs 64 solver threads, in both cases within a single process. Our intention was to assess whether making full use of all advertised virtual CPUs is actually beneficial.

The 2023 iteration of the ISC marks the first year where competitors in the sequential main track were allowed to specify the proof validation tool which should be used for their submission [Bal+23b]. Three different proof checking systems were submitted, two of which allow for more powerful reasoning techniques than the classical resolution calculus most conventional CDCL solvers still use. Even though parallel solvers are not required to emit proofs, this change indirectly proved to be relevant for all tracks: Since the ISC enforces sequential solver submissions to "*Bring Your Own Benchmarks*", the submissions using other proof systems were accompanied by benchmark instances which are known to be difficult to solve with pure resolution [Bog+23; CRB23].

The 2023 cloud track featured a small but strong set of competitors, with our submission of MallobSat winning by the smallest margin so far (yet still decidedly). On satisfiable instances the new system PRS-distributed [ZCC23] (where clause sharing is limited to a ring communication structure) achieved performance close to MallobSat whereas on unsatisfiable instances the system Malloblin [Cho23] performed similarly and in fact solved slightly more instances. The latter system is a fork of MallobSat that replaced Lingeling's local search solver YalSAT with an alternative named yallin. Since this local search solver is unable to find unsatisfiability, we attribute Malloblin's success on unsatisfiable instances to the choice of Lingeling vs. our mixed portfolio. In the parallel track, Malloblin achieved the second place on unsatisfiable instances, similar to our own Lingeling-based submission one year earlier. On satisfiable instances and in the overall rating, our own submission of MallobSat with 64 solvers scored the second place after PRS-parallel [ZCC23] (the successor to ParkissatRS). Our 32-solver variant performed considerably worse than the 64-solver variant, indicating that using more than half of the virtual CPUs is beneficial on the used AWS instance type.

Despite the stronger competition, four instances were only solved by MallobSat submissions across the three principal tracks, among them—coming full circle with our introduction (Section 1.1)—an arithmetic circuit equivalence problem[17] [Kau+19].

4.8 Conclusion

In order to improve the scalability and resource-efficiency of general-purpose SAT solving in large distributed environments, we presented a novel malleable SAT solving engine MallobSat within our job scheduling framework Mallob. Our engine is a consequent overhaul of HordeSat and incorporates a compact clause sharing approach, state-of-the-art solver backends, and various practical improvements. We showed that our standalone SAT solver significantly outperforms the prior state of the art in massively parallel solving and consistently leads to improved speedups. In particular, we observed scaling up to 3072 cores especially for unsatisfiable instances. We also discussed the results of four iterations of the International SAT Competition, where our system performed very favorably compared to other (massively) parallel solvers. Last but not least, we showed that Mallob's combination of parallel job processing and flexible parallel SAT solving is able to reduce scheduling and response times and to maximize resource efficiency in a distributed environment.

While we have advanced the state of the art in distributed SAT solving, our evaluations also indicate promising directions for future research. An important task is to further improve our system's running times and scaling behavior on satisfiable instances, especially under fluctuating resources. For this purpose, we may pursue more dynamic and aggressive diversification of solvers. Other relevant directions to extend our work may include the integration of GPUs, the parallelization of pre- and inprocessing techniques, and a data-driven selection of diverse solver portfolios via machine learning techniques (*cf.* [BIB22]).

[17] `eqspctbk14spwtcl14.cnf`

Chapter 5

5 Unsatisfiability Proofs for Distributed SAT Solving

Our previous efforts on distributed clause-sharing SAT solving have led to impressive speedups over sequential SAT solvers by sharing derived information among many solvers working on the same problem. Unlike sequential solvers, however, distributed solvers have not been able to produce proofs of unsatisfiability in a scalable manner, which limits their use in critical applications. In this chapter, we present a method to produce unsatisfiability proofs for our distributed SAT solver by combining the partial proofs produced by each sequential solver into a single, linear proof. Our approach is more scalable and general than previous explorations for parallel clause-sharing solvers. We propose a simple sequential algorithm as well as a fully distributed algorithm for proof composition. Our empirical evaluation shows for a setup with 100 nodes × 16 hardware threads that our distributed approach allows proof composition and checking with reasonable overhead. We analyze the overhead and discuss how and where future efforts may further improve performance.

Author's Notes. *This chapter is based on* "Unsatisfiability Proofs for Distributed Solvers" *[Mic+23], a joint publication by Dawn Michaelson, Benjamin Kiesl-Reiter, Marijn Heule, Michael W. Whalen, and myself. Dawn Michaelson and I are equal main authors of that publication. While I revised large parts of the publication, some parts of this chapter are copied verbatim or with minor changes from the publication. The basic proof production approach, the implementation of necessary changes to* CaDiCaL, *and the experiments conducted in the AWS cloud are due to my co-authors. I designed, implemented, and analyzed the distributed proof production approach, implemented the vast majority of necessary changes to* MallobSat, *contributed to the experimental setup, authored most of the presentation and discussion of experiments, and contributed significantly to the remaining sections. Compared to the original publication, this chapter only briefly outlines basic proof production. In turn, this chapter features some added content, in particular a proof of correctness for our distributed proof production and a revised evaluation section.*

5.1 Introduction

Distributed clause-sharing SAT solvers such as MallobSat are demonstrably some of the most powerful tools available for solving hard SAT problems [Fro+21; Coo21].

However, there is an important caveat: unlike sequential solvers, current distributed clause-sharing solvers cannot produce proofs of unsatisfiability (Section 2.2.4). A direct consequence is that these distributed solvers cannot be used for proving theorems [HKM16; Heu18; SH23]. Even in cases where proofs are not strictly required, it is important to be able to trust the output of an algorithm.[1] For instance, in bounded model checking [Cla+01], one of the most prominent applications of SAT solving, a formula's reported unsatisfiability is interpreted as a system behaving correctly up to the considered depth (see Section 2.2.5). Therefore, the safety and reliability of crucial systems may depend on a SAT solver answering correctly.

In this sense, we argue that distributed clause-sharing solvers are lacking behind sequential SAT solvers in terms of general trustworthiness. The former, while complex pieces of software, not only generate verifiable proofs but are also being rigorously tested (e.g., [BLB10; Bie21]) and feature limited external interfaces. Distributed clause-sharing solvers, on the other hand,

– are more costly (and thus more difficult) to test rigorously;
– make use of several different SAT solvers configured in many different ways [BSS15];
– run many execution threads concurrently; and
– make use of non-trivial interfaces for data transfer such as message passing [BSS15] and/or inter-process communication [SS21b].

All of these properties have some potential to introduce faults or correctness issues in certain corner cases, which makes it all the more critical to be able to independently verify the system's output.

While there has been foundational work in producing proofs for shared-memory clause-sharing SAT solvers [HMP14; BF22b], existing approaches are not general enough for large-scale distributed portfolio solvers (see Section 2.3.4). In this chapter, we address this issue and present the first scalable approach for generating proofs for such solvers. To construct proofs, we maintain *provenance* information about shared clauses in order to track how they are used in the global solving process, and we use the recently-developed LRAT proof format [Cru+17] to track dependencies among partial proofs produced by solver instances. By exploiting these dependencies, we are then able to reconstruct a single linear proof from all the partial proofs produced by the sequential solvers. We first outline a simple sequential algorithm for proof reconstruction before devising a parallel algorithm that we implemented in a fully distributed way. Both algorithms produce independently-verifiable proofs in the LRAT format. We demonstrate our approaches using an LRAT-producing version of the sequential SAT solver CaDiCaL [Bie+20a] to turn it into a clause-sharing solver, and then modify MallobSat (see Chapter 4) to orchestrate a portfolio of such CaDiCaL instances while tracking the IDs of all shared clauses.

We conduct an evaluation of our approaches from the perspective of efficiency, benchmarking the performance of our clause-sharing portfolio solver against the winners of the cloud track, parallel track, and sequential track from the ISC 2022. Our approach introduces overhead in terms of solving, proof reconstruction, and proof checking.

[1]There are certain exceptions, such as *Monte Carlo* algorithms where providing some incorrect results is an explicit part of the specification.

We examine this overhead in detail and show that our approach is still considerably faster than sequential approaches. We also demonstrate that our approach substantially outperforms prior work on proof production for clause-sharing portfolios [HMP14]. We argue that much of the overhead of our current setup can be compensated, among other measures, by improving support for LRAT in solver backends. We thus hope that our work incentivizes researchers to add LRAT support to other solvers.

This chapter is structured as follows. In Section 5.2, we present common proof formats which are relevant for the discussion of our work. For a discussion of related work we refer to Section 2, in particular Sections 2.3.2 and 2.3.3.a for clause-sharing portfolios and Section 2.3.4 for parallel certified SAT solving. In Section 5.3, we outline the general problem of producing proofs for distributed SAT solving and outline a first simple algorithm for proof combination. In Section 5.4, we describe an efficient distributed version of our algorithm. We discuss implementation details in Section 5.5. Finally, we present the results of our empirical evaluation in Section 5.6 and conclude with a summary and an outlook for future work in Section 5.7.

5.2 Proof Formats

For a general introduction to proofs of unsatisfiability, we refer to Section 2.2.4. In terms of specific proof formats, the current standard format for sequential SAT solving is called DRAT [Heu16]. An example DRAT proof is given in Figure 5.1 (center), formatted in the commonly used DIMACS CNF syntax (see Section 2.4.1). Each line of the proof is either an *addition* or a *deletion* statement. Additions are lines that represent clauses which were derived ("learned") by the solver. Each clause addition must preserve satisfiability. Furthermore, each added clause must adhere to a certain criterion which allows checkers to reliably confirm that the clause is indeed a logical consequence of the prior clause set. The specifics of this *RAT criterion* [Heu16] are not essential to our work. Deletions are lines that start with a '`d`', followed by a clause; they identify clauses that were deleted by the solver. Clause deletions can only make a formula "more satisfiable", meaning that they are not required for deriving unsatisfiability, but they drastically speed up proof checking [Heu16]. A valid DRAT proof of unsatisfiability ends with the derivation of the empty clause. As the empty clause is trivially unsatisfiable (and since each proof step preserves satisfiability) the unsatisfiability of the original formula can then be concluded.

The more recent LRAT proof format [Cru+17] augments each clause addition with *hints*, or *dependencies*, which identify the clauses that were required to derive the current clause. This makes proof checking more efficient, and in fact the usual pipeline for trusted proof checking is to first use an efficient but unverified tool (like `drat-trim` [Heu16]) to transform a DRAT proof into an LRAT proof, and then check the resulting LRAT proof with a formally verified proof checker [Cru+17; Heu+17; Lam20; THM21]. Figure 5.1 shows an LRAT proof corresponding to a DRAT proof. Each proof line starts with a clause ID. The numbering starts with ID `9` because the eight clauses of the original formula are assigned the IDs `1` to `8`.

DIMACS CNF
```
p cnf 4 8
1 -2 0
2 -4 0
1 2 4 0
-1 -3 0
1 -3 0
-1 3 0
1 3 -4 0
1 3 4 0
```

DRAT proof
```
-3 0
1 2 0
-1 0
d -3 0
2 3 -4 0
1 2 3 0
0
```

LRAT proof
```
9 -3 0 5 4 0
10 1 2 0 3 2 0
11 -1 0 6 9 0
11 d 9 0
12 2 3 -4 0 7 11 0
13 1 2 3 0 8 12 0
14 0 11 10 1 0
```

Figure 5.1: CNF formula and corresponding DRAT and LRAT proofs. Headers and separators are colored gray, deletions are colored red, clause IDs are colored blue, and hints are colored orange. Clause literals are always colored black.

Each clause addition first lists the literals of the clause and then the clause's dependencies in the form of clause IDs. Clause deletions just state the ID(s) of the clauses to delete, as in the later deletion of clause `9`. In our work, we exploit the hints of LRAT to determine dependencies among distributed solvers.

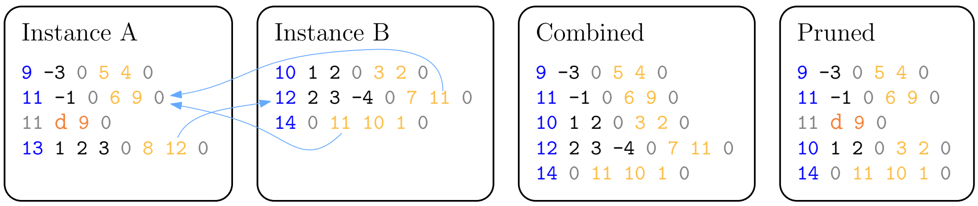
5.3 Basic Proof Production

Our goal is to produce checkable unsatisfiability proofs for problems solved by distributed clause-sharing SAT solvers. We propose to reuse the work done on proofs for sequential solvers by having each solver produce a partial proof containing the clauses it learned. These partial proofs are invalid in general because each sequential solver can rely on clauses shared by other solvers when learning new clauses. For example, when solver A derives a new clause, it might rely on clauses from solvers B and C, which in turn relied on clauses from solvers D and E, and so on. The justification of A's clause derivation is thus spread across multiple partial proofs. We need to combine the partial proofs into a single valid proof in which the clauses are in *dependency order*, meaning that each clause can be derived from previous clauses.

To produce efficiently checkable proofs in a scalable way, we address three challenges:

(i) Provide metadata to identify which solver produced each learned clause.
(ii) Efficiently sort learned clauses in dependency order across all solvers.
(iii) Reduce proof size by removing unnecessary clauses.

Switching from DRAT to the LRAT proof format provides the mechanism to unlock all three challenges. First, we specialize the clause-numbering scheme used by LRAT in order to distinguish the clauses produced by each solver. Second, we use the dependency information from LRAT to construct a complete proof from the partial proofs produced by each solver. Finally, we determine which clauses are unnecessary (or used only for parts of the proof) to trim the proof where possible.



Instance A

```
9 -3 0 5 4 0
11 -1 0 6 9 0
11 d 9 0
13 1 2 3 0 8 12 0
```

Instance B

```
10 1 2 0 3 2 0
12 2 3 -4 0 7 11 0
14 0 11 10 1 0
```

Combined

```
9 -3 0 5 4 0
11 -1 0 6 9 0
10 1 2 0 3 2 0
12 2 3 -4 0 7 11 0
14 0 11 10 1 0
```

Pruned

```
9 -3 0 5 4 0
11 -1 0 6 9 0
11 d 9 0
10 1 2 0 3 2 0
14 0 11 10 1 0
```

Figure 5.2: Partial proofs and combined and pruned proof, colored as in Fig. 5.1. The blue arrows indicate remote dependencies across the partial proofs.

5.3.1 Partial Proof Production

To combine the partial proofs into a complete proof, we modify the mechanism how LRAT proofs are produced in each of the individual sequential solvers. We assign to each clause an ID that is unique across solvers and identifies which solver originally derived it. The following mapping from clauses to IDs achieves this:

Definition 5.1

Let o be the number of clauses in the original formula and let p be the number of sequential solvers. Then the ID of the k-th derived clause $(k \geq 0)$ of solver i is defined as $ID_k^i = o + i + pk$.

Given ID_k^i, we can easily determine the producing solver i using modular arithmetic.

We extend our clause sharing to send each clause together with its ID. A receiving solver stores the clause with its ID and uses the ID in proof hints when the clause is used locally, as it does with locally-derived clauses. Unlike locally-derived clauses, we add no derivation lines for incoming clauses to the local proof. Instead, these derivations will be added to the final proof when combining the partial proofs.

5.3.2 Partial Proof Combination

Once the distributed solver reports unsatisfiability, we have p partial proofs. The derivations in these proofs can refer to clauses of other partial proofs, but they are, locally, in dependency order. We can thus combine the partial proofs without reordering their clauses beforehand. We can simply *interleave* their clauses so the resulting proof is also in dependency order, ignoring any deletions in the partial proofs.

Our *combination algorithm* traverses the partial proofs round-robin. At each step, we read and output the next clause c from the current partial proof as long as all dependencies of c have already been output. Checking whether a dependency d has already been output is simple: We determine which solver produced d (see Def. 5.1) and check if the next clause of the corresponding partial proof has an ID higher than d. Our algorithm terminates when it emits the empty clause.

Suppose that two clause-sharing solver instances, A and B, found the formula from Figure 5.1 to be unsatisfiable and emitted two partial proofs as displayed in Figure 5.2.

Starting with A, we can emit clause `9` (only depending on original clauses) and `11` (depending on original clauses and clause `9`). We cannot emit clause `13` since it depends on clause `12` from B. Proceeding with B, we can now emit the remaining clauses `10`, `12`, and `14`. Since clause `14` is the empty clause, we finish with the complete proof shown in Figure 5.2 (right). Note that clause `13` was not added to the combined proof—it was not required to satisfy any dependencies of the empty clause.

5.3.3 Proof Pruning

The combined proof our procedure produces is valid but not efficiently checkable because (1) it can contain superfluous clauses and (2) it does not contain deletion lines, meaning that a proof checker must maintain *all* learned clauses in memory throughout the checking process. To reduce size and improve checking performance, we *prune* our combined proof to only contain necessary clauses, and we add deletion statements for clauses as soon as they are not needed anymore.

Our *pruning algorithm* walks the combined proof *in reverse* to find all transitive dependencies of the empty clause, similar to backward checking of DRAT proofs [HHW13]. We maintain a set R of clause IDs *required* in the proof, initialized to the ID of the empty clause alone. We then read all clauses in reverse order, including the empty clause. When encountering a clause derivation whose ID is in R, we check for each of its dependencies whether this is the first time (from the proof's end) we see this dependency. In such cases, we can emit a deletion line for the dependency since it will never be used again in the proof. After checking all its dependencies, we output the required clause derivation and add its dependencies (except for original clauses) to R. The final output of the algorithm is a proof in reversed order, where each clause is required for some derivation and deleted as soon as it is no longer required. Reversing this output line by line yields a sound and compact proof.

Consider the combined proof from Figure 5.2. Working backward from clause 14, with clause IDs 11 and 10 added to R, we determine that clause 12 is *not* required, so it is ignored. Additionally, prior to clause 11, clause 9 is not in R, so it can be deleted after the derivation of clause 11. As such, we arrive at the pruned proof in Figure 5.2.

On realistic proofs, we show in Section 5.6 that pruning can sometimes reduce the proof size by several orders of magnitude.

5.4 Distributed Proof Production

The proof production as described above is sequential and may process huge amounts of data, all of which needs to be accessible from the machine that executes the procedure. In addition, maintaining the required clause IDs during the procedure can require a prohibitive amount of memory for large proofs. In the following, we propose an efficient distributed approach to proof production.

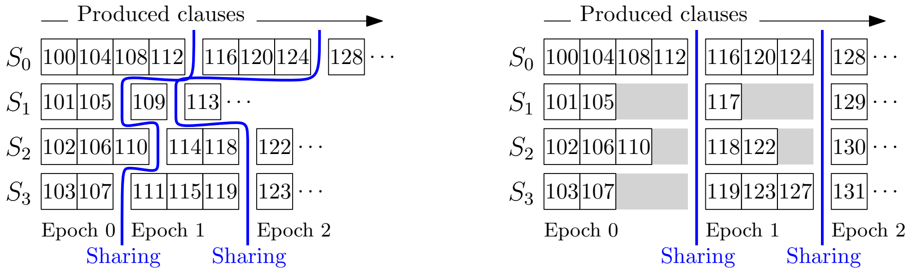

Produced clauses
S0 100 104 108 112 116 120 124 128 ···
S1 101 105 109 113 ···
S2 102 106 110 114 118 122 ···
S3 103 107 111 115 119 123 ···
Epoch 0
Epoch 1
Epoch 2
Sharing
Sharing
Produced clauses
S0 100 104 108 112 116 120 124 128 ···
S1 101 105 117 129 ···
S2 102 106 110 118 122 130 ···
S3 103 107 119 123 127 131 ···
Epoch 0
Epoch 1
Epoch 2
Sharing
Sharing

Figure 5.3: Four solvers work on a formula with 99 original clauses, produce new clauses (depicted by their ID) and share clauses periodically, without (left) and with (right) clause ID alignment.

5.4.1 Overview

Our sequential algorithm first combines all partial proofs into a single proof and then prunes unneeded proof lines. In contrast, our distributed algorithm first prunes all partial proofs in parallel and only then merges the required lines into one file.

We have m processes with c solvers each, amounting to a total of $p = mc$ solvers. We make use of the fact that the solvers exchange clauses in periodic intervals (see Section 4.3). We refer to these intervals between subsequent sharing operations as *epochs*. Consider Fig. 5.3 (left): Clause 118 was produced by S_2 in epoch 1. Its derivation may depend on local clause 114 and on any of the 11 clauses produced in epoch 0, but it cannot depend, e.g., on clause 109 or 111 since these clauses have been produced after the last clause sharing. More generally, a clause c produced by solver i during epoch e can only depend on (i) earlier clauses by solver i produced during epoch e or earlier, and (ii) clauses by solvers $j \neq i$ produced *before* epoch e.

Using this knowledge, we can *rewind* the solving procedure. Each process reads its partial proofs in reverse, outputs each line which adds a required clause, and adds the line's dependencies to the required clauses. Required *remote* clauses produced in epoch e are transferred to their origin before any process begins to read proof lines from epoch e. As such, whenever a process reads a proof line, it knows if the clause is required. We later explain how the outputs of all processes can be merged.

5.4.2 Clause ID Alignment

To synchronize the reading and redistribution of clause IDs in our distributed pruning, we need a way to decide from which epoch a remote clause ID originates. However, solvers generally produce clauses with different speeds, so the IDs by different solvers will likely be in dissimilar ranges within the same epoch over time. For instance, in Fig. 5.3 (left) solver S_3 has no way of knowing from which epoch clause 118 originates. To solve this issue, we propose to align all produced clause IDs after each sharing. During solving, we add a certain offset δ_i^e to each ID produced by solver i in epoch e.

As such, we can associate each epoch e with a global interval $[A_e, A_{e+1})$ that contains all clause IDs produced in that epoch. In Fig. 5.3 (right), $A_0 = 0$, $A_1 = 116$, and $A_2 = 128$. Clause 118 on the left has been aligned to 122 on the right ($\delta_2^1 = 4$) and due to $A_1 \leq 122 < A_2$ *all* solvers know that this clause originates from epoch 1.

Initially, $\delta_i^0 := 0$ for all i. Let I_i^e be the first original (unaligned) ID produced by solver i in epoch e. With the sharing that initiates epoch $e > 0$, we want to define the common start of epoch e, A_e, to be larger than all aligned clause IDs from epoch $e-1$ but no larger than any aligned clause ID from epoch e. We align each solver's first ID from epoch e via the prior offset: $I_i^e + \delta_i^{e-1}$. We then normalize each such ID by subtracting the solver's individual offset i. Since two subsequent unaligned clause IDs always differ by p (the total number of solvers) and since $i < p$, we know that $I_i^e + \delta_i^{e-1} - i$ is larger than the last ID solver i produced in epoch $e-1$. We thus compute A_e as the maximum of these values: $A_e := \max_i\{I_i^e + \delta_i^{e-1} - i\}$. Next, we want to compute new offsets δ_i^e in such a way that the first aligned clause ID of solver i in epoch e, $I_i^e + \delta_i^e$, is equal to $A_e + i$. Consequently, we set $\delta_i^e := A_e + i - I_i^e$.

If we export a clause produced in epoch e by solver i, we add δ_i^e to its ID, and if we import shared clauses to i, we filter any clauses produced by i itself. Note that we do not modify the solvers' internal ID counters nor the proofs they output, and there is no need to block or synchronize the solver threads at any time. Later, when reading the partial proof of solver i at epoch e, we need to add δ_i^e to each ID originating from i. All remote clause IDs in the partial proofs are already aligned.

5.4.3 Rewind Algorithm

Assume that solver $u \in \{1, \ldots, p\}$ has derived the empty clause in epoch $\hat{e}$. For each process-local solver i, each process has a *frontier* F_i of required clauses produced by i. In addition, each process has a *backlog* B of remote required clauses. B and F_i are maximum-first priority queues of clause IDs. Initially, F_u contains the ID of the empty clause while all other frontiers and backlogs are empty. Iteration $x \geq 0$ of our algorithm processes epoch $\hat{e} - x$ and features two stages:

1. Processing: Each process continues to read its partial proofs in reverse order from the final derived clause of the current epoch. If a line from solver i is read whose clause ID is at the top of F_i, then the ID is removed from F_i, the line is output, and each clause ID hint h in the line is treated as follows:

- h is inserted in F_j if local solver j (possibly $j = i$) produced h.
- h is inserted in B if a remote solver produced h.
- h is ignored if h is an ID of an original clause of the problem.

Reading stops as soon as a line's ID precedes epoch $e = \hat{e} - x$. Each F_i as well as B now only contain clauses produced *before* e.

2. Task redistribution: Each process extracts all clause IDs from B that were produced during $\hat{e} - x - 1$. These clause IDs are aggregated among all processes. For this means, we reuse our compact clause exchange operation (see Section 4.3.2), adjusted to aggregate clause IDs instead of clauses. This also allows us to eliminate duplicates among the redistributed IDs. Each process then traverses the aggregated clause IDs, and each clause produced by a local solver i is added to F_i.

Our algorithm stops in iteration $\hat{e}$ after the processing stage, at which point all frontiers and backlogs are empty and all relevant proof lines have been output.

5.4.4 Correctness

We now establish the correctness of our proof production. First, we show that our clause ID alignment works as intended:

Lemma 5.2

The alignment of clause IDs as described above results in a sequence $A_0, A_1, A_2, \dots, A_{\hat{e}}$ such that for any clause with unaligned ID j produced by the i-th solver, $A_e \leq j + \delta_i^e < A_{e+1}$ holds if and only if j was produced in epoch e.

Proof. We perform induction over epoch e in which a clause was produced.

For $e = 0$, we set $A_0 = 0$. The first sharing defines $A_1 = \max_i\{I_i^1 + \delta_i^0 - i\} = \max_i\{I_i^1 - i\}$. I_i^1 is the first clause ID the i-th solver produced in epoch 1 and i is smaller than the difference p between two of its subsequent clause IDs. Therefore, $I_i^1 - i$ is larger than any ID it produced in epoch 0. Consequently, A_1 is larger than any ID produced in epoch 0 by *any* solver. It follows that a clause with ID j was produced in epoch 0 if and only if $A_0 = 0 \leq j < A_1$.

Assuming that the lemma holds for clauses produced in epochs $0, \dots, e$, we show that the lemma also holds for clauses produced in epoch $e + 1$.

Due to induction, a clause from the i-th solver with unaligned ID j was produced in epoch e if and only if $A_e \leq j + \delta_i^e < A_{e+1}$. We need to show that a clause with unaligned ID j was produced in epoch $e + 1$ if and only if $A_{e+1} \leq j + \delta_i^{e+1} < A_{e+2}$.

The induction prerequisite enforces that A_{e+1} exactly separates the aligned clause IDs produced in epoch e from the aligned clause IDs produced in later epochs. Therefore, $A_{e+1} \leq j + \delta_i^{e+1}$ if and only if j was produced in epoch $e + 1$ *or later*.

Concerning the upper bound, our procedure defines $A_{e+2} = \max_i\{I_i^{e+2} + \delta_i^{e+1} - i\} = \max_i\{I_i^{e+2} + (A_{e+1} + i) - I_i^{e+1} - i\} = A_{e+1} + \max_i\{I_i^{e+2} - I_i^{e+1}\}$. Since $\delta_j^{e+1} = A_{e+1} + i - I_i^{e+1}$, it follows that $j + \delta_i^{e+1} < A_{e+2}$ holds if and only if $j + i - I_i^{e+1} < \max_i\{I_i^{e+2} - I_i^{e+1}\}$, which is equivalent to (A) $j < I_i^{e+1} + \max_i\{I_i^{e+2} - I_i^{e+1}\} - i$. Since the first clause ID produced by the i-th solver in epoch $e + 2$ is $I_i^{e+2} \geq I_i^{e+1} + \max_i\{I_i^{e+2} - I_i^{e+1}\} - i$, (A) holds if and only if j was produced *before* epoch $e + 2$. □

Next, we need to formally define a partial proof for an individual solver thread.

Definition 5.3

Let S be a sequential solver which runs within a distributed clause-sharing solver. A partial proof *for CNF formula $\mathcal{F}$ is a sequence $\mathcal{P} = \langle l_1, \dots, l_n \rangle$ of LRAT proof lines output by S without any clause deletions, where for each line $l_i = (j, c, D)$ with ID j, clause c, and dependencies D, (i) and (ii) hold:*

(i) Each dependency $d \in D$ references either (a) an original clause in $\mathcal{F}$ or (b) a clause derived in an earlier line l_j $(j < i)$ or (c) a clause from another partial proof for $\mathcal{F}$. There must not be any cyclic dependencies.
(ii) l_i constitutes a valid LRAT derivation of c if given the referenced dependencies.

The following theorem states the correctness of our proof production under the assumption that the individual solvers output valid partial proofs.

Theorem 5.4
Let $\mathcal{P}_1, \ldots, \mathcal{P}_m$ be the partial proofs for an unsatisfiable CNF formula $\mathcal{F}$ of a completed run of a distributed solver which performs all-to-all clause sharing with clause ID alignment as outlined above. Let $O := \langle O_1, \ldots, O_m \rangle$ be the proof line output of each solver thread from our rewind procedure, and let $\tilde{O}$ be a flat sequence of all proof lines in O sorted by ID in ascending order. Then $\tilde{O}$ constitutes a sound LRAT proof for $\mathcal{F}$.

Proof. First, we state that $\tilde{O}$ contains the empty clause due to construction: Since $\mathcal{F}$ is unsatisfiable and the distributed solver's run completed, the empty clause has been found by at least one solver and is consequently output by some solver at the beginning of the rewind procedure.

Next, we show for any line $l = (j, c, D) \in \tilde{O}$ that a linear pass through $\tilde{O}$ establishes all dependencies $d \in D$ before l itself is reached. Since $l \in \tilde{O}$, there is a solver i whose partial proof $\mathcal{P}_i$ contains l in epoch e and where j is considered required such that l is output. We distinguish three cases.

(a) If d references an original clause in $\mathcal{F}$, the dependency is trivially established.

(b) If d references an earlier clause derived in $\mathcal{P}_i$, then dependency d is inserted in F_i as l is read from $\mathcal{P}_i$. We know that $d < j$: Each solver assigns clause IDs in a strictly monotonic manner and the alignment of clause IDs preserves this property. Since the derivation of d is contained in $\mathcal{P}_i$ and since the IDs in $\mathcal{P}_i$ are processed in decreasing order, the line deriving d is read from $\mathcal{P}_i$ at some later point in time.
At this point, d must be at the top of F_i for the following arguments. IDs extracted from F_i are monotonically decreasing because F_i functions as a maximum-first priority queue and because each ID inserted in F_i is necessarily smaller than the last ID extracted from F_i. If a higher ID $d' > d$ is at the top of F_i, then the required dependency d' was not matched with any former line in $\mathcal{P}_i$ and, due to the processing order of IDs in $\mathcal{P}_i$, is not matched with any later line either. As our procedure ensures that F_i only contains IDs produced by solver i, this constitutes a contradiction to $\mathcal{P}_i$ being a valid partial proof. If a lower ID is at the top of F_i or if F_i is empty, then d was removed from F_i earlier, which means that d can be matched with several lines from $\mathcal{P}_i$—a contradiction to the uniqueness of derived clause IDs in $\mathcal{P}_i$.
Since d is at the top of F_i as its derivation is read from $\mathcal{P}_i$, d is considered required and thus output. Due to $d < j$, dependency d is in fact established before l is reached.

(c) In the third case, d references a clause c' from another solver's partial proof $\mathcal{P}'$. Due to the structure of clause sharing, any such remote clause c' originates from a strictly smaller epoch e' than the epoch e from which l itself originates.

Our clause ID alignment (Lemma 5.2) hence ensures that the ID d of c' is strictly smaller than j and that d would be featured in $\tilde{O}$ earlier than l.
It remains to be shown that $\tilde{O}$ does contain d. Since l is output, the remote dependency d is inserted either in backlog B_i (if d originates from a different process) or directly in the producing solver's frontier F'. In the former case, before epoch e' is processed, d is extracted from B_i, redistributed to the producing solver, and *then* inserted in that solver's frontier F'. During the processing of epoch e', the derivation of d is read from $\mathcal{P}'$. At this point in time, the ID d must be at the top of F' for exactly the same arguments as in case (b) for F_i. Since d is at the top of F', d is considered required and therefore output as well.

All in all, $\tilde{O}$ is a sequence of valid LRAT derivations in dependency order which eventually features the empty clause. Therefore, $\tilde{O}$ constitutes a sound LRAT proof of unsatisfiability for $\mathcal{F}$. □

To complement Theorem 5.4, we can also argue that any proof line $l \in \tilde{O}$ is necessarily a transitive requirement of the empty clause and that, therefore, $\tilde{O}$ is minimal in the sense that all lines in $\tilde{O}$ are in some way required for the proof at hand. There can still be ways to achieve smaller proofs for $\mathcal{F}$. For instance, the same clause c may be featured multiple times with different IDs in $\tilde{O}$ or there may be an entirely different, shorter chain of reasoning leading to the empty clause.

5.4.5 Analysis

In terms of total work performed, all partial proofs are read completely. For each required clause we may perform an insertion into some B, a deletion from said B, an insertion into some F_i, and a deletion from said F_i. If V_{in} is the combined size of all partial proofs, V_{out} is the size of the output proof, and we assume logarithmic work for each insertion and deletion, then the work for these operations is in $\mathcal{O}(V_{in} + V_{out} \log V_{out})$. In addition, due to the redistribution of clause IDs we have $\hat{e}$ iterations of communication whose overall cost is bounded by the communication done during solving. In fact, since only a subset of shared clauses is required and we only share 64 bits per clause, we expect strictly less communication than during solving. The computation of A_e for each epoch e during solving can be integrated into the all-reduction of the filter bitset v (see Section 4.3.4.c) and is therefore negligible.

In terms of memory usage, the size of each B and each F_i can be proportional to the combined size of all required lines of the according partial proofs. This memory requirement may become problematic for large-scale runs. We thus suggest to employ external-memory priority queues (e.g., [San00]) which keep most of their data on disk.

5.4.6 Merging Step

For each partial proof processed during the pruning step, we have a stream of proof lines sorted in reverse chronological order, i.e., starting with the highest clause ID. The remaining task is to merge all these lines into a single, sorted proof file.

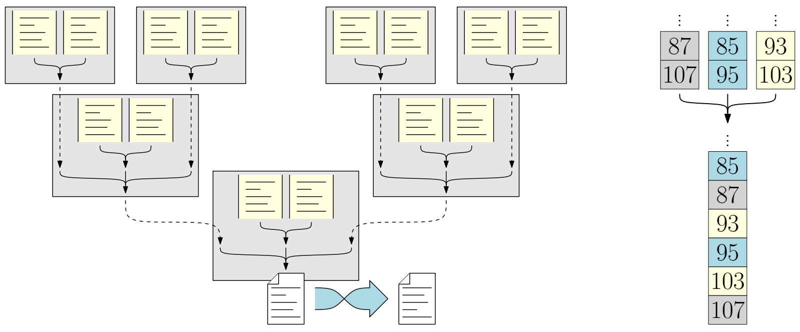


Figure 5.4: Left: Proof merging with seven processes and 14 solvers. Each box represents a process with two local proof sources. Dashed arrows denote communication. Right: Example of merging three streams of LRAT lines into a single stream. Each number i represents an LRAT line describing a clause of ID i.

We arrange all processes in a k-ary tree as shown in Fig. 5.4 (left) for $k = 2$. At each node of this tree, we can easily merge a number of sorted input streams into a single sorted output stream by repeatedly outputting the line with the highest ID among all inputs (Fig. 5.4 right). This way, we can hierarchically merge all streams along the tree. At the tree's root, the output stream is directed into a file. This is a sequential I/O task that limits the speed of merging. Finally, since the produced file is in reverse order, a buffered operation reverses the file's content.

In general, there may be more scalable ways to sort the available proof information (e.g., [JK03]), especially if we allow several processes to output slices of the final sorted proof in parallel. However, our algorithm assumes that (a) we require a single proof file on a single process and (b) the proof volume is so large that we need to stream it from disk memory. Moreover, since other steps in our pipeline (postprocessing and proof checking, see Section 5.6.1) process the final proof sequentially, our merging approach does not constitute a bottleneck.

A final challenge is to add clause deletion statements to the final proof. Before a line is written to the combined proof file, we can scan its hints and output a deletion line for each hint we did not encounter before (see Section 5.3.3). However, implementing this in an exact manner requires maintaining a set of clause IDs which scales with the final proof size. Since our proof remains valid even if we omit some clause deletions, we can use an approximate membership query (AMQ) structure with fixed size and a small false positive rate, e.g., a Bloom filter [Blo70].

5.5 Implementation

We employ a solver portfolio based on the sequential SAT solver CaDiCaL [Bie+20a]. We modified CaDiCaL to output LRAT proof lines and to assign clause IDs as described in Section 5.3.1. To ensure sound LRAT proof logging, we needed to turn off some features of CaDiCaL, such as bounded variable elimination, hyper-ternary resolution, and vivification. Similarly, MallobSat's original portfolio of CaDiCaL configurations features several options that are incompatible with our CaDiCaL as of yet. We thus created a smaller portfolio of "safe" configurations that include shuffling variable priorities, adjusted restart intervals, and disabled inprocessing. We also use different random seeds and sparse random variable phases (Section 4.4.2).

We modified MallobSat [Sch22] to associate each clause with a 64-bit clause ID. For consistent bookkeeping of sharing epochs, we defer clause sharing until all processes have fully initialized their solvers. While several solvers may derive the empty clause simultaneously, only one of them is chosen as the "winner" whose empty clause will be traced. The distributed proof production features communication epochs similar to MallobSat's clause sharing. To keep memory requirements of our proof assembly manageable even for huge proofs, we implement the clause ID priority queues F_i and B with a simple semi-external datastructure based on an in-memory priority queue Q for the current epoch and one external-memory stack E_e for each epoch e still to be processed. Upon reaching a new epoch e, all clause IDs from e are read from E_e and inserted into Q to allow for efficient polling and insertion.

To merge the pruned partial proofs, we use point-to-point messages to query and send buffers of proof lines between processes. We perform pruning and merging simultaneously to avoid writing the pruned partial proofs to disk. We use a fixed-size Bloom filter to add deletion lines to the final proof.

5.6 Evaluation

In this section, we present an evaluation of our proof production approaches. We provide all software and experimental data online (see Appendix A).

5.6.1 Experimental Setup

Supporting proofs introduces several kinds of performance overhead for clause-sharing portfolios in terms of solving, proof reconstruction, and proof checking. We wish to examine how well our proof-producing solver performs against (1) state-of-the-art (massively) parallel solvers that do not produce proofs, (2) previous approaches to proof-producing parallel solvers, and (3) state-of-the-art sequential solving with and without proof production. We analyze the overhead introduced by each phase of the process, and we discuss how and where future efforts might improve performance.

We use the following pipeline for our proof-producing solvers: First, the input formula is preprocessed via exhaustive unit propagation—a necessity due to a technical limitation of our LRAT-producing modification of CaDiCaL. Second, we execute our proof-producing variant of MallobSat on the preprocessed formula. Third, we prune and combine all partial proofs, using either our sequential proof production or our distributed proof production. Fourth, we merge the preprocessor's proof and our produced proof, compressing all clause IDs into a compact domain.

Fifth and finally, we run `lrat-check`[2] to check the final proof. Only steps two and three of this pipeline are parallelized (step three depending on the particular experiment). We refer to the first two steps as *solving*, the third step as *assembly*, the fourth step as *postprocessing*, and the fifth step as *checking*.

To analyze solving performance, we compare our parallel (MallobSatP64) and cloud (MallobSatP1600) solvers with proof production to several other solvers. First, we include the winners of the ISC 2022 cloud track (MallobSat1600-KCLG, using Kissat, CaDiCaL, Lingeling, Glucose), parallel track (ParkissatRS [ZCC22], using Kissat), and sequential track (KissatMABHyWalk [Zhe+22]), as well as the recent shared-memory parallel solver Gimsatul[3] which also supports proof production. In addition, we reconfigured MallobSat1600-KCLG to only use CaDiCaL (its original version, i.e., without LRAT capabilities) with the restricted configuration options used by MallobSatP1600 and MallobSatP64. We run this solver on a parallel (MallobSat64-C) and cloud (MallobSat1600-C) scale.

Since prior work on proof production for clause-sharing portfolios [HMP14] is no longer competitive in terms of solving time, we only compare proof-checking times. Specifically, we measure the overhead of checking un-pruned DRAT proofs as produced by the earlier approach [HMP14]. As such, we can get a picture of the performance of the earlier approach if it was realized with today's solving techniques. We generate un-pruned DRAT proofs from the original (un-pruned) LRAT proof by stripping out dependency information and adding delete lines for the last use of each clause.

We ran our experiments in Amazon Web Services (AWS) infrastructure. Specifically, following the ISC setup, each cloud solver runs on 100 `m6i.4xlarge` EC2 instances (16 hardware threads, 64 GB RAM), each parallel solver runs on a single `m6i.16xlarge` EC2 instance (64 hardware threads, 256 GB RAM), and the sequential KissatMABHyWalk runs on a single `m6i.4xlarge` EC2 instance. We use all 400 benchmark instances from ISC 2022. We set the timeout for the solving step to 1000 s and the timeout for all subsequent steps put together to 4000 s.

5.6.2 Results

First we examine the performance overhead of changing portfolios to enable proof generation (see Section 5.5) regarding solving times *only*. Fig. 5.5 and Table 5.1 show this data. Our CaDiCaL portfolio MallobSat64-C drastically outperforms KissatMABHyWalk as well as Gimsatul and is almost on par with ParkissatRS. Similarly, MallobSat1600-C solves eight instances less than MallobSat1600-KCLG but performs almost equally well otherwise. In both cases, we have constructed solvers which are almost on par with the state of the art.

For our proof-producing solvers MallobSatP64 and MallobSatP1600, we noticed a more pronounced decline in solving performance. On top of the overhead introduced by proof logging and our preprocessing, we experienced a few technical

[2]`https://github.com/marijnheule/drat-trim`
[3]We use the ISC 2022 data, where Gimsatul [BF22b] was still in an early stage of development (*cf.* [BFP23]).

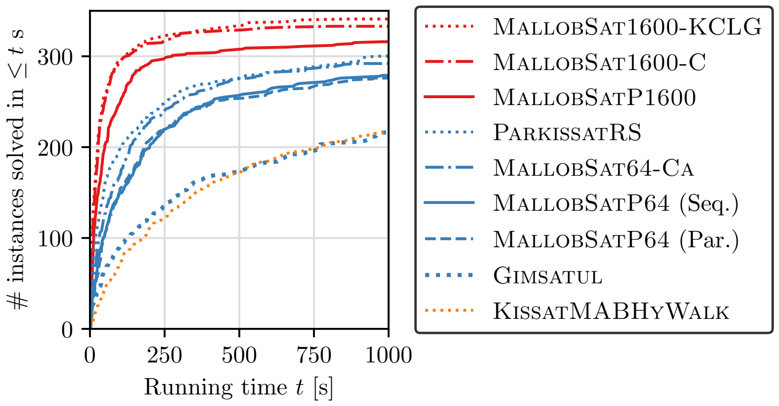


Figure 5.5: Solving times of considered solvers. MallobSatP1600, MallobSatP64, and KissatMABHyWalk output proof information during solving.

Type	Solver	#	# SAT	# UNSAT	PAR-2
S	KissatMABHyWalk	218	118	100	1065.7
P	ParKissatRS	300	155	145	603.0
	Gimsatul	216	119	97	1058.0
	MallobSat64-C	292	145	147	641.6
	MallobSatP64 (Seq.)	279	140	139	719.8
	MallobSatP64 (Par.)	276	141	135	731.4
C	MallobSat1600-KCLG	341	165	176	344.8
	MallobSat1600-C	333	163	170	378.0
	MallobSatP1600	316	159	157	480.5

Table 5.1: Performance of (S)equential, (P)arallel, and (C)loud solvers in terms of solved instances and PAR-2 scores (see Section 2.4.2.a).

problems, including memory issues,[4] which resulted in a drop in the number of instances solved and also caused MallobSatP64 with parallel proof production to solve three instances less than with sequential proof production. We believe that we can overcome these issues in future versions of our system. That being said, our proof-producing solvers do clearly outperform all of the solvers at a lower scale.

Next, we examine statistics on proof reconstruction and checking, showing results in Table 5.2. Since we want to investigate our approaches' overhead compared to pure solving, we measure running times as a multiple of the solving time. Note that we provide results in terms of absolute running times in Appendix C, Table 9.2.

[4]We disabled MallobSat's memory panic (Section 4.5.1) to ensure consistent proof logging.

Property	#	min	p10	med	mean	p90	max
DRAT check	81	0.512	1.725	7.442	10.370	67.065	169.869
Seq. assembly	139	0.019	0.305	1.376	1.387	5.747	13.289
Seq. postprocessing	139	0.001	0.012	0.131	0.112	0.790	2.218
Seq. checking	139	0.007	0.043	0.572	0.469	3.970	10.980
Seq. asm+post+chk	139	0.037	0.412	2.110	2.129	10.834	26.487
Par. assembly	135	0.059	0.080	0.365	0.408	2.227	7.475
Par. postprocessing	135	0.001	0.016	0.156	0.128	0.861	2.300
Par. checking	135	0.007	0.042	0.622	0.471	3.540	11.645
Par. asm+post+chk	135	0.067	0.167	1.097	1.062	6.611	21.420
Cld. assembly	157	0.121	0.194	1.680	1.204	5.348	43.853
Cld. postprocessing	157	0.003	0.051	0.744	0.634	4.744	35.667
Cld. checking	157	0.032	0.215	3.391	2.499	21.908	135.737
Cld. asm+post+chk	157	0.162	0.579	5.174	4.819	31.968	215.257
DRAT proof size (GB)	139	0.012	0.366	1.236	3.246	8.395	29.308
Seq. proof size (GB)	139	0.016	0.223	2.379	5.384	16.082	46.986
Par. proof size (GB)	135	0.006	0.173	2.034	5.345	13.164	57.739
Cld. proof size (GB)	157	0.016	0.269	4.595	11.138	34.457	92.276
Cld. pruning factor	157	2.080	5.312	16.472	28.319	299.858	8415.070

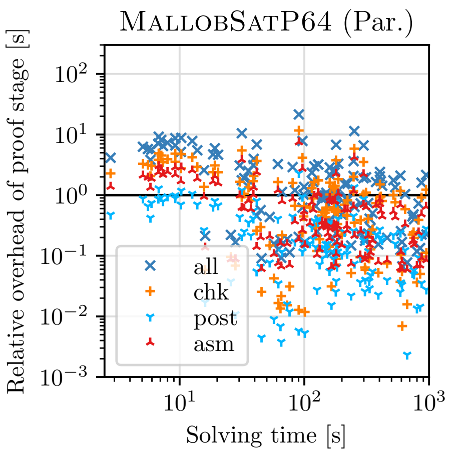
Table 5.2: Statistics on proof production and checking. All properties except for file sizes and pruning factor are given as a multiple of the solving time. We list minima, maxima, medians, means, and the 10th and 90th percentiles—using the arithmetic mean for proof sizes and the geometric mean for all ratios.

The prefix "Seq." denotes MallobSatP64 with sequential proof production, "Par." denotes MallobSatP64 with distributed proof production run on a single machine, and "Cld." denotes MallobSatP1600 with distributed proof production.

DRAT checking succeeded in 81 out of 139 cases and timed out in 58 cases. For the successful cases, DRAT checking took 10.4× the solving time[5] whereas our sequential assembly, postprocessing and checking *combined* succeeded in 139 cases and only took 2.1× the solving time. This result confirms that our approach successfully overcomes the major scalability problems of earlier work [HMP14]. In terms of uncompressed proof sizes, our LRAT proofs can be about twice as large as the DRAT proofs, which seems more than acceptable considering the dramatic difference in performance. Given that DRAT-based checking was ineffective at the scale of parallel solvers, we decided to omit it in our distributed experiments which feature even larger proofs.

The *parallel* proof production of MallobSatP64 reduces proof assembly times from 1.4× down to 0.4× the solving time, which also significantly reduces the overall overhead of proof production and checking (2.13× down to 1.06× the solving time). Fig. 5.6 (left) illustrates these relative overheads (y direction, as multiples of solving time) in relation to the actual solving time (x direction).

[5]Throughout this discussion, we use the geometric mean if we refer to a mean of ratios.



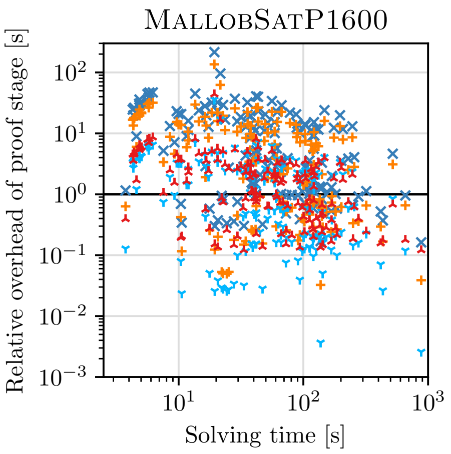


Figure 5.6: Overhead of proof-related stages (assembly, postprocessing, checking, and overall) relative to solving time, for MallobSatP64 with parallel proof production (left) and for MallobSatP1600 (right). Note the logarithmic scaling.

The results for MallobSatP1600 demonstrate that our proof assembly is feasible, still taking only around 1.2× the solving time on average. By contrast, the sequential stages of postprocessing and checking do not scale and therefore become more noticeable relative to the solving time (see Fig. 5.6 right). The proofs produced are about twice as large as for MallobSatP64. Considering that the proofs originate from 25 times as many solvers, this increase in size is quite modest, which is partly due to our proof pruning. We captured the *pruning factor*—the number of clauses in all partial proofs divided by the number of clauses in the combined proof—for each instance. Our pruning reduces the derived clauses by a mean factor of 28.3 (median 16.4) and by a factor of 300 or more for 10% of all instances. This underlines that our pruning is a crucial technique to feasibly combine and check proofs. We also managed to produce and check a proof of unsatisfiability for a formula whose unsatisfiability has not been verified before to our knowledge (`PancakeVsInsertSort_8_7.cnf`).

To compare our approaches with the state of the art in sequential solving, we analyzed `drat-trim` checking times of KissatMABHyWalk (Table 5.3), kindly provided by the competition organizers, and arrived at a mean overhead of 1.2× its own solving time. Using this data, we computed the speedups of our parallel trusted approaches over KissatMABHyWalk—once where both the sequential and the parallel approach perform solving only and once where both approaches perform solving, proof production, and checking. Note that the sequential solvers in the ISC are executed on different hardware than the parallel solvers. Our speedup measures are thus not fully reliable and only meant to give a rough impression.

	#	min	p10	med	(g/a)mean	p90	max
Ratio	146	0.137	0.269	1.109	1.208	6.394	66.494
Time	146	8.978	59.325	576.400	2675.813	5246.770	(Timeout)

Table 5.3: DRAT-trim checking overhead of KissatMABHyWalk in the SAT Competition 2022, in terms of multiples of its solving time ("Ratio") and in terms of absolute running times ("Time").

		#	min	p10	med	gmean	p90	max	total
Solve only	Seq. (64×)	263	0.028	0.742	3.869	3.806	23.224	122.854	4.429
	Par. (64×)	260	0.051	0.865	3.786	3.831	21.147	901.653	4.599
	Distr. (1600×)	283	0.100	2.179	10.887	11.253	64.624	1235.170	13.336
Solve+Check	Seq. (64×)	263	0.028	0.572	3.275	3.234	21.665	127.707	5.385
	Par. (64×)	260	0.051	0.866	4.120	3.851	21.217	901.654	6.472
	Distr. (1600×)	283	0.101	1.093	6.362	6.818	60.380	1235.170	7.226

Table 5.4: Speedups over KissatMABHyWalk in terms of solving times (top) and the entire trusted solving pipeline (bottom) for both approaches respectively.

Table 5.4 shows these speedups. The mean speedup in terms of pure solving times is about 4 with 64 solvers and about 11 with 1600 solvers—still comparable to the speedups which have earlier been reported by MallobSat's precursor HordeSat (with no proof production capabilities) at similar scales [BSS15]. In terms of the full trusted solving pipeline, our parallel proof production with 64 solvers actually achieves slightly larger speedups than if we only consider solving times—indicating that our LRAT-based proof production and checking pipeline is highly efficient and practical when compared to a sequential DRAT-based proof pipeline. The speedup at a distributed scale, by constrast, drops by roughly 40% when also considering the production and checking of proofs. As analyzed above, this is in large parts due to the sequential and therefore non scalable postprocessing and checking steps in our pipeline. While pre- and postprocessing is a technical necessity in our current setup, large portions of it can be eliminated in the future with further engineering. For instance, enhancing the LRAT support of solver backends to natively handle unit clauses in the input will allow us to skip preprocessing and simplify postprocessing.[6] As such, while the current speedups of our trusted solving pipeline are considerably below speedups achieved without proof production (see Section 4.6.3) due to different kinds of overhead, we consider them encouraging results towards efficient trusted general-purpose SAT solving in distributed environments.

[6]Most recently, CaDiCaL has indeed received full LRAT support [PFB23], which we were not able to exploit for the work at hand but which we intend to make use of in the near future.

5.7 Conclusion

Distributed clause-sharing solvers are currently the fastest tools for solving a wide range of difficult SAT problems. Nevertheless, they have previously not supported proof-generation techniques, leading to potential soundness concerns. In this chapter, we have examined mechanisms to add efficient support for proof generation to clause-sharing portfolio solvers. We have introduced a distributed system with reasonable SAT solving performance which takes about five times its own solving time to assemble and check a proof of unsatisfiability based on partial proofs generated during solving. As such, our results demonstrate that it is feasible to make distributed clause-sharing solvers fully trustworthy and therefore viable for critical applications.

Following our research, more work is required to reduce overhead in the different steps involved and to improve scalability of the end-to-end procedure. This may include designing more efficient (perhaps even parallel) LRAT checkers, examining proof-streaming techniques to eliminate most I/O operations, and improving LRAT support in solver backends. In fact, it might be possible to generalize our approach to DRAT-based solvers by adding additional metadata, and this might allow easier retrofitting of the approach onto larger portfolios of solvers. We also intend to investigate producing proofs in Mallob for the case where several MallobSat instances run concurrently and are rescaled dynamically (Chapter 3).

Chapter 6

Lifted Hierarchical Planning: A Case Study in Applied SAT

6

Domain-independent automated planning, or AI planning, is one of the oldest and most well-established applications of SAT solving. A planning task is encoded into a sequence of propositional formulas and a SAT solver is used iteratively in order to eventually find a plan to the task at hand. In Hierarchical Task Network (HTN) planning, a popular extension of automated planning, all SAT-based approaches so far need to perform an expensive preprocessing step called grounding, which oftentimes introduces a combinatorial blowup in terms of the size of the problem to encode. Our contribution named LILOTANE *(Lifted Logic for Task Networks) eliminates this issue for Totally Ordered HTN planning by directly encoding the lifted representation of the problem at hand. We lazily instantiate the problem hierarchy layer by layer and use a novel SAT encoding which allows us to defer decisions regarding method arguments to the stage of SAT solving. We show the correctness of our encoding and compare it to the best performing prior SAT encoding in a worst-case analysis. Empirical evaluations confirm that* LILOTANE *outperforms prior SAT-based approaches, often by orders of magnitude, produces much smaller formulas on average, and compares favorably to prior HTN planners regarding robustness and plan quality.*

Author's Notes. *This chapter is a shortened and adapted version of my journal article* "Lilotane: A Lifted SAT-based Approach to Hierarchical Planning" *[Sch21d], which featured purely original research. Large portions of this chapter are copied verbatim or with minor changes from that article. Previous publications [Sch+19a; Sch+19b] which emerged from my master's thesis [Sch18] provided the inspiration for this work.*

6.1 Introduction

Automated planning is a branch of AI which is sometimes referred to as "classical AI". In contrast to data-driven AI methods based on machine learning, automated planning deals with an idealistic, deterministic world and an agent with perfect knowledge [GNT04]. The agent can execute certain actions in order to manipulate the world state and eventually reach a certain goal (state). Domain-independent automated planning offers a generic interface to problem solving—not unlike SAT—and is used in spacecraft control [Fuk+97], autonomous robotics [RP12], logistics [Gar+13], puzzle generation and solving [BF22a], and business process management [Mar19].

Over the last decades, *hierarchical planning* gained traction and high popularity among researchers and users alike [GA15; BAH19]. Specifically, in *Hierarchical Task Network* (HTN) planning, the domain at hand is enriched with hierarchical expert knowledge which results in better guidance for planners and in well-structured and intuitive plans. In 2020, the collective interest in hierarchical planning resulted in the first *International Planning Competition* (IPC) for HTN Planning [BHB21]. Today's applications for HTN planning include robot planning [WOZ10] and coordination [OB05; Bev+15], web service composition [Sir+04], AI in video games [Mah+14], pedagogic courseware generation [Ull08], and automatic workflow design [Kie+12].

In addition to *operators* known from classical planning which are templates for atomic manipulations of the world state, HTN planning features *tasks* which provide an abstract notion of something that needs to be achieved, and *methods* which provide conditional "recipes" for decomposing a specific task into smaller tasks. In an HTN planning problem, a number of *initial tasks* are successively decomposed by applicable methods under the notion of stepwise refinement until the resulting tasks can be achieved atomically by operators [GA15]. In this work we focus on a highly popular[1] subclass of HTN planning where the domain model fixes a particular order on each method's subtasks, named *Totally Ordered HTN* (TOHTN) planning.

Among various established algorithms to resolve HTN planning problems as efficiently as possible [BAH19], one popular approach is to reduce the problem to SAT. First, the HTN planning problem at hand is transformed into an easier to handle representation through a preprocessing stage called *grounding* which instantiates all possible (i.e., reachable) argument combinations of each operator and each method. Then, the structurally simpler ground problem is encoded into a sequence of propositional logic formulas which are handed to a SAT solver. Eventually, if the problem is solvable then at some point a satisfying assignment will be found and can be decoded into a plan. Recently, various efficient SAT encodings were introduced [BHB18; BHB19a; BHB19b; Sch+19a; Sch+19b] while on the other hand new grounding approaches [Ram+17; Beh+20] result in smaller encodings and therefore better performance.

In all these recent contributions, grounding is essential and has, in fact, clear merits: The objects which result from grounding are structurally much simpler than in the *lifted*, i.e., un-grounded representation, and unreachable operators and methods can be identified as such and therefore be discarded. However, as a correct grounding procedure must enumerate all instantiations of operators and methods that may be part of a plan, grounding implies a combinatorial blowup in the worst case. As such, grounding can also be a heavyweight task and become a bottleneck for the whole planning procedure in terms of running time and memory footprint [WTH19]. For these reasons, planners which rely on a ground representation may fail to scale to large problems when compared to *lifted planners*, which directly operate on the parametrized structures and therefore do not perform any grounding.

[1]For instance, in the IPC 2020, the Total Order track had twice the number of competitors of the Partial Order track, and 40 totally ordered planning domains from nine different groups but only eleven partially ordered planning domains from two groups were submitted [BHB21].

With our contributions, we circumvent the problems tied to grounding for TOHTN planning by designing a SAT-based approach that omits this phase of the planning procedure. Our approach LILOTANE (*Lifted Logic for Task Networks*) generates an incremental sequence of propositional formulas from a still parametrized TOHTN problem description by instantiating operators and methods in a lazy manner and preserving free arguments where appropriate. As such, LILOTANE defers any non-trivial argument substitution choices up until the stage of SAT solving and therefore follows the well-known *least-commitment principle* which suggests to defer a planner's decisions for as long as possible [Wel94]. In terms of plan quality, we employ incremental SAT solving to iteratively tighten the bounds on possible plan lengths at a certain depth [Sch+19b] and obtain successively shorter plans in the process. Experiments indicate that our planning approach outperforms previous SAT-based HTN planners in terms of running times by more than one order of magnitude on the majority of instances and in most cases produces substantially smaller SAT encodings of TOHTN planning problems. LILOTANE also compares favorably to other HTN planners: A preliminary version of LILOTANE participated in the IPC 2020 and scored the second place. We conclude that LILOTANE is an appealing engine for TOHTN planning based on its robustness and effective quality awareness.

The chapter is structured as follows: First, in Section 6.2 we provide the necessary background for our work. Section 6.3 provides an overview of our planning approach and describes its different components as well as a number of optimizations and improvements. In Section 6.4 we provide our SAT encoding as well as a proof of correctness and a worst-case analysis of the encoding's size. We outline an anytime plan improvement technique for our approach in Section 6.5. In Section 6.6 we evaluate our approach. A conclusion and an outlook follow in Section 6.7.

6.2 Preliminaries

In this section we introduce necessary preliminaries for presenting our contributions.

6.2.1 TOHTN Planning

Several HTN planning formalisms with differing expressive power have been introduced [EHN96; ABA15]. We refine the model by Schreiber et al. [Sch+19b] as this work provides the foundation for our approach. In terms of expressiveness, this notation is equivalent to TOHTN planning with variables as specified by Alford et al. [ABA15] and compatible with the TOHTN formalism used in the IPC 2020 [BHB21].

6.2.1.a HTN Structures

We begin with some basic definitions. A *constant* $c \in C$ is an atomic symbol from a finite domain C. A *signature* $\sigma(a_1, \dots, a_k)$ is a syntactical construct with a name σ and a list of $k \geq 0$ *arguments*. k is a fixed constant for each σ. We call k the *arity* of σ.

Each argument a_i is either a constant or a *variable*; a variable acts as a placeholder for a constant. The domain of each a_i of σ is limited to a fixed subset $\tau_i \subseteq C$ of constants, the *type* of the i-th argument of σ. We call a signature *ground* if all of its arguments are constants, i.e., $\forall i \in \{1, \dots, k\} : a_i \in C$. We call a signature *lifted* if it is not ground. We use the term *free arguments* to refer to any variables left in a signature.

Predicates are special signatures which represent Boolean features of our problem's world state. A *fact* is a predicate supplied with a polarity (positive or negative). For any set s of literals, we define $s^+ := \{p \in s \mid p \text{ is positive}\}$ and $s^- := \{\neg p \mid p \in s,\ p \text{ is negative}\}$. A *state* s is a set of positive ground facts. When we interpret a state s as the world state of a planning problem, due to the closed-world assumption [Rei81] we assume that every fact *not* contained in s is negative.

A *task* is a non-predicate signature $t(a_1, \dots, a_k)$ and a syntactical footprint of something that needs to be achieved. An *operator* $o = (sig(o), pre(o), eff(o))$ is a tuple of a task $sig(o)$ and two sets $pre(o), eff(o)$ of literals whose arguments are arguments of $sig(o)$. An *action* is an operator o where $sig(o)$ is ground. A *method* is a tuple $m = (sig(m), task(m), pre(m), subtasks(m))$ where $sig(m)$ is a non-predicate signature, $task(m)$ is a task, $pre(m)$ is a set of literals, $subtasks(m) = \langle t_1, \dots, t_j \rangle$ is a sequence of $j \geq 0$ tasks, and all arguments in $task(m), pre(m)$, and $subtasks(m)$ are arguments of $sig(m)$. A *reduction* is a method m where $sig(m)$ is ground. Note that in general HTN planning, $subtasks(m)$ is not a sequence but rather a set of tasks supplied with a precedence relation. We will not pursue this more general case any further but restrict subtasks to be totally ordered, hence the name *Totally Ordered* HTN planning. We use the term *operation* to refer to an object that is either an action or a reduction.

Operators and methods are "recipes" to achieve an existing task, either by applying a matching operator that alters the world state or by replacing the task with the subtasks of a matching method. In both cases, the structures' preconditions $pre(\cdot)$ need to hold in the world state immediately before this refinement is performed. Given a method m for some task t such that $task(m) = t$, we say that m *matches* t and that t is *compound.* Given an operator o for some task t such that $sig(o) = t$, we say that o matches t and that t is *primitive.* Each task is *either* compound *or* primitive. Since we enforce signature names of distinct operators and methods to be unique, there can only be a single operator matching a primitive task whereas there may be several methods matching a single compound task.

6.2.1.b Problem Definition

A *TOHTN domain* (Totally Ordered Hierarchical Task Network) $D = (C, P, O, M)$ consists of constants C, predicates P, operators O, and methods M. The domain's actions A and reductions R are defined as the result of exhaustively substituting the arguments in each operator/method with all possible combinations of constants.

A *TOHTN problem* $\Pi = (D, s_I, T)$ consists of a TOHTN domain D, *initial state* s_I, and *initial task network* T. s_I is a state and T is a list of ground tasks. In the following, let "$\circ$" denote the concatenation of two sequences.

Definition 6.1

A sequence of actions π is a solution *to a TOHTN problem $\Pi = (D, s_I, T)$ iff one of the following cases holds and the resulting recursion is well-defined.*

(i) ***(Base case.)*** *$\pi = \langle\rangle$ and $T = \langle\rangle$.*

(ii) ***(Applying a reduction.)*** *$T = \langle t\rangle \circ T'$, $t = sig(r)$ for some $r \in R$, $pre^+(r) \subseteq s_I$, $pre^-(r) \cap s_I = \varnothing$, and π is a solution to $\Pi' := (D, s_I, subtasks(r) \circ T')$.*

(iii) ***(Applying an action.)*** *$T = \langle t\rangle \circ T'$, $\pi = \langle a\rangle \circ \pi'$, $t = sig(a)$ for some $a \in A$, $pre^+(a) \subseteq s_I$, $pre^-(a) \cap s_I = \varnothing$, and π' is a solution to:*
$\Pi' := (D, (s_I \setminus eff^-(a)) \cup eff^+(a), T')$.

Note that this definition directly provides a recursive algorithm to resolve a TOHTN problem—so-called *progression search* planners such as SHOP [Nau+99] are essentially refinements of this algorithm. Alternative (i) solves the empty problem where there is nothing to achieve ($T = \langle\rangle$) hence no actions are performed ($\pi = \langle\rangle$). In alternative (ii), a reduction r is applied which matches the current first task and whose preconditions hold in s: The matched task is replaced with the subtasks of r. In alternative (iii), an action a is applied which matches the current first task and whose preconditions hold in s: a is appended to the plan, its effects are applied to the current state, and the matched task is removed from the list of tasks yet to achieve.

The decisions left to a planner which follows the above algorithm are limited to picking a reduction in case (ii). This includes the decision for a particular method and the choice of argument substitutions that ground the method into a reduction. The third case does not induce any decisions: the tasks in T are invariantly ground, so there can only be one particular action matching any given task $t \in T$.

To reproduce and verify a solution, it should not only contain a flat sequence of actions but the full trace leading to this plan, which we define semi-formally as follows:

Definition 6.2 (semi-formal)

A directed tree $H = (V, E)$ with a total node ordering relation $\prec \subseteq V \times V$ is a hierarchical solution *to a problem Π if:*

(1) Each leaf node v corresponds to some action $a_v \in A$, and each inner node v corresponds to some reduction $r_v \in R$. In particular, the root node $\hat{v}$ corresponds to the initial reduction*, i.e., a reduction r_0 with $subtasks(r_0) = T$.*

(2) The children of an inner node u, ordered by $\prec$, correspond to the subtasks of r_u.

(3) A depth-first traversal of H from $\hat{v}$, using $\prec$ as a node ordering relation and departing from world state s_I, yields a sequence of operations which satisfy the definition of a solution π (Def. 6.1).

We formalize Def. 6.2 in Appendix D.1. Essentially, we define H as a witness for a particular "path" through Def. 6.1 yielding the given classical solution π. In particular, π can be read from H just by enumerating all leaf nodes in H according to $\prec$. The structure of H closely resembles the plan output format [BBH20] required for the IPC 2020, and H can be transformed easily into this format.

General HTN planning is a strictly semi-decidable problem [EHN94]. By contrast, TOHTN planning is decidable due to its more rigid and predictable structure [EHN94].

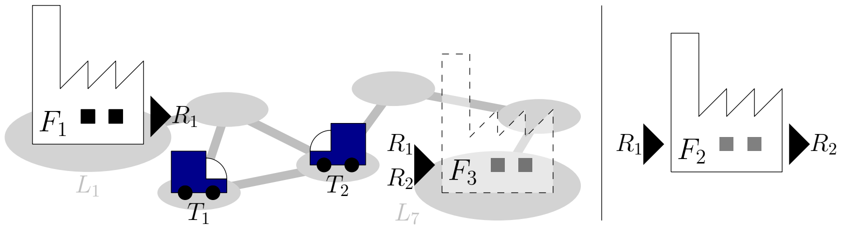
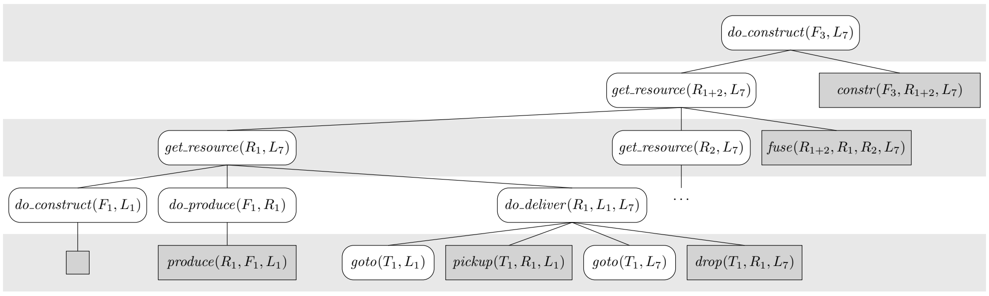


Figure 6.1: Example instance of the `Factories` planning domain.



$do\_construct(F_3, L_7)$
$get\_resource(R_{1+2}, L_7)$
$constr(F_3, R_{1+2}, L_7)$
$get\_resource(R_1, L_7)$
$get\_resource(R_2, L_7)$
$fuse(R_{1+2}, R_1, R_2, L_7)$
$do\_construct(F_1, L_1)$
$do\_produce(F_1, R_1)$
$do\_deliver(R_1, L_1, L_7)$
...
$produce(R_1, F_1, L_1)$
$goto(T_1, L_1)$
$pickup(T_1, R_1, L_1)$
$goto(T_1, L_7)$
$drop(T_1, R_1, L_7)$

Figure 6.2: Selected parts of a task network for above `Factories` planning instance. Primitive tasks are gray rectangles, compound tasks have rounded corners.

Specifically, the property that makes TOHTN planning decidable is that it prevents arbitrary interleavings of subtasks. However, Alford et al. showed that TOHTN planning in our setting, i.e., with variables, is 2-EXPTIME-complete [ABA15] and as such conjectured to be strictly more difficult than classical automated planning, which Bylander showed to be PSPACE-complete [Byl94].

A last technical detail to note is that Def. 6.2 alters the problem such that exactly one (virtual) reduction r_0 is the hierarchy's root which then features the initial tasks T as its subtasks. This transformation originates from `pandaPIparser` [Beh+20], a parser for (TO)HTN problem descriptions which we make use of.

6.2.1.c Example

Throughout the paper we will use the domain `Factories` [SS21c] as an example for our formalism. In the planning instance illustrated in Fig. 6.1, two trucks T_1, T_2 can transport resources from one location to another and a factory F_1 can indefinitely produce resource R_1. The objective is to construct factory F_3 at location L_7, for which resources R_1 and R_2 are required. In addition we have a blueprint for factory F_2 which is able to produce resource R_2. However, one unit of resource R_1 is consumed in order to construct F_2 and also for each unit of resource R_2 that F_2 produces.

In this planning domain we have predicates such as $at(o, l)$ (Is object o at location l?), $requires(f, r)$ (Does factory f require resource r to be built?) and $free(l)$ (Can a factory be built at l?). Possible operators include $move(t, l_1, l_2)$ to move truck t in between connected positions, *pickup(t, r, l)* and *drop(t, r, l)* for picking up and dropping resources, and so on. Fig. 6.2 depicts a partially expanded task network for our domain. The only initial task $do\_construct(F_3, L_7)$ is achieved via a method $m(f, r, l)$ for $f := F_3$, $r := R_{1+2}$ (where R_{1+2} represents the union of R_1 and R_2), and $l := L_7$. The method has preconditions $\{requires(f, r), free(l), \neg constructed(f)\}$ and subtasks $\langle get\_resource(r, l), construct(f, r, l)\rangle$. In words, we can construct F_3 at L_7 by bringing one unit of R_{1+2} to L_7 and then, atomically, performing the actual construction. We get resource R_{1+2} to L_7 by getting both R_1 and R_2 to L_7 and then fusing the two resources; we get R_1 to L_7 by ensuring that F_1 is constructed at L_1, producing R_1, and finally delivering it to L_7; and so on. In the bottom left, the chosen method for task $do\_construct(F_1, L_1)$ leads to an empty sequence of subtasks: This method has a precondition $at(F_1, L_1)$ which ensures that F_1 is already present, and thus expands to an empty "no-op" action.

As we repeatedly apply such methods and instantiate more and more *layers* of the network, we successively decompose the task network into more and more concrete tasks until only actions remain. If this sequence of actions is executable from left to right, beginning with initial state s_I, then we found a solution to our problem.

6.2.2 Grounding

While many HTN planners operate on the lifted problem description (e.g., [Nau+99; MMdS21]), ground approaches operate on a simplified and "flattened" representation of the problem. Through grounding, all facts, actions, and reductions which may be relevant to solve the problem are enumerated and compressed into compact data structures. Finding this subset of relevant facts and operations is a difficult problem in itself: For general HTN planning it can even be shown that it is undecidable whether a given action can be part of a plan [Beh+20].

Grounding procedures in automated planning generally perform graph-based reachability analyses to only accumulate instantiations which may be reachable during planning [Hel09]. The science of grounding HTN domains is comparably young as a designated area of research with only two notable prior publications [Ram+17; Beh+20]. Two different analyses that are employed in HTN grounding are (i) top-down reachability analyses where only the operations reachable from the problem's initial tasks over transitive subtask relationships are instantiated, and (ii) bottom-up reachability analyses. The latter perform a state-based reachability analysis on the classical planning problem, for example a *delete-relaxed* reachability analysis based on planning graphs [Hel09], obtain an upper bound on the set of reachable world states and actions, and discard any operations (and potentially their parent operations in the hierarchy) which are never applicable regarding their preconditions.

The grade of success of such approaches varies depending on the problem at hand. In many cases grounding is successful in practice and can be beneficial for the subsequent search algorithm [Ram+17]. For instance, grounding procedures can be able to prune an operation o because some unavoidable (transitive) child of o is impossible to achieve.

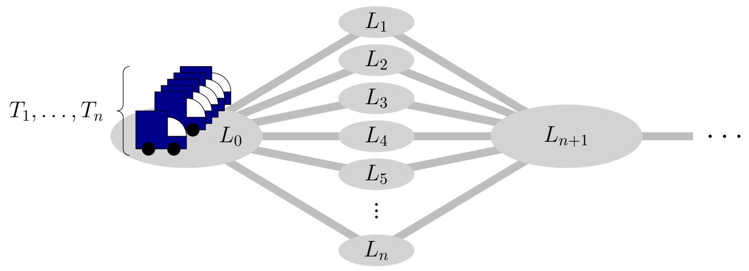


Figure 6.3: Excerpt of an artificial planning instance from the domain `Factories`

Lifted planners may get lost in search space without this knowledge. However, on some planning domains, grounding suffers from intrinsic scaling problems. Fig. 6.3 illustrates a simple example from the domain `Factories`. There are n trucks at L_0 and the objective of the problem (i.e., resources to be transported) is located at the far right beyond L_{n+1}. Assume that only a single truck is required for this objective. Any complete grounding procedure is required to instantiate operations for each of the n trucks to traverse any of the locations $L_1, \dots, L_n$ in order to get to L_{n+1} and achieve the actual task. None of the $\mathcal{O}(n^2)$ operations can be omitted because every operation is reachable and can be part of a plan.[2] More generally, domains can be constructed for which the minimum number of operations produced through grounding is a high order polynomial in the problem size. The maximum arity of any operation signature provides an upper bound on the polynomial's order. By contrast, a lifted progression search planner can simply make an ad-hoc decision on the truck and the route to take.

6.2.3 SAT-Based Planning

As we outlined in Section 2.2.5, most SAT encodings for planning problems encode a sequence of SAT formulas of increasing "problem horizon" until, at some point, a formula is found to be satisfiable. All relevant SAT-based classical planners have been following this kind of procedure [KS98; KSH06; Rin14], using the maximum number of considered steps as the problem horizon and sometimes allowing for several non-conflicting actions to be executed in a single step [Rin14].

Gocht and Balyo [GB17] introduced incremental SAT solving (Section 2.2.6.a) to SAT-based planning as an improvement. Automated planning can be considered a very natural application of incremental SAT solving since the sequence of propositional formulas to solve can be formulated as a single set of clauses that grows monotonically, with the exception of few assumption literals which represent that all goals are reached at the currently considered makespan. Schreiber et al. [Sch+19a; Sch+19b] were the first to adopt this technique for hierarchical planning.

[2]An interesting direction of research could be to explore *incomplete* grounding approaches which detect symmetries in the problem and only instantiate some sufficient subset of operations.

6.2.3.a Prior SAT-based HTN Planning

We now discuss prior SAT-based HTN planning approaches. For an overview of other HTN planning approaches, we refer to corresponding survey articles [GA15; BAH19].

The first propositional logic encodings for HTN planning problems [MK98] were restricted to non-recursive (or acyclic) domains. An HTN domain is non-recursive when the graph of all subtask relationships is acyclic. For such domains there is a fixed maximum number of actions any given task can induce, which renders the formalism relatively inexpressive and therefore arguably less interesting.[3]

After these initial encodings, two decades passed until Behnke et al. [BHB18] revisited the topic and developed a novel encoding approach which was designed for TOHTN planning, showing that it outperformed several prior HTN planning algorithms. Behnke et al. refined this approach to support general HTN planning [BHB19a] and optimal planning regarding the number of actions [BHB19b]. The employed encoding is expanded iteratively, but handed to a conventional SAT solver at each iteration. All these techniques have been integrated into the PANDA planning system, so we will refer to this branch of approaches as "PANDA-SAT".

Independently and almost simultaneously, Schreiber et al. [Sch+19a] developed a SAT encoding for TOHTN planning problems which exploits incremental SAT solving by simulating a stack machine of tasks and using the number of stack machine transitions as the problem horizon to increase. An enhancement of this approach resulted in the Tree-REX planner [Sch+19b] which performs significantly better than its precursor and allows to optimize plan quality.

Although their authors were unaware of each other's work, Tree-REX and PANDA-SAT share a similar encoding structure: The encodings are iteratively extended not along the length of a final plan (as is the case for encodings for classical planning) but instead along the depth of the hierarchy. The main difference between the encodings is that Tree-REX encodes states, preconditions, and effects at every layer of the problem while PANDA-SAT propagates all fact-based constraints to the current final layer. Both approaches rely on a grounding stage prior to the encoding and solving stage. Tree-REX uses the grounding procedure by Ramoul et al. [Ram+17] and enhances it by a top-down reachability analysis. The PANDA planning system uses a separate grounding procedure which was recently used as a basis for `pandaPIgrounder`, the most efficient HTN grounder to date [Beh+20]; yet, the original grounder of PANDA is significantly slower and of lower quality. A direct comparison by Schreiber et al. [Sch+19b] suggested that Tree-REX mostly finds plans faster and of comparable or better quality compared to PANDA-SAT.

6.2.4 Lifted Encodings

The idea of reducing the number of encoded actions in SAT-based planning ranges back to Kautz et al. [KS92] who proposed to "factorize" actions by splitting signatures.

[3] In the IPC 2020, 40 TOHTN planning domains but only three non-recursive domains were submitted, and an advertised Acyclic Track [BHB21] was cancelled due to lack of participants.

Based on this idea, factorized encodings have been established which encode arguments explicitly and can therefore be considered lifted SAT encodings [EMW97]. Executing multiple non-interfering actions at a single step can be problematic and requires nontrivial adjustments [Rob+09; Wil20]. To reduce grounding overhead and encoding size, Cashmore et al. [CFG13] suggested an approach based on Quantified Boolean Formulas (QBF). Bonet and Geffner [BG20] described a lifted "meta-encoding" of planning domains to infer a first-order symbolic representation from a state space structure. Most recently, Höller and Behnke [HB22] proposed a lifted encoding approach which avoids to encode (intermediate) states completely.

Our hierarchical encoding shares some of the issues of previous lifted encodings such as more complex frame axioms [EMW97]. Yet, due to the rigid layout of operations induced by the given problem hierarchy, we do not consider parallel action execution and are faced with new challenges and opportunities, as elaborated in the following.

6.3 Planning Approach

We now present our overall planning algorithm. In order to solve a TOHTN planning problem, we partition the hierarchy into a sequence of *hierarchical layers* $\{L_0, L_1, \ldots\}$. The first layer L_0 only contains the initial reduction r_0. For $i > 0$, layer L_i contains all operations which match a subtask of some operation at layer L_{i-1}. Intuitively, the index or *depth* i of a layer can be seen as the degree of refinement of the planning problem. Each layer is represented as a sequence of *positions* to account for the total ordering of all operations; scanning a layer from left to right chronologically traverses the possible operations at the given degree of refinement. The central challenge of planning, which we will delegate to a SAT solver, is to choose *exactly one* operation from each position of each layer such that a valid hierarchical solution emerges. We refer to these chosen operations as *active*.

The structure of our planning algorithm resembles the one of Tree-REX [Sch+19b]. As illustrated in Algorithm 6.1, we begin to construct the first two hierarchical layers L_0, L_1 of the problem, encode them into propositional logic, and then perform a first solving attempt of the formula in line 24 under the logical assumption that all active operations at the currently final layer are primitive, i.e., actions. As long as the SAT solver reports unsatisfiability, we construct the next layer, extend our formula by that layer's encoding, and attempt to solve it in the same way. When satisfiability is reported, we either directly decode and return a plan from the satisfying assignment or we employ a plan improvement procedure as desired (see Section 6.5).

6.3.1 Instantiation

Let us now take a closer look at how hierarchical layers are defined and constructed.

Let $P_{l,x}$ denote the x-th position of the l-th layer L_l. For $l = 0$, there is only one position in L_0, $P_{0,0}$, which only contains the initial reduction r_0. For $l > 0$, given layer

Algorithm 6.1 : LILOTANE Planning Procedure

```
Input : $\Pi = (D, s_I, T)$
Result : Plan $\pi$
1  Preprocess $\Pi$;                                          — parsing and simplification
2  $H := \langle\rangle$;
3  $L_0 := \langle$ CreateInitialPosition$(T, s_I)$ $\rangle$;
4  $H := H \circ \langle L_0 \rangle$;
5  $F := \varnothing$;                                              — global relevant facts
6  for $l = 0, 1, \ldots$ do                                     — instantiate new layer
7      $L_{l+1} := \langle\rangle$;
8      $S^0_{l+1} := (s_I, \varnothing)$;                          — reachable facts at this layer
9      $x' := 0$;
10     for $x = 0, \ldots, |L_l| - 1$ do
11         $e_{l,x} := \max\{1, \max\{|subtasks(r)| \mid r \in P_{l,x}\}\}$;   — max. expansion size
12         for $z = 0, \ldots, e_{l,x} - 1$ do
13             $P_{l+1,x'} :=$ Instantiate$(P_{l,x}, z, S^{x'}_{l+1})$;
14             $L_{l+1} := L_{l+1} \circ \langle P_{l+1,x'} \rangle$;
15             $S^{x'+1}_{l+1} := S^{x'}_{l+1} \cup possibleFactChanges(P_{l+1,x'})$;
16             $F := F \cup relevantFacts(P_{l+1,x'})$;
17             $x' := x' + 1$;
18         end
19     end
20     for $x' = 0, \ldots, |L_{l+1}| - 1$ do                        — encode new layer
21         Encode$(P_{l+1,x'}, F)$;
22     end
23     $H := H \circ L_l$;                                            — finalize layer
24     $result :=$ Solve$(H)$;                                       — attempt to solve
25     if $result$ is satisfying assignment then
26         while further plan improvement is desired do         — optional plan optimization
27             OptimizeCurrentPlan$(H)$;
28             if $plan$ $is$ $depth$-$optimal$ break;
29         end
30         return Decode$(H, result)$;
31     end
32 end
```

$L_l = \langle P_{l,0}, P_{l,1}, \ldots, P_{l,x}, \ldots \rangle$, we compute the possible children of the operations at each $P_{l,x}$ and append respective new positions to the subsequent layer L_{l+1}.

Consider the situation in Fig. 6.4: We are instantiating layer $l + 1$. Positions $P_{l,0}, \ldots, P_{l,x-1}$ of layer L_l have already been processed and resulted in new positions $P_{l+1,0}, \ldots, P_{l+1,x'-1}$ at layer L_{l+1} for some x'. Consequently, the children of $P_{l,x}$ begin at index x'. We refer to this index x' as $s_l(x)$, the first *successor position* (or child position) of $P_{l,x}$. In Fig. 6.4, $P_{l,x}$ features three possible operations: reductions r and r' and action a. Assume that r has three subtasks and r' has two subtasks.

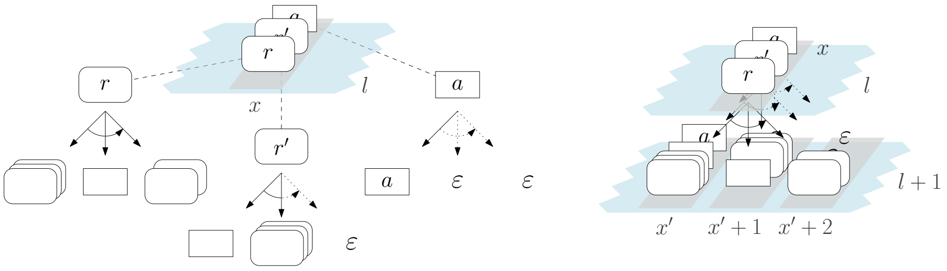


Figure 6.4: Layer construction. Rectangles are actions, rounded rectangles are reductions. Stacked operations denote a set of alternatives ("or"). Left: $P_{l,x}$ has three operations whose expansions are normalized to the maximum expansion size of three. Right: The child operations are aggregated into three new positions.

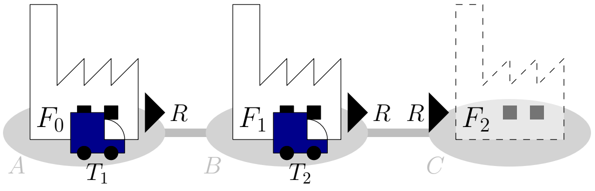


Figure 6.5: Simple planning example from the `Factories` domain.

Each subtask of an operation may be achieved by any of several operations, depending on whether an operator matches the subtask or, otherwise, how many distinct methods match the subtask. For the final plan, only a single operation from each position will be chosen. We denote the set of operations which can result from the z-th subtask of an operation o as $children(o, z)$ for $z \geq 0$. As we know for every operation how many children it must have (1 for an action and $|subtasks(r)|$ for a reduction r), we can compute the *maximum expansion size* $e_{l,x}$ of any operation at $P_{l,x}$.

As a consequence, we know that position $P_{l,x}$ induces $e_{l,x}$ child positions beginning from $s_l(x)$. To ensure that each child position is well-defined, we define $children(o, z) := \{\varepsilon\}$ if either (a) o is a reduction and $z \geq |subtasks(o)|$, or (b) o is an action and $z \geq 1$. We define ε as a special action with $pre(\varepsilon) = eff(\varepsilon) = \varnothing$ which we treat as a normal action but omit in the final plan. As such, in Fig. 6.4 we have constructed three new positions each of which contains the union of all children at the respective offset.

6.3.1.a Example

We now illustrate this approach with the `Factories` domain introduced in Section 6.2.1.c. The goal is to construct factory F_2 at location C which requires one unit of resource R (Fig. 6.5). This resource can be produced without any prerequisites either by F_0 at A or by F_1 at B and must be transported to C by truck T_1 or T_2.

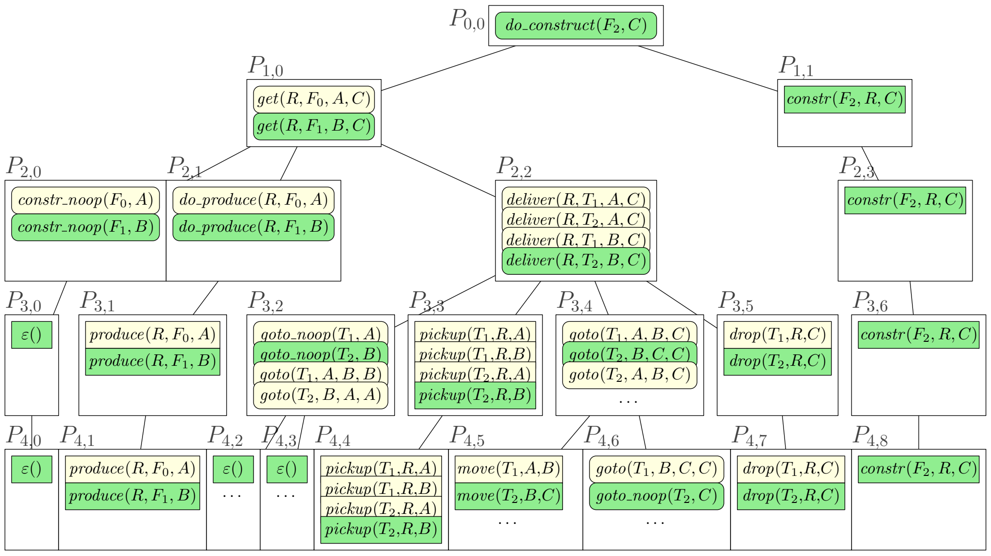


Figure 6.6: Five hierarchical layers of the `Factories` instance from Fig. 6.5. Each white rectangle is a position. Black lines connect a position to its child positions. Actions are displayed as rectangles, reductions are displayed as rounded rectangles. A set of operations leading to a valid plan is colored green.

Fig. 6.6 illustrates the layers instantiated by LILOTANE for solving this planning problem. The tree-like structure is similar to the illustration of a task network in Fig. 6.2. However, note two important differences: First, while the nodes in Fig. 6.2 represent compound and primitive *tasks*, the nodes in the above tree feature *reductions and actions* instead. Secondly, the task network in Fig. 6.2 represents one particular (partial) expansion of a problem whereas the structure in Fig. 6.6 represents *all* possible expansions (abbreviated where necessary).

Layers L_0 through L_4 are displayed from top to bottom. L_0 only contains a single initial reduction $do\_construct(F_2, C)$ at position $P_{0,0}$.[4] This reduction induces two child positions at layer L_1, $P_{1,0}$ and $P_{1,1}$, with all possible operations which match the first and second subtask of $do\_construct(F_2, C)$ respectively. Further down, we omitted some operations at positions marked with "…".

An "almost complete" hierarchical plan for the problem at hand is colored green. This plan involves producing R at factory F_1 and using truck T_2 to transport it in a single move action to location C where R is used to construct F_2. The plan is not entirely finished: Operation $goto\_noop(T_2, C)$ at $P_{4,6}$ is not primitive and still needs to be concretized. This reduction at $P_{4,6}$ will decompose into an ε-action at the next layer L_5. Then all chosen operations at the final layer are primitive and we found a solution.

[4]For the sake of simplicity, the illustration deviates from our actual model where the initial reduction is a virtual operation which features the actual initial reduction(s) as its children.

Note that the inability of our basic approach to find a plan at L_4 in the above example can be corrected by treating certain reductions as actions, as introduced in earlier work [Sch+19b]. We refer to the full journal article for details [Sch21d].

6.3.1.b Pseudo-Constants

Unlike previous SAT-based approaches which perform a complete grounding, LILOTANE lazily instantiates each operation from a parent's definition just when needed. In addition, as we explain next, this instantiation is done minimalistically: We do not fully instantiate child operations with free arguments but instead keep them lifted.

Consider position $P_{1,0}$ in Fig. 6.6. On an intuitive level, this position features the production and transportation of resource R to location C. Both F_0 and F_1 are able to produce R. In addition, at position $P_{2,2}$ truck T_1 or T_2 must be chosen to transport R. The combination of these two decisions leads to *four* different deliver operations at position $P_{2,2}$. In general, the full instantiation of such argument combinations can drastically increase the number of operations at each position (see Section 6.2.2).

To alleviate this issue, we keep method arguments instead of instantiating them. Let ϕ be a factory, λ the location of ϕ, and θ a truck. Then we can express both operations at $P_{1,0}$ as $\mathit{get}(R, \phi, \lambda, C)$ and all four operations at $P_{2,2}$ as $\mathit{deliver}(R, \theta, \lambda, C)$. We call the new argument symbols *pseudo-constants*: At a later point, ϕ must be substituted with either F_0 or F_1, θ must be substituted with either T_1 or T_2, and so on. The encoding we present in Section 6.4 will allow us to let a SAT solver decide which substitution to apply for each pseudo-constant. In general, for each free argument a_i of operation o, we initialize the *effective domain*, $\mathit{dom}(\alpha_i)$, of pseudo-constant α_i with the argument type τ_i. We then remove any constants from $\mathit{dom}(\alpha_i)$ for which a precondition of o becomes impossible at the current position. (We explain in Section 6.3.2 how this can be checked.) Only if $|\mathit{dom}(\alpha_i)| = 1$, we directly substitute a_i with the only valid constant. Essentially, instead of introducing $\prod_i |\mathit{dom}(\alpha_i)|$ operations, we introduce one lifted operation with $\sum_i |\mathit{dom}(\alpha_i)|$ pseudo-constant values to handle. We still need to enumerate and encode all (ground) preconditions and effects that can result from such a lifted operation. Similar to other works on lifted planning [Cor+20], we build upon the assumption that the problem features significantly fewer ground facts than (reachable) ground operations.

Fig. 6.7 applies the example from Fig. 6.6 to the use of pseudo-constants. This figure serves as a running example throughout the following sections and hence also introduces certain fact collections at each layer which we will discuss in Section 6.3.2. For now, let us concentrate on the operations that occur within the layers. At $P_{1,0}$ we introduce pseudo-constant ϕ for the factory producing R and at $P_{2,2}$ we introduce pseudo-constant θ for the truck to transport R to C. At positions $P_{1,0}, P_{3,2}, \ldots$ we introduce pseudo-constants $\lambda_1, \lambda_2, \ldots$ to represent particular locations. Just like normal arguments, pseudo-constants are propagated down to the (transitive) children of the operation they originated from. In the top left corner the chosen substitution for each relevant pseudo-constant is displayed: This information is essential to decode

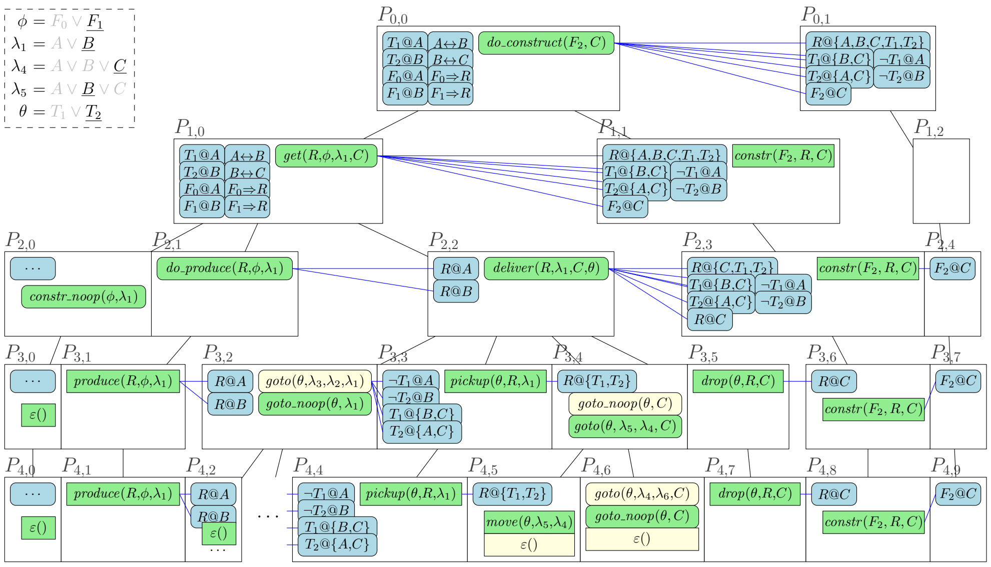


Figure 6.7: Hierarchy as in Fig. 6.6 but with pseudo-constants and reachable facts. Blue boxes with rounded corners represent facts occurring for the first time in a layer. Horizontal lines connect operations with "new" facts they may cause.

a valid plan from the chosen operations. A pseudo-constant is only relevant if the operation it originates from is part of the (hierarchical) solution.

6.3.2 Reachability Analysis for Facts and Operations

In order to minimize the number of considered operations, we perform a reachability analysis at each layer where we examine the possible world states at each position.

For each operation o which we instantiate, we define $pfc(o)$ as the *possible fact changes* of o—an over-approximation of all positive and negative facts which may be caused by o or by any transitive child of o. At each layer L_{l+1}, we construct and successively update S_{l+1}, which represents all such facts which may have been effected up to a certain position. We define the x-th update of S_{l+1} as follows:

$$S^x_{l+1} := (+S^x_{l+1}, -S^x_{l+1}) := \Big(\bigcup_{i=0}^{x-1} \bigcup_{o \in P_{l+1,i}} pfc(o)^+, \bigcup_{i=0}^{x-1} \bigcup_{o \in P_{l+1,i}} pfc(o)^- \Big).$$

As such, $+S^x_{l+1}$ ($-S^x_{l+1}$) consists of all positive (negative) facts that may be produced by some operation up to the x-th position.

For position $P_{l,x}$, we define a positive fact f to be *reachable* at $P_{l,x}$ if $f \in s_I \cup +S^x_l$. Similarly, a negative fact $\neg f$ is reachable at $P_{l,x}$ if $f \in -S^x_l$ or $f \notin s_I$. If fact f is not reachable at $P_{l,x}$, then we call f *invariantly false* at $P_{l,x}$.

If the negation $\neg f$ of a fact f is not reachable at $P_{l,x}$, then we call f *invariantly true* at $P_{l,x}$. Note that if fact f is invariantly true (false) at $P_{l,x}$, then $\neg f$ is invariantly false (true) at $P_{l,x}$. We use S_l and these definitions to check the invariance of preconditions in the "Instantiate" procedure in line 13 of Alg. 6.1 to prune impossible operations.

Fig. 6.7 illustrates a number of facts. We write $F_0@A$ to denote that factory F_0 is present at location A, $A{\leftrightarrow}B$ to denote that there is a path between A and B, and $F_0 \Rightarrow R$ to denote that factory F_0 produces resource R. Some redundant facts are omitted: Each fact $(\neg)\phi@\lambda$ for factory ϕ and location λ also implies $(\neg)$*constructed*(ϕ). In each position $P_{l,0}$ all facts in the initial state are displayed (abbreviated for $l \geq 2$). In each position $P_{l,x}$ for $x > 0$ all facts are displayed which were invariantly false at $P_{l,0}, \dots, P_{l,x-1}$ and which become reachable at $P_{l,x}$. Horizontal lines indicate which facts are caused by which operation. For instance, *get*(R, ϕ, λ_1, C) at $P_{1,0}$ may cause resource R to be located anywhere (due to *do_produce* and *deliver*) and both trucks T_1, T_2 to be located anywhere (because the *goto* subprocedure in *deliver* is recursive and may lead a truck to any location). These fact changes include the negative facts $\neg T_1@A$ and $\neg T_2@B$ as they were unreachable before (because $T_1@A$ and $T_2@B$ hold initially). According to *pfc*$(\cdot)$, the operation may even cause F_2 to be constructed—this false positive will be discussed further in Section 6.3.2.b.

Reduction $r =$ *get*(R, ϕ, λ_1, C) at $P_{1,0}$ encompasses the construction of some factory ϕ at location λ_1 as its first subtask. Without any further knowledge, we would assume that *dom*$(\phi) = \{F_0, F_1, F_2\}$ and *dom*$(\lambda_1) = \{A, B, C\}$. However, r has a precondition $\phi \Rightarrow R$. According to S_1^0, this precondition is invariantly false for $\phi = F_2$. For this reason, we initialize ϕ with a smaller domain, *dom*$(\phi) = \{F_0, F_1\}$. Reducing the domain of a pseudo-constant is a form of pruning, as it cuts the number of ground operations that are represented by the enclosing lifted operation.

It can also occur that an entire operation is pruned. The first subtask of r, namely the construction of ϕ at λ_1, is achieved at position $P_{2,0}$. This task can be matched by *constr_noop*(ϕ, λ_1) (with a precondition $\phi@\lambda_1$) or *do_construct*(ϕ, λ_1) (with a precondition $\neg$*constructed*(ϕ)). As both F_0 and F_1 are already constructed according to S_2^0, we know that the precondition of *do_construct*(ϕ, λ_1) is invariantly false for *all* substitutions of ϕ. As such, this operation is not included in position $P_{2,0}$.

Note that we implemented a kind of *precondition inference* which, similar to the *pullup* compilation in HyperTensioN [MMdS21], propagates such crucial preconditions up the hierarchy. This helps preserve the effectiveness of our instantiation approach even for domains which feature preconditions only in actions but not in methods. We refer to the full journal article for details [Sch21d].

6.3.2.a Computation and Correctness

We now describe how to efficiently compute our reachability analysis. S_{l+1}^0 is initialized with the initial state s_I and no negative facts in line 8 of Alg. 6.1. All facts not featured in S_{l+1}^x are considered invariantly false.

For $x' \geq 0$, we construct $S_{l+1}^{x'+1}$ via *possibleFactChanges*$(P_{l+1,x'})$ in line 15 of Alg. 6.1. In that call we collect all possible fact changes of $P_{l+1,x'}$, $PFC_{l+1,x'} := \bigcup_{o \in P_{l+1,x'}} pfc(o)$, and then update $S_{l+1}^{x'+1} := (+S_{l+1}^{x'} \cup PFC_{l+1,x'}^{+}, -S_{l+1}^{x'} \cup PFC_{l+1,x'}^{-})$. For any operation o, we compute $pfc(o)$ as follows:

If o is primitive, then $pfc(o) = g(\mathit{eff}(o))$, where $g(\cdot)$ is the *ground hull* of a set of facts: Any lifted fact in $\mathit{eff}(o)$ is fully instantiated into a set of ground facts.

Otherwise, $pfc(o) = g(\bigcup_{z=0}^{e_{l,x}} \bigcup_{o' \in children(o,z)} pfc(o'))$, i.e., we recursively compute the possible fact changes of each possible child of o and ground the resulting facts.

To avoid infinite recursion, we remember each visited method together with its subset of ground arguments and break recursion when an equivalent signature occurs again. We ensure that fact changes are computed only once per operator and method.

We now establish a central (semi-formal) correctness property of S_l:

Theorem 6.3

For $l \geq 0$ and $x \geq 0$, let $O_l := \langle o_0, \ldots, o_x \rangle$ be a sequence of operations where each o_i is a possible operation at position $P_{l,i}$. Decompose each $o_i \in O_l$ to some sequence $\mathcal{O}_i$ of actions such that $\mathcal{O} := \mathcal{O}_0 \circ \ldots \circ \mathcal{O}_x$ is executable from s_I.
(1) If fact f holds after executing $\mathcal{O}$, then f is reachable at $P_{l,x+1}$ according to S_l^{x+1}.
(2) Similarly, if f does not hold after executing $\mathcal{O}$, then $\neg f$ is reachable at $P_{l,x+1}$.

The proof of this theorem is given in Appendix D.2. As a direct consequence, if some fact $(\neg)f$ is not reachable at some position, then there is no way how any execution of the operations before this position could lead to f being true (false). For this reason, we can safely prune any operations for which a precondition is not reachable according to S_l, as in this case the precondition is definitely impossible.

6.3.2.b Relevant Facts and Retroactive Pruning

The procedure we just described gains knowledge the deeper we explore the problem hierarchy: The more concrete our operations become, the more exact $pfc(\cdot)$ becomes. For instance, in Fig. 6.7, $F_2@C$ is invariantly false at $P_{1,0}$ but reachable at $P_{1,1}$: As $pfc(\cdot)$ ignores the preconditions of children, it finds that operation $get(R, \phi, \lambda_1, C)$ has a possible child *do_construct*$(\cdot)$ which might achieve the construction of F_2. One layer later, it becomes apparent that this is an over-approximation: There is no operation before $P_{2,3}$ which can cause the construction of F_2.

One consequence of this stepwise refinement is that some facts which in practice are irrelevant for the planning task may be added to S_l. To counteract this, we maintain a set of *relevant facts* F which grows monotonically with each further layer. A fact is *relevant* and consequently added to F if it occurs as a (positive or negative) action effect or as a (positive or negative) precondition of some operation added to the current layer. Later, when encoding the layer, we can check for each fact whether it is relevant and should be encoded. Otherwise, we know that the fact cannot be actively involved in the planning task so far and we omit it from our encoding.

Another consequence of our technique is that we may encode an operation o at layer L_l and then notice at layer L_{l+k} that a precondition of some required transitive child o' of o is not reachable. In this case, o turns out to be impossible to achieve.

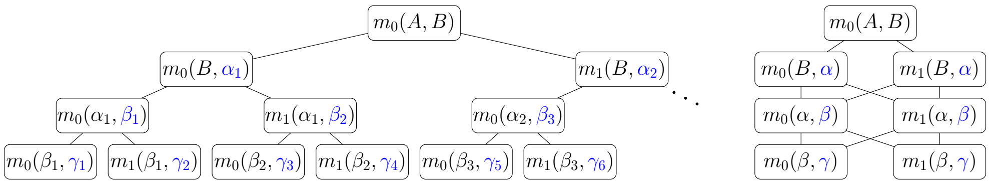


Figure 6.8: Naive (left) and true (right) children of m_0. New pseudo-constants are colored blue. An edge from u to v denotes that v matches a subtask of u.

We mentioned in Section 6.2.2 how ground approaches are often able to prune such operations *a priori* while our approach does not have the necessary knowledge to do so. In our approach, if we encounter a subtree of impossible operations, we (i) prune this subtree from our instantiated structures and (ii) add a unit clause to our encoding that forbids the root operation o to be chosen. As such, while the pruned subtree remains in our encoding, it is logically "switched off" and will not be explored any further. We refer to our journal article for details [Sch21d].

6.3.3 Shared Pseudo-Constants and Dominated Operations

Our approach is based on the idea that every operation at some position will induce one new pseudo-constant κ for each of its free arguments, with domain $dom(\kappa)$ as described in Section 6.3.1.b, and that child operations at subsequent layers naturally inherit some of these pseudo-constants in addition to introducing new pseudo-constants themselves. There are scenarios where this leads to undesired behavior. For example, consider methods $m_0(a,b)$ and $m_1(a,b)$ with $subtasks(m_0(a,b)) = subtasks(m_1(a,b)) = \langle t(b) \rangle$ where task $t(b)$ can be achieved both by $m_0(b,a)$ and $m_1(b,a)$. The transitive children of $m_0(a,b)$ will blow up in our approach as depicted in Fig. 6.8 (left): At each further layer, each operation branches into two new operations, and all operations syntactically differ due to the unique names of pseudo-constants. As such, the number of operations grows exponentially in the explored depth whereas the true structure of m_0 only results in two distinct operations as depicted in Fig. 6.8 (right).

As only a single operation can be active at each position, we suggest to share the same new pseudo-constant among multiple operations. When two or more operations have a free argument which would lead to exactly the same effective domain of a pseudo-constant, we introduce the same pseudo-constant for these arguments. With this change, the recursive children of $m_0(a,b)$ are already computed properly in our minimalistic example. Secondly, to account for pseudo-constant domains that are similar but not identical, we unify certain operations after their instantiation. We define that an operation o *dominates* another operation o' if (i) $sig(o)$ and $sig(o')$ are syntactically equivalent except for any number of argument positions i where both arguments a_i of o and a'_i of o' are pseudo-constants, and (ii) both a_i and a'_i originate from the same position and $dom(a_i) \supseteq dom(a'_i)$ for each such i. After instantiating a position P, we identify operations $o' \in P$ dominated by another operation $o \in P$.

In such a case, we remove each dominated operation o' from P and update the possible parents of o' to feature o as a child instead. In addition, we logically restrict the pseudo-constants of o to be equivalent to those of o' as necessary.

6.4 Encoding

We now present our encoding $\mathcal{L}_l(\Pi)$ of layers $L_0, \dots, L_l$ of a TOHTN planning problem Π into propositional logic. We first provide succinct definitions for all clauses to encode. Thereafter, we explain how to decode a plan from a satisfying assignment, refer to a proof of correctness, and present a worst-case complexity analysis.

6.4.1 Base Encoding

Some parts of the following encoding, namely those specified in Section 6.4.1.a, are taken from the Tree-REX approach [Sch+19b]. The fundamental difference between Tree-REX and our new encoding is that we must now handle lifted actions, reductions, and, consequently, lifted fact constraints.

We call operations in our hierarchy actions and reductions regardless of whether they contain pseudo-constants or not. We use the term *ground fact* for a fact without pseudo-constants and the term *pseudo-fact* for a fact with pseudo-constants. We use the following Boolean variables: o^l_x denotes that operation o is active at position $P_{l,x}$ and f^l_x denotes that ground fact f holds at $P_{l,x}$. Variables $prim^l_x$ represent whether a primitive operation, i.e., an action, is active at position $P_{l,x}$. In addition, for each pseudo-constant κ introduced to the problem we introduce variables "$[\kappa/c]$" for each $c \in dom(\kappa)$ which represent that κ is substituted with constant c.

6.4.1.a Basic Constraints

We begin with enforcing the initial reduction at the only position of the first layer:

$$(r_0)^0_0 \tag{6.1}$$

To avoid encoding superfluous facts, we make use of the set F_l of relevant facts (see Section 6.3.2.b). We introduce a Boolean variable for each relevant fact and enforce it to assume a polarity according to the initial state at the zeroth position:

$$\begin{aligned} \forall f \in F_l \cap s_I &: \quad f^l_0 \\ \forall f \in F_l \setminus s_I &: \quad \neg f^l_0 \end{aligned} \tag{6.2}$$

If an action a occurs at position x at layer l, then we define the respective position as primitive. Similarly, if a reduction occurs, we define the position as non-primitive.

$$\begin{aligned} a^l_x &\Rightarrow prim^l_x \\ r^l_x &\Rightarrow \neg prim^l_x \end{aligned} \tag{6.3}$$

At most one action and at most one reduction may occur at the same position. Together with Eq. 6.3, this enforces that at most one operation is active at each position.

$$\begin{aligned} \forall a \neq a' \in P_{l,x} &: \neg a_x^l \vee \neg (a')_x^l \\ \forall r \neq r' \in P_{l,x} &: \neg r_x^l \vee \neg (r')_x^l \end{aligned} \tag{6.4}$$

This constraint adds $\mathcal{O}(n^2)$ clauses if there are n actions, or reductions, at the same position. In our case this is not as problematic as for Tree-REX because we generally instantiate significantly fewer operations. Still, if n does become too large, we use a logically equivalent encoding which introduces $\log(n)$ helper variables but only encodes $\mathcal{O}(n \log(n))$ clauses. Based on preliminary experiments we use the latter encoding if $n \geq 50$. We refer to Schreiber [Sch18, Appendix A] for a precise specification.

Any operation o at $P_{l,x}$ enforces its preconditions at $P_{l,x}$:

$$o_x^l \Rightarrow \bigwedge_{f \in pre(o)^+} f_x^l \wedge \bigwedge_{f \in pre(o)^-} \neg f_x^l \tag{6.5}$$

Similarly, any action a enforces its effects at $P_{l,x+1}$:

$$a_x^l \Rightarrow \bigwedge_{f \in \mathit{eff}(a)^+} f_{x+1}^l \wedge \bigwedge_{f \in \mathit{eff}(a)^-} \neg f_{x+1}^l \tag{6.6}$$

Later on (Section 6.4.1.b) we introduce so-called *frame axioms* which complement the correct enforcement of action effects. Also note that each f in the above formulas may contain pseudo-constants. For now we treat such pseudo-facts as we treat ground facts and encode each of them with a Boolean variable.

To logically connect subsequent layers with each another we use the following clauses. First, a ground fact f that holds at $P_{l,x}$ must be logically equivalent to the same ground fact at the first successor position $P_{l+1,s_l(x)}$ of $P_{l,x}$:

$$f_x^l \Leftrightarrow f_{s_l(x)}^{l+1} \tag{6.7}$$

Note that in practice these clauses do not occur in our encoding; instead we use exactly the same Boolean variable for f_x^l and $f_{s_l(x)}^{l+1}$ in the first place.

Next, we describe the sufficient and necessary conditions for operations at a new layer: When a parent o is active at $P_{l,x}$, then for each offset z one of its children o' at offset z must be active at $P_{l+1,s_l(x)+z}$.

$$\forall z \in \{0, \dots, e_{l,x} - 1\} : o_x^l \Rightarrow \bigvee_{o' \in children(o,z)} (o')_{s_l(x)+z}^{l+1} \tag{6.8}$$

Remember that $children(o,z)$ is well-defined for all such z: For each action a and $z > 0$, $children(a,z) = \{\varepsilon\}$, and for each reduction r and $z \geq |subtasks(r)|$, $children(r,z) = \{\varepsilon\}$.

Schreiber et al. [Sch+19b] found that it is beneficial for SAT solving performance to also redundantly enforce the opposite direction: When a child o' is active at $P_{l+1,s_l(x)+z}$, then any of its possible parents o must be active at $P_{l,x}$.

$$\forall z \in \{0, \dots, e_{l,x} - 1\} : (o')_{s_l(x)+z}^{l+1} \Rightarrow \bigvee_{o \mid o' \in children(o,z)} o_x^l \tag{6.9}$$

To conclude the clause sets which TREE-REX featured in a similar form [Sch+19b], we enforce the currently deepest layer $L_{l'}$ to be fully primitive. This implies a fully expanded hierarchical task network. The following unit clauses are added as *assumptions*, i.e., they are considered axioms by the SAT solver for the next solving attempt and discarded afterwards.

$$\forall x \in \{0, \dots, |L_{l'}| - 1\} \ : \ \mathit{prim}_x^{l'} \tag{6.10}$$

6.4.1.b Pseudo-Constants and Pseudo-Facts

Next we define the semantics of pseudo-constants and their enclosing structures.

For each pseudo-constant κ introduced by some operation o, κ must be substituted with at most one constant from its possible domain, $\mathit{dom}(\kappa)$, and if o is active then exactly one such substitution must hold:

$$\bigwedge_{c_1 \neq c_2 \in \mathit{dom}(\kappa)} \neg[\kappa/c_1] \vee \neg[\kappa/c_2] \tag{6.11}$$

$$o_x^l \Rightarrow \bigvee_{c \in \mathit{dom}(\kappa)} [\kappa/c] \tag{6.12}$$

As in Eq. 6.4, we employ an asymptotically better encoding instead of Eq. 6.11 if κ has at least $n = 50$ substitutions.

Consider a pseudo-fact f_p with pseudo-constants $\kappa_1, \dots, \kappa_k$ $(k \geq 1)$. Assume that substituting each such pseudo-constant κ_i with a particular constant $c_i \in \mathit{dom}(\kappa_i)$ yields ground fact f. Then we define:

$$\big([\kappa_1/c_1] \wedge [\kappa_2/c_2] \wedge \dots \wedge [\kappa_k/c_k]\big) \Rightarrow \big((f_p)_x^l \Leftrightarrow f_x^l\big) \tag{6.13}$$

In words, we enforce a pseudo-fact to be equivalent to the ground fact it corresponds to when performing particular substitutions.

In most automated planning encodings, so-called *frame axioms* logically specify the *necessary* conditions for a fact change in between two adjacent time steps. In other words, frame axioms are necessary to prevent a SAT solver from arbitrarily changing the world state without executing a supporting action [KS92]. In our encoding, frame axioms are needed for ground facts only, as the pseudo-facts are well-defined by Eq. 6.13. We define a ground fact's support, $\mathit{supp}((\neg)f)$, as the set of actions which have f as a positive (negative) effect. Also, we define the fact's indirect support, $\mathit{isupp}((\neg)f)$, as the set of actions which are not in $\mathit{supp}((\neg)f)$ but which have some pseudo-fact f_p as a positive (negative) effect that can be syntactically unified with f. We add two types of clauses to specify frame axioms:

(i) If fact f changes its value, then either a reduction is responsible for the change (represented by $\neg\mathit{prim}_x^l$), or some action *directly* or *indirectly* supports this fact change.

$$\begin{aligned} f_x^l \wedge \neg f_{x+1}^l &\Rightarrow \neg\mathit{prim}_x^l \vee \bigvee_{a \in \mathit{supp}(\neg f)} a_x^l \vee \bigvee_{a \in \mathit{isupp}(\neg f)} a_x^l \\ \neg f_x^l \wedge f_{x+1}^l &\Rightarrow \neg\mathit{prim}_x^l \vee \bigvee_{a \in \mathit{supp}(f)} a_x^l \vee \bigvee_{a \in \mathit{isupp}(f)} a_x^l \end{aligned} \tag{6.14}$$

(ii) If fact f changes its value and some action $a \in \mathit{isupp}((\neg)f)$ is applied, then some set of substitutions must be active which unifies an effect f_p of a with f.

$$f_x^l \wedge \neg f_{x+1}^l \wedge a_x^l \;\Rightarrow\; \bigvee_{\substack{f_p \in \mathit{eff}(a)^-,\\ f_p[\kappa_1/c_1]\ldots[\kappa_k/c_k]=f}} \Big(\bigwedge_{i=1}^{k} [\kappa_i/c_i] \Big) \tag{6.15}$$

$$\neg f_x^l \wedge f_{x+1}^l \wedge a_x^l \;\Rightarrow\; \bigvee_{\substack{f_p \in \mathit{eff}(a)^+,\\ f_p[\kappa_1/c_1]\ldots[\kappa_k/c_k]=f}} \Big(\bigwedge_{i=1}^{k} [\kappa_i/c_i] \Big)$$

Frame axioms (ii) in the above rule are not in Conjunctive Normal Form (CNF): We require a transformation of *Disjunctive Normal Form* (DNF) into CNF whenever a features multiple effects which can be unified to f. Considering that the arity of predicates is commonly very small in planning domains (see Table 9.3, Section F), we just arrange the substitution constraints as a tree of Boolean literals in a heuristically chosen order and then encode the branches as CNF. No additional variables are required. We refer to the original publication [Sch21d] for details.

6.4.1.c Argument Type Restrictions

Assume in the `Factories` domain that we can transport resources not only with trucks but with airplanes as well, and that trucks and airplanes share a common "vehicle" type τ. While some operation $\mathit{deliver}(r, \lambda, l, \nu)$ to transport resource r from λ to l may introduce ν as a pseudo-constant of type τ, some child operations $\mathit{drive\_to}(\nu, \cdot)$ and $\mathit{fly\_to}(\nu, \cdot)$ may force ν to be of the type "truck" or "airplane". More generally, it can happen that a child restricts the valid domain τ of a pseudo-constant from an earlier layer to some $\tau' \subset \tau$. We explicitly deal with such type restrictions by either forbidding all illegal substitutions or enforcing one of the valid substitutions:

$$\forall c \in \tau \setminus \tau' \;:\; o_x^l \Rightarrow \neg[\kappa/c] \tag{6.16}$$

$$o_x^l \Rightarrow \bigvee_{c \in \tau'} [\kappa/c] \tag{6.17}$$

We dynamically decide on whether to encode Eq. 6.16 or Eq. 6.17 based on which of the sets induces a smaller overall number of Boolean literals. Empirically we found that oftentimes one of the two sets is very small.

6.4.1.d Actions with Contradictory Effects

Planning descriptions allow an operator o to have both f and $\neg f$ as an effect. Seeming contradictory at first glance, it is indeed consistent with the common semantics of applying an action in automated planning: First all negative effects are deleted from the state and then all positive effects are added to the state [GNT04, Def. 2.7]. For example, this allows to consistently instantiate an operator "*go from x to y*" for the case $x = y$, first deleting and then re-adding the current location x.

In encodings generated from ground representations, each action can be trivially preprocessed by deleting each effect $\neg f$ for which f is also an effect. In our lifted encoding, whether an action contains contradictory effects generally depends on which substitutions are applied. Our encoding specified so far may then logically imply both f^l_x and $\neg f^l_x$ at the same time, rendering the formula unsatisfiable.

We propose the following solution to this problem. Each positive effect of action a is encoded normally. For each negative effect $f \in \mathit{eff}(a)^-$ we collect all positive effects $f' \in \mathit{eff}(a)^+$ such that f and f' share the same predicate. We then compute the set Σ of all substitution sets which unify f with such an f'. We distinguish three cases:

(i) If $\Sigma = \varnothing$, then there are no conflicting positive effects for negative effect $\neg f$ and the effect will be encoded normally as in Eq. 6.6.
(ii) If $\varnothing \in \Sigma$, i.e., f is already unified with some $f' \in \mathit{eff}(a)^+$ without applying any substitution, then f and f' are syntactically equal: The negative effect is discarded because it is always overridden by the positive effect.
(iii) Otherwise, $\Sigma := \{\Sigma_1, \dots, \Sigma_m\}$ where each Σ_i unifies some $f' \in \mathit{eff}(a)^+$ with f. We enforce that either the negative effect holds or some Σ_i is active:

$$o^l_x \Rightarrow \neg f^l_{x+1} \vee \bigvee_{i=1}^{m} \bigwedge_{[\kappa/\alpha] \in \Sigma_i} [\kappa/\alpha] \tag{6.18}$$

Let us examine the above symbols of the form $[\kappa/\alpha]$. Whenever one effect's argument κ is a pseudo-constant while the other's α is a constant, $[\kappa/\alpha]$ is a substitution variable. Also, at least one of κ and α must be a pseudo-constant: For constants $c \neq c'$, substitutions of the form $[c/c']$ are invalid and substitutions of the form $[c/c]$ are redundant and hence omitted. What remains is the special case of unifying a pair of pseudo-constants, i.e., $\kappa' := \alpha$ is a pseudo-constant as well. For each such case we introduce a new Boolean variable and give it the meaning: "κ and κ' are equal."

To have a variable $[\kappa/\kappa']$ assume this meaning, we introduce additional clauses: First the intersection of both domains, $I = \mathit{dom}(\kappa) \cap \mathit{dom}(\kappa')$, and the respective differences, $D := \mathit{dom}(\kappa) \setminus I$ and $D' := \mathit{dom}(\kappa') \setminus I$, are computed. If I is empty, then the pseudo-constants cannot be equal: $[\kappa/\kappa']$ is `false`. Otherwise, we encode clauses to guarantee that $[\kappa/\kappa']$ holds if and only if both pseudo-constants are substituted with the same constant:

$$\begin{aligned}
\forall c \in I &: [\kappa/\kappa'] &&\Rightarrow ([\kappa/c] \Leftrightarrow [\kappa'/c]) \\
\forall c \in I &: ([\kappa/c] \wedge [\kappa'/c]) &&\Rightarrow [\kappa/\kappa'] \\
\forall c \in D &: [\kappa/c] &&\Rightarrow \neg[\kappa/\kappa'] \\
\forall c \in D' &: [\kappa'/c] &&\Rightarrow \neg[\kappa/\kappa']
\end{aligned} \tag{6.19}$$

We encode Eq. 6.18 with a DNF-to-CNF transformation as for Eq. 6.15 and encode Eq. 6.19 whenever a new equality variable emerges which did not occur before.

6.4.1.e Dominated Operations

Last but not least, we turn to the situation where some operation o dominates another operation o' as described in Section 6.3.3. Whenever o becomes active as a child of one of the parents of o' (but no parent of o), we restrict the pseudo-constants of o to be equivalent to those of o'. Again, we make use of variables $[\kappa/\kappa']$ as defined above.

$$\forall o_p \in P_{l,x} \mid o' \in \mathit{children}(o_p, z),\ o \notin \mathit{children}(o_p, z):\ (o_p)^l_x \wedge o^{l+1}_{s_l(x)+z} \Rightarrow \bigwedge_{\kappa\in o, \kappa'\in o'} [\kappa/\kappa'] \tag{6.20}$$

This concludes the set of clauses which are required for the correctness of our approach in conjunction with the techniques described in Section 6.3.

6.4.2 Optimizations

In the following, we describe some optimizations for our encoding. Some minor techniques are omitted and can be found in the full journal article [Sch21d].

So far, we encoded all facts which appear as a precondition or as an effect at some position $P_{l,x}$. However, our reachability analysis allows us to identify invariant facts, i.e., (ground) facts which are definitely true or definitely false at a certain position (see Section 6.3.2). We avoid to introduce Boolean variables for invariant facts. Instead of encoding each relevant fact at the zeroth position, we adjust Eq. 6.2 as follows:

$$\begin{aligned} &\forall f \in F_l \mid f \text{ is invariantly true at } P_{l,x} \text{ but not at } P_{l,x+1}:\ \ f^l_x \\ &\forall f \in F_l \mid f \text{ is invariantly false at } P_{l,x} \text{ but not at } P_{l,x+1}:\ \ \neg f^l_x \end{aligned} \tag{6.21}$$

In words, at each layer L_l we *delay* the encoding and initialization of fact f as a Boolean variable until the last position where f is invariant. Note that all facts are invariant at the zeroth position due to s_I. Replacing Eq. 6.2 with Eq. 6.21 allows us to skip many trivial frame axioms which preserve the polarity of an unchanging fact (see Eq. 6.14 with empty supports) and completely omits the encoding of globally invariant facts. Whenever f is invariant and hence not encoded, we consequently do not encode the equivalence of f to any present pseudo-fact f_p (Eq. 6.13). However, to preserve correctness, we may need to introduce some other constraints instead.

First, if an operation (action) at $P_{l,x}$ has a ground precondition (effect) that is invariantly true at $P_{l,x}$ ($P_{l,x+1}$), then we can simply omit Eq. 6.5 (Eq. 6.6) for this particular constraint. Note that effects are never invariantly false due to construction, and operations with invariantly false preconditions are pruned during instantiation.

Secondly, assume that operation o has k preconditions with pseudo-constants. For each such precondition f, each ground fact resulting from f is either invariantly false or invariantly true or not invariant. All non-invariant facts are handled as before by linking them with a pseudo-fact (Eq. 6.13). In addition, we must make sure that no substitution is applied which transforms f into an invariantly false precondition. Generally, for each $1 \le i \le k$, there are n_i sets $\Pi_{i1}, \dots, \Pi_{in_i}$ of substitutions rendering precondition i

invariantly false, and m_i remaining sets $\Sigma_{i1}, \dots, \Sigma_{im_i}$ for which precondition i may hold (invariantly or not). For each precondition, we either enforce a valid substitution set (Eq. 6.22) or that no invalid substitution set can hold (Eq. 6.23).

$$\forall i \in \{1, \dots, k\} \ : \ o_x^l \Rightarrow \bigvee_{j=1}^{m_i} \bigwedge_{[c/\kappa] \in \Sigma_{ij}} [c/\kappa] \tag{6.22}$$

$$\forall i \in \{1, \dots, k\} \ \forall j \in \{1, \dots, n_i\} \ : \ o_x^l \Rightarrow \bigvee_{[c/\kappa] \in \Pi_{ij}} \neg [c/\kappa] \tag{6.23}$$

As in Eq. 6.16–6.17, we encode the smaller of these sets. Eq. 6.22 is realized with a DNF-to-CNF transformation.

As a special case, if *all* ground facts which may result from a precondition are invariant, we can omit the pseudo-fact corresponding to the precondition (Eq. 6.13). This situation occurs frequently in practice because most planning domains contain so-called *rigid predicates* [GNT04, p. 43] which are not featured in any action effect. In our `Factories` example (Fig. 6.7), some rigid predicates are $A{\leftrightarrow}B$ (there is a road between A and B) and $F_1{\Rightarrow}R$ (factory F_1 can produce resource R).

We now turn to effects with pseudo-constants. Each ground fact that can result from such an effect is considered a possible fact change by our reachability analysis. As such, an effect is never invariantly false. It can happen, however, that some substitutions which turn the effect into a ground fact are known to be invalid because a precondition of the same operation becomes invariantly false for these substitutions. We omit Eq. 6.13 for each such substitution; the according substitutions are already prohibited by the precondition's encoding. If all ground facts resulting from the effect are either omitted this way or invariantly true, we do not encode an according pseudo-fact.

6.4.3 Decoding a Plan

We now explain the process of decoding a classical and hierarchical solution from a satisfying assignment to our encoding found by a SAT solver. Assume that the encoding $\mathcal{L}_{l'}$ of layers L_0 through $L_{l'}$ is satisfiable and a satisfying assignment $\mathcal{A}$ to all variables is available. We define that an operation o is *active* at $P_{l,x}$ iff $\mathcal{A}(o_x^{l'}) = \texttt{true}$. Similarly, a substitution $[\kappa/c]$ is *active* at $P_{l,x}$ iff $\mathcal{A}([\kappa/c]) = \texttt{true}$.

We first decode a plan π (see Def. 6.1) from $\mathcal{A}$: We begin with an empty plan $\pi := \langle\rangle$. For each position index $x = 0, \dots, |L_{l'}| - 1$ we find an active action $a \in P_{l',x}$. If a is an ε-action, we discard it. Otherwise, if a is ground, we append it to π. Otherwise, for each pseudo-constant κ of a we find an active substitution $[\kappa/c]$ and substitute all occurrences of κ in action a with c. The resulting ground action $\tilde{a}$ is appended to π.

To obtain the hierarchical solution leading to π (see Def. 6.2) we begin with a graph H without any edges and a single node $(r_0, 0, 0)$ which represents the initial reduction r_0 at the zeroth position of layer L_0. We traverse all layers in the order of their instantiation. For each position $P_{l+1, s_l(x)+z}$ where $l + 1 > 0$, we first examine the parent node (o, l, x) in H. If o is an action, then we ignore the child position and continue. Otherwise we find an active child operation o' at $P_{l+1, s_l(x)+z}$.

We add a new node $(\tilde{o}', l+1, s_l(x)+z)$ where $\tilde{o}'$ is the ground representation of o' (as explained above for actions $\tilde{a}$), and we add edge $((o,l,x),(\tilde{o}', l+1, s_l(x)+z))$ to H.

In our `Factories` example in Fig. 6.7 we obtain π by traversing L_4 from left to right and collecting all active actions which are not ε-actions. For each pseudo-constant in each action, we apply the according highlighted equivalence in the top left corner of Fig. 6.7. Reduction $goto\_noop(\theta, C)$ at $P_{4,6}$ is treated as an action but is omitted from π due to our optimizations. Fig. 6.7 also contains a representation of the hierarchical solution H: Each position with a highlighted operation corresponds to a node in H except for positions with repeated actions from an earlier layer ($P_{3,1}$, $P_{4,1}$, $P_{2,3}$, etc.) and positions where an ε-action is active.

6.4.4 Correctness

The following theorem establishes the correctness of our encoding:

Theorem 6.4

Let (π, H) be the decoded (classical and hierarchical) solution for the Lilotane *encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Then π is a valid solution for Π and H is a valid hierarchical solution for Π.*

Proof. See Appendix D.3. □

A corollary of this theorem is the correctness of the previous Tree-REX encoding [Sch+19b] where no pseudo-constants are introduced, only ground operations are added, and all relevant facts are encoded at position zero of each layer.

While we do not provide a proof of completeness for our encoding, note that a similar chain of arguments as in our correctness proof can be made to show that whenever a problem Π has a solution (π, H) at depth l', our encoding will be satisfiable at layer l' and enable us to extract a valid solution from a satisfying assignment.

6.4.5 Complexity

In the following, we assess the complexity of the Lilotane encoding, providing the worst case asymptotic number of variables and clauses in the encoding of $\mathcal{L}_{l'}(\Pi)$.

Consider a worst-case hierarchy which grows exponentially both in the size of its layers and the encoded operations at each position. Let X be the maximum number of subtasks of any method and let B be the maximum number of methods with different signature names which achieve the same task. Given the initial layer size $|L_0| = 1$, the size of layer l' is in $\mathcal{O}(X^{l'})$. Since the encoded operations per position can multiply by a factor of B per layer, the total number R of encoded operations is in $\mathcal{O}(X^{l'}B^{l'})$. Let V be the maximum arity (i.e., the number of arguments) of any operation, and let U be the maximum number of free arguments of any method. Let C be the number of constants, and let P (E) be the maximum number of preconditions (effects) of any operation. Then the number of encoded variables is in

$$\mathcal{O}\Big(X^{l'}\big(F + B^{l'}(UC + P + E + V^2)\big)\Big) \tag{6.24}$$

as we derive in Appendix E.1. Essentially, at each position we need to encode each fact ($X^{l'}F$), and for each operation we need to encode each pseudo-constant ($X^{l'}B^{l'}UC$), each pseudo-fact emerging from a precondition or effect ($X^{l'}B^{l'}(P+E)$), and possibly an equality variable for each pair of pseudo-constants in the action ($X^{l'}B^{l'}V^2$).

Let us compare this to the previous Tree-REX approach [Sch+19b] for a worst-case result of grounding: When expanding a position in the hierarchy, each operation at some position can lead not to B, but instead to $B \cdot C^U$ new operations for each subtask because we fully ground all operations. This creates $\mathcal{O}(X^{l'}(BC^U)^{l'})$ operations in total and overall leads to $\mathcal{O}(X^{l'}(F + (BC^U)^{l'}))$ variables. However, also note that for Tree-REX the number of operations per position cannot grow indefinitely but is bounded by $\mathcal{O}((|M| + |O|)C^V)$, i.e., the number of instantiated operations, while for Lilotane the number of operations at each position is unbounded: At each layer, new pseudo-constants and thus "new" operations may be introduced. We counteract this with shared pseudo-constants and dominating operations (Section 6.3.3).

Next, let us consider the number of encoded clauses. We define Y as the maximum predicate arity. We arrive at the following asymptotic complexity:

$$\mathcal{O}\Big(X^{l'}B^{l'}\big(l'\log B + C(U\log C + V^2) + F(P + YE) + YE^2\big)\Big) \tag{6.25}$$

For the derivation we again refer to Appendix E.2. If we assume V, U, and Y to be constants, then we arrive at

$$\mathcal{O}\Big(X^{l'}B^{l'}\big(l'\log B + C\log C + F(P + E) + E^2\big)\Big) \tag{6.26}$$

clauses. Terms $l'\log B$ and $C\log C$ are caused by at-most-one constraints over the operations at each position and over the substitutions for a pseudo-constant. Term $F(P+E)$ is implied by (among others) the semantics of pseudo-facts and term E^2 originates from the encoding of contradictory action effects.

For Tree-REX, applying our complexity model under above assumptions yields

$$\mathcal{O}\Big(X^{l'}\big(T \cdot (\log T + P + E) + F\big)\Big) \tag{6.27}$$

clauses (see [Sch+19b], Complexity) where $T := \min\{B^{l'}C^{Ul'}, (|M| + |O|)C^V\}$.

The advantages of Tree-REX are that only a constant number of clauses is added for each fact, each precondition and each effect at each position and that, again, there is an upper bound on the number of operations at each position. By contrast, while Lilotane may encode more clauses per operation due to its more complex handling of facts, its much smaller initial branching factor leads to fewer operations by a factor of $C^{Ul'}$ as long as the upper bound for T is not reached.

The exponential complexity of both encodings, which compile 2-EXPTIME-complete TOHTN planning to NP-complete SAT (see Section 6.2.1.b), is presumably unavoidable. However, the number of clauses and variables encoded by LILOTANE is exponential only in l' while for TREE-REX the encoding size is exponential both in l' and in either Ul' or V. We conclude that the LILOTANE encoding compared to TREE-REX is a new and to some degree orthogonal approach which focuses on reducing the number of encoded operations at the cost of a more complex logic related to the problem's state space features. Our experiments in Section 6.6 will complement this analysis with some empirical practical insights.

6.5 Plan Improvement

The length of a given plan $\pi = \langle a_0, \dots, a_{k-1} \rangle$ is given by $k = |\pi|$. We are interested in finding as short plans as possible because in a real-world application each action will require some effort in order to be executed: In most cases, shorter plans are more efficient and executed faster. However, our base algorithm generally produces suboptimal plans because it can introduce ε-actions which do not contribute to the plan length. In order to minimize $|\pi|$ we can maximize the number of active ε-actions at the layer l where a plan was found. Such an optimization will yield an optimal plan at layer l, which we call a *depth-optimal plan*, but not necessarily a globally optimal plan: A different choice of methods which require a larger depth to be fully expanded may be able to induce a higher number of ε-actions. Hence, we may find an even shorter plan by admitting a deeper hierarchy [BHB19b]. In our plan improvement, we construct a depth-optimal plan but no globally optimal plan.

Let us reiterate the plan improvement approach of TREE-REX [Sch+19b]:

(i) After finding an initial plan π_0 at layer $L_{l'}$, all positions at $L_{l'}$ are permanently enforced to be primitive by adding Eq. 6.10 as unit clauses instead of assumptions.

(ii) We encode further variables and clauses to count the length of a plan in such a way that specific assumptions can restrict the plan length for a single solver call.

(iii) Beginning with $i = 0$, we add assumptions to forbid any plan length $k \geq |\pi_i|$ and call the SAT solver again. In case of satisfiability, we decode the new plan π_{i+1}, count its new length $|\pi_{i+1}| < |\pi_i|$, and repeat (iii) for incremented i. In case of unsatisfiability we return π_i which proved to be depth-optimal.

This procedure belongs to a broad class of optimization approaches which feature the decision problem "Is there a solution of cost $\leq k$?" for a sequence of different k. This class of approaches has been studied by Rintanen [Rin04] for scheduling SAT-based planning, was generalized by Streeter and Smith [SS07], and has been used in SAT-based HTN planning in different contexts [Sch18; BHB19b]. The above strategy of monotonically decreasing k is appealing because it produces a series of constructive SAT results concluded by a single UNSAT result. As such, an improved plan can be decoded from every intermediate result. In addition, a plan of length $k' < k$ may be found which allows to skip tests for the intermediate values.

We adapted and improved the Tree-REX plan improvement for Lilotane. More specifically, we encode a mechanism which counts the number of positions where a normal action, i.e., an action other than a_ε, is active. We then successively add constraints of the form $|\pi_i| \neq k$ as *unit clauses* rather than assumptions: SAT solvers can perform more simplification once a unit constraint is known to be permanent [NR12; FBS19a]. We refer to our article [Sch21d] for details on the encoding.

6.6 Evaluation

In the following, we discuss an experimental evaluation of Lilotane. Our software and experimental data are available online (see Appendix A).

6.6.1 Implementation

We have implemented our approach in C++17. We use `pandaPIparser` [Beh+20] for parsing and performing light preprocessing of input files in the HDDL (Hierarchical Domain Description Language) format [Höl+20a]. We used the interface called IPASIR (Section 2.2.6.a) to link our software with any SAT solver supporting this interface. We linked Lilotane with the popular incremental solver Glucose [AS09] for all evaluations in this chapter.[5]

6.6.2 Lilotane as a SAT-Based HTN Planner

First and foremost, we compare our planner to previous SAT-based HTN planners.[6] We included the up-to-date version of Lilotane; a quality-aware variant LilotaneQ which finds a depth-optimal plan at the layer where the initial plan was found; Tree-REX without plan improvement; TOHTN planner PANDA-totSAT [BHB18]; and optimal HTN planner PANDA-SAT-OPT [BHB19b]. We use PANDA with its default SAT solver CryptoMiniSAT [SNC09] and use the configuration `sat-exists-forbidden-implication` and search strategy `BIN` for the optimal variant.

We used all twelve HTN benchmark domains for which equivalent formulations for all planners' input formats exist (see [Sch+19b]). Due to technical limitations we cannot compare the plans output by Tree-REX and by Lilotane in a fair manner, which is why we did not include a plan improving variant of Tree-REX. We fixed some notable issues with the grounding backend of Tree-REX to ensure a fair comparison; see our article [Sch21d] for details. We set a timeout of 300 s and a memory limit of 8 GB. We used a desktop PC running Ubuntu 18.04 with a quad-core Intel i7-6700 processor clocked at 3.40GHz with 32GB of RAM.

[5]We later observed moderately improved performance [Sch21b] using CaDiCaL [Bie17] instead of Glucose. We use this improved setup in Chapter 7.

[6]Note that the state of the art in TOHTN planning has progressed since we performed this experimental evaluation in 2020/21. In Section 8.2 we briefly discuss Lilotane's impact and current role regarding the state of the art.

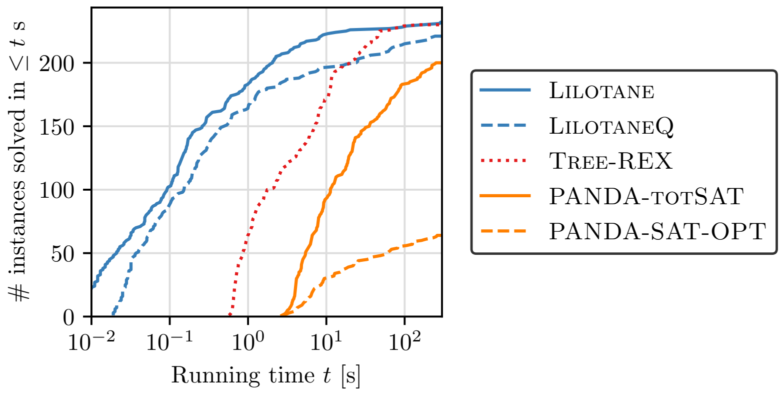


Figure 6.9: Running times of PANDA-SAT, Tree-REX, and Lilotane. Note the logarithmic scale along the x axis.

6.6.2.a Overview

Fig. 6.9 provides an overview of running times. PANDA-SAT-OPT found an optimal plan on 64 out of 242 instances and found *some* plan on 193 instances (not pictured). PANDA-totSAT solved 200 instances, Tree-REX solved 230 instances and Lilotane solved 232 instances. The quality-aware variant LilotaneQ completed plan improvement on 221 instances, including 189 instances for which PANDA-SAT-OPT found some plan. Among these 189 instances, LilotaneQ found a shorter plan in 89 cases and matched the plan length of PANDA-SAT-OPT otherwise. In particular, LilotaneQ found an optimal plan wherever PANDA found an optimal plan.

More detailed comparisons are shown in Fig. 6.10. Each point (x, y) corresponds to an instance. For points along the diagonal $y = x$ both approaches performed equally well. For points on the i-th diagonal above (below) the central diagonal, Lilotane performed better (worse) by i orders of magnitude.

In the left graphs, raw solving times are compared. Both PANDA and Tree-REX incur a considerable amount of overhead which leads to a large relative difference in running times at the bottom left, i.e., for easier instances. This gap is much more pronounced for PANDA which has a quite slow preprocessing [Beh+20].

218/242 problems (90.0%) have been resolved by both Lilotane and Tree-REX, and 197 problems (81.4%) have been resolved by both Lilotane and PANDA. Among the instances solved by both, on 98.2% (68.4% / 7.3%) Lilotane outperformed Tree-REX (by more than one / two orders of magnitude). On 99.5% (97.5% / 59.9% / 5.1%) Lilotane outperformed PANDA (by more than one / two / three orders of magnitude). `Satellite` is the only domain where Lilotane is slower than Tree-REX for multiple instances. This domain heavily features recursive subtask relationships which can be simplified by the grounding of Tree-REX.

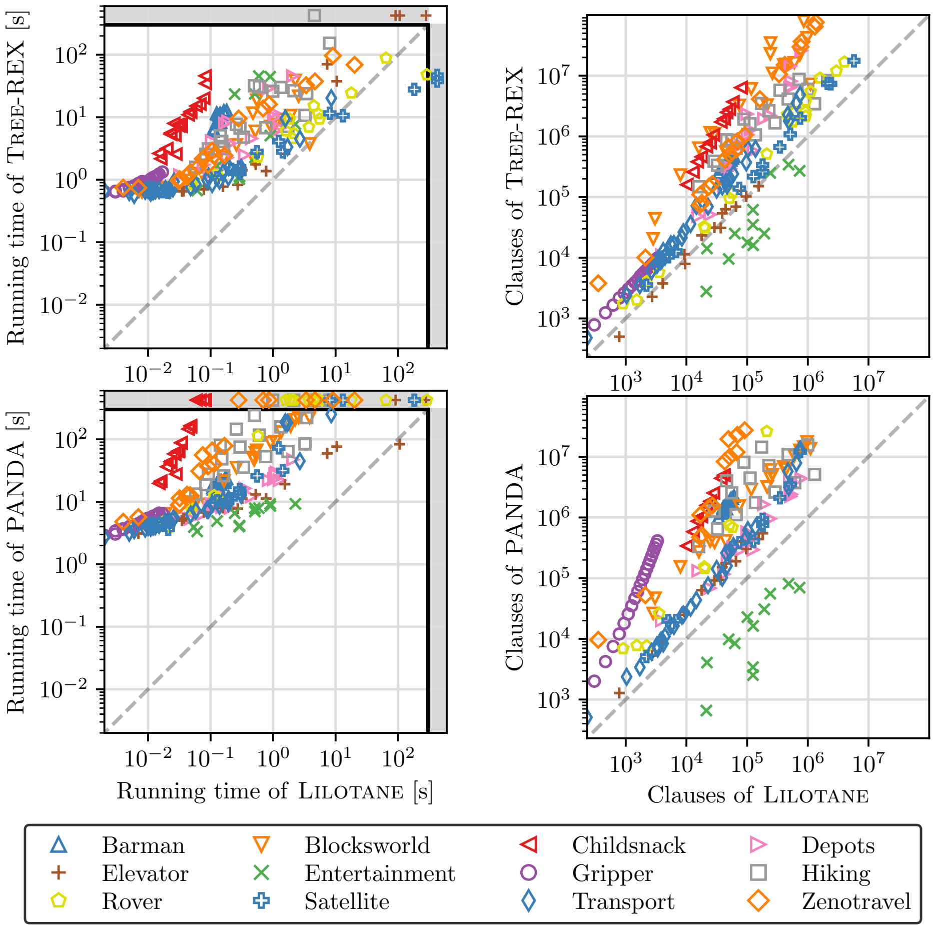


Figure 6.10: Log-log run times (left) and encoded clauses (right) of Lilotane vs. Tree-REX (top) and vs. PANDA-totSAT (bottom).

6.6.2.b Encoding Properties

In the right graphs in Fig. 6.10, the number of encoded clauses is compared, providing more insight into the relative quality of our encoding while implementation-dependent performance differences are excluded. For some domains, the respective set of points resembles a line of slope $m > 1$ in log-log scale, which indicates a polynomial factor in encoding size as the problem size increases. This effect is visible for the domains `Childsnack`, `Depots` and `Zenotravel` in comparison to Tree-REX and for `Childsnack`, `Transport` and `Gripper` in comparison to PANDA, which confirms our claim that grounding can lead to a severe blowup in problem size (Section 6.2.2).

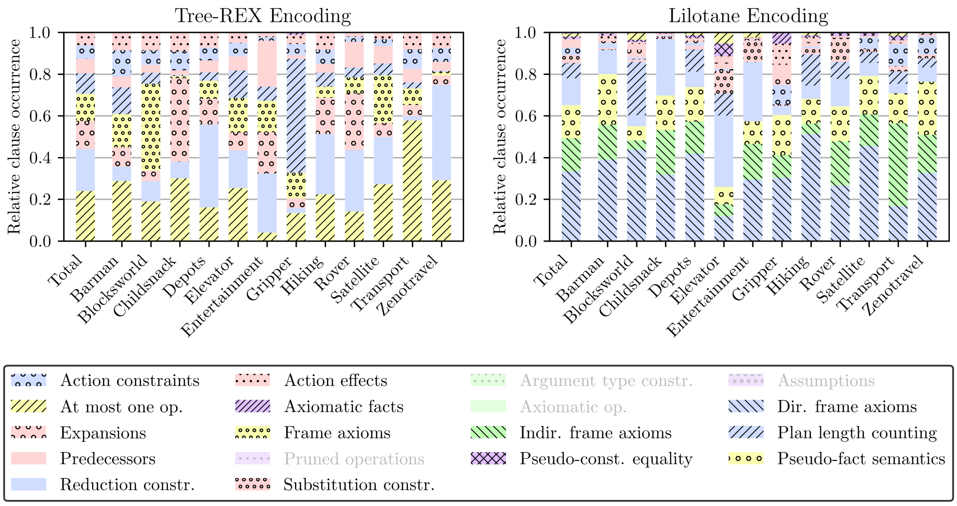


Figure 6.11: Distribution over different clause categories per domain in the Tree-REX and Lilotane encodings.

For instance, the methods decomposing each initial task of `Childsnack` have four free arguments. Each of these arguments has $\mathcal{O}(n)$ possible values where n defines the problem difficulty. For each initial task, Tree-REX instantiates $\mathcal{O}(n^4)$ reductions whereas Lilotane instantiates $\mathcal{O}(1)$ reductions. Both instantiate $\mathcal{O}(n)$ facts.

We do not observe any exponential differences in encoding size. In the `Childsnack` domain in particular, the problem hierarchy has a constant depth of two, hence the described blowup in the number of child operations occurs only once before the problem is solved. More generally speaking, most domains do not have a hierarchy which expands indefinitely with respect to the density of operations (as assumed in our worst case analysis in Section 5.5) but instead become quite simple after few layers.

`Entertainment` is the only domain for which the Lilotane encoding is consistently larger—by up a factor of 50. For these instances, the grounding procedures of Tree-REX and PANDA prune large parts of search space before encoding them. Our algorithm prunes these parts retroactively when they turn out to be impossible to achieve, after the clauses were already added. The shown results indicate that our lifted approach may be at a disadvantage in cases where the ground problem becomes substantially smaller and simpler than the lifted representation.

We visualized the relative occurrence of different kinds of clauses in Fig. 6.11. In the Tree-REX encoding, substantially more operations are encoded due to full grounding, hence at-most-one constraints over operations are the most expensive category of clauses followed by reduction constraints (i.e. non-primitiveness and preconditions) and expansion constraints. In the Lilotane encoding, it is the frame axioms which consistently make up large parts of the encoding, which is why we split them into direct frame axioms (Eq. 6.14) and indirect frame axioms (Eq. 6.15). Pseudo-fact semantics are the next most costly clauses, followed only then by reduction constraints (which also include constraints of substitutions, Eq. 6.22–6.23). Simply put, for Tree-REX the encoding of operations is the main bottleneck and for Lilotane the encoding of (pseudo-)facts is the main bottleneck. For `Entertainment`, while no single clause category is alone responsible for Lilotane's much larger encodings, we do see a relatively high ratio of reduction constraints. Tree-REX encodes `Entertainment` problems with the lowest ratio of at-most-one constraints throughout all domains, implying a vastly simplified task network. This confirms that Lilotane encodes a much larger problem as it has no access to valuable information gained from grounding.

We provide some further material in Appendix F. Notable insights include that Lilotane produces longer clauses than PANDA-totSAT (3.15 vs. 2.5 literals on average), only uses a fraction of the memory required by the other planners, and is able to spend more than 85% of its time on SAT solving (<40% for the other planners).

6.6.3 International Planning Competition 2020

We now discuss the International Planning Competition (IPC) 2020 [BHB21] and the performance of Lilotane in this competitive event.

The IPC was run on an exceptionally large and difficult set of benchmarks for HTN planning (see Table 9.3, Appendix F). The benchmarks have confirmed our assumption that the maximum arity of predicates is a small constant (four in the IPC's `Entertainment` domain and at most three everywhere else) and is mostly smaller but never larger than the maximum arity of actions and reductions.

Planners were rated according to the following metric: If a planner solves an instance within one second, a score of 1 is attributed. If a planner solves an instance within $1 < t \leq T$ seconds (where $T = 30$ min is the time limit), a score of $1 - \log(t)/\log(T)$ is attributed [BHB21]. This so-called *agile metric* favors planners which find plans very quickly over slower but more robust planners: If a planner solves five instances in 2 seconds while not solving five other instances, it is attributed a score of around $5 \cdot 0.91 = 4.55$. If a planner solves each of the ten instances in two minutes, it is attributed a score of around $10 \cdot 0.36 = 3.6$. Furthermore, plan quality is ignored.

The competition consisted of two tracks: Total Order and Partial Order. Six planners were submitted to the Total Order track while only three planners were submitted to the Partial Order track (one of which was disqualified). Among these six planners are a preliminary version of Lilotane, the lifted progression search planner HyperTensioN, the ground progression search planner PDDL4J in a Total Order and a Partial Order version, the lifted progression search planner SIADEX, and the plan-space planner pyHiPOP. All are described in the IPC proceedings [BHB21].

6.6.3.a Results

Tab. 6.1 shows the results of the IPC. In addition to the IPC score explained above, we include the *coverage* metric which counts the number of solved instances.

	HyperT.		Lilotane		P4JTO		P4JPO		SIADEX		pyHiPOP	
Domain	NC	IPC	NC	IPC	NC	IPC	NC	IPC	NC	IPC	NC	IPC
AssemblyHierarchical	0.10	0.08	**0.17**	**0.12**	0.07	0.06	0.07	0.06	0.00	0.00	0.03	0.02
Barman-BDI	**1.00**	**1.00**	0.80	0.74	0.55	0.48	0.55	0.49	**1.00**	0.92	0.00	0.00
Blocksworld-GTOHP	0.53	0.43	**0.77**	**0.64**	0.53	0.41	0.57	0.43	0.47	0.35	0.03	0.01
Blocksworld-HPDDL	**1.00**	**0.89**	0.03	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Childsnack	**1.00**	**1.00**	0.97	0.87	0.70	0.46	0.70	0.47	0.73	0.50	0.00	0.00
Depots	**0.80**	**0.76**	**0.80**	0.73	0.77	0.57	0.77	0.60	0.73	0.70	0.00	0.00
Elevator-Learned	**1.00**	**1.00**	**1.00**	0.78	0.01	0.01	0.01	0.01	0.07	0.07	0.01	0.01
Entertainment	0.00	0.00	**0.42**	0.14	**0.42**	0.19	0.25	**0.27**	0.00	0.00	0.08	0.07
Factories-simple	0.15	0.14	**0.20**	**0.19**	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.01
Freecell-Learned	0.00	0.00	**0.15**	**0.05**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Hiking	**0.83**	**0.83**	0.73	0.60	0.57	0.32	0.50	0.39	0.00	0.00	0.00	0.00
Logistics-Learned	0.28	0.26	**0.55**	**0.32**	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Minecraft-Regular	**0.98**	**0.88**	0.54	0.35	0.39	0.32	0.39	0.32	0.59	0.33	0.00	0.00
Minecraft-Player	**0.25**	**0.25**	0.05	0.03	0.05	0.03	0.05	0.03	0.15	0.13	0.00	0.00
Monroe-Fully-Obs.	0.00	0.00	**1.00**	**0.78**	**1.00**	0.49	**1.00**	0.58	0.50	0.27	0.00	0.00
Monroe-Partially-Obs.	0.00	0.00	**1.00**	**0.73**	0.05	0.03	0.05	0.03	0.00	0.00	0.00	0.00
Multiarm-Blocksworld	**0.11**	**0.11**	0.05	0.03	0.00	0.00	0.00	0.00	0.01	0.01	0.00	0.00
Robot	**1.00**	**0.96**	0.55	0.52	0.30	0.27	0.30	0.27	0.00	0.00	0.05	0.05
Rover-GTOHP	**1.00**	**0.92**	0.77	0.55	**1.00**	0.60	0.87	0.65	**1.00**	0.77	0.20	0.14
Satellite-GTOHP	**1.00**	**1.00**	0.75	0.59	**1.00**	0.44	0.50	0.73	0.00	0.00	0.35	0.19
Snake	**1.00**	**1.00**	0.90	0.74	**1.00**	0.71	**1.00**	0.71	0.35	0.29	0.10	0.03
Towers	**0.85**	**0.77**	0.50	0.39	0.80	0.58	0.75	0.61	0.55	0.47	0.10	0.09
Transport	**1.00**	**1.00**	0.88	0.76	0.85	0.65	0.82	0.71	0.03	0.03	0.45	0.23
Woodworking	0.23	0.23	**1.00**	**0.98**	0.20	0.17	0.20	0.17	0.10	0.10	0.13	0.09
Total rel. score	14.12	**13.51**	**14.57**	11.60	10.25	7.47	9.35	6.36	6.29	4.93	1.60	0.94

Table 6.1: IPC results. NC = normalized coverage, IPC = agile runtime score (higher is better). Best scores per line are printed in bold for each metric.

As each run was repeated ten times, we count an instance as solved if it was solved in any of the runs. We normalized both kinds of scores per domain by the number of instances in the domain (e.g., a normalized coverage score of 0.6 means that 60% of all instances within the domain were solved).

Regarding IPC scores, which decided the official ranking, Lilotane scored second, behind HyperTensioN by a decent margin. All further competitors scored significantly lower. Notably, Lilotane outperformed ground approach PDDL4J on all but four domains (`Entertainment`, `Minecraft-Player`, `Rover`, and `Towers`). HyperTensioN scored best on 15/24 domains and Lilotane scored best on 8/24 domains, leaving only one domain where PDDL4J scored best. Lilotane's weakest domains are `Blocksworld-HPDDL`, `Minecraft-Player`, and `Multiarm-Blocksworld`. Each of these domains leads to deep and large task networks which favor progression search planners over planners such as Lilotane which are required to instantiate the entire hierarchy with all alternatives up to the layer where a plan can be found. By contrast, our planner excels on the domains `Monroe` [BA05; Höl+18] and `Woodworking` (manufacturing and processing tasks with many arguments per operator and method).

Lilotane solved three more instances (548) than HyperTensioN (545) and scored slightly better. Yet, Lilotane solved 14 of these instances only in some of the runs, while HyperTensioN solved each instance consistently with one exception. Overall, while the agile score benefits the very fast execution times of HyperTensioN, Lilotane performed similarly to HyperTensioN in terms of robustness and, unlike HyperTensioN, was able to solve some instance(s) on every single domain.

6.6.4 Follow-Up Evaluation

We now present our own evaluation based on IPC benchmarks. First, we improved Lilotane in various aspects after the submission deadline of the IPC.[7] Secondly, as solution quality did not matter in the IPC, we also want to evaluate the quality of our approach with and without plan improvement on a large set of benchmarks.

As PDDL4J, SIADEX, and pyHiPOP were mostly dominated regarding both IPC scores and coverage, we do not include them in the following evaluations. We do include the winner HyperTensioN [MMdS21] and different versions of Lilotane: (i) Prelilotane (the preliminary version submitted to the IPC), (ii) Lilotane (the up-to-date version without quality awareness), (iii) LilotaneQ (a quality-aware configuration which finds the depth-optimal plan at the layer where a plan is found), and (iv) LilotaneQ+ (a configuration which instantiates one extra layer after finding an initial plan and then finds the depth-optimal plan). For approaches (iii) and (iv), unfinished runs where *some* plan was output are considered unsolved.

The evaluations were conducted on an server with an AMD EPYC 7702P 64-Core processor (plus hyperthreading) clocked between 2.0 and 3.35 GHz with 1024 GB of DDR4 RAM, running Ubuntu 20.04. We executed up to 63 runs in parallel and set a time limit of 30 minutes and a memory limit of 8 GB as in the IPC.

6.6.4.a Overview

A first overview of the results is provided in Fig. 6.12. HyperTensioN solved 539 out of 892 instances (60.4%) and Prelilotane solved 529 instances (59.3%). The lower coverages compared to the IPC data can be explained by (a) different hardware and (b) the fact that we performed only one run for each competitor-instance combination. HyperTensioN retains its status being fastest on the majority of benchmarks. However, up-to-date Lilotane solved 558 instances (62.6%) making it more robust in the long run. LilotaneQ finished plan improvement on 523 instances and LilotaneQ+ finished on 496 instances. In other words, the quality-aware configurations of Lilotane found a depth-optimal plan at the first solvable layer on 93.7% of the solved instances and a depth-optimal plan at the subsequent layer on 88.9%. We provide some more details in Appendix F, Fig. 9.3–9.4 and Tab. 9.4.

[7]Note that we integrated each of these improvements before gaining access to the IPC benchmarks and only applied bugfixes thereafter. No fine-tuning with respect to the benchmarks was done.

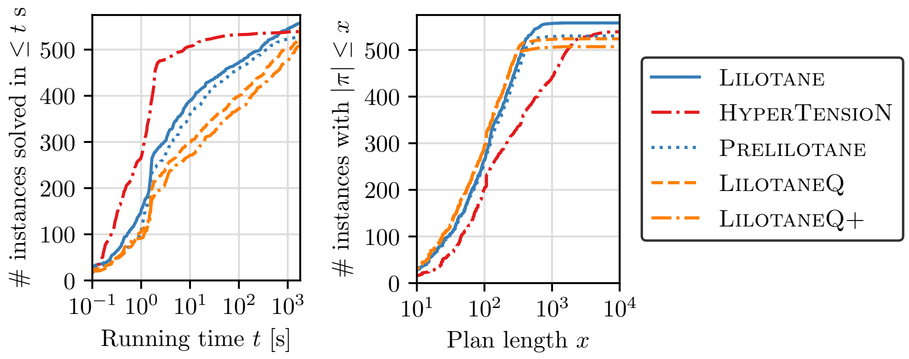


Figure 6.12: CDFs for running times and found plan lengths of HyperTensioN and Lilotane. Note the logarithmic scale along the x axis.

Domain	#	HyperTensioN	PreLilotane	Lilotane	LilotaneQ	LilotaneQ+
AssemblyHierarchical	3	0.80	1.00	1.00	1.00	1.00
Barman-BDI	16	0.55	0.64	0.63	0.92	1.00
Blocksworld-GTOHP	16	0.87	0.89	0.92	1.00	1.00
Blocksworld-HPDDL	1	0.91	1.00	1.00	1.00	1.00
Childsnack	29	1.00	1.00	1.00	1.00	1.00
Depots	23	1.00	0.96	0.97	1.00	1.00
Elevator-Learned	144	0.35	0.83	0.83	1.00	0.98
Factories-simple	3	0.58	0.62	0.67	1.00	1.00
Hiking	21	0.98	0.96	0.96	1.00	1.00
Logistics-Learned	22	0.78	0.72	0.71	1.00	1.00
Minecraft-Player	1	1.00	0.85	0.74	1.00	1.00
Minecraft-Regular	28	1.00	0.85	0.87	1.00	1.00
Multiarm-Blocksworld	4	0.91	0.82	0.85	1.00	1.00
Robot	11	0.51	1.00	1.00	1.00	1.00
Rover-GTOHP	20	0.51	0.69	0.69	0.99	0.96
Satellite-GTOHP	13	0.50	0.58	0.60	1.00	0.99
Snake	17	0.28	0.92	0.89	1.00	1.00
Towers	9	1.00	1.00	1.00	1.00	1.00
Transport	33	0.73	0.75	0.74	1.00	0.99
Woodworking	7	0.94	0.93	0.95	0.98	0.98
Total	421	15.20	17.01	17.02	19.89	19.90

Table 6.2: Normalized *plan length scores* over all IPC instances for which HyperTensioN and all configurations of Lilotane found some plan.

6.6.4.b Plan Quality

In terms of coverage and plan quality, LILOTANE outperforms HYPERTENSION (see Fig. 6.12 right). The plans output by HYPERTENSION are longer than the plans found by LILOTANE even without employing plan improvement. This is because LILOTANE always finds a plan at the smallest depth possible, which strongly correlates with the potential length of plans. By contrast, HYPERTENSION performs depth-first search and hence traverses search space more greedily [MMdS21], resulting in larger plans.

Table 6.2 provides more insights into the approaches' plan quality. We use *satisficing IPC scores* [Col+12]: If π is the shortest plan found by an algorithm for an instance, a score of $|\pi|^*/|\pi|$ is attributed where π^* is some reference plan. In our case, π^* is the best plan found by any of the competitors for this instance. To exclude coverage results, we only considered the instances for which each competitor found some (not necessarily final) plan. For each competitor we summed up the scores within a domain and then normalized the result by the number of solved instances in that domain.

Computed over all 421 such instances, all configurations of LILOTANE outperform HYPERTENSION with respect to this metric. Since none of the used post-IPC improvements impact plan lengths, PRELILOTANE and LILOTANE achieve very similar results. LILOTANEQ improves upon LILOTANE's score by almost three points. The degree of improvement heavily depends on the domain model. For instance, choosing different trucks and/or construction locations in `Factories` can lead to high variations in plan length, which makes the domain well-suited for evaluating quality-aware planning [SS21c]. Other domains such as `Childsnack` or `Towers` feature a rigid hierarchy and leave no room for plan improvement. LILOTANEQ+ improves on LILOTANEQ's final plan lengths only on 17 instances from six domains. `Barman-BDI` is the only domain where this improvement is reflected in the scores. LILOTANEQ+ occasionally scores lower than LILOTANEQ since plan improvement on the first solvable layer can be much easier to perform than on the subsequent layer. Among all IPC instances, LILOTANEQ+ finished on 477 instances without any improvement over LILOTANEQ. This indicates that depth-optimal plans found at the first solvable layer can rarely be improved any further when the hierarchy is extended by another layer. We expect improvements to further diminish if we instantiate even more layers. As such, LILOTANEQ in particular appears to strike a good balance between high plan quality and low running times and often results in optimal or near-optimal plans.

6.7 Conclusion

We have presented an approach of grounding-free SAT-based TOHTN planning motivated by the combinatorial blowup which grounding can induce. To process the lifted problem representation as is, we proposed a lazy instantiation approach coupled with a reachability analysis and the introduction of non-committal *pseudo-constants* whenever free method arguments are encountered. We presented an according SAT encoding designed for incremental SAT solving and showed its correctness.

We performed a worst-case analysis where we found that our encoding is exponential along fewer dimensions than the prior Tree-REX encoding but introduces more complex logic related to the facts in the problem. We enhanced an existing plan improvement procedure to make Lilotane quality-aware.

Our evaluations suggest that Lilotane outperforms prior SAT-based TOHTN approaches and produces the smallest SAT encodings among these approaches. In comparison with the arguably best prior TOHTN planner, HyperTensioN, Lilotane is more robust if sufficient time is available but often takes more time than the more lightweight greedy progression search planner. Still, investing additional time to solve an instance with Lilotane is worthwhile because the found plans are of high quality, even without any explicit plan improvement. As such, we consider HyperTensioN and Lilotane to correspond to two different points on the Pareto frontier between speed, robustness, and quality.

As we have seen in our evaluations, there are planning domains where grounding is very beneficial while on some other domains it is next to infeasible. We believe that our reachability analysis can be improved further by computing a better approximation of the possible fact changes an operation may effect. This could lead to earlier and more effective pruning of irrelevant operations and may render Lilotane more competitive regarding the domains where grounding is highly effective. Nikolai Schnell [Sch21b] was able to make some initial progress in this direction. Nonetheless, it remains an open question whether the "best of both worlds" can be achieved by creating an efficient instantiation approach which keeps the problem lifted but effectively performs the same *a priori* pruning that is achieved by high quality grounding. Orthogonally, we have begun to investigate parallel portfolios of TOHTN planners which execute lifted and ground approaches in parallel [BWS22].

Chapter 7

7

Distributed Incremental SAT Solving for Hierarchical Planning

We conclude the contributions of this thesis with a liaison of our SAT-based TOHTN planning approach and our work on decentralized job scheduling and scalable SAT solving. Based on prior experimental results, we analyze the potential of parallelizing the SAT backend of our planning application. We propose a low-latency interface for incremental jobs and specifically for IPASIR-style incremental SAT solving to our decentralized SAT processing platform MALLOB*. Moreover, we suggest to process many independent planning instances in parallel, making use of* MALLOB*'s malleable job scheduling. In an experiment where 587 TOHTN planning problems are resolved in parallel on 2348 cores, we find that our measures to minimize overhead in our interface are crucial for achieving speedups. Thanks to the rapid scheduling and resizing of problems, we observe noticeable speedups for several planning domains where SAT solving constitutes a major part of* LILOTANE*'s running times. As such, our findings indicate that our work may be useful for a wide range of SAT applications in the future.*

Author's Notes. *This chapter features previously unpublished content. A preliminary form of the parallelization approach I present in this chapter was featured before in a student research project by Colin Bretl and Niko Wilhelm [BWS22]. I provided the idea for this approach and implemented and evaluated it in the scope of this chapter.*

7.1 Introduction

In the earlier chapters of this thesis, we have explored how distributed environments can be exploited effectively for SAT solving—both with (massively) parallel SAT solving (Chapter 4) and by processing many tasks in parallel (Chapter 3). We have also seen how a specific application, TOHTN planning, can profit from a novel SAT encoding approach which omits a central stage of prior SAT-based approaches (Chapter 6). In this chapter, we extend this case study to distributed environments by connecting it to our distributed SAT processing framework. This endeavor serves two purposes. First, we specifically explore a possible avenue to parallelizing TOHTN planning—a challenging problem due to the 2-EXPTIME-hard nature of TOHTN planning [ABA15]. Secondly, we evaluate and improve the practical merit of our distributed platform from an application perspective and thus provide an outlook on how applications of incremental SAT solving may profit from our work in the future.

We first analyze the data gathered in our earlier evaluation (Section 6.6) and investigate requirements for an effective parallelization of LILOTANE's SAT solving backend. We conclude that an incremental job interface with very low latencies is needed to account for the many trivial SAT calls made by applications like LILOTANE. We then enhance MALLOB by such a low-latency interface for incremental jobs and extend MALLOBSAT to support incremental SAT solving. We conduct an experiment where we use a single large-scale run of MALLOB to process many planning tasks at the same time. We observe noticeable speedups on the planning problems on which our planning approach makes sufficiently difficult SAT calls.

This chapter is structured as follows. We present a brief requirement analysis in Section 7.2; we describe our approach in Section 7.3; we present experimental results in Section 7.4; and we conclude the chapter in Section 7.5.

7.2 Requirement Analysis

In the following, we analyze the data gathered in earlier evaluations of LILOTANE with respect to the opportunities and challenges of parallelizing its SAT solving calls.

During planning, each SAT solving call of LILOTANE corresponds to one hierarchical layer of the problem (see Section 6.3.1). As such, LILOTANE emits a series of unsatisfiable problems until it reaches a layer where a plan can be found and the encoding is satisfiable. Since different domains result in task networks of varying depth, this number of calls varies significantly across domains. For the median IPC 2020 instance, LILOTANE makes ten SAT solving calls. Instances from the `Childsnack` domain, which feature a shallow and fixed hierarchy, always result in three calls. The other extreme is the `Towers` domain at 120.6 calls on average.

In terms of the *total* time which LILOTANE spent on successfully solved instances from the IPC 2020 benchmarks, around 78% was spent on SAT solving itself. On the other hand, considering the *median* instance, only 9.5% of time was spent on SAT solving. Fig. 7.1 (left) shows this ratio for every IPC instance solved by LILOTANE. We found a weak to moderate correlation between an instance's running time and the ratio of time spent on SAT solving (Pearson's $r = 0.497$). Such a correlation is desirable since difficult instances with long running times bear the highest potential for meaningful speedups. On 75% of instances which take at least 20 s to solve, LILOTANE spends the majority of time on SAT solving. The same holds for 80% of the instances which take at least 100 s to solve. From a domain-dependent view, only two domains (`Logistics` and `Assembly`) feature such a SAT solving ratio larger than 50% for the median instance. Eight domains do not feature a single instance where this ratio even exceeds 10%. As such, the effectiveness of a parallelization of LILOTANE's SAT backend will be highly domain-dependent. For fully scalable TOHTN planning across the entire range of available domains, LILOTANE's instantiation and encoding approach would need to be parallelized as well, which we do not consider in this work.

Another point to consider is that the time LILOTANE spends on SAT solving is split into thousands of individual SAT calls, many of which are made in rapid succession

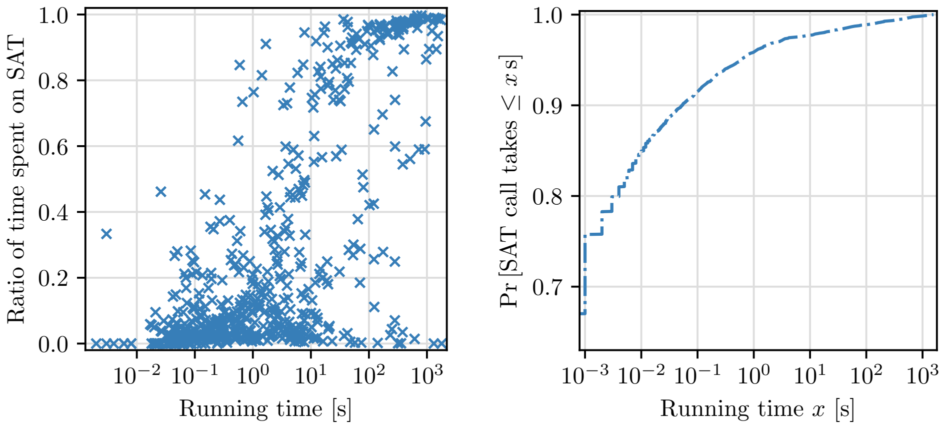


Figure 7.1: Analysis of Lilotane on IPC 2020 instances (see Section 6.6.4). Left: Ratio of time spent on SAT solving for solved instances. Right: CDF for the duration of individual SAT calls on solved instances (note the offset of the y axis). Both x axes are scaled logarithmically.

and only few of which actually prove to be non-trivial. Fig. 7.1 (right) shows that two thirds of all SAT calls take less than a millisecond and more than 95% of SAT calls take less than a second. Therefore, in order to achieve good average speedups with a parallel SAT solving backend, it is crucial to carefully engineer an efficient interface with low turnaround times for individual SAT calls. Parallelization should be performed in a *cautious* manner—only investing work in scaling up a SAT call if it proves to be non-trivial. The job model of Mallob is specifically designed for these criteria, offering low-latency scheduling of incoming jobs and a flexible amount of parallelization based on the system state and a job's perceived difficulty.

7.3 Approach

In the following, we describe how we extended Mallob to support incremental (SAT) jobs and how we designed the interface between a Mallob process and an application.

7.3.1 Incremental Jobs in Mallob

The interface of Lilotane expects an incremental SAT solver. It is possible to simulate an incremental SAT solving interface with a non-incremental solver by adding assumptions as permanent unit clauses and by in turn starting a completely new solver instance for each increment [NR12]. However, our requirement analysis suggests that our application relies on very low response times for decent performance.

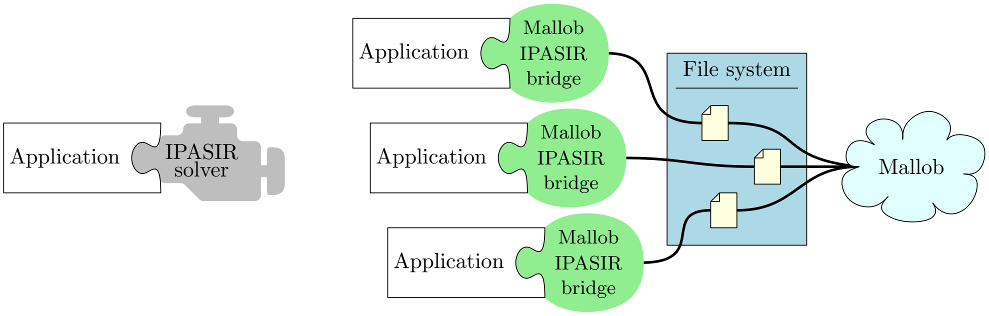


Figure 7.2: Incremental SAT application interface. Left: conventional IPASIR interface with direct linkage of the application code to SAT solver code. Right: our approach. The application is linked with bridge code, which communicates with a running instance of Mallob via the local machine's file system and via inter-process communication (IPC).

For this reason and also to allow a plethora of further applications of incremental SAT solving [AHL08; Liu+16; GB17; KBS19; Sur+22] to use our system, we implemented a "native" pipeline for processing incremental jobs to Mallob.

We model a simple protocol for incremental jobs in Mallob as follows: A job can have multiple *revisions* (which correspond to the increments of incremental SAT solving) indexed by $r = 0, 1, \ldots$. Each revision r adds a certain payload P_r to the job and can be submitted as soon as the previous revision $r - 1$ has finished (or was cancelled).[1] In Mallob, the root worker of an incremental job is present and active from the scheduling of revision 0 until the conclusion of the final revision. Any further workers of the job are suspended whenever revision $r - 1$ is done and revision r is not present yet. Revision payloads are transferred to workers down the job tree just like usual job descriptions, with the constraint that a worker can only begin to process revision r after all payloads $P_0, \ldots, P_r$ have arrived at this worker.

Fig. 7.2 illustrates our technical setup for incremental SAT solving specifically. We emulate the interface IPASIR (Section 2.2.6.a) for the application side and thus provide a drop-in replacement for sequential IPASIR solvers. On a technical level, an application process communicates with Mallob by writing a small (JSON) file, describing the job, to a certain directory watched by a local Mallob process. The job description itself is then transferred from the application process to the Mallob process via inter-process communication (IPC). The application process then waits for the Mallob process to transfer back a result. We use named (UNIX) pipes for inter-process communication to ensure fast transfer of large amounts of data (i.e., formulas and satisfying assignments). We use the Linux subsystem `inotify` for watching files in a directory, which allows Mallob to react to incoming jobs immediately.

[1]We have considered an extension to this model where multiple increments of a job can be solved in parallel. This extended protocol is still in an early experimental stage.

7.3.2 Incremental SAT Solving Engine

In order to prepare MallobSat (Chapter 4) for incremental SAT solving, we restrict our portfolio to CaDiCaL and Lingeling which both natively support incremental SAT solving. We perform clause sharing just like for non-incremental solving. The only SAT-specific invariant we need to maintain is that a solver never imports any clauses from a future revision $r' > r$ with respect to its own current revision r. We achieve this by marking each contribution to a clause buffer with the current revision of the exporting SAT process. When aggregating buffers, the maximum revision that is present is propagated and is then broadcast together with the final buffer.

7.3.3 Reducing Revision Turnaround Times

Our analyses of Lilotane's behavior on IPC instances showed that low latencies of individual SAT solving calls are crucial for its performance. We made a few small modifications to Mallob to account for this requirement and to minimize the average turnaround time of a trivial SAT call.

First, we remove large parts of the initial scheduling latency of a job for subsequent revisions by preserving the root worker of each job across revisions. To introduce a new revision for a job, only a single message (featuring the new revision payload) must be sent to this root worker. Additional workers are then allocated as usual by emitting a balancing event, computing new fair job volumes, and emitting job request messages (see Section 3.4). Secondly, Mallob's main thread usually checks the status of a job every 10 ms. This interval is now set to 1 ms whenever a new worker is scheduled on the process and then increases successively until it reaches 10 ms. Thirdly, we reduced the latency of a SAT solving process reporting a trivial result by letting Mallob's main thread read and parcel small job results itself immediately instead of always running a separate background worker for this task.

7.4 Evaluation

We now turn to the experimental evaluation of our approach. Our software and experimental data are available online (see Appendix A). We run all experiments on SuperMUC-NG (see Section 3.6.1), using GCC 11.2 and Intel MPI 2019.12.

As a sequential baseline, we run Lilotane on the IPC 2020 benchmarks. We changed Lilotane's backend solver from Glucose to CaDiCaL since the latter proved to result in overall better performance [Sch21b].

We refer to our approach as Mallotane [BWS22]—a portmanteau of Mallob and Lilotane. We use a subset of the IPC 2020 benchmarks, filtering out two kinds of instances based on our sequential baseline: (a) instances where Lilotane exceeded the main memory limit of 8 GB, and (b) instances where the time spent on SAT solving accounts for less than 0.1% of the total running time. 34.2% of all instances match these criteria, leaving 587 instances for our experiment. Evidently, to improve performance on the filtered instances, more invasive changes to Lilotane itself would be in order.

Note that the remaining benchmark set still features instances from several domains where the amount of time spent on SAT solving is negligible, therefore allowing us to analyze the worst-case behavior of Mallotane.

We construct a similar scenario as in the *Massively Parallel Processing of SAT Jobs* (Section 4.6.6): We allocate a fixed amount of computational resources and attempt to resolve as many problems as possible within a given time frame. Since we only parallelize parts of our planning procedure, we decided to set up our experiment in such a way that each planner instance initially receives very few resources (four cores) and then successively scales up to 30-40 cores given that the task is sufficiently difficult. Specifically, we configure each job to grow by another job tree layer every 0.5 seconds: We initialize job demand d_j (the desired number of processes, see Section 3.3) to 1 at the arrival of j and update $d_j := 2 \cdot d_j + 1$ every 0.5 seconds until d_j has reached the number of processes in the system.

Deploying Lilotane and Mallob at the same time on SuperMUC-NG proved to be challenging due to the small amount of main memory available (2 GB per core or 1 GB per hardware thread). Since one instance of Lilotane on average requires considerably more memory than one SAT solver instance in our system, we decided to deploy Lilotane and Mallob as follows:

- We allocate $4 \times 587 = 2\,348$ cores of 49 compute nodes, spawning one Mallob process for each set of four cores.
- We configure MallobSat to only employ *three* solver threads per process.
- Together with each Mallob process we execute one instance of our planner, which then communicates with the Mallob process as a SAT interface. Note that this Mallob process only introduces the corresponding job to the decentralized scheduler—the job's first worker may be scheduled to *any* Mallob process.

Note that, in order to avoid allocating additional compute nodes, we distributed the planning problems over the compute nodes in a balanced manner based on the amount of memory required by sequential Lilotane. In a real-world application of our system, we would expect the application code (i.e., our planners) to be run on client machines that are separated from Mallob's worker processes.

7.4.1 Results

We now discuss the results of our experiment. Tab. 7.1 lists the speedups of Mallotane over Lilotane and Fig. 7.3 shows a visual comparison of individual running times. As expected, our setup incurs some overhead compared to a sequential SAT solver that is compiled into the planning application. This overhead, however, proves to be small and mostly concerns easy problems which are resolved in less than a second. Whether Mallotane results in speedups and how high these speedups can reach is, unsurprisingly, highly dependent on the planning domain. We observed a median speedup greater than two for the five domains `Elevator`, `Freecell`, `Logistics`, `Minecraft-Player`, and `Multiarm-Blocksworld`. Mallotane achieved the highest domain-specific performance for `Elevator` at a median speedup of 3.4 and also achieved the largest individual speedup (24.8) on a problem from this domain.

Domain	#	min.	med.	geom.	max.	total
Assembly	5	0.15	1.11	0.772	3.19	2.821
Barman	18	0.18	0.84	0.808	4.12	2.069
BlocksworldG	22	0.12	0.81	0.639	1.14	0.946
BlocksworldH	1	0.91	0.91	0.910	0.91	0.914
Childsnack	26	0.19	0.46	0.460	1.09	0.930
Depots	22	0.32	0.77	0.734	1.33	1.196
Elevator	146	0.14	3.40	3.103	24.82	9.054
Entertainment	2	0.93	0.99	0.960	0.99	0.938
Factories	4	0.16	1.75	0.839	2.06	1.835
Freecell	12	1.01	2.28	2.429	5.17	3.228
Hiking	23	0.29	0.82	0.815	2.06	1.455
Logistics	50	0.54	2.02	1.948	3.86	2.171
MinecraftP	4	0.78	2.16	1.717	2.67	2.294
MinecraftR	32	0.37	0.96	0.895	1.09	0.990
MonroeFO	19	0.58	0.91	0.899	1.14	0.949
MonroePO	19	0.83	0.94	1.014	2.68	1.058
Multiarm	4	1.02	2.20	1.649	3.17	2.869
Robot	8	0.10	0.34	0.375	1.37	1.249
Rover	24	0.12	1.89	1.528	5.85	3.477
Satellite	15	0.16	1.04	0.849	1.42	1.125
Snake	20	0.14	0.74	0.586	1.11	1.027
Towers	6	0.20	0.84	0.581	0.89	0.864
Transport	32	0.13	0.54	0.645	8.63	3.207
Woodworking	22	0.14	0.65	0.547	1.01	0.824

Table 7.1: Speedups of MALLOTANE over LILOTANE by domain on the commonly solved instances from the considered subset of IPC 2020 benchmarks, in terms of minimum, median, geometric mean, maximum, and total speedup.

In addition, MALLOTANE is able to solve ten additional instances, specifically from the domains `Logistics` (5), `Assembly` (2), `Rover` (2), and `Transport` (1). There are three domains on which MALLOTANE never achieves any speedups, namely `BlocksworldH`, `Entertainment`, and `Towers` (which together only amount to nine considered instances). For a total of 15 domains, the median speedup achieved is less than one, indicating a slowdown for the majority of instances from these domains. `MinecraftR` is one of them and constitutes a good example for a domain where very little time is required for SAT solving. On eight instances in this domain, LILOTANE spent less than 1% of its time on SAT solving with running times ranging between 12 and 89 seconds. MALLOTANE's "speedups" range between 0.924 and 0.972 on these instances, indicating an overhead between 2.8% and 7.6%.

As Fig. 7.3 shows, almost all instances which result in a considerable slowdown are solved very quickly in terms of absolute running times. As such, while the median speedup of MALLOTANE on the considered benchmark set is 1.018, its total speedup is 2.63. In addition, we set each speedup in relation to the ratio of time (sequential) LILOTANE spent on SAT calls. Fig. 7.4 confirms that most slowdowns occur on instances where SAT solving constitutes a small ratio of the overall running time.

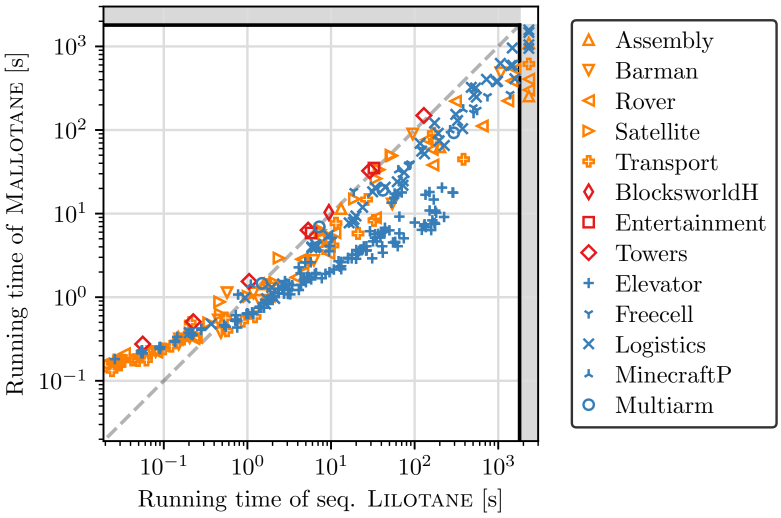


Figure 7.3: Selected (log-log) running times of LILOTANE vs. MALLOTANE, showing five domains with median speedup ≥ 2 (blue), three domains where no instance was sped up (red), and five domains with notable outliers (orange).

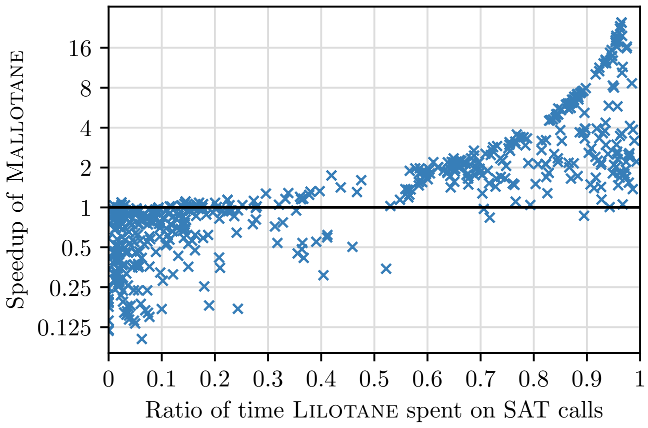


Figure 7.4: Speedups of MALLOTANE (log. scale) relative to the ratio of time sequential LILOTANE spent on SAT calls, with $y = 1$ highlighted by a black line.

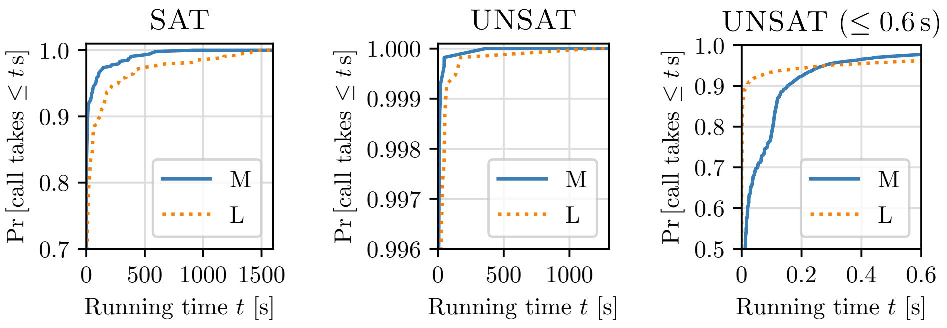


Figure 7.5: Distribution over the duration of individual SAT solving calls for sequential Lilotane (L) and for Mallotane (M), separated into SAT and UNSAT results, for commonly solved instances only. The y axes are offset to allow for discerning relevant differences. Since almost all UNSAT calls are very short, we added a separate plot only showing calls which take less than 0.6 s.

Indeed, whether or not Mallotane achieves a speedup > 1 can be predicted rather accurately based on whether sequential Lilotane spends the majority of its time on SAT solving. Such a classifier wrongly attributes no speedup to 41 instances with some speedup and wrongly attributes a speedup to 4 instances with no speedup. All remaining 491 instances are classified correctly.

On the 536 commonly solved instances, both the sequential and the parallel approach made a total of 6181 SAT solving calls, thereof 5645 calls resulting in unsatisfiability ("UNSAT calls") and the remaining 536 calls resulting in satisfiability ("SAT calls"). Fig. 7.5 shows how the duration of these calls is distributed for the sequential and the parallel backend. With the sequential backend, 96.86% of UNSAT calls and 57.65% of SAT calls took less than one second—confirming that the performance of our planner crucially depends on low overhead for individual calls. The break-off point where our Mallob backend begins to outperform the sequential backend is at 0.23 s: The same ratio of calls (90.4%) finished within this time for both approaches. For UNSAT calls this break-off point is at 0.28 s (95.1% of calls finished) while for SAT calls it is already at 0.053 s (31.7% of calls finished). Still, since both satisfiable and unsatisfiable increments can occasionally be very hard, the total speedup is comparable for SAT calls vs. UNSAT calls (3.257 vs. 2.908) and, notably, closely matches the on average three SAT solving threads per instance.

Fig. 7.6 illustrates how a job's volume changes over time in our experiment. The displayed instance from the `Assembly` domain requires a total of 31 SAT solving calls and finishes within 18.4 minutes of total running time. Lilotane with a sequential CaDiCaL backend on this instance exceeds its time limit (30 minutes) more than 22 minutes into its 27th SAT solving call. Mallotane solves that increment 263 s after application start, taking 108 s for the SAT solving call itself.

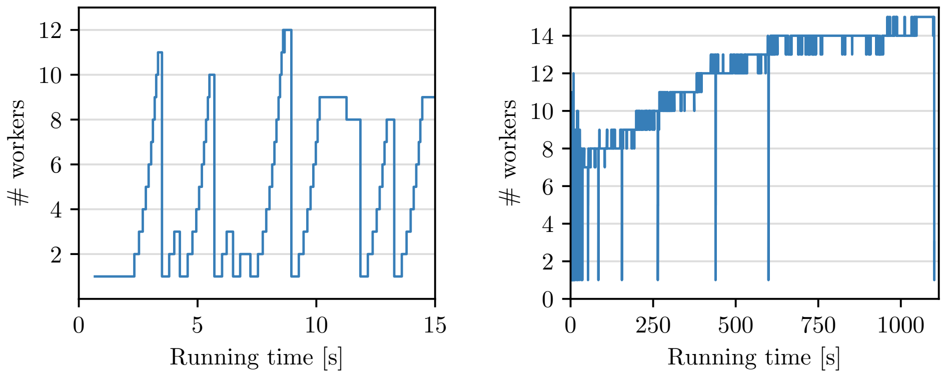


Figure 7.6: Job volume (# workers) for a difficult `Assembly` instance, in the first few seconds (left) and over the job's entire processing time (right).

Whenever an increment is solved, the job shrinks down to a single "standby" worker (the job's root worker). Whenever a new revision is introduced, the job's demand and consequently its volume increase exponentially over time until the demand has reached the current fair share of resources which the job is entitled to. The most difficult increment to solve is the final unsatisfiable increment at a SAT solving time of 501.5 seconds, after which the very final (satisfiable) increment only takes 0.6 s to solve. In the latter half of the job's life time, the number of cores associated with the job mostly ranges between 39 (13 workers) and 45 (15 workers).

We can conclude from our experiments that connecting SAT-based applications to Mallob as a service-based backend can indeed be beneficial and can result in improved performance. Naturally, the degree of improvement heavily depends on the amount of work the application performs apart from SAT solving. In this sense, (SAT-based) hierarchical planning proved to be a challenging yet interesting application to explore since the problem domains are extremely diverse in their behavior and therefore highlight the strong points of our approach—good speedups for difficult solve calls and improved performance via flexible rescaling of jobs—as well as its remaining weakness, namely mild slowdowns on instances with no difficult SAT calls.

7.5 Conclusion

We explored the connection of our SAT-based hierarchical planning system Lilotane to our decentralized job processing and SAT solving platform Mallob. We analyzed the requirements for such an undertaking and consequently integrated a low-latency interface for incremental jobs in Mallob and extended MallobSat to support incremental SAT solving. Our experiments showed that a combination of parallel job processing and (modestly) parallel SAT solving can result in appealing speedups in cases where our planner emits sufficiently hard SAT increments. We consider these results encouraging for the future use of MALLOB as a SAT backend for other applications relying on incremental SAT solving, e.g., for electronic design automation [Liu+16], model checking [KBS19], or multi-agent path finding [Sur+22].

In terms of future work, it may be promising to go beyond LILOTANE's trivial sequential makespan scheduling and instead attempt to solve multiple increments of an instance in parallel. By generalizing the incremental job model of MALLOB and carefully extending its application interface, such an approach may also benefit other applications of incremental SAT solving (*cf.* [WH13a]). Furthermore, we intend to explore how incremental and non-incremental SAT solvers may cooperate via careful clause exchange in order to exploit the advantages of both approaches.

Chapter 8

8 Conclusion

In this chapter we sum up the central contributions and results obtained throughout this thesis. We estimate the impact of our research, both in terms of future research and practical application, and provide an outlook on possible directions for future work.

8.1 Conclusion

In this thesis, we addressed a number of scalability challenges in the realm of applied SAT solving, guided by three research questions: (i) How can we efficiently exploit modern distributed computing environments for SAT solving? (ii) How can we render SAT solving systems in such environments trustworthy for critical applications? And: (iii) How can complex applications make more efficient use of SAT solvers in order to handle previously infeasible inputs?

On an algorithmic level, we have presented decentralized approaches to schedule malleable tasks with unknown processing time in large distributed environments; we have introduced a compact approach to periodic all-to-all clause sharing and subsequent clause filtering; we have introduced the first feasible approach to produce independently verifiable proofs for distributed clause-sharing solvers; and we have presented the first SAT-based approach to TOHTN planning which keeps a lifted, i.e., parametrized, problem description while encoding and solving a planning problem.

On a practical level, we have introduced MALLOB, a decentralized distributed framework for large-scale job processing which, in particular, features the award-winning distributed SAT solving engine MALLOBSAT. Our framework features practical implementations of our algorithmic contributions to scheduling, SAT solving, and proof production and proved to be fast and scalable compared to baseline approaches. Furthermore, we have presented LILOTANE—an efficient implementation of our lifted SAT-based TOHTN planning approach. Both of these major practical contributions have proven themselves in international research-oriented competitive events on previously unseen inputs, and the liaison of both systems successfully demonstrates how applications of (incremental) SAT solving can profit from MALLOB.

We now discuss whether we were able to address our research questions. Regarding question (i), we successfully demonstrated ways to exploit HPC and cloud environments for SAT solving more efficiently than before—both by introducing a scalable SAT solver that doubles previously achieved speedups and also by scheduling many SAT tasks at once on thousands of cores with great flexibility and resource efficiency.

We also addressed question (ii) with our scalable proof production approach, although it remains an open engineering task to reduce the overhead incurred by our prototypical setup and to allow for malleable scheduling of our trusted SAT solver. Regarding question (iii), we studied the application of TOHTN planning and advanced the state of the art by carefully designing a more direct encoding of the problem at hand, by exploiting incremental SAT solving, and by introducing an approach to *distributed* incremental SAT solving which may benefit many further applications in the future. Our planning approach is indeed able to handle some problems which prior approaches have been unable to solve (see Section 6.6.3). Put together, we argue that we have been able to address our three research questions in a satisfactory (although, of course, not exhaustive) manner.

8.2 Impact

We now estimate the impact of our contributions, in particular considering the reactions and subsequent works they have prompted so far.

As we discussed in detail in Section 4.7, our work on parallel and distributed SAT solving has received a large amount of attention through its success in four subsequent iterations of the International SAT Competition. In 2021 considered "*by a **wide** margin, the most powerful SAT solver on the planet*" by Amazon scholar Byron Cook [Coo21], MallobSat has established itself as the state of the art in distributed SAT solving. So far there have been two third-party submissions of MallobSat to a competition [Man22; Cho23]. We also noticed first interest from the application side [EME22]. While other distributed SAT solving systems begin to gain traction due to improved diversification techniques [ZCC23], we believe that our clause sharing approach is, as of 2023, the single most beneficial approach to exchange information across a large number of solvers. Other disciplines related to SAT may also profit from this clause sharing approach or slight variations thereof. Examples for this are parallel Satisfiability Modulo Theories (SMT) solving (*cf.* [MHS16]) and parallel Quantified Boolean Formula (QBF) solving (*cf.* [BL16])—see Section 2.2.6.

For some applications, the question arises as to whether Mallob should be used as a (distributed) SAT solving backend for the separate application program or rather as the scheduling environment which dynamically deploys and processes the entire application task with an integrated malleable engine. To prepare Mallob for its role as a SAT backend, we have connected our TOHTN planning approach to Mallob (Chapter 7)—ensuring that it features a generic and low-latency interface applications can be connected to. We believe that this alone may be sufficient to allow many applications of SAT, such as multi-agent path finding [Sur+22] or bounded model checking [KT14; KBS19], to make (better) use of parallel and distributed environments and in turn process high-priority problems faster. We also have conducted preliminary case studies (Section 3.5.3) which, overall, leave us optimistic that implementing new application interfaces into Mallob is reasonably convenient and can be performed without any intricate knowledge on the system's distributed algorithms.

We consider our contributions to producing proofs of unsatisfiability from distributed clause-sharing solvers to be important pioneering work towards fully trustworthy distributed general-purpose SAT solvers that are similarly efficient as their less trustworthy counterparts. The recent implementation of full LRAT support in the CaDiCaL solver [PFB23] was partially motivated by our work. We intend to integrate this version of CaDiCaL into MallobSat for even better performance than with our prototypical setup. According to Marijn Heule, it is now a distinct possibility to use our system for solving some of the next open problems of mathematics. Furthermore, our work opens up vast possibilities for closely analyzing the proof structure of modern clause sharing solvers, which may help to better understand their behavior and their scalability limits (see Section 2.3.2.d).

Lilotane in the context of this thesis serves as a case study on how taking new and radical approaches to encoding applications into SAT can lead to efficient application solvers which outperform existing “direct”, i.e., non-SAT-based, approaches. Since the initial publication of our work [Sch21d], many new approaches and techniques for SAT-based and other ways of HTN planning have emerged [Höl+20b; Beh21; Höl21; BS21; Beh+22]. Depending on the planning domain, our planning system is still one of the best performing TOHTN planners.[1] In the IPC 2023, two independent third-party submissions were based on Lilotane. First, a configuration of LiftedLinear [Wu+23] (a preprocessor which transforms HTN problems into TOHTN problems) was powered by Lilotane as its planning backend and achieved very respectable results as the best performing lifted planning system in the Partial Order Agile track—overall ranked 6th out of 15 and achieving 94.6% of the winning system’s score points [BSA23]. Secondly, the system LTP (Lifted Tree Path) [QPF23] uses both the algorithmic framework and the codebase of Lilotane as a foundation. As such, we observe that our work has gained traction in the automated planning community and is being considered for future efforts on efficient hierarchical planning.

8.3 Future Work

We now outline some directions for future work concerning this thesis as a whole. Note that we also discuss more specific future work in Sections 3.7, 4.8, 5.7, 6.7, and 7.5.

Application studies. An important line of future work is the transfer of our work to application domains. We have made crucial first steps by preparing Mallob as a scalable incremental SAT solving backend and initiating some case studies for supporting applications beyond SAT solving in Mallob. As a next step, we intend to design and implement a distributed approach to QBF solving (see Section 2.2.6.b) as a “first-class” application engine in Mallob. Furthermore, we plan to conduct additional case studies where we connect SMT solvers and other SAT-based verification tools to Mallob. Beyond SAT-related applications, we are interested in exploring more classical applications of malleable scheduling [Bla+04; Hun04; SS12].

[1] For instance, see `Monroe` and `Woodworking` in the 2022 evaluations by Behnke et al. [Beh+22].

For these applications, the given input data is not replicated on all workers but rather redistributed dynamically whenever a job is resized. Our research so far was mainly focused on supporting frequent, rapid, and significant changes of a job's resources, which presumably renders constant data redistribution prohibitively expensive. In future work we would like to consider our approach for scheduling Big Data applications (*cf.* [Reu+18]) where replicating the input on all workers is considered infeasible.

On a related note, it may be worthwhile to explore alternative communication protocols for job-internal communication which does not solely rely on Mallob's job tree. Such a communicator may prove to be crucial for some malleable algorithms such as randomized work stealing or efficient data redistribution.

Improving parallel SAT solving. As of now, general-purpose parallel SAT solving appears to be dominated by clause-sharing portfolio solvers. It remains to be seen whether tightly integrated parallel solvers such as Gimsatul [FB22] continue to gain traction and eventually outperform conventional, modular portfolio solvers due to specialized data structures and better efficiency. In this context, a natural extension of our work would be to orchestrate such integrated solvers, one per machine or per socket, using our decentralized scheduling and distributed clause sharing techniques. Conversely, there are recent developments of new and carefully designed application interfaces emerging for SAT solvers, such as IPASIR-UP [Faz+23] and IPASIR 2,[2] which offer sophisticated ways to orchestrate solvers. Developing efficient portfolio solvers without any kind of solver-specific code may hence become a possibility.

With the increasing attention towards proof systems beyond resolution [Bal+23a], an important question for future work is how to extend parallel solvers to support these more sophisticated reasoning techniques. In particular, a preprocessing technique called *Structured Bounded Variable Addition* (SBVA) [HGH23; HG23] led to impressive results in the sequential track of the most recent SAT Competition [Bal+23a]. It appears natural to investigate (a) how to enable sound clause sharing after some or all solvers performed SBVA, (b) how to extend our distributed proof production approach to such a setup, and (c) how to parallelize SBVA itself—considering that the preprocessing can take many minutes on some formulas [HGH23]. The latter question falls into the broader category of parallel pre- and inprocessing, a problem which has so far not been addressed in a fully satisfactory manner [HW13; GM13; IBS19].

Heterogeneous and fault-tolerant computing. Throughout this thesis, we have dealt with homogeneous hardware that solely features CPUs. In recent years, graphical processing units (GPUs) have become increasingly prevalent in HPC environments and, as of now, contribute significant amounts of today's top supercomputers' performance [Kha+21]. We thus suggest that several parts of our work should be extended to exploit heterogeneous environments which feature a combination of CPUs and GPUs on several types of compute nodes. For example, recent works in terms of SAT solving exploit GPUs for parallel inprocessing [OWB21a] and clause strengthening [PSM21].

[2] Under active development, see `https://github.com/ipasir2`

Another important topic is to make distributed computations fault-tolerant [CL99; Phi05; Hüb+21], i.e., to have them gracefully handle software and hardware failures. Our decentralized scheduling, not relying on any single point of failure, appears to be well-suited for such an undertaking. All in all, extending our work to heterogeneous and fault-tolerant computing may allow different applications to untap the full potential of modern HPC environments.

Appendix

Chapter 9

9 Appendix

A Online Repository: Software and Experimental Data

The online repository accompanying this work is available at `https://zenodo.org/doi/10.5281/zenodo.10184679`. This repository contains the gathered experimental data as well as references to the used benchmarks and the evaluated software.

B Scalable SAT Solving: Supplementary Material

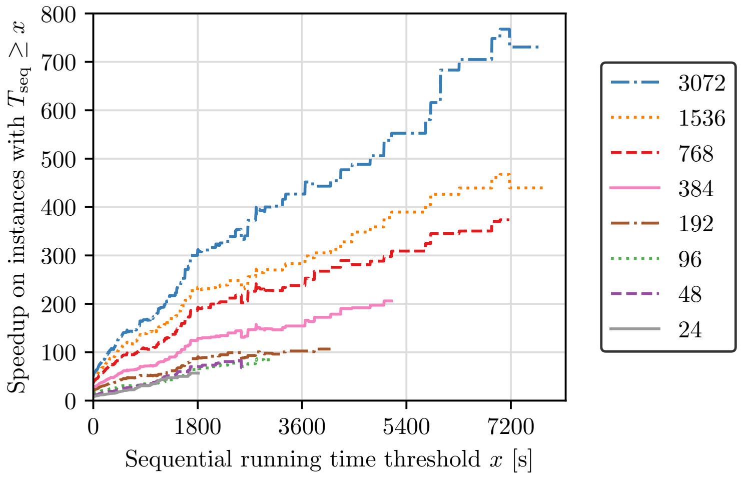


Figure 9.1: Weak scaling of MallobSat on 24 to 3072 cores. For a given sequential running time threshold x, speedups are computed based on commonly solved instances which took KissatMABHyWalk $T_{\text{seq}} \geq x$ seconds to solve. At each scale, we display data until less than 25 instances are considered.

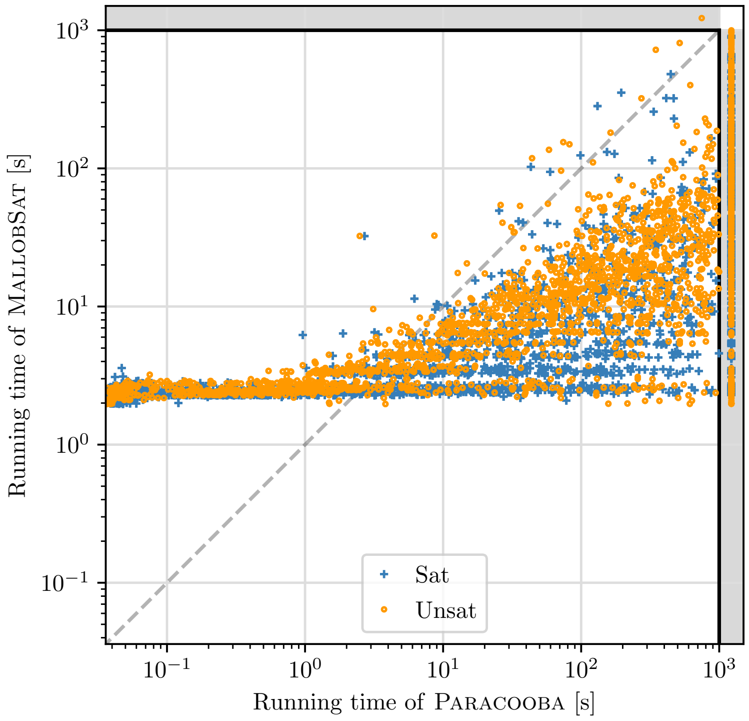


Figure 9.2: Performance of MallobSat vs. Paracooba in the ISC 2022 Anniversary cloud track (5355 instances). MallobSat's performance on simple instances is impacted (a) by the startup time of OpenMPI in the AWS setup, whereas Paracooba immediately begins executing a sequential Kissat instance [Hei22], and (b) by the manager script only polling for a response from our solver once a second. Most instances which take more than a few seconds to solve are solved substantially faster (or exclusively) by MallobSat.

	Idx.	Configuration
Lingeling	A	classify=0
	0	gluescale=5
	1	plain=1 decompose=1
	2	plain=0|1 locs=-1 locsrtc=1 locswait=0 locsclim=16777216
	3	restartint=100
	4	sweeprtc=1
	5	restartint=1000
	6	scincinc=50
	7	restartint=4
	8	phase=1
	9	phase=-1
	10	block=0 cce=0
Glucose	A	!adaptStrats simplify [>2]randomizeFirstDescent [>2]rndInitAct
	0	adaptStrats simplify
	1	lubyRestart lubyRestartFactor=100 (max)VarDecay=.999
	2	chanseokStrat coLBDBound=4 glureduce 1stReduceDB=2k clsBeforeReduce=2k !incReduceDb
	3	(max)VarDecay=.95 firstReduceDB=4k lbdQueueSize=100 K=0.7 incReduceDB=500
	4	!adaptStrats !simplify
	5	chanseokStrat
	6	adaptStrats !simplify
	7	chanseokStrat coLBDBound=3 glureduce 1stReduceDB=30k (max)VarDecay=.99 randomizeOnRestarts
CaDiCaL	0	phase=0
	1	config=sat
	2	elim=0
	3	config=unsat
	4	condition=1
	5	walk=0
	6	restartint=100
	7	cover=1
	8	shuffle=1 shufflerandom=1
	9	inprocessing=0
Kissat	A	quiet=1 check=0
	0	eliminate=0
	1	delay=10
	2	restartint=100
	3	walkinitially=1
	4	restartint=1000
	5	sweep=0
	6	config=unsat
	7	config=sat
	8	probe=0
	9	failedcont=50 failedrounds=10
	10	minimizedepth=10^4
	11	modeconflicts=10^5 modeticks=10^9
	12	reducefraction=90
	13	vivifyeffort=1000
	14	xorsclslim=8

Table 9.1: Cyclic solver configuration of MallobSat (Chapter 4.4.1) used in our experiments. “A” corresponds to the default for all configurations; “[>2]” denotes that all but the first three solvers are configured as such.

C Distributed UNSAT Proofs: Supplementary Material

	#	min	p10	med	mean	p90	max
DRAT check	81	24.564	161.947	636.053	1025.771	2675.848	3399.476
Seq. assembly	139	6.141	37.998	158.011	277.023	747.614	1571.190
Seq. postprocessing	139	0.120	1.695	13.776	31.376	87.583	231.958
Seq. checking	139	0.716	7.627	60.587	140.934	368.082	1200.319
Seq. asm+post+chk	139	7.924	62.542	242.040	449.334	1208.350	2831.480
Par. assembly	135	2.196	10.763	41.781	96.167	231.383	1054.070
Par. postprocessing	135	0.202	1.552	16.708	34.587	82.215	338.245
Par. checking	135	0.867	5.157	59.240	148.206	377.040	1469.763
Par. asm+post+chk	135	3.406	18.492	113.739	278.960	697.353	2862.080
Cld. assembly	157	1.474	11.008	61.019	108.122	277.119	848.708
Cld. postprocessing	157	0.249	2.944	31.703	87.176	266.439	690.279
Cld. checking	157	1.141	9.564	130.755	347.430	1006.636	2626.983
Cld. asm+post+chk	157	3.626	36.400	217.736	542.728	1526.270	4165.970

Table 9.2: Statistics on proof production and checking given *in seconds*.

D Lilotane: Formal Definitions and Proofs

D.1 Hierarchical Solutions

Definition 9.1 (Formalization of Def. 6.2)

A directed tree $H = (V, E)$ with a total node ordering relation $\prec \subseteq V \times V$ is a hierarchical solution *to a problem Π iff (1)–(3) hold.*

(1) Each leaf node v corresponds to some action $a_v \in A$, and each inner node v corresponds to some reduction $r_v \in R$. In particular, the root node $\hat{v}$ corresponds to the initial reduction, *i.e., a reduction r_0 with subtasks$(r_0) = T$.*

(2) If an inner node u has k outgoing edges $(u, v_1), \dots, (u, v_k)$, sorted such that $v_i \prec v_j$ if $i < j$, then r_u has k subtasks and each v_i corresponds to an operation which matches the i-th subtask of r_u.

(3) Let $\Omega := \langle o_1, o_2, \dots, o_k \rangle$ be a sequence of operations which results from a depth-first traversal of H beginning from $\hat{v}$ and using $\prec$ as an ordering. Specifically, if node v is visited, its corresponding operation is appended to Ω and all children of v are added according to "$\prec$" to the frontier of nodes to visit (i.e., v_1 is visited before v_2 if $v_1 \prec v_2$). Then there is a sequence of actions π such that (A) holds: (A) π is a solution for Π according to Def. 6.1 where either case 1 applies or case 2 with $r := o_1$ applies or case 3 with $a := o_1$ applies; and in the two latter cases, (A) holds recursively for the resulting π', the resulting Π', and for $\Omega' := \langle o_2, \dots, o_k \rangle$.

D.2 Reachability Analysis

Theorem (6.3, Section 6.3.2.a)

For $l \geq 0$ and $x \geq 0$, let $O_l := \langle o_0, \dots, o_x \rangle$ be a sequence of operations where each o_i is a possible operation at position $P_{l,i}$. Decompose each $o_i \in O_l$ to some sequence $\mathcal{O}_i$ of actions such that $\mathcal{O} := \mathcal{O}_0 \circ \dots \circ \mathcal{O}_x$ is executable from s_I.
(1) If fact f holds after executing $\mathcal{O}$, then f is reachable at $P_{l,x+1}$ according to S_l^{x+1}.
(2) Similarly, if f does not hold after executing $\mathcal{O}$, then $\neg f$ is reachable at $P_{l,x+1}$.

Proof. By definition, the sets $\pm S_l^{x+1}$ grow monotonically in x.

(1) If f holds after executing $\mathcal{O}$, then either (i) $f \in s_I$ and f never changed, or (ii) the execution of $\mathcal{O}$ causes f as an effect in some action. In case (i), f is reachable at $P_{l,x+1}$ by definition. In case (ii), there is some action o_f which causes f as a direct effect, implying $f \in pfc(o_f)$ and either (a) $o_f = o_j$ for $0 \leq j \leq x$ or (b) o_f is a (transitive) child of one such o_j. In case (a), $PFC^+_{l,j+1} \supseteq pfc(o_f) \ni f$ by definition, and in case (b), $PFC^+_{l,j+1} \supseteq pfc(o_j) \supseteq pfc(o_f) \ni f$ because o_f is a child of o_j. In both cases (a) and (b) $PFC^+_{l,j+1}$ is added to $+S_l^{j+1}$ and, since $j \leq x$ and $+S_l^x$ grows monotonically in x, we obtain $f \in +S_l^{x+1}$, hence f is reachable according to S_l^{x+1}.

(2) If f does not hold after executing $\mathcal{O}$, then either (i) $f \notin s_I$ and f never changed, or (ii) as (1)(ii) but with a negative effect. In case (i), $\neg f$ is reachable at $P_{l,x+1}$ by definition. In case (ii), similar to (1)(ii) we obtain $f \in PFC^-_{l,x+1}$ and consequently $f \in -S_l^{j+1}$. Since $-S_l^x$ is monotonic, we obtain $f \in -S_l^{x+1}$, hence $\neg f$ is reachable. □

D.3 Correctness of Encoding

We show the correctness of the Lilotane encoding, beginning with some prerequisites.

Lemma 9.2

For any satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ there is exactly one active operation at each position $P_{l,x}$ for $l \in \{0, \dots, l'\}$ and $x \in \{0, \dots, |L_l| - 1\}$.

Proof. At position $P_{0,0}$ exactly the initial reduction is active due to construction and Eq. 6.1. At each further position $P_{l,x}$ at most one action and at most one reduction is active due to Eq. 6.4 which, together with Eq. 6.3, ensures that at most one operation is active. Also, there is at least one active operation at each position: If there were not a single active operation at some position, then, due to Eq. 6.8 and our instantiation technique where all offset positions are filled with children (possibly with ε-actions), the parent position would be empty as well. Repeatedly applying this argument leads to a contradiction to Eq. 6.1. □

Lemma 9.3

The result π of decoding a plan from a satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ as described in Section 6.4.3 is well-defined and unambiguous.

Proof. First we observe that at the final layer l' there must be an action (and no reduction) at every position due to Eq. 6.10 together with Lemma 9.2.

Next we argue that the substitution of pseudo-constants with actual constants in any active operation o is unambiguous. For each pseudo-constant κ, due to Eq. 6.11 at most one substitution can be active. Furthermore, if the operation o_p from which κ originated is active, then due to Eq. 6.12 each pseudo-constant κ must have at least one and thus exactly one constant substituted for it. The only possibility for o to be active yet o_p to be inactive is that o came into effect by dominating some other operation o' which has another parent and does not contain κ. In that case, Eq. 6.20 will enforce at least one substitution of κ to be active. Otherwise, by construction κ can only occur in the hierarchy induced by the origin operation of κ. Hence, for each active operation in a solution there is exactly one set of active substitutions which replace each pseudo-constant with actual constants.

Together with Lemma 9.2, this implies that there is exactly one operation at each position with exactly one ground representation each, and the final layer only consists of actions. This renders our plan decoding well-defined and unambiguous. □

Next, we establish a mapping of satisfying assignments of fact variables to world states:

Lemma 9.4

Let $\mathcal{A}$ be a satisfying assignment for $\mathcal{L}_{l'}(\Pi)$. Then for each position $P_{l,x}$ we can unambiguously infer a world state $s_{l,x}$ based on $\mathcal{A}$ and s_I.

Proof. At each position $P_{l,x}$ there is a set of variables f^l_x which represent ground facts. Each such fact f has a polarity assigned to it by $\mathcal{A}$. Additionally, some positive facts may not have been encoded (yet) at $P_{l,x}$ but must hold nevertheless: These are exactly the unencoded facts which are contained in the initial state (see Chapter 6.4.2). Consequently we define $s_{l,x} := \{f \mid \mathcal{A}(f^l_x) = \texttt{true}\} \cup \{f \in s_I \mid \mathcal{A}(f^l_x) = \bot\}$. □

In order to reason about the correct application of actions, we show that whenever a particular action with pseudo-constants is active, all preconditions and effects of the implied ground action are enforced correctly.

Lemma 9.5

Consider a satisfying assignment $\mathcal{A}$ for $\mathcal{L}_{l'}(\Pi)$ and an active operation o at position $P_{l,x}$ which becomes a ground operation $\tilde{o}$ through zero or more active substitutions. Then the following holds:
(i) $pre(\tilde{o})^+ \subseteq s_{l,x}$ and $pre(\tilde{o})^- \cap s_{l,x} = \emptyset$.
(ii) If $\tilde{o}$ is an action, then $s_{l,x+1} = (s_{l,x} \setminus eff(\tilde{o})^-) \cup eff(\tilde{o})^+$.

Proof. Eq. 6.5 and Eq. 6.6 enforce that each precondition of o holds at position $P_{l,x}$ and (if o is an action) each effect of o holds at position $P_{l,x+1}$; however, each precondition or effect may contain pseudo-constants. Eq. 6.18 and Eq. 6.19 ensure that the effects are enforced in a consistent way: If some (possibly empty) set of active substitutions unifies a pair of effects to be contradictory, i.e., $\{f, \neg f\} \subseteq pre(\tilde{o})$ for some

f, then the positive effect is enforced as usual while Eq. 6.18 and Eq. 6.19 allow the negative effect to be ignored because an appropriate set of substitutions is active.

As $\tilde{o}$ was obtained from o by substituting each of its pseudo-constants κ with the unique substitution $[\kappa/c]$ that is active in $\mathcal{A}$, Eq. 6.13 enforces that any non-ground preconditions and effects of o are logically equivalent to the respective ground preconditions and effects of $\tilde{o}$—as long as these ground facts are indeed encoded. Some ground precondition (effect) f arising from a precondition (effect) of o through active substitutions may not have been encoded because f is invariant there. In case of an effect, f must be invariantly true because due to construction an effect cannot be invariantly false. In case of a precondition, Eq. 6.22 or Eq. 6.23 prevent any combination of substitutions which unify a precondition of o with an invariantly false fact, hence f must be invariantly true as well. It follows from the correctness of our reachability analysis (Theorem 6.3) that f holds in $s_{l,x}$. Hence, $s_{l,x}$ and $s_{l,x+1}$ induced by assignment $\mathcal{A}$ are consistent with the constraints of $\tilde{o}$ at $P_{l,x}$ and $P_{l,x+1}$.

If o is an action, then any fact *not* featured as an effect does not change, following from our frame axioms: As the position is primitive due to Eq. 6.3 if $\tilde{o}$ is an action, Eq. 6.14 constrains each fact to change only if an action from its direct or indirect support is active. Hence, for each such fact f that changes its polarity, either $o \in \mathit{supp}(f)$ or $o \in \mathit{isupp}(f)$. In the former case, f is a direct effect of o and it follows directly that f is also a direct effect of $\tilde{o}$. In the latter case, Eq. 6.15 implies that a set of substitutions must be active which unify a pseudo-fact effect of o with f. As we know that the active substitutions for the pseudo-constants of o unify o with $\tilde{o}$, we also know that the respective pseudo-fact effect of o must be an effect of $\tilde{o}$ as well. □

Inducing over the length of the final layer (using Lemma 9.5) leads us to the following central property which guarantees that the decoded classical plan is executable:

Lemma 9.6

When a plan $\pi = \langle a_0, \dots, a_{k-1} \rangle$ *is decoded from a valid satisfying assignment* $\mathcal{A}$ *for* $\mathcal{L}_{l'}(\Pi)$*, there is a sequence of states* $Q = \langle s_0 := s_I, s_1, \dots, s_k \rangle$ *such that* $\mathit{pre}(a_i)^+ \subseteq s_i$*,* $\mathit{pre}(a_i)^- \cap s_i = \varnothing$*, and* $s_{i+1} = (s_i \setminus \mathit{pre}(a_i)^-) \cup \mathit{pre}(a_i)^+$ *hold for* $0 \le i < k$*.*

Proof. We construct Q from π and from the states $s_{l',x}$ induced by $\mathcal{A}$.

When plan π contains k actions, the size of the final layer, $k' := |L_{l'}|$, may be larger than k: Layer $L_{l'}$ contains $k' - k \ge 0$ ε-actions which do not contribute to π and which have no preconditions or effects. We inductively construct Q for $k' \ge 0$.

For the base case $k' = 0$ and $\pi = \langle\rangle$, we note that at position 0 all fact assignments must be consistent with the initial state s_I due to construction: When a fact is introduced at some position, then its assignment is fixed according to the initial state (Eq. 6.2 or Eq. 6.21). Consequently $Q := \langle s_I \rangle$ fulfils above requirements.

Let $k' > 0$. By induction, for some $k < k'$ we have a valid sequence of states $Q_{k'-1} := \langle s_0, \dots, s_k \rangle$ for $\pi_{k'-1} := \langle a_0, \dots, a_{k-1} \rangle$ decoded from positions $0, \dots, k'-1$, and we want to construct $Q_{k'}$ for $\pi_{k'}$ decoded from positions $0, \dots, k'$. There is exactly one action active at position k' (see Lemma 9.3)—an ε-action or a normal action.

In the former case, we note by trivially applying Lemma 9.5 that $s_{l',k'} = s_{l',k'-1}$, i.e., no fact changes are possible over the course of this position, so $Q_{k'} := Q_{k'-1}$ fulfils above requirements for the unchanged plan $\pi_{k'} = \pi_{k'-1}$. In the latter case, we have $\pi_{k'} = \pi_{k'-1} \circ \langle \tilde{a} \rangle$, where action $\tilde{a}$ was constructed from the active action a at position k' through a set of substitutions. We apply Lemma 9.5 and obtain $\mathit{pre}(\tilde{a})^+ \subseteq s_{l',k'}$, $\mathit{pre}(\tilde{a})^- \cap s_{l',k'} = \emptyset$, and $s_{l',k'+1} = (s_{l',k'} \setminus \mathit{pre}(\tilde{a})^-) \cup \mathit{pre}(\tilde{a})^+$. As a result, setting $Q_{k'} := Q_{k'-1} \circ \langle s_{l',k'+1} \rangle$ fulfils the above requirements. □

Next, we turn to the hierarchical solution for our problem.

Lemma 9.7

Let H be the decoded hierarchical solution for $\mathcal{L}_{l'}(\Pi)$. Then the structure of H resembles an actual hierarchical solution, i.e., H satisfies (1) and (2) from Def. D.1.

Proof. First we note that the structure of H resembles the structure of hierarchical layers, i.e., H is a tree where each node (o, l, x) has depth l within H and each of its outgoing edges lead to nodes of depth $l + 1$. We perform an induction over the maximum depth $l' \geq 0$ of H counted from its root $(r_0, 0, 0)$ and show that each node (o, l, x) corresponds to a valid operation o and, if o is a reduction, either resides at the maximum depth l' or has outgoing edges to valid child nodes according to $\mathit{subtasks}(o)$. All leaves being actions then follows from the well-definedness of the plan decoding procedure (Lemma 9.3), notably from only actions being active at layer l'.

Let $l' = 0$. Then the only node in the graph is $(r_0, 0, 0)$ where r_0 is fully ground and a valid reduction of the problem by definition.

Let $l' > 0$. Assume that H up to layer $L_{l'-1}$ fulfills each of the Lemma's requirements. We first show that each node at layer $L_{l'-1}$ has the correct number of children at layer $L_{l'}$. Let $(o, l' - 1, x)$ be a node at layer $L_{l'-1}$. If o is an action, then the node has no children by construction. In the following, let o be a reduction. By construction, node $(o, l' - 1, x)$ has $n \geq 0$ outgoing edges to nodes $(o'_k, l', s_{l'-1}(x) + z_k)$ where each z_k is a valid offset for reduction o: $0 \leq z_k < e_{l'-1,x}$. Each o'_k is a ground instantiation of some child action or reduction of o at offset z_k. For each $0 \leq z_k < |\mathit{subtasks}(o)|$ there must be exactly one such child node due to Eq. 6.8 and Lemma 9.2, and for $z_k \geq |\mathit{subtasks}(o)|$ there are no such children because by construction o induces ε-actions as children at those offsets which are never added to H. As a consequence the given parent node has the correct number of children.

Next we show for $0 \leq k < |\mathit{subtasks}(o)|$ that o'_k from child node $(o'_k, l', s_{l'-1}(x) + k)$ matches the k-th subtask of o. Operation o'_k was constructed from the original active operation $\hat{o}'_k$ at position $s_{l'-1}(x) + k$ of layer l' by a series of zero or more active substitutions. We know from Eq. 6.8 and Eq. 6.9 that $\hat{o}'_k$ matches the k-th subtask of its parent $\hat{o}$ from which o was constructed, the only exception being differing pseudo-constant names if the original child of $\hat{o}$ was dominated by another operation (see Chapter 6.3.3). For each substitution $[\kappa/c]$ involved in the transformation of $\hat{o}'_k$ into o'_k, pseudo-constant κ originated either (a) from $\hat{o}'_k$ itself or (b) from parent $\hat{o}$ or some common ancestor or (c) from (a parent of) an operation which dominated the original child of $\hat{o}$. In case (a) we know that the argument of $\hat{o}'_k$ which took κ as its value is a new, free argument not bound by $\hat{o}$. In case (b) we know that κ is globally substituted with the same single constant c. In case (c), Eq. 6.20 sets κ equivalent to the original pseudo-constant, hence (a) or (b) applies. In all cases the arguments of $\hat{o}'_k$ must correspond to the arguments of the k-th subtask of $\hat{o}$, hence o'_k matches the respective subtask of o.

Concerning argument types, the constant c which κ is substituted with is in the valid domain of the respective argument of its origin operation due to Eq. 6.12. Eq. 6.16 and/or Eq. 6.17 enforce any further restrictions to the type of κ in child operations.

All in all, o'_k matches the k-th subtask of o: edge $((o, l'-1, x), (o'_k, l', s_{l'-1}(x)+k))$ represents a valid subtask instantiation o'_k of reduction o. □

Similar to the executability of the classical plan shown in Lemma 9.6, we argue that the hierarchical plan is executable as well:

Lemma 9.8

Let (π, H) be the decoded (classical and hierarchical) solution for the Lilotane *encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Traverse H as described in (3) in Def. D.1 with the node ordering relation $(g,l,x) \prec (g',l',x') \leftrightarrow x < x'$ and maintain a state s which is initialized as s_I and updated with the effects of each visited action. Then the preconditions of all visited actions and reductions hold in s.*

Proof. When performing a depth-first traversal of H as specified, we traverse all action nodes in exactly the order in which the respective actions occur in π. As changes to s are made only when encountering an action node, it follows from Lemma 9.6 that whenever we visit an action node (a,l,x), s is equivalent to the state $s_{l,x}$ that can be inferred from $\mathcal{A}$ at position $P_{l,x}$. In particular, the preconditions of each visited action are met in s.

In the following, we show that when we visit a reduction node (r,l,x) during the traversal of H, the preconditions of r must hold in s. We know that r was constructed from some active reduction $\hat{r}$ and a set of substitutions, hence Lemma 9.5 implies that the preconditions of r hold in the state $s_{l,x}$ inferred from $\mathcal{A}$ at position $P_{l,x}$. Therefore, we know that each precondition f of r is either invariantly true at $P_{l,x}$ or encoded as a direct fact constraint as in Eq. 6.5, possibly via a pseudo-fact (Eq. 6.13).

In the first case, we know due to the correctness of our reachability analysis (Theorem 6.3) that f must hold in all reachable states so far. In particular, f holds in s_I and no action visited so far may have changed it. For this reason, f also holds in s.

In the second case, due to Eq. 6.7, these constraints are propagated down to the final layer $L_{l'}$ from where π was extracted. At this point, assume that we already visited $k \geq 0$ actions. Consequently $s = s_k$ (where s_k is defined in Lemma 9.6 as the intermediate state after applying k actions) and we are currently traversing a subtree to the right of action a_k. It follows that the precondition constraints of r from layer L_l propagate down to a position at the final layer where a_k has already been applied but a_{k+1} has not yet been applied, which is exactly where s_k holds.

As a consequence, fact constraints from layer L_l also directly enforce the preconditions to hold in state s_k and consequently in s. □

Assembling all the parts, we can finally show that our encoding functions as intended:

Theorem 9.9

Let (π, H) be the decoded (classical and hierarchical) solution for the LILOTANE *encoding $\mathcal{L}_{l'}(\Pi)$ for TOHTN planning problem Π. Then π is a valid solution for Π and H is a valid hierarchical solution for Π.*

Proof. We know due to Lemma 9.7 that H satisfies criteria (1) and (2) from Def. D.1 for a hierarchical solution. It remains to be shown that H satisfies criterion (3) for the decoded plan π. Namely, we show the following: If $\Omega := \langle o_1, o_2, \dots, o_k \rangle$ is the sequence of operations visited by a depth-first traversal of H with the node ordering relation $(o, l, i) \prec (o', l', j) \Leftrightarrow i < j$, then π is a (classical) solution for Π in such a way that Ω is a "witness" for which operation should be applied at each recursive step of Def. 6.1.

Let $T := \langle t_0 \rangle$ be the initial tasks of Π where, according to the initial transformation of T we perform, t_0 is a virtual task with r_0 as its only reduction. Let Γ be the frontier of unvisited nodes during the depth-first traversal of H: Initially $\Gamma := \langle (r_0, 0, 0) \rangle$, and each step of the traversal removes node v from the front of Γ and pushes the children of v with respect to $\prec$ to the front of Γ. Clearly, the i-th operation o_i in Ω corresponds to the node at the front of Γ after $i - 1$ nodes have already been visited.

In the following, we transform the recursive Definition 6.1 into an iterative procedure in a straight forward manner to obtain a more natural proof. We maintain a state s initialized with s_I, a task sequence $\mathcal{T}$ initialized with T, and an action sequence $\mathcal{P}$ initialized with π. At each iteration, if we can apply case 2 or case 3 of Def. 6.1 to problem $\Pi := (D, s, \mathcal{T})$ and to solution $\pi := \mathcal{P}$ and obtain some altered Π' and π', then we update s, $\mathcal{T}$, and $\mathcal{P}$ accordingly. Specifically, we apply the i-th operation visited in H at the i-th iteration. If there is no such operation left and we can apply case 1 instead, the procedure terminates.

We show invariant (I): At all times of this procedure, $|\Gamma| = |\mathcal{T}|$ holds and for each $0 \le i < |\mathcal{T}|$ the operation corresponding to the i-th node in Γ matches the i-th task in $\mathcal{T}$. Initially, (I) holds because $\Gamma = \langle (r_0, 0, 0) \rangle$ and $\mathcal{T} = \langle t_0 \rangle$. Next, assume that (I) holds at some point during the procedure. If task t at the front of $\mathcal{T}$ is compound, then according to Def. 6.1, case 2 we replace it with its subtasks. Due to (I), the frontmost node in Γ then corresponds to a reduction matching t. Following the definition of the depth-first traversal, this node in Γ is replaced with operations matching its subtasks in the correct order (Lemma 9.7). If task t is primitive, then according to Def. 6.1, case 3 we remove it from $\mathcal{T}$. Due to (I), the frontmost node in Γ then corresponds to an action matching t and, as such, is a leaf in H, hence it is removed from Γ. Both cases preserve (I).

Next, state s from the above procedure is equivalent to state s from Lemma 9.8 because we begin with $s := s_I$ and alter s with the effects of each visited action.

Now we show that the above procedure is well-defined and terminates. At iteration zero, we have tasks $\mathcal{T} := \langle t_0 \rangle$ and frontier $\Gamma := \langle (r_0, 0, 0) \rangle$. By definition, r_0 matches t_0 and r_0 has no preconditions; we can apply case 2 of Def. 6.1.

At the i-th iteration, we have state s, tasks $\mathcal{T}$, and action sequence $\mathcal{P}$. If $\mathcal{T}$ is not empty yet, then we know due to (I) that operation o corresponding to the first node in Γ matches the first task in $\mathcal{T}$. Lemma 9.8 implies that $\mathit{pre}(o)$ hold in s. Hence, if o is an action, we can apply case 3—we remove a from the front of $\mathcal{P}$ and apply $\mathit{eff}(a)$ to s—and if o is a reduction, we can apply case 2.

If $\mathcal{T}$ is empty, then due to (I) Γ is empty and the traversal of H is finished. All actions in π (each of which corresponds to a leaf node in H) have been visited, so each of them has been removed from $\mathcal{P}$. As a result, $\mathcal{P} = \langle\rangle$ and case 1 applies.

To conclude, we have shown that we can recursively apply the definition for a classical solution for Π to π using the sequence of visited operations in H. This proves that π is a classical solution for Π and that criterion (3) of Def. D.1 is satisfied for H, concluding the proof that H is a valid hierarchical solution for Π. □

E Lilotane: Derivation of Complexity Results

We now derive the complexity results discussed in Chapter 6.4.5.

E.1 Number of Variables

With our encoding approach, each operation at some position of some layer may induce up to $X \cdot B$ new operations at the subsequent layer: At each of the X child positions, up to B new children are possible. Initial layer L_0 has size $|L_0| = 1$ and contains exactly $R_l = 1$ operation per position. Given layer L_{l-1} of size $|L_{l-1}|$ and with R_{l-1} operations at each position, the next layer L_l has a maximum size of $|L_l| \leq X \cdot |L_{l-1}|$ and up to $R_l \leq R_{l-1} \cdot B$ operations at each position. From these inequalities we can deduce $|L_l| \leq X^l$ and $R_l \leq B^l$. For encoding $\mathcal{L}_{l'}$ we have $R := \sum_{l=0}^{l'} |L_l| \cdot R_l \leq \sum_{l=0}^{l'} X^l \cdot B^l = 1 + XB + X^2B^2 + \ldots + X^{l'}B^{l'} \in \mathcal{O}(X^{l'}B^{l'})$ operations in total.

Each operation with U free arguments induces up to U pseudo-constants and thus $U \cdot C$ new variables to represent their possible substitutions. We arrive at $RUC \in \mathcal{O}(X^{l'}B^{l'}UC)$ variables for pseudo-constant substitutions in total.

Let F be the number of relevant facts during the planning process. In the worst case where we need to encode each fact at each position, we obtain a total of $\sum_{l=0}^{l'} |L_l| \cdot F \in \mathcal{O}(X^{l'}F)$ variables representing ground facts.

At each position, P preconditions (and E effects) of each operation (action) may introduce a variable representing a pseudo-fact. We arrive at $R \cdot (P+E) \in \mathcal{O}(X^{l'}B^{l'}(P+E))$ variables for pseudo-facts. For some action effects and for each dominated operation, a variable denoting the equivalence of a pair of pseudo-constants may be encoded. For an action of arity V the equality of $\mathcal{O}(V^2)$ pairs of pseudo-constants may be encoded which leads to $\mathcal{O}(R \cdot V^2)$ variables. If each encoded operation dominates another (unencoded) operation, $\mathcal{O}(R \cdot U)$ equality variables may be added.

For each position we add one variable representing the primitiveness of the position, leading to $\mathcal{O}(X^{l'})$ additional variables. Finally, as explained in the next section, we introduce $\log n$ helper variables whereever we encode at-most-constraints over n variables. This results in $\mathcal{O}(X^{l'} \log B^{l'})$ variables from Eq. 6.4 and in $\mathcal{O}(RU \log C)$ variables from Eq. 6.11.

In total, the asymptotic number of encoded variables evaluates to

$$\begin{aligned} & R + RUC + X^{l'}F + R(P+E) + RV^2 + X^{l'} + X^{l'} \log B^{l'} + RU \log C \\ \in\; & \mathcal{O}(X^{l'}(B^{l'}(1 + UC + P + E + V^2 + U \log C) + F + 1 + \log B^{l'})) \\ =\; & \mathcal{O}(X^{l'}(F + B^{l'}(UC + P + E + V^2))). \end{aligned} \tag{9.1}$$

E.2 Number of Clauses

We traverse our encoding in the same order as presented in Chapter 6.4.1 and provide the asymptotic number of clauses added by each rule.

Eq. 6.1 is a single unit clause. The introduction of new facts in Eq. 6.2 introduces each fact once per layer, leading to $\mathcal{O}(l'F)$ unit clauses, while its optimized replacement Eq. 6.21 subsumes this measure. Eq. 6.3, the primitiveness of a position w.r.t. its active operation, is added once for each operation, leading to $\mathcal{O}(R)$ clauses.

For at-most-one constraints over operations, Eq. 6.4 presents the naive method of adding $\mathcal{O}(n^2)$ binary clauses to restrict n variables. However, if n becomes sufficiently large we employ a more sophisticated encoding with $\mathcal{O}(n \log n)$ clauses (see [Sch18, Appendix A]). As a consequence, Eq. 6.4 asymptotically leads to $\mathcal{O}(X^{l'}(B^{l'} \log B^{l'})) = \mathcal{O}(R \log B^{l'})$ clauses, as $B^{l'}$ operations may occur at each position.

The preconditions and effects in Eq. 6.5 and Eq. 6.6 make up for $\mathcal{O}(R(P+E))$ clauses. The fact propagations introduced by Eq. 6.7 are virtual as explained earlier.

To define child/parent relationships, we add $\mathcal{O}(R)$ clauses from Eq. 6.8 and Eq. 6.9.

Eq. 6.10 leads to $X^{l'}$ assumptions (which are not permanently added to the encoding). The enforcement of at least one active substitution for each pseudo-constant as defined in Eq. 6.12 leads to $\mathcal{O}(RU)$ clauses; one clause for each introduced pseudo-constant. Again employing a binary at-most-one constraint scheme for substitutions instead of Eq. 6.11 we obtain $\mathcal{O}(RU(C \log C))$ further clauses. Eq. 6.13 links each pseudo-fact to its respective ground fact for each possible substitution combination. Overall $\mathcal{O}(R(P+E))$ pseudo-facts are encoded, one for each precondition and effect. In the worst case, up to F ground facts may correspond to a single pseudo-fact and overall we need to add $\mathcal{O}(R(P+E)F)$ clauses.

Direct frame axioms, Eq. 6.14, add up to two clauses for each fact at each position, leading to $\mathcal{O}(X^{l'}F)$ clauses. Indirect frame axioms in Eq. 6.15 are added for each fact f for each action that may indirectly support the change of f, so we may need to instantiate the formula $\mathcal{O}(FR)$ times. For each axiom, our DNF-to-CNF transformation incurs EY clauses, resulting in $\mathcal{O}(FREY)$ indirect frame axiom clauses.

Clauses from Eq. 6.22 and Eq. 6.23 may be added for each precondition of each operation. There can be $\mathcal{O}(F)$ possible substitution combinations for a given pseudo-fact. We can either encode the invalid options (Eq. 6.23) leading to one clause for each invalid substitution or encode the valid options (Eq. 6.22) which may induce $\mathcal{O}(Y)$ clauses for each valid substitution if realized with our DNF-to-CNF transformation. In the worst case we have around $F/2$ valid substitutions and $F/2$ invalid substitutions, in which case we add $\mathcal{O}(RPF)$ clauses in total.

Type restrictions for pseudo-constants—Eq. 6.16 or Eq. 6.17—make up a constant number of clauses per pseudo-constant if chosen correctly, leading to $\mathcal{O}(RU)$ clauses.

To handle contradictory effects, in Eq. 6.18 we enumerate the substitution combinations which unify two given effects of an action. Each operator may have $E/2$ negative effects which can each be unified with each of the remaining $E/2$ positive effects. This leads to $\mathcal{O}(E)$ DNF-to-CNF transformations over combinations of substitution and equality variables, similar to our indirect frame axioms (where pseudo-facts are unified with a *ground* fact). Each transformation can add $\mathcal{O}(EY)$ clauses, so each operation adds $\mathcal{O}(E^2 \cdot Y)$ clauses, leading to $\mathcal{O}(RE^2Y)$ clauses overall.

The equality of pairs of pseudo-constants is encoded as in Eq. 6.19 in some cases. For up to RV^2 variables, $\mathcal{O}(C)$ clauses are encoded, leading to $\mathcal{O}(RV^2C)$ clauses.

If each encoded operation dominates another (unencoded) operation, Eq. 6.20 leads to $\mathcal{O}(R)$ clauses and up to RU equality variables, adding $\mathcal{O}(RUC)$ further clauses.

In total we arrive at an asymptotic number of

$$\begin{aligned}
&\mathcal{O}\Big(l'F + R + R\log B^{l'} + R(P+E) + R + RU + RU(C\log C) + R(P+E)F + X^{l'}F \\
&\qquad + FREY + RPF + RU + RE^2Y + RV^2C + RUC\Big) \\
&= \mathcal{O}\Big(R\big(l'\log B + UC\log C + (P+E)F + FEY + PF + E^2Y + V^2C\big) + FX^{l'}\Big) \\
&= \mathcal{O}\Big(R\big(l'\log B + C(U\log C + V^2) + F(P+YE) + YE^2\big)\Big) \qquad (9.2)
\end{aligned}$$

permanent clauses added to the encoding.

F Lilotane: Supplementary Figures

Domain	#	O	M	X	B	U	P_a	P_r	E	Y_f	Y_a	Y_r	C	$\lvert s_I \rvert$	$\lvert T \rvert$
Barman	20	23	21	7	3	5	7	4	8	2	6	8	41	80	14
Blocksworld	20	13	10	4	2	1	4	4	5	2	2	2	24	29	24
Childsnack	20	9	2	5	2	5	5	5	5	2	5	9	77	101	16
Depots	20	17	14	4	4	3	5	4	6	2	5	5	28	45	8
Elevator	20	16	15	3	3	2	5	5	2	2	3	3	32	299	1
Entertainment	12	15	23	3	9	2	10	7	4	2	4	4	45	403	3
Gripper	20	8	4	6	2	3	3	2	3	2	3	6	27	28	12
Hiking	20	20	20	9	4	5	8	7	8	3	8	8	27	52	1
Rover	20	30	16	4	3	3	6	4	4	3	6	6	58	877	16
Satellite	20	14	12	3	3	2	5	3	3	2	4	4	121	176	79
Transport	30	6	10	4	3	2	4	1	4	2	5	5	25	43	9
Zenotravel	5	11	9	4	3	3	8	8	4	2	9	12	15	20	4
AssemblyHierarchical	30	17	20	2	8	2	9	6	3	3	7	7	116	368	1
Barman-BDI	20	33	23	6	3	5	7	7	8	2	6	6	59	89	11
Blocksworld-GTOHP	30	13	10	4	2	1	4	4	5	2	2	2	138	148	144
Blocksworld-HPDDL	30	14	14	4	5	2	200	200	5	2	200	200	200	413	1
Childsnack	30	9	2	5	2	5	5	5	5	2	5	9	262	390	62
Depots	30	17	14	4	4	3	5	4	6	2	5	5	83	150	48
Elevator-Learned	147	41	25	5	3	2	4	3	2	2	3	3	47	660	16
Entertainment	12	19	26	7	4	2	7	2	2	4	4	7	45	403	1
Factories-simple	20	17	11	4	3	3	4	3	4	3	4	4	73	98	1
Freecell-Learned	60	283	304	7	15	6	14	8	7	2	8	8	52	177	4
Hiking	30	20	20	9	4	5	8	7	8	3	8	8	44	87	1
Logistics-Learned	80	56	54	3	8	3	4	4	2	2	5	5	55	48	22
Minecraft-Player	20	21	24	6	4	3	4	4	2	3	5	16	752	217363	1
Minecraft-Regular	59	15	14	6	3	2	3	3	2	3	5	16	22150	129799	1
Monroe-Fully-Obs.	20	68	84	6	10	3	6	1	6	3	5	8	91	419	1
Monroe-Partially-Obs.	20	67	83	6	10	3	5	1	6	3	5	8	91	420	1
Multiarm-Blocksworld	74	15	15	4	5	2	54	54	5	2	54	55	58	118	4
Robot	20	6	13	2	4	3	3	3	3	3	3	3	60	97	1
Rover-GTOHP	30	30	16	4	3	3	6	4	4	3	6	6	95	2351	28
Satellite-GTOHP	20	14	12	3	3	2	5	3	3	2	4	4	121	176	79
Snake	20	7	5	3	3	3	28	28	8	3	28	28	29	149	1
Towers	20	7	10	2	3	3	4	3	6	2	5	5	14	111	1
Transport	40	4	6	4	3	2	4	0	4	2	5	5	55	125	26
Woodworking	30	15	51	3	4	4	9	0	7	2	12	12	110	241	22

Table 9.3: Averaged per-domain properties of HDDL benchmarks (after preprocessing), divided into prior benchmarks (see Section 6.6.2) and IPC benchmarks. Left to right: Number of operators (O), methods (M); max. expansion size (X), max. methods per task (B), max. free non-trivial arguments per method (U); number of preconditions of actions (P_a) / reductions (P_r), effects (E); arity of facts (Y_f) / actions (Y_a) / reductions (Y_r); number of constants (C) / initial true facts ($|s_I|$) / initial tasks ($|T|$).

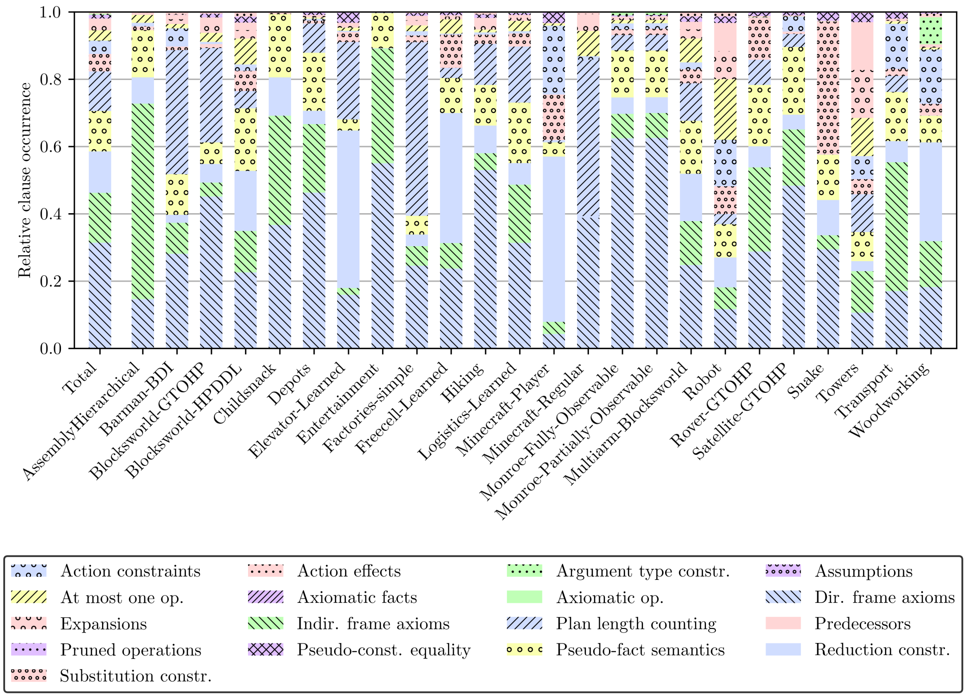


Figure 9.3: Distribution of occurrences of different clause categories encoded by LILOTANEQ on the IPC benchmarks, overall (leftmost column) and per domain. All instances for which LILOTANEQ found some initial plan were considered.

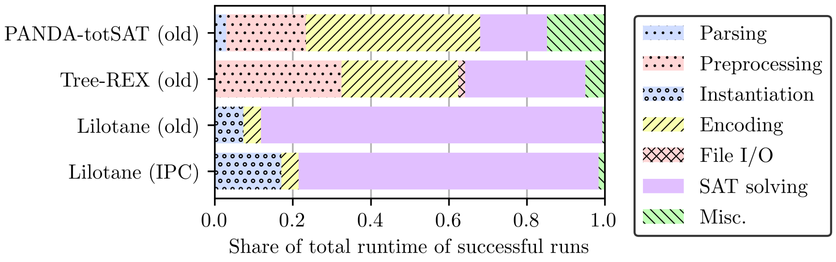


Figure 9.4: Partitioning of running times for three SAT-based planners on prior benchmarks [Sch+19b] and for LILOTANE on IPC benchmarks.

Domain	#	pos.	lay.	cls./10^6	$\frac{\text{act.}}{\text{pos.}}$	$\frac{\text{red.}}{\text{pos.}}$	$\frac{\text{psc.}}{\text{pos.}}$	ret.pr.	pruned	dom.	+prec.
AssemblyH.	5	136.00	14.00	0.43	9.75	1.54	1.94	50.60	50.60	71.80	76.00
Barman	17	702.53	6.00	0.14	2.15	0.54	0.59	0.00	0.00	0.00	2.00
Blocksw-G.	23	8215.70	10.39	0.66	2.22	0.07	0.05	7.48	80.61	26.00	6.00
Blocksw-H.	1	10929.00	13.00	3.34	3.99	3.25	1.25	0.00	0.00	3.00	11.00
Childsnack	29	284.38	2.00	1.67	0.83	0.17	0.66	0.00	0.00	0.00	8.00
Depots	24	1696.58	6.00	1.48	2.59	0.24	0.42	0.96	7.12	25.67	11.00
Elevator	147	3547.09	8.99	1.05	2.23	0.61	0.25	0.01	0.01	0.00	7.00
Entertainm.	4	510.75	7.25	20.31	3.26	2.00	0.95	0.00	0.00	282.00	64.00
Factories	4	4747.75	12.75	0.51	2.01	0.94	0.79	0.00	0.00	21.25	0.00
Freecell	12	3147.08	9.33	13.85	8.85	11.00	3.12	3.75	3.75	3653.75	642.00
Hiking	23	10578.04	12.48	0.78	1.53	0.15	0.32	18.57	3873.09	0.00	7.00
Logistics	45	3483.71	11.91	1.99	4.65	1.29	0.73	0.00	0.00	0.00	41.07
Minecraft-P.	2	777.50	8.50	5.05	4.98	3.07	1.50	19.50	47.50	0.50	5.00
Minecraft-R.	35	10699.46	12.60	1.22	4.25	0.69	0.00	94.46	630.17	0.00	2.00
Monroe-FO.	20	1153.95	7.85	0.84	3.14	0.67	0.60	6.55	28.15	86.45	112.75
Monroe-PO.	20	1811.55	8.00	1.32	3.22	0.64	0.58	4.35	15.95	134.45	89.80
Multiarm-Bl.	4	22465.75	14.50	5.26	4.16	1.90	0.88	0.00	0.00	7.25	11.00
Robot	11	186.73	15.00	0.01	4.25	0.55	0.45	2.91	3.09	0.00	13.00
Rover	23	826.22	4.57	1.44	2.07	0.23	0.30	0.04	0.04	0.00	43.43
Satellite	16	1562.56	7.00	3.12	1.95	0.43	0.31	0.00	0.00	0.00	7.00
Snake	20	244.20	8.55	1.00	2.44	0.56	0.59	0.00	0.00	0.00	7.00
Towers	9	19601.33	119.56	0.30	2.47	0.01	0.02	0.00	0.00	0.00	4.00
Transport	33	393.64	4.64	0.27	2.46	0.29	0.60	0.24	0.24	14.94	13.00
Woodworking	30	249.47	4.63	0.09	1.53	0.56	1.60	4.97	5.10	0.57	126.77

Table 9.4: Statistics overview of all IPC instances solved by LILOTANE. From left to right: Solved instances; created positions / layers / clauses; actions / reductions / pseudo-constants per position; retroactive prunings, operations pruned by these prunings, dominated operations, inferred preconditions. All but the three "per position" measures are arithmetic averages over all solved instances in the domain. Measures for operations and pseudo-constants per position also include the objects which are later removed again due to retroactive prunings or dominated operations.

	Mean clause length				Memory peak (GB)	
	min	med	mean	max	med	mean
PANDA-TOTSAT	2.00	2.31	2.50	7.36	0.515	1.069
TREE-REX	2.46	2.66	3.13	12.89	0.369	0.790
LILOTANE	2.25	2.95	3.15	5.56	0.026	0.128

Table 9.5: Aggregation over TOHTN planner statistics reported per instance.

List of Acronyms

AE	Algorithm Engineering 4, 5
AI	Artificial Intelligence 2, 21, 23, 127, 128
AMQ	Approximate Membership Query 56, 75, 118
API	Application Programming Interface 52, 53
AWS	Amazon Web Services 101, 105, 120
CBS	Count Based Speedup 36, 94, 96
CDCL	Conflict-Driven Clause Learning 15–18, 20, 24, 28, 30, 80, 104
CDF	Cumulative Distribution Function 38, 88, 162, 167
CNF	Conjunctive Normal Form 12–14, 23, 34, 81, 109, 110, 115, 116, 148, 149, 151
CPU	Central Processing Unit 37, 43, 44, 58, 59, 83, 90, 91, 180
CSAR	Commonly Solved Average Runtime 35, 84, 85, 87–90
DAG	Directed Acyclic Graph 15, 32
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science 34, 109
DP	Davis-Putnam 14–16
DPLL	Davis-Putnam-Logemann-Loveland 14–18, 24
DRAT	Delete Reverse Asymmetric Tautology 109, 110, 112, 120, 122, 124, 125, 188
GPU	Graphical Processing Unit 180
HDDL	Hierarchical Domain Description Language 155, 198
HPC	High Performance Computing v, vii, 4, 5, 11, 43, 101, 177, 180, 181
HTN	Hierarchical Task Network xiii, 127–133, 135, 154, 155, 159, 179
IPASIR	Re-entrant Incremental Satisfiability Application Program Interface 23, 155, 165, 168, 180
IPC	International Planning Competition 6, 128, 129, 131, 135, 159–163, 166–169, 171, 179, 198–200
IPC	Inter-Process Communication 82, 168
ISC	International SAT Competition 2, 25, 27, 31, 32, 37–39, 59, 66, 67, 79, 81, 83, 84, 90, 94, 95, 97–104, 108, 120, 123, 186

LBD	Literal Block Distance xii, 17, 19, 28, 29, 31, 32, 66, 68–70, 73, 74, 78, 79, 84, 87
LRAT	Linear Reverse Asymmetric Tautology 108–111, 115–120, 122, 124, 125, 179
MPI	Message Passing Interface 10, 32, 52–54, 58, 69, 82–84, 100, 101, 169
NUMA	Non-Uniform Memory Access 9
OOS	Optimal Offline Schedule 99, 100
PAR	Penalized Average Runtime 34, 35, 39, 83–91, 95–99, 102, 104, 121
QBF	Quantified Boolean Formula 7, 22–24, 136, 178, 179
RAM	Random Access Memory 9, 67, 82, 84, 85, 87, 101, 120, 155, 161
SAT	Propositional Satisfiability v–viii, xi–xiv, 1–7, 9, 12–35, 37–39, 41–44, 48, 52, 55–59, 63, 65–70, 72, 74, 76, 78–86, 88, 90–92, 94–98, 100–105, 107–110, 112, 114, 116, 118, 120–122, 124, 125, 127–130, 132, 134–136, 138, 140, 142, 144, 146–148, 150–152, 154–156, 158–160, 162–175, 177–180, 199
SMT	Satisfiability Modulo Theories 7, 22, 178, 179
TOHTN	Totally Ordered Hierarchical Task Network vi, viii, xiii, 6, 7, 56, 128–132, 135, 136, 145, 152, 154, 155, 163–166, 177–179, 193, 194, 200
VBS	Virtual Best Solver 26

Publications and Supervised Theses

In Conference Proceedings

Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomáš Balyo. “Efficient SAT Encodings for Hierarchical Planning”. In: *Proc. ICAART*. 2019, pp. 531–538. DOI: `10.5220/0007343305310538`

Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomáš Balyo. “Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning”. In: *Proc. ICAPS*. 2019, pp. 382–390. DOI: `10.1609/icaps.v29i1.3502`

Nils Froleyks, Tomáš Balyo, and Dominik Schreiber. “PASAR—Planning as Satisfiability with Abstraction Refinement”. In: *Proc. SoCS*. 2019, pp. 70–78. URL: `https://ojs.aaai.org/index.php/SOCS/article/download/18504/18295`

Kai Fieger, Tomas Balyo, Christian Schulz, and Dominik Schreiber. “Finding optimal longest paths by dynamic programming in parallel”. In: *Proc. SoCS*. 2019, pp. 61–69. URL: `https://ojs.aaai.org/index.php/SOCS/article/download/18503/18294/22019`

Dominik Schreiber and Peter Sanders. “Scalable SAT Solving in the Cloud”. In: *Proc. SAT*. Springer. 2021, pp. 518–534. DOI: `10.1007/978-3-030-80223-3_35`

Peter Sanders and Dominik Schreiber. “Decentralized online scheduling of malleable NP-hard jobs”. In: *Proc. Euro-Par*. Springer. 2022, pp. 119–135. DOI: `10.1007/978-3-031-12597-3_8`. **Nominated for a Best Paper Award.**

Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, Benjamin Kiesl-Reiter, and Michael W. Whalen. “Unsatisfiability proofs for distributed clause-sharing SAT solvers”. In: *Proc. TACAS*. Springer. 2023, pp. 348–366. DOI: `10.1007/978-3-031-30823-9_18`. **Nominated for a Best Paper Award.**

Journal Articles

Dominik Schreiber. “Lilotane: A Lifted SAT-Based Approach to Hierarchical Planning”. In: *JAIR* 70 (2021), pp. 1117–1181. DOI: `10.1613/jair.1.12520`

Peter Sanders and Dominik Schreiber. “Mallob: Scalable SAT Solving On Demand With Decentralized Job Scheduling”. In: *JOSS* 7.76 (2022), p. 4591. DOI: `10.21105/joss.04591`

Technical Reports

Nils Froleyks, Tomáš Balyo, and Dominik Schreiber. "PASAR Entering the Sparkle Planning Challenge 2019". In: *Proc. Sparkle Planning Challenge 2019.* 2019

Dominik Schreiber. "Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track". In: *Proc. SAT Competition.* 2020, pp. 45–46

Dominik Schreiber. "Lifted logic for task networks: TOHTN planner Lilotane in the IPC 2020". In: *Proc. International Planning Competition.* 2021, pp. 9–12

Malte Sönnichsen and Dominik Schreiber. "The "Factories" HTN Domain". In: *Proc. International Planning Competition.* 2021, pp. 45–46

Dominik Schreiber. "Mallob in the SAT Competition 2021". In: *Proc. SAT Competition.* 2021, pp. 38–39

Dominik Schreiber. "Mallob in the SAT Competition 2022". In: *Proc. SAT Competition.* 2022, pp. 46–47

Dominik Schreiber. "Mallob{32,64,1600} in the SAT Competition 2023". In: *Proc. SAT Competition.* 2023, pp. 46–47

Theses

Dominik Schreiber. "Energieeffiziente Ausführung von qualitätsbewussten Algorithmen für Mobile Simulationen". Bachelor's thesis. Universität Stuttgart, 2016

Dominik Schreiber. "Hierarchical task network planning using SAT techniques". Master's thesis. Grenoble Institut National Polytechnique (INP) and Karlsruhe Institute of Technology (KIT), 2018. DOI: `10.5445/IR/1000104165`

Supervised Theses

Samuel Born. "Sharing Clauses Across Different Problems in Distributed SAT Solving". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2023

Michael Dörr. "K-Means in a Malleable Distributed Environment". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2022

Nils Froleyks. "PASAR—Planning as Satisfiability with Abstraction Refinement". Master's thesis. Karlsruhe Institute of Technology (KIT), 2020. **Faculty Prize for best student thesis.**

Tan Grumser. "Engineering Optimal Solvers for Rubik's Cubes". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2022

Jean-Pierre von der Heydt. "Cube&Conquer-inspired Techniques for Parallel Automated Planning". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2019

Jens Manig. "Kompressionstechniken für Beschreibungen von SAT Formeln". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2019

Ha Phuong Nguyen. “An Empirical Study on Clause Selection and Filtering in Distributed SAT Solving”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2023

Maximilian Schick. “Cube&Conquer-inspired Malleable Distributed SAT Solving”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2021

Nikolai Schnell. “Pruning Techniques for Lifted SAT-based Hierarchical Planning”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2021

Malte Sönnichsen. “Asynchronous Clause Exchange for Malleable SAT Solving”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2021

Niko Wilhelm. “Malleable Distributed Hierarchical Planning”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2022

Marvin Williams. “Partially Instantiated Representations for Automated Planning”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2020

Bibliography

All URLs have last been accessed September 20, 2023.

[ABA15] Ron Alford, Pascal Bercher, and David Aha. "Tight bounds for HTN planning". In: *Proc. ICAPS*. 2015, pp. 7–15. DOI: 10.1609/icaps.v25i1.13721. [see pages 56, 129, 132, 165]

[AC12] Alejandro Arbelaez and Philippe Codognet. "Massively parallel local search for SAT". In: *Proc. ICTAI*. IEEE. 2012, pp. 57–64. [see pages 32, 34]

[AHL08] Roland Axelsson, Keijo Heljanko, and Martin Lange. "Analyzing context-free grammars using an incremental SAT solver". In: *Proc. ICALP*. Springer. 2008, pp. 410–422. DOI: 10.1007/978-3-540-70583-3_34. [see pages 22, 23, 168]

[Aig+13] Martin Aigner, Armin Biere, Christoph M. Kirsch, et al. "Analysis of Portfolio-Style Parallel SAT Solving on Current Multi-Core Architectures." In: *Proc. Pragmatics of SAT*. 2013, pp. 28–40. DOI: 10.29007/73n4. [see pages 26, 32]

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. "Sorting in $\log n$ Parallel Steps". In: *Combinatorica* 3.1 (1983), pp. 1–19: Springer. DOI: 10.1109/tc.1985.5009385. [see page 47]

[Alo+09] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. "NP-hardness of Euclidean sum-of-squares clustering". In: *Machine learning* 75 (2009), pp. 245–248: Springer. DOI: 10.1007/s10994-009-5103-0. [see page 57]

[Alq+18] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. "An Analysis of Network-Partitioning Failures in Cloud Systems". In: *Proc. Symp. Operating Systems Design and Implementation*. 2018, pp. 51–68. [see page 43]

[ALS13] Gilles Audemard, Jean-Marie Lagniez, and Laurent Simon. "Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction". In: *Proc. SAT*. Springer. 2013, pp. 309–317. DOI: 10.1007/978-3-642-39071-5_23. [see page 23]

[AS09] Gilles Audemard and Laurent Simon. "Predicting learnt clauses quality in modern SAT solvers". In: *Proc. IJCAI*. 2009, pp. 399–404. [see pages 2, 17, 28, 32, 38, 39, 78–80, 155]

[AS12] Gilles Audemard and Laurent Simon. "Refining restarts strategies for SAT and UNSAT". In: *Proc. CP*. Springer. 2012, pp. 118–126. DOI: 10.1007/978-3-642-33558-7_11. [see pages 16, 17, 19]

[AS14] Gilles Audemard and Laurent Simon. “Lazy clause exchange policy for parallel SAT solvers”. In: *Proc. SAT*. Springer. 2014, pp. 197–205. DOI: 10.1007/978-3-319-09284-3_15. [see pages 29, 74, 79, 80, 82]

[AS17a] Gilles Audemard and Laurent Simon. “Glucose and Syrup in the SAT'17”. In: *Proc. SAT Competition*. 2017, pp. 16–17. [see pages 27, 29, 32]

[AS17b] Michael Axtmann and Peter Sanders. “Robust Massively Parallel Sorting”. In: *Proc. ALENEX*. 2017, pp. 83–97. DOI: 10.1137/1.9781611974768.7. [see page 47]

[Aud+12] Gilles Audemard, Benoît Hoessen, Said Jabbour, et al. “Revisiting clause exchange in parallel SAT solving”. In: *Proc. SAT*. Springer. 2012, pp. 200–213. DOI: 10.1007/978-3-642-31612-8_16. [see page 28]

[Aud+14] Gilles Audemard, Benoît Hoessen, Said Jabbour, and Cédric Piette. “Dolius: A Distributed Parallel SAT Solving Framework.” In: *Proc. Pragmatics of SAT*. Citeseer. 2014, pp. 1–11. DOI: 10.29007/hvqt. [see page 31]

[Aud+16] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. “An adaptive parallel SAT solver”. In: *Proc. CP*. Springer. 2016, pp. 30–48. DOI: 10.1007/978-3-319-44953-1_3. [see pages 25, 27, 29]

[Aud+17] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. “A distributed version of syrup”. In: *Proc. SAT*. Springer. 2017, pp. 215–232. DOI: 10.1007/978-3-319-66263-3_14. [see pages 27, 29, 32, 67, 68, 70, 89]

[BA05] Nate Blaylock and James Allen. “Generating artificial corpora for plan recognition”. In: *Int. Conf. User Modeling*. Springer. 2005, pp. 179–188. DOI: 10.1007/11527886_24. [see page 160]

[BAH19] Pascal Bercher, Ron Alford, and Daniel Höller. “A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations”. In: *Proc. IJCAI*. 2019, pp. 6267–6275. DOI: 10.24963/ijcai.2019/875. [see pages 6, 128, 135]

[Bal+15] Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. “Overview and analysis of the SAT Challenge 2012 solver competition”. In: *Artificial Intelligence* 223 (2015), pp. 120–155: Elsevier. DOI: 10.1016/j.artint.2015.01.002. [see page 39]

[Bal+16] Tomáš Balyo, Armin Biere, Markus Iser, and Carsten Sinz. “SAT race 2015”. In: *Artificial Intelligence* 241 (2016), pp. 45–65: Elsevier. DOI: 10.1016/j.artint.2016.08.007. [see pages 23, 27]

[Bal+20a] Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, et al. *SAT Competition*. Accessed: 2021-03-19. 2020. URL: https://satcompetition.github.io/2020/. [see page 101]

[Bal+20b] Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, et al. *The Results of SAT Competition 2020*. 2020. URL: https://satcompetition.github.io/2020/downloads/satcomp20slides.pdf. [see pages 101, 102]

[Bal+21a] Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, et al., eds. *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, 2021. [see page 84]

[Bal+21b] Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, et al. *The Results of SAT Competition 2021.* 2021. URL: `https://satcompetition.github.io/2021/slides/ISC2021-fixed.pdf`. [see page 102]

[Bal+22a] Tomáš Balyo, Marijn J. H. Heule, Markus Iser, et al., eds. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions.* Department of Computer Science, University of Helsinki, 2022. [see page 84]

[Bal+22b] Tomáš Balyo, Marijn J. H. Heule, Markus Iser, et al. *The Results of SAT Competition 2022.* 2022. URL: `https://satcompetition.github.io/2022/slides/satcomp22slides.pdf`. [see pages 102, 103]

[Bal+23a] Tomáš Balyo, Nils Froleyks, Marijn J. H. Heule, et al. *SAT Competition.* Accessed: 2023-08-04. 2023. URL: `https://satcompetition.github.io/2023/`. [see pages 2, 31, 102, 180]

[Bal+23b] Tomáš Balyo, Marijn J. H. Heule, Markus Iser, et al. *The Results of SAT Competition 2023.* 2023. URL: `https://satcompetition.github.io/2023/downloads/satcomp23slides.pdf`. [see pages 30, 104]

[BBH20] Gregor Behnke, Pascal Bercher, and Daniel Höller. *Plan Verification.* 2020. URL: `http://gki.informatik.uni-freiburg.de/ipc2020/format.pdf`. [see page 131]

[Beh+20] Gregor Behnke, Daniel Höller, Alexander Schmid, et al. "On Succinct Groundings of HTN Planning Problems". In: *Proc. AAAI.* 2020, pp. 9775–9784. DOI: `10.1609/aaai.v34i06.6529`. [see pages 128, 132, 133, 135, 155, 156]

[Beh+22] Gregor Behnke, Florian Pollitt, Daniel Höller, et al. "Making translations to classical planning competitive with other HTN planners". In: *Proc. AAAI.* 2022, pp. 9687–9697. DOI: `10.1609/aaai.v36i9.21203`. [see page 179]

[Beh21] Gregor Behnke. "Block compression and invariant pruning for SAT-based totally-ordered HTN planning". In: *Proc. ICAPS.* 2021, pp. 25–35. DOI: `10.1609/icaps.v31i1.15943`. [see page 179]

[Bel+14] Anton Belov, Daniel Diepold, Marijn J. H. Heule, and Matti Järvisalo. *SAT Competition.* 2014. URL: `http://satcompetition.org/2014/index.shtml`. [see page 25]

[Ber+00] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. "Balanced allocations: The heavily loaded case". In: *Proc. ACM Symp. on Theory of computing.* 2000, pp. 745–754. DOI: `10.1145/335305.335411`. [see page 50]

[Bev+15] Giuseppe Bevacqua, Jonathan Cacace, Alberto Finzi, and Vincenzo Lippiello. "Mixed-initiative planning and execution for multiple drones in search and rescue missions". In: *Proc. ICAPS.* 2015, pp. 315–323. DOI: `10.1609/icaps.v25i1.13700`. [see page 128]

[Bey+21] Olaf Beyersdorff, Mikoláš Janota, Florian Lonsing, and Martina Seidl. "Quantified boolean formulas". In: *Handbook of Satisfiability.* IOS Press, 2021, pp. 1177–1221. DOI: `10.3233/faia200987`. [see pages 23, 24]

[BF15] Armin Biere and Andreas Fröhlich. "Evaluating CDCL variable scoring schemes". In: *Proc. SAT.* Springer. 2015, pp. 405–422. DOI: `10.1007/978-3-319-24318-4_29`. [see page 18]

[BF20] Armin Biere and Mathias Fleury. “Chasing target phases”. In: *Proc. Pragmatics of SAT*. 2020. [see page 18]

[BF22a] Tomáš Balyo and Nils Froleyks. “AI Assisted Design of Sokoban Puzzles Using Automated Planning”. In: *Proc. ArtsIT*. Springer. 2022, pp. 424–441. DOI: 10.1007/978-3-030-95531-1_29. [see page 127]

[BF22b] Armin Biere and Mathias Fleury. “Gimsatul, IsaSAT, Kissat Entering the SAT Competition 2022”. In: *Proc. SAT Competition*. 2022, pp. 10–11. [see pages 3, 30, 33, 108, 120]

[BFH21] Armin Biere, Mathias Fleury, and Maximilian Heisinger. “CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021”. In: *Proc. SAT Competition*. 2021, pp. 10–12. [see page 15]

[BFP23] Armin Biere, Mathias Fleury, and Florian Pollitt. “CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT Entering the SAT Competition 2023”. In: *Proc. SAT Competition*. 2023, p. 14. [see page 120]

[BG20] Blai Bonet and Hector Geffner. “Learning First-Order Symbolic Representations for Planning from the Structure of the State Space”. In: *Proc. ECAI*. 2020. [see page 136]

[BGP17] Felix Brandt, Christian Geist, and Dominik Peters. “Optimal bounds for the no-show paradox via SAT solving”. In: *Mathematical Social Sciences* 90 (2017), pp. 18–27: Elsevier. DOI: 10.1016/j.mathsocsci.2016.09.003. [see page 21]

[BH19] Armin Biere and Marijn J. H. Heule. “The Effect of Scrambling CNFs”. In: *Proc. Pragmatics of SAT*. 2019, pp. 111–126. [see pages 39, 81]

[BHB18] Gregor Behnke, Daniel Höller, and Susanne Biundo. “totSAT – Totally-ordered hierarchical planning through SAT”. In: *Proc. AAAI*. 2018, pp. 6110–6118. DOI: 10.1609/aaai.v32i1.12083. [see pages 128, 135, 155]

[BHB19a] Gregor Behnke, Daniel Höller, and Susanne Biundo. “Bringing order to chaos – A compact representation of partial order in SAT-based HTN planning”. In: *Proc. AAAI*. 2019, pp. 7520–7529. DOI: 10.1609/aaai.v33i01.33017520. [see pages 128, 135]

[BHB19b] Gregor Behnke, Daniel Höller, and Susanne Biundo. “Finding Optimal Solutions in HTN Planning – A SAT-based Approach”. In: *Proc. IJCAI*. 2019, pp. 5500–5508. DOI: 10.24963/ijcai.2019/764. [see pages 128, 135, 154, 155]

[BHB21] Gregor Behnke, Daniel Höller, and Pascal Bercher, eds. *Proceedings of the 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*. 2021. URL: https://ipc2020.hierarchical-task.net/publications/IPC2020Booklet.pdf. [see pages 128, 129, 135, 159]

[BHJ17] Tomás Balyo, Marijn J. H. Heule, and Matti Jarvisalo. “SAT Competition 2016: Recent developments”. In: *Proc. AAAI*. 2017. DOI: 10.1609/aaai.v31i1.10641. [see page 27]

[BIB22] Jakob Bach, Markus Iser, and Klemens Böhm. “A Comprehensive Study of k-Portfolios of Recent SAT Solvers”. In: *Proc. Pragmatics of SAT*. 2022. [see pages 27, 34, 79, 105]

[Bie+20a] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020". In: *Proc. SAT Competition*. 2020, p. 50. [see pages 2, 16, 17, 19, 29, 78–80, 108, 118]

[Bie+20b] Armin Biere, Matti Järvisalo, Daniel Le Berre, et al. *The SAT Practitioner's Manifesto*. Version 1.0. 2020. DOI: `10.5281/zenodo.4500928`. [see page 103]

[Bie+21] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*. 2nd ed. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021. DOI: `10.3233/faia336`. [see page 12]

[Bie+22] Armin Biere, Md Solimul Chowdhury, Marijn J. H. Heule, et al. "Migrating Solver State". In: *Proc. SAT*. 2022. DOI: `10.4230/LIPIcs.SAT.2022.27`. [see pages 33, 44]

[Bie+99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. "Symbolic model checking without BDDs". In: *Proc. TACAS*. Springer. 1999, pp. 193–207. DOI: `10.1007/3-540-49059-0_14`. [see page 22]

[Bie10] Armin Biere. "Lingeling, Plingeling, Picosat and Precosat at SAT race 2010". In: *Proc. SAT Competition*. 2010. [see pages 3, 17, 27–29, 79]

[Bie12] Armin Biere. "Lingeling and friends entering the SAT Challenge 2012". In: *Proc. SAT Challenge*. 2012, pp. 33–34. [see pages 25, 79, 83]

[Bie13] Armin Biere. "Lingeling, Plingeling and Treengeling entering the SAT competition 2013". In: *Proc. SAT Competition*. 2013, p. 1. [see pages 28, 29, 79, 80]

[Bie14] Armin Biere. "Yet another local search solver and Lingeling and friends entering the SAT Competition 2014". In: *Proc. SAT Competition*. 2014, p. 65. [see pages 25, 29, 79]

[Bie15] Armin Biere. "Lingeling and Friends Entering the SAT Race 2015". In: *FMV Reports Series*. 2015. DOI: `10.35011/fmvtr.2015-2`. [see page 79]

[Bie16a] Armin Biere. "Collection of combinational arithmetic miters submitted to the SAT Competition 2016". In: *Proc. SAT Competition*. 2016, pp. 65–66. [see page 2]

[Bie16b] Armin Biere. "Splatz, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2016". In: *Proc. SAT Competition*. 2016, pp. 44–45. [see pages 3, 82]

[Bie17] Armin Biere. "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT Entering the SAT Competition 2017". In: *Proc. SAT Competition*. 2017. [see pages 17, 80, 155]

[Bie18] Armin Biere. "CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2018". In: *Proc. SAT Competition*. 2018, pp. 14–15. [see pages 16, 18, 19, 79, 80, 83]

[Bie19] Armin Biere. "CaDiCaL at the SAT race 2019". In: *Proc. SAT Race*. 2019, pp. 8–9. [see pages 18, 19]

[Bie21] Armin Biere. "CNF Encodings of Complete Pairwise Combinatorial Testing of our SAT Solver SATCH". In: *Proc. SAT Competition*. 2021, p. 46. [see pages 21, 94, 108]

[Bie22] Armin Biere. *Kissat SAT Solver*. 2022. URL: `http://fmv.jku.at/kissat/`. [see page 38]

[Bir+15] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, et al. "Intel® omni-path architecture: Enabling scalable, high performance fabrics". In: *Proc. IEEE Symp. High-Performance Interconnects*. IEEE. 2015, pp. 1–9. DOI: `10.1109/HOTI.2015.22`. [see pages 10, 58]

[BJK21] Armin Biere, Matti Järvisalo, and Benjamin Kiesl. "Preprocessing in SAT Solving". In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 391–435. DOI: `10.3233/faia200987`. [see page 19]

[BK92] Michael Buro and Hans Kleine-Büning. *Report on a SAT competition*. Fachbereich Math.-Informatik, Univ. Gesamthochschule Paderborn, Germany, 1992. [see page 37]

[BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. "Towards understanding and harnessing the potential of clause learning". In: *JAIR* 22 (2004), pp. 319–351. DOI: `10.1613/jair.1410`. [see page 16]

[BL16] Tomáš Balyo and Florian Lonsing. "HordeQBF: A modular and massively parallel QBF solver". In: *Proc. SAT*. Springer. 2016, pp. 531–538. DOI: `10.1007/978-3-319-40970-2_33`. [see page 178]

[BL99] Robert D. Blumofe and Charles E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *J. ACM* 46.5 (1999), pp. 720–748. DOI: `10.1145/324133.324234`. [see page 25]

[Bla+04] Jacek Blazewicz, Maciej Machowiak, Jan Weglarz, et al. "Scheduling malleable tasks on parallel processors to minimize the makespan". In: *Annals of Operations Research* 129.1-4 (2004), p. 65: Springer. DOI: `10.1023/b:anor.0000030682.25673.c0`. [see pages 43, 179]

[Bla+06] Jacek Blazewicz, Mikhail Y. Kovalyov, Maciej Machowiak, et al. "Preemptable malleable task scheduling problem". In: *IEEE Transactions on Computers* 55.4 (2006), pp. 486–490: IEEE. DOI: `10.1109/tc.2006.58`. [see page 43]

[Bla+19] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, et al. *Handbook on scheduling*. Springer, 2019. DOI: `10.1007/978-3-319-99849-7`. [see page 43]

[BLB10] Robert Brummayer, Florian Lonsing, and Armin Biere. "Automated testing and debugging of SAT and QBF solvers". In: *Proc. SAT*. Springer. 2010, pp. 44–57. DOI: `10.1007/978-3-642-14186-7_6`. [see page 108]

[Ble96] Guy E. Blelloch. "Programming parallel algorithms". In: *Comm. ACM* 39.3 (1996), pp. 85–97: ACM. DOI: `10.1145/227234.227246`. [see page 11]

[Blo70] Burton H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Comm. ACM* 13.7 (1970), pp. 422–426: ACM. DOI: `10.1145/362686.362692`. [see pages 31, 75, 118]

[Bog+23] Bart Bogaerts, Jakob Nordström, Andy Oertel, and Cagrı Uluç Yıldırımoglu. "Crafted Benchmark Formulas Requiring Symmetry Breaking and/or Parity Reasoning". In: *Proc. SAT Competition*. 2023, p. 67. [see page 104]

[Boo47] George Boole. *The mathematical analysis of logic*. Philosophical Library, 1847. [see page 1]

[Bor23] Samuel Born. “Sharing Clauses Across Different Problems in Distributed SAT Solving”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2023. [see page 204]

[BS18] Tomáš Balyo and Carsten Sinz. “Parallel Satisfiability”. In: *Handbook of Parallel Constraint Reasoning*. Ed. by Youssef Hamadi and Lakhdar Sais. Springer, 2018. Chap. 1. DOI: 10.1007/978-3-319-63516-3_1. [see pages 3, 24, 26, 28, 29, 31, 35, 37]

[BS21] Gregor Behnke and David Speck. “Symbolic search for optimal total-order HTN planning”. In: *Proc. AAAI*. 2021, pp. 11744–11754. DOI: 10.1609/aaai.v35i13.17396. [see page 179]

[BS96] Max Böhm and Ewald Speckenmeyer. “A fast parallel SAT-solver – Efficient workload balancing”. In: *Annals of Mathematics and Artificial Intelligence* 17 (1996), pp. 381–400: Springer. DOI: 10.1007/bf02127976. [see pages 24, 31]

[BSA23] Gregor Behnke, Dominik Schreiber, and Ron Alford. *IPC 2023 – HTN Tracks*. 2023. URL: https://ipc2023-htn.github.io/results-presentation.pdf. [see page 179]

[BSK03] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. “Parallel propositional satisfiability checking with distributed dynamic learning”. In: *Parallel Computing* 29.7 (2003), pp. 969–994: Elsevier. DOI: 10.1016/s0167-8191(03)00068-1. [see page 25]

[BSS15] Tomáš Balyo, Peter Sanders, and Carsten Sinz. “Hordesat: A massively parallel portfolio SAT solver”. In: *Proc. SAT*. Springer. 2015, pp. 156–172. DOI: 10.1007/978-3-319-24318-4_12. [see pages 4–6, 26–29, 31, 32, 35–37, 39, 66–68, 74, 75, 81, 87, 91, 95, 108, 124]

[Bui+07] Jérémy Buisson, Ozan Sonmez, Hashim Mohamed, et al. “Scheduling malleable applications in multicluster systems”. In: *Proc. Cluster Computing*. IEEE. 2007, pp. 372–381. DOI: 10.1109/clustr.2007.4629252. [see page 43]

[Bur+22] Mark Alexander Burgess, Charles Gretton, Josh Milthorpe, et al. “Dagster: Parallel Structured Search with Case Studies”. In: *Pacific Rim Int. Conf. Artificial Intelligence*. Springer. 2022, pp. 75–89. DOI: 10.1007/978-3-031-20862-1_6. [see pages 32, 65]

[BWS22] Colin Bretl, Niko Wilhelm, and Dominik Schreiber. “Parallel and Distributed TOHTN Planning”. Student research project. 2022. [see pages 164, 165, 169]

[Byl94] Tom Bylander. “The computational complexity of propositional STRIPS planning”. In: *Artificial Intelligence* 69.1-2 (1994), pp. 165–204. DOI: 10.1016/0004-3702(94)90081-7. [see pages 21, 132]

[Cai+22] Shaowei Cai, Xindi Zhang, Mathias Fleury, and Armin Biere. “Better decision heuristics in CDCL through local search and target phases”. In: *JAIR* 74 (2022), pp. 1515–1563 . [see pages 2, 20]

[CF05] Colin Cooper and Alan Frieze. “The cover time of random regular graphs”. In: *J. Discrete Mathematics* 18.4 (2005), pp. 728–740: SIAM. DOI: 10.1137/s0895480103428478. [see page 49]

[CFG13] Michael Cashmore, Maria Fox, and Enrico Giunchiglia. "Partially grounded planning as quantified Boolean formula". In: *Proc. ICAPS*. 2013, pp. 29–36. DOI: `10.1609/icaps.v23i1.13549`. [see pages 23, 136]

[Cho23] Md Solimul Chowdhury. "kissat-hywalk-gb, kissat-hywalk-exp, kissat-hywalk-exp-gb, and malloblin Entering the SAT Competition-2023". In: *Proc. SAT Competition*. 2023, p. 28. [see pages 105, 178]

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A tool for checking ANSI-C programs". In: *Proc. TACAS*. Springer. 2004, pp. 168–176. DOI: `10.1007/978-3-540-24730-2_15`. [see page 88]

[CL99] Miguel Castro and Barbara Liskov. "Practical byzantine fault tolerance". In: *Proc. Symp. Operating Systems Design and Implementation*. 1999, pp. 173–186. [see page 181]

[Cla+01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. "Bounded model checking using satisfiability solving". In: *Formal methods in system design* 19 (2001), pp. 7–34: Springer. DOI: `10.1023/A:1011276507260`. [see pages 2, 21, 22, 108]

[Col+12] Amanda Coles, Andrew Coles, Angel García Olaya, et al. "A survey of the seventh international planning competition". In: *AI Magazine* 33.1 (2012), pp. 83–88. DOI: `10.1609/aimag.v33i1.2392`. [see page 163]

[Coo00] Stephen Cook. "The P versus NP problem". In: *Clay Mathematics Institute* 2 (2000) . [see pages 1, 13]

[Coo21] Byron Cook. *Automated reasoning's scientific frontiers*. 2021. URL: `https://www.amazon.science/blog/automated-reasonings-scientific-frontiers`. [see pages 65, 103, 107, 178]

[Coo71] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proc. ACM Symp. on Theory of computing*. 1971, pp. 151–158. DOI: `10.1145/800157.805047`. [see pages 1, 13]

[Cor+20] Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Frances. "Lifted Successor Generation Using Query Optimization Techniques". In: *Proc. ICAPS*. 2020, pp. 80–89. DOI: `10.1609/icaps.v30i1.6648`. [see page 140]

[CRB23] Cayden R. Codel, Joseph E. Reeves, and Randal E. Bryant. "Pigeon Hole and Mutilated Chessboard with Mixed Constraint Encodings and Symmetry-Breaking". In: *Proc. SAT Competition*. 2023, p. 72. [see page 104]

[Cru+17] Luis Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt Jr., et al. "Efficient Certified RAT Verification". In: *Proc. CADE*. 2017, pp. 220–236. DOI: `10.1007/978-3-319-63046-5_14`. [see pages 21, 108, 109]

[CV11] José Coelho and Mario Vanhoucke. "Multi-mode resource-constrained project scheduling using RCPSP and SAT solvers". In: *European Journal of Operational Research* 213.1 (2011), pp. 73–82: Elsevier. DOI: `10.1016/j.ejor.2011.03.019`. [see page 22]

[CW03] Wahid Chrabakh and Rich Wolski. "GrADSAT: A parallel SAT solver for the grid". In: *Proc. IEEE SC03*. 2003. [see pages 3, 25]

[Dar20] Adnan Darwiche. "Three modern roles for logic in AI". In: *Proc. ACM SIGMOD-SIGACT-SIGAI Symp. Principles of Database Systems.* 2020, pp. 229–243. DOI: `10.4204/eptcs.326.0.2`. [see pages 2, 21]

[DEV07] Travis Desell, Kaoutar El Maghraoui, and Carlos A. Varela. "Malleable applications for scalable high performance computing". In: *Cluster Computing* 10.3 (2007), pp. 323–337: Springer. DOI: `10.1007/s10586-007-0032-9`. [see pages 41, 43, 44]

[DHK05] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. "Bounded model checking with QBF". In: *Proc. SAT.* Springer. 2005, pp. 408–414. DOI: `10.1007/11499107_32`. [see page 23]

[DKW08] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), pp. 1165–1178: IEEE. DOI: `10.1109/tcad.2008.923410`. [see page 22]

[DLL62] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Comm. ACM* 5.7 (1962), pp. 394–397: ACM. DOI: `10.1145/368273.368557`. [see page 14]

[DM21] Adnan Darwiche and Pierre Marquis. "On quantifying literals in Boolean logic and its applications to explainable AI". In: *JAIR* 72 (2021), pp. 285–328. DOI: `10.1613/jair.1.12756`. [see page 23]

[Dör22] Michael Dörr. "K-Means in a Malleable Distributed Environment". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2022. [see pages 41, 57, 204]

[DP21] Adnan Darwiche and Knot Pipatsrisawat. "Complete Algorithms". In: *Handbook of Satisfiability.* IOS Press, 2021, pp. 101–132. DOI: `10.3233/faia200987`. [see pages 14–16]

[DP60] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *JACM* 7.3 (1960), pp. 201–215: ACM. DOI: `10.1145/321033.321034`. [see page 14]

[EB05] Niklas Eén and Armin Biere. "Effective preprocessing in SAT through variable and clause elimination". In: *Proc. SAT.* Springer. 2005, pp. 61–75. DOI: `10.1007/11499107_5`. [see page 19]

[Ehl+20] Thorsten Ehlers, Mitja Kulczynski, Dirk Nowotka, and Philipp Sieweck. "TopoSAT 2". In: *Proc. SAT Competition.* 2020, p. 60. [see pages 28, 32]

[EHN94] Kutluhan Erol, James Hendler, and Dana S. Nau. "HTN planning: Complexity and expressivity". In: *Proc. AAAI.* 1994, pp. 1123–1128. [see page 131]

[EHN96] Kutluhan Erol, James Hendler, and Dana S. Nau. "Complexity results for HTN planning". In: *Annals of Mathematics and Artificial Intelligence* 18.1 (1996), pp. 69–93: Springer. DOI: `10.1007/bf02136175`. [see page 129]

[EME22] Johannes Erlacher, Florian Mendel, and Maria Eichlseder. "Bounds for the security of ascon against differential and linear cryptanalysis". In: *IACR Transactions on Symmetric Cryptology* (2022), pp. 64–87. DOI: `10.46586/tosc.v2022.i1.64-87`. [see page 178]

[EMW97] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. “Automatic SAT-compilation of planning problems”. In: *Proc. IJCAI.* 1997, pp. 1169–1176. [see page 136]

[EN16] Jan Elffers and Jakob Nordström. “Documentation of some combinatorial benchmarks”. In: *Proc. SAT Competition.* 2016. [see page 94]

[EN19] Thorsten Ehlers and Dirk Nowotka. “Tuning parallel SAT solvers”. In: *Proc. Pragmatics of SAT.* 2019, pp. 127–143. DOI: 10.29007/z3g2. [see pages 4, 29, 30, 32, 68, 78, 87]

[ENS14] Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. “Communication in massively-parallel SAT solving”. In: *Proc. ICTAI.* IEEE. 2014, pp. 709–716. DOI: 10.1109/ictai.2014.111. [see pages 4, 27, 28, 32, 67, 68]

[ES03] Niklas Eén and Niklas Sörensson. “Temporal induction by incremental SAT solving”. In: *Electronic Notes in Theoretical Computer Science* 89.4 (2003), pp. 543–560: Elsevier. DOI: 10.1016/s1571-0661(05)82542-3. [see pages 22, 23]

[ES04] Niklas Eén and Niklas Sörensson. “An extensible SAT-solver”. In: *Proc. SAT.* Springer. 2004, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37. [see pages 2, 17–19, 79]

[Exp+13] Roberto R. Expósito, Guillermo L. Taboada, Sabela Ramos, et al. “Performance analysis of HPC applications in the cloud”. In: *Future Generation Computer Systems* 29.1 (2013), pp. 218–229: Elsevier. DOI: 10.1016/j.future.2012.06.009. [see page 11]

[Faz+23] Katalin Fazekas, Aina Niemetz, Mathias Preiner, et al. “IPASIR-UP: User Propagators for CDCL”. In: *Proc. SAT.* 2023. DOI: 10.4230/LIPIcs.SAT.2023.8. [see page 180]

[FB21] Mathias Fleury and Armin Biere. “Efficient all-UIP learned clause minimization”. In: *Proc. SAT.* Springer. 2021, pp. 171–187. DOI: 10.1007/978-3-030-80223-3_12. [see page 15]

[FB22] Mathias Fleury and Armin Biere. “Scalable Proof Producing Multi-Threaded SAT Solving with Gimsatul through Sharing instead of Copying Clauses”. In: *Proc. Pragmatics of SAT.* 2022. [see pages 30, 82, 180]

[FBS19a] Katalin Fazekas, Armin Biere, and Christoph Scholl. “Incremental inprocessing in SAT solving”. In: *Proc. SAT.* Springer. 2019, pp. 136–154. DOI: 10.1007/978-3-030-24258-9_9. [see pages 23, 155]

[FBS19b] Nils Froleyks, Tomáš Balyo, and Dominik Schreiber. “PASAR Entering the Sparkle Planning Challenge 2019”. In: *Proc. Sparkle Planning Challenge 2019.* 2019. [see page 204]

[FBS19c] Nils Froleyks, Tomáš Balyo, and Dominik Schreiber. “PASAR—Planning as Satisfiability with Abstraction Refinement”. In: *Proc. SoCS.* 2019, pp. 70–78. URL: https://ojs.aaai.org/index.php/SOCS/article/download/18504/18295. [see pages 23, 203]

[Fei97] Dror G. Feitelson. “Job scheduling in multiprogrammed parallel systems”. In: *IBM Research Report* (1997). Citeseer . [see pages 41, 43]

[FG00] Paolo Ferraris and Enrico Giunchiglia. "Planning as satisfiability in nondeterministic domains". In: *Proc. AAAI/IAAI*. Citeseer. 2000, pp. 748–753. [see page 22]

[FHS20] Johannes K. Fichte, Markus Hecher, and Stefan Szeider. "A time leap challenge for SAT-solving". In: *Proc. CP*. Springer. 2020, pp. 267–285. DOI: `10.1007/978-3-030-58475-7_16`. [see page 38]

[Fic+23] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. "The Silent (R)evolution of SAT". In: *Comm. ACM* 66.6 (2023), pp. 64–72: ACM. DOI: `10.1145/3560469`. [see pages 2, 14]

[Fie+19] Kai Fieger, Tomas Balyo, Christian Schulz, and Dominik Schreiber. "Finding optimal longest paths by dynamic programming in parallel". In: *Proc. SoCS*. 2019, pp. 61–69. URL: `https://ojs.aaai.org/index.php/SOCS/article/download/18503/18294/22019`. [see page 203]

[Fof+22] Ronak Fofaliya, Jim Grundy, Robert Jones, et al. "AWS CBMC Benchmarks". In: *Proc. SAT Competition*. 2022, p. 54. [see page 88]

[Fos+08] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. "Cloud computing and grid computing 360-degree compared". In: *Grid computing environments workshop*. IEEE. 2008, pp. 1–10. [see pages 4, 5]

[Fro+21] Nils Froleyks, Marijn J. H. Heule, Markus Iser, et al. "SAT competition 2020". In: *Artificial Intelligence* 301 (2021), p. 103572: Elsevier. DOI: `10.1016/j.artint.2021.103572`. [see pages 2, 27, 29, 31, 34, 37, 39, 65, 107]

[Fro20a] Nils Froleyks. "PASAR—Planning as Satisfiability with Abstraction Refinement". Master's thesis. Karlsruhe Institute of Technology (KIT), 2020. [see page 204]

[Fro20b] Nils Froleyks. "Planning track benchmarks". In: *Proc. SAT Competition*. 2020, p. 64. [see page 88]

[FS18] Rohan Fossé and Laurent Simon. "On the non-degeneracy of unsatisfiability proof graphs produced by SAT solvers". In: *Proc. CP*. Springer. 2018, pp. 128–143. DOI: `10.1007/978-3-319-98334-9_9`. [see page 30]

[Fuk+97] Alex Fukunaga, Gregg Rabideau, Steve Chien, and David Yan. "Aspen: A framework for automated planning and scheduling of spacecraft control and operations". In: *Proc. Int. Symp. AI, Robotics and Automation in Space*. 1997, pp. 181–187. [see page 127]

[FW86] Philip J. Fleming and John J. Wallace. "How not to lie with statistics: the correct way to summarize benchmark results". In: *Comm. ACM* 29.3 (1986), pp. 218–221: ACM. DOI: `10.1145/5666.5673`. [see page 35]

[GA15] Ilche Georgievski and Marco Aiello. "HTN planning: Overview, comparison, and beyond". In: *Artificial Intelligence* 222 (2015), pp. 124–156: Elsevier. DOI: `10.1016/j.artint.2015.02.002`. [see pages 128, 135]

[Gan+19] Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. "SAT-encodings for treecut width and treedepth". In: *Proc. ALENEX*. SIAM. 2019, pp. 117–129. DOI: `10.1137/1.9781611975499.10`. [see page 3]

[Gao23] Yu Gao. "Kissat_MAB_prop in SAT Competition 2023". In: *Proc. SAT Competition*. 2023, p. 16. [see page 2]

[Gar+13] Javier García, José E. Florez, Álvaro Torralba, et al. "Combining linear programming and automated planning to solve intermodal transportation problems". In: *European Journal of Operational Research* 227.1 (2013), pp. 216–226: Elsevier. DOI: 10.1016/j.ejor.2012.12.018. [see page 127]

[Gar70] Martin Gardner. "The Fantastic Combinations of John Conway's New Solitaire Game "Life"". In: *Sc. Am.* 223 (1970), pp. 20–123 . [see page 97]

[GB17] Stephan Gocht and Tomas Balyo. "Accelerating SAT Based Planning with Incremental SAT Solving". In: *Proc. ICAPS.* 2017, pp. 135–139. DOI: 10.1609/icaps.v27i1.13798. [see pages 22, 23, 134, 168]

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractability: A Guide to the Theory of NP-Completeness.* 1979. DOI: 10.1137/1024022. [see page 23]

[GL96] William Gropp and Ewing Lusk. *User's Guide for mpich, a Portable Implementation of MPI.* 1996. URL: https://ftp.mcs.anl.gov/pub/mpi/old/userguide.pdf. [see page 10]

[Glo+19] Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Nicolas Szczepanski. "An incremental SAT-based approach to the graph colouring problem". In: *Proc. CP.* Springer. 2019, pp. 213–231. DOI: 10.1007/978-3-030-30048-7_13. [see page 23]

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface.* Vol. 1. MIT press, 1999. ISBN: 9780262326605. [see page 52]

[GM13] Kilian Gebhardt and Norbert Manthey. "Parallel variable elimination on CNF formulas". In: *Annual Conference on Artificial Intelligence.* Springer. 2013, pp. 61–73. DOI: 10.1007/978-3-642-40942-4_6. [see page 180]

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice.* Elsevier, 2004. DOI: 10.1016/B978-1-55860-856-6.X5000-5. [see pages 21, 127, 148, 151]

[Gom+00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems". In: *J. Autom. Reason.* 24.1 (2000), pp. 67–100: Springer. DOI: 10.1023/A:1006314320276. [see page 18]

[Gra+06] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, et al. "Open MPI: A high-performance, heterogeneous MPI". In: *IEEE Int. Conf. Cluster Computing.* IEEE. 2006, pp. 1–9. DOI: 10.1109/clustr.2006.311904. [see pages 10, 101]

[Gra+79] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. "Optimization and approximation in deterministic sequencing and scheduling: a survey". In: *Annals of discrete mathematics.* Vol. 5. Elsevier, 1979, pp. 287–326. DOI: 10.1016/S0167-5060(08)70356-X. [see page 22]

[Gri17] Alexander van der Grinten. "Design, implementation and evaluation of a distributed CDCL framework". PhD thesis. Universität zu Köln, 2017. [see page 31]

[Gro+12] Peter Großmann, Steffen Hölldobler, Norbert Manthey, et al. "Solving periodic event scheduling problems with SAT". In: *Proc. IEA/AIE.* Springer. 2012, pp. 166–175. DOI: 10.1007/978-3-642-31087-4_18. [see pages 2, 22]

[Gru22] Tan Grumser. “Engineering Optimal Solvers for Rubik’s Cubes”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2022. [see page 204]

[GS03] Carla P. Gomes and Bart Selman. “Algorithm portfolios.” In: *Artificial Intelligence* 126.1-2 (Nov. 19, 2003), pp. 43–62. DOI: 10.1016/s0004-3702(00)00081-3. [see page 26]

[GT99] Fausto Giunchiglia and Paolo Traverso. “Planning as model checking”. In: *Proc. European Conference on Planning*. Springer. 1999, pp. 1–20. [see page 22]

[Guo+10] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. “Diversification and intensification in parallel SAT solving”. In: *Proc. CP*. Springer. 2010, pp. 252–265. DOI: 10.1007/978-3-642-15396-9_22. [see page 27]

[Gup+14] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kalé. “Towards realizing the potential of malleable jobs”. In: *Proc. HiPC*. IEEE. 2014, pp. 1–10. DOI: 10.1109/hipc.2014.7116905. [see pages 41, 43]

[Gus88] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Comm. ACM* 31.5 (1988), pp. 532–533: ACM. DOI: 10.1145/42411.42415. [see page 12]

[Hak85] Armin Haken. “The intractability of resolution”. In: *Theoretical computer science* 39 (1985), pp. 297–308: Elsevier. DOI: 10.1016/0304-3975(85)90144-6. [see page 20]

[HB22] Daniel Höller and Gregor Behnke. “Encoding lifted classical planning in propositional logic”. In: *Proc. ICAPS*. 2022, pp. 134–144. DOI: 10.1609/icaps.v32i1.19794. [see pages 3, 136]

[HBB12] Andrei Horbach, Thomas Bartsch, and Dirk Briskorn. “Using a SAT-solver to schedule sports leagues”. In: *J. Scheduling* 15 (2012), pp. 117–125: Springer. DOI: 10.1007/s10951-010-0194-9. [see page 22]

[Hei22] Maximilian L. Heisinger. “Paracooba Enters SAT Competition 2022”. In: *Proc. SAT Competition*. 2022, p. 42. [see pages 103, 186]

[Hel09] Malte Helmert. “Concise finite-domain representations for PDDL planning tasks”. In: *Artificial Intelligence* 173.5-6 (2009), pp. 503–535: Elsevier. DOI: 10.1016/j.artint.2008.10.013. [see page 133]

[Heu+11] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. “Cube and conquer: Guiding CDCL SAT solvers by lookaheads”. In: *Haifa Verification Conference*. Springer. 2011, pp. 50–65. DOI: 10.1007/978-3-642-34188-5_8. [see pages 25, 31]

[Heu+17] Marijn J. H. Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. “Efficient, Verified Checking of Propositional Proofs”. In: *Proc. ITP*. 2017, pp. 269–284. DOI: 10.1007/978-3-319-66107-0_18. [see page 109]

[Heu16] Marijn J. H. Heule. “The DRAT format and DRAT-trim checker”. In: *CoRR* abs/1610.06229 (2016). arXiv: 1610.06229. [see pages 4, 21, 109]

[Heu18] Marijn J. H. Heule. “Schur number five”. In: *Proc. AAAI*. 2018. DOI: 10.1609/aaai.v32i1.12209. [see pages 3, 25, 31, 66, 108]

[Heu21a] Marijn J. H. Heule. “Hamiltonian Cycle Instances using the Chinese Remainder Encoding”. In: *Proc. SAT Competition*. 2021, p. 54. [see page 92]

[Heu21b] Marijn J. H. Heule. “Proofs of unsatisfiability”. In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 635–668. DOI: 10.3233/faia200987. [see pages 6, 20]

[HFB20] Maximilian Heisinger, Mathias Fleury, and Armin Biere. “Distributed Cube and Conquer with Paracooba”. In: *Proc. SAT*. Springer. 2020, pp. 114–122. DOI: 10.1007/978-3-030-51825-7_9. [see pages 4, 25, 31, 33, 65]

[HG23] Andrew Haberlandt and Harrison Green. “SBVA-CADICAL and SBVA-KISSAT: Structured Bounded Variable Addition”. In: *Proc. SAT Competition*. 2023, p. 18. [see page 180]

[HGH23] Andrew Haberlandt, Harrison Green, and Marijn J. H. Heule. “Effective Auxiliary Variables via Structured Reencoding”. In: *Proc. SAT*. 2023. DOI: 10.4230/LIPIcs.SAT.2023.11. [see pages 80, 180]

[HHW13] Marijn J. H. Heule, Warren Hunt, and Nathan Wetzler. “Trimming while checking clausal proofs”. In: *Proc. FMCAD*. IEEE. 2013, pp. 181–188. DOI: 10.1109/fmcad.2013.6679408. [see page 112]

[HJS10] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. “ManySAT: a parallel SAT solver”. In: *JSAT* 6.4 (2010), pp. 245–262: IOS Press. DOI: 10.3233/sat190070. [see pages 3, 26–28]

[HJS12] Youssef Hamadi, Said Jabbour, and Jabbour Sais. “Control-based clause sharing in parallel SAT solving”. In: *Autonomous Search* (2012), pp. 245–267: Springer. DOI: 10.1007/978-3-642-21434-9_10. [see pages 27, 28, 69]

[HKM16] Marijn J. H. Heule, Oliver Kullmann, and Victor Marek. “Solving and verifying the boolean pythagorean triples problem via cube-and-conquer”. In: *Proc. SAT*. Springer. 2016, pp. 228–245. DOI: 10.1007/978-3-319-40970-2_15. [see pages 2, 3, 21, 25, 31, 33, 66, 108]

[HLK03] Chao Huang, Orion Lawlor, and Laxmikant V. Kale. “Adaptive MPI”. In: *Proc. International workshop on languages and compilers for parallel computing*. Springer. 2003, pp. 306–322. DOI: 10.1007/978-3-540-24644-2_20. [see page 44]

[HMP14] Marijn J. H. Heule, Norbert Manthey, and Tobias Philipp. “Validating Unsatisfiability Results of Clause Sharing Parallel SAT Solvers.” In: *Proc. Pragmatics of SAT*. 2014, pp. 12–25. DOI: 10.29007/6vwg. [see pages 4, 33, 108, 109, 120, 122]

[HMS10] Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. “Improving resource-unaware SAT solvers”. In: *Proc. LPAR*. Springer. 2010, pp. 519–534. DOI: 10.1007/978-3-642-16242-8_37. [see page 17]

[Höl+18] Daniel Höller, Gregor Behnke, Pascal Bercher, and Susanne Biundo. “Plan and goal recognition as HTN planning”. In: *Proc. ICTAI*. 2018, pp. 466–473. DOI: 10.1109/ictai.2018.00078. [see page 160]

[Höl+20a] Daniel Höller, Gregor Behnke, Pascal Bercher, et al. “HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems”. In: *Proc. AAAI*. 2020, pp. 9883–9891. DOI: 10.1609/aaai.v34i06.6542. [see page 155]

[Höl+20b] Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. “HTN planning as heuristic progression search”. In: *JAIR* 67 (2020), pp. 835–880. DOI: 10.1613/jair.1.11282. [see page 179]

[Höl21] Daniel Höller. “Translating totally ordered HTN planning problems to classical planning problems using regular approximation of context-free languages”. In: *Proc. ICAPS*. 2021, pp. 159–167. DOI: `10.1609/icaps.v31i1.15958`. [see page 179]

[HS00] Holger H. Hoos and Thomas Stützle. “Local search algorithms for SAT: An empirical evaluation”. In: *J. Autom. Reason.* 24.4 (2000), pp. 421–481: Springer. DOI: `10.1023/A:1006350622830`. [see page 20]

[HS18] Youssef Hamadi and Lakhdar Sais. *Handbook of Parallel Constraint Reasoning.* Springer, 2018. DOI: `10.1007/978-3-319-63516-3`. [see page 25]

[Hua07] Jinbo Huang. “The Effect of Restarts on the Efficiency of Clause Learning”. In: *Proc. IJCAI.* 2007, pp. 2318–2323. [see pages 16, 19]

[Hüb+21] Lukas Hübner, Alexey M. Kozlov, Demian Hespe, et al. “Exploring parallel MPI fault tolerance mechanisms for phylogenetic inference with RAxML-NG”. In: *Bioinformatics* 37.22 (2021), pp. 4056–4063: Oxford University Press. DOI: `10.1101/2021.01.15.426773`. [see page 181]

[Hun04] Jan Hungershöfer. “On the combined scheduling of malleable and rigid jobs”. In: *Proc. Symp. Computer Architecture and HPC.* IEEE. 2004, pp. 206–213. DOI: `10.1109/sbac-pad.2004.27`. [see pages 41, 43, 179]

[Hut+14] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. “Algorithm runtime prediction: Methods & evaluation”. In: *Artificial Intelligence* 206 (2014), pp. 79–111. DOI: `10.1016/j.artint.2013.10.003`. [see page 42]

[HvM21] Marijn J. H. Heule and Hans van Maaren. “Look-Ahead Based SAT Solvers”. In: *Handbook of Satisfiability.* IOS Press, 2021, pp. 155–184. DOI: `10.3233/faia200987`. [see page 20]

[HW13] Youssef Hamadi and Christoph Wintersteiger. “Seven challenges in parallel SAT solving”. In: *AI Magazine* 34.2 (2013), pp. 99–99. DOI: `10.1609/aimag.v34i2.2450`. [see pages 30, 180]

[HW79] John A. Hartigan and Manchek A. Wong. “Algorithm AS 136: A k-means clustering algorithm”. In: *Journal of the royal statistical society* 28.1 (1979), pp. 100–108: JSTOR. DOI: `10.2307/2346830`. [see page 57]

[IBS19] Markus Iser, Tomás Balyo, and Carsten Sinz. “Memory efficient parallel SAT solving with inprocessing”. In: *Proc. ICTAI.* IEEE. 2019, pp. 64–70. DOI: `10.1109/ictai.2019.00018`. [see pages 82, 180]

[IMM17] Alexey Ignatiev, António Morgado, and João Marques-Silva. “On tackling the limits of resolution in SAT solving”. In: *Proc. SAT.* Springer. 2017, pp. 164–183. DOI: `10.1007/978-3-319-66263-3_11`. [see page 80]

[Int23] Intel. *Intel® MPI Library.* 2023. URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html`. [see pages 10, 58]

[IS19] Markus Iser and Carsten Sinz. “A problem meta-data library for research in SAT”. In: *Proc. Pragmatics of SAT.* 2019, pp. 144–152. DOI: `10.29007/gdbb`. [see page 92]

[JHB12] Matti Järvisalo, Marijn J.H. Heule, and Armin Biere. "Inprocessing Rules". In: *Proc. IJCAR.* 2012, pp. 355–370. DOI: 10.1007/978-3-642-31365-3_28. [see page 19]

[JK03] Minsoo Jeon and Dongseung Kim. "Parallel merge sort with load balancing". In: *J. Parallel Programming* 31 (2003), pp. 21–33: Springer. DOI: 10.1023/A:1021734202931. [see page 118]

[JLU01] Bernard Jurkowiak, Chu Min Li, and Gil Utard. "Parallelizing Satz using dynamic workload balancing". In: *Electronic Notes in Discrete Mathematics* 9 (2001), pp. 174–189: Elsevier. DOI: 10.1016/s1571-0653(04)00321-x. [see page 26]

[JLU05] Bernard Jurkowiak, Chu Min Li, and Gil Utard. "A parallelization scheme based on work stealing for a class of SAT solvers". In: *J. Autom. Reason.* 34.1 (2005), pp. 73–101: Springer. DOI: 10.1007/s10817-005-1970-7. [see page 25]

[JT96] David S. Johnson and Michael A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993.* Vol. 26. American Mathematical Soc., 1996. [see page 34]

[JW90] Robert G. Jeroslow and Jinchang Wang. "Solving propositional satisfiability problems". In: *Annals of mathematics and Artificial Intelligence* 1.1 (1990), pp. 167–187: Springer. DOI: 10.1007/bf01531077. [see page 18]

[Kat+13] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. "Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers". In: *Proc. AAAI.* 2013, pp. 481–488. DOI: 10.1609/aaai.v27i1.8660. [see page 30]

[Kau+19] Daniela Kaufmann, Manuel Kauers, Armin Biere, and David Cok. "Arithmetic verification problems submitted to the SAT Race 2019". In: *Proc. SAT Race.* 2019. [see page 105]

[KBS19] Marko Kleine-Büning, Tomáš Balyo, and Carsten Sinz. "Using DimSpec for Bounded and Unbounded Software Model Checking". In: *Proc. ICFEM.* Springer. 2019, pp. 19–35. DOI: 10.1007/978-3-030-32409-4_2. [see pages 22, 23, 168, 175, 178]

[Kha+21] Awais Khan, Hyogi Sim, Sudharshan S. Vazhkudai, et al. "An analysis of system balance and architectural trends based on TOP500 supercomputers". In: *Proc. High Performance Computing in Asia-Pacific Region.* 2021, pp. 11–22. DOI: 10.2172/1649132. [see page 180]

[Khu+16] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. "SATenstein: Automatically building local search SAT solvers from components". In: *Artificial Intelligence* 232 (2016), pp. 20–42: Elsevier. DOI: 10.1016/j.artint.2015.11.002. [see page 34]

[Kie+12] Jörg-Uwe Kietz, Floarea Serban, Abraham Bernstein, et al. "Designing KDD-workflows via HTN-planning for intelligent discovery assistance". In: *5th Planning To Learn Workshop WS28 at ECAI 2012.* 2012. [see page 128]

[KK11] Stephan Kottler and Michael Kaufmann. "SArTagnan-a parallel portfolio SAT solver with lockless physical clause sharing". In: *Proc. Pragmatics of SAT.* 2011. [see page 28]

[KMM13] Vladimir Klebanov, Norbert Manthey, and Christian Muise. “SAT-based analysis and quantification of information flow in programs”. In: *Int. Conf. Quantitative Evaluation of Systems.* Springer. 2013, pp. 177–192. DOI: 10.1007/978-3-642-40196-1_16. [see page 22]

[Kos+10] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa. “Solving open job-shop scheduling problems by SAT encoding”. In: *IEICE Transactions on Information and Systems* 93.8 (2010), pp. 2316–2318: The Institute of Electronics, Information and Communication Engineers. DOI: 10.1587/transinf.e93.d.2316. [see page 22]

[KR68] S. Katti and A. Vijaya Rao. *Handbook of the poisson distribution.* 1968. DOI: 10.1080/00401706.1968.10490580. [see page 60]

[KS92] Henry A. Kautz and Bart Selman. “Planning as Satisfiability”. In: *Proc. ECAI.* Citeseer. 1992, pp. 359–363. [see pages 2, 21, 22, 135, 147]

[KS96] Henry Kautz and Bart Selman. “Pushing the envelope: Planning, propositional logic, and stochastic search”. In: *Proc. AAAI.* 1996, pp. 1194–1201. [see page 21]

[KS98] Henry Kautz and Bart Selman. “BLACKBOX: A new approach to the application of theorem proving to problem solving”. In: *AIPS98 workshop on planning as combinatorial search.* 1998, pp. 58–60. [see page 134]

[KSH06] Henry Kautz, Bart Selman, and Joerg Hoffmann. “SatPlan: Planning as satisfiability”. In: *Proc. International Planning Competition.* 2006, p. 156. [see page 134]

[KSS21] Henry A. Kautz, Ashish Sabharwal, and Bart Selman. “Incomplete Algorithms”. In: *Handbook of Satisfiability.* IOS Press, 2021, pp. 185–203. DOI: 10.3233/faia200987. [see page 20]

[KT14] Daniel Kroening and Michael Tautschnig. “CBMC–C Bounded Model Checker: (Competition Contribution)”. In: *Proc. TACAS.* Springer. 2014, pp. 389–391. [see page 178]

[Lam20] Peter Lammich. “Efficient Verified (UN)SAT Certificate Checking”. In: *J. Autom. Reason.* 64.3 (2020), pp. 513–532: Springer. DOI: 10.1007/s10817-019-09525-z. [see page 109]

[Lar+09] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. “Architectural breakdown of end-to-end latency in a TCP/IP network”. In: *J. Parallel Programming* 37 (2009), pp. 556–571: Springer. DOI: 10.1007/s10766-009-0109-6. [see page 10]

[Le +09] Daniel Le Berre, Olivier Roussel, Laurent Simon, et al. *The SAT 2009 competition results.* 2009. URL: http://www.satcompetition.org/2009/sat09comp-slides.pdf. [see page 17]

[Le +17a] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. “painless-maplecomsps”. In: *Proc. SAT Competition.* 2017, p. 26. [see page 29]

[Le +17b] Ludovic Le Frioux, Souheib Baarir, Julien Sopena, and Fabrice Kordon. “PaInleSS: a framework for parallel SAT solving”. In: *Proc. SAT.* Springer. 2017, pp. 233–250. DOI: 10.1007/978-3-319-66263-3_15. [see page 29]

[Lei+20] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, et al. “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” In: *Science* 368.6495 (2020), eaam9744: American Association for the Advancement of Science. DOI: 10.1126/science.aam9744. [see page 3]

[Li+13] Yinan Li, Ippokratis Pandis, Rene Mueller, et al. “NUMA-aware algorithms: the case of data shuffling”. In: *Proc. CIDR*. 2013. [see page 9]

[Li+20] Chu-Min Li, Fan Xiao, Mao Luo, et al. “Clause vivification by unit propagation in CDCL SAT solvers”. In: *Artificial Intelligence* 279 (2020), p. 103197: Elsevier. DOI: 10.1016/j.artint.2019.103197. [see page 19]

[Lia+16a] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. “Learning rate based branching heuristic for SAT solvers”. In: *Proc. SAT*. Springer. 2016, pp. 123–140. DOI: 10.1007/978-3-319-40970-2_9. [see pages 2, 17, 18, 29, 39]

[Lia+16b] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, et al. “MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB”. In: *Proc. SAT Competition*. 2016. [see page 29]

[Liu+16] Duo Liu, Cunxi Yu, Xiangyu Zhang, and Daniel Holcomb. “Oracle-guided incremental SAT solving to reverse engineer camouflaged logic circuits”. In: *Proc. DATE*. IEEE. 2016, pp. 433–438. DOI: 10.3850/9783981537079_0915. [see pages 22, 168, 175]

[LM21] Chu Min Li and Felip Manya. “MaxSAT, hard and soft constraints”. In: *Handbook of Satisfiability*. IOS Press, 2021, pp. 903–927. DOI: 10.3233/faia200987. [see page 22]

[LSB07] Matthew Lewis, Tobias Schubert, and Bernd Becker. “Multithreaded SAT solving”. In: *Asia and South Pacific Design Automation Conference*. IEEE. 2007, pp. 926–931. DOI: 10.1109/aspdac.2007.358108. [see pages 3, 25]

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. “Optimal speedup of Las Vegas algorithms”. In: *Information Processing Letters* 47.4 (1993), pp. 173–180: Elsevier. DOI: 10.1016/0020-0190(93)90029-9. [see page 19]

[Luo+17] Mao Luo, Chu-Min Li, Fan Xiao, et al. “An effective learnt clause minimization approach for CDCL SAT solvers”. In: *Proc. IJCAI*. 2017, pp. 703–711. DOI: 10.24963/ijcai.2017/98. [see pages 17, 19]

[Mah+14] Ibrahim M. Mahmoud, Lianchao Li, Dieter Wloka, and Mostafa Z. Ali. “Believable NPCs in serious games: HTN planning approach based on visual perception”. In: *2014 IEEE Conference on Computational Intelligence and Games*. IEEE. 2014, pp. 1–8. DOI: 10.1109/cig.2014.6932891. [see page 128]

[Mal+13] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. “Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering.” In: *Proc. IJCAI*. 2013, pp. 608–614. [see page 34]

[Man19] Jens Manig. “Kompressionstechniken für Beschreibungen von SAT Formeln”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2019. [see page 204]

[Man22] Norbert Manthey. “MergeSat, Merge-Mallob and Mallob-MergeCadLing”. In: *Proc. SAT Competition*. 2022, p. 25. [see page 178]

[Man23] Norbert Manthey. "Testing the ASCON Hash Function". In: *Proc. SAT Competition.* 2023, p. 63. [see page 39]

[Mar19] Andrea Marrella. "Automated planning for business process management". In: *J. data semantics* 8.2 (2019), pp. 79–98: Springer. DOI: 10.1007/s13740-018-0096-0. [see page 127]

[MB19] Sibylle Möhle and Armin Biere. "Backing backtracking". In: *Proc. SAT*. Springer. 2019, pp. 250–266. DOI: 10.1007/978-3-030-24258-9_18. [see page 16]

[McM03] Kenneth L. McMillan. "Interpolation and SAT-based model checking". In: *Proc. CAV*. Springer. 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1. [see page 22]

[Men21] Stefan Mengel. "A Naive SAT-Encoding of Cluster Editing". In: *Proc. SAT Competition.* 2021, p. 62. [see page 94]

[Met+05] Alexander Metzner, Martin Franzle, Christian Herde, and Ingo Stierand. "Scheduling distributed real-time systems by satisfiability checking". In: *Proc. RTCSA*. IEEE. 2005, pp. 409–415. DOI: 10.1109/rtcsa.2005.90. [see pages 21, 22]

[MG11] Peter Mell and Tim Grance. *The NIST definition of cloud computing.* Tech. rep. National Institute of Standards and Technology, 2011. DOI: 10.6028/nist.sp.800-145. [see page 11]

[MG99] João Marques-Silva and Thomas Glass. "Combinational equivalence checking using satisfiability and recursive learning". In: *Proc. Design, Automation and Test in Europe.* 1999, pp. 145–149. [see page 2]

[MHS16] Matteo Marescotti, Antti EJ Hyvärinen, and Natasha Sharygina. "Clause sharing and partitioning for cloud-based SMT solving". In: *Proc. ATVA*. Springer. 2016, pp. 428–443. DOI: 10.1007/978-3-319-46520-3_27. [see page 178]

[Mic+23] Dawn Michaelson, Dominik Schreiber, Marijn J. H. Heule, et al. "Unsatisfiability proofs for distributed clause-sharing SAT solvers". In: *Proc. TACAS*. Springer. 2023, pp. 348–366. DOI: 10.1007/978-3-031-30823-9_18. [see pages 7, 107, 203]

[MK98] Amol Dattatraya Mali and Subbarao Kambhampati. "Encoding HTN Planning in Propositional Logic". In: *Artificial Intelligence Planning Systems.* 1998, pp. 190–198. [see pages 22, 135]

[ML03] Stephen M. Majercik and Michael L. Littman. "Contingent planning under uncertainty via stochastic satisfiability". In: *Artificial Intelligence* 147.1-2 (2003), pp. 119–162: Elsevier. DOI: 10.1016/s0004-3702(02)00379-x. [see page 22]

[MLM21] João Marques-Silva, Inês Lynce, and Sharad Malik. "CDCL SAT Solving". In: *Handbook of Satisfiability.* IOS Press, 2021, pp. 131–153. DOI: 10.3233/faia200987. [see page 15]

[MMdS21] M.C. Magnaguagno, F. Meneguzzi, and L. de Silva. "HyperTensioN – A three-stage compiler for planning". In: *Proc. International Planning Competition.* 2021, pp. 5–8. [see pages 133, 142, 161, 163]

[MML12] Ruben Martins, Vasco Manquinho, and Inês Lynce. "An overview of parallel SAT solving". In: *Constraints* 17 (2012), pp. 304–347: Springer. DOI: 10.1007/s10601-012-9121-3. [see page 24]

[Mon16] David Monniaux. "A survey of satisfiability modulo theory". In: *Proc. CASC*. Springer. 2016, pp. 401–425. DOI: 10.1007/978-3-319-45641-6_26. [see page 22]

[Mos+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, et al. "Chaff: Engineering an efficient SAT solver". In: *Proc. DAC*. 2001, pp. 530–535. DOI: 10.1145/378239.379017. [see pages 2, 17, 18]

[MS00] João P. Marques-Silva and Karem A. Sakallah. "Boolean satisfiability in electronic design automation". In: *Proc. DAC*. 2000, pp. 675–680. DOI: 10.1145/337292.337611. [see pages 2, 21]

[MS96] João Marques-Silva and Karem A. Sakallah. "GRASP—a new search algorithm for satisfiability". In: *Proc. ICCAD*. IEEE. 1996, pp. 220–227. [see page 15]

[MS99] João Marques-Silva and Karem A. Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521: IEEE. DOI: 10.1109/12.769433. [see pages 2, 15]

[Nai+22] Abhishek Nair, Saranyu Chattopadhyay, Haoze Wu, et al. "Proof-Stitch: Proof Combination for Divide and Conquer SAT Solvers". In: *Proc. FMCAD*. 2022, pp. 84–88. [see page 33]

[Nau+99] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. "SHOP: Simple hierarchical ordered planner". In: *Proc. IJCAI*. 1999, pp. 968–973. [see pages 131, 133]

[NCT19] Yanik Ngoko, Christophe Cérin, and Denis Trystram. "Solving SAT in a distributed cloud: a portfolio approach". In: *J. Applied Mathematics and Computer Science* 29.2 (2019), pp. 261–274: Sciendo. DOI: 10.2478/amcs-2019-0019. [see pages 4, 33]

[Ngu23] Ha Phuong Nguyen. "An Empirical Study on Clause Selection and Filtering in Distributed SAT Solving". Bachelor's thesis. Karlsruhe Institute of Technology (KIT), 2023. [see page 205]

[NI20] Hidetomo Nabeshima and Katsumi Inoue. "Reproducible efficient parallel SAT solving". In: *Proc. SAT*. Springer. 2020, pp. 123–138. DOI: 10.1007/978-3-030-51825-7_10. [see page 30]

[NR12] Alexander Nadel and Vadim Ryvchin. "Efficient SAT solving under assumptions". In: *Proc. SAT*. Springer. 2012, pp. 242–255. DOI: 10.1007/978-3-642-31612-8_19. [see pages 23, 155, 167]

[NR18] Alexander Nadel and Vadim Ryvchin. "Chronological backtracking". In: *Proc. SAT*. Springer. 2018, pp. 111–121. DOI: 10.1007/978-3-319-94144-8_7. [see pages 16, 39]

[NTC17] Yanik Ngoko, Denis Trystram, and Christophe Cérin. "A Distributed Cloud Service for the Resolution of SAT". In: *Proc. IEEE SC2*. IEEE. 2017, pp. 1–8. DOI: 10.1109/sc2.2017.9. [see pages 33, 65]

[OB05] Oliver Obst and Joschka Boedecker. "Flexible coordination of multiagent team behavior using HTN planning". In: *RoboCup*. Springer. 2005, pp. 521–528. DOI: 10.1007/11780519_49. [see page 128]

[Oh15] Chanseok Oh. "Between SAT and UNSAT: the fundamental difference in CDCL SAT". In: *Proc. SAT*. Springer. 2015, pp. 307–323. DOI: 10.1007/978-3-319-24318-4_23. [see pages 17, 19, 34, 35]

[Oh16] Chanseok Oh. "Improving SAT solvers by exploiting empirical characteristics of CDCL". PhD thesis. New York University, 2016. [see page 29]

[ORe21] Gerard O'Regan. "Foundations of Computing". In: *A Brief History of Computing.* 3rd ed. Springer, 2021, pp. 35–51. DOI: 10.1007/978-3-030-66599-9. [see page 1]

[Ost+10] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, et al. "A performance analysis of EC2 cloud computing services for scientific computing". In: *Proc. CloudComp*. Springer. 2010, pp. 115–131. DOI: 10.1007/978-3-642-12636-9_9. [see page 11]

[OU09] Kei Ohmura and Kazunori Ueda. "c-sat: A Parallel SAT Solver for Clusters". In: *Proc. SAT*. Springer. 2009, pp. 524–537. DOI: 10.1007/978-3-642-02777-2_47. [see page 35]

[OW21] Muhammad Osama and Anton Wijs. "Verifying String Safety Properties in AWS C99 Package with CBMC". In: *Proc. SAT Competition*. 2021, p. 64. [see pages 88, 92]

[OWB21a] Muhammad Osama, Anton Wijs, and Armin Biere. "SAT solving with GPU accelerated inprocessing". In: *Proc. TACAS*. Springer. 2021, pp. 133–151. DOI: 10.26226/morressier.604907f41a80aac83ca25cee. [see page 180]

[OWB21b] Alex Ozdemir, Haoze Wu, and Clark Barrett. "SAT Solving in the Serverless Cloud". In: *Proc. FMCAD*. IEEE. 2021, pp. 241–245. DOI: 10.34727/2021/isbn.978-3-85448-046-4_33. [see page 33]

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. "A survey of recent advances in SAT-based formal verification". In: *J. Software Tools for Technology Transfer* 7 (2005), pp. 156–173: Springer. DOI: 10.1007/s10009-004-0183-4. [see page 22]

[PD07] Knot Pipatsrisawat and Adnan Darwiche. "A lightweight component caching scheme for satisfiability solvers". In: *Proc. SAT*. Springer. 2007, pp. 294–299. DOI: 10.1007/978-3-540-72788-0_28. [see page 18]

[PFB23] Florian Pollitt, Mathias Fleury, and Armin Biere. "Faster LRAT checking than solving with CaDiCaL". In: *Proc. SAT*. 2023. DOI: 10.4230/LIPIcs.SAT.2023.21. [see pages 124, 179]

[Pfi01] Gregory F. Pfister. "An introduction to the infiniband architecture". In: *High performance mass storage and parallel I/O* 42.617-632 (2001), p. 102 . [see page 10]

[Phi05] Ian Philp. "Software failures and the road to a petaflop machine". In: *Proc. HPCA*. 2005, pp. 125–128. [see page 181]

[PHS08] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. "Vivifying propositional clausal formulae". In: *Proc. ECAI*. 2008, pp. 525–529. [see page 19]

[Pop34] Karl R. Popper. *Logik der Forschung*. 1934. [see page 4]

[PSM21] Nicolas Prevot, Mate Soos, and Kuldeep S. Meel. "Leveraging GPUs for Effective Clause Sharing in Parallel SAT Solving". In: *Proc. SAT*. Springer. 2021, pp. 471–487. DOI: `10.1007/978-3-030-80223-3_32`. [see pages 28, 180]

[QPF23] Gaspard Quenard, Damien Pellier, and Humbert Fiorino. "LTP: Lifted Tree Path". In: *Proc. International Planning Competition*. 2023. To appear. [see page 179]

[Ram+17] Abdeldjalil Ramoul, Damien Pellier, Humbert Fiorino, and Sylvie Pesty. "Grounding of HTN planning domain". In: *J. Artificial Intelligence Tools* 26.05 (2017), p. 1760021: World Scientific. DOI: `10.1142/s0218213017600211`. [see pages 128, 133, 135]

[RB23] Joseph E. Reeves and Randal E. Bryant. "Preprocessors PReLearn and ReEncode Entering the SAT Competition 2023". In: *Proc. SAT Competition*. 2023, p. 23. [see page 80]

[Rec23] Leibniz Rechenzentrum. *SuperMUC-NG*. 2023. URL: `https://doku.lrz.de/supermuc-ng-10745965.html`. [see page 57]

[Rei81] Raymond Reiter. "On closed world data bases". In: *Readings in artificial intelligence*. Elsevier, 1981, pp. 119–140. DOI: `10.1016/b978-0-934613-03-3.50014-3`. [see page 130]

[Reu+18] Albert Reuther, Chansup Byun, William Arcand, et al. "Scalable system scheduling for HPC and big data". In: *J. Parallel and Distributed Computing* 111 (2018), pp. 76–92: Elsevier. DOI: `10.1016/j.jpdc.2017.06.009`. [see pages 11, 180]

[RG15] Masood Feyzbakhsh Rankooh and Gholamreza Ghassem-Sani. "ITSAT: an efficient sat-based temporal planner". In: *JAIR* 53 (2015), pp. 541–632. DOI: `10.1613/jair.4697`. [see page 22]

[Rin04] Jussi Rintanen. "Evaluation strategies for planning as satisfiability". In: *Proc. ECAI*. 2004, p. 682. [see page 154]

[Rin14] Jussi Rintanen. "Madagascar: Scalable planning with SAT". In: *Proc. International Planning Competition*. 2014. [see pages 22, 134]

[Rob+09] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. "SAT-Based Parallel Planning Using a Split Representation of Actions". In: *Proc. ICAPS*. 2009, pp. 281–288. DOI: `10.1609/icaps.v19i1.13368`. [see page 136]

[Rob65] John Alan Robinson. "A machine-oriented logic based on the resolution principle". In: *JACM* 12.1 (1965), pp. 23–41: ACM. DOI: `10.1145/321250.321253`. [see page 14]

[Rou12] Olivier Roussel. "Description of ppfolio (2011)". In: *Proc. SAT Challenge*. 2012, p. 46. [see page 28]

[RP12] Purushothaman Raja and Sivagurunathan Pugazhenthi. "Optimal path planning of mobile robots: A review". In: *J. Physical Sciences* 7.9 (2012), pp. 1314–1320 . [see page 127]

[RS98] Martin Raab and Angelika Steger. ""Balls into bins"—A simple and tight analysis". In: *Int. Workshop on Randomization and Approximation Techniques in Computer Science*. 1998, pp. 159–170. [see page 50]

[Rya04] Lawrence Ryan. "Efficient algorithms for clause-learning SAT solvers". PhD thesis. Theses (School of Computing Science)/Simon Fraser University, 2004. [see pages 17, 18]

[San+19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019. DOI: 10.1007/978-3-030-25209-0. [see pages 9, 10, 12, 51, 70]

[San00] Peter Sanders. "Fast priority queues for cached memory". In: *JEA* 5 (2000), 7–es: ACM. DOI: 10.1145/351827.384249. [see page 117]

[San02] Peter Sanders. "Randomized receiver initiated load-balancing algorithms for tree-shaped computations". In: *The Computer Journal* 45.5 (2002), pp. 561–573: Oxford University Press. DOI: 10.1093/comjnl/45.5.561. [see page 56]

[San09] Peter Sanders. "Algorithm engineering–an attempt at a definition". In: *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday* (2009), pp. 321–340: Springer. DOI: 10.1007/978-3-642-03456-5_22. [see pages 4, 5]

[SB09] Niklas Sörensson and Armin Biere. "Minimizing learned clauses". In: *Proc. SAT*. Springer. 2009, pp. 237–243. DOI: 10.1007/978-3-642-02777-2_23. [see page 15]

[SB10] Sven Schulz and Wolfgang Blochinger. "Cooperate and compete! A hybrid solving strategy for task-parallel SAT solving on peer-to-peer desktop grids". In: *Proc. Int. Conf. HPC & Simulation*. IEEE. 2010, pp. 314–323. [see page 25]

[SBA18] Thomas Sterling, Maciej Brodowicz, and Matthew Anderson. *High performance computing: modern systems and practices*. Morgan Kaufmann, 2018. DOI: 10.1016/C2013-0-09704-6. [see pages 4, 11]

[SBK01] Carsten Sinz, Wolfgang Blochinger, and Wolfgang Küchlin. "PaSAT – Parallel SAT-checking with lemma exchange: Implementation and applications". In: *Electronic Notes in Discrete Mathematics* 9 (2001), pp. 205–216: Elsevier. DOI: 10.1016/s1571-0653(04)00323-3. [see pages 25, 27, 28]

[Sch+19a] Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomáš Balyo. "Efficient SAT Encodings for Hierarchical Planning". In: *Proc. ICAART*. 2019, pp. 531–538. DOI: 10.5220/0007343305310538. [see pages 23, 127, 128, 134, 135, 203]

[Sch+19b] Dominik Schreiber, Damien Pellier, Humbert Fiorino, and Tomáš Balyo. "Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning". In: *Proc. ICAPS*. 2019, pp. 382–390. DOI: 10.1609/icaps.v29i1.3502. [see pages 22, 23, 127–129, 134–136, 140, 145–147, 152–155, 199, 203]

[Sch16] Dominik Schreiber. "Energieeffiziente Ausführung von qualitätsbewussten Algorithmen für Mobile Simulationen". Bachelor's thesis. Universität Stuttgart, 2016. [see page 204]

[Sch18] Dominik Schreiber. "Hierarchical task network planning using SAT techniques". Master's thesis. Grenoble Institut National Polytechnique (INP) and Karlsruhe Institute of Technology (KIT), 2018. DOI: 10.5445/IR/1000104165. [see pages 127, 146, 154, 196, 204]

[Sch20] Dominik Schreiber. “Engineering HordeSat towards malleability: mallob-mono in the SAT 2020 cloud track”. In: *Proc. SAT Competition*. 2020, pp. 45–46. [see pages 7, 65, 79, 101, 204]

[Sch21a] Maximilian Schick. “Cube&Conquer-inspired Malleable Distributed SAT Solving”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2021. [see pages 25, 41, 55, 80, 205]

[Sch21b] Nikolai Schnell. “Pruning Techniques for Lifted SAT-based Hierarchical Planning”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2021. [see pages 155, 164, 169, 205]

[Sch21c] Dominik Schreiber. “Lifted logic for task networks: TOHTN planner Lilotane in the IPC 2020”. In: *Proc. International Planning Competition*. 2021, pp. 9–12. [see page 204]

[Sch21d] Dominik Schreiber. “Lilotane: A Lifted SAT-Based Approach to Hierarchical Planning”. In: *JAIR* 70 (2021), pp. 1117–1181. DOI: `10.1613/jair.1.12520`. [see pages 7, 127, 140, 142, 144, 148, 150, 155, 179, 203]

[Sch21e] Dominik Schreiber. “Mallob in the SAT Competition 2021”. In: *Proc. SAT Competition*. 2021, pp. 38–39. [see pages 7, 65, 102, 204]

[Sch22] Dominik Schreiber. “Mallob in the SAT Competition 2022”. In: *Proc. SAT Competition*. 2022, pp. 46–47. [see pages 7, 65, 103, 119, 204]

[Sch23] Dominik Schreiber. “Mallob{32,64,1600} in the SAT Competition 2023”. In: *Proc. SAT Competition*. 2023, pp. 46–47. [see pages 7, 65, 83, 104, 204]

[Sco+21] Joseph Scott, Aina Niemetz, Mathias Preiner, et al. “MachSMT: A machine learning-based algorithm selector for SMT solvers”. In: *Proc. TACAS*. Springer. 2021, pp. 303–325. DOI: `10.1007/978-3-030-72013-1_16`. [see page 34]

[SH23] Bernardo Subercaseaux and Marijn J. H. Heule. “The packing chromatic number of the infinite square grid is 15”. In: *Proc. TACAS*. Springer. 2023, pp. 389–406. DOI: `10.1007/978-3-031-30823-9_20`. [see pages 3, 31, 108]

[Sim14] Laurent Simon. “Post Mortem Analysis of SAT Solver Proofs.” In: *Proc. Pragmatics of SAT*. 2014, pp. 26–40. [see pages 30, 35]

[Sin07] Carsten Sinz. “Visualizing SAT Instances and Runs of the DPLL Algorithm”. In: *J. Autom. Reason.* 39.2 (2007), pp. 219–243: Springer. DOI: `10.1007/s10817-007-9074-1`. [see page 3]

[Sir+04] Evren Sirin, Bijan Parsia, Dan Wu, et al. “HTN planning for web service composition using SHOP2”. In: *J. Web Semantics* 1.4 (2004), pp. 377–396: Elsevier. DOI: `10.1016/j.websem.2004.06.005`. [see page 128]

[SLH05] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. “The SAT2002 competition”. In: *Annals of Mathematics and Artificial Intelligence* 43 (2005), pp. 307–342: Springer. DOI: `10.1007/s10472-005-0424-6`. [see page 38]

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. “A New Method for Solving Hard Satisfiability Problems”. In: *Proc. AAAI*. 1992, pp. 440–446. [see page 2]

[Smi22] Robin Smith. *Aristotle’s Logic. The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta and Uri Nodelman. 2022. URL: `https://plato.stanford.edu/archives/win2022/entries/aristotle-logic/`. [see page 1]

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Proc. SAT*. Springer. 2009, pp. 244–257. DOI: 10.1007/978-3-642-02777-2_24. [see pages 2, 17, 21, 155]

[Sön21] Malte Sönnichsen. "Asynchronous Clause Exchange for Malleable SAT Solving". Master's thesis. Karlsruhe Institute of Technology (KIT), 2021. [see pages 41, 55, 205]

[Spe17] Ivor Spence. "Balanced random SAT benchmarks". In: *Proc. SAT Competition*. 2017, p. 53. [see page 94]

[SS07] Matthew J. Streeter and Stephen F. Smith. "Using Decision Procedures Efficiently for Optimization". In: *Proc. ICAPS*. 2007, pp. 312–319. [see page 154]

[SS11] Peter Sanders and Jochen Speck. "Efficient parallel scheduling of malleable tasks". In: *Proc. International Parallel & Distributed Processing Symp.* IEEE. 2011, pp. 1156–1166. DOI: 10.1109/ipdps.2011.110. [see page 43]

[SS12] Peter Sanders and Jochen Speck. "Energy efficient frequency scaling and scheduling for malleable tasks". In: *Proc. Euro-Par*. Springer. 2012, pp. 167–178. DOI: 10.1007/978-3-642-32820-6_18. [see pages 41, 43, 179]

[SS21a] André Schidler and Stefan Szeider. "SAT-based decision tree learning for large data sets". In: *Proc. AAAI*. 2021, pp. 3904–3912. DOI: 10.1609/aaai.v35i5.16509. [see page 21]

[SS21b] Dominik Schreiber and Peter Sanders. "Scalable SAT Solving in the Cloud". In: *Proc. SAT*. Springer. 2021, pp. 518–534. DOI: 10.1007/978-3-030-80223-3_35. [see pages 7, 59, 65, 74, 75, 79, 87, 108, 203]

[SS21c] Malte Sönnichsen and Dominik Schreiber. "The "Factories" HTN Domain". In: *Proc. International Planning Competition*. 2021, pp. 45–46. [see pages 132, 163, 204]

[SS22a] Peter Sanders and Dominik Schreiber. "Decentralized online scheduling of malleable NP-hard jobs". In: *Proc. Euro-Par*. Springer. 2022, pp. 119–135. DOI: 10.1007/978-3-031-12597-3_8. [see pages 7, 41, 61, 203]

[SS22b] Peter Sanders and Dominik Schreiber. "Mallob: Scalable SAT Solving On Demand With Decentralized Job Scheduling". In: *JOSS* 7.76 (2022), p. 4591. DOI: 10.21105/joss.04591. [see page 203]

[SS97] JOM Silva and Karem A. Sakallah. "Robust search algorithms for test pattern generation". In: *Proc. IEEE Int. Symp. Fault Tolerant Computing*. IEEE. 1997, pp. 152–161. DOI: 10.1109/ftcs.1997.614088. [see page 21]

[Sur+22] Pavel Surynek, Roni Stern, Eli Boyarski, and Ariel Felner. "Migrating Techniques from Search-Based Multi-Agent Path Finding Solvers to SAT-Based Approach". In: *JAIR* 73 (2022), pp. 553–618. DOI: 10.1613/jair.1.13318. [see pages 3, 22, 23, 168, 175, 178]

[Sur12] Pavel Surynek. "Towards optimal cooperative path planning in hard setups through satisfiability solving". In: *Pacific Rim Int. Conf. Artificial Intelligence*. Springer. 2012, pp. 564–576. DOI: 10.1007/978-3-642-32695-0_50. [see page 22]

[SvdP22] Irfansha Shaik and Jaco van de Pol. “Classical planning as QBF without grounding”. In: *Proc. ICAPS.* 2022, pp. 329–337. DOI: 10.1609/icaps.v32i1.19817. [see page 23]

[Tau+02] Rizwan Ali Tau Leng, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. “An empirical study of hyper-threading in high performance computing clusters”. In: *Linux HPC Revolution* 45 (2002) . [see page 9]

[THM21] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. “cake_lpr: Verified Propagation Redundancy Checking in CakeML”. In: *Proc. TACAS.* Springer. 2021, pp. 223–241. DOI: 10.1007/978-3-030-72013-1_12. [see page 109]

[THM23] Yong Kiam Tan, Marijn J. H. Heule, and Magnus Myreen. “Verified LRAT and LPR Proof Checking with cake_lpr”. In: *Proc. SAT Competition.* 2023, p. 89. [see page 34]

[TM12] Markus Triska and Nysret Musliu. “An improved SAT formulation for the social golfer problem”. In: *Annals of Operations Research* 194.1 (2012), pp. 427–438: Citeseer. DOI: 10.1007/s10479-010-0702-5. [see page 3]

[Tse83] Grigori S. Tseitin. “On the complexity of derivation in propositional calculus”. In: *Automation of reasoning: 2: Classical papers on computational logic 1967–1970* (1983), pp. 466–483: Springer. DOI: 10.1007/978-3-642-81955-1_28. [see page 13]

[Ull08] Carsten Ullrich. *Pedagogically founded courseware generation for web-based learning: an HTN-planning-based approach implemented in PAIGOS.* Vol. 5260. Springer, 2008. DOI: 10.1007/978-3-540-88215-2. [see page 128]

[Vai+15] Karthikeyan Vaidyanathan, Dhiraj D. Kalamkar, Kiran Pamnany, et al. “Improving concurrency and asynchrony in multithreaded MPI applications using software offloading”. In: *Proc. Int. Conf. HPC, Networking, Storage and Analysis.* 2015, pp. 1–12. DOI: 10.1145/2807591.2807602. [see page 53]

[Val+20a] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, et al. “Community and LBD-based clause sharing policy for parallel SAT solving”. In: *Proc. SAT.* Springer. 2020, pp. 11–27. DOI: 10.1007/978-3-030-51825-7_2. [see pages 28, 29, 39]

[Val+20b] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, et al. “P-MCOMSPS-STR: a Painless-based Portfolio of MapleCOMSPS with Clause Strengthening”. In: *Proc. SAT Competition.* 2020, p. 56. [see page 28]

[Val+21] Vincent Vallade, Ludovic Le Frioux, Razvan Oanea, et al. “New concurrent and distributed painless solvers: p-mcomsps, p-mcomsps-com, p-mcomsps-mpi, and p-mcomsps-com-mpi”. In: *Proc. SAT Competition.* 2021, p. 40. [see page 29]

[Van08] Allen Van Gelder. “Verifying RUP Proofs of Propositional Unsatisfiability.” In: *ISAIM.* 2008. [see page 21]

[vdHey19] Jean-Pierre von der Heydt. “Cube&Conquer-inspired Techniques for Parallel Automated Planning”. Bachelor’s thesis. Karlsruhe Institute of Technology (KIT), 2019. [see page 204]

[VWM15] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. “Boolean Satisfiability Solvers and Their Applications in Model Checking”. In: *Proc. IEEE.* 2015, pp. 2021–2035. DOI: 10.1109/JPROC.2015.2455034. [see page 22]

[Web05] Tjark Weber. “A SAT-based Sudoku solver”. In: *LPAR*. 2005, pp. 11–15. [see pages 21, 26]

[Wel94] Daniel S. Weld. “An introduction to least commitment planning”. In: *AI Magazine* 15 (1994), pp. 27–61 . [see page 129]

[WH13a] Siert Wieringa and Keijo Heljanko. “Asynchronous multi-core incremental SAT solving”. In: *Proc. TACAS*. Springer. 2013, pp. 139–153. DOI: 10.1007/978-3-642-36742-7_10. [see pages 23, 175]

[WH13b] Siert Wieringa and Keijo Heljanko. “Concurrent clause strengthening”. In: *Proc. SAT*. Springer. 2013, pp. 116–132. DOI: 10.1007/978-3-642-39071-5_10. [see pages 19, 28]

[WH20] Sean Weaver and Marijn J. H. Heule. “Constructing minimal perfect hash functions using SAT technology”. In: *Proc. AAAI*. 2020, pp. 1668–1675. DOI: 10.1609/aaai.v34i02.5529. [see page 21]

[Wil20] Marvin Williams. “Partially Instantiated Representations for Automated Planning”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2020. [see pages 136, 205]

[Wil22] Niko Wilhelm. “Malleable Distributed Hierarchical Planning”. Master’s thesis. Karlsruhe Institute of Technology (KIT), 2022. [see pages 41, 56, 205]

[Wot+12] Andreas Wotzlaw, Alexander van der Grinten, Ewald Speckenmeyer, and Stefan Porschen. “pfolioUZK: Solver description”. In: *Proc. SAT Challenge*. 2012. [see page 28]

[WOZ10] Martin Weser, Dominik Off, and Jianwei Zhang. “HTN robot planning in partially observable dynamic environments”. In: *IEEE Int. Conf. Robotics and Automation*. IEEE. 2010, pp. 1505–1510. DOI: 10.1109/robot.2010.5509770. [see page 128]

[WTH19] Julia Wichlacz, Alvaro Torralba, and Jörg Hoffmann. “Construction-planning models in minecraft”. In: *Proc. ICAPS Workshop on Hierarchical Planning*. 2019. [see page 128]

[Wu+23] Ying Xian Wu, Conny Olz, Songtuan Lin, and Pascal Bercher. “Grounded (Lifted) Linearizer: Solving Partial Order HTN Problems by Linearizing Them”. In: *Proc. International Planning Competition*. 2023. To appear. [see page 179]

[XA05] Yichen Xie and Alex Aiken. “Saturn: A SAT-based tool for bug detection”. In: *Proc. CAV*. Springer. 2005, pp. 139–143. DOI: 10.1007/11513988_13. [see page 22]

[XFJ16] Bruno Xavier, Tiago Ferreto, and Luis Jersak. “Time provisioning evaluation of kvm, docker and unikernels in a cloud platform”. In: *Proc. IEEE/ACM CCGrid*. IEEE. 2016, pp. 277–280. DOI: 10.1109/ccgrid.2016.86. [see page 11]

[Xu+12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. “Evaluating component solver contributions to portfolio-based algorithm selectors”. In: *Proc. SAT*. Springer. 2012, pp. 228–241. DOI: 10.1007/978-3-642-31612-8_18. [see pages 26, 27, 79]

[YZ13] Haihang You and Hao Zhang. “Comprehensive workload analysis and modeling of a petascale supercomputer”. In: *Job Scheduling Strategies for Parallel Processing: International Workshop, JSSPP*. Springer. 2013, pp. 253–271. DOI: 10.1007/978-3-642-35867-8_14. [see page 11]

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. “PSATO: a distributed propositional prover and its application to quasigroup problems”. In: *J. Symbolic Computation* 21.4-6 (1996), pp. 543–560: Academic Press. DOI: 10.1006/jsco.1996.0030. [see pages 3, 24, 31]

[ZCC22] Xindi Zhang, Zhihan Chen, and Shaowei Cai. “ParKissat: Random Shuffle Based and Pre-processing Extended Parallel Solvers with Clause Sharing”. In: *Proc. SAT Competition*. 2022, p. 51. [see pages 29, 103, 120]

[ZCC23] Xindi Zhang, Zhihan Chen, and Shaowei Cai. “PRS: A new parallel/distributed framework for SAT”. In: *Proc. SAT Competition*. 2023, pp. 39–40. [see pages 29, 104, 105, 178]

[Zha+01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. “Efficient conflict driven learning in a boolean satisfiability solver”. In: *Proc. IEEE/ACM ICCAD*. IEEE. 2001, pp. 279–285. DOI: 10.1109/iccad.2001.968634. [see page 15]

[Zha+11] Jing Zhang, Gongqing Wu, Xuegang Hu, et al. “A parallel k-means clustering algorithm with MPI”. In: *Proc. Symp. Parallel Architectures, Algorithms and Programming*. IEEE. 2011, pp. 60–64. DOI: 10.1109/paap.2011.17. [see page 57]

[Zha02] Hantao Zhang. “Generating college conference basketball schedules by a SAT solver”. In: *Proc. SAT*. Springer. 2002, pp. 281–291. [see page 22]

[Zha96] Jian Zhang. “Constructing finite algebras with FALCON”. In: *J. Autom. Reason.* 17 (1996), pp. 1–22: Springer. DOI: 10.1007/bf00247667. [see page 21]

[Zhe+14] Yili Zheng, Amir Kamil, Michael B. Driscoll, et al. “UPC++: a PGAS extension for C++”. In: *IEEE Int. Parallel and Distributed Processing Symp.* 2014, pp. 1105–1114. [see page 10]

[Zhe+22] Jiongzhi Zheng, Kun He, Zhuo Chen, et al. “Combining Hybrid Walking Strategy with Kissat MAB, CaDiCaL, and LStech-Maple”. In: *Proc. SAT Competition*. 2022, p. 20. [see pages 90, 120]

[ZK17] Neng-Fa Zhou and Håkan Kjellerstrand. “Optimizing SAT encodings for arithmetic constraints”. In: *Proc. CP*. Springer. 2017, pp. 671–686. DOI: 10.1007/978-3-319-66158-2_43. [see page 3]

[ZM88] Ramin Zabih and David A. McAllester. “A Rearrangement Search Strategy for Determining Propositional Satisfiability”. In: *Proc. AAAI*. 1988, pp. 155–160. [see page 14]