SAT Solving in the Serverless Cloud

Alex Ozdemir[§], Haoze Wu[§], and Clark Barrett
Stanford University, USA.
{aozdemir, haozewu, barrett}@cs.stanford.edu

***Abstract*—In recent years, cloud service providers have sold computation in increasingly granular units. Most recently, "serverless" executors run a single executable with restricted network access and for a limited time. The benefit of these restrictions is scale: thousand-way parallelism can be allocated in seconds, and CPU time is billed with sub-second granularity. To exploit these executors, we introduce `gg-SAT`: an implementation of divide-and-conquer SAT solving. Infrastructurally, `gg-SAT` departs substantially from previous implementations: rather than handling process or server management itself, `gg-SAT` builds on the `gg` framework, allowing computations to be executed on a configurable backend, including serverless offerings such as AWS Lambda. Our experiments suggest that when run on the same hardware, `gg-SAT` performs competitively with other D&C solvers, and that the 1000-way parallelism it offers (through AWS Lambda) is useful for some challenging SAT instances.**

***Index Terms*—parallel SAT, serverless computing, divide and conquer.**

I. Introduction

Modern Boolean satisfiability (SAT) solvers have been successfully applied to important practical and theoretical domains, such as hardware verification, planning, and mathematics. Progress in the scalability of these tools has come from both algorithmic improvements and better leveraging of multi-processing hardware. While the number of processors on a single machine is limited, and maintaining a warm cluster to run occasional tasks is expensive, cloud-computing is a promising approach for leveraging on-demand parallelism at low cost.

Recent cloud-computing services are offered at increasingly fine granularity and low latency. Instead of renting a server or a cluster, one can now rent state-free executors, which can be rapidly and plentifully provisioned at a low price—a paradigm referred to as *serverless computing*. Serverless executors generally have restricted network access, limited memory, and limited runtime. For example, Amazon's Lambda service rents a Linux container to run arbitrary x86-64 executables for up to 15 minutes, with less than a second of startup time and no charge when idle. Similar services are offered by Google, Microsoft, Alibaba, and IBM. Previous research has used serverless computing as a "burstable supercomputer" for video processing [2], neural network training [25], and more [13]–[15], [33]. These successes beg the question: "can serverless computing be leveraged for massively parallel SAT-solving?"

There are two traditional parallel SAT-solving paradigms: 1) the portfolio approach, where each thread runs a different SAT solver on the same instance; and 2) the divide-and-conquer (D&C) approach, where a problem is partitioned into independent sub-problems to be solved in parallel. While the former approach in combination with clause-sharing leads to surprisingly good performance for small portfolio sizes, the benefits decrease as parallel computing power increases, and this approach is also not well aligned with the runtime and communication limitations of serverless executors. In this paper, we follow the second approach and present `gg-SAT`, a divide-and-conquer (D&C) SAT solver compatible with serverless computing. `gg-SAT` makes black-box use of a *solver* (e.g., `CaDiCaL` [8]) and a *divider* (e.g., `march` [28]) to solve and partition the problems, respectively. Problem division is performed throughout the search, whenever a sub-problem reaches a timeout imposed by either the user or the cloud-service. Infrastructurally, `gg-SAT` differs substantially from previous D&C implementations: rather than handling process or server management itself, `gg-SAT` builds on top of the `gg` framework for parallel computation. By expressing D&C search using `gg`, `gg-SAT` can execute that search on any mixture of user-specified backends; supported backends currently include local processes, remote machines, and serverless cloud-services such as AWS Lambda and Google Cloud Functions. To implement `gg-SAT`, we designed and built `pygg`, a novel and idiomatic Python interface to `gg`. We expect that `pygg` will be independently useful for other future projects, perhaps including parallel SMT solving.

We evaluate `gg-SAT` using local processes and AWS Lambda as backends. Local experiments suggest that `gg-SAT` performs competitively with the original Cube-and-Conquer prototype [19], a recent reimplementation of it [18], and a portfolio solver `PLingeling` [7], on benchmarks taken from [18], [19]. Cloud experiments suggest that `gg-SAT` unlocks levels of parallelism which are useful for solving some challenging instances from the 2020 SAT Competition.

II. Background & Related Work

A. Parallel SAT

Propositional satisfiability is an old problem; we refer the reader to the handbook of satisfiability [9] for an introduction. Parallel SAT-solving also has a lengthy history, with two main approaches.

The first approach is *portfolio solving*, pioneered in [16], [22], [34]. In a portfolio solver, each thread runs a different solver or configuration on the same original formula. An instance is solved as quickly as the best individual solver for that instance. Portfolio solvers include: `ManySAT`

[§]Equal contribution

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_33

[17], CryptoMinisat [32], PLingeling [7], Syrup [3], HordSAT [6], and Painless [26]. Some portfolio solvers also use *clause sharing* [11], [31]: sharing learnt clauses among the different solvers.

Another approach to parallelizing SAT is *divide-and-conquer* (D&C). D&C solvers attempt to divide a SAT instance into easier SAT instances, which can then be solved in parallel by a base solver. Typically, D&C solvers divide instances by partitioning the search space. The important questions—how and when to divide—are answered heuristically, typically with heuristics derived from look-ahead solvers and CDCL solvers. There has been substantial work on D&C SAT solving [10], [23], [24], including: Psato [35], Painless [27], and AMPHAROS [29]. One prominent approach, "cube-and-conquer" [19] uses a lookahead solver to divide instances and a CDCL solver to solve subproblems; this approach has been successful for large mathematical problems [21].

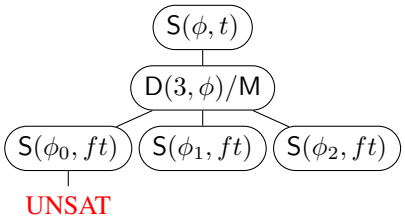
B. Distributed SAT

A number of systems attempt parallel SAT solving using a cluster of computers, possibly rented from the cloud. Most of these systems (Qsat [30], HordSAT [6], TopoSAT [12], SLIME [20]) follow the portfolio approach. One recent system (Paracooba [18]) follows the D&C approach. All of these systems operate in the "cluster" computational model, in which long-running processes on each node communicate over the network.

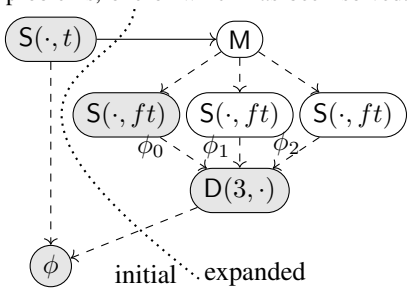
C. Serverless Computing

Cloud service providers, such as Microsoft Azure, rent out computational resources including compute, storage, and accelerators. Over the past decade, service providers have rented compute with increasing granularity, scale, and availability. Their recent offerings include *serverless* services, which run a single executable for a limited time, with limited memory and restricted network access. While restricted, serverless computing has strengths: it offers massive parallelism that can be rapidly provisioned, with fine-grained billing. For example, AWS Lambda [4] runs executables for up to 15 minutes, with 3GB of memory and 500MB of disk space; the runs are billed at sub-second granularity, and a thousand executors can be provisioned in seconds.

While serverless computing was designed for operational convenience, recent work has explored using it as a "burstable supercomputer-on-demand" [13], for tasks such as video processing [2], ray tracing [14], and machine learning [25]. One system, gg [13], provides a general framework for leveraging minimal executors (including serverless ones). It uses a configurable backend (such as a local machine, remote machines, or serverless executors) to evaluate a programmer-defined dependency graph of *thunks*: programs that take files as inputs. Thunks can output files or new thunks; the latter causes the dependency graph to dynamically grow. Dynamic dependency graphs can express many applications; gg has been used for tasks such as neural network verification [33], compilation [13], and video encoding [15].



(a) The D&C search tree. ϕ's solve query times out and is split into three sub-problems, one of which has been solved.



(b) The gg dependency graph. Dashed arrows denote dependencies; if a node produces multiple outputs, the dependency edges are labelled. The solid arrow denotes a thunk that returns another thunk. Shaded thunks have been evaluated.

Fig. 1: A D&C search snapshot and its corresponding dependency graph. In both diagrams, S, M, and D denote solve, merge, and divide, respectively.

III. Description

A. Algorithm

gg-SAT uses a D&C algorithm with multiplicatively growing timeouts. It is parameterized by a *base solver* and a *divider*. The base solver can be any SAT solver. The divider's job is to partition a problem into a requested number of sub-problems such that the disjunction of the sub-problems is equisatisfiable with the original problem. Other parameters to the algorithm include the timeout t, the timeout growth factor f, the number of initial partitions p_i, and the number of partitions for each sub-problem, p_s.

Figure 1a illustrates the solving of formula ϕ as a tree, with $p_i = 1$ and $p_s = 3$. The number of initial divisions is 1, so the base solver first attempts the original problem ϕ with timeout t. This times out, so the divider runs and splits ϕ into sub-problems (ϕ_0, ϕ_1, ϕ_2), each of which is attempted with timeout ft. The sub-problem ϕ_0 is determined to be UNSAT; other sub-problems have yet to be solved, and may be divided again. The process ends when all sub-problems are determined to be UNSAT or any sub-problem is determined to be SAT.

B. Implementation

To apply D&C to SAT, we must instantiate its primitive notions (sub-problems, solving, and dividing) for SAT. We follow previous work [19], [24] by using a lookahead solver (march) to build sub-problems described by *cubes* (lists of asserted literals) and by using a CDCL solver (CaDiCaL [8]) to attempt to solve problems and sub-problems. march can

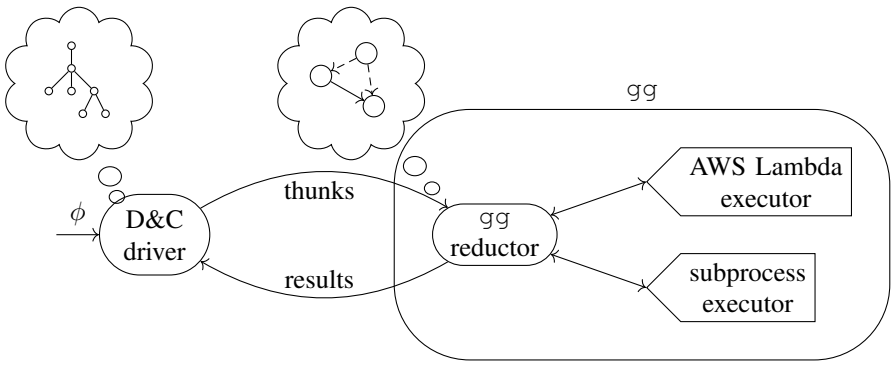


Fig. 2: gg-SAT expresses D&C search as a dynamically expanding dependency graph and uses gg to evaluate that graph using a back-end of the user's choice.

produce a large number of cubes (e.g., millions) and can take a long time. This was appropriate for cube-and-conquer (which ran march exactly once per problem) but is inappropriate for divide-and-conquer (which runs march many times seeking a small number of sub-problems each time). To address this, we configure march with a maximum cube length, which substantially reduces its runtime.

Our D&C implementation uses the gg framework for parallel execution [13]. Recall (§II) that using gg requires the computation to be expressed as a dependency graph of thunks, each of which is an individual executable. For D&C, there are three kinds of thunks. *Solve thunks* run the base solver; if it returns a result, the thunk returns that result as well; otherwise, the solve thunk returns a *merge thunk*, which combines the solutions to sub-problems that are produced by a *divide thunk*, which runs the divider. Figure 1 illustrates the relationship between an in-progress D&C search and the gg dependency graph. When D&C attempts to solve $\mathsf{S}(\phi, t)$, the dependency graph contains only the nodes left of the dotted line. However, when that query times out, the corresponding thunk returns 5 new thunks: a divide thunk to create 3 sub-problems, three solve thunks to (attempt to) solve them, and a merge thunk, whose output should be taken as the output of the original S thunk.

By expressing D&C search as a gg dependency graph, we can use gg to execute that search using a back-end (or combination of back-ends) of the user's choice. Figure 2 visualizes the different runtime components of the system. Our driver translates the D&C search tree into a graph. The reductor analyzes this graph, searching for thunks whose dependencies are fully evaluated; it sends these to a configured backend. When an executor returns values or subgraphs, the reductor updates its graph. When the graph is reduced to a single value, the reductor returns that value to the driver. For more details about the execution process, see [13].

To ease the development of gg-SAT, we built pygg, a python library for building dynamic gg dependency graphs. While gg is conceptually simple, using it typically requires programmers to write many different shell scripts for tasks such as embedding values in the gg graph, creating different kinds of thunks, and reformatting files for different solvers. With pygg, the entire computation can be expressed as a single python script. Different kinds of thunks are just different python functions, each of which can return basic python values, one or more files, or the output of some combination of other thunks. With pygg, our D&C implementation fits in a single python script of less than 200 lines. pygg has been merged upstream into the gg project.

IV. Experiments

gg-SAT is the first SAT solver targeting serverless computation, so we cannot compare with previous tools on our infrastructure of interest. Nonetheless, we perform two experiments. First, we compare gg-SAT with other multithreaded solvers on a single multicore machine, to validate the general architecture and performance of gg-SAT. Second, we use 1000 serverless executors to attempt unsolved benchmarks from the SAT 2020 competition, showing the utility of the massive parallelism that gg-SAT unlocks.

A. Local experiment

We compare with the default configurations of three parallel solvers: 1) the original Cube-and-Conquer prototype (denoted CnC) [1] [19]; 2) Paracooba[2] [18], a recent Cube-and-Conquer re-implementation that is optimized for distributed computing; 3) Treengeling [3] [8], a divide-and-conquer SAT solver; and 4) PLingeling [8], a state-of-the-art portfolio SAT solver. We evaluate on the benchmarks reported in [18], [19]. We run gg-SAT with $p_i = 64$, $p_s = 4$, $t = 10$, and $f = 1.5$, a set of parameters empirically determined to work well. For the other four solvers, we use the default parameters except that the number of threads is set to 64. Our testbed machines have two 2.70GHz Xeon Platinum 8280 CPUs, running CentOS 7. Each job is run with a 256 GB memory limit, and a 1-hour wall-clock timeout.

Table I shows the solvers' wall-clock runtime for each benchmark. Given the small set of benchmarks, we can

[1]https://github.com/marijnheule/CnC/tree/ee8f8aab3729b46bc92dc

[2]https://github.com/maximaximal/Paracooba/tree/d905b67304eb780

[3]https://github.com/arminbiere/lingeling/tree/7d5db72420b95ab (same for PLingeling)

TABLE I: Runtime (s) of gg-SAT, CnC, Paracooba, Treengeling, and PLingeling on the benchmarks reported in [18], [19]

benchmark	Result	gg-SAT	CnC	Paracooba	Treengeling	PLingeling
9dlx_vliw_at_b_iq8	UNSAT	850	–	966	–	155
9dlx_vliw_at_b_iq9	UNSAT	2830	–	1302	–	222
AProVE07-25	UNSAT	599	–	2091	1596	–
cruxmiter32.cnf	UNSAT	717	496	–	2078	–
dated-5-19-u	UNSAT	1723	436	1819	891	1030
eq.atree.braun.12	UNSAT	466	170	465	384	605
eq.atree.braun.13	UNSAT	3225	826	–	1615	1517
gss-24-s100	SAT	1166	–	–	1618	335
gss-26-s100	SAT	3509	–	–	560	–
gus-md5-14	–	–	–	–	–	–
ndhf_xits_09_UNS	UNSAT	948	–	–	–	1633
rbcl_xits_09_UNK	UNSAT	629	–	–	–	2965
rpoc_xits_09_UNS	UNSAT	331	–	–	–	1267
sortnet-8-ipc5-h19	SAT	–	–	3008	–	225
total-10-17-u	UNSAT	1098	388	919	310	666
total-5-15-u	UNSAT	–	1440	–	3253	–

draw only limited conclusions. Nonetheless, the results suggest gg-SAT's performance is reasonable. It solves more benchmarks than the other three divide-and-conquer solvers, corroborating past research [1] that interleaving look-ahead with CDCL can be beneficial. It also solves more than PLingeling, suggesting that the divide-and-conquer approach can be preferable to the portfolio approach in some cases. Note, however, that each other solver can solve at least one benchmark that gg-SAT cannot, suggesting that the approaches are complementary.

B. Serverless experiment

Our second experiment demonstrates the utility of the thousand-way parallelism that gg-SAT makes convenient. We find that with this parallelism, gg-SAT can solve challenging instances that are out of reach for solvers running at lower levels of parallelism.

We sample 8 instances from the Cloud track of the SAT Competition 2020 [5], none of which were solved during the competition.[4] As summarized in Table II, four of the five solvers from the previous section (using the same configurations) are unable to solve any of these instances within 4 hours. Treengeling solves one instance, Steiner-81-21-bce, in 9331 seconds. However, with gg-SAT running on AWS Lambda with 1000-way parallelism, we find that three instances: Steiner-81-21-bce, bv-term-small-rw_350.smt2, and mulhs16.smt2 are UNSAT in 2559, 1455, and 2866 seconds respectively. For AWS Lambda, we configure gg-SAT with $p_i = 1024$, $p_s = 8$, $t = 10$, and $f = 1.5$.[5]

[4] Steiner-81-21-bce, abw-I-ash85.mtx-w24, ccp-s8-facto4, bv-term-small-rw_350.smt2, Steiner-405-71-bce, mulhs16.smt2, LED_round_29-32_faultAt_29_fault_injections_5_seed_1579630418, PRESENT_round_1-32_faultAt_30_fault_injections_10_seed_1579630418

[5] Our experiment is incomparable with the results of the 2020 SAT cloud track. The competition environment differs substantially from our testbed; it uses 1600 cores, 20 minutes, and different hardware.

TABLE II: Solver performance on 8 hard instances from the SAT Competition 2020

Solver	Executor	Parallelism	Time Limit (h)	Solved
CnC	local threads	64	4	0
Paracooba	local threads	64	4	0
Treengeling	local threads	64	4	1
PLingeling	local threads	64	4	0
gg-SAT	local threads	64	4	0
gg-SAT	AWS Lambda	1000	1	3

V. Discussion

We have presented gg-SAT, a parallel D&C SAT solver compatible with serverless-computing. gg-SAT is built on top of gg, an infrastructure for evaluating parallel computations. gg-SAT appears competitive with other parallel SAT solvers, and easily unlocks ad-hoc large-scale parallelism through execution on serverless cloud-services. This massive parallelism appears to be effective in solving some challenging instances. To implement gg-SAT, we also built pygg, a novel python interface to gg, which we hope will be useful for other applications, such as parallel SMT solving.

Future Work: gg-SAT itself could be substantially improved. Currently, its search strategy (e.g., how many sub-problems to create, when to re-divide) is independent of the number of idle workers and the number of unsolved problems. This can cause one of two undesirable dynamics: most workers sitting idle while a few tackle challenging sub-problems (that would ideally be immediately divided) or too much time being spent re-dividing (even though all workers are already busy). In the future, we hope to adjust the search strategy depending on the current workload of the system, dividing more when workers are idle, and less when they are not. We suspect that this will improve performance while also reducing the number of parameters for the system.

Other future directions for gg-SAT include proof-generation, new dividers, and trying to retain useful clauses from failed base solver attempts.

REFERENCES

[1] T. Ahmed, O. Kullmann, and H. Snevily. On the van der Waerden numbers w (2; 3, t). *Discrete Applied Mathematics*, 174:27–51, 2014.

[2] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, 2018.

[3] G. Audemard and L. Simon. Lazy clause exchange policy for parallel sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 197–205. Springer, 2014.

[4] AWS lambda. https://docs.aws.amazon.com/lambda/index.html.

[5] T. Balyo, N. Froleyks, M. J. Heule, M. Iser, M. Järvisalo, and M. Suda. Proceedings of sat competition 2020: Solver and benchmark descriptions. 2020.

[6] T. Balyo, P. Sanders, and C. Sinz. Hordesat: A massively parallel portfolio sat solver. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172. Springer, 2015.

[7] A. Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In T. Balyo, M. Heule, and M. Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[8] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[9] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[10] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[11] W. Chrabakh and R. Wolski. Gridsat: Design and implementation of a computational grid application. *Journal of Grid Computing*, 4(2):177, 2006.

[12] T. Ehlers and D. Nowotka. Tuning parallel sat solvers. *Proceedings of Pragmatics of SAT*, 59:127–143, 2019.

[13] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019*, pages 475–488, 2019.

[14] S. Fouladi and B. Shacklett. R2-t2. https://github.com/r2t2-project/r2t2.

[15] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association.

[16] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[17] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2010.

[18] M. Heisinger, M. Fleury, and A. Biere. Distributed cube and conquer with paracooba. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–122. Springer, 2020.

[19] M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, volume 7261 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2011.

[20] M. J. Heule, M. Järvisalo, and M. Suda. Proceedings of sat race 2019: Solver and benchmark descriptions. 2019.

[21] M. J. H. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[22] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[23] A. E. Hyvärinen, T. Junttila, and I. Niemelä. A distribution method for solving sat in grids. In *International conference on theory and applications of satisfiability testing*, pages 430–435. Springer, 2006.

[24] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Partitioning sat instances for distributed solving. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 372–386, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[25] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[26] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Painless: a framework for parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 233–250. Springer, 2017.

[27] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Modular and efficient divide-and-conquer sat solver on top of the painless framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 135–151. Springer, 2019.

[28] S. Mijnders, B. De Wilde, and M. Heule. Symbiosis of search and heuristics for random 3-sat. *arXiv preprint arXiv:1402.4455*, 2014.

[29] S. Nejati, Z. Newsham, J. Scott, J. H. Liang, C. Gebotys, P. Poupart, and V. Ganesh. A propagation rate based splitting heuristic for divide-and-conquer solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 251–260. Springer, 2017.

[30] Y. Ngoko, C. Cérin, and D. Trystram. Solving sat in a distributed cloud: a portfolio approach. *International Journal of Applied Mathematics and Computer Science*, 29(2):261–274, 2019.

[31] C. Sinz, W. Blochinger, and W. Küchlin. Pasat—parallel sat-checking with lemma exchange: Implementation and applications. *Electronic Notes in Discrete Mathematics*, 9:205–216, 2001.

[32] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[33] H. Wu, A. Ozdemir, A. Zeljić, K. Julian, A. Irfan, D. Gopinath, S. Fouladi, G. Katz, C. Pasareanu, and C. Barrett. Parallelization techniques for verifying neural networks. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 128–137. IEEE, 2020.

[34] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[35] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.