2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

An Effective Distributed D&C Approach for the Satisfiability Problem

Gilles Audemard Benoît Hoessen Saïd Jabbour Cédric Piette
Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{audemard,hoessen,jabbour,piette}@cril.fr

***Abstract*—Most of state-of-the-art parallel SAT solvers are portfolio-based ones. They aim at running several times the same solver with different parameters. In this paper, we propose a solver called `Dolius`, based on the divide and conquer paradigm. In contrast to most current parallel efficient engines, `Dolius` does not need shared memory, can be distributed, and scales well when a large number of computing units is available.**

I. Introduction

The propositional satisfiability problem, called SAT, asks whether a given propositional formula in conjunctive normal form (CNF) has a satisfying assignment. SAT is the canonical NP-complete problem [1], and its practical resolution first came in the context of automated theorem proving. Today, many hard problems are translated into a SAT problem, which is now the cornerstone of multiple application domains, such that electronic design automation, computational biology, automated planning and formal verification to name of few.

These last years, the wide availability of cheap multicore platforms has made parallel SAT solving more and more popular, and this topic has received major attention lately. Numerous attempts have been made, but the focus is now on *portfolio* solvers, which aim at running on the initial problem, different versions of a same solver, with different parameters.

Nevertheless, portfolio-based techniques do not scale well when a large number of computing units is available. In this case, they clearly show their limits, and other ways to parallelize SAT solving should be envisaged. We propose in this paper a new distributed approach, based on the divide and conquer (D&C) paradigm. We compare this framework with the portfolio one in the SAT context, and introduce `Dolius`, our new D&C-based SAT solver.

This paper is organized as follows: in the next section, we briefly present background knowledge about SAT solving; we present state-of-the-art parallel techniques to solve SAT, and discuss the respective pros and cons of portfolio-based and divide and conquer approaches in Section III. Our new distributed SAT solver, called `Dolius`, is presented in details in Section IV and empirically evaluated in Section V. Next, we present different works related to divide and conquer techniques for SAT in Section VI, and finally conclude with some perspectives.

II. Technical Background

First of all, we assume that the reader is familiar with logic notions (variables, literal, clause, unit clause, interpretations, `CNF` formula). Many modern SAT solvers are called CDCL, for ”Conflict-Driven Clause Learning”, and they are getting far from the original DP [2] and DLL [3] procedures. Nowadays, high-performance SAT engines rely on different powerful mecanisms which we briefly recall in the following. A typical branch of a CDCL solver can be seen as a sequence of decisions followed by propagations, repeated until a conflict is reached. Each decision literal, chosen by some heuristic, usually activity-based ones, is assigned at its own level, shared with all propagated literals assigned at the same level. Each time a conflict is reached, a *nogood* is extracted using a particular method, usually the First UIP (Unique Implication Point) one [4], [5]. The learnt clause is then added to the learnt clause database and a *backjumping* level is computed from it. Periodically, restarts are performed. The interested reader can refer to [6] for a more detailled presentation of SAT solving.

In this paper, parallel CDCL-based solvers are considered. Such searches are called *workers* in the following, or *slaves* in the divide and conquer case. A worker (or slave) is said *active* if it is currently working, namely searching to a solution to the CNF. Otherwise, it is said *idle*.

III. PortFolio VS Divide and Conquer

Two main approaches are commonly explored to parallelize SAT solvers, namely portfolio and divide and conquer.

On the one hand, the parallel portfolio strategy exploits the complementarity between different sequential CDCL strategies to let them compete and cooperate on the same formula [7], [8], [9]. The general objective is to cover the space of good search strategies in the best possible way. In order to improve the capabilities of the portfolio solver, some have implemented communication of learnt clauses [10], [11]. Using this technique, better results are obtained, but they cannot improve greatly their results by increasing the computing ressources.

On the other hand, the *divide and conquer* idea divides the search space into subspaces, successively allocated to SAT workers. Each time a worker finishes its job (whereas the other ones are still doing their task), a load balancing strategy is invoked, and dynamically transfers subspaces to this idle worker [12], [13]. Unfortunately, this framework exhibits certain pathological cases in the SAT context, illustrated with the following example.

Example 1: Let ϕ be a CNF formula. $\Sigma = ((a \vee b) \wedge \phi) \wedge ((a \vee \neg b) \wedge \phi)$ is also a CNF formula. Dividing the search on either a or b causes some problems.

1) The Ping Pong Effect: If the search on Σ is divided using a, one of the subsequent task is very light, since it is easy to prove that $\Sigma \vDash a$. Hence, just using unit propagation, it is possible to show that $\Sigma \wedge (\neg a) \vDash \bot$. The slave that receives such (sub)-formula can prove it inconsistent without any exploration at all, and asks again for work very quickly. This is a problem, since work division has a cost, particularly because of network communication.

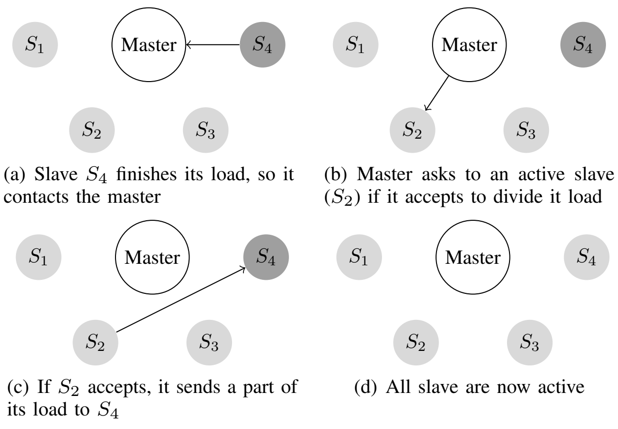
If bad choices are successively made when dividing the CNF, then one of the worker repetitively receives a trivial subproblem, and spends more time asking for work than actually solving the problem. This phenomenon is called *Ping-Pong effect* in earlier work [14].

2) Useless Division: Back to Example 1. If the search is divided on b, then each slave actually works on the same formula: $a \wedge \phi$. This is clearly not ideal, since redundant work has to be avoided as much as possible.

Furthermore, if ϕ does not admit any model, it is only proved inconsistent when the *last* slave delivers its answer. Note that using a portfolio on Σ, the *first* worker that finishes to prove the formula inconsistent puts an end to the global search. Hence, in such a situation, it would be desirable to divide the search with respect to a variable from $Var(\phi)$ rather than either a or b. Those results pled for a careful analysis of the division strategy, based on the concept of *guiding path*. This is studied in the Section IV-B of this paper.

IV. Main Features of Dolius

A. Work Stealing



(a) Slave S_4 finishes its load, so it contacts the master

(b) Master asks to an active slave (S_2) if it accepts to divide it load

(c) If S_2 accepts, it sends a part of its load to S_4

(d) All slave are now active

Fig. 1. Load Balance through Work Stealing Procedure of `Dolius`

Partitioning a CNF into subformulas that represent similar amount of work for several CDCL-based workers appears to be a difficult task. Ideally, all slaves should have an equivalent amount of work to achieve, yet predicting the difficulty of a given formula is very computationally heavy.

In order to be as opportunistic as possible with respect to available resources, a load balancing technique must then be implemented. Indeed, in practice, certain slaves finish their task before the others and become idle. `Dolius` is then given a well-known load balancing mechanism: *work stealing*. One of the advantage of such a solution is that any active slave does not have to listen to other (possibly) idle ones. Idle slaves ask for work by themselves.

The different steps of the work stealing process implemented in `Dolius` are illustrated in Figure 1. When a slave (denoted S_4) finished its task, it gets in touch with the Master to signal that it has no work left, (step 1(a)). The master asks to one of the active slave (denoted S_2 in Figure 1) if it accepts to divide its task (step 1(b)). If S_2 accepts, it comes into contact with S_4 to give it some of its load (step 1(d)). In this last step, slaves communicate directly without needing the master that is only contacted to put in touch a newly idle slave to an active one. In order to assure communication between the master and the workers, four different cases are depicted in Fig. 2.

1) The first case (Fig. 2(a)) depicts how the first worker can receive its workload. It uses a *work request* and the master responds by asking this slave to work on the initial problem.
2) The second case (Fig. 2(b)) represents an idle worker (*Worker S2*) requesting work to the master. The master puts the requesting worker on hold and contacts an active worker (*Worker S1*) to divide its search space. The latter contacts the requesting worker, transmits a guiding path and sends the clauses that can be applied to the search space. After acknowledgment from the receiver of this information, both workers send a *ready* message to the master. When both *ready* messages are received, the master finally sends a *start* message to both workers.
3) The third case (Fig. 2(c)) depicts a worker having ended its search with an *unsatisfiable* answer. The master acknowledges the answer and may tell the worker that a new *work request* can be made as the solution for the complete problem has not been found yet.
4) The fourth case (Fig. 2(d)) represents a worker that finds a *satisfiable* answer to the problem. This worker gets in contact with the master to warn it that there is no need to continue the overall search. The master sends a *stop* message (not depicted in Fig. 2(d)) to every worker in order to stop them.

In our implementation, a slave can refuse to divide its load only if one of these conditions occurs:

1) the slave has just found a model to the CNF, so there is no use to continue the search
2) the slave has just proved its subproblem is inconsistent, and has actually become idle
3) the slave is only working for a very few time (in practice $< x$ seconds, where x can be specified. By default, $x = 2.5$)
4) the slave is already dividing its task with another idle slave

In addition, when a slave becomes idle and contacts the master, this last one has to choose an active slave to ask

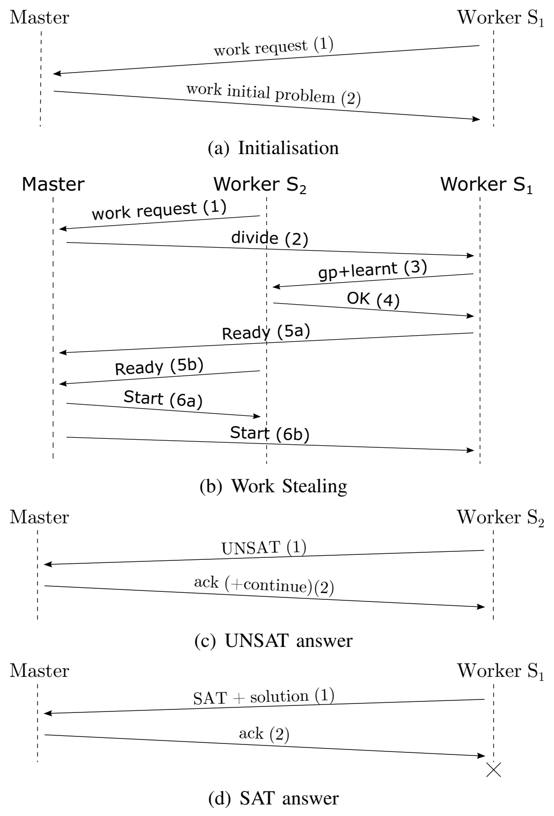


Fig. 2. Network Communication at different steps of `Dolius`

him to divide its load. Different criteria can be considered to choose this active slave. We propose the following work-stealing scheduling strategy: in `Dolius`, the list of active slaves are stored in a FIFO data structure, and when work is asked, the master chooses the first slave in this FIFO to get in touch with it. This choice has been made to avoid contacting the same active slave several times in a row in case of simultaneous requests for work to the master if more than one worker is active. Moreover, as some of our guiding path heuristics rely on VSIDS (see next Section), it is desirable to let the solver works with the CNF a certain amount of time to initialize the counters used by VSIDS. Indeed, the different counters within a CDCL-based worker need some time to provide conflicting variables, and this time is needed to make a more "accurate" division and to avoid one of the pathological cases depicted in Example 1.

B. Generation of Guiding Paths

When a work request has been sent, the way a slave divides its task is an essential factor for the general efficiency of a D&C algorithm. Such algorithms generally rely on the concept of *guiding path* for load balancing.

In most current implementations of D&C for SAT, the *guiding path* is reduced to a single variable. So, the active slave assigns (for itself) one the variables occuring in the CNF, and sends the opposite of this variable to the idle slave [15], [16]. Finding the best possible variable is actually both important and difficult.

This problem is the same than choosing a variable to branch in the sequential case, since it is about partitioning the search space to explore it in the best possible way. Unfortunately, it is well-known that picking the best variable to branch (in order to reduce as much as possible the subsequent search subspace) is NP-hard. Accordingly, guiding paths have to rely on some heuristical choices. However, these choices are much more important than in the sequential case. Indeed, as depicted by Example 1, one has to be careful while generating the guiding path: there exists different pathological cases that not only duplicate the load for several workers, but can also slow down the whole solving process.

Accordingly, we propose a new heuristical technique to choose the best variable for the guiding path. First, each worker is added with a counter for each variable of the considered formula. The counter of each variable, initially set to 0, is incremented whenever the variable is assigned in the partial assignation to the opposite value to its previous assignation. This enables to keep track of variables whose interpretation value very often changes, during the search.

When a worker is asked to divide its load, it uses this counter to select the 1% of the variables whose counters exhibit the highest values. To tie break those promising variables, we propose to use a so-called *"look ahead"* technique [17], [18]. By denoting $UP(\Sigma \wedge l)$ the number of literals that can be unit-propagated once the literal l is assigned, the selected variable to balance load is the one the maximize a function f (with $a \in Var(\Sigma)$) defined such that:

$$f(a) = 1024 \times UP(\Sigma \wedge a) \times UP(\Sigma \wedge \neg a) + UP(\Sigma \wedge a) + UP(\Sigma \wedge \neg a)$$

This function aims at selecting the variable whose assignment causes the most unit propagations, leading to the smaller possible subproblems to solve. Moreover, we choose this heuristic because it also enables to select a variable that tends to obtain subproblems of similar sizes, and hopefully with similar difficulty to solve. Indeed, let Σ be a CNF, $i, j, k \in Var(\Sigma)$, and the following number of propagations:

- $UP(\Sigma \wedge i)$ = 4 $UP(\Sigma \wedge \neg i)$ = 10
- $UP(\Sigma \wedge j)$ = 11 $UP(\Sigma \wedge \neg j)$ = 7
- $UP(\Sigma \wedge k)$ = 31 $UP(\Sigma \wedge \neg k)$ = 2

In front of this situation, one could argue that the most promising variable is k, since 31 unit propagations are triggered after its positive assignment. However, when assigned negatively, only 2 literals are propagated. The disequilibrium between the two branches (positive and negative) may very likely causes a *ping-pong effect*, since one worker deals with a much easier subproblem than the other one, that has to solve a subproblem similar in size to the original one. Hence, variables like k then has to be avoided to be used as guiding path.

In the opposite, variables i and j exhibit a better balance (w.r.t. propagated literals) after their positive and negative assignments. Using the look ahead heuristic, the different scores are $f(i) = 40974$, $f(j) = 78866$ and $f(k) = 63521$. In such a situation, j would be chosen as a guiding path. The intuition behind this choice is the need for a variable whose assignment produces two (preferably well-balanced) subproblems, much easier to solve.

V. Empirical Evaluation

We have experimentally tested `Dolius`, wherein each slave uses a modified version of the well known CDCL solver minisat [19]. The experimentations of this paper have been conducted on different computing units that exhibit the exact same features: dual socket `Intel XEON X5550` quad-core 2.66 GHz with 8 MB of cache and a RAM limit of 32GB, under Linux CentOS 6 (kernel 2.6.32). All machines are linked through a HP ProCurve 4108gl switch using a gigabit connection, allowing each node to communicate with any other node with a maximum speed of 1Gbps.

We did not perform any comparison with any other parallel solver for different reasons. First, we were not able to obtain any other distributed divide and conquer independent from the underlying architecture. Second, no comparison were made against portfolio solvers since they are mostly thread based, communicating through shared memory without using any locks.

The timeout is set to 1200 seconds wall clock (WC) for each instance. If no answer is delivered within this amount of time, the instance is considered unsolved. We use the instances from the application track of the 2011 SAT Competition. The measured time comes from the moment the master was started up to the moment that the last node receives a stop signal. Each test is performed five times.

Over the 300 instances, 141 were solved at least once. The obtained results can be divided in three categories. The first, about 54% (39 SAT, 38 UNSAT), consists of instances where a speedup can be observed. Out of this categorie are the instance depicted in Figure 3.

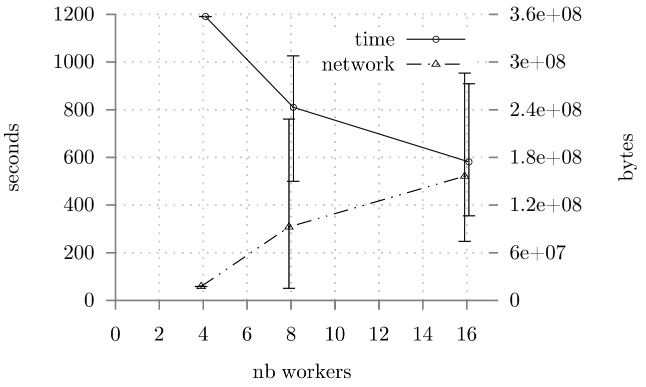


Fig. 3. Instance: UCG-15-10p1. Number of variables: 200003, number of clauses: 1019221, SAT

The first element that we have to point out for instance depicted at Fig. 3 is that using less than 4 workers, no answer is found under 1200 seconds. The acceleration for eight workers compared to four is 68%, 71% using sixteen workers. As for the network, with eight workers, we use 5.26 times more network than with four workers. With sixteen workers, we use 1.68 times more than with eight workers.

The second category consists of about 31% (29 SAT, 15 UNSAT). For those instance, generally solved quite fast, our approach does not provide any speedup. This can be explained by the fact that time is needed in order to activate every node, read the instance file, split the work, etc. However, as we are trying to provide a new framework to solve difficult instances, this category is not the most important to us.

The last category is where the problems are. They are about 14% (9 SAT, 11 UNSAT). Instances within this category tend to take more time as the number of resources increases. This is partially due to the fact that on this set of instances ping pong effects are still observed.

Table I provides some information about the 141 solved instances. It shows for each number of worker (#w column), how many instances are solved 0, 1, 2, 3, 4 or 5 times. The most significant evolution is the number of instance unsolved. Using only 1 worker, there are 48 of such instances. However, using 16 workers there are only nine instances of the 141 that are unsolved. This result shows clearly that dividing the instance is really beneficial as we were able to solve instances that are not solved using only one worker.

# w	0 time	1 time	2 times	3 times	4 times	5 times
1	48	1	4	1	0	87
2	35	5	4	5	8	84
4	26	8	6	5	7	89
8	22	9	6	6	8	90
16	9	6	12	8	20	86

TABLE I. Number of solved instances w.r.t. number of workers

Finally, Table II provides some details for representing instances. For different number of workers it gives the average running time (or "—" if timeout is reached for all 5 runs) and the average number of work divisions. The first four instances come from the category where a speedup is observed. Note for example that `vmpc` instance can be solved only using 16 workers. In such instances, using only 2 workers leads to too few work divisions, here the divide and conquer approach seems to be the good choice. The instance `rand_net60-30-1.shuffled` is a typical too easy instance: there are no division. The last instance provides an example where increasing the number of workers leads to increase the running time. Note that with only 1 worker the instance is solved in 37 seconds and (of course) without divisions and adding workers increases (obviously) the number of divisions but decreases the running time. We suppose that this is a typical ping-pong effect.

VI. Related Works

Different divide and conquer algorithms have been proposed in the past to solve SAT.

First, the `SAT@HOME` project [20] initiated in 2010, relies on the open source system for grid computing `BOINC` [21] to achieve a massive divide and conquer solving. This work inherits of the architecture of the famous `SETI@HOME` project, which aims at distribute to volunteers the computation resulting from the analysis of radio signals, searching for signs of extra terrestrial intelligence. `SAT@HOME` is based on a static load balance, namely the instance is divided before the actual search starts, and cannot be modified latter.

There also exists other divide and conquer implementations for SAT. For instance, Feldman *et al.* [22] propose such an

instance	SAT ?	1 worker		2 worker		4 worker		8 worker		16 worker	
		time	div	time	div	time	div	time	div	time	div
`AProVE07-21`	UNSAT	415	0	213	3	127	11	84	28	**65**	47
`md5_48_3`	SAT	931	0	471	1	294	3	220	9	**219**	19
`rand_net60-40-10.shuffled`	UNSAT	—	—	753	6	301	22	146	47	**101**	97
`vmpc_36.renamed-as.sat05-1922`	SAT	—	—	—	—	—	—	668	13	**214**	40
`rand_net60-30-1.shuffled`	UNSAT	**26**	0	**26**	0	**26**	0	**27**	0	**27**	0
`hsat_vc12062`	UNSAT	**37**	0	62	7	139	55	176	145	218	348

TABLE II. AVERAGE RUNNING TIME IN SECONDS ON 5 RUNS FOR A SELECTION OF INSTANCES, WITH AN INCREASING NUMBER OF WORKERS

implementation, that is able to share nogoods between the different slaves. However, this nogood sharing is done in a central memory that must be common to all slaves, making this implementation impossible to distribute to different computers.

Let us also cite the `GridSat` [23] portal. Initiated in 2000, it aims at providing a public easy-to-use interface to perform SAT solving using widely distributed and possibly heterogeneous resources. This ambitious portal is unfortunately old, and does not exploit recent advances in SAT solving. Indeed, it is based on `Chaff`, the solver that first introduced *watched literals*. However, this solver is not representative of state-of-the-art solvers anymore.

Finally, Schulz and Blochinger [24] have proposed an original approach to distribute SAT. Indeed, in contrast to most other distributed techniques where a central point (often called *master*) is needed, their contribution consists in proposing a full *peer-to-peer* (P2P) system, where all workers play exactly the same role, without any need of centralization. The resulting P2P system is called `SatCiety` [24].

VII. CONCLUSION

In this paper, we have proposed an all new framework for solving SAT using divide and conquer techniques. We have illustrated the empirical interest of such an approach with intensive experiments.

In future works, we plan to make `Dolius` even more generic, and to propose an API to enable any solver to be used in our framework with only few minor changes, provided that it is able to divide its task. We also plan to use some so-called ”formula caching” methods to improve again the practical behavior our framework. Some of them have been proposed recently (see e.g. [25]), and we believe that such techniques make sense in the divide and conquer framework.

ACKNOWLEDGMENTS

This work has been supported by CNRS and OSEO, under the ISI project “Pajero”.

REFERENCES

[1] S. Cook, “The complexity of theorem proving procedures,” 1971.

[2] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *J. ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[3] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *J. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[4] J. Marques-Silva and K. Sakallah, “GRASP - A New Search Algorithm for Satisfiability,” in *ICCAD'96*, 1996, pp. 220–227.

[5] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, “Efficient conflict driven learning in boolean satisfiability solver,” in *proceedings of ICCAD*, 2001, pp. 279–285.

[6] A. Darwiche and K. Pipatsrisawat, *Complete Algorithms*. IOS Press, 2009, ch. 3, pp. 99–130.

[7] O. Roussel, “ppfolio,” http://www.cril.univ-artois.fr/~roussel/ppfolio.

[8] A. Biere, “(p)lingeling,” http://fmv.jku.at/lingeling.

[9] S. Kottler and M. Kaufmann, “SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing,” in *POS'11*, 2011.

[10] Y. Hamadi, S. Jabbour, and L. Saïs, “Manysat: a parallel SAT solver.” *JSAT*, vol. 6, pp. 245–262, 2009.

[11] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette, “Revisiting clause exchange in parallel SAT solving,” in *SAT'12*, 2012, pp. 200–213.

[12] W. Chrabakh and R. Wolski, “GrADSAT: A parallel SAT solver for the grid,” UCSB, Tech. Rep., 2003.

[13] G. Chu, P. J. Stuckey, and A. Harwood, “Pminisat: a parallelization of minisat 2.0,” SAT Race, Tech. Rep., 2008.

[14] B. Jurkowiak, C. M. Li, and G. Utard, “Parallelizing satz using dynamic workload balancing,” *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 174–189, 2001.

[15] R. Martins, V. Manquinho, and I. Lynce, “Improving search space splitting for parallel SAT solving,” in *ICTAI'10*, 2010, pp. 336–343.

[16] A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä, “Grid-based SAT solving with iterative partitioning and clause learning,” in *CP'11*, 2011, pp. 385–399.

[17] R. G. Jeroslow and J. Wang, “Solving propositional satisfiability problems,” *Ann. Math. Artif. Intell.*, vol. 1, pp. 167–187, 1990.

[18] D. Le Berre, “Exploiting the real power of unit propagation lookahead,” *Electronic Notes in Discrete Mathematics*, vol. 9, pp. 59–80, 2001.

[19] N. Een and N. Sörensson, “An extensible SAT-solver,” in *SAT'03*, 2003, pp. 502–518.

[20] M. Posypkin, A. Semenov, and O. Zaikin, “Sathome web page,” http://sat.isa.ru/pdsat.

[21] C. Ries, *BOINC: Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing*. Springer, 2012.

[22] Y. Feldman, N. Dershowitz, and Z. Hanna, “Parallel multithreaded satisfiability solver: Design and implementation,” *Electr. Notes Theor. Comput. Sci.*, vol. 128, no. 3, pp. 75–90, 2005.

[23] W. Chrabakh and R. Wolski, “The gridsat portal: a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 6, pp. 795–808, 2007.

[24] S. Schulz and W. Blochinger, “Parallel SAT solving on peer-to-peer desktop grids,” *J. of Grid Computing*, vol. 8, no. 3, pp. 443–471, 2010.

[25] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi, “Combining component caching and clause learning for effective model counting,” in *in Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004.