Annales Mathematicae et Informaticae
***42** (2013) pp. 9–21*
`http://ami.ektf.hu`

Cube-and-Conquer approach for SAT solving on grids*

Csaba Biró[a], Gergely Kovásznai[b], Armin Biere[c], Gábor Kusper[a], Gábor Geda[a]

[a]Eszterházy Károly College
`birocs,gkusper,gedag@aries.ektf.hu`

[b]Vienna University of Technology
`kova@forsyte.tuwien.ac.at`

[c]Johannes Kepler University
`biere@jku.at`

Submitted May 10, 2013 — Accepted December 9, 2013

Abstract

Our goal is to develop techniques for using distributed computing resources to efficiently solve instances of the propositional satisfiability problem (SAT). We claim that computational grids provide a distributed computing environment suitable for SAT solving. In this paper we apply the Cube and Conquer approach to SAT solving on grids and present our parallel SAT solver `CCGrid` (Cube and Conquer on Grid) on computational grid infrastructure.

Our solver consists of two major components. The master application runs `march_cc`, which applies a lookahead SAT solver, in order to partition the input SAT instance into work units distributed on the grid. The client application executes an `iLingeling` instance, which is a multi-threaded CDCL SAT solver. We use BOINC middleware, which is part of the SZTAKI Desktop Grid package and supports the Distributed Computing Application Programming Interface (DC-API). Our preliminary results suggest that our approach can gain significant speedup and shows a potential for future investigation and development.

Keywords: grid, SAT, parallel SAT solving, lookahead, march_cc, iLingeling, SZTAKI Desktop Grid, BOINC, DC-API

*Supported by Austro-Hungarian Action Foundation, project ID: 83öu17.

1. Introduction

Propositional satisfiability is the problem of determining, for a formula of the propositional logic, if there is an assignment of truth values to its variables for which that formula evaluates to true. By SAT we mean the problem of propositional satisfiability for formulas in conjunctive normal form (CNF). SAT is one of the most-researched NP-complete problems [8] in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification [5]. Also it should be noted that the hardness of the problem is caused by the possibly increasing number of the variables, since by a fixed set of variables SAT and n-SAT are regular languages and therefore there is a deterministic (theoretical) linear time algorithm to solve them, see, [21, 23].

Modern sequential SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) [9] algorithm. This algorithm performs Boolean constraint propagation (BCP) and backtrack search, i.e., at each node of the search tree it selects a decision variable, assigns a truth value to it, and steps back when conflict occurs. Conflict-driven clause learning (CDCL) [5, Chpt. 4] is based on the idea that conflicts can be exploited to reduce the search space. If the method finds a conflict, then it analyzes this situation, determines a sufficient condition for this conflict to occur, in form of a learned clause, which is then added to the formula, and thus avoids that the same conflict occurs again. This form of clause learning was first introduced in the SAT solver `GRASP` [19] in 1996. Besides clause learning, lazy data structures are one of the key techniques for the success of CDCL SAT solvers, such as "watched literals" as pioneered in 2001, by the CDCL solver `Chaff` [20, 18] . Another important technique is the use of the VSIDS heuristics and the first-UIP backtracking scheme. In the state-of-the-art CDCL solvers, like `PrecoSAT` and `Lingeling` [3, 4], several other improvements are applied. Besides enhanced preprocessing techniques like e.g. failed literal detection, variable elimination, and blocked clause elimination, clause deletion strategies and restart policies have a great impact to the performance of the CDCL solver.

Lookahead SAT solvers [5, Chpt. 5] combine the DPLL algorithm with lookaheads, which are used in each search node to select a decision variable and at the same time to simplify the formula. One popular way of lookahead measures the effect of assigning a certain variable to a certain truth value: BCP is applied, and then the difference between the original clause set and the reduced clause set is measured (by using heuristics). In general, the variable for which the lookahead on *both* truth values results in a large reduction of the clause set is chosen as the decision variable. The first lookahead SAT solver was `posit` [10] in 1995. It already applied important heuristics for pre-selecting the "important" variables, for selecting a decision variable, and for selecting a truth value for it. The lookahead solvers `satz` [17] and `OKsolver` [16] further optimized and simplified the heuristics, e.g., `satz` does not use heuristics for selecting a truth value (rather prefers *true*), and `OKsolver` does not apply any pre-selection heuristics. Furthermore, `OKsolver` added improvements like local learning and autarky reasoning. In 2002, the solver

`march` [13] further improved the data structures and introduced preprocessing techniques. As a variant of `march`, `march_cc` [14] can be considered as a case splitting tool. It produces a set of cubes, where each cube represents a branch cutoff in the DPLL tree constructed by the lookahead solver. It is also worth to mention that `march_cc` outputs learnt clauses as well, which represent refuted branches in the DPLL tree. The resulting set of cubes represents the remaining part of the search tree, which was not refuted by the lookahead solver itself.

There are two types of basic appearance of parallelism in computations, the "and-parallelism" and the "or-parallelism" [22]. The first is used in high performance computing, while the latter is more similar to nondeterministic guesses (data parallel). SAT can (theoretically effectively) be solved by several new computing paradigms using or-parallelism and by using, roughly speaking, exponential number of threads. Since multi-core architectures are common today, the need for parallel SAT solvers using multiple cores has increased considerably.

In essence, there are two approaches to parallel SAT solving [12]. The first group of solvers typically follow a divide-and-conquer approach. They split the search space into several subproblems, sequential DPLL workers solve the subproblems, and then these solutions are combined in order to create a solution to the original problem. This first group uses relatively intensive communication between the nodes. They do for example load balancing, and dynamic sharing of learned clauses.

The second group apply portfolio-based SAT solving. The idea is to run independent sequential SAT solvers with different restart policies, branching heuristics, learning heuristics, etc. `ManySAT` [11] was the first portfolio-based parallel SAT solver. `ManySAT` applies several strategies to the sequential SAT solver `MiniSAT`. `Plingeling` [3, 4] follows a similar approach, and uses the sequential SAT solver `Lingeling`. In most of the state-of-the-art portfolio-based parallel SAT solvers (e.g. `ppfolio`, `pfolioUZK`, `SATzilla`) not only different strategies, but even different sequential solvers compete and, to a limited extent, cooperate on the same formula. In such approaches there is no load balancing and the communication is limited to the sharing of learned clauses.

`GridSAT` [7, 6] was the first complete and parallel SAT solver employing a grid. It belongs to the divide-and-conquer group. It is based on the sequential SAT solver `zChaff`. Besides achieving significant speedup in the case of some (satisfiable and even unsatisfiable) instances, `GridSAT` is able to solve some problems for which sequential `zChaff` exceeds time out. `GridSAT` distributes only the short learned clauses over the nodes, therefore it minimizes the communication overhead. Search space splitting is based on the selection of a so-called pivot variable x on the second decision level, and then creating two subproblems by adding a new decision on x resp. $\neg x$ to the first decision level. If sufficient resources are available, the subproblems can further be partitioned recursively. Each new subproblem is defined by a clause set, including learned clauses, and a decision stack.

[15] proposes a more sophisticated approach, based on using "partition functions", in order to split a problem into a fixed number of subproblems. Two partition functions were compared, a scattering-based and a DPLL-based one with

lookahead. A partition function can be applied even in a recursive way, by repartitioning difficult subproblems (e.g., the ones that exceeds time out). For some of the experiments, an open source grid infrastructure called Nordugrid was used.

`SAT@home` [25] is a large volunteer SAT-solving project on grid, which involves more than 2000 clients. The project is based on the Berkeley Open Infrastructure for Network Computing (BOINC) [1], which is an open source middleware system for volunteer grid computing. On top of BOINC, the project was implemented by using the SZTAKI Desktop Grid [24], which provides the Distributed Computing Application Programming Interface (DC-API), in order to simplify the development, and then also to deploy and distribute applications to multiple grid environments. [25] proposes a rather simple partitioning approach: given a set of n selected variables, called a decomposition, a set of 2^n subproblems is generated. The key issue is how to select a decomposition. One way to solve this issue, is to derive the set of "important" decomposition variables from the original problem formulation, which, however, then is problem-specific, and needs human guidance. For instance, in the context of SAT-based cryptoanalysis of keystream generators, a decomposition set can be obtained from the encoding of the initial state of the linear feedback shift registers [25]. `SAT@home` uses no data exchange among clients.

Our approach, called `CCGrid`, also uses BOINC and the SZTAKI Desktop Grid, as it is detailed in Sect. 3, but is based on the Cube and Conquer approach [14]. For partitioning the input problem, we use `march_cc`. Our approach differs from the previous ones in the fact that it uses a parallel SAT solver, `iLingeling`, for solving the particular subproblems, on each client. In Sect. 4 we present some experiments and preliminary results.

2. Preliminaries

Given a Boolean variable x, there exist two *literals*, the positive literal x and the negative literal $\overline{x}$. A *clause* is a disjunction of literals, a *cube* is a conjunction of literals. Either a clause or a cube can be considered as a finite set of literals.

A *truth assignment* for a (finite) clause set or cube set F is a function ϕ that maps literals in F to $\{0, 1\}$, such that if $\phi(x) = v$, then $\phi(\overline{x}) = 1 - v$. A clause resp. cube C is satisfied by ϕ if $\phi(l) = 1$ for some resp. every $l \in C$. A clause set resp. cube set F is satisfied by ϕ if ϕ satisfies C for every resp. some $C \in F$.

For representing the input clause set for a SAT solver, the DIMACS CNF format is commonly used, which references a Boolean variable by its (1-based) index. A negative literal is referenced by the negated reference to its variable. A clause is represented by a sequence of the references to its literals, terminated by a "0". The iCNF format extends the CNF format with a cube set.[1] A cube, called an assumption, is represented by a leading character "a" followed by the references to its literals and a terminating "0".

[1] `http://users.ics.tkk.fi/swiering/icnf/`

3. Architecture

Our application is a variant of the Cube and Conquer approach [14] and consists of two major components: a master application and a client application. The master is responsible for dividing the global input data into smaller chunks and distributing these chunks in the form of work units. Interpreting the output generated by the clients out of the work units and combining them to form a global output is also the job of the master. The architecture is depicted in Fig. 1. Similar to [25], the environment for running our system is the SZTAKI Desktop Grid [24] and BOINC [1], and was implemented by the use of the DC-API.

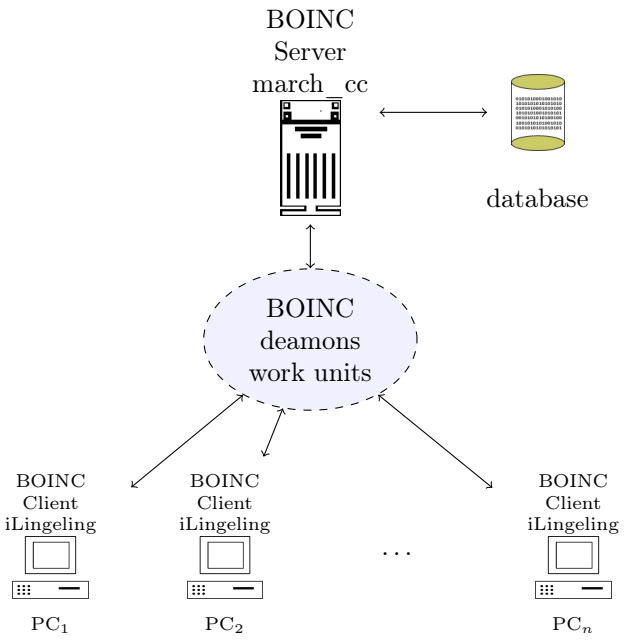

Figure 1: CCGrid architecture

The master

The master executes a partitioning tool called `march_cc` [14], which is based on the lookahead SAT solver `march`. Given a CNF file, `march_cc` primarily tries to refute the input clause set. If this does not succeed, `march_cc` outputs a set of assumptions (cubes) that describe the cutoff branches in the DPLL tree. These assumptions cover all subproblems of the input clause set that have not been refuted during the partitioning procedure. Given these assumptions, the master application creates work units, each of which consists of the input CNF file and a slice of the

assumption set. As it can be seen in Fig. 2, if one of the clients reports one of the work units to be satisfiable, then the master outputs the satisfying model and destroys all the running work units. If every clients report unsatisfiability, then the master outputs unsatisfiability.

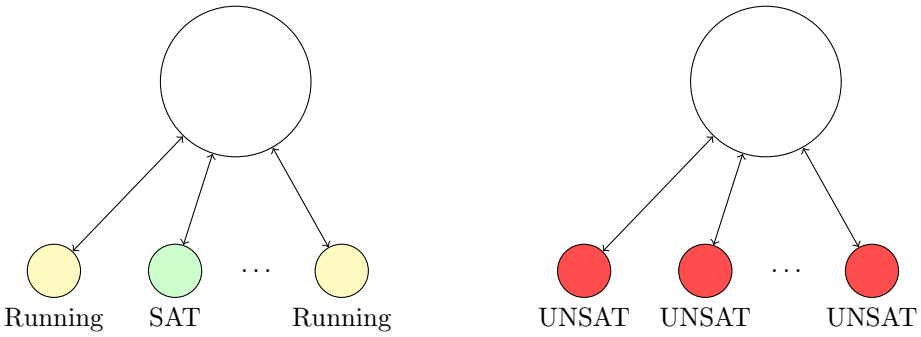

Figure 2: (a) If the problem is SAT, it is enough to find a SAT derived instance. (b) If the problem is UNSAT, one must show all derived instances UNSAT.

The pseudocode below shows how the master application works. It is divided into three procedures; the MAIN procedure is shown in Algorithm 1. It shares two constants with the other procedures: (i) *maxAsmCount* defines the maximum number of assumptions per work unit; (ii) *rfsInterval* gives a refresh interval at which DC-API events are processed. The master application uses several global variables; all of them are self-explanatory. In loop 6-9, work units are created, by calling the procedure CREATEWORKUNIT. Loop 10-13 then processes DC-API events generated by those work units that have finished solving their subproblems. Processing DC-API events is done by calling a callback function which has been previously set to PROCESSWORKUNITRESULT in line 3. The loop stops if either one of the work units returns a SAT result or all the work units completed.

CREATEWORKUNIT, shown in Algorithm 2, creates and submits a work unit to the grid. First, the CNF file is added to the new work unit. Then, in the loop, at most *maxAsmCount* assumptions from *asmFile* are copied into the new file *asmChunkFile*. Note that *asmFile* is global, it has been opened by the MAIN procedure (Algorithm 1, line 4), and therefore its current file position is held. Finally, *asmChunkFile* is added to the work unit, which is then submitted to the grid.

As already mentioned, PROCESSWORKUNITRESULT, shown in Algorithm 3, works as a callback function for DC-API events. It processes the result returned by a work unit.

Algorithm 1 Master: main procedure

```
Require: global constants maxAsmCount, rfsInterval
Require: global variables cnfFile, asmFile, wuCount, res, resFile
 1: procedure Main
 2:     initialize DC-API master
 3:     set ProcessWorkUnitResult as result callback
 4:     open asmFile
 5:     wuCount ← 0
 6:     while not EOF(asmFile) do
 7:         CreateWorkUnit
 8:         wuCount ← wuCount + 1
 9:     end while
10:     while res ≠ SAT and wuCount > 0 do
11:         wait rfsInterval
12:         process DC-API events
13:     end while
14:     if res ≠ SAT then
15:         res ← UNSAT
16:         cancel all work units
17:     end if
18: end procedure
```

Algorithm 2 Master: creating work units

```
 1: procedure CreateWorkUnit
 2:     wu ← new work unit
 3:     wu.cnfFile ← cnfFile
 4:     asmChunkFile ← new file
 5:     for i ← 1 to maxAsmCount do
 6:         if EOF(asmFile) then
 7:             break
 8:         end if
 9:         copy next assumption from asmFile to asmChunkFile
10:         i ← i + 1
11:     end for
12:     wu.asmFile ← asmChunkFile
13:     submit wu to the grid
14: end procedure
```

Algorithm 3 Master: processing work unit result

```
1: procedure ProcessWorkUnitResult(wu)
2:     if wu.res = SAT then
3:         res ← SAT
4:         copy wu.resFile to resFile
5:     end if
6:     wuCount ← wuCount − 1
7: end procedure
```

The client

Each client executes the parallel CDCL solver `iLingeling` [14, 4], for a fixed number of threads. Each thread executes a separate `lingeling` instance. `iLingeling` expects as input an iCNF file, including 1 or more assumptions, which is then loaded into a working queue. Each `lingeling` instance reads the input clause set, and then, in each iteration, gets the first assumption from the working queue.

If one of the `lingeling` instances can prove that the clause set is satisfiable under the given assumptions, then `iLingeling` reports that the clause set itself is satisfiable, the satisfying model is returned, and hence the remaining assumptions in the working queue can be ignored. Otherwise, i.e., if a `lingeling` instance reports unsatisfiability, then the assumption is retrieved from the working queue and the same SAT solver instance continues with the solving procedure. If the working queue becomes empty, then `iLingeling` reports that the clause set under the given set of assumptions is unsatisfiable.

Algorithm 4 shows the client's MAIN procedure. It uses one global constant, *thrCount*, which specifies the number of worker threads to use. First, the procedure creates an `iLingeling` instance with *thrCount* worker threads, loads both the CNF and the assumption files, and runs `iLingeling`. In loop 7-12, the results by all the threads are checked: if any of them is SAT then the result for the work unit is SAT; otherwise it is UNSAT (line 14). The result, as well as the satisfying model, is written into a result file by the procedure CREATERESULTFILE, shown in Algorithm 5.

Algorithm 4 Client: main procedure

Require: global constant *thrCount*
1: **procedure** MAIN(*wu*)
2: initialize DC-API client
3: *iLingeling* ← new iLingeling instance using *thrCount* threads
4: load *wu.cnfFile* into *iLingeling*
5: load *wu.asmFile* into *iLingeling*
6: run *iLingeling*
7: **for** $i \leftarrow 1$ **to** *thrCount* **do**
8: **if** ith thread's result is SAT **then**
9: *wu.res* ← SAT
10: **break**
11: **end if**
12: **end for**
13: **if** $i >$ *thrCount* **then**
14: *wu.res* ← UNSAT
15: **end if**
16: CREATERESULTFILE(*wu*, *iLingeling*)
17: **end procedure**

Algorithm 5 Client: creating result file

```
1: procedure CreateResultFile(wu, iLingeling)
2:     resFile ← new file
3:     write wu.res into resFile
4:     if wu.res = SAT then
5:         model ← satisfying assignment from iLingeling
6:         write model into wu.resFile
7:     end if
8:     wu.resFile ← resFile
9: end procedure
```

4. Results and testing environment

Our implementation consists of a quad-core SUN server with 6 GB memory, used as a master, and 20 quad-core PCs with 2 GB memory, used as clients. In our experiments, we used instances from the SAT Challenge 2012, from the Application (SAT + UNSAT) and the Hard Combinatorial (SAT + UNSAT) tracks. Results are presented in Tab. 1 and Tab. 2. The 1st column represents the instance's name. In the 2nd column, **A** resp. **HC** denotes Application resp. Hard Combinatorial problems. The 4th column shows the number of cubes, generated by `march_cc`. The 3rd resp. 5th column shows the runtime of `march_cc` resp. `iLingeling`, being executed on the master. The 6th column contains the sum of the previous two numbers, which represents the overall runtime of the cube-and-conquer approach running on a single (quad-core) machine. The total runtime of `CCGrid` is shown in the 7th column, while the 8th column measures the speedup as the ratio of the runtimes in the 6th and 7th columns.

In our approach, `CCgrid` have been executed without any communication among clients. Even though they do not cooperate and do not exchange learnt clauses, `CCGrid` shows a wide range of speedups. We achieved speedup up to ca. 8.5 on UNSAT instances (*QG-gensys-icl003.sat05-2715.reshuffled-07*) and up to ca. 7 on SAT instances (*sgen1-sat-160-100*).

Since the master has to distribute quite large work units over the network, communication overhead matters in the case of small instances, where communication costs are significant compared to the input size. Therefore, although we used a 1Gbps LAN in our experiments, cube-and-conquer running on a single machine outperformed `CCgrid` on some instances. If we look at the *battleship-16-31-sat* row in Tab. 1, we can see that `march_cc` and `iLingeling` can solve this problem on 1 client a bit faster than `CCGrid` on 20 clients.

In the case of satisfiable instances, we might be lucky, finding a model quickly, or unlucky. If there are many satisfying models, then it is not worth to distribute the problem over many clients. However, if there exist only a few models, then it is a good idea to use many clients, since the more clients we use, the more probable it is for a client to be lucky enough to find one of those few solutions. Unfortunately, we have no information about how many models the instances in Tab. 1 have.

		march_cc	# of cubes	iLingeling	cube-and-conquer	CCGrid	speedup
# of clients		**1**		**1**	**1**	**20**	
vmpc_26	**A**	8.38	296	40.22	48.6	13.64	**3.56**
AProVE09-07	**A**	65.93	4245	19.12	85.05	79.16	**1.07**
clauses-4	**A**	29.68	25	59.01	88.69	81.98	**1.08**
gss-16-s100	**A**	155.28	6292	201.21	356.49	171.71	**2.08**
IBM_FV_2004_rule_batch_22_SAT_dat. k65	**A**	17.93	361	148.95	166.88	154.02	**1.08**
ezfact64_3. sat05-450. reshuffled-07	**HC**	458.71	469428	63.71	522.42	505.72	**1.03**
sgen1-sat-160-100	**HC**	10.65	210168	419.92	430.57	62.28	**6.91**
em_7_4_8_exp	**HC**	20.06	19419	170.9	190.96	46.47	**4.11**
battleship-16-31-sat	**HC**	174.89	91757	2.69	177.58	180.89	**0.98**
Hidoku_enu_6	**HC**	125.02	256225	91.61	216.63	159.31	**1.36**

Table 1: Runtimes and speedup
all instances are SAT

`CCGrid` seems to be much better in distributing satisfiable instances from the **HC** track than the ones from the **A** track, since `march_cc` seems to generate much more cubes for the previous ones.

In the case of unsatisfiable instances, we cannot be lucky to find an early solution since there is no satisfying model. When comparing the speedups in Tab. 1 and Tab. 2, we can see that speedups around 1 are more frequent on satisfiable instances.

This shows that in the case of unsatisfiable instances there is less risk of wasting resources without any speedup.

5. Future work and conclusion

This paper presents a first attempt of applying the Cube and Conquer approach [14] to computational grids. We presented the parallel SAT solver `CCGrid`, which runs on the MTA SZTAKI Grid using BOINC. In this version, the master application applies `march_cc`, using a lookhead solver, to split a SAT instance. The client application uses the parallel SAT solver `iLingeling` to deal with several assumptions. The client creates a separate `iLingeling` instance for each work unit, and destroys it after completing the work unit. For the sake of improving our current results, in future work, we would like to preserve the state of `iLingeling` instances, including learnt clauses.

In our experience, the cube generation phase implemented in `march_cc` makes up a significant part of the runtime. As a consequence, we were mostly able to achieve significant speedup on such instances on which the cube generation phase

		march_cc	# of cubes	iLingeling	cube-and-conquer	CCGrid	speedup
# of clients		**1**	**1**	**1**	**20**		
counting-clqcolor-unsat-set-b-clqcolor-08-06-07.sat05-1257.reshuffled-07	**A**	5.77	112757	12.77	18.54	8.71	**2.13**
gensys-ukn002.sat05-2744. reshuffled-07	**A**	12.70	21408	230.01	242.71	71.55	**3.39**
Q32inK09	**A**	12.15	5279	35.89	48.04	14.38	**3.34**
QG6-dead-dnd002.sat05-2713. reshuffled-07	**A**	2.38	12147	35.79	38.17	4.74	**8.05**
QG-gensys-icl003.sat05-2715.reshuffled-07	**A**	25.43	38466	291.05	316.48	37.24	**8.49**
instance_n6 i7_pp_ci_ce	**A**	103.78	29290	111.72	215.50	117.20	**1.84**
AProVE07-09.cnf	**A**	34.43	86048	4.55	37.98	37.46	**1.01**
battleship-10-10-unsat	**HC**	0.36	2317	15.47	15.83	4.48	**3.53**
rand_net60-40-10.shuffled	**HC**	111.23	130227	13.24	124.47	115.62	**1.08**
smtlib-qfbv-aigs-ext_con_032008_0256-tseitin	**HC**	67.24	22384	7.29	74.53	71.47	**1.04**

Table 2: Runtimes and speedup
all instances are UNSAT

took a relatively short time. Therefore, our further aim to reduce the time spent on cube generation by parallelizing the look-ahead solver. We plan to adapt `march_cc` to a cluster infrastructure and to investigate the possibility of merging our BOINC-based approach with a cluster-based master application. We expect further improvement by analyzing the generated cubes and then, based on the result of the analysis, partitioning the cube set in a more sophisticated way.

In order to achieve larger speedup on unsatisfiable instances, it might be useful to call `march_cc` not only while partitioning the original problem, but also for repartitioning difficult subproblems, e.g., those on which a client exceeds a certain time limit. Finally, it might be interesting to apply similar techniques not only to clusters resp. grids, but also to cloud computing platforms.

References

[1] D. P. Anderson, BOINC: A System for Public-Resource Computing and Storage. *Proc. of GRID 2004*, pp. 4–10, 2004.

[2] A. Balint, A. Belov, D. Diepold, A. Gerber, M. Järvisalo, C. Sinz (eds), Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions. *Department of Computer Science Series of Publications B*, vol. B-2012-2, University of Helsinki, 2012.

[3] A. Biere, Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report 10/1, FMV Reports Series, JKU, 2010.

[4] A. Biere, Lingeling and Friends Entering the SAT Challenge 2012. *Haifa Verification Conference*, *Department of Computer Science Series of Publications B*, vol. B-2012-2, pp. 33–34, 2012.

[5] A. Biere, M. Heule, H. van Maaren, T. Walsh, Handbook of Satisfiability. IOS Press, Amsterdam, 2009.

[6] W. Chrabakh, R. Wolski, GridSAT: A Chaff-based Distributed SAT Solver for the Grid. *Proc. of SC'03*, pp. 37–49, 2003.

[7] W. Chrabakh, R. Wolski, GrADSAT: A Parallel SAT Solver for the Grid. Technical report, UCSB Computer Science, 2003.

[8] S. A. Cook, The Complexity of Theorem-Proving Procedures. Proc. of STOC'71, pp. 151–158, 1971.

[9] M. Davis, G. Logemann, D. Loveland, A Machine Program for Theorem Proving. *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[10] J.W. Freeman, Improvements to Propositional Satisfiability Search Algorithms. Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, May, 1995.

[11] Y. Hamadi S. Jabbour, L. Sais, ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, pp. 245–262, 2009.

[12] Y. Hamadi, S. Jabbour, C. Piette, L. Saïs, Deterministic Parallel DPLL. *JSAT*, vol. 7. no. 4, pp. 127–132, 2011.

[13] M. Heule, March: Towards a Look-ahead SAT Solver for General Purposes. Master thesis, TU Delft, The Netherlands, 2004.

[14] M. Heule, O. Kullmann, S. Wieringa, A. Biere, Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. *Proc. of Haifa Verification Conference*, *Lecture Notes in Computer Science*, vol. 7261, pp. 50–65, 2011.

[15] A. E. J. Hyvärinen, T. Junttila, I. Niemelä, Partitioning SAT instances for distributed solving. *Proc. of LPAR'10*, pp. 372–386, 2010.

[16] O. Kullmann, Investigating the Behaviour of a SAT Solver on Random Formulas. Technical Report CSR 23-2002, Swansea University, Computer Science Report Series, October, 2002.

[17] C.M. Li and Anbulagan, Look-Ahead versus Look-Back for Satisfiability Problems. *Lecture Notes in Computer Science*, vol. 1330, pp. 342–356, 1997.

[18] Y. S. Mahajan, Z. Fu, S. Malik, Zchaff2004: An Efficient SAT Solver. *Lecture Notes in Computer Science: Theory and Applications of Satisfiability Testing*, vol. 3542, pp. 360–375, 2005.

[19] J. P. Marques-Silva, K. A. Sakallah, GRASP: A New Search Algorithm for Satisfiability. *Proc. of ICCAD'96*, pp. 220–227, 1996.

[20] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver. *Proc. of DAC'01*, pp. 530–535, 2001.

[21] B. Nagy, The Languages of SAT and n-SAT over Finitely Many Variables are Regular. *Bulletin of the EATCS*, no. 82, pp. 286–297, 2004.

[22] B. Nagy, On the Notion of Parallelism in Artificial and Computational Intelligence. *Proc. of HUCI 2006*, pp. 533–541, 2006.

[23] B. Nagy, On Efficient Algorithms for SAT. *Proc. of CMC 2012*, Lecture Notes in Computer Science, vol. 7762, pp. 295–310, 2013.

[24] P. Kacsuk, J. Kovács, Z. Farkas, A. C. Marosi, G. Gombás, Z. Balaton, SZTAKI Desktop Grid (SZDG): A Flexible and Scalable Desktop Grid System. *Journal of Grid Computing*, vol. 7, no. 4, pp. 439–461, 2009.

[25] M. Posypkin, A. Semenov, O. Zaikin, Using BOINC Desktop Grid to Solve Large Scale SAT Problems. *Computer Science*, vol. 13, no. 1, pp. 25–34, 2012.