Dolius: A Distributed Parallel SAT Solving Framework

Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette *

Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{audemard,hoessen,jabbour,piette}@cril.fr

Abstract

Over the years, parallel SAT solving becomes more and more important. However, 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 tool called `Dolius`, mainly 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. Furthermore, our tool contains an API allowing to plug any SAT solver in a simple way.

1 Introduction

Cloud computing can change the landscape of computer science: it is now possible to request a virtually unlimited number of computing units that can be allocated within a few seconds. In the case of SAT solving, this fact means that larger formulas, much more difficult to solve could be considered, assuming that we dispose of a parallel SAT solver that scales well across different computing units.

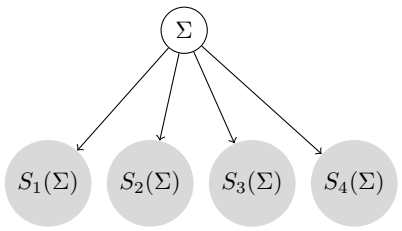
Unfortunately, such a scalable solver does not actually exist. Worst, a recent study about parallelization of SAT [18] shows that it appears to be very difficult to benefit from portfolio parallelization for modern CDCL solvers. Yet, since the emergence of multi-core CPUs, numerous parallel SAT solvers have been proposed by the community. From the simple script that runs in parallel the best known sequential solvers (e.g. `ppfolio` [22]) to complex engines that are able to share knowledge (`PeneLoPe` [2], `plingeling` [6]...) , most of the (empirically) best attempts are based on the portfolio schema which exhibits *per se* limitations in term of scalability.

The goal of this paper is twofold: first, propose a framework allowing to facilitate the creation of distributed divide and conquer (D&C) solvers. Those are alternatives to the parallel paradigm that behaves the best in practice for SAT solving: portfolio solvers. Indeed, portfolio techniques monopolize the prizes of each SAT competition, since the parallel track has appeared. By reducing the cost of creating a new distributed solver with our framework, D&C can possibly challenge this position. The second goal is to contribute to improve scalability in parallel solutions for SAT. The paper is organized as follows: in the next Section, we present the differences between the main schemes to parallelize SAT solvers and their implications. In Section 3 we present the main features of our framework called `Dolius`. Its API, presented in Section 4, enables any SAT solver to be easily plugged to it. Next, we evaluate in Section 5 the efficiency of our framework when instanced with one of the best current sequential solvers, and we finally conclude with some perspectives.

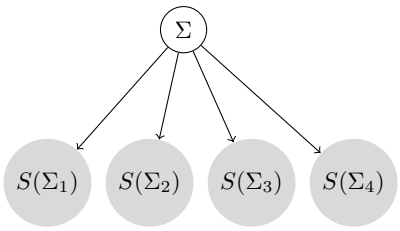
2 Mains schemes to parallelize SAT solving

Two main approaches are commonly explored to parallelize SAT solvers, namely portfolio and divide and conquer. Each general approach has its own pros and cons. In this section, we present those two

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



(a) Portfolio – Each S_i is the incarnation of a same solver S tuned with different parameters (restart cutoffs, nogood database cleaning strategy, etc.), and works with the original formula Σ. The formula is solved as soon as one of the S_i finishes its work.



(b) Divide & Conquer – All S represent the same solver that solves a (independent) subset of Σ. More precisely, $\Sigma \equiv \Sigma_1 \vee \Sigma_2 \vee \Sigma_3 \vee \Sigma_4$ and $\Sigma_i \wedge \Sigma_j \vDash \bot$ if $i \neq j$. The formula is solved when all S prove their problem inconsistent, or one of them finds a model.

Figure 1: Illustration of the main schemes for parallel SAT solving with 4 workers

frameworks for solving SAT in parallel, and compare them in a theoretical point of view.

Main differences between those two frameworks are illustrated in Figure 1, where 4 workers are considered.

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 [22, 6, 19]. With this approach, the crafting of the strategies is important, especially with a small number of workers. In general, the 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 [12, 2]. Using this technique, better results are obtained, but they cannot improve greatly their results by increasing the computing power.

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 [7, 9]. Those subspaces can be defined using the concept of *guiding path* [25].

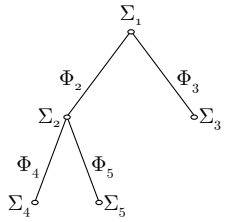


Figure 2: Guiding tree for the instance Σ

A guiding path is the formula added to the instance Σ in order to divide the search space. This guiding path is created by recursion. First, we define Σ_1 as the undivided instance we want to solve Σ, and therefore $\Sigma_1 = \Sigma$. For the ease of the explanation we want to define Σ_i as a conjunction of some formulae ϕ and Σ. Thus, we define Σ_1 as $\Sigma = \Sigma \wedge \Phi_1 \equiv \top$. Second, we divide Σ_1 in two sub-spaces to obtain Σ_2 and Σ_3 by using respectively the formula Φ_2 and Φ_3. Later, when a Σ_i needs to be divided into Σ_{i*2} and Σ_{i*2+1}, its respective division formula, or guiding path, will be $\Phi_{i*2} \wedge \Phi_i \wedge \Phi_{i/2} \wedge \ldots \wedge \Phi_1$ and $\Phi_{i*2+1} \wedge \Phi_i \wedge \Phi_{i/2} \wedge \ldots \wedge \Phi_1$. The conjunction of those guiding

path can be represented as a tree, called the guiding tree. An example of such depiction is shown at Figure 2, applied on the instance Σ. From the figure, we can deduce $\Sigma_2 = \Phi_2 \wedge \Sigma$, $\Sigma_3 = \Phi_3 \wedge \Sigma$, $\Sigma_4 = \Phi_4 \wedge \Phi_2 \wedge \Sigma$, $\Sigma_5 = \Phi_5 \wedge \Phi_2 \wedge \Sigma$.

Note also that the end of the search differs with the both approaches. In the portfolio case, the *first* worker that finishes to solve the formula (satisfiable or not) puts an end to the global search. In divide and conquer approach, the same occurs if the formula is satisfiable, but in the case of unsatisfiable formula, it is only proved inconsistent when the *last* slave delivers its answer.

2.1 Pathological Cases of D&C

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.*

2.1.1 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 $\Sigma \models a$. Hence, just using unit propagation, it is possible to show that $\Sigma \wedge (\neg a) \models \bot$. 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 [17].

2.1.2 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.

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.

2.2 Toward Scalability

Portfolio strategies are efficient on multicore architectures, but find their limits when they are used with a large number of computing resources. Indeed, adding more and more resources is not helpful for this kind of approach. This just leads to a large amount of redundant work, since in practice, the same parts of the search space are very often explored by several workers simultaneously. Moreover, a recent study shows that portfolio is a resolution schema that exhibits *per se* clear efficiency limitations [18].

On the contrary, adding more resources benefits better to the D&C framework, providing that useless divisions are not regularly achieved. In the next Section, we formally present our divide and conquer SAT solver, called `Dolius`.

3 Dolius

Since our goal is to contribute to improve scalability in parallel solutions and to deploy such solution on distributed architectures, it appears better suited to propose a novel approach based on the divide and conquer paradigm. This is the main objective of our framework `Dolius`. As presented in the next

section, our framework can easily plug any available SAT solver using a simple API and can be extended as a portfolio SAT solver. Let us start with the main architecture of `Dolius`.

`Dolius` uses one master and many slaves. This architecture was chosen for different reasons: first of all, it allows a much easier development. Such an architecture is used by webservers, which can handle ten thousands of concurrent connections. As depicted later, the work of the master in `Dolius` is very light, and only consists in putting in touch hungry slaves with active ones. Therefore, as our master's task is lighter than one of those webservers, such an architecture appears appropriate.

Even so, in contrast to full-decentralized techniques, this approach can clearly lead to bottlenecks. If this case would occurs, a tree structure could be implemented (a slave could be composed of a `Dolius` master that uses its own sub-slaves), as the one provided with the Domain Name Server (DNS) system.

Each slave is a SAT solver, whereas the master is a process that does not participate actively to the search, and is only used as the cornerstone for communication between the slaves. The master knows all its active slaves (the latter ones contact the former one in order to register) but slaves do not know each other. Moreover, the master is designed to be flexible, and workers can be added on the fly during the search. To divide the work, `Dolius` allows a divide and conquer approach through guiding-paths. Such guiding paths are not reduced to a single variable but can also split the formula with two sets of clauses. However, in that case, one needs to be careful on some properties that the sets of clauses must verify.

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. There exists two main schemas to achieve this load balance. With the first one, each busy worker regularly looks up in some defined "neighbourhood" of workers to (possibly) find an idle worker and *push* some work to it. The second schema makes responsible idle workers to contact an active worker in order to *steal* a part of its load.

Actually, in `Dolius`, when a slave becomes idle, it does not contact an active worker but the master. This last one has to choose an active slave to ask him to divide its load. Different criteria can be considered to choose this active slave. We propose the following work-stealing scheduling strategy: the master node stores a FIFO data structure of currently working nodes, proposes the first node to balance its load and puts this node at the end of the list. This system allows to ensure that work request is sent fairly between active workers. In addition, 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. A work request can be denied by the active worker if the underlying solver has not worked enough.

In order to achieve good performances and reduce the effect of an eventual bad division [3], solvers can choose to send to each other (through the master) some clauses learnt during the search. This induces that the solver can also find which part of its guiding path is responsible for the UNSAT answer and send it to the other worker. If there is no guilty part in the guiding path, an empty clause can be sent to stop all other workers. This is extremely useful as it can balance a bad division. To understand the opportunity of this mechanism, let us suppose that a worker W has the guiding path $\mathcal{G} = l_1 \wedge ... \wedge l_n$. It is possible that W generates the clause $\mathcal{C} = \neg l_1 \vee \neg l_2$ during its search. This clause can be very useful as it leads to the termination of every active node using $l_1 \wedge l_2$ as part of their guiding path. Furthermore, if a worker is able to generate an UNSAT proof without using its guiding path, an empty clause can be sent in order to stop every other worker.

When solving an UNSAT instance with a portfolio approach, the process stops after the first thread has stopped. In classical divide and conquer, the answer can be given only when every sub-problems have been found UNSAT. Therefore, sending the responsible part of the guiding path can bring us back to the portfolio situation, since any worker can possibly prove the original CNF unsatisfiable.

When a worker has to divide its work, it is also possible to send the learnt clauses or a selected subset. The choice of sending the clauses, and sending which clauses is somewhat important as some

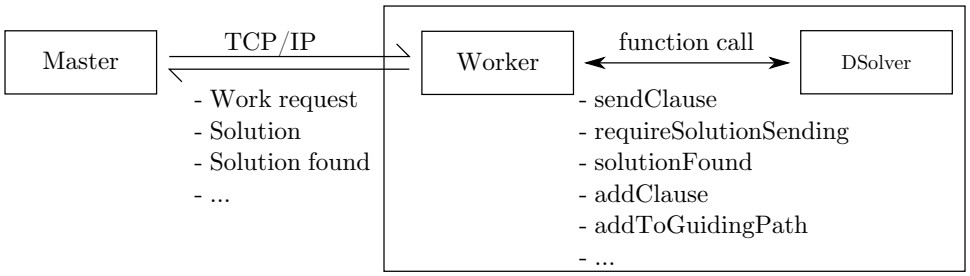


Figure 3: Communication between the master, the worker and the solver

	start	**search**	**end**
mandatory	`initialization` `set_GP`	`run(), stop()` `create_GP`	`sat_related_info`
optional	`addLearntClause()`	`addClause()` `iterators`	

Table 1: Summary of functionalities used by the `Dolius` API

clauses will not be used through the next search, but others might. Moreover, as communication is not done through shared memory, the cost is not negligible. The communication is made through TCP/IP, minimizing the need for external API. This choice was mainly made for portability and reliability of communications.

`Dolius` has a clear separation between the main platform (master, slaves, communication...) and the SAT solver as shown in Figure 3. This offers multiple advantages. The main one is that each worker does not need to use the same SAT solver (`minisat`[10], `PeneLoPe` [2], ...) allowing to easily introduce a portfolio approach inside the divide and conquer paradigm. This clear separation is possible with the provided API introduced in the next section. The reader can refer to [3] for more details on the architecture of `Dolius`.

4 Application Programming Interface

We detail in this section the proposed API to make any solver able to be used with `Dolius`. This API has actually been designed to be as complete as possible, but all functions are not needed, some of them are optional. First, we provide the minimal set of functionalities that have to be implemented to make use of `Dolius`.

To make a solver distributed with `Dolius`, only a few functions have to be fully implemented. Those functions are summarized in Table 1 that considers the life of a SAT search as 3 main phases: its start/initialization, the actual search period, and its end of the search.

The implementation of some of those functions should not be hard. For instance, the functions dedicated to retrieve information about the status of the search are easy to implement: `solutionFound()` is a simple state function that is used to know whether the search is still active, or if a solution has been already obtained. `isSolutionFoundSAT()` is used to know the nature of the delivered answer (SAT

```
//initialization
void setCNFFile(const char* inputFile);
void initialize(int nbVar, int nbClauses);
//thread related functions
void run();
void stop();
//clause database modification
void addLearntClause(const std::vector<int>& clause);
void addClause(const std::vector<int>& clause);
//iterators
void learntClauseIteratorRestart();
void learntClauseIteratorNext(std::vector<int>& clause);
void guidingPathIteratorRestart();
void guidingPathIteratorNext(std::vector<int>& clause);
int  getGuidingPathSize() const;
//guiding path modificators
bool createGuidingPath(std::vector<std::vector<int> >& gpA,
                       std::vector<std::vector<int> >& gpB);
void addToGuidingPath(const std::vector<int>& clauses);
//sat related information
bool solutionFound() const;
bool isSolutionFoundSAT() const;
int  getNbVar() const;
int  getSolutionLiteral(int var) const;
int  getNbLearntClauses() const;
```

Figure 4: functions to implement in order to incorporate a solver in `Dolius`

/ UNSAT), whereas the function `getSolutionLiteral()` is called to get the model, when a SAT answer has been obtained.

In the start phase, `initialize()` serves to allocate memory, and the input CNF file can be read through `setCNFFile()`, preparing the solver to be runned. In the "search" phase, the function `run()` is called to actually start the search process in its dedicated thread, whereas `stop()` is on the contrary used to interrupt this process.

Only 2 functions need more code to distribute a solver: `addToGuidingPath()` is used to restrain the search space of a worker, with respect to a guiding path given in parameter. As the guiding path is provided as a set of clauses, it allows developers to create new heuristics to divide the work. The other one, `createGuidingPath()`, represents the heart of load balancing, since it is called when an *idle* worker makes a work-steal to an active one. Thus, the active solver has to divide its own task to give a part of it to the active solver. In our current implementation, the division is made using a unit clause selected through a look-ahead procedure (see [3]), but other division strategies can be implemented in this `createGuidingPath()` function.

Iterators on learnt clauses are also used to send those on work division. By tweaking which clauses are iterated on, a heuristic can be implemented to choose how many clauses are sent when the work is divided. Those will be added by the *idle* worker through `addLearntClause()`.

In addition, the API proposes numerous optional functions (e.g. iterators on the guiding path, `getGuidingPathSize()` , etc.) that can be implemented for statistical and debug purposes. Hence,

our API has been designed to be both complete and easy-to-implement. As an example, the code needed to plug GLUCOSE [4] with `Dolius` consists of less than 300 lines of code.

Let us also note that a solver integrated in `Dolius` has also access to functionalities of our framework (log files, etc.). And in order to ease the integration of solvers, several tools have been developed such as a graphical depiction of the resulting guiding tree that can be animated and colouration of the log files.[1]

5 Evaluation

In the following, the hardware used is 2 Dell R910 with 4 Intel Xeon X7550 providing each 8 cores making 32 cores available per node. Each node has a gigabit ethernet controller and 256GB of RAM. The installed operating system is CentOS 6. For each experiment, given an instance and a number of workers, we choose a time limit of 20 minutes. Let us also note that we consider *wall clock* time, instead of CPU time, in this Section. We only made one run for a given instance and a given number of workers, because of limited resources. Indeed, one run, for a given number of workers, can make use of 17 hours in the worst case (20 minutes $\times$ 51 instances). As we consider 7 numbers of workers, with and without clause sharing, one entire run lasts more than one week. Running 10 times each instance would have been computationally very expensive.

In order to evaluate our platform, two elements are needed: instances and a solver that will be plugged in. Concerning instances, as the resources needed to make tests with many slaves may be quite important, a subset of the instances from the SAT Competition 2013 (application track) [5] are used. To be used, an instance needs to be solved by at least one of the five first parallel solver form the SAT Competition 2013 and may not be solved by everyone of them. Those criteria insure us to evaluate ourselves against *reachable* instances. From that set, we skim large instance families. We reduced that set to have a manageable number of instances: 51. In the final set, there are more unsatisfiable instances (33) than satisfiable ones (18).[2]

As solver, we have modified the solver GLUCOSE to be compatible. The work division strategy implemented in `createGP` is based on unit clauses where the chosen literal is one of the literal with the highest *VSIDS* value when the work is divided and must also provide a good balance value between the two branches. The balance value is obtained by using look-ahead techniques. As we are using unit clauses for the guiding path, we incorporated them in the `assumption` vector of the solver, as presented in [15]. This allows us to provide us more information whenever we find UNSAT. Indeed, through the `analyzeFinal` function that was designed in `Minisat`, we are able to find the responsible part of the guiding path, if any. Once found, this information can be sent to other solver (through the master) to avoid exploring redundant search spaces. Using `assumption` also allows us keep clauses whenever the guiding path is changed, allowing us to keep clauses that were generated. Two flavors were tested: in the first one the learnt clause iterator send the learnt clauses under the condition that their *literal block distance (LBD)* value needs to be lower than 4 and the maximum number of clauses sent must be lower than 10% of the total amount of learnt clauses. The results of this version of `Dolius` + GLUCOSE are shown in Figure 5 (a). As we can see, the cactus plot that represents 1 worker is higher than other curves showing that the work division helps in reducing the overall solving time. However, the curve for 16 workers is lower than the one using 32 workers, for a equivalent number of solved instances.[3] The main reason for this is the communication cost.

[1] The results of those tools are presented at `http://www.cril.fr/~hoessen/dolius.html`

[2] The exhaustive list can be found at `http://www.cril.fr/~hoessen/dolius.html`

[3] Using 32 workers, we were able to solve 2 instances more but due to the non determinism, solving 2 instances more is not really significant.

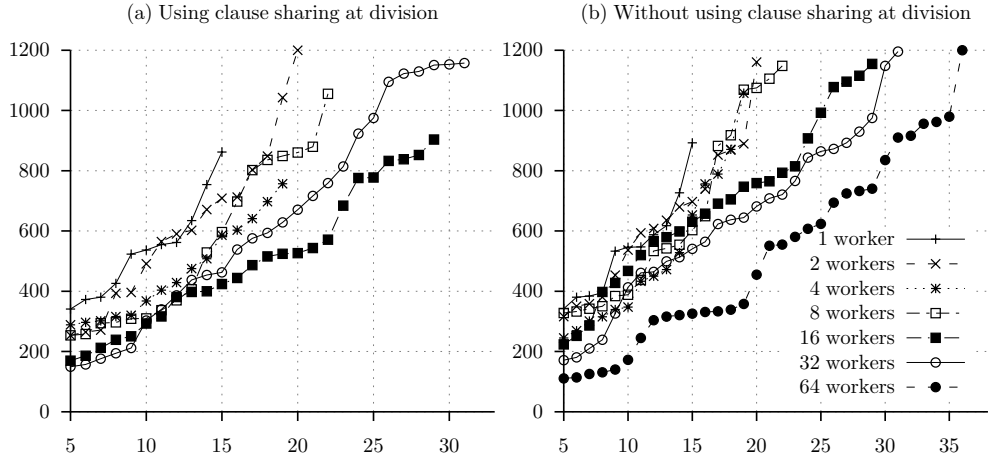


Figure 5: Scalability of the Dolius + GLUCOSE implementation

Solver	#threads	SAT	UNSAT	Total
PCASSO	32	9	21	30
plingeling	32	14	23	37
PeneLoPe	32	16	26	42
Dolius + GLUCOSE	32	9	23	32
Dolius + GLUCOSE	64	13	24	37

Table 2: Results for some of the parallel SAT solvers submitted to the SAT Competition 2013 and Dolius + GLUCOSE

The second flavor is obtained by simply deactivate the communication of learnt clauses at division through a different implementation of the learnt clause iterator. The results of this version are shown in Figure 5 (b). As we can see, in this version, the 32 workers curve is lower than the 16 workers curve. This motivated us to try this flavor using 64 workers. The gain obtained is quite significant, the curve being always lower than any other curve and providing answer for 5 more instances for a grand total of 37 solved instances.

Both flavors have their limitation, communication providing better results with a low number of resources and no communication with a higher number of resources. This should mean that in order to develop a good heuristic concerning the communication of learnt clause, the number of workers should be taken into account, in order to avoid the case where the learnt clause communication is completely disabled. This appears essential to keep an effective scalable communication when a large number of workers is involved. The experiments underlines that new heuristics that are able to take into consideration this increase of computing units have to be designed.

In order to understand those results and compare those with the current *state-of-the-art*, let us compare to 3 shared memory solvers among the best ones proposed in the SAT Competition 2013: plingeling [6], PCASSO [16] and PeneLoPe [1]. They were choosen for the following reasons: plingeling was the 2013 winner, PCASSO uses a division strategy and the authors wanted to compare against themselves with our previous solver, PeneLoPe. Each solver is launched using 32 threads as it is the highest number of cores available on a single machine. Results are shown in Table 2. First,

we have to recall that those solvers use shared memory for communication making their communication *de facto* faster than the one proposed in Dolius. Nevertheless, shared memory are a lot less scalable than distributed ones. Moreover, PeneLoPe and plingeling are portfolio: they use different configurations in each thread, allowing to increase the orthogonality of the search. And with a greater orthogonality comes a more robust solver against different families of benchmarks. With those details in mind, we can see that using twice the resources of plingeling –the solver who won the SAT Competition 2013–, we are able to obtain the same number of instances solved. Finally, PCASSO is partitioning the search space iteratively, an approach similar to the one described here. We can see that we achieve similar results with the same amount of threads/workers.

6 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.* [11] propose such an 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 [8] 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.

Schulz and Blochinger [23] 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 [23].

There also are very recent attempts to make use of the divide-and-conquer paradigm. Let us cite Treengeling that is a so-called *concurrent cube-and-concur* implementation [24, 13] using the CDCL solver Lingeling. Unfortunately, this implementation is not distributed. The solver by M. Soos CryptoMinisat, also benefits from a network parallelization. Hence, it can be distributed, but a portfolio scheme, where only unary and binary clause are exchanged, has been chosen to this extend.

Finally, a study on static partitioning of CNF formula has been proposed [14]. This partitioning viewing CNF formulas as trees has to be performed before the actual search. Such static partitioning are not the most efficient to obtain well-balanced (i.e. of similar solving difficulty) subformulas, but the authors argue that once the CNF is partitioned, its sub-formulas can be solved using one of many available solvers, without having to modify its source code, using it as a black box.

7 Conclusion

In this paper, we have presented a novel framework called Dolius for solving SAT in parallel which, in contrast to most of previously proposed attempts, is not based on portfolio, but on divide-on-conquer

paradigm. The resulting tool is distributed and scales well with respect to the number of computing units it makes use. Moreover, `Dolius` is not tied to a certain solver, but proposes an API to easily plug any available SAT solver.

As `Dolius` is very flexible, we plan in the next future to use it with plugging different SAT solvers, in order to hybridize the two main approaches to solve SAT in parallel.

References

[1] Gilles Audemard, Benoıt Hoessen, Saıd Jabbour, Jean-Marie Lagniez, and Cédric Piette. Penelope in SAT competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 66, 2013.

[2] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel sat solving. In *Fifteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, pages 200–213, may 2012.

[3] Gilles Audemard, Benoît Hoessen, Said Jabbour, and Cédric Piette. An effective distributed D&C approach for the satisfiability problem. In *22nd Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP'14)*, february 2014.

[4] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *proceedings of IJCAI*, pages 399–404, 2009.

[5] Adrian Balint, Marijn JH Heule, Anton Belov, and Matti Järvisalo. The application and the hard combinatorial benchmarks in sat competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 99, 2013.

[6] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 51, 2013. `http://fmv.jku.at/lingeling`.

[7] Wahid Chrabakh and Rich Wolski. GrADSAT: A parallel SAT solver for the grid. Technical report, UCSB, 2003.

[8] Wahid Chrabakh and Richard Wolski. The gridsat portal: a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure. *Concurrency and Computation: Practice and Experience*, 19(6):795–808, 2007.

[9] Geoffrey Chu, Peter J. Stuckey, and AAron Harwood. Pminisat: a parallelization of minisat 2.0. Technical report, SAT Race, 2008.

[10] Niklas Een and Niklas Sörensson. An extensible SAT-solver. In *proceedings of SAT*, pages 502–518, 2003.

[11] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.

[12] Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Manysat: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.

[13] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference (HVC'11)*, pages 50–65, 2011.

[14] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 372–386, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *proceedings of CP*, pages 385–399, 2011.

[16] Ahmed Irfan, Davide Lanti, and Norbert Manthey. Pcasso–a parallel cooperative sat solver. *Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions*, page 64, 2013.

[17] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9:174–189, 2001.

[18] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of sat solvers. *AAAI'13*, 2013.

[19] Stephan Kottler and Michael Kaufmann. SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[20] Mikhail Posypkin, Alexander Semenov, and Oleg Zaikin. Sathome web page. `http://sat.isa.ru/pdsat`, 2008 (a verifier).

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

[22] Olivier Roussel. ppfolio. `http://www.cril.univ-artois.fr/~roussel/ppfolio`.

[23] Sven Schulz and Wolfgang Blochinger. Parallel sat solving on peer-to-peer desktop grids. *Journal of Grid Computing*, 8(3):443–471, 2010.

[24] P. van der Tak, Marijn Heule, , and A. Biere. Concurrent cube-and-concur. In *Proceedings of the 3rd International Workshop on Pragmatics of SAT (POS'12)*, 2012.

[25] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.