Chapter 1
Parallel Satisfiability

Tomáš Balyo and Carsten Sinz

Logic is the beginning of
wisdom, not the end –
Leonard Nimoy

Abstract The propositional satisfiability problem (SAT) is one of the fundamental problems in theoretical computer science, but it also has many practical applications. Parallel algorithms for the SAT problem have been proposed and implemented since the 1990s. This chapter provides an overview of current approaches and their evolution over recent decades towards efficiently solving hard combinatorial problems on multi-core computers and clusters.

1.1 Introduction

SAT is one the most important problems in computer science. It was the first problem proven to be NP-hard [16]. Despite its complexity there are very efficient SAT solvers which make it possible to design successful algorithms for hard problems by translating them to SAT.

Parallelizing algorithms for combinatorial decision problems, such as SAT, is not an easy task, as the search space is highly irregular and different search heuristics can have a tremendous effect on the observed run-time. Theoretical results vary from super-linear speedups for random problems [55] on the positive side to profound proof-theoretic limitations [34] on the negative.

Nevertheless, important parallelization techniques for SAT have been developed, including divide-and-conquer approaches, portfolio solvers, and parallel local search solvers.

As the increase in compute power of a single processor core has been stagnating over recent years, it has become even more important to invent and engineer parallel algorithms that can make optimal use of current and future computer architectures.

Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

© Springer International Publishing AG, part of Springer Nature 2018

Y. Hamadi und L. Sais (eds.), *Handbook of Parallel Constraint Reasoning*,
https://doi.org/10.1007/978-3-319-63516-3_1

This chapter is organized as follows: after an introduction to basic notions and algorithms for the SAT problem, parallel computing architectures and the problem of measuring speedups are discussed. Then the current main lines for parallel SAT algorithms are presented, namely divide-and-conquer (also known as search space partitioning), portfolios (diversify-and-conquer), and local search solvers. The chapter closes with a look at future challenges.

1.2 Preliminaries

In this section we give the basic definitions and properties of the satisfiability problem, which can also be found in any SAT-related textbook (for example the Handbook of Satisfiability [7]).

1.2.1 Satisfiability (SAT)

We start with the definition of a formula, which is the input of the SAT problem.

Definition 1 (CNF Formula). A *Boolean variable* is a variable with two possible values, *True* and *False*. A *literal* of a Boolean variable x is either x or $\neg x$, i.e., *positive* or *negative literal* of x. A *clause* is a disjunction (OR) of literals. A *conjunctive normal form (*CNF*) formula* is a conjunction (AND) of clauses. We can also regard a clause as a set of literals and a CNF formula as a set of clauses, since the ordering is not important in either case.

In the remainder of the chapter we will just use the term formula instead of CNF formula. Next we define what is a satisfying assignment.

Definition 2 (Satisfying Assignment). A *truth assignment* ϕ of a formula F assigns a *truth value* to its variables. The assignment ϕ satisfies

- a positive literal if it assigns the value True to its variable,
- a negative literal if it assigns the value False to its variable,
- a clause if it satisfies at least one of its literals,
- a CNF formula if it satisfies each one of its clauses.

If ϕ satisfies a formula F, then ϕ is called a *satisfying assignment* for F.

A clause with no literals is called an *empty clause*. Such a clause cannot be satisfied by any truth assignment. The definition of satisfiability follows.

Definition 3 (Satisfiability). A formula F is said to be *satisfiable* if there is a truth assignment ϕ that satisfies F, i.e., ϕ is a satisfying assignment of F. Otherwise, the formula ϕ is *unsatisfiable*.

The *problem of satisfiability (SAT)* is to determine whether a given formula F is satisfiable or unsatisfiable.

A *SAT solver* is a procedure that solves the SAT problem. For satisfiable formulas we also expect a SAT solver to produce a satisfying assignment. An example of a satisfiable formula with its satisfying assignment follows.

Example 1. $F = (x_1 \vee x_2 \vee \neg x_4) \wedge (x_3 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2)$ is a CNF formula with three clauses: $\{(x_1 \vee x_2 \vee \neg x_4), (x_3 \vee \neg x_1), (\neg x_1 \vee \neg x_2)\}$ and six literals $\{x_1, \neg x_1, x_2, \neg x_2, x_3, \neg x_4\}$ on four variables $\{x_1, x_2, x_3, x_4\}$. F is satisfiable with $\phi = \{x_1 \rightarrow False, x_2 \rightarrow True, x_3 \rightarrow True, x_4 \rightarrow True\}$ being a satisfying truth assignment of F.

1.2.2 Local Search Algorithms for SAT

The simplest approach to SAT solving is local search. A generic local search algorithm starts with a truth assignment (usually random, i.e., each variable has a random truth value assigned) and then iteratively selects a variable whose value is flipped (changed to False if it was True and vice versa) until a satisfying assignment is reached. The pseudo-code of this generic local search is presented as Algorithm 1.1.

Obviously, this algorithm only works for satisfiable formulas; for unsatisfiable instances it does not terminate. The performance of the algorithm depends on the initial truth assignment and the way of selecting variables for flipping. In the best-case scenario the initial assignment is already satisfying and we are finished. Also, if the variable selection were ideal, we could reach a satisfying assignment from any initial assignment in at most n steps (where n is the number of variables). In practice we need to use heuristics for both these steps and the main loop (the variable flipping) is executed only a limited number of times after which the algorithm gives up.

The initial truth assignment is often chosen randomly and for the variable selection a heuristic minimizing the number of unsatisfied clauses is used. Two historically important examples of local search algorithms are GSAT [49] and WalkSat [48]. GSAT select a variable that reduces the number of unsatisfied clauses most when flipped. WalkSat first randomly selects a clause that is not satisfied under the current assignment and flips one of its literals based on the number of clauses that become satisfied and unsatisfied after the flip. WalkSat additionally performs a random selection of the literal to flip in a certain percentage of the flips to emulate random walk, hence the name WalkSat.

Local search algorithms are usually the best choice for randomly generated satisfiable formulas and some combinatorial problems encoded to SAT.

1.2.3 The DPLL Algorithm

Most of the current state-of-the-art SAT solvers are based on the CDCL (conflict-driven clause learning) algorithm [41], which is in turn based on the Davis Putnam

Algorithm 1.1: A Generic Local Search Algorithm

```
1 Function LS(Formula f)
2     φ ← generate truth assignment
3     while F not satisfied by φ do
4         v ← pick a variable
5         φ[v] ← ¬φ[v]
6     return true
```

Logemann Loveland (DPLL) algorithm [17]. Before we give a description of CDCL in the following section we review DPLL here. The DPLL algorithm is basically a depth-first search of partial truth assignments (truth assignments where some variables remain unassigned) with three additional enhancements. The explanation of these enhancements follows.

- *Early termination.* If all literals are False in some clause, we can backtrack since it is obvious that the current partial truth assignment cannot be extended into a satisfying assignment. If all clauses are satisfied we can stop the search. The remaining unassigned Boolean variables can be assigned arbitrarily.
- *Pure literal elimination.* Given a partial truth assignment ϕ a pure literal is a literal the negation of which does not appear in any of the clauses not satisfied by ϕ. The variable corresponding to a pure literal can be assigned to make each clause where it appears true. This might lead to the appearance of new pure literals.
- *Unit propagation.* A clause is called unit if all but one of its literals are false under ϕ and the remaining literal is unassigned. The unassigned literal of a unit clause must be assigned to be true. This can make other clauses unit and thus force new assignments. The cascade of such assignments is called unit propagation.

In the DPLL procedure the enhancements are used after each decision assignment of the depth-first search. First we check the termination condition. If the formula is neither satisfied nor unsatisfied by the current partial assignment, we continue by unit propagation. Finally we apply the pure literal elimination. Unit propagation is called before pure literal elimination because it can cause the appearance of new pure literals. The other way around, pure literal elimination will never produce a new unit clause, since it does not make any literals false. Pseudo-code of DPLL is presented as Algorithm 1.2.

We can see that DPLL is a sound and complete algorithm (always terminates and answers correctly) from the fact that DPLL is a systematic depth-first search of partial truth assignments. The enhancements only filter out some branches that represent non-satisfying assignments.

The time complexity of this procedure is exponential in the number of variables. That corresponds to the number of vertices of a binary search tree with depth n, where n is the number of variables. However, in practice, thanks to unit propagation

Algorithm 1.2: The DPLL Algorithm

```
1 Function DPLL(Formula F, Assignment φ)
2     doUnitPropagation(F,φ)
3     if all literals false in some clause then
4         return false
5     doPureLiteralElimination(F,φ)
6     if all clauses satisfied then
7         return true
8     x ← choose an unassigned variable
9     return DPLL(F, φ[x] = True) or DPLL(F, φ[x] = False)
```

and early termination, the DPLL procedure never goes as deep as n in the search tree. The maximal depth reached during search is often a fraction of n. This makes DPLL run much faster on instances with n variables than one would expect from the formula 2^n.

1.2.4 Resolution Refutation

Resolution is a rule of inference which produces a new clause from clauses containing complementary literals. Two clauses C and D are said to contain complementary literals if there is a Boolean variable x such that $x \in C$ and $\neg x \in D$. The produced clause (containing all the literals from C and D except for x and $\neg x$) is called the *resolvent* of C and D (notation: $R(C,D)$).

A formula F containing C and D is satisfiable if and only if $F \wedge R(C,D)$ is satisfiable. This implies that if the empty clause can be resolved from the clauses of a formula then this formula is unsatisfiable (since the empty clause cannot be satisfied). The *Resolution Refutation* algorithm keeps adding resolvents to its input formula until either the empty clause is added (which means the input formula is unsatisfiable) or no more new resolvents can be added (in which case the input formula is satisfiable). Note that resolvents added in one step can be used as input clauses for resolutions in later steps.

Although the resolution refutation algorithm is sound and complete it is not very efficient in practice since it has exponential memory complexity (in general there are exponentially many possible resolvents for a formula).

1.2.5 The CDCL Algorithm

The conflict-driven clause learning (CDCL) algorithm is the state-of-the-art algorithm for solving SAT problems. It was first implemented in the SAT solver Grasp [41].

Algorithm 1.3: The CDCL Algorithm

```
1  Function CDCL(Formula f)
2      decLev ← 0
3      φ ← ∅
4      if doUnitPropagation(f,φ) = CONFLICT then
5          return false
6      while not all variables assigned do
7          decVar ← pick decision variable
8          decVal ← pick a truth value
9          decLev ← decLev + 1
10         φ[decVar] = decVal with decision level decLev
11         if doUnitPropagation(f,φ) = CONFLICT then
12             (learnedClause, backLev) ← analyze conflict
13             if backLev ≥ 0 then
14                 decLev ← backLev
15                 f ← f ∧ learnClause
16                 φ ← unassign variables with decision level ≥ backLev
17             else
18                 return false
19     return true
```

In this subsection we describe only the basic concepts behind CDCL. For a more detailed comprehensive description please refer to [7].

The CDCL algorithm combines ideas of DPLL search and resolution refutation. The pseudo-code of CDCL is presented in Algorithm 1.3. Similarly to DPLL the algorithm performs depth-first search of partial truth assignments and uses improvements such as unit propagation and early termination. Additionally, CDCL performs a procedure called *conflict analysis* each time a conflict state is reached, i.e., every literal becomes false in some clause under the current partial assignment.

The conflict analysis determines which decisions and which clauses (via unit propagation) are responsible for the conflict. The clauses responsible for the conflict are called *reason clauses*. By resolving the reason clauses of a conflict we get new clauses that can be added to the formula. Clauses added this way are called *learned clauses*.

In the CDCL algorithm each truth value assignment to a variable has an attribute called its *decision level*. The assignments implied by the initial unit propagation have decision level zero, the assignments coming from the first branching decision and the unit propagation that follows it have decision level one, and so on. In DPLL the decision level represents the depth of the recursive call during which the variable was assigned. The decision level increases by one after every branching decision and is decreased by one after a conflict is encountered and we backtrack to the previous decision.

In CDCL the decision level can decrease by more than one during backtracking. This is called *non-chronological backtracking* or *backjumping*. The decision level to which the algorithm “backjumps” is calculated during conflict analysis.

1.2.6 Parallel Computing Architectures

In this subsection we review the basic notions related to parallel computing, such as parallel architectures, memory models, and definitions of speedup and parallel efficiency.

Based on the access to the main memory used in a parallel system we can distinguish two kinds of parallel architectures.

- *Shared Memory Architectures*. The main memory is shared between all processing elements in a single address space. It is used on single computers with multiple (multi-core) processors. The advantages of this approach are that all processes have very fast access to the shared data and less total memory is used, which allows the solution of larger problems. The disadvantage is that race conditions must be addressed (usually with locks), which may lead to parallel slowdown or even deadlocks, and this makes implementations error prone.
- *Distributed Memory Architectures*. Each processing element has its own address space and communication is usually done by message passing. This approach can be used on single computers or on grids/clusters of computers. The speed of communication is lower than in the case of shared-memory architectures but the design and implementation of such a system is usually simpler.

A parallel system can also use a combination of these architectures. For example the parallel solver HordeSat[6], which was designed to run on clusters of multi-core computers, uses shared-memory communication inside the nodes and distributed-memory communication between the nodes.

1.2.7 Measuring Speedups

The speedup of a parallel solver P compared to a sequential solver S for a given benchmark is the ratio of run times that the solvers need to solve that benchmark, i.e., $s = t_P/t_S$, where t_P and t_S are the runtimes of the parallel and sequential solver respectively.

In parallel processing, one usually wants good scalability in the sense that the speedup over the best sequential algorithm goes up near linearly with the number of processors. Measuring scalability in a reliable and meaningful way is difficult for SAT solving since running times are highly nondeterministic. Hence, we need careful experiments on a large benchmark set chosen in an unbiased way.

By averaging the speedups for each benchmark instance we can compute the *average speedup*. The average speedup is not a very robust measure since it is highly dependent on a few very large speedups that might be just due to luck. For this reason we often get very large average speedup values that are not representative for the entire benchmark set. Calculating the median of the speedups gives us the *median speedup*. The value of the median speedup is often very small if the benchmark set contains a large number of easy benchmarks where parallelization does not bring any benefit, and therefore it is not an ideal measure either. A better measure is the *total speedup* which is the sum of runtimes for the parallel solver divided by the sum of runtimes for the sequential solver on the benchmark set.

Nevertheless, all these measures can treat a massively parallel solver (a solver designed for hundreds or thousands of processors) unfairly when most instances are actually too easy to justify investing in a lot of hardware. Indeed, in parallel computing, it is usual to analyze the performance on many processors using *weak scaling* where one increases the amount of work involved in the considered instances proportionally to the number of processors. Therefore the set of benchmarks considered for calculating the average, median, and total speedups is usually restricted to those instances where the sequential solver needs at least $c \times p$ seconds where p is the number of processors used by the parallel solver and c is constant.

1.3 Divide-and-Conquer Approaches

Historically, the first parallelization approaches for the SAT problem were based on splitting the search space. Here, different tasks search for a satisfying assignment in disjunct portions of the search space. Different ways to split the search space have been proposed [10, 11, 12, 14, 15, 20, 30, 32, 33, 38, 46, 50, 58, 59]. Splitting the search space should preferably yield portions of potentially equal size to balance the search evenly among different tasks. Predicting the size of the search space for a DPLL or CDCL search is extremely hard and no satisfactory solutions exist up to now, even though some promising attempts have been made [36, 37].

Thus, the search space is typically not split up statically (at the start of the algorithm), but dynamically, as soon as one processor involved in the search becomes idle.

1.3.1 Problem Decomposition and Load Balancing

Problem decomposition plays a central role within the design process of parallel algorithms, since it influences all other design phases. In this stage, the whole problem is divided into appropriate subproblems (called *tasks*) which can be executed in parallel by the available processors. Problem decomposition must achieve two (typically conflicting) goals:

- Minimize idle times of available processors.
- Minimize overhead due to communication and excess computation.

Basically, problem decomposition can be carried out statically (i.e. tasks are defined at compile time) or in a dynamic manner, where tasks are generated (on demand) at run-time. In the latter case, tasks are explicit objects within the parallel program which can be dynamically assigned to processors for execution.

Due to the sophisticated heuristics employed by contemporary DPLL-based SAT solvers it is virtually impossible to predict the time needed to solve a specific SAT instance. Accordingly, the run-time of an individual task cannot be predicted and the run-times of different tasks may vary considerably. For SAT instances exhibiting such a highly irregular problem structure a static approach to decomposition can result in significant processor idling. Thus, for realizing a robust parallel SAT-solving method, dynamic problem decomposition becomes mandatory.

For problems based on heuristic search, typically an *exploratory* approach to problem decomposition is employed where tasks represent untried branches of the search tree. Specifically, this technique enables a running solving task to efficiently split off a part of its own search space, generating a new task. The parallel computation terminates when all generated tasks have been completed or when a task reports a solution. In the latter case, the remaining tasks can be canceled.

Technically, exploratory decomposition can be accomplished by a transformation of the assignment stack of a running solving process. It narrows the search space of the solving process by fixing the decision and the corresponding implications of the first level. The released search space is defined by the top-level assignments and the flipped decision. In this way, tasks can be represented by a set of assignments. The splitting procedure is depicted in Figure 1.1. This technique was first described by Chrabakh and Wolski [15]. It represents a refinement of the guiding-path approach developed by Zhang et al. [59].

In order to enable dynamic problem decomposition, a sequential solver must be adapted to support the discussed transformation of the assignment stack and it must also be capable of initializing the level 0 of the assignment stack according to the set of assignments delivered by a task.

The parallel computation starts with a single task that is responsible for the whole search space (i.e. the task is defined by an empty set of assignments). When idle processors are detected (either initially or upon completion of a task), search space splitting is performed to induce additional parallelism. This procedure is steered by the load-balancing process, which we discuss next.

Generally, dynamic problem decomposition requires explicit load balancing, i.e., tasks have to be assigned to processors at run-time. Especially for problems with high irregularity, the *task pool* model should be employed. It decouples problem decomposition and load balancing by using an explicit data structure holding tasks resulting from dynamic decomposition operations.

The task pool model can be organized in either a centralized or a distributed fashion. In a centralized approach, a master processor maintains a global task pool from which processors can request new tasks when they become idle. The master also

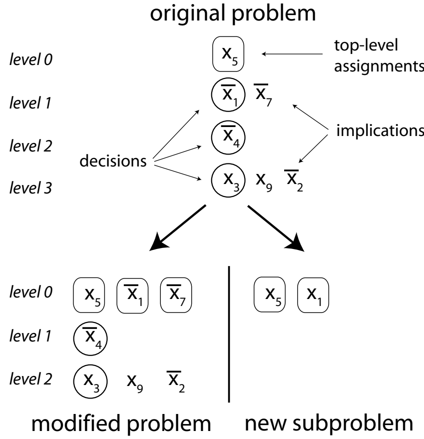

Fig. 1.1: Problem Decomposition

keeps track of the activity of each processor. Thus it can select an active processor to perform problem decomposition when the size of the pool falls below a given threshold. This ensures that the task pool is sufficiently filled to serve task requests in a timely fashion.

Fully distributed load balancing requires that every processor maintains its own task pool. In this setting, problem decomposition and load balancing must be accomplished autonomously by the processors. If a processor runs out of tasks it chooses another processor (e.g., by a round-robin or a randomized scheme) from which to request new tasks. When a predefined amount of time has elapsed without a reply, a request to a different processor is issued. On the other hand, active processors perform problem decomposition when the size of the task pool falls below a certain threshold. In order to prevent parallelism being generated in an uncontrolled way, the number of splitting operations a processor performs must be limited, e.g., by a minimum time interval between two consecutive split operations. In the distributed task model, choosing appropriate threshold and timing values is a subtle task, which can in practice only be managed by extensive experimentation. Due to the lack of a central controller, detecting the end of a parallel computation (i.e., all generated tasks have been executed) requires explicit protocols, e.g., Dijkstra's token-ring-based termination detection algorithm [18].

In general, a centralized approach can establish a more accurate view of the state of the processors and is more easy to implement, particularly on shared-memory architectures. However, with an increasing number of processors, centralized components soon become a sequential bottleneck of a parallel computation, which can seriously limit the overall efficiency. Thus, at least for distributed-memory architectures with a large number of processors a distributed design should be preferred.

The decomposition procedure we have discussed in this section represents the approach taken by most of the existing parallel SAT solvers. However, in the light

of the latest generation of sequential SAT-solving methods, exploratory decomposition can become a source of work anomalies. The search spaces of the generated subproblems are mutually disjoint, but their union isn't necessarily identical to the search space covered by the sequential algorithms (e.g., due to the failure-driven assertion technique). Consequently, the total amount of work carried out may differ significantly between the sequential and the parallel algorithm. On the one hand, this can result in poor speedups (due to excess computation) and on the other hand super-linear speedups are possible.

1.3.2 Implementations of Search-Space-Splitting Solvers

Table 1.1 shows implementations of search-space-splitting parallel DPLL SAT solvers. (Abd El Klalek *et al.* [19] also provide an overview and classification of many parallel SAT solvers.) For each solver the target infrastructure is indicated as well as whether the implementation provides fault detection and clause exchange.

Solver / Author(s)	Year	Ref.	Infra-structure	Fault Det.?	Clause Exch.?	Comment
Böhm & Speckenmeyer	1994	[14]	Transputer	no	no	First parallel SAT implementation
PSATO	1994	[58]	Cluster	yes	no	Introduced notion of "Guiding Path"
PSolver	1998	[38]	Grid	yes	no	Master / slave approach allowing integration of different sequential solvers
PaSAT	2001	[50]	SMP	no	yes	First solver with clause exchange
//Satz	2001	[32]	Cluster	no	no	Defined the notion of "ping-ping phenomenon"
GridSAT / GradSAT	2003	[15]	Grid	no	yes	Refined search space splitting
ySAT	2005	[20]	SMP	no	yes	Focus on cache performance
ZetaSAT	2005	[12]	Grid	yes	no	Runs on heterogeneous grids
NorduGrid	2006	[30]	Grid	yes	no	Splitting based on the "scattering rule"
PaMiraXT	2009	[47]	SMP & Cluster	no	yes	Master/client model

Table 1.1: Some early DPLL-based search-space-splitting SAT Solvers

Problem decomposition via dynamic search space splitting results in highly irregular run-times. This is shown in Figure 1.2. For a thousand runs on each instance, the run-time distribution is depicted; the two instances on top are satisfiable, the ones on the bottom unsatisfiable.

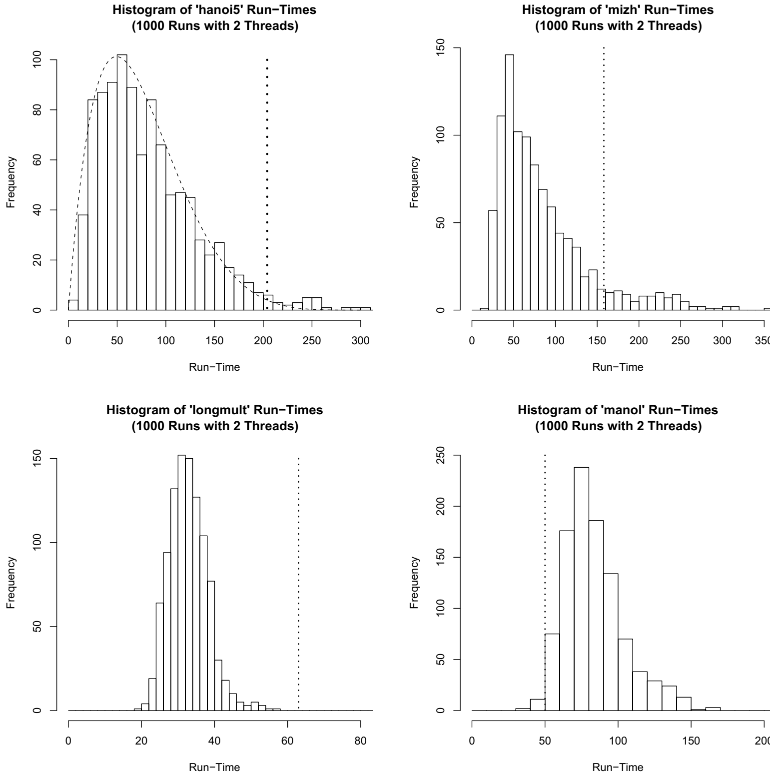

Fig. 1.2: Run-time distributions for 1,000 runs of the parallel solver PaSAT [50] on four selected instances (towers of Hanoi; cryptanalysis; hardware multiplier; pipelined microprocessor). The two instances on the top (Hanoi and mizh) are satisfiable, the ones on the bottom are unsatisfiable. On the x-axis, the run-time of PaSAT on two cores is shown. The y-axis indicates the number of times the run-times were in the given interval. The vertical line depicts the sequential run-time of the solver (one core). In the top left figure, the dashed curve indicates a beta distribution with $\alpha = 2.0$ and $\beta = 6.5$

1.3.3 Search Space Splitting in CDCL

As CDCL traverses the search space in a less structured way than DPLL, approaches based on the guiding path and dynamic decomposition cannot be directly applied for search-space-splitting CDCL solvers.

Adaptations have been implemented in PCASSO [40], Treengeling [9], and Ampharos [3]. The employed techniques are similar to the Cube and Conquer approach described below.

1.3.4 Cube and Conquer

The basic idea of the Cube and Conquer [28] approach is to use look-ahead techniques to split the problem into a large number (thousands) of subproblems, which can then be solved in parallel. The Cube and Conquer approach is discussed in detail in Chapter 2.

1.4 Parallel Portfolios – Diversify and Conquer!

In this section we discuss the simplest and yet (currently) most powerful approach to parallelizing SAT – parallel portfolios. We start by explaining the concept of the virtual best solver that served as the inspiration for the portfolio approach. Then we discuss clause sharing, which is an important component of any portfolio SAT solver. We conclude the section by reviewing some of the existing portfolio SAT solvers.

1.4.1 Virtual Best Solver

In a SAT competition a collection of SAT solvers submitted by researchers from all over the world is run on a pre-selected set of benchmark problems with some time limit (usually 1 hour or 5,000 seconds per instance). The results of a solver are defined as the set of run times for each problem solved by that solver. The solver solving the highest number of problems (within the time limit) is the winner of the competition; ties are broken by comparing the average run times.

When a SAT competition is organized and the results are published it is common to include the results for the *virtual best solver* (VBS) along with the results of the actual solvers participating in the competition. The results of the VBS are calculated as follows. For each benchmark that was solved by at least one of the participating solvers we take the best run time from the run times of the solvers on that benchmark. This implies that no solver has better run time than the VBS on any of the benchmarks or solves a benchmark not solved by the VBS.

Is it possible to have a real solver that is as good as the VBS? Such a solver would need to have the ability to instantly select the best SAT solver for any benchmark. This seems to be rather difficult; however, if we have a parallel architecture and only care about wall-time, there is a simple solution. We run all the available solvers in parallel on the given problem and as soon as one of the solvers finds a solution we terminate all the remaining solvers. This parallel solver would clearly achieve the same results as the VBS. A solver like this is called a *parallel portfolio solver*.

1.4.2 Pure Portfolio Solvers and Diversification

In the 2011 SAT Competition the PPfolio [45] solver demonstrated that it is possible to win several tracks of the competition by just taking the best solvers from the previous competition and trivially combining them using a shell script into a portfolio. The author of PPfolio argues that such a simple portfolio solver can serve as an approximation of the virtual best solver. But he also "shamelessly claims" [31] that “it’s probably the laziest and most stupid solver ever written” which “does not even parse the CNF” and “knows nothing about the clauses”. This most basic kind of portfolio solver is called a *pure portfolio* and the results obtained by this portfolio are only due to the base solvers selected.

A pure portfolio solver winning the competition can be very demotivating for the developers of the included solvers since someone else is winning with their solver.[1] To avoid this situation the following SAT competitions restricted or completely prohibited the participation of such portfolios.

A portfolio can be also created by using just one SAT solver, which is run several times in parallel with different configuration settings. The motivation behind this approach is that the performance of SAT solvers is heavily influenced by a high number of different settings and parameters of the search such as the heuristic used to select a decision literal in DPLL/CDCL, different restart policies or clause learning and deleting schemes in CDCL. Numerous parameter configurations are possible but none of them dominates all the other configurations on each problem instance.

The process of selecting good configurations for a portfolio solver is called *diversification*. Similarly to stock market portfolios a SAT solver portfolio should be diversified to achieve variety and increase the robustness of the solver. In a well diversified parallel portfolio solver each core solver explores a different region of the search space and therefore the overlap, i.e., redundant work, is minimized. The usual parameters that are diversified are related to decision heuristics (for example community branching [53] and block branching [52]), restart heuristics [51], and clause deletion strategies [22]. These configurations are often selected by hand but methods for automatic configuration of SAT solvers for portfolios are also studied [57].

1.4.3 Clause-Sharing Portfolios

If a portfolio is based on CDCL (conflict-driven clause learning) solvers then learned-clause exchange can be implemented. This grants a considerable boost to the solvers performance. Together with diversification it is an important mechanism to reduce duplicate work, i.e., parallel searches working on the same part of the search space.

---

[1] It should be noted that non-portfolio solvers are often derived from existing solvers too. However, they typically make reference to the original solver, e.g., by having a name derived from the original solver’s name. Moreover, some competitions include a “Hack Track”, in which small modifications to an existing solver can be submitted.

A clause learned from a conflict by one CDCL instance distributed to all the other CDCL instances will prevent them from doing the same work again in the future.

The problems related to clause sharing are to decide how many and which clauses should be exchanged. Exchanging all the learned clauses is unfeasible especially in the case of large-scale parallelism due to communication overhead. Also having too many clauses slows down a CDCL solver. A simple solution is to distribute all the clauses that satisfy some conditions. The conditions are usually related to the length of the clauses (number of literals in them) and/or their glue value [4] (the number of different decision levels associated with the literals of the clauses). A technique to dynamically adjust the size of shared clauses has been proposed in [25].

An interesting technique called "lazy clause exchange" was introduced in a recent paper [5] and used in the parallel version of the SAT solver Glucose [4]. In this policy a solver does not share a clause immediately after it is learned, but only after it proves its worth by being useful locally. Being useful locally means that the clause appears in conflicts as a reason clause at least a given number of times. This restriction does not apply to short clauses (at most two literals) and clauses with a low glue value. The policy also contains a strategy for importing clauses from other solver instances. The incoming clauses are put in "probation" before a potential entry into the clause database. This limits the negative impact of importing too many clauses. The probation phase is implemented by watching only one literal in these clauses, which means that they are not used for unit propagation and are only detected when they become unsatisfied. At that point they leave probation and are promoted to the regular learned-clause status. The experimental data in [5] show that only 10% of the imported clauses leave probation on average, which demonstrates how well-founded this strategy is.

Similarly to sequential SAT solvers, clause-sharing portfolio solvers can produce proofs of unsatisfiability and therefore be validated [27].

Clause sharing can be implemented in a lockless fashion as demonstrated by the SAT solver SArTagnan [25, 35].

Clause sharing in a parallel environment introduces non-determinism to the solver, which might not be desirable for practitioners who expect run time reproducibility. This issue has been addressed in [23] where a fully deterministic parallel portfolio solver was designed.

1.4.4 Impact of Diversification and Clause Sharing

Diversification and clause sharing are both essential components of a successful CDCL portfolio SAT solver. But to better understand them let us take a look at the impact of these techniques in isolation for satisfiable and unsatisfiable random 3-SAT instances.

By looking at the cactus plots in Figure 1.3 we can observe that clause sharing is essential for unsatisfiable instances while not very beneficial and even slightly

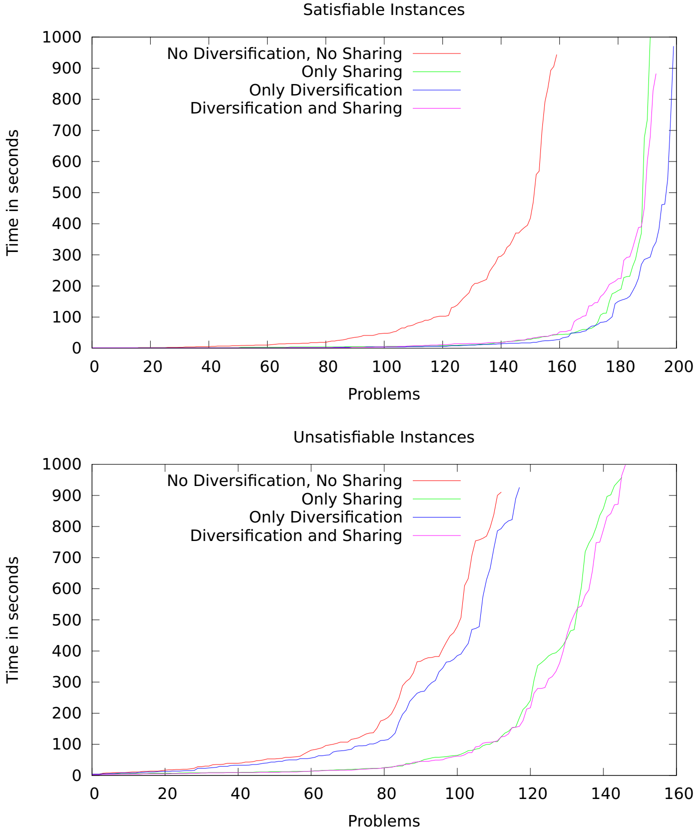

Fig. 1.3: The influence of diversification and clause sharing on the performance of HordeSat[6] on random 3-SAT problems. Plot is taken from [6]

detrimental for satisfiable problems. On the other hand, diversification has only a small benefit for unsatisfiable instances but high impact for satisfiable ones.

This is actually in accordance with what one would expect. To solve a satisfiable formula we only need to find a satisfying assignment anywhere in the search space. To do this efficiently we only need to diversify the search (which we also do when only allowing clause sharing). On the other hand, for unsatisfiable problems we must actually construct a resolution proof. The different solvers in the portfolio construct different segments of the proof and to get the complete proof we join these segments via clause sharing. Without clause sharing each solver must construct the complete proof alone.

1.4.5 *Examples of Parallel Portfolio Solvers*

ManySat

ManySat [24] was the first successful clause-sharing parallel portfolio SAT solver. It was developed in 2008 and won first place in the Parallel Track of the 2008 SAT Race and 2009 SAT Competition. Most previous parallel SAT solvers were designed using the divide-and-conquer paradigm but since ManySat the parallel tracks of all SAT Competitions and SAT Races have been dominated by portfolio solvers.

ManySat was implemented on top of the well-known sequential SAT solver MiniSat [54] and the basic idea is that each parallel process should exploit a particular parameter set such that their combination represents a set of orthogonal yet complementary strategies.

In the original version the authors defined four different strategies. Each of the four strategies featured a different restart scheme and different decision literal polarity heuristic. All four strategies used the VSIDS [54] decision variable selection heuristic with a different percentage of random choices. Additionally half of the strategies employed extended clause learning, which allows for bigger backjumps.

ManySat was designed for parallel systems with the shared-memory architecture – basically for multi-core/multi-CPU computers. The clause sharing was organized via lockless queues containing the clauses a particular solver wants to share. Unit clauses were imported only at restarts while longer clauses were imported immediately on the fly. Overall, all the clauses with eight or fewer literals were shared. The value eight was determined based on experimental evaluation using SAT Competition benchmarks.

The performance of ManySAT was evaluated (in the 2008 SAT Race) on four-core computers where it achieved a super-linear average speedup of 6.02 [24]. ManySAT was the most successful parallel solver (considering SAT Competitions/Races results) until 2010 when Plingeling [9] took over.

Plingeling

Plingeling is the parallel version of the CDCL SAT solver Lingeling [8]. Both solvers first appeared in the 2010 SAT Race, where Lingeling placed second in the Main Track and Plingeling won the Parallel Track. In most of the following SAT Competitions and SAT Races[2] Plingeling won and Lingeling placed among the top three solvers. Plingeling constantly evolved during these years and in the remainder of this subsection we will describe the changes since the 2010 version up to the 2016 version.

2010. Similarly to ManySAT, Plingeling is a portfolio solver implemented on top of Lingeling using Pthreads. A boss thread reads the input formula and generates

[2] At least up until the 2016 SAT Competition, which was the latest competition at the time of writing this text.

separate solver instances for worker threads. The diversification between the worker threads is achieved by setting different random seeds, preprocessing effort, and decision heuristics. The workers share clauses, but only unit clauses (clauses with one literal) are exchanged. This is done via the boss thread at regular intervals. The boss thread maintains a global unit table that is lazily synchronized among the workers.

2011. Additionally to unit clauses, literal equivalences are shared in the 2011 version of Plingeling. An equivalence between literals l_i and l_j can be viewed as a pair of binary clauses $(l_i \vee \neg l_j)$ and $(\neg l_i \vee l_j)$, therefore Plingeling now shares unit and some binary clauses. However, this is not how Plingeling implements this feature. The sharing of equivalences is implemented via a global union-find data structure of equivalences maintained by the boss thread.

2012. There is not much change regarding the features of the portfolio, however, there are changes in implementation. Now the boss thread alone does the input parsing and preprocessing and only then the worker thread solver instances are created. This reduces the memory consumption and makes the solver more robust for large instances, since worker thread addition can be stopped if the amount of available memory is running low.

2013. The sharing of longer clauses is added to Plingeling. Clauses with up to 40 literals are exchanged if their glue value[3] is at most 8. The sharing is implemented via a global clause stack maintained by the boss thread. The worker threads read the clauses from the global stack in the oldest first order, therefore it acts like a queue.

2014. Local Search is added to Plingeling. First the formula is examined in order to figure out whether it resembles a uniform random instance. This is done by looking at the average number of literal occurrences and its standard deviation. If the formula looks random several worker threads (or even all but one) run local search instead of CDCL (Lingeling). This was done to make Plingeling competitive also on random satisfiable problems that are still best solved by local search algorithms.

2015. Diversification is improved. Automatic parameter configuration techniques [29] are applied to find optimal parameter settings for various families of benchmarks from previous competitions. Each one of the worker threads uses one of these configurations.

2016. The parallel front-end is identical to the previous version, i.e., no changes besides use of the newest version of Lingeling by the worker threads.

In Figure 1.4 we plot the number of problems solved per time limit for each version of Plingeling. Bear in mind that each different Plingeling version uses a different Lingeling version at its core. We can see that the 2015 and 2016 versions perform best and are very similar, which is not at all surprising based on their description. The third best is the 2013 version followed by a large gap and then 2014 and the remaining versions. The natural question here is what went wrong with the 2014 version that it fell behind so much compared to the 2013 version. We believe it was caused by local search being used on instances that were not uniform random. Based on experimental logs we know that version 2014 solved significantly fewer

[3] The number of distinct decision levels associated with the literals of the learned clause.

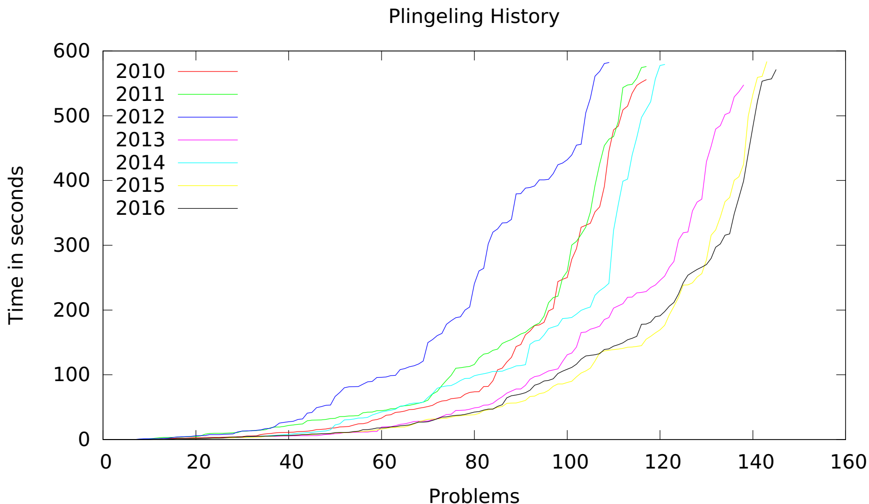

Fig. 1.4: The performance of Plingeling versions 2010 - 2016 on the benchmark of problems of the 2016 SAT Competition

unsatisfiable problems than 2013, 2015, and 2016 while solving a similar number of satisfiable problems.

HordeSat

HordeSat is a portfolio SAT solver designed for massively parallel architectures, i.e., computer clusters with hundreds of multi-core computers. An overview of the high-level design decisions made when designing HordeSat follows.

Modular Design. Rather than committing to any particular SAT solver HordeSat uses an interface that is universal and can be efficiently implemented by current state-of-the-art SAT solvers. This results in a more general implementation and the possibility to easily add new SAT solvers to the portfolio.

Decentralization. All the nodes in the parallel system are equivalent. There is no boss or central node that manages the search or the communication. Decentralized design allows more scalability and also simplifies the algorithm.

Overlapping Search and Communication. The search and the clause exchange procedures run in different (hardware) threads in parallel. The system is implemented in such a way that the search procedure never waits for any shared resources, at the expense of losing some of the shared clauses.

Hierarchical Parallelization. HordeSat is designed to run on clusters of computers (nodes) with multiple processor cores, i.e., we have two levels of parallelization. The first level uses the shared-memory model to communicate between solvers running

on the same node and the second level relies on message passing between the nodes of a cluster.

HordeSat defines a C++ interface that is used to access the instances of the core solvers. This interface has the following methods.

- `addClause(vector<int> clause)` add clauses of the input formula
- `solve()` start solving, returns SAT/UNSAT/UNKNOWN
- `setInterrupt()` tell the solver to stop the search
- `unsetInterrupt()` allow the solver to continue solving
- `setPhase(int var, bool val)` suggest a truth value for a variable. This is just a recommendation and can be ignored by the solver.
- `diversify(int rank, int size)` tell the core solver to diversify its settings. The specifics of diversification are left to the solver. The provided parameters can be used by the solver to determine how many solvers are working on this problem (`size`) and which one of those is this solver (`rank`). A trivial implementation of this method could be to set the pseudo-random number generator seed of the core solver to `rank`.
- `addLearnedClause(vector<int> clause)` add a learned clause to the core solver. The solver can decide whether and how long this clause is useful for it.
- `setLearnedClauseCallback(LCCallback* lcc)` the solver calls the callback function when it learns a clause to share it.

The interface is designed to closely match current CDCL SAT solvers, but any kind of SAT solver can be used. For example a local search SAT solver could implement the interface by ignoring the calls to the clause-sharing-related methods.

Since HordeSat can only access its core solvers via the interface defined above, the only tools for diversification are setting phases using the `setPhase` method and calling the solver-specific `diversify` method.

The `setPhase` method allows the partitioning of the search space in a semi-explicit fashion. An explicit search space splitting into disjoint subspaces is usually done by imposing phase restrictions instead of just recommending them. The explicit approach is used in parallel solvers based on the divide-and-conquer methodology described in Section 1.3.

In HordeSat each variable in each core solver gets a random phase recommendation with a probability of $(\#solvers)^{-1}$, where $\#solvers$ is the total number of core solvers in the portfolio. This is done in conjunction with the `diversify` method whose behavior is defined by the core solvers.

The clause sharing in HordeSat happens periodically in rounds. Each round a fixed sized (1,500 integers in the implementation) message containing the literals of the shared clauses is exchanged by all the processes in an all-to-all fashion. Each process prepares the message by collecting the learned clauses from its core solvers. The clauses are filtered to remove duplicates. The fixed-sized message buffer is filled up with the clauses; shorter clauses are preferred. Clauses that do not fit are discarded.

The detection of duplicate clauses is implemented by using Bloom filters [13]. A Bloom filter is a space-efficient probabilistic set data structure that allows false-

positive matches, which in this case means that some clauses might be considered to be duplicates even if they are not.

Although important learned clauses might get lost, we believe that this relaxed approach is still beneficial since it allows a simpler and more efficient implementation of clause sharing.

1.5 Parallel Local Search

There are two kinds of approaches to parallelizing local search. One is doing multiple flips in parallel and the other is the portfolio approach described above for CDCL algorithms. A special kind of local search called Survey Propagation has also been parallelized using GPU computation [39].

1.5.1 Multiple Flips

The parallel version of the local search solver GSAT [49] called PGSAT [44] first divides the set of variables into k groups (typically k is the number of processors) and then in each iteration each processor flips one of the variables from its group until a solution is found. The variable is selected using the GSAT heuristic.

Experiments with PGSAT have shown that speedup is achieved only up to a specific value of k. After this optimal value of k (denoted by k^*) the performance drops. An interesting observation is that the value of k^* depends on the instance we want to solve and appears to be correlated to the average connectivity of the variable-clause graph of the instance [44].

The solver PGWSAT [43] is a combination of PGSAT [44] and WalkSat [48]. In each iteration PGWSAT either acts like WalkSat (flipping a literal from one of the unsatisfied clauses) or PGSAT. The behavior is chosen randomly with the WalkSat strategy being used on between 50% and 70% of the steps. PGWSAT is shown to outperform PGSAT on random 3-SAT instances.

A parallel version of the solver genSat (generalized GSAT) [56] flips all the variables that have the best GSAT score (number of unsatisfied clauses after the flip) in each iteration. This is in contrast to other GSAT-style algorithms where only one of the variables with the best score is flipped (ties are broken randomly). Parallel genSat is experimentally shown to require fewer flips to solve a problem than the original GSAT algorithm.

1.5.2 Portfolios

The basic idea of portfolios (running several different solvers in parallel) can be applied to local solvers the same way as it is used for CDCL. The first local search solver to do this was gNovelty+ (v. 2)[42] in the 2009 SAT Competition, where it achieved first place in the parallel random category. The solver did not do any kind of sharing so it was pure portfolio.

It is not clear what kind of information should be shared in a portfolio of local search solvers. The sharing of learned clauses cannot be adopted from CDCL portfolios since local search solvers cannot produce them.

Several strategies of information sharing in local search portfolios were suggested in [2]. The solvers exchange their best assignment (satisfying the highest number of clauses) and based on these assignments a new starting assignment is constructed. This construction is different for each cooperation strategy. The strategies range from certain voting mechanisms to various probabilistic constructions. The most successful strategy, called *Prob-NormalizedW*, constructs the assignment using a probabilistic method that ensures that better variable values (w.r.t. satisfied clauses) have a higher chance of being adopted.

In a follow-up work [1] it was shown that this approach does not scale well for massively parallel systems and the proposed solution was to split the solvers into smaller groups (e.g., 16 solvers) that cooperate internally but do not exchange information between the different groups.

1.6 Future Challenges

In 2013 Hamadi and Wintersteiger published a paper listing seven challenges in parallel SAT solving [26]. The first challenge is to design a way to automatically estimate the number of parallel processes that should be used to solve a formula. The second challenge is about finding new ways to decompose the input formulas or the search space of a SAT-solving algorithm that outperform current techniques. The third challenge is about parallelizing preprocessing techniques used in modern SAT solvers. The next challenge is related to clause sharing and it asks for better techniques for estimating the local quality of learned clauses coming from other solvers. The technique called "lazy clause exchange" [5] used in the parallel version of the SAT solver Glucose [4] is a step towards solving this challenge. Challenges five and six ask for new encodings that would be specifically designed for parallel SAT solvers. Finally, the seventh challenge is to design a completely new parallel SAT algorithm from scratch, i.e., not based on existing algorithms, that performs on a par with or better than the current state of the art. We extend this list by adding the following three new challenges:

Massively Parallel Sat Solving. Design SAT solvers scalable in highly parallel environments, i.e., computer clusters with thousands or even millions of processors. Such solvers could potentially be used to solve large hard problem instances coming

from computational biology and chemistry and even resolve open problems from fields such as combinatorics and number theory.

Utilizing Graphics Processing Units (GPUs) for SAT. Modern graphics cards are highly parallel computing units with hundreds of cores. General-purpose computing on GPUs is useful to accelerate various algorithms (notably for problems involving matrices) but has not yet been successfully used for SAT solving. Although there have been attempts to adapt existing SAT algorithms for GPU we are yet to see a GPU-based solver outperform standard CPU solvers. It seems that completely new algorithms need to be developed for the GPU.

Parallel Incremental SAT Solving. Many applications of SAT are based on solving a sequence of very similar SAT instances that often only differ in a few clauses. Although these instances can be solved independently, it can be very inefficient compared to an incremental SAT solver, which can reuse knowledge acquired while solving the previous instances (for example some of the learned clauses). Several SAT solvers support incremental SAT solving, however none of them is parallel. Since incremental solvers are very useful for improving performance in practical applications it is very important to develop highly scalable parallel incremental SAT solvers.

Acknowledgements

We would like to thank Wolfgang Blochinger for allowing us to use material from an unpublished draft in Section 1.3.1.

References

[1] Arbelaez, A., Codognet, P.: Massively parallel local search for SAT. In: 2012 IEEE 24th International Conference on Tools with Artificial Intelligence. vol. 1, pp. 57–64. IEEE (2012)

[2] Arbelaez, A., Hamadi, Y.: Improving parallel local search for SAT. In: International Conference on Learning and Intelligent Optimization. pp. 46–60. Springer (2011)

[3] Audemard, G., Lagniez, J.M., Szczepanski, N., Tabary, S.: An adaptive parallel SAT solver. In: International Conference on Principles and Practice of Constraint Programming. pp. 30–48. Springer (2016)

[4] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: International Joint Conference on Artificial Intelligence (IJCAI). vol. 9, pp. 399–404 (2009)

[5] Audemard, G., Simon, L.: Lazy clause exchange policy for parallel SAT solvers. In: Theory and Applications of Satisfiability Testing (SAT), pp. 197–205. Springer (2014)

[6] Balyo, T., Sanders, P., Sinz, C.: Hordesat: A massively parallel portfolio SAT solver. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing (SAT), Lecture Notes in Computer Science, vol. 9340, pp. 156–172. Springer International Publishing (2015)
[7] Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands (2009)
[8] Biere, A.: Lingeling, plingeling, picosat and precosat at SAT race 2010. In: Technical Report 10/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University (2010)
[9] Biere, A.: Lingeling, plingeling and treengeling entering the SAT competition 2013. In: Proceedings of SAT Competition 2013, University of Helsinki. pp. 51–52 (2013)
[10] Blochinger, W., Sinz, C., Küchlin, W.: Distributed parallel SAT checking with dynamic learning using DOTS. In: Gonzales, T. (ed.) Proc. of the IASTED Intl. Conference Parallel and Distributed Computing and Systems (PDCS 2001). pp. 396–401. ACTA Press, Anaheim, CA (Aug 2001)
[11] Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing 29(7), 969–994 (2003)
[12] Blochinger, W., Westje, W., Küchlin, W., Wedeniwski, S.: ZetaSAT – boolean SATisfiability solving on desktop grids. In: IEEE International Symposium on Cluster Computing and the Grid. pp. 1079–1086 (2005)
[13] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
[14] Boehm, M., Speckenmeyer, E.: A fast parallel SAT-solver – efficient workload balancing. Annals of Mathematics and Artificial Intelligence 17(3-4), 381–400 (1996)
[15] Chrabakh, W., Wolski, R.: GridSAT: A Chaff-based distributed SAT solver for the grid. In: Proc. of Supercomputing 03. Phoenix, Arizona, USA (2003)
[16] Cook, S.A.: The complexity of theorem-proving procedures. In: ACM Symposium on Theory of Computing. pp. 151–158. ACM, New York, NY, USA (1971)
[17] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
[18] Dijkstra, E.W., W.H.J.Feijen, van Gasteren, A.: Derivation of a termination detection algorithm for distributed computations. Inf. Proc. Letters 16, 217–219 (1983)
[19] El Khalek, Y.A., Safar, M., El-Kharashi, M.W.: On the parallelization of sat solvers. In: Computer Engineering & Systems (ICCES), 2015 Tenth International Conference on. pp. 119–128. IEEE (2015)
[20] Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel multithreaded satisfiability solver: Design and implementation. Electr. Notes Theor. Comput. Sci. 128(3), 75–90 (2005)

[21] Gu, J.: The multi-sat algorithm. Discrete Applied Mathematics 96-97, 111–126 (1999)
[22] Guo, L., Jabbour, S., Lonlac, J., Saïs, L.: Diversification by clauses deletion strategies in portfolio parallel SAT solving. In: Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on. pp. 701–708. IEEE (2014)
[23] Hamadi, Y., Jabbour, S., Piette, C., Sais, L.: Deterministic parallel DPLL. Journal on Satisfiability, Boolean Modeling and Computation 7, 127–132 (2011)
[24] Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel SAT solver. In: Satisfiability, Boolean Modeling and Computation. vol. 6, pp. 245–262 (2008)
[25] Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel sat solving. In: Twenty-First International Joint Conference on Artificial Intelligence (2009)
[26] Hamadi, Y., Wintersteiger, C.: Seven challenges in parallel SAT solving. AI Magazine 34(2), 99 (2013)
[27] Heule, M., Manthey, N., Philipp, T.: Validating unsatisfiability results of clause sharing parallel SAT solvers. In: POS@ SAT. pp. 12–25 (2014)
[28] Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding cdcl SAT solvers by lookaheads. In: Haifa Verification Conference. pp. 50–65. Springer (2011)
[29] Hutter, F., Hoos, H.H., Leyton-Brown, K., Stuetzle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (October 2009)
[30] Hyvärinen, A.E., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: International Conference on Theory and Applications of Satisfiability Testing (SAT'06). pp. 430–435 (2006)
[31] Järvisalo, M., Le Berre, D., Roussel, O.: The SAT 2011 Competition – Results of Phase 1 – slides. http://www.cril.univ-artois.fr/SAT11/phase1.pdf (2011), accessed: 2015-12-18
[32] Jurkowiak, B., Li, C., Utard, G.: Parallelizing Satz using dynamic workload balancing. In: Kautz, H., Selman, B. (eds.) LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001). Electronic Notes in Discrete Mathematics, vol. 9. Elsevier Science Publishers, Boston, MA (Jun 2001)
[33] Jurkowiak, B., Li, C.M., Utard, G.: A parallelization scheme based on work stealing for a class of SAT solvers. Journal of Automated Reasoning 34(1), 73–101 (2005)
[34] Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L.: Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In: Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA. (2013)
[35] Kaufmann, M., Kottler, S.: Sartagnan parallel portfolio SAT solver with lockless physical clause sharing. In: Pragmatics of SAT. Citeseer (2011)
[36] Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Estimating search tree size. In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2. pp. 1014–1019. AAAI'06, AAAI Press (2006)

[37] Knuth, D.E.: Estimating the efficiency of backtrack programs. Mathematics of Computation 29(129), 121–136 (Jan 1975)
[38] Kokotov, L.: Distributed SAT solver framework (1998)
[39] Manolios, P., Zhang, Y.: Implementing survey propagation on graphics processing units. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 311–324. Springer (2006)
[40] Manthey, N.: Towards Next Generation Sequential and Parallel SAT Solvers. Ph.D. thesis, Technischen Universität Dresden, Fakultät Informatik (Jan 2014)
[41] Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
[42] Pham, D.N., Gretton, C.: gnovelty+ (v. 2). In: Proceedings of SAT Competition 2009, Artois University. pp. 9–10 (2009)
[43] Roli, A., Blesa, M., Blum, C.: Random walk and parallelism in local search. In: Proceedings of MIC'2005 – Meta–heuristics International Conference. Vienna, Austria (2005)
[44] Roli, A.: Criticality and parallelism in structured SAT instances. In: International Conference on Principles and Practice of Constraint Programming. pp. 714–719. Springer (2002)
[45] Roussel, O.: Description of ppfolio 2012. Proc. SAT Challenge p. 46 (2012)
[46] Schubert, T., Lewis, M., Becker, B.: PaMira - a parallel SAT solver with knowledge sharing. In: 6th International Workshop on Microprocessor Test and Verification (2005)
[47] Schubert, T., Lewis, M.D.T., Becker, B.: Pamiraxt: Parallel SAT solving with threads and message passing. JSAT 6(4), 203–222 (2009)
[48] Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: AAAI. vol. 94, pp. 337–343 (1994)
[49] Selman, B., Levesque, H.J., Mitchell, D.G., et al.: A new method for solving hard satisfiability problems. In: AAAI. vol. 92, pp. 440–446 (1992)
[50] Sinz, C., Blochinger, W., Küchlin, W.: PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In: Kautz, H., Selman, B. (eds.) LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001). Electronic Notes in Discrete Mathematics, vol. 9. Elsevier Science Publishers, Boston, MA (Jun 2001)
[51] Sonobe, T., Inaba, M.: Counter implication restart for parallel SAT solvers. In: Learning and Intelligent Optimization, pp. 485–490. Springer (2012)
[52] Sonobe, T., Inaba, M.: Portfolio with block branching for parallel SAT solvers. In: International Conference on Learning and Intelligent Optimization. pp. 247–252. Springer (2013)
[53] Sonobe, T., Kondoh, S., Inaba, M.: Community branching for parallel portfolio SAT solvers. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 188–196. Springer (2014)
[54] Sorensson, N., Een, N.: Minisat v1.13 a SAT solver with conflict-clause minimization. Tech. rep., Chalmers University of Technology, Sweden (2005)

[55] Speckenmeyer, E., Monien, B., Vornberger, O.: Superlinear speedup for parallel backtracking, pp. 985–993. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
[56] Strohmaier, A.: Multi-flip networks: parallelizing gensat. In: Annual Conference on Artificial Intelligence. pp. 349–360. Springer (1997)
[57] Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. AAAI Conference on Artificial Intelligence (2010)
[58] Zhang, H., Bonacina, M.P.: Cumulating search in a distributed computing environment: A case study in parallel satisfiability. In: Proc. of the First Int. Symp. on Parallel Symbolic Computation. pp. 422–431. Linz, Austria (1994)
[59] Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21, 543–560 (1996)