A Short Overview on Modern Parallel SAT-Solvers

Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, Julian Stecklina, Peter Steinke
Faculty of Computer Science, Technische Universität Dresden, 01062 Dresden, Germany
sh@iccl.tu-dresden.de

Abstract

This paper surveys modern parallel SAT-solvers. It focusses on recent successful techniques and points out weaknesses that have to be overcome to exploit the full power of modern multi-core processors.

1. Introduction

The Boolean Satisfiability Problem (SAT) is one of most-researched NP-complete problem in Computer Science [6]. Over the last twenty years improvements on all levels, from the logic over calculi, heuristics and data structures to low-level processes, have turned modern sequential SAT-solvers into practical tools in many application areas like hardware and software verification, planning, or bioinformatics (see [2]).

An analysis of modern sequential SAT-solvers has revealed that their efficiency and speed hinges on the appropriate usage of the available hardware and low level processes (see e.g. [16] for details). The steady improvement of single processor systems in the last decades has also helped to increase the performance of SAT-solvers considerably. However, we currently observe a dramatic change from single processor systems to multi-core designs including integrated memory controllers and large caches leading to non-uniform memory access latencies.

Modern SAT-solvers should utilize multi-core design. Being inherently sequential from a theoretical and worst-case analysis point of view, SAT gives rise to various open problems: How can SAT be solved in a multi-core system effectively? Which calculi support multi-core systems? Which data structures, heuristics, and low level processes make appropriate use of the non-uniform memory access latencies? Which kind of communication shall be employed? How can we compare the performance of parallel SAT-solvers?

After stating preliminaries in Section 2, sequential SAT-solvers are characterized in Section 3. The main contribution is a survey on modern parallel SAT-solvers in Section 4. In Section 5 we review modern multi-core architectures. A second contribution is a first attempt to specify requirements for the implementation of SAT-solvers on multi-core designs in Section 6.

2. Preliminaries

A problem that should be solved by a SAT-solver is usually specified in conjunctive normal form (CNF) of propositional logic. This form restricts propositional logic to variables and the connectives *negation* ($\neg$), *disjunction* ($\vee$), and *conjunction* ($\wedge$). A *literal* consists of a variable A and is either positive (A) or negated ($\neg A$). A *clause* is a disjunction of literals. Finally, a *(CNF) formula* is a conjunction of clauses.

An *interpretation* is a mapping from formulas to the set of truth values $\{\top, \bot\}$. SAT-solvers use *partial interpretations*, where only some of the variables are assigned. Partial interpretations are applied to formulas by replacing the already assigned variables by their truth values and simplifying the formula accordingly.

The Boolean Satisfiability Problem is the question whether there exists an interpretation for a formula such that the formula evaluates to $\top$ under this interpretation. For more details about SAT see [2].

3. Sequential SAT-Solving

Modern sequential SAT-solvers are based on the *Davis-Putnam-Loveland-Logemann (DPLL)* algorithm [7]. The algorithm consists of rules which are applied to generate and traverse a binary (semantic) search tree. Each branch of the search tree represents a (partial) interpretation. Let F be the given formula and I the interpretation represented by branch B. We distinguish the following cases: (i) If I maps F to $\top$, then a model has been found and F is satisfiable. (ii) If I does not map F to a truth value, then B is expanded by the so-called *split rule*: a currently unassigned variable is assigned a new truth value and a backtrack point is recorded. Afterwards, I is extended by all the implications that can be found with respect to the formula. This is mainly done by the *unit propagation rule*, but additional simplification rules are possible. (iii) If a clause of F is mapped to $\perp$ by I, then this clause is called *conflict (clause)* and B can be closed. Thereafter, naive backtracking is applied to explore most recent alternative branches in the search tree.

The idea to analyze the conflict clause further led to the *conflict driven clause learning (CDCL)* algorithm [33]. Resolving the conflict clause and the clauses which have been used in the implications, new clauses are learned. These learned clauses can be added to the given formula leading to an improved backtracking behavior, where larger parts of the search tree are closed by a single conflict. The most commonly used version of this algorithm is described in [37].

Restarts are another technique to improve the performance. In this case, the search for a satisfying assignment is started from scratch again. Since previously learned clauses are kept, the search tree is usually explored in a very different way (see e.g. [31]). In certain cases it appears to be promising to *look-ahead* by assuming unit literals to gather extra information about the formula.

The most commonly used SAT-solver that implements most of the mentioned techniques is MINISAT 2.2 ([29], [8]). This solver has been first implemented in 2003 and has been improved each year, providing a basis for many sequential and parallel SAT-solvers.

4. Development of Parallel SAT-Solvers

Based on single-core architectures, the first parallel SAT-solvers were based on network communication. With the occurrence of multi-core CPUs, shared memory was also used for communication. Orthogonally, with the move from DPLL- to CDCL-solvers including restarts, different techniques were developed to parallelize the algorithms for solving SAT-instances.

In 2006, a first overview on parallel SAT-solvers was presented in [34]. However, much of the work on parallel SAT-solvers was done in the last five years and we focus on these recent developments. We further restrict our attention to complete solvers, whose parallelization should reach a superlinear speedup for satisfiable instances. Due to lack of space we present only the–from our point of view–most important contributions. A complete overview including all references which we are aware of is given in [17].

4.1. DPLL-based parallelizations

The recursive application of the split rule in the DPLL algorithm provides a natural way to parallelize the search. Initially, each computing node receives the given formula. Thereafter, only partial interpretations are communicated such that each computing node receives a different partial interpretation. The first parallel SAT-solvers used single-core CPUs which communicated via a network. When shared memory architectures became available, shared memory communication was utilized. Two parallelizations of the SAT-solver SATZ have been compared: a version using single-core CPUs with network communication and a version using a multi-core CPU with shared memory communication. The study showed (i) that the decision heuristic in SATZ is more powerful than picking random variables or using the MOMS heuristic, and (ii) that network communication seems to outperform shared memory approaches. The latter is due to the observation that the parallel incarnations have to share resources over the shared memory, whereas the network communication has only overhead in the communication time and there are only few communication instances.

4.2. CDCL-based parallelizations

The success of the first CDCL-based SAT-solvers led to a dramatic change. Due to the addition of learned clauses the search space is traversed much less orderly than in DPLL and new questions arose like: Which learned clauses should be shared? Which learned clauses should be incorporated into the own search? When shall learned clauses be deleted?

4.2.1. Network communication. The first grid based solver was GRIDSAT. It used a master-slave approach where slaves could be added dynamically. An in-depth discussion is given in [34]. Afterwards, many implementations and new approaches followed.

PMSAT [10] is based on MINISAT 1.14 and MPI. The solver uses a master-slave approach with a fixed number of slaves. The master creates the assumptions that are used to split an instance. For k slaves, $3k$ splitting variables are selected such that 2^{3k} jobs have to be handled. The master stores the splitting variables and sends a partial interpretation to the next free slave. Load balancing is implemented by providing sufficiently many tasks. If a slave has shown that its subformula is undecidable, it returns the 50 most active learned clauses of length less than 21 to the master. Because these clauses are logical consequences of the input formula, the master can forward them to the running slaves. Additionally, the master removes tasks which became unsatisfiable based on the newly received clauses from its tasks queue.

PMSAT implements two approaches for selecting the splitting variables and for applying the selected variables to create subtasks. Either the most frequent variables or the variables that occur most frequently in large clauses are selected. Subtasks are generated by either creating a simple binary search tree based on the possible assignments of these variables or by *scattering* [21]. Considering the four configurations, experiments have been carried out by comparing the performance of the best and the worst one. For the best configuration and unsatisfiable SAT-instances an efficiency[1] upper bound of two has been reported. For satisfiable instances almost always a super-linear speedup could be reached. However, there is the open question of how to determine the best configuration given a SAT-instance.

Another method to solve SAT-instances is by applying different search strategies in parallel and independently. In [19] different restart strategies have been implemented into MINISAT 1.14 to be executed in a grid. In particular, the restart schemas based on the Luby series and the exponential series $2^{1.2X}$, where X is the number of the next restart, are investigated. Two restart parallelization schemas are analyzed: the *straightforward* and the *faithful* schema. Based on the interval between two restarts, tasks are generated. Note that there is no communication between the solvers in the grid as they solve their tasks independently. Without grid delay the efficiency ranges between 0.5 and 1. However, by including the grid there is no super-linear speedup. The study also revealed that the more parallel solvers are used, the less important is the used restart strategy. A small number of parallel solvers suffices to solve instances fast, but the scalability is limited. Still, by using a grid, large sets of instances could be tackled by the approach.

Grid solutions, where learned clauses of running tasks are collected and shared among future, parallel tasks, are investigated in [18]. The sharing increases the performance most of the time, but no superior sharing heuristic has been found. Moreover, some hard instances, which sequential solvers could not solve, were solvable by the proposed approach.

The parallel solver C-SAT [30] combines two ways of parallelism: cooperative parallelism by splitting the search space and competitive parallelism by executing different solver configurations. The solver is based on MINISAT 1.14 and MPI. Two decision heuristics are used: the VSIDS heuristics [28] and a heuristics, where the activity is stored per literal. Search space splitting is implemented by selecting split variables from the current path in the search tree. C-SAT is organized in three layers. In the first layer a grand-master connects to several masters as its slaves. The grand-master distributes the input formula to the masters and shares learned (and simplified) clauses among the masters. All masters work on the same input formula. Each master maintains a group of slaves that work on subtasks. The highest performance of C-SAT has been achieved by using both decision heuristics in parallel. In particular, the efficiency of this approach is super-linear for satisfiable SAT-instances and more than 0.6 for unsatisfiable SAT-instances.[2] The experiments further revealed that by running more slaves the chance of learning redundant clauses increases. Finally, the stability of the solver with respect to runtime increases if more processing units are used.

Another new approach is to not only solve subformulas but also the input formula itself in parallel. Based on that idea three search tree partitionings are compared in [20]: running solvers in parallel, using look ahead to select branch variables, and scattering. Due to the grid environment, each job has a timeout: splitting was limited tp 5 and solving to 90 minutes. The comparison of the partition techniques shows that the DPLL look ahead partitioning can solve instances faster than scattering. However, the latter is able to solve more instances. The disadvantage of not using learned clauses at all has been tackled in [22]: In the *assumption tagging* all learned clauses are extended with the literals on the path to the current subformula, whereas the *flag-based tagging* approach tags each learned clause that is based on the assumptions as invalid. Experimental results show that *flag-based tagging* slows down solving simple formulas but improves for more difficult SAT-instances.

4.2.2. Shared-memory communication. The first multi-core system were built by combining two single-core CPUs, where each CPU had its own memory bus and main memory. Unfortunately, the non uniform memory accesses slowed the performance down. By combining several cores into a single CPU memory access became uniform again and the access times of sequential and parallel solver became more similar, speeding up the solver again.

The first shared memory solver based on CDCL is PASAT and is discussed in more detail in the survey [34]. Again, many implementations and new approaches followed. The performance of parallel solvers

1. The efficiency is the ratio of the sequential time T_s and the product of the parallel time T_p and the number of processors p.

2. The means reported for C-SAT are geometric means, whereas all the other reported means are arithmetic means.

can be increased further by combining several multi-core computing systems to a network. This approach has been applied to PaSAT in [3] where the incarnations used different heuristics and shared clauses over the network. The authors notice that the runtime distribution heavily depends on clause learning and sharing, because different parts of the search space might be analyzed in different runs.

The solver ySAT has been used to study the efficiency of shared memory solvers in [9]. The formula, the task queue as well as the structure for clause sharing are globally accessible. The authors report a write blocking overhead of up to 10 % for their parallel solver. The performed scalability analysis is based on several architectures and an input formula of size 1.5 MB: The number of learned clauses scales with the number of used cores. The clocks per instruction grow with the number of used cores. When 4 threads are used, this ratio increases to 3.7 and the number of stall cycles increases [16] because the memory is shared. It can be assumed that the measured effect is much smaller, if size of the input formula is increased. However, [9] concludes that using multi-core parallelization cannot be done efficiently.

The solver MiraXT [26] differs from ySAT in several aspects. The preprocessor SatELite is applied, which appears to remove some *bad* splitting variables. To implement the two-watched-literal unit propagation, each task uses additional literal references to store the currently watched literals. Whereas all learned clauses are shared without delay as in ySAT, each task can decide which clause it incorporates. An experimental evaluation showed that the two-core solver is more powerful than the single core solver with an efficiency below 1. Extending MiraXT to networks [32] increases the performance further. Although the efficiency of PaMiraXT [32] is only about 0.25 for 8 cores, it is still able to solve many instances faster than reference solvers as MiniSAT 2.

A novel idea of the solver pMiniSATis to keep learned clauses if they reduce to a unit under a partial interpretation of some task in the task queue. Thus, whenever an idle solver is assigned the next task, it can propagate the corresponding units by incorporating the shared clauses.

[27] analyses the splitting of an instance. The solver first splits the instance and if the runtime for the splitting mechanism reaches a certain limit, it switches to the *portfolio approach* (see Section 4.2.3). The novel idea for splitting is to execute several CDCL solvers with the VSIDS decision heuristic in parallel. After a certain time, the activities of the variables are cumulated and the variables with the highest activities are chosen as splitting variables. It turned out that the hybrid solver is more powerful than a single one, and running four Sat4J [25] incarnations on a quad-core CPU in parallel slows down each solver by 25 %.

Another recent approach is to combine a look-ahead SAT-solver with CDCL-solvers [15]. The expensive look-ahead procedure is used to split the formula into many sub-formulas, which are solved by the CDCL-solvers afterwards. Because the sub-instances can be solved in parallel, this approach is well suited for a multi-core architecture and it has been shown that it solves some hard instances, which could not be solved before in reasonable time.

4.2.3. Pure Portfolio solvers.

The gain of frequent restarts in sequential SAT-solvers led many researchers to move from cooperative parallelism by splitting the search space to competitive parallelism where all parallel solvers try to find a solution for the same SAT-instance. The latter are often called *portfolio solvers*.

With ManySAT [12] portfolio solvers became popular. ManySAT applies several restart, decision and learning heuristics to its four parallel instances. The obtained efficiency for the chosen combination of the heuristics is reported with 1.5 when compared to MiniSAT 2.1. The size threshold for clause sharing has been moved from static to dynamic because the size of learned clauses increases over time. The dynamic threshold is controlled by the number of communications between two tasks. This led to an improved performance of *ManySAT*. By additionally adding a quality measure, 6 more instances out of the 201 instances of the SAT Race 2008 could be solved.

Exchanging extra information and dividing incarnations into masters and slaves boosts ManySAT further [11]. Experiments showed that slaves and masters searching in the same part of the search space (which is achieved by exchanging the information about the literals involved in the last conflict analysis of the master) yields the best performance. It turned out that the most promising topology for four threads consists of two masters and two slaves. In comparison to the original version of ManySAT, this configuration is able to solve 221 out of 292 instances on the industrial benchmark of the SAT Competition 2009.

Because parallel solvers are non-deterministic, it is hard to reproduce previous runs. It has been shown in [13] that sensitively adding sharing *barriers* to the solver solves the problem. The number of conflicts between two barriers is determined per thread to avoid waiting times. With this technique, the deterministic parallel solver performed at least as high as the non-deterministic version of ManySAT.

A different portfolio approach has been implemented in the solver SArTagnan [24], which supports eight cores. The first six cores follow the CDCL algorithm, the seventh core uses *decision making with reference points* and the last core tries to simplify the formula. Because the formula is shared physically, all simplifications become visible for all thread immediately.

5. Modern Architecture

Single processor systems have dominated computing for a long time. The steady performance improvements per processor generation left little incentive to parallelize applications. Improvements to uniprocessor performance started to decline about ten years ago due to power dissipation problems, almost fully utilized instruction-level parallelism, and missing improvements in memory latency [14]. Parallelism has started to dominate the performance of computer systems.

Another development that is indirectly caused by increased parallelism is the move away from front side bus architectures, in which every processor accesses main memory via a shared bus, because the front side bus can become a performance bottleneck [5]. Current systems avoid this bottleneck by integrating memory controllers inside the processor die [38]. On systems with multiple dies, memory accesses are distributed across multiple controllers. This creates non-uniform memory access latencies depending on which memory controller has to serve the request. This property is exaggerated by large caches that are meant to hide comparatively long memory access delays [4]. To fully utilize the available processing power, an application needs to be aware of which memory is cheap to access.

While the trend toward commercial many-core architectures is not as fast as predicted [1], chip manufacturers are experimenting with designs that incorporate an order of magnitude more cores on a chip than commercially available today, such as Intel's 80-core prototype [36] and its *Single-Chip Cloud Computer* (SCC) [23]. The SCC combines 48 standard, but comparatively weak, cores on a single die. Two cores form a node in a mesh network. Access to main memory is provided by four memory controllers sitting on the edge of the mesh. Communication between cores is facilitated by dedicated message passing buffers. While this processor is not meant for production, similar network-on-a-chip processors are expected to be commercially available soon [35]. Serial applications cannot exploit such systems.

It is never easy to give a reliable outlook into the future, but regarding future processors one can formulate solid assumptions: (1) Multi- and many-core systems will be the norm. A single processor system will be the exception. (2) Single core performance improvements will further decline. (3) Memory access latency and bandwidth will be increasingly non-uniform. While the future may be foggy, it is clear that current software needs to change to adapt to hardware developments.

6. Conclusion

Until now, modern CPUs contain only few cores. Recent parallel SAT-solvers for such shared-memory architectures are mainly based on the techniques developed for sequential SAT-solvers. In most cases, they either run few sequential solvers independently with different random seeds and heuristics or they share selected learned clauses. However, as soon as the individual solvers running on different cores share memory, the efficiency of the individual solvers decreases. None of these parallel solvers seem to scale well if hundreds or even thousands of cores become available. None of these parallel solvers seem to be aware and make use of the specific features of modern computer architectures like non-uniform memory access latencies.

A lesson learned from the development of sequential SAT-solvers is that these solvers are only fast if they take the specific features of the underlying hardware into account. We expect that this holds for parallel SAT-solvers as well. Efficient parallel SAT-solvers running on many-core systems should be based on the following principles: (1) The solver should be aware of the underlying hardware to utilize the non-uniform memory access. (2) The exchange of learned clauses and other data between the processes running on different cores has to be carefully balanced on all levels from the calculus and heuristics level up to the data structure, implementation, and hardware level.

Designing a parallel SAT-solver respecting these principles seems to be difficult on the basis of current sequential state-of-the-art solvers. To create a parallel solver for many-core systems, the complete design of sequential solvers needs to be reviewed and redesigned. It remains an open question of how to design future SAT-solvers to achieve reasonable efficiency and high scalability on modern multi-core architectures.

References

[1] K. Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, U. of California, Berkeley, 2006.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[3] W. Blochinger, C. Sinz, and W. Küchlin. A Universal Parallel SAT Checking Kernel. In *Proc. PDPTA:1720–1725*, 2003.

[4] S. Boyd-Wickizer et al. Corey: an operating system for many cores. In *Proc. OSDI:43–57*, 2008.

[5] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *Micro, IEEE*, 27(2):10–21, 2007.

[6] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC:151–158*, 1971.

[7] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *CACM*, 5(7):394–397, 1962.

[8] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. SAT:502–518*, 2004.

[9] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. In *Proc. PDMC:75–90*, 2005.

[10] L. Gil, P. Flores, and L. M. Silveira. PMSat: a parallel version of MiniSAT. *JSAT*, 6:71–98, 2008.

[11] L. Guo et al. Diversification and intensification in parallel SAT solving. In *Proc CP:252–265*, 2010.

[12] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT 6:245–262*, 2009.

[13] Y. Hamadi et al. Deterministic Parallel DPLL: System Description. In *Pragmatics of SAT*, 2011. To appear.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.

[15] M. J. Heule et al. Cube and conquer: Guiding CDCL SAT solvers by lookaheads, 2011. Accepted for HVC 2011.

[16] S. Hölldobler, N. Manthey, and A. Saptawijaya. Improving resource-unaware SAT solvers. In *Proc. LPAR: 357–371*, 2010.

[17] S. Hölldobler et al. Modern Parallel SAT-Solvers, TR 2011-6, Knowledge Representation and Reasoning Group, TU Dresden, Germany, 2011.

[18] A. E. Hyvärinen, T. Junttila, and I. Niemelä. Incorporating Learning in Grid-Based Randomized SAT Solving. In *Proc. AIMSA:247–261*, 2008.

[19] A. E. Hyvärinen, T. Junttila, and I. Niemelä. Strategies for Solving SAT in Grids by Randomized Search. In *Proc. AISC/MKM/Calculemus:125–140*, 2008.

[20] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In *Proc. LPAR:372–386*, 2010.

[21] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. A distribution method for solving SAT in grids. In *In Proc. SAT:430–435*, 2006.

[22] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In *Proc. CP:385–399*, 2011.

[23] Intel. *SCC External Architecture Specification*, 2010.

[24] S. Kottler and M. Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011. To appear.

[25] D. Le Berre. Sat4j: a reasoning engine in Java based on the SATisfiability problem (SAT). http://www.sat4j.org.

[26] M. D. T. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *Proc. ASP-DAC:926-931*, 2007.

[27] R. Martins, V. Manquinho, and I. Lynce. Improving Search Space Splitting for Parallel SAT Solving. In *Proc. ICTAI:336–343*, 2010.

[28] M. W. Moskewicz et al. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC:530-535*, 2001.

[29] Niklas Sörensson. MiniSAT2.2 and MiniSAT++1.1. http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf, 2010.

[30] K. Ohmura and K. Ueda. c-sat: A Parallel SAT Solver for Clusters. In *Proc. SAT:524–537*, 2009.

[31] A. Ramos, P. van der Tak, and M. Heule. Between Restarts and Backjumps. In *Proc. SAT:216-229*, 2011.

[32] T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *JSAT 6:203–222*, 2009.

[33] J. P. M. Silva and K. A. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *Proc. ICCAD: 220–227*, 1996.

[34] D. Singer. *Parallel Resolution of the Satisfiability Problem: A Survey*. Wiley Interscience, 2006.

[35] Tilera. TILE-Gx 3000 Series Overview, 2011.

[36] S. Vangal et al. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, 2008.

[37] L. Zhang, C. F. Madigan, and M. H. Moskewicz. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proc. ICCAD:279–285*, 2001.

[38] D. Ziakas et al. Intel QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *Proc. HOTI:1–6*, 2010.