PaSAT – Parallel SAT-Checking with Lemma Exchange: Implementation and Applications

Carsten Sinz, Wolfgang Blochinger, Wolfgang Küchlin

Symbolic Computation Group, WSI for Computer Science
University of Tübingen, 72076 Tübingen, Germany

Abstract

We present PaSAT, a parallel implementation of a Davis-Putnam-style propositional satisfiability checker incorporating dynamic search space partitioning, intelligent backjumping, as well as lemma generation and exchange; the main focus of our implementation is on speeding up SAT-checking of propositional encodings of real-world combinatorial problems. We investigate and analyze the speed-ups obtained by parallelization in conjunction with lemma exchange and describe the effects we observed during our experiments. Finally, we present performance measurements from the application of our prover in the areas of formal consistency checking of industrial product documentation, cryptanalysis, and hardware verification.

Key words: parallel algorithms, learning cooperating agents, applications

1 Introduction

Although SAT is an NP-complete problem and therefore problem instances with exponential run-times can occur in the worst case, speed-up by a constant factor—as it can at best be achieved by parallelization—is nevertheless of interest [BS96, ZBHa96]; here are several reasons for this: First, real-world applications of SAT like hardware verification usually are at the limit of what today's solvers offer, problems beyond the limit are handled with different methods like testing. In the former case reduced run-times enable faster verification cycles, or even lower the limit for incorporating formal methods. Second, in other applications like consistency checking of product documentation [KS00], one has to cope with huge series of propositional proofs. Most of these proofs are very easy and are processed in under a second, but there are rare cases in which run-time is considerably higher.

Here, parallelization can reduce average waiting time for the personnel and thus increase acceptability of SAT-based tools. And third, we may hope to improve sequential algorithms by using new insights gained by the parallel approach. In some cases, especially with cooperating prover tasks, we observe superlinear speed-ups, which may be carried over to a sequential algorithm.

In this paper we present PaSAT, a parallel implementation of a variant of the Davis-Putnam algorithm. It is designated to run on multi-processor computers—which are becoming more and more common—or distributed over a networked cluster of standard PCs. It incorporates search-space pruning techniques like intelligent back-jumping and lemma generation, which is also sometimes referred to as *learning*. Additionally, we integrated the possibility of exchanging lemmas between prover tasks working on different parts of the search tree of the same problem instance, thereby achieving cooperating agents.

The rest of the paper is organized as follows: First we describe the cornerstones of our implementation in-depth, then we present an approach to analyze the effect of learning. After briefly introducing different real-world application areas of SAT, we present experimental results we obtained with PaSAT on these problem classes. We conclude with an overview of related work.

2   Our Parallel Implementation: PaSAT

In the following, we present the cornerstones of our implementation of the Davis-Putnam (DP) algorithm. These are based on ideas from Zhang [ZBHa96] and Silva and Sakallah [SS96]. We will now describe how we adopted these ideas in our prover.

Although we assume the reader to be familiar with the basic DP algorithm, a simplified version of our implementation's algorithm is depicted in Figure 1. The main procedure DP is initially called with an input clause set S and a starting level $d = 0$. The backjump-level is a global variable that is set by the conflict analysis subroutine as soon as an empty clause is derived. Conflict analysis generates a dependency graph with a node for each clause becoming unit or empty, encoding the reasons for these events [SS96]. We will explain this later in more detail.

There has been a lot of work on literal selection strategies [HV95]. However, in PaSAT we did not emphasize on this, and implemented only three simple heuristics: choose a literal from a shortest positive clause (SPC), choose a literal with the maximal number of occurrences (MO), or choose a literal with the maximal number of binary occurrences (MBO). We now turn to the technical details of our prover.

PaSAT is a list-based implementation in C++, i.e. clauses are stored as lists of liter-

```
boolean DP(ClauseSet S, Level d )
{
    while ( S contains a unit clause {L} ) {
        register cause of L becoming unit;    // conflict management
        delete from S clauses containing L;   // unit-subsumption
        delete L̄ from all clauses in S;       // unit-resolution
    }
    if ( ∅ ∈ S ) {                            // empty clause?
        generate conflict induced clause C_C;  // conflict management
        compute backjump-level; add C_C to S;
        return FALSE;
    }
    if ( S = ∅ ) return TRUE;                 // no clauses?
    choose a literal L_{d+1} occurring in S;  // case-splitting
    if ( DP(S ∪ {L_{d+1}}), d + 1 ) return TRUE;       // first branch
    else if ( backjump-level < d) ) return FALSE;
    else if ( DP(S ∪ {L̄_{d+1}}), d + 1) return TRUE;   // second branch
    else return FALSE;
}
```

Fig. 1. The Sequential Davis-Putnam Algorithm with Conflict Management.

als. For each variable we keep pointers to the clauses in which it occurs in order to speed up unit propagation [ZS96]. As parallelization platform we have employed the Distributed Object-Oriented Threads System (DOTS) [BKLW99]. DOTS is a parallel programming toolkit for C++ that integrates a wide range of different computing platforms into a single system environment for high performance computing. DOTS provides a platform independent threads programming API, enhanced with primitives supporting the parallelization of applications from the field of Symbolic Computation, like *join-any* and *cancel* constructs for threads. Furthermore, it makes the threads programming paradigm available in a distributed memory environment. Thus with DOTS, a hierarchical multiprocessor, consisting of a (heterogeneous) cluster of shared-memory multiprocessor systems can be efficiently programmed using a single paradigm (fork/join). In our implementation DOTS threads are dynamically created when processing capacity becomes available.

Dynamic Search Space Partitioning

For the parallel execution of the Davis-Putnam algorithm the search space has to be divided into mutually disjoint portions to be treated in parallel. However, static generation of balanced subproblems is not feasible, since it is very hard to predict the extent of the problem reduction delivered by the unit propagation step and by the conflict analysis in advance. Consequently, when parallelizing the Davis-Putnam algorithm we have do deal with considerably different and completely unpredictable run-times of the subproblems.

We adopt a search space splitting technique presented in [ZBHa96] which is based on the notion of a *guiding path*. A guiding path describes the current state of the search process. More precisely, a guiding path is a path in the search tree from the root to the current node, with additional labels attached to the edges. Each level of the tree where a case splitting literal is added to clause set C, i.e. each (recursive)

call to the DP procedure, corresponds to an entry in the guiding path, and each entry consists in turn of the following information:

(1) The literal L_{d+1} which was selected at level d.
(2) A flag indicating whether we are in the first or in the second branch. We use B to indicate the first branch, where backtracking is needed, and N to indicate the second branch, where no backtracking is needed.

Each entry in the guiding path with flag B set is a potential candidate for a search space division, as the sequential search may have to backtrack to this point and examine the second branch later. The whole subtree rooted at the node corresponding to this entry can thus be examined by another independent process, where at the same time the first process switches the flag in the guiding path from B to N. The second recursive call of the DP procedure is only executed if the backtrack flag is set to B.

Lemma Generation

Lemma generation aims at reducing the search space by adding information derived during the search to the problem instance's clause set. This works as follows. As soon as the DP algorithm reaches a leaf of the search tree which is not a solution, i.e. when an empty clause is found, the reason for the generation of the empty clause (or *conflict*) is analyzed [SS96]. Often not all selected splitting literals L_d are necessary for the conflict to emerge, such that we obtain a set $L_{d_0}, \ldots, L_{d_k}$ of remaining literals, which are a sufficient condition for the conflict. By adding the *conflict-induced clause* $C_C = \overline{L}_{d_0} \vee \ldots \vee \overline{L}_{d_k}$ to the clause set, we can thus prevent a useless repeated search of the same subtree. We will not describe in detail how the literals evoking a conflict are computed, but refer the reader to the literature instead, which describes also some variants of the above idea, e.g. considering not only splitting literals [Zha00], or identification of UIPs (unique implication points) [SS96]. In our implementation we use a procedure similar to Zhang's.

Adding all conflict clauses to the clause set can result in exponential blow-up. Therefore it is common practice to limit the addition of clauses to those containing less than a fixed number of literals. In our implementation we keep clauses up to a fixed length l_1 (in our experiments we set $l_1 = 10$), and moreover we retain all clauses up to a length $l_2 >= l_1$ as long as they are unit clauses, discarding them as soon as two or more literals are becoming unassigned.

Intelligent Backjumping

The generated conflict induced clauses can be used to perform another kind of search space pruning called intelligent backjumping [SS96]. At first, note, that when an empty clause occurs at level d then the conflict-induced clause, say C_1,

must contain literal L_d, as otherwise the empty clause would be independent of the assignment to L_d, and thus already would have appeared on a lower level. Now, if at level $d - 1$ backtracking is required, and examination of the second branch, i.e. adding $\overline{L}_d$ to the clause set, immediately generates another conflict, the new conflict-induced clause C_2 must contain $\overline{L}_d$. The resolvent of both clauses, however, only contains literals set at lower levels, up to a maximum of d_{max}. On all levels $d > d_{\mathrm{max}}$ the detected merged conflict remains, so that we can skip all backtracks until we have reached level d_{max}. We call d_{max} the backjump-level (see also Figure 1).

In conjunction with parallel search space exploration, intelligent backjumping may skip a subtree that was forked off to another prover task. In this case the forked task should be signaled to terminate its search. Incorporating lemma exchange as explained below, allows automatic detection and handling of such cases.

Lemma Exchange

To improve the performance of parallel problem solvers, it is quite common to use cooperation between individual tasks working on the same problem which also integrates the exchange of information. In the realm of propositional SAT-checking, however, we do not know of any work in this direction.

Tasks working on independent parts of the search space and independently generating lemmas for their own use can be considered as learning agents. Each lemma generated by a conflicting assignment is a piece of information that the prover found out about the problem instance. This information may be used—as mentioned above—during subsequent search to cut off parts of the search space. In this light it seems to be quite natural to extend the idea of lemma generation to the case of communicating prover tasks that exchange especially well-suited lemmas. Thereby they share information that is not derivable by unit propagation alone.

Technically, we have put this idea into practice using a globally accessible data structure, the *global lemma store*, which in our current version is held in shared memory. This has the advantage of fast access and easy handling, but on the other hand limits access to the structure to prover tasks running on the same machine. The data structure is realized as a set of queues, especially designed to allow concurrent access without synchronization in order to prevent the global lemma store from becoming a sequential bottleneck. Each prover task now filters the lemmas it generates (using the length as a simple criterion), and puts the best ones into the global lemma store. In regular intervals the prover tasks check if there are new lemmas in the global store, and if this is the case, they integrate them into their clause sets. Before integration a second filter step can be applied, but this is not implemented yet.

A priori, it does not seem to be clear how much the prover tasks can profit from the

exchange of lemmas. Our experiments show that at least on some SAT instances exchanging lemmas can speed up the search process considerably.

3 Analyzing the Effects of Lemma Generation

In order to get a more concise picture of the effect of lemma generation and exchange on the search process, we made some experiments concerning the progress of the search. We expected two antagonistic effects to become apparent when integrating newly derived lemmas into the clause set. First, due to the continuous learning of new facts which allows cutting off growing pieces of the search space, the speed of traversal should accelerate. Second, insertion of new clauses means increasing run-times for the basic operations of the DP algorithm like unit-propagation and literal selection. Which of the two effects dominates the other, and the conditions under which this is the case, we tried to determine empirically.

As a measure of the fraction of the search space that is already processed at a certain point of time the guiding paths of Section 2 are very helpful. Given a SAT instance with n variables, we assume the whole search space to consist of 2^n states, representing the possible assignment functions. Obviously, unit propagation cuts off huge parts of the search space, but we take the view that these parts have still been processed implicitly. During search, we continuously update the guiding path to the current position in the search space, which consists, assuming we are at level d, of the literals $L_1, \ldots, L_d$ together with the information whether we are in the first or second branch at each level. Thus, the guiding path can serve as a description of the current position in the search space. The fraction $f(P_C)$ of the search space that we have already traversed can then be computed from the current guiding path $P_C = ((L_1, b_1), \ldots, (L_d, b_d))$ by

$$f(P_C) = \sum_{i=1}^{d} w(b_i) \cdot 2^{-i} \ ,$$

where $w(\mathsf{N}) = 1$ and $w(\mathsf{B}) = 0$. For our experiments, we periodically interrupted the prover and registered the current time, the current guiding path, and the number of nodes of the search tree processed so far. We use the last figure to determine the slowdown caused by additional lemma clauses, noting that the number of search nodes coincides with the number of unit propagations and literal selections performed, and thus can be used to compute the average time of these operations.

The results of our experiments are exemplarily shown for the satisfiable data set `DES-R3-B1-K1.1` in Figure 2. For this instance we generated and stored lemmas of length up to 10. A solution was found after 725.5 seconds, having processed approx. 16% of the search space. In the left diagram, the traversal pattern of the search space is displayed, on the right hand side the cumulated number of visited
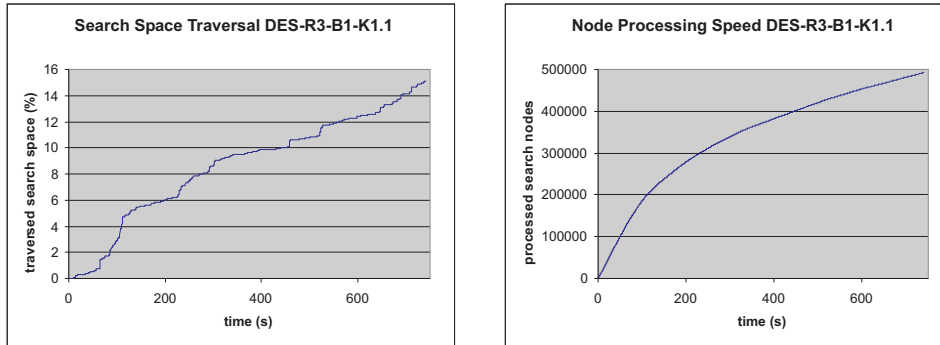
Fig. 2. Search Space Analysis for DES-R3-B1-K1.1 with Lemma Generation.

nodes of the search tree over time can be seen. Other SAT instances show similar behavior.

Our preliminary observations are as follows. For a particular problem instance the search space traversal graph is almost independent of lemma generation or exchange (up to scaling of the time and node axes due to speed-up). This suggests that the pattern of search space traversal is mainly controlled by the problem instance and the splitting variable heuristic. On the other hand, as can be seen from the right diagram, lemma insertion can cause a considerable slow-down of the search engine. We have not yet directly analyzed the effects of lemma exchange with our method, and expect that further work is needed to gain a deeper understanding of the effects of lemma generation and exchange in general.

4 Applications

We now turn to three areas we have chosen to evaluate our implementation: Consistency checking of automotive product documentation, logical cryptanalysis and hardware verification via bounded model-checking.

Consistency Checking of Product Documentation

Most of the manufacturing industry for complex products employs data bases to store information about which products they can produce. This is necessary as there are usually many sales options a customer can choose from, but not all combinations can actually be manufactured. Constraints on valid combinations are often expressed in a finite domain logic (or even Boolean logic), and consistency aspects of the data base as a whole play a vital role, as otherwise valid orders may be rejected or orders that cannot be manufactured may be accepted.

Using SAT-checkers to ascertain consistency generates loads of SAT instances, the greatest part of which is fortunately easily solvable [KS00]. As the number

of proofs for a complete check may well be in the thousands, parallel processing of these proof obligations is highly appreciated, e.g. to reduce the waiting time in an interactive environment. Our parallelization approach in this case is therefore twofold: First, we start several proofs in parallel. Should the proof time for an individual proof exceed a timing threshold value, the proof is suspended and put into a queue of long-running proofs, which are then, in a second phase, processed by our parallel prover PaSAT [BSK01].

Logical Cryptanalysis

Logical cryptanalysis deals with finding keys and analyzing properties of encryption techniques with logical methods. This means, e.g. for the former, that finding a key is equivalent to finding a model of a corresponding formula. Here, we consider analysis of parts of the DES standard, which was proposed as a hard SAT benchmark different from random instances [MM00]. In our experiments we analyzed a simplified version of DES using only 3 encryption rounds instead of 16 as in the standard.

Bounded Model-Checking

Symbolic model-checking is a widely accepted way for the verification of hardware and reactive systems. Bounded model-checking (BMC) [BCCZ99] was proposed as an alternative to the traditional BDD-based method for model-checking. Bounded model-checking uses propositional decision procedures instead of binary decision diagrams [Bry86] to check temporal/modal properties of these systems.

In the BMC approach in order to verify a property p of a logical hardware description H, a bounded number of steps of the transition relation of H is transformed into a propositional formula, which then, together with the propositional translation of p, is tested for satisfiability. The property p is usually given as a temporal logic formula.

For our experiments we checked the equivalence of a 16-bit sequential shift-and-add multiplier with a combinatorial multiplier (see [BCCZ99] for further information).

5 Experimental Results

We conducted experiments with PaSAT on all of the aforementioned application areas, and report in this paper on the last two of them. Results concerning the area of consistency checking for automotive product data can be found elsewhere

[BSK01].

Our experiments were performed on a Sun Ultra E450 with 4 UltraSparcII processors (@400 MHz) and 1 GB of main memory running under Solaris 7. Although DOTS supports distribution on multiple machines we did not make use of this feature due to the current implementation of clause exchange that relies on shared memory.

Logical Cryptanalysis

We used the data files from Massacci[1] that encode the DES algorithm using 3 rounds. The problem is to find the encryption key given a certain number of plaintext/ciphertext blocks. The number of these blocks varies between 1 and 4 (as indicated by the B-part of the problem instance name). All problem instances are satisfiable, but have only a few models, often just one.

As parameters for lemma generation we used $l_1 = 10$ and $l_2 = 100$, The literal selection strategy picked a literal of a shortest positive clause (SPC). Lemma exchange was limited to clauses of length ≤ 5.

Table 1
Performance Measurements of the 3-Round DES Instances.

problem instance	t_{SATO}	t_{Seq}	t_{Par}	t_{PaSAT}	$s_{\mathrm{Par/Seq}}$	$s_{\mathrm{PaSAT/Seq}}$
DES-R3-B1-K1.1	949.93	725.5	20.69	19.23	35.07	37.73
DES-R3-B1-K1.2	631.07	1566.35	112.14	51.14	13.97	30.63
DES-R3-B2-K1.1	1.3	3.84	2.13	1.86	1.80	2.06
DES-R3-B2-K1.2	287.04	2.96	3.07	2.58	0.96	1.15
DES-R3-B3-K1.1	42.75	31.29	4.3	4.22	7.28	7.41
DES-R3-B3-K1.2	194.31	1480.47	297.26	64.01	4.98	23.13
DES-R3-B4-K1.1	492.36	2.75	2.79	2.42	0.99	1.14
DES-R3-B4-K1.2	495.48	10.15	10.04	9.48	1.01	1.07

The results of our experiments are shown in Table 1. We have given the runtimes t_{Seq} for the sequential version (using only one processor), t_{Par} for the parallel version without lemma exchange and t_{PaSAT} for the parallel version with lemma exchange. For comparison reasons we also added the runtimes obtained with version 3.2 of SATO using the default settings. The last two columns indicate the speedups of the parallel versions relative to the sequential version, without resp. with exchange of lemmas. As the problem instances are all satisfiable the observation of

---

[1] `http://www.dis.uniroma1.it/~massacci/cryptoSAT`

213

superlinear speed-ups is not surprising. Adding lemma exchange further accelerates the model search considerably as can be seen, e.g., from DES-R3-B3-K1.2.

Bounded Model-Checking

The SAT instances we examined stem from a 16-bit multiplier[2]. The 16 datafiles of this benchmark set correspond to the 16 output bits of the multiplier. All instances are unsatisfiable. We set the parameters for lemma generation to $l_1 = 10, l_2 = 1000$ and exchanged lemmas up to a length of 5. The literal selection strategy was MBO (see Section 2).

Table 2
Performance Measurements of the Longmult Instances.

problem inst.	t_{SATO}	t_{Seq}	t_{Par}	t_{PaSAT}	$s_{\text{Par/Seq}}$	$s_{\text{PaSAT/Seq}}$
longmult6	42.42	20.68	5.73	4.85	3.61	4.26
longmult7	160.99	73.78	22.24	15.49	3.32	4.76
longmult8	390.25	176.91	51.54	42.76	3.43	4.14
longmult9	577.60	291.00	79.51	77.78	3.66	3.74
longmult10	1586.71	414.49	113.92	138.71	3.64	2.99
longmult11	1625.74	541.14	145.81	179.28	3.71	3.02
longmult12	728.09	646.43	163.66	162.78	3.95	3.97
longmult13	2198.44	809.67	202.92	227.93	3.99	3.55
longmult14	1294.64	929.21	242.13	248.59	3.84	3.74

The results of our experiments are shown in Table 2. The column headings have the same meaning as in the last section. Without lemma exchange we achieved speed-ups of up to 3.99, for the seven long-running examples we get an average efficiency of over 93%. In the presence of lemma exchange we get a higher deviation, and sometimes even observe superlinear speed-ups. These can be explained by the additional search space pruning effect of lemma exchange, which yields information that is otherwise usable only for a smaller part of the search.

6 Related and Future Work

Böhm and Speckenmeyer presented a parallel SAT-solver for a Transputer system consisting of 256 processors [BS96]. Their work concentrates on workload balanc-

---
[2] `http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html`

ing between the processors; the DP algorithm executed on each processor node employs a special variable selection heuristic, but no lemma generation or exchange.

PSATO [ZBHa96] is a distributed propositional prover for networks of workstations, based on the sequential prover SATO, which also incorporates lemma generation. PSATO is focused on solving open quasigroup problems. However, in PSATO no lemma exchange or communication between prover tasks is implemented, e.g. to interrupt processors working on unnecessary parts of the work space.

We see the main contribution of our paper in the integration of lemma exchange into a parallel version of the DP procedure. Moreover, we started to analyze the effects of lemma exchange, and expect that more work in this direction can reveal interesting facts about learning behavior.

The performance gains by lemma exchange are in many practical cases considerable, as our experiments have shown. Extending lemma exchange to more processor nodes by using agents distributed over a network seems to be a promising way for future developments.

References

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS. Springer-Verlag, 1999.

[BKLW99] Wolfgang Blochinger, Wolfgang Küchlin, Christoph Ludwig, and Andreas Weber. An object-oriented platform for distributed high-performance symbolic computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35(8), pages 677–691, Aug. 1986.

[BS96] M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. In *Annals of Mathematics and Artificial Intelligence*, volume 17(3-4), pages 381–400. Kluwer Academic Publishers, 1996.

[BSK01] W. Blochinger, C. Sinz, and W. Küchlin. Parallel consistency checking of automotive product data. 2001. Submitted to ParCo'01.

[HV95]   J. N. Hooker and V. Vinay. Branching rules for satisfiability. In *Journal of Automated Reasoning*, volume 15(3), pages 359–383, 1995.

[KS00]   W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1–2):145–163, February 2000.

[MM00]   F. Massacci and L. Marraro. Logical cryptanalysis as a sat problem. *Journal of Automated Reasoning*, 24(1-2):165–203, February 2000.

[SS96]   J. P. M. Silva and K. A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Nov. 1996.

[ZBHa96] H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[Zha00]   H. Zhang. Lemma generation in the Davis-Putnam method. In *Proceedings of the CADE-17 Workshop on Model Computation*, June 2000.

[ZS96]   H. Zhang and M. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, 1996.