W. Blochinger, C. Sinz, and W. Küchlin. A universal parallel sat checking kernel. In *To appear in Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2003)*, 2003.

A Universal Parallel SAT Checking Kernel

Wolfgang Blochinger
Carsten Sinz
Wolfgang Küchlin
Symbolic Computation Group, WSI
University of Tübingen, Sand 14
72076 Tübingen, Germany
http://www-sr.informatik.uni-tuebingen.de

Abstract

We present a novel approach to parallel Boolean satisfiability (SAT) checking. A distinctive feature of our parallel SAT checker is that it incorporates all essential heuristics employed by the state-of-the-art sequential SAT checking algorithm. This property makes our parallel SAT checker applicable in a wide range of different application domains. For its distributed execution a combination of the strict multithreading and the mobile agent programming model is employed. We give results of run-time measurements for problem instances taken from different application domains, indicating the usefulness of the presented method.

Keywords: *Parallel SAT Checking, Parallel Symbolic Computation, Multithreading, Mobile Agents.*

1 Introduction

The SAT problem asks whether or not one can find for a given Boolean formula a variable assignment such that the formula evaluates to TRUE. SAT is of particular relevance, since problems from a wide range of disciplines can be encoded as SAT instances. Prominent examples of application domains of SAT are hardware verification, cryptanalysis, planning, and scheduling. Consequently, SAT checkers (also called SAT provers) can be used as a universal tool for solving problems from manifold application domains.

Since SAT is NP-complete [8], for all currently known SAT checking algorithms there exist problem instances exhibiting exponential run-times in the number of boolean variables. However, sophisticated heuristics have been found that can dramatically reduce the computation time, especially for many problem classes of practical relevance. For these cases, parallel SAT checking is an important means to additionally reduce the run-time.

The classical *Davis-Putnam* (DP) SAT algorithm [10, 9] has been improved essentially by incorporating two classes of heuristics: sophisticated branching rules [11] and dynamic learning [12]. Both heuristics can significantly reduce the search space of variable assignments which has to be traversed for finding a solution or for proving that no solution exists. Often speedups of several orders of magnitude can result when applying an appropriate heuristics. But the actual effect of these heuristics depends largely on the problem instance. Thus an issue which still remains to be handled is to choose a proper heuristics for a given problem instance.

In previous work in parallel SAT checking we investigated the parallelization of SAT checkers that employ dynamic learning techniques [4, 5, 6]. In this paper we deal with the tight integration of both types of heuristics, branching rules and dynamic learning in oder to amplify the effects of both heuristics when applied in parallel.

The rest of the paper is organized as follows: Section 2 gives a brief introduction to modern SAT checking methods. Section 3 describes our universal parallel SAT checking kernel in detail. The results of run-time measurements are discussed in Section 4. Sections 5 and 6 give a comparison of related work and a conclusion, respectively.

2 Modern SAT Checking Techniques

In this section we give some basic definitions and a brief overview of the state-of-the-art sequential SAT algorithm in order to provide appropriate background information for understanding the subsequent presentation of our parallel SAT checker. For a detailed treatment of these topics the reader is referred to the literature.

2.1 Basic Definitions

We consider Boolean formulae in Conjunctive Normal Form (CNF) which are defined as conjunctions ($\wedge$) of *clauses*, where a *clause* is a disjunction ($\vee$) of *literals*, and a literal is a propositional *variable* or its negation. A clause containing exactly one literal is called a *unit clause*, the *empty clause* $\emptyset$ is a clause containing no literals at all. A solution to a SAT problem instance assigns to each variable a value (either TRUE or FALSE), such that in each clause at least one literal becomes TRUE, and thus all clauses are simultaneously satisfied. If one of the clauses of a formula is the empty clause it has no solution.

Since in a formula in CNF the logical connectives ($\vee$ and $\wedge$) are determined by its structure, they are often omitted. Clauses are then represented as sets of literals, and formulae as sets of clauses. For example, the Boolean formula

$$(x_2 \vee \overline{x_3}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge x_3,$$

which is in CNF, translates into the set of clauses

$$\{\{x_2, \overline{x_3}\}, \{x_1, x_3\}, \{\overline{x_1}, \overline{x_2}, \overline{x_3}\}, \{x_3\}\}.$$

2.2 Davis-Putnam Algorithm

Basically, by trying all possible variable assignments one after the other, one can find a solution to a given SAT problem, resp. prove its unsatisfiability. Evidently, the run-time of this naive procedure is exponential in the number of variables. The more sophisticated *Davis-Putnam* (DP) algorithm [10, 9] performs an improved search by a stepwise extension of partial variable assignments. At each step, the resulting subproblems are simplified by recursively applying two reduction operations called *unit subsumption* and *unit resolution*. If an empty clause has been derived, backtracking is performed, and if an empty clause set results, a solution is found (see Figure 1). This process can also be described as an unbalanced search tree, where the leaves either represent a *solution* (in each clause at least one literal becomes TRUE under the current variable assignment) or a *conflict* (under the current variable assignment all literals of at least one clause are FALSE).

As mentioned earlier the performance of this basic DP procedure can be enhanced by two types of heuristics: applying special branching rules and incorporating dynamic learning.

The branching rule is responsible for selecting a new literal at each step of the extension of the partial variable assignment. It turns out that this selection can greatly affect the performance of subsequently applied reduction operations and consequently affects the performance of the whole DP method. Substantial work has been carried out for finding beneficial branching rules [11] leading to a rich set of available rules to choose from. However, there is no universal branching rule which fits well for all possible problem instances. Our SAT checker provides the following six built-in rules (and also allows the easy integration of additional rules):

- FOV: Choose first open literal occurring in clause set.
- SPC: Choose a literal from the shortest positive clause.
- MO: Choose a literal with the maximal number of occurrences.
- MBO: Choose a literal with the maximal number of binary occurrences.
- SMB: First build the set of all literals occurring in positive clauses of minimal length, and choose a variable from this set with the highest number of binary occurrences in all clauses.
- SHM: First build the set of all positive literals occurring in non-Horn clauses of minimal length, and choose a variable from this set with the highest number of binary occurrences in all clauses.

Silva and Sakallah added to the DP algorithm a dynamic learning process carried out during the search [12]. At each conflict leaf of the search tree, a conflict analysis procedure produces additional knowledge about the problem instance in the form of a so called conflict induced clause (also called *lemma*) that may help to prune the search tree during the further search process. A conflict induced clause reflects a minimal variable assignment that causes the occurrence of the considered conflict. When added to the input clause set a conflict induced clause prevents the search process from reproducing this conflict in other regions of the search space and therefore can prune the search space to be treated. It can be shown that the addition of lemmas to the clause set does not affect the correctness of the DP algorithm. However the growing set of input clauses also causes a slowdown of the unit subsumption and unit resolution procedures which may outweigh the performance gain of search space pruning. Therefore it is common practice to limit additional lemmas to those containing less than a fixed number of literals.

3 The Parallel SAT Checking Kernel

We describe our parallel SAT checking kernel in a bottom up manner. A brief overview of the employed parallel middleware DOTS is followed by a discussion of details of the parallel execution process.

```
boolean DP(ClauseSet S, Level d )
{
  while ( S contains a unit clause {L} ) {
    delete clauses from S containing L;      // unit-subsumption
    delete ¬L from all clauses in S;         // unit-resolution
  }
  if ( ∅ ∈ S ) {                             // empty clause?
    generate conflict induced clause C_C;    // conflict management
    add C_C to S;
    return FALSE;                            // backtracking
  }
  if ( S = ∅ ) return TRUE;                  // empty clause set?
  choose a literal L_{d+1} occurring in S;   // apply branching rule
  if ( DP(S ∪ {{L_{d+1}}}), d+1 )
  return TRUE;                               // first branch
  else if ( DP(S ∪ {{¬L_{d+1}}}), d+1)
  return TRUE;                               // second branch
  else return FALSE;
}
```

Figure 1. Sequential Davis-Putnam Algorithm

3.1 DOTS Middleware

DOTS [3] is a system environment for building and executing distributed parallel C++ applications which is able to integrate a wide range of different computing platforms into a homogeneous parallel environment. Up to now, DOTS has been deployed on (heterogeneous) clusters composed of the following platforms: Microsoft Windows 98/NT/2000/XP, SUN Solaris, SGI IRIX, IBM AIX, FreeBSD, Linux, QNX Realtime Platform, and IBM Parallel Sysplex Cluster [2].

3.1.1 Parallel Programming Models

DOTS provides several parallel programming models (each represented by its own API) forming a comprehensive set of tools for realizing distributed parallel applications.

- **Task API**
  The Task API represents the basic API layer of DOTS on which all other APIs are based. DOTS task objects are instances of application specific classes which are derived from the base class DOTS_Task and implement a run() method. The code provided in the run() method is executed on its own thread when the task object is scheduled for execution. The base class also provides methods for explicit program controlled migration of DOTS tasks.

- **Active Message API**
  The Active Message API provides support for object-oriented message passing. After a message object is transferred to its destination node it becomes an active object, i.e. a new thread is created that executes application specific code contained in the message object.

- **Multithreading API**
  Multithreaded computations are generalizations of asynchronous (remote) procedure calls. The DOTS Multithreading API provides a compact set of completely orthogonal primitives for realizing the comprehensive class of strict multithreaded computations. In such computations data-dependencies of a thread can go to any ancestor in the spawn tree. The DOTS multithreading programming model is enhanced with object-oriented features and support for highly-irregular non-deterministic computations.

- **Autonomous Task API**
  The Autonomous Tasks API can be used to realize task objects that operate as mobile agents. In contrast to standard task objects, the execution of an autonomous task is not determined by the load distribution mechanism of DOTS. Instead, its execution locations can be explicitly determined by the programmer. For facilitating the control of autonomous tasks, the API provides higher level migration primitives, e.g. for realizing round trips of mobile agents within the distributed environment.

In particular, the Multithreading and Autonomous Tasks parallel programming models are used for implementing our parallel SAT checking kernel.

3.1.2 System Architecture

The DOTS system architecture is organized in several layers. The lowest layer is the OS adaptation layer which is realized using the ACE (ADAPTIVE Communication Environment) toolkit [1]. It homogenizes system APIs for a wide range of different platforms. On top of the OS

adaptation layer, the DOTS run-time system is built. It provides several low-level services like object-serialization, message transmission, data encoding/decoding, TCP connection caching, and support for an internal component-architecture. Additionally, higher level services like object migration, node directory, logging, and a load monitoring framework are provided by the DOTS run-time. All services are used to implement the DOTS Task API and consequently all other APIs.

3.2 Hierarchically Parallel SAT checking

3.2.1 Overview

Our approach to build a generic parallel SAT checking kernel combines different forms of logical parallelism on two hierarchical layers. On the top level, competition parallelism is employed by concurrently executing several incarnations of our high-performance SAT-checker PaSAT [13] on the input problem, each applying a different branching rule heuristics. On the second level, each PaSAT incarnation establishes a parallel treatment of the problem instance.

The top-level approach of running several variable selection strategies concurrently provides robustness and stability under different applications, while the lower level parallelization in PaSAT provides speed of the individual SAT incarnations and consequently a reduction of the overall time needed to treat a problem instance.

The challenge of our work lies in combining this hierarchical parallel approach with the sequential lemma generation heuristic. Note that optimized sequential search algorithms are often hard to parallelize because they tend to accumulate knowledge in search state which they then use to take short-cuts. In the case of modern SAT-checking, knowledge is accumulated in the form of conflict avoidance lemmas which prevent repeated fruitless searches of subspaces.

For the distributed execution of PaSAT, we solved the problem of combining lemma accumulation with parallel search by an approach which lies orthogonal to parallelization of the backtracking search. Lemmas found by the concurrent search processes are scooped up by mobile agents and transported across the nodes of the distributed system [4]. We now take this approach one level higher and also use agents to transport lemmas discovered by any one of the PaSAT incarnations to all other PaSAT siblings. (Note that each lemma is always a valid consequence of the given problem instance, regardless under which branching rule it has been deduced.) This cross-fertilization between incarnations with different branching rules causes an additional reduction of the run-time of each prover instance and consequently a reduction of the overall time needed to treat a problem instance.

3.2.2 Implementation using DOTS

The realization of the top level competition parallelism is trivial. For each available branching rule a DOTS thread is forked which executes the corresponding prover instance on the given input formula.

Basically, on the second level the parallelization of a combinatorial search problem is carried out. We accomplish this task by employing the DOTS Multithreading API. Since in the considered case it is generally not possible to statically generate subproblems of equal size a dynamic problem decomposition process is required. As stated above, the execution of a PaSAT incarnation starts with one thread which has the complete search space assigned. During the whole computation all DOTS threads periodically monitor the length of the local task queue. If the length falls below a predefined limit, a new DOTS thread is forked. The parent thread splits off a region of its search space and assigns it to the new DOTS thread. Details of the applied search space splitting heuristics can be found in [14]. The newly created DOTS thread is queued and can be executed by another local processor or can be transferred to other nodes. A thread stealing strategy with randomized victim selection is employed for load distribution. Since DOTS supports the comprehensive class of strict multithreaded computations the results of each thread generated can be joined by the initial thread and need not be joined by its actual parent. This ensures scalability and also considerably simplifies the program code.

The huge amount of new knowledge generated on each node (at every conflict leaf in the search tree a new lemma is deduced) makes it impossible to exchange all generated lemmas while at the same time preserving scalability of the parallel algorithm. Therefore a strategy for selecting suitable lemmas is required. To establish a lemma exchange between PaSAT incarnations, for each PaSAT incarnation a mobile agent is created (employing the DOTS Autonomous Task API) that visits all PaSAT siblings and looks for new lemmas that fulfill some criteria. As selection criteria the lengths of the lemmas and the requirement that the considered lemma is not already subsumed ”at home” is applied. (A lemma is subsumed when it is already logically implied in the current state of the search of the agent’s associated SAT checker instance, its ”home”.) In order to perform this test, the lemma exchange agent carries with it a description of which part the home SAT checker is currently working on, and this description is refreshed every time the agent visits its home node to unload its collected lemmas.

4 Experimental Results

The parallel system employed for carrying out performance measurements consisted of a cluster of 3 Sun Ul-

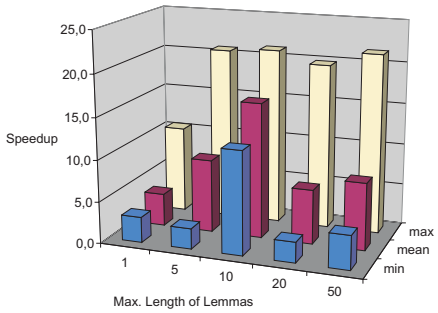

Figure 2. Speedups for Benchmark bw_large.d

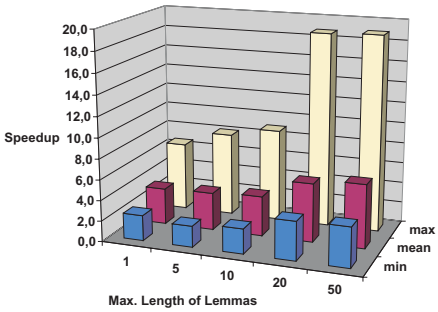

Figure 3. Speedups for Benchmark longmult15

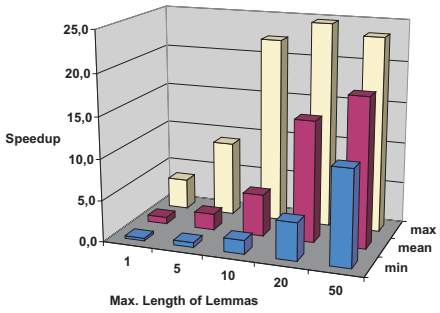

Figure 4. Speedups for Benchmark qg6-14

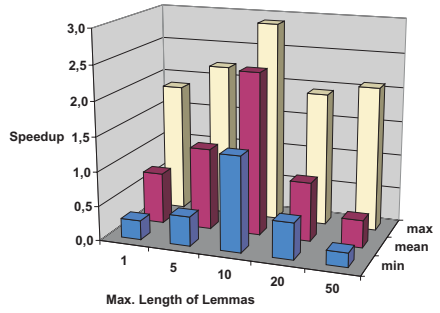

Figure 5. Speedups for Benchmark facts7hh.12

tra E450, each with 4 UltraSparcII processors (@400 MHz) and 2 GB of main memory, connected by a 100 Mbps Etherne network.

Four benchmarks from different application domains have been chosen:

- Planning (bw_large.d)
- Hardware Verification (longmult15)
- Quasigroup Existence (qg6-14)
- Logistics (facts7hh.12)

All 6 built-in heuristics (see Section 2) have been employed. Since a total number of 12 processors was available, each PaSAT incarnation could be executed on 2 processors in parallel. The results of the measurements are given in Figures 2,3,4, and 5. The speedup values are based on the run-time of the sequential algorithm using the best performing branching rule heuristics for each individual benchmark. For all parameter settings 10 parallel runs have been carried out. Besides the arithmetic mean, additionally the minimum and maximum speedup values are given, since the individual run-times exhibited a high dispersion. We investigated different maximum lengths of lemmas to be collected by the lemma agents. It turned out, that for all considered problem instances considerable speedups could be obtained using lemma exchange agents starting with maximum lemma sizes of 5 to 10.

Moreover, the results show two types of phenomena: high variabilities in the run-times of executions with the same input and settings as well as superlinear speedups. Both can be traced back to the parallel dynamic learning process. The exchange of lemmas in a distributed system is highly non-deterministic causing a highly variable distribution of new knowledge in individual runs. Thus, important lemmas may not yet be available when treating a particular region of the search space. Superlinear speedups occur because of cross-fertilization effects of the two types of heuristics which are as such not possible in the sequential algorithm. It is possible, that a lemma is generated using branching rule A which can never be deduced when applying branching rule B. Nevertheless, this lemma can be very effective for reducing the search space in a search process using branching rule B.

5 Related Work

Böhm and Speckenmeyer presented a parallel SAT checker on a Transputer [7]. Their work concentrates on workload balancing between the processors; the DP algorithm executed on each node employs the same branching rule heuristics, and no lemma generation or exchange is carried out.

PSATO [14] is a distributed propositional prover for networks of workstations, based on the sequential prover SATO, which also uses dynamic learning. PSATO is focused on solving open quasigroup problems. However, in PSATO no lemma exchange or communication between tasks is implemented. Also a dedicated master performs search space splitting. This approach involves additional communication and the master can easily become a sequential bottleneck. Moreover, sophisticated load distribution schemes cannot be realized.

6 Conclusion

In this paper we presented a novel method for parallel SAT checking which incorporates competition parallelism, parallel combinatorial search with dynamic problem decomposition and a distributed dynamic learning process accomplished by mobile agents. This approach permits to beneficially transfer all key heuristics employed by the state-of-the-art sequential SAT checking algorithm to its parallel counterpart leading to a cross-fertilization of heuristics.

References

[1] The ADAPTIVE Communication Environment (ACE). *http://www.cs.wustl.edu/~schmidt/ACE.html*.

[2] W. Blochinger. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (IPDPS 2001)*, page 90, San Francisco, CA, U.S.A., April 2001. IEEE Computer Society Press.

[3] W. Blochinger, W. Küchlin, C. Ludwig, and A. Weber. An object-oriented platform for distributed high-performance Symbolic Computation. *Mathematics and Computers in Simulation*, 49:161–178, 1999.

[4] W. Blochinger, C. Sinz, and W. Küchlin. Distributed parallel SAT checking with dynamic learning using DOTS. In T. Gonzales, editor, *Proc. of the IASTED Intl. Conference Parallel and Distributed Computing and Systems (PDCS 2001)*, pages 396–401, Anaheim, CA, August 2001. ACTA Press.

[5] W. Blochinger, C. Sinz, and W. Küchlin. Parallel consistency checking of automotive product data. In G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *Proc. of the Intl. Conf. ParCo 2001: Parallel Computing – Advances and Current Issues*, pages 50–57, Naples, Italy, 2002. Imperial College Press.

[6] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 2003. To appear.

[7] M. Boehm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.

[8] S. A. Cook. The complexity of theorem proving procedures. In *3rd Symp. on Theory of Computing*, pages 151–158. ACM press, 1971.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

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

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

[13] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[14] 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.