Distributed Parallel SAT Checking with Dynamic Learning using DOTS

Wolfgang Blochinger Carsten Sinz and Wolfgang Küchlin
Symbolic Computation Group, WSI
University of Tuebingen
Sand 13
D-72076 Tuebingen, Germany
email: {blochinger,sinz,kuechlin}@informatik.uni-tuebingen.de

ABSTRACT
We present a novel method for distributed parallel automatic theorem proving. Our approach uses a dynamically learning parallel SAT checker incorporating distributed multi-threading and mobile agents. Individual threads process dynamically created subproblems, while agents collect and distribute new knowledge created by the learning process. As parallelization platform we use the Distributed Object-Oriented Threads System (DOTS) that provides support for both distributed threads and mobile agents. We present experiments indicating the usefulness of the presented approach for different application domains.

KEY WORDS
Parallel/Distributed Algorithms, Parallel/Distributed Computing Systems, Performance Evaluation and Measurements, Parallel SAT Checking

1. Introduction

Satisfiability checking for Boolean formulae (SAT checking) has many important applications, e.g. hardware verification [1], cryptanalysis [2] or checking formal assembly conditions of motor cars [3]. Although SAT is an NP-complete problem and therefore problem instances with exponential run-times can occur in the worst case, advanced algorithms along with sophisticated heuristics can dramatically reduce the computation time for many problem classes of practical relevance. For these cases, parallel SAT checking is an important means to additionally reduce computation time. Recently, an extension of the well-known Davis Putnam SAT checking algorithm by dynamic learning techniques has been proposed in the literature [4]. In this paper we present a SAT checking application that incorporates these new dynamic learning techniques with distributed parallel execution. As parallelization platform the Distributed Object-Oriented Threads System (DOTS) has been used. One major design goal of DOTS is to provide efficient support for the parallelization of state-of-the-art SAT checking techniques.

2. DOTS

DOTS (Distributed Object-Oriented Threads System) 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. Although DOTS was originally designed as a software platform for the parallelization of algorithms from the realm of Symbolic Computation [5], it is also used in other application domains like parallel computer graphics [6]. One of the major design goals of DOTS is to achieve a high degree of overall usability. Using DOTS, tasks like application programming as well as testing, evaluation and documentation of results can be carried out easily [7]. This makes DOTS ready-to-use in existing parallel environments, for example pools of workstations. DOTS supports a wide range of hardware and software platforms. Currently, DOTS can be employed on the following platforms: Windows 98/NT/2000, Solaris (Sparc/Intel), IRIX, AIX, FreeBSD, Linux, QNX Realtime Platform and IBM Parallel Sysplex Clusters (clusters composed of IBM mainframes running under OS /390) [8].

2.1 DOTS APIs

As shown in Figure 1 the programmer can use different APIs to develop distributed parallel applications with DOTS.

2.1.1 Task API

The Task API represents the low level API layer of DOTS on which also the other APIs are based. It provides support for so called DOTS Task objects. These are objects that are derived from the base class DOTS_Task and implement a run() method. After having registered a task object with DOTS, the code provided in the run() method is executed on its own thread when the task object is scheduled for execution (see Section 2.2). The base class DOTS_Task provides also methods for program controlled task migration in the case of distributed execution of a DOTS application. When called within the run() method of a task object, the object

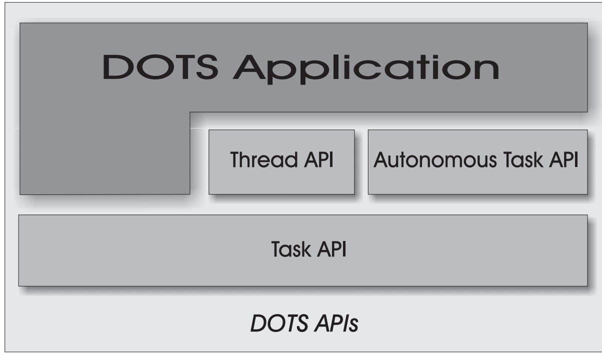

Figure 1. DOTS Application Programming Interface

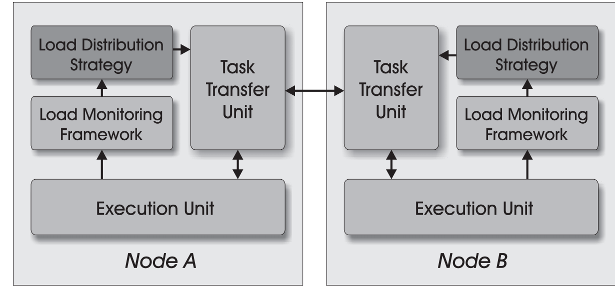

Figure 2. DOTS Architecture

is transferred to the given node along with its current state. On the destination node the object state is reconstituted and the run() method is restarted.

2.1.2 Thread API

The Thread API is the main API of DOTS. 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. The DOTS API provides primitives for DOTS thread creation (dots_fork), synchronization with the results computed by other DOTS threads (dots_join) and DOTS thread cancellation (dots_cancel). All primitives can also be used in conjunction with so called *thread groups*. Thread groups are a means of representing related DOTS threads. When applied with thread groups, the semantics of each primitive is automatically changed to the appropriate group semantics. E.g. when using the join primitive with a thread group, *join-any* semantics will be applied. The Thread API is implemented on top of the Task API by defining a special class of task objects called DOTS_Thread, that contain argument and result objects and additional information like the procedure to be forked.

2.1.3 Autonomous Tasks API

The Autonomous Tasks API can be used to realize task objects that operate as autonomous agents. In contrast to normal task objects, they are not affected by the load distribution mechanism of DOTS. Instead, the execution location can be explicitly chosen by the programmer. Therefore, the API contains primitives for controlling and facilitating the migration process of the autonomous tasks, for example for realizing round trips of agents within the parallel environment.

2.2 DOTS Architecture

Figure 2 shows the main functional units of the architecture of DOTS.

DOTS tasks are executed within the *Execution Unit* (Figure 3). They can be executed in immediate mode or in queued mode. In the former case, an OS native thread is created that executes the run() method of the task object. DOTS tasks that are intended for queued execution are placed into a *task queue*. A pool of (pre-forked, OS native) worker threads dequeue task objects from the queue and execute the corresponding run() method. The number of worker threads can be determined by the programmer. Normally, for each node the number of available processors is chosen. After the execution of a DOTS task object is completed, it is placed into a so called *ready queue*.

To support the execution of DOTS tasks in a distributed environment, the DOTS architecture includes additional components. The *Task Transfer Unit* transfers (serialized) task objects between queues of execution units residing on different nodes. Task Transfer is needed for task migration or load distribution. *The Load Monitoring Framework* traces all events concerning the execution of DOTS tasks and provides status information like the current load or the current length of the task queue. Based on the Load Monitoring Framework, different load distribution strategies can be implemented. A load distribution strategy is responsible for triggering the transfer of task objects and selecting destination nodes according to a particular strategy. Using the Load Monitoring Framework, custom load distribution schemes can easily be realized and integrated into DOTS.

3. Parallel SAT checking

The SAT problem asks whether or not a Boolean formula has a model, or, alternatively, whether or not a set C of Boolean constraints has a solution. Usually the constraints are kept in conjunctive normal form (CNF). Each constraint is then also called a clause and consists of a set of literals, where a literal is a variable or its negation. A clause containing exactly one literal is called a unit clause. A solution

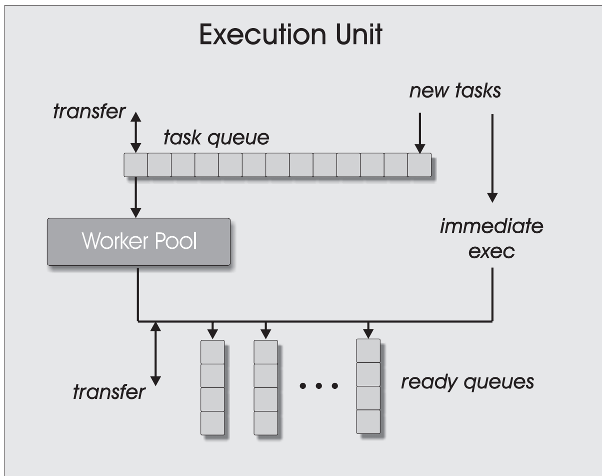

Figure 3. DOTS Execution Unit

assigns to each variable a value (either true or false), such that in each clause at least one literal becomes true.

3.1 Davis Putnam Algorithm with Dynamic Learning

Basically, by trying all possible variable assignments one after the other, one finally finds a solution to a given SAT problem, provided that one exists. Evidently, the run-time of this naive procedure is exponential in the number of variables. However, for many problem classes the run-time can be dramatically reduced by applying the *Davis-Putnam* (DP) algorithm [9]. It performs an optimized search by extending partial variable assignments, and by simplifying the resulting subproblems by recursively applying an operation called *unit propagation*. This process can be described as a search in 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).

Silva and Sakallah propose a so called conflict analysis procedure carried out during the search process [4]. At each conflict leaf the conflict analysis produces additional knowledge about the problem instance in the form of so called conflict induced clauses (also called lemmas) 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 occurance 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 prunes the search space. This process is referred to as dynamic learning. It can be shown that the addition of conflict induced clauses to the clause set does not affect the correctness of the Davis Putnam algorithm. However the growing set of input clauses causes also a slowdown of the unit propagation procedure which may outweigh the performance gain of search space pruning. Therefore it is common practice to limit the adding of lemmas to those containing less than a fixed number of literals.

3.2 Parallelization

The distributed parallel execution of the presented SAT checker with dynamic learning is carried out at two logical layers. On the first layer workload is distributed among the nodes using the DOTS Thread API. While on the second layer knowledge represented by lemmas is exchanged among the nodes by autonomous tasks. The following subsections describe both layers in detail.

3.2.1 Workload Distribution with Threads

Basically, we are dealing with the parallelization of a combinatorial search problem. This implies that the search space has to be divided into mutually disjoint portions to be treated in parallel. However, a (static) generation of balanced subproblems is not feasible, since it is virtually impossible to predict the extent of the problem reduction delivered by unit propagation in advance.

Instead, a dynamic search space splitting approach is carried out. To start the execution the main thread forks one DOTS thread that has the entire search space assigned. During the whole computation all created DOTS threads periodically monitor the length of the local task queue (see Section 2.2). If the task queue is empty, 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 [10]. To prevent the uncontrolled splitting-off of very small fractions of the search space, a predefined time interval has to be waited before the next split can be carried out by the DOTS thread. The newly created DOTS task object is queued and can be executed by another local worker thread or can be transferred to other nodes.

The described splitting procedure generates subproblems on demand. This ensures that new subproblems are generated on the one hand during the initialization phase of the computation to exploit the available processing capacity and on the other hand every time a subproblem has been completely processed without finding a solution.

After forking the initial DOTS thread, the main thread immediately calls dots_join to wait for the created DOTS threads. All DOTS threads (except the initial one) are created with the dots_subfork primitive. This means that they can be joined by the main thread (and are not joined by their actual parent threads). The result of a DOTS thread indicates whether a problem solution was found within its assigned search region or not. The processing is completed either if all created DOTS threads have been joined without

returning a solution, or when the first DOTS thread that has found a solution is joined. In the latter case, all remaining DOTS threads are immediately canceled.

We use task stealing with randomized victim selection as load distribution scheme. It has been shown that applying a randomized work-stealing strategy to distribute the load in backtrack search algorithms is likely to yield a speedup within a constant factor from optimal (when all solution are required) [11].

3.2.2 Knowledge Exchange using Autonomous Tasks

For each available processor on a node a *clause store* object is created that holds the set of clauses for a SAT checker instance. The clause set consists of initial input clauses as well as lemmas generated by the associated SAT checker. Between clause store objects residing on the same node, lemmas can easily be exchanged through shared memory. Lemmas from clause stores on other nodes are exchanged by employing DOTS autonomous tasks. Therefore for each clause store object a DOTS autonomous task object is created that acts as an agent for gathering lemmas on other nodes. It constantly visits all nodes in a round robin fashion looking for new lemmas. Every time it is back on its home node it inserts the gathered lemmas into the local clause store.

Because of the huge amount of generated lemmas it is impossible to exchange all generated lemmas. Therefore, agents gather only lemmas that fulfill some criteria. Currently we use as selection criteria the length of the lemmas and the requirement that the considered lemma is not already subsumed, i.e. does not contain information that is obviously irrelevant in the part of the search space assigned to the agent's associated SAT checker task. The description of which part the SAT checker task is currently working on is transferred to the lemma exchange agent every time the agent visits its home node.

4. Experimental Results

The parallel environment used for the presented performance measurements consisted of a cluster made up of the following components (all nodes were connected with 100 Mbps switched Fast-Ethernet).

- 2 Sun Ultra E450, each with 4 UltraSparcII processors (@400 MHz) and 1 GB of main memory, running under Solaris 7.
- 1 PC, with 2 PentiumIII processor (@500MHz) and 256 MB of main memory, running under Solaris7.
- 4 PCs, each with 1 PentiumII processor (@400MHz) and 128 MB of main memory, running under Solaris7.

To measure the effects of lemma exchange we used benchmarks of both theoretical and practical importance.

QG7-12 encodes one of the quasigroup existence problems of kind QG7 given by Fujita *et al.* [12]. Quasigroup existence problems are, according to Zhang *et al.* [10], much better benchmark problems for SAT checkers than the quite common randomly generated instances, as they have fixed solutions, simple descriptions, are easy to communicate, and, most interestingly, some cases are still open.

DES stems from the area of logical cryptanalysis [2], and encodes the problem of finding an encryption key given three plaintext/ciphertext blocks that were produced using three rounds of the DES algorithm.

Our last benchmark problem, LONGMULT, is taken from the realm of hardware verification using bounded model checking [1]. It represents the hardest problem out of 16 formulae expressing the equivalence of two different 16-bit multiplier hardware designs.

The results of our experiments are shown in Tables 1,2 and 3. The given sequential runtimes represent the weighted means of sequential program runs on each of the different processors. The parallel runtimes represent wall-clock times of the parallel execution on the aforementioned environment. Since the parallel execution of the SAT checker can exhibit a non-deterministic behavior we show runtimes of ten individual measurements for each benchmark instead of averaged times. In column "leaves" the number of leaves of the processed search tree is shown.

Discussion of Results

For most quasigroup problems the effect of lemma generation is only marginal, which can also be seen (from Table 1) in the similar runtimes of the parallel versions with and without lemma exchange. Speedups compared to the sequential version are, however, quite good, reaching an efficiency of 75.7%.

As there is (at least one) key matching the plaintext/ciphertext blocks, the DES-problems are all satisfiable. This also explains the superlinear speedups that we could observe during our experiments (Table 2), as there is a higher probability that a prover process starts its search near a solution. Lemma exchange even amplifies this effect.

Although the benchmark formula LONGMULT is unsatisfiable, we nevertheless observed superlinear speedups (see Table 3). 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. ("Knowledge from the future can be used".)

5. Related Work

Selman *et al.* proposed a different method for solving real-world propositional problems based on local search [13]. Kautz and Selman applied it successfully e.g. to planning

time_seq (mean): 1504.4 sec					
no lemma exchange			lemma exchange		
time (sec)	speedup	leaves	time (sec)	speedup	leaves
138	10.9	247,325	143	10.5	193,488
145	10.4	247,526	139	10.8	198,639
141	10.7	254,319	135	11.1	196,322
137	11.0	248,393	143	10.5	196,360
145	10.4	256,027	141	10.7	202,314
138	10.9	243,492	148	10.2	203,966
147	10.2	260,576	140	10.7	195,374
142	10.6	254,730	139	10.8	198,975
145	10.4	254,417	139	10.8	198,693
143	10.5	258,499	145	10.4	207,590

Table 1. QG7-12

time_seq (mean): 1595.0 sec					
no lemma exchange			lemma exchange		
time (sec)	speedup	leaves	time (sec)	speedup	leaves
312	5.1	1,217,581	56	28.5	119,377
308	5.2	1,287,493	63	25.3	132,456
312	5.1	1,250,425	64	24.9	122,319
310	5.1	1,074,611	61	26.1	140,352
312	5.1	1,109,904	74	21.6	187,450
314	5.1	1,119,698	94	17.0	264,180
308	5.2	1,189,128	89	17.9	244,282
312	5.1	1,194,960	87	18.3	227,919
312	5.1	1,327,524	89	17.9	285,094
309	5.2	1,000,395	103	15.5	298,306

Table 2. DES

time_seq (mean): 2212.0 sec					
no lemma exchange			lemma exchange		
time (sec)	speedup	leaves	time (sec)	speedup	leaves
149	14.8	376,184	109	20.3	252,183
154	14.4	405,902	98	22.6	228,835
174	12.7	434,409	101	21.9	236,340
160	13.8	404,967	99	22.3	231,383
145	15.3	365,113	114	19.4	264,961
224	9.9	583,491	97	22.8	222,615
145	15.3	362,368	91	24.3	209,383
189	11.7	462,577	87	25.4	200,894
146	15.2	356,542	111	19.9	238,787
149	14.8	373,029	101	21.9	235,955

Table 3. LONGMULT

problems [14]. Although their algorithm in many cases outperforms the DP method, it suffers from the drawback that it is incomplete, and therefore cannot prove unsatisfiability.

Böhm and Speckenmeyer presented a parallel SAT-solver for a Transputer system [15]. Their work concentrates on workload balancing between the processors; the DP algorithm executed on each processor node employs a special variable selection heuristic, but no lemma generation or exchange.

PSATO [10] 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 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.

References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] Michael Meißner, Tobias Hüttner, Wolfgang Blochinger, and Andreas Weber. Parallel direct volume rendering on PC networks. In H. R. Arabnia, editor, *Proceedings of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)*, Las Vegas, NV, U.S.A., July 1998. CSREA Press.

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

[8] Wolfgang Blochinger, Reinhard Bündgen, and Andreas Heinemann. Dependable high performance computing on a parallel sysplex cluster. In H. R. Arabnia, editor, *Proceedings of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, volume 3, pages 1627–1633, Las Vegas, NV, U.S.A., June 2000. CSREA Press.

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

[10] 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.

[11] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[12] M. Fujita, J. Slaney, and F. Bennett. Automatic generation of some results in finite algebra. In *Proc. International Joint Conference on Artificial Intelligence IJCAI*, pages 52–59, Chambéry, France, 1993. Morgan Kaufmann.

[13] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[14] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.

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