Parallel propositional satisfiability checking with distributed dynamic learning

Wolfgang Blochinger, Carsten Sinz and Wolfgang Küchlin

(Accepted Peer-Reviewed Manuscript Version)

BibTeX

```
@article{Blochinger2003,
       author = "Wolfgang Blochinger and Carsten Sinz and Wolfgang Kuechlin",
       title = "Parallel propositional satisfiability checking with distributed
                dynamic learning",
       journal = "Parallel Computing",
       volume = "29",
       number = "7",
       pages = "969--994",
       year = "2003",
       issn = "0167-8191",
       doi = "https://doi.org/10.1016/S0167-8191(03)00068-1",
       url = "http://www.sciencedirect.com/science/article/pii/S0167819103000681"
}
```

Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning

Wolfgang Blochinger Carsten Sinz Wolfgang Küchlin

Symbolic Computation Group, WSI
University of Tübingen, 72076 Tübingen, Germany
http://www-sr.informatik.uni-tuebingen.de

Abstract

We address the parallelization and distributed execution of an algorithm from the area of symbolic computation: propositional satisfiability (SAT) checking with dynamic learning. Our parallel programming models are strict multithreading for the core SAT checking procedure, complemented by mobile agents realizing a distributed dynamic learning process. Individual threads treat dynamically created subproblems, while mobile agents collect and distribute pertinent knowledge obtained during the learning process. The parallel algorithm runs on top of our parallel system platform DOTS (Distributed Object-Oriented Threads System), which provides support for our parallel programming models in highly heterogeneous distributed systems. We present performance measurements evaluating the performance gains by our approach in different application domains with practical significance.

Key words: parallel symbolic computation, parallel propositional satisfiability checking, distributed multithreading

1 Introduction

This paper deals with the parallelization of a novel propositional satisfiability (SAT) checking algorithm with dynamic learning. The SAT problem asks whether one can find for a given Boolean formula a variable assignment such that the formula evaluates to true. Besides its theoretical importance, also many problems with practical relevance from a wide range of disciplines, including hardware verification, cryptanalysis, or planning and scheduling, can be encoded as SAT instances and efficiently solved by SAT checkers.

SAT has been the first problem proven to be NP-complete [12]. Consequently, for all currently known SAT algorithms there exist problem instances with exponential run-times. However, advanced methods along with sophisticated heuristics can dramatically reduce the computation time for many problem classes of practical relevance. In these cases, parallel SAT checking is an important means to reduce the computation time even further.

The classical Davis-Putnam-Logemann-Loveland SAT procedure [14,13] was introduced in the early 1960s, and parallel versions of this algorithm have been developed by Zhang *et al.* [36] and by Boehm and Speckenmeyer [8] in 1996, both using similar techniques. In this paper we address the parallelization of the state-of-the-art algorithm introduced by Marques-Silva and Sakallah [24] which enhances the Davis-Putnam-Logemann-Loveland method with dynamic learning techniques based on conflict analysis and lemma generation. The dynamic learning process can further dramatically reduce the run-time for a number of important problem classes. (Table 1 in Section 5 shows examples of performance improvements for the sequential algorithm which can be obtained using the dynamic learning technique.) Due to this significant improvement of the sequential algorithm it is crucial that the parallel variant also incorporates a dynamic learning process.

Generally, the parallelization of algorithms from the field of symbolic computation, like SAT checking, has not been as extensively investigated as the parallelization of numerical algorithms. One reason for this might be that symbolic algorithms tend to be more unstable in several respects. First, symbolic algorithms are typically very data dependent and therefore highly irregular in their course of action; as a consequence, static parallelization or static load balancing are not feasible. Second, theoretical enhancements of the sequential algorithms often lead to dramatic performance gains. It is therefore crucial to base the parallel application on the best known sequential algorithm, but optimized algorithms frequently accumulate knowledge in complex data-structures or state information which must now be distributed to the parallel tasks, increasing their synchronization overhead. Nevertheless, parallelization can also be very beneficial, because in the field of symbolic computation algorithm complexities are very high and there is little other hardware support.

The remainder of the paper is organized as follows. In Section 2 a brief introduction to the SAT problem is given and the state-of-the-art sequential SAT checking algorithm is explained. Section 3 discusses suitable parallel programming models for its parallelization and presents our parallel approach. Section 4 gives a brief description of our parallel system platform DOTS and focuses on details of the implementation of our parallel algorithm using DOTS. The results of performance measurements are reported in Section 5. Section 6 discusses related work, and Section 7 contains a conclusion.

2 Introduction to SAT Checking

2.1 Problem Description

The SAT problem asks whether or not a Boolean formula has a model. We may assume w.l.o.g. that the formula is in conjunctive normal form (CNF), i.e., it is a conjunction of *clauses*, where a *clause* is a disjunction 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. Thus, a set of clauses containing the empty clause is *inconsistent*, because it never has a solution.

Since in a formula in CNF the logical connectives (disjunction $\vee$ and conjunction $\wedge$) 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 logic 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

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

For this formula, resp. clause set, the function assigning TRUE to x_2 and x_3, and FALSE to x_1, is the only model resp. solution.

2.2 The DP Algorithm with Dynamic Learning

Basically, by trying out all possible variable assignments one after the other, one finally finds a solution to a given SAT-instance, provided that such a solution exists. The *Davis-Putnam-Logemann-Loveland* algorithm [14,13] (also commonly known as the *DP algorithm*) performs an optimized search by extending partial variable assignments, and by simplifying the resulting subproblems by applying two constraint propagation operations known as *unit subsumption* and *unit resolution*. In 1996, Marques-Silva and Sakallah proposed an extension of the classical DP algorithm by dynamic learning techniques based on conflict analysis and—as a by-product—non-chronological backtracking [24]. This enhancement often leads to considerable improvements, especially on structured real-world SAT instances. It has now become a quasi-standard, and is implemented in most of today's SAT checkers [27,35].

```
boolean DP(ClauseSet S, Level d )
{
  while (S contains a unit clause {L}) {
    register cause of L becoming unit        // conflict management
    delete clauses containing L from S       // 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
  }
  if (S = ∅) return TRUE                     // no clauses?
  choose a literal L_{d+1} occurring in S    // case-splitting on L_{d+1}
  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
}
```

Fig. 1. The Sequential DP Algorithm with Dynamic Learning

Since our parallel SAT checker also employs these new techniques, this section presents the basic concepts of the DP algorithm with dynamic learning. The algorithm is shown in Figure 1, where the first call to DP is made with the initial problem description S and a starting level of $d = 0$.

In the following, we associate with each run of DP a search tree, which is a finite binary tree generated by the recursive calls of the case splitting step. The nodes of the tree represent execution states of DP with a fixed input clause set S. We will label the outgoing edges of each node with the literal L, resp. $\overline{L}$, which is conceptually added to S to generate the new subproblem. Figure 2 depicts such a search tree.

Dynamic learning aims at reducing the search space by adding information to the problem instance's clause set which is derived during the search. This works as follows: As soon as DP 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 [24]. Often, not all selected splitting literals L_d are a necessary condition for the conflict to emerge, and therefore we obtain a set $L_{d_1}, \ldots, L_{d_k}$ of remaining literals whose simultaneous satisfaction is a sufficient condition for the conflict. By adding the *conflict-induced clause*, also called *lemma*, $C_C = \{\overline{L}_{d_1}, \ldots, \overline{L}_{d_k}\}$ to the clause set, we can thus prevent a useless repeated search of the same subtree in other regions of the search space. We will not describe in detail how the literals evoking a conflict are computed, but refer the reader to the literature instead (see for example [24]).

Adding all conflict clauses can result in an exponential blow-up of the clause

set. Therefore it is common practice to limit the addition of clauses to those containing less than a threshold number of literals. The use of this simple size parameter is justified by the fact that smaller clauses have the potential to cut off larger fractions of the search space: a clause of length n can truncate up to $\frac{1}{2^n}$ of the search space.

3 Parallel SAT Checking with Distributed Dynamic Learning

3.1 Basic Problem Decomposition Technique

For the parallel execution of the DP algorithm the search space has to be divided into mutually disjoint portions to be treated in parallel. We adopt a dynamic search space splitting technique proposed by Zhang *et al.* [36] 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 the clause set S, 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 has 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 task, where at the same time the first task switches the flag in its 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.

As an example, assume that the search process has reached the state indicated by the marked guiding path in Figure 2. In this situation a new task may be started with literal $\overline{x}$ set to true, as this part of the search tree has not been examined so far. So by starting a new task with an initial guiding path of $((\overline{x}, \mathsf{N}))$ we start a parallel search of independent subtrees. The spawning task can proceed with its search, after having changed its guiding path from $((x, \mathsf{B}), (y, \mathsf{N}), (\overline{z}, \mathsf{B}))$ to $((x, \mathsf{N}), (y, \mathsf{N}), (\overline{z}, \mathsf{B}))$. It has proved advantageous to choose for splitting purposes the backtracking node that is closest to the root

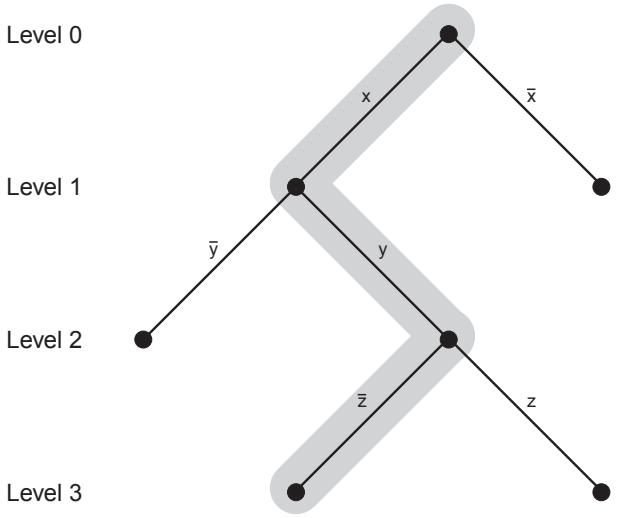

Fig. 2. Guiding Path $((x, \mathsf{B}), (y, \mathsf{N}), (\overline{z}, \mathsf{B}))$, Left-to-right Search Tree Traversal

of the search tree, i.e. to select i in such a way that $f_j = \mathsf{N}$ for all $j < i$.

The guiding path approach allows dynamic problem decomposition, as at any point in time during the search any task may decide to further split its portion of the search space. Moreover, the selected literals coincide with the selections of the sequential version—at least in the absence of dynamic learning. Approved literal selection strategies may therefore be carried over to the parallel version of the DP algorithm. We have modified algorithm `DP` to be started at an arbitrary point in the search tree, specified by an initial guiding path.

3.2 Distributed Learning

For a distributed parallel version of the DP algorithm with dynamic learning, splitting the search space is only one problem to be settled. The other one is to find a suitable scheme for distributing newly gained knowledge. Such a scheme has to decide about questions such as:

- When should knowledge be exchanged between two tasks?
- Which newly derived facts should be made available to other tasks?
- Which knowledge is relevant and should be integrated into a task's clause set?

For all of these questions side conditions have to be considered, such as network bandwidth and time to assemble and incorporate new knowledge.

The simplest schema for distributed learning is to have all tasks working independently and to exchange no knowledge at all. This scheme has the advan-

tage of being very simple (to implement, too), but it suffers from the drawback that important knowledge may be utilized incompletely. This holds particularly when learning has a considerable effect, as is the case with structured real-world instances of the SAT problem.

Our approach is to exchange selected knowledge—in the form of newly derived lemmas (conflict clauses)—between the nodes of the distributed environment. The lemmas that are made available to other tasks are selected using one simple criterion: the clause length. All lemmas with fewer than a fixed number of literals are offered to all other tasks. This policy is consistent with the schema applied for deciding the lemmas to be added to the clause set in the sequential algorithm (see Section 2.2). In order to determine an appropriate value for this size parameter we conducted experiments, which are reported in Section 5. When inserting foreign lemmas into a task's clause set, these are filtered, and only those lemmas are incorporated that are not subsumed by the task's initial guiding path. This prevents insertion of lemmas that are superfluous for the currently examined part of the search space.

3.3 Parallel Programming Models and Implementation Concepts

Since the dynamic learning process represents an orthogonal procedure to the search process in the sequential algorithm, it is natural to also separate both processes in the parallel algorithm. Thus, the parallel execution is organized in two different logical layers. On the first layer, a parallel search process is carried out, while on the second layer the exchange of newly created knowledge between the processors is accomplished.

In this section the selection of suitable parallel programming models and the resulting conceptual parallel organization of the two logical layers is discussed. Section 4 gives further details of the actual implementation of both layers using the DOTS parallel system platform.

Many parallel programming models have been proposed in the past [32,2]. Besides the different levels of abstraction they provide, they also differ with respect to their applicability to specific problem domains. Subsequently, we identify specific requirements for the parallelization of our algorithm and choose suitable parallel programming models.

3.3.1 Parallel Search

Using the guiding path technique, it is easily possible to create disjoint subproblems for carrying out a parallel search. But it is in principle impossible to estimate the run-time of a subproblem, since the extent of problem reduction

delivered by the constraint-propagation and especially by the dynamic learning process cannot be predicted. Especially when dealing with SAT encodings of real-world problems, the run-times of the created subproblems differ considerably. This means that for the realization of the parallel search process on the first logical layer, the programming model has to support task parallel programs exhibiting highly irregular workloads and resulting in highly irregular communication patterns. In particular, it should be possible to efficiently implement the presented dynamic search space splitting technique.

The application of higher level parallel programming models which are supported by parallelizing compilers is mainly restricted to regular applications and thus not possible in our case. Also, lower level models like message passing are not well suited for the efficient parallelization of applications with highly irregular communication structures, because for every send primitive also a corresponding receive primitive has to be executed. The explicit placement of these receive statements in the program code turns out to be difficult when dealing with highly irregular communication patterns.

The multithreading programming paradigm, located at a medium level of abstraction, provides transparent synchronization on the receiver side and hides communication by an argument-result abstraction. It turned out to be a well suited model for the parallelization of highly irregular applications. Moreover using this parallel programming model, different load balancing strategies can be applied orthogonally.

Multithreaded computations can further be classified according to the number and course of data-dependency edges present in the execution graph of a computation [7]. In *simple multithreaded computations* (also called *asynchronous procedure calls* or *fork/join computations*) each thread produces one result which is consumed by its parent. In *fully strict multithreaded computations* a thread can produce an arbitrary number of results, consumed by its parent. The more general *strict multithreaded computations* allow that the results of a thread can be consumed by an ancestor of the thread in the activation tree. (Further details on this classification can be found in [7].)

The strict multithreading model is very well suited for implementing the parallel DP search procedure within a scalable *dynamic* master-slave approach. Dynamic problem decomposition is achieved by dynamically creating new threads, where a parent thread assigns a portion of its own search space to its child by applying the described search space splitting technique, and continues the search in its (reduced) search space. Both, parent and child, can create additional threads to further increase the available parallelism. All threads pass the result of their search process directly to the master thread. The master thread only spawns the initial thread and subsequently collects the results of all dynamically generated threads until a model is found or all created threads

have finished their search without finding a model.

3.3.2 Knowledge Exchange

To establish a global learning process considering all generated knowledge, the lemmas have to be exchanged between the nodes. Since at every leaf in the search tree a conflict analysis is carried out, a vast number of lemmas are generated at each node. This makes the exchange of all created lemmas among all nodes difficult, if bandwidth limitations of the network exist or when a large number of nodes are used. Thus lemmas must be filtered at the source, and the underlying programming model used for implementing the lemma exchange should support such a selection process.

In case of a parallel computer with physical shared memory, the knowledge exchange can be carried out by maintaining a shared clause store object into which all created lemmas are inserted and from which they are selectively read by all prover instances. Using distributed shared memory models this concept could also be used in distributed architectures. However, this approach suffers from limited scalability. Another possibility would be to use broadcasting capabilities of message passing environments, but filtering at the source is difficult to realize with this technique.

In our approach we use (a simple form of) mobile agents [34] to gather suitable new knowledge on other nodes. For each SAT prover instance, a mobile agent is created that visits the nodes in the distributed system. The agents gather new lemmas according to the criteria specified in Section 3.2. Whenever the mobile agent delivers the collected lemmas on its home node, information about the current state of the local search process is passed to the agent and is used in the selection process on its next trip to the other nodes. The mobile agent paradigm is distinguished by its high scalability, thus enabling the deployment of our algorithm in large scale parallel environments, e.g. computational grids.

4 Implementation on the DOTS parallel platform

DOTS (Distributed Object-Oriented Threads System) is a system platform for building and executing parallel C++ programs that integrates a wide range of different computing systems into a homogeneous parallel environment. Although DOTS was originally designed for the parallelization of algorithms from the realm of symbolic computation [6], it has also been used in other application domains like parallel computer graphics [26]. The primary design goal of DOTS is to provide a flexible and handy tool for the rapid prototyping of algorithm designs especially for highly irregular symbolic computations, e.g.

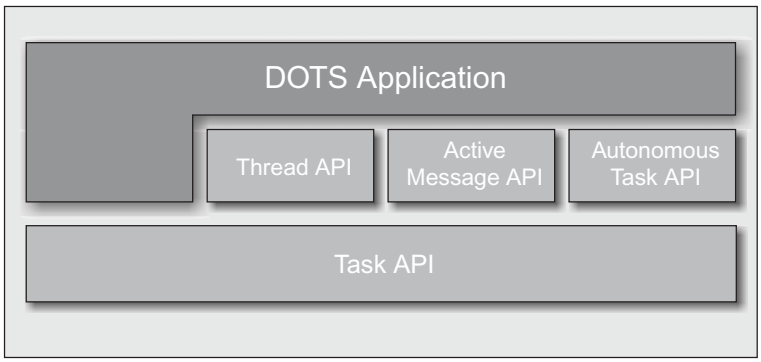

Fig. 3. DOTS Application Programming Interface

in data-dependent divide-and-conquer algorithms.

DOTS supports a wide range of hardware and software platforms [4]. Up to now, it has been deployed on (heterogeneous) clusters composed of the following platforms:

- Microsoft Windows 98/NT/2000/XP
- Solaris, IRIX, AIX
- FreeBSD, Linux
- QNX Realtime Platform and
- IBM Parallel Sysplex Clusters (clusters of IBM S/390 (respectively zSeries) mainframes running under OS/390) [5]

4.1 System Overview

4.1.1 DOTS APIs

DOTS applications can be based on several APIs, see Figure 3. It is possible to mix primitives from different APIs within an application.

Task API. The Task API represents the basic API layer of DOTS on which the other APIs are based. It provides support for DOTS task objects, which are instances of application specific classes that 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 (see Section 4.1.2). The base class DOTS_Task also provides methods for explicit program controlled migration of the task object in the case of a distributed execution of the parallel application.

Autonomous Tasks API. The Autonomous Tasks API can be used to create task objects that operate as mobile agents. In contrast to normal task objects, the execution of an autonomous task is not subject to 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 organizing round trips of mobile agents within the distributed environment.

Active Message API. The Active Message API provides support for object-oriented message passing. After a message object has been 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.

Thread API. The Thread API offers support for strict multithreading. To facilitate the parallelization of C++ programs using this programming model, the Thread API is enhanced with object-oriented features, like argument and result objects for threads. The basic primitives provided by the Thread API are dots_fork and dots_hyperfork for thread creation, dots_join for synchronizing with the results computed by other threads (which are returned using dots_return), and dots_cancel for thread cancellation.

Each thread of a computation is assigned to a *thread group*. Depending on the primitive used for its creation, a thread is placed explicitly or implicitly into a thread group. If a thread is created using the dots_fork primitive, it is explicitly placed into a specified thread group. If a thread is created using dots_hyperfork, it is implicitly placed in the same thread group as its closest ancestor in the thread activation tree which has been created using dots_fork.

When dots_join is called on a thread group, *join-any* semantics is applied: The first result which becomes available from a thread in the given group is delivered, regardless of whether the thread has been placed explicitly or implicitly into the group. If no result is available, the calling thread is blocked until one thread of the group delivers a result.

The Thread API is implemented on top of the Task API by defining a special class of DOTS task objects called DOTS_Thread, which encapsulates argument and result objects and additional information like the procedure to be executed.

4.1.2 DOTS Architecture

The major design goal for the DOTS architecture is the strict separation of execution and distribution aspects. The benefits of this approach are that DOTS

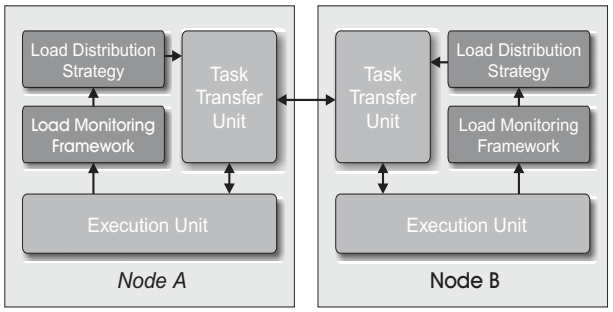

Fig. 4. DOTS Architecture

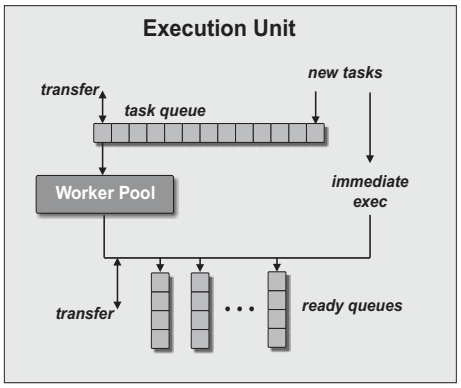

Fig. 5. DOTS Execution Unit

applications can be efficiently executed on SMPs as well as on clusters without any modifications (e.g. recompilation). Moreover, custom load distribution schemes and new functionality (e.g. checkpointing) can easily be integrated.

Figure 4 shows the main functional units of the architecture. DOTS tasks are executed within the *Execution Unit* (shown in Figure 5). 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. However, in some cases an oversaturation with worker threads can be desirable, so that communication latency is implicitly hidden by running another thread. After the execution of a DOTS task object is completed, it is placed into a *ready queue*. Ready queues correspond to thread groups on the API level; for each thread group that is created, a ready queue is allocated within the execution unit. The ready queue is removed from the execution unit when the corresponding thread group is canceled.

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 integrated. The occurrence of an event is transformed by the Load Monitoring Framework into a call of a corresponding event handler method that can be (re-)implemented by a load distribution strategy class derived from the Load Monitoring Framework. In turn, event handler methods initiate appropriate actions, like task transfer or sending requests for task transfer. As an example for the usage of the Load Monitoring

Framework, Figure 6 shows the code for a simple randomized work-stealing distribution scheme. Using the Load Monitoring Framework, the programmer can easily register custom load distribution schemes within an application without any modification of DOTS internals.

Basic distributed schemes based on sender initiated work-sharing or receiver initiated work-stealing are predefined. The target, resp. victim node, can be selected randomly, by round-robbin selection, or by an application specific strategy.

4.2 Implementation of the parallel SAT checker using DOTS

In this section, we present the implementation of the two logical layers of our parallel SAT checker (see Section 3.3) using DOTS.

4.2.1 Parallel Search with DOTS Threads

To initiate the parallel search process the main thread forks one DOTS thread that has the entire search space assigned. During the whole computation all created threads periodically monitor the length of the local task queue. If a thread sees that the length of the task queue falls below a given threshold, it forks a new DOTS thread. The parent thread splits off a region of its search space (see Section 3.1) and assigns it to the new thread. To prevent the uncontrolled splitting-off of very small fractions of the search space, a predefined time interval has to elapse before the next split can be carried out by the thread. The newly created DOTS task object is queued and can be executed by a local worker thread or can be transferred to other nodes.

The above 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 all subsequently created threads. All DOTS threads (except the initial one) are created with the dots_hyperfork primitive. This has the effect that these threads can be joined by the main thread (and are not to be joined by their actual parent threads). The result of a thread indicates whether a solution was found within the assigned search region. The processing is completed either if all created DOTS threads have been joined (indicated by a return value of 0 from dots_join) without returning a solution, or when the first DOTS thread is joined that has found a solution. In the latter case,

```
#include "DOTS_Load_Monitoring_Framework.h"

class DOTS_Simple_Randomized_Workstealing_Strategy :
   public DOTS_Load_Monitoring_Framework
{
public:
  DOTS_Simple_Randomized_Workstealing_Strategy() {
    // subscribe for exec_queue_min_threshold_event
    // steal threshold: 0, queue check interval: 100 ms
    set_exec_queue_min_threshold(0, 100);
  }

  void exec_queue_min_threshold_event_handler(void) {
    // are there any other nodes?
    if (DOTS_NODE_TABLE->get_size()<2)
      return;

    // create message containing own node ID
    DOTS_Archive arch;
    arch << DOTS_NODE_ID;
    DOTS_Msg msg(DOTS_MSG_STEAL_TASK, &arch);

    // choose victim node randomly
    DOTS_Node_ID victim_id;
    do {
      victim_id = rand() % DOTS_NODE_TABLE->get_size();
    } while (victim_id == DOTS_NODE_ID);

    // send message to victim node
    msg.send(victim_id);
  }

  void message_handler(DOTS_Msg* msg) {
    // unpack message
    DOTS_Archive* arch = msg->get_archive();
    DOTS_Node_ID dest_id;
    *arch >> dest_id;

    // try to transfer task to destination node ID
    DOTS_TASK_TRANSFER_UNIT->transfer_task(dest_id);

    delete msg;
  }
};
```

Fig. 6. A Simple Randomized Load Distribution Strategy

all remaining DOTS threads are immediately canceled.

As load distribution scheme, task stealing with randomized victim selection (similar to the example code given in Figure 6) was used. 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) [22]. Since this load distribution scheme involves only local information the scalability of the parallel search algorithm is ensured.

4.2.2 Knowledge Exchange using DOTS 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. Lemmas can easily be exchanged between clause store objects residing on the same node, using shared memory. Lemmas from clause stores on other nodes are exchanged by employing DOTS autonomous tasks. In Figure 7 the (simplified) code of an autonomous task is shown.

For each clause store object a DOTS autonomous task object is created that acts as a mobile agent for gathering lemmas from other nodes. It visits all nodes in a round robin fashion looking for new lemmas. Every time it is back on its home node it inserts the collected lemmas into the local clause store. Because of the huge amount of generated lemmas it is impossible to exchange all lemmas in larger distributed systems. Therefore, agents gather only lemmas that meet some criteria. As selection criteria the length of the lemmas and the requirement that the considered lemma is not already *subsumed* is used. (If a lemma is subsumed, it contains only 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—in the form of a list of *fixed literals*—is transferred to the lemma exchange agent every time the agent visits its home node.

5 Experimental Results

For a performance evaluation of our parallel approach to SAT checking with dynamic learning, we carried out a series of run-time measurements in a cluster composed of 24 SUN workstations. Each node of the cluster was equipped with an UltraSparcII processor running at 500 MHz and 512 MByte of main memory. All nodes were connected by a 100 Mbps switched Ethernet. Our run-times and speedup values are based on measurements of the wall-clock

```
class LemmaAgent : public DOTS_Autonomous_Task
{
private:
  List<Clause> new_lemmas;
  List<Literal> fixed_literals;

  int max_length;

public:
  run() {
    if (home_node()) {
      // deliver gatherd lemmas
      CLAUSE_STORE.insert_new_lemmas(new_lemmas);

      // get current state of search process
      fixed_literals = PROVER.get_fixed_literals();

      // continue trip
      travel_to_next_node();
    }
    else {
      Clause l;

      // gather new lemmas
      while ((l = CLAUSE_STORE.read_new_lemma()) != 0) {

        // discard if subsumed or max. length exceeded
        if (!l.subsumed(fixed_literals) && l.length() <= max_length)
          new_lemmas.append(l);
      }

      // continue trip
      travel_to_next_node();
    }
  }
};
```

Fig. 7. Simplified Code of the Lemma Agent

time of program runs. Since it turned out that the parallel execution of the SAT checker with dynamic learning exhibits a significant non-deterministic behavior in some cases, we performed ten individual parallel runs for each setting. In addition to the arithmetic mean of the measured results we also give the minimum and maximum values, if the individual values show a significant spread.

As benchmarks we used the following SAT encoded problems of both theoret-

ical and practical importance:

- *QG7-12*: quasigroup existence problem
  This problem encodes a quasigroup existence problem of the kind QG7 given by Fujita *et al.* [19]. A quasigroup is a cancellative finite groupoid consisting of a base set S and a binary multiplication $*$. The multiplication table is also known as a "latin square", i.e. each row and column is a permutation of the base set S. QG7-x asks for the existence of a quasigroup of order x with the additional property that $((x * y) * x) * y = x$ holds for all x, y. As no quasigroup of order 12 with this property exists, QG7-12 is unsatisfiable.
- *DES*: logical cryptanalysis
  This benchmark stems from the area of logical cryptanalysis [25] and encodes the problem of finding an encryption key given three plaintext and three ciphertext blocks that were produced using three rounds of the DES algorithm. As there is (at least one) key matching the plaintext/ciphertext blocks, the DES problems are satisfiable.
- *LONGMULT*: hardware verification
  This benchmark problem is taken from the realm of hardware verification using bounded model checking [3]. It represents a boolean formula expressing the equivalence of two different 16-bit multiplier hardware designs. Using the usual problem encoding technique of bounded model checking, equivalence of hardware designs is represented by unsatisfiability.

Table 1 shows the the sequential run-times of the benchmark problems with and without dynamic learning measured on one cluster node. The computation time of all benchmark problems can be reduced by dynamic learning. While the effect imposed by dynamic learning is moderate for the quasigroup existence problem, the run-time of the other considered benchmark problems can be dramatically reduced using this technique.

Benchmark	satisfiable ?	sequential time without learning	sequential time with learning
QG7-12	no	5,203 sec	2,772 sec
DES	yes	>7 days	2,984 sec
LONGMULT	no	99,009 sec	4,313 sec

Table 1
Sequential run-times of the benchmarks

5.1 Evaluation of the Work-Stealing Load Sharing Strategy

In a first series of measurements the performance of our work-stealing load sharing strategy was analyzed and then optimized for subsequent measure-

ments. In order to get suitable data for this analysis, we used as input the *QG7-12* benchmark where among the considered benchmarks the effect of dynamic learning is the smallest and performed no lemma exchange. This strategy minimizes both, the non-determinism of parallel runs (see also Figure 13 below), and the overhead imposed by the mobile agents on the parallel search.

5.1.1 Work-Stealing Threshold Parameters

The work-stealing load sharing strategy is controlled by the value of two different threshold parameters:

- *Split-Threshold*
  When the length of the local task queue is less than, or equal to, this threshold, a search space split is performed to produce an additional thread (which can be stolen by other nodes).
- *Steal-Threshold*
  When the length of the local task queue is less than, or equal to, this threshold, the load distribution system tries to steal a thread from the run queue of a randomly chosen victim node.

Figure 8 shows the obtained speedups and the number of dynamically generated threads for different values of the *Split-Threshold* and *Steal-Threshold* parameter using 24 nodes with one worker thread on each node. In Figure 9 the results of the corresponding measurements using two worker threads per node is given.

Discussion
The speedups obtained using one worker thread ranged from 15.7 to 18.1 and with two worker threads per node they ranged from 14.9 to 17.6.

The best speedup could be achieved by using a value of 0 for the *Split-Threshold* and the *Steal-Threshold* parameter, both when using one or two worker threads. Also the number of created threads is the smallest with this parameter setting.

Using a larger value for the *Steal-Threshold* causes additional task transfers over the network leading to smaller speedups. Also, using a larger *Split-Threshold* causes the creation of a larger number of threads, particularly for smaller values for the *Steal-Threshold* parameter. Thus, the strategy of trying to keep a larger number of parallel task available turned out to be not beneficial for this application.

Using two worker threads per node decreases the performance of the parallel application. The additional synchronization overhead needed when using two

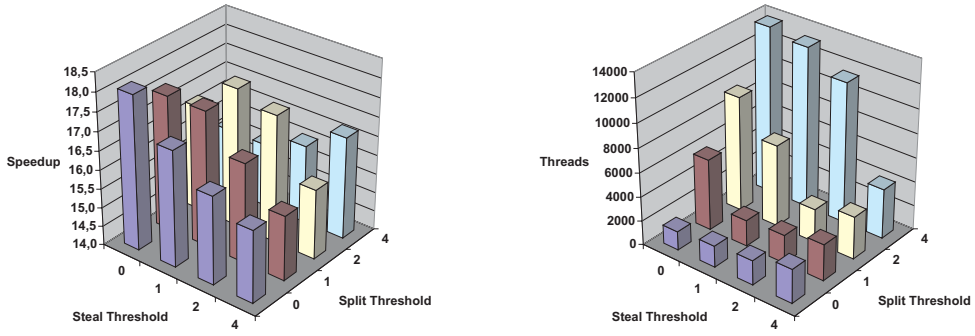

Fig. 8. Speedups and number of generated threads for *QG7-12* using different work-stealing thresholds with one worker thread per node

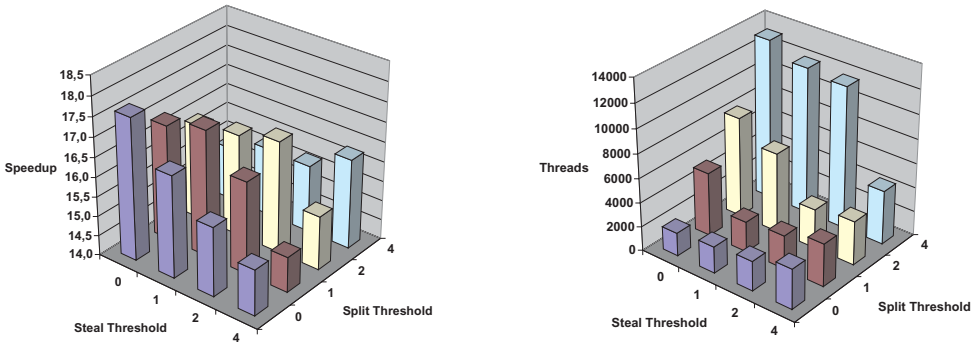

Fig. 9. Speedups and number of generated threads for *QG7-12* using different work-stealing thresholds with two worker threads per node

prover threads outweighs the performance improvements obtained by overlapping computation and communication. In general it turned out that due to the internal multithreaded implementation of the communication system of DOTS the effect of using a larger number of worker threads is only marginal. Despite the high variability in the number of generated threads, the resulting speedups are relatively stable. E.g., for the parameter setting that results in the highest thread load when using one worker thread, still about 87 percent of the maximal speedup could be obtained.

According to the results of this test, for all subsequent measurements the value of the *Split-Threshold* and the *Steal-Threshold* parameter were set to 0 and one worker thread per node was used.

5.1.2 Work-Stealing Timing Parameters

After finding the optimal thresholds for the work-stealing load sharing strategy, we studied the influence of the following related timing parameters on the speedup and the number of created threads.

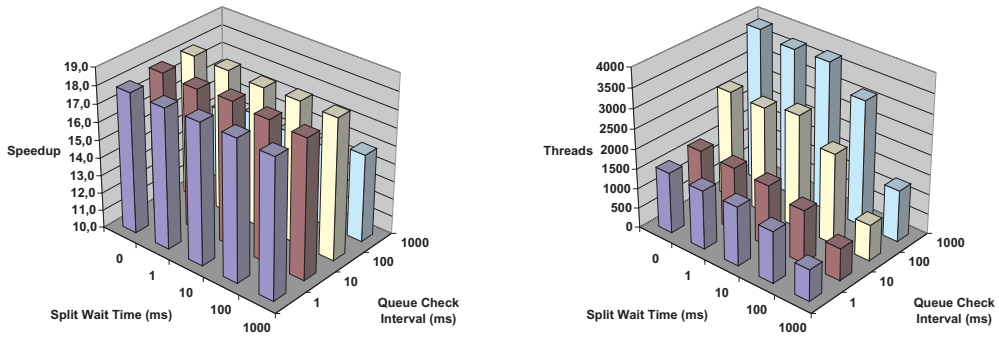

Fig. 10. Speedups and number of generated threads for *QG7-12* using different work-stealing timing parameters

- *Split-Wait Time*
  This is the minimal time interval that has to elapse between consecutive search space splits on a node.
- *Queue-Check Interval*
  This parameter controls how frequently the local task queue is checked whether its length has fallen below the given threshold parameter.

Figure 10 shows the speedups and number of generated threads using different combinations of the described timing parameters.

Discussion
For a wide range of parameter settings the obtained speedups differ only marginally (ranging from 17.8 to 18.1), while the number of generated threads is affected by these parameters to a greater extent. Only when using a comparatively large queue check interval of 1000 ms the speedups drop significantly to values in the range between 14.1 to 14.9, and the largest number of threads occur. In this case, mainly at the end of the computation, many nodes ran idle for a larger time interval leading to a poor parallel efficiency. Moreover, only a few nodes could be chosen as victim for work-stealing. Consequently, these nodes had to perform many search space splits within a short time interval. The resulting small subproblems amplified the described effect, also leading to a larger number of generated threads.

For all further measurements a *Split-Wait Time* value of 10ms and a *Queue-Check Interval* value of 10ms were chosen since for these values the speedup obtained was the highest.

For analyzing the effects of knowledge exchange realized by our mobile agents approach, we studied for each benchmark the influence of the maximum length of the lemmas to be gathered by the mobile agents. This parameter mainly influences the footprint (the size) of the mobile agents. On the one hand, smaller values for the maximum length lead to faster round trip times of the agents, so that the transferred new knowledge is rapidly available on other nodes of the distributed system. Also the overhead imposed by the mobile agents on the parallel search process in terms of cpu-time and bandwidth consumption can be reduced. On the other hand, by choosing a too small size for the lemmas to be exchanged, too many generated lemmas are filtered out from the global learning process, so that important knowledge may not be provided to other nodes, leading to a larger degree of redundant searching.

Figure 11 shows the speedups and the total size of the search tree (given in the number of leaves of the tree) for the *LONGMULT* benchmark using different maximum length values. A maximum length of 0 represents the case where no lemmas are exchanged at all, and consequently no mobile agents are created. Figure 12 and Figure 13 present the corresponding results for the *DES* and *QG7-12* benchmarks.

Discussion
Coordinating the distributed learning process with our mobile agent approach leads to significantly better speedup values and to smaller search trees in all our benchmarks. Moreover, it could be observed that the maximum size of the collected lemmas influences the obtained speedups as well as the size of the search tree for all benchmarks. In particular, this effect is very pronounced for the *DES* benchmark.

It turned out that for all benchmarks the maximum size of exchanged lemmas that leads to the best speedups is smaller than the maximum lemma size which results in the smallest search tree. By choosing a larger size, more lemmas are selected and exchanged, which on the one hand can further reduce the search tree since more knowledge is distributed. But on the other hand, the clause set gets bigger, slowing down the constraint propagation process of the DP algorithm and thus decreasing the overall performance. Moreover the resource consumption of the agents is increased, additionally slowing down the parallel search.

Also, super-linear speedups could be observed for the *LONGMULT* and the *DES* benchmarks. One typical source of super-linear speedups are system effects such as cache or memory effects: due to the problem decomposition the resulting smaller subproblems can often be processed more efficiently. In our

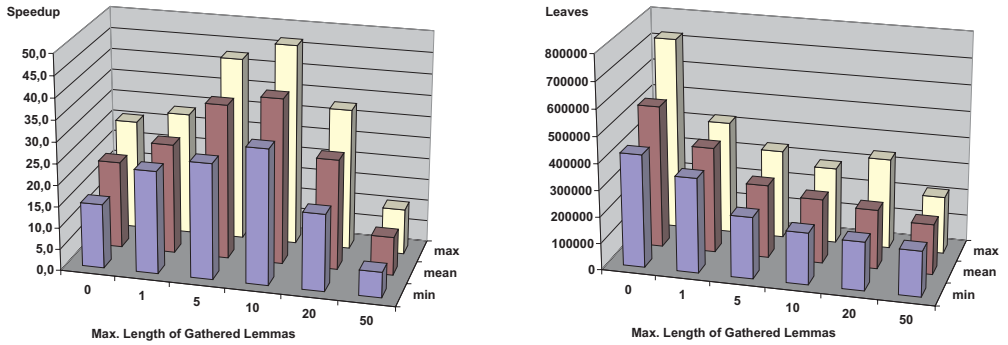

Fig. 11. Influence of Lemma Exchange for the *LONGMULT* benchmark

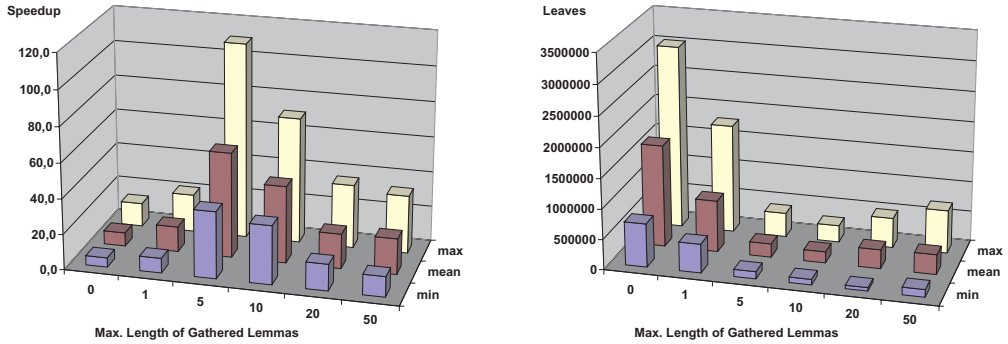

Fig. 12. Influence of Lemma Exchange for the *DES* benchmark

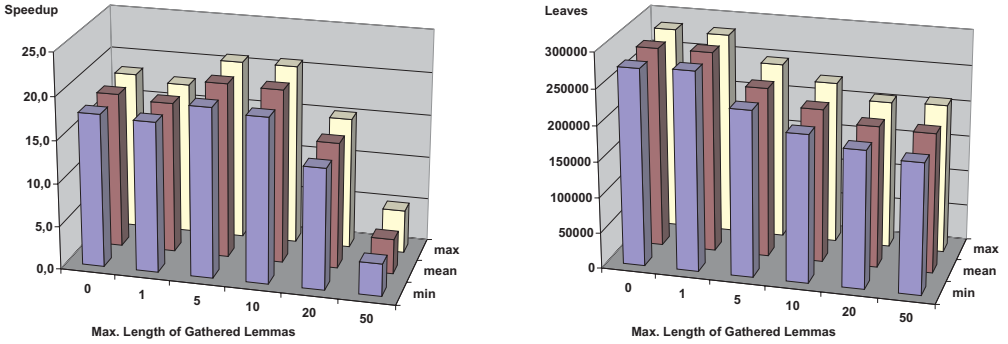

Fig. 13. Influence of Lemma Exchange for the *QG7-12* benchmark

application this is unlikely to be the case since for treating a subproblem a complete prover instance is needed which requires the same resources as in the sequential execution.

Another typical source of super-linear speedups are algorithmic effects. In parallel search processes these occur, if the processing of a parallel subproblem quickly leads to a solution. Since the DP algorithm is essentially a search algorithm, super-linear speedups can occur due to this parallel search effect, if the considered problem instance is satisfiable.

Parallel dynamic learning processes are another potential source for algorithmic super-linear speedups. Compared to the sequential execution, it is possible that in the parallel case additional important knowledge becomes available when treating a particular region of the search space. Thus, the search tree can be reduced to a greater extent than in the sequential case.

This observation provides an explanation of the occurrence of super-linear speedups in the parallel execution of the *LONGMULT* benchmark. For the *DES* benchmark, super-linear speedups are the result of a superposition of both parallel search and parallel learning.

6 Related Work

6.1 Parallel SAT Checking and Combinatorial Optimization

Böhm and Speckenmeyer presented a parallel SAT-solver for a Transputer system consisting of 256 processors [8]. Their work concentrates on aspects of the parallelization of SAT checking for hard randomly generated SAT instances. The SAT algorithm executed on each processor is based on the classical DP algorithm without conflict analysis and lemma generation and consequently no inter-node learning process is realized in this work. A dynamic problem decomposition technique similar to the guiding path technique is used. The employed load balancing scheme depends on a workload measure of subproblems which is based on the number of unset variables of the subproblem. This approach is feasible for random SAT instances where the extent of problem reduction delivered by the constraint-propagation step of the DP algorithm can be assumed to be the same for all subproblems. However, for many real-world SAT instances, this assumption can not be made. Therefore our approach uses a dynamic receiver initiated load distribution scheme in order to better cope with the highly irregular workload when treating real-world SAT instances.

Zhang's PSATO [36] is a distributed parallel propositional prover for networks of workstations, based on the sequential prover SATO. Compared to PSATO, our work implements a distributed dynamic learning process and applies more sophisticated load distribution schemes leading to better scalability. PSATO also employs a master-slave approach for carrying out the parallel search. In contrast to our work, the master is additionally responsible for performing search space splitting and scheduling of the created subproblems. In PSATO, new tasks are not created on demand (when a processor is available), but the following procedure is applied: After a predefined time interval a slave terminates its computation and reports its current guiding path to the master. If there is another idle processor, the master performs a search space split and

assigns the two resulting subproblems to the idle processors. Due to this highly centralized policy of dynamic problem decomposition scalability is limited, especially when treating highly irregular problem instances.

In the neighboring field of combinatorial optimization, parallelization is also an active area of research. Grama and Kumar [20] give a comprehensive overview of the state-of-the-art in this field. They report on the parallelization of different search algorithms, the role of heuristics, and the phenomenon of speed-up anomalies.

Habbas *et al.* [21] examine different load balancing strategies for parallel forward search with conflict based backjumping. Search algorithms of this kind frequently occur in constraint satisfaction problems (CSPs) and are similar to the DP algorithm with conflict analysis presented here. In the CSP community, the analogy to lemma generation is no-good generation, but this is not part of Habbas' analysis. So, in their parallel algorithms the focus is on load balancing, and distributed learning is not considered.

Brüngger's ZRAM search library [10] offers a framework for parallel search. In their work they exemplarily apply their system to the traveling salesman problem (TSP) and to the quadratic assignment problem (QAP). Their library offers different parallel search engines, for example for branch and bound search, and includes a search space estimator for irregular search problems, which is based on ideas of Knuth [23]. In their framework, communication between different tasks is done exclusively for load distribution, so the focus again is not on distributed learning.

6.2 System Platforms for Multithreading in Distributed Systems

In Section 3.3 we have stated the general suitability of multithreading for the parallelization of highly irregular combinatorial search problems. In the last decade, many distributed multithreaded environments for high performance computing have been developed. In this section we carry out a classification of these approaches into three categories according to functional aspects and the intended purpose of the system platform. For the category to which DOTS belongs, we make a more detailed comparison with other systems.

Shared Memory Multithreading based on Distributed Shared Memory (DSM)
Here, the common goal of approaches to distributed multithreading is to use the shared memory multithreading programming model, e.g. provided by many modern operating systems, on parallel architectures with distributed memory as transparently as possible. Typical representatives of this category are: DSM-Threads [28], Millipede [18], or DSM-PM2 [1].

Due to the consistency problem caused by the replication of shared data objects, all approaches to DSM multithreading have to cope with the problem of combining efficiency and programmability (transparency) on large scale distributed (heterogeneous) parallel machines [33]. In principle, strict multithreading and mobile agents could be realized using DSM, if the stated drawbacks are not relevant for the intended application area. Since SAT checking does not further profit from DSM, we argue that it is better to use DOTS, whose system model supports larger scale heterogeneous distributed systems.

HPC Middleware for Integrating Communication and Multithreading

The system platforms in this category pursue the tight integration of communication and multithreading. However, the intention of the realized parallel programming model is not to carry out communication completely transparently. Examples of members of this category are Nexus [17], Panda [31], or Athapascan-0 [9].

These system platforms are primarily designed to be used as compiler targets or as middleware for building higher-level parallel system platforms. Due to their general nature, strict multithreading as well as mobile agents could be realized with all of these platforms. But using their low level programming models would lead to relatively complex programs when implementing both strict multithreading and autonomous tasks.

Platforms Supporting the Fork/Join Multithreading Programming Model

Systems in this category are most similar to DOTS; they realize distributed multithreading by employing the fork/join parallel programming model or generalizations thereof, like strict multithreading (see Section 3.3.1). This approach to distributed multithreading carries out communication completely transparently by using argument-result semantics. However, communication between the threads of a computation is restricted to specific points during the execution of a thread. The simplest form are asynchronous remote procedure calls that allow the passing of one argument from the parent thread to the child thread and the communication of one result back to the parent thread. More general models, like strict multithreading, additionally allow the transfer of possibly several results to any ancestor of a thread in the call tree.

DTS [11] (which is the predecessor of DOTS) realizes asynchronous remote procedure calls in C and Fortran. No support for object-oriented programming was provided in DTS, and its deployment was limited to distributed systems composed of UNIX nodes.

Cilk [30] is a language for multithreaded parallel programming that represents a superset of ANSI C. It uses pre-compilation techniques for static code in-

strumentation in order to support the Cilk runtime system. There exists a prototype implementation of a distributed version of Cilk, called distributed Cilk [16], that spans clusters of SMPs. DOTS is library based and therefore avoids typical problems of systems that extend standard languages, like the lack of standard development tools (e.g. debuggers). Moreover, DOTS is based on C++ and supports object-oriented programming. Since distributed Cilk is currently available only on a few platforms, its usability in highly heterogeneous distributed environments is limited.

Virtual Data Space (VDS) [15] is a load balancing system for irregular applications also supporting strict multithreading. It is implemented in C and therefore provides no direct support for object-oriented programming. It is not available for a wider range of common platforms, resulting in a limited support for heterogeneous high performance computing.

PM2 (Parallel Multithreaded Machine) [29] is a distributed multithreaded environment designed to efficiently support irregular parallel applications on distributed architectures. A key feature of PM2 is its thread migration mechanism.

While all these system platforms provide for fork/join multithreading in distributed systems, only VDS and PM2 could also be used to implement the autonomous tasks that act as mobile agents supporting distributed learning. Their implementation could be based on the VDS task model resp. on the sophisticated migration facilities of PM2.

DOTS is distinguished from these systems by its native support for object-oriented programming, its support for highly heterogeneous environments, and by its tight integration of strict multithreading and autonomous tasks realized by its basic task execution model.

7 Conclusion

In this paper we presented the parallelization of the state-of-the-art SAT checking algorithm which enhances the classical Davis-Putnam-Logemann-Loveland SAT checking procedure with dynamic learning techniques. Our approach uses strict multithreading to cope with the highly irregular search process and employs a randomized work stealing strategy. The knowledge exchange is carried out by using the mobile agent paradigm.

The main contribution of this paper is the beneficial combination of parallel combinatorial search with distributed learning. We thus achieve an improved global learning effect on a set of collaborating search tasks. Our experiments

indicate that distributed learning can result in considerable speed-up compared to independently learning individuals.

As the learning effect helps reveal the internal structure of a problem instance, it is especially well-suited for structured real-world problems. This is the case, for example, in hardware verification, where SAT algorithms (as the core of a bounded model checker) are increasingly employed, and where a parallel approach allows further progress.

Research directions for the future may include adaptation to large-scale applications by employing grid computing, as well as a thorough theoretical investigation of the effects of distributed learning.

References

[1] Antoniu, G., and Bougé, L. DSM-PM2: A portable implementation platform for multithreaded dsm consistency protocols. In *In Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)* (San Francisco, April 2001), vol. 2026 of *Lect. Notes in Comp. Science*, Springer-Verlag, pp. 55–70.

[2] Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. Programming languages for distributed computing systems. *ACM Computing Surveys 21*, 3 (September 1989), 261–322.

[3] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)* (1999), no. 1579 in LNCS, Springer-Verlag.

[4] Blochinger, W. Distributed high performance computing in heterogeneous environments with DOTS. In *Proc. of Intl. Parallel and Distributed Processing Symp. (10th Heterogeneous Computing Workshop)* (San Francisco, CA, U.S.A., April 2001), IEEE Computer Society Press, p. 90.

[5] Blochinger, W., Bündgen, R., and Heinemann, A. Dependable high performance computing on a Parallel Sysplex cluster. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)* (Las Vegas, NV, U.S.A., June 2000), H. R. Arabnia, Ed., vol. 3, CSREA Press, pp. 1627–1633.

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

[7] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science (FOCS '94)* (Mexico, November 1994), pp. 356–368.

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

[9] Briat, J., Ginzburg, I., Pasin, M., and Plateau, B. Athapascan runtime: Efficiency for irregular problems. In *Proc. of the Europar'97 Conference* (Passau, Germany, August 1997), Springer Verlag, pp. 590–599.

[10] Brüngger, A., Marzetta, A., Clausen, J., and Perregaard, M. Solving large-scale QAP problems in parallel with the search library ZRAM. *Journal of Parallel and Distributed Computing 50*, 1 (May 1998), 157–169.

[11] Bubeck, T., Hiller, M., Küchlin, W., and Rosenstiel, W. Distributed symbolic computation with DTS. In *Parallel Algorithms for Irregularly Structured Problems, IRREGULAR'95* (Lyon, France, Sep 1995), A. Ferreira and J. Rolim, Eds., vol. 980 of *LNCS*, Springer-Verlag, pp. 231–248.

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

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

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

[15] Decker, T. Virtual data space - load balancing for irregular applications. *Parallel Computing 26* (2000), 1825–1860.

[16] distributed Cilk. *http://supertech.lcs.mit.edu/cilk/home/distcilk5.1.html*.

[17] Foster, I., Kesselman, C., and Tuecke, S. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing 37* (1996), 70–82.

[18] Friedman, R., Goldin, M., Itzkovitz, A., and Schuster, A. Millipede: Easy parallel programming in available distributed environment. *Software: Practice and Experience 27*, 8 (August 1997), 929–965.

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

[20] Grama, A., and Kumar, V. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering 11*, 1 (1999), 28–35.

[21] Habbas, Z., Herrmann, F., Mérel, P.-P., and Singer, D. Load balancing strategies for parallel forward search algorithm with conflict based backjumping. In *Proc. of the 1997 Intl. Conf. on Parallel and Distributed Systems (ICPADS'97)* (Seoul, Korea, 1997), IEEE Computer Society Press, pp. 376–381.

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

[23] Knuth, D. Estimating the efficiency of backtrack programs. *Mathematics of Computation 29*, 129 (Jan. 1975), 121–136.

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

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

[26] Meissner, M., Hüttner, T., Blochinger, W., and Weber, A. Parallel direct volume rendering on PC networks. In *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '98)* (Las Vegas, NV, U.S.A., July 1998), H. R. Arabnia, Ed., CSREA Press.

[27] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)* (2001), ACM, pp. 530–535.

[28] Mueller, F. On the design and implementation of DSM-threads. In *Int. Conference on Parallel and Distributed Processing Techniques and Applications* (June 1997), pp. 315–324.

[29] Namyst, R. *PM2 : an environment for a portable design and an efficient execution of irregular parallel applications*. PhD thesis, Univ. de Lille 1, Jan. 1997. In French.

[30] Randall, K. H. *Cilk: Efficient Multithreaded Computing*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, June 1998.

[31] Rühl, T., Bal, H. E., Benson, G., Bhoedjang, R. A. F., and Langendoen, K. Experience with a portability layer for implementing parallel programming systems. In *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)* (Sunnyvale, CA, 1996), pp. 1477–1488.

[32] Skillicorn, D. B., and Talia, D. Models and languages for parallel computing. *ACM Computing Surveys 30* (1998), 123–169.

[33] Tanenbaum, A. S., and van Steen, M. *Distributed Systems – Principles and Paradigms*. Prentice-Hall, 2002.

[34] Woolridge, M., and Jennings, N. Intelligent agents: Theory and practice. *Knowledge Engineering Review 10*, 2 (1995).

[35] Zhang, H. SATO: An efficient propositional prover. In *Proc. 14th Intl. Conf. on Automated Deduction (CADE-97)* (1997), vol. 1249 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 272–275.

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