Hochschule Reutlingen
Reutlingen University

Parallel and Distributed Computing Group
Department of Computer Science
Reutlingen University

Parallel SAT Solving on Peer-to-Peer Desktop Grids

Sven Schulz and Wolfgang Blochinger

(Accepted Peer-Reviewed Manuscript Version)

The final publication is available at link.springer.com:
https://link.springer.com/article/10.1007/s10723-010-9160-1

BibTeX

```
@Article{Schulz2010b,
	author="Schulz, Sven and Blochinger, Wolfgang",
	title="Parallel SAT Solving on Peer-to-Peer Desktop Grids",
	journal="Journal of Grid Computing",
	year="2010",
	volume="8",
	number="3",
	pages="443--471",
	issn="1572-9184",
	doi="10.1007/s10723-010-9160-1",
	url="https://doi.org/10.1007/s10723-010-9160-1"
}
```

Journal of Grid Computing manuscript No.
(will be inserted by the editor)

Parallel SAT-Solving on Peer-to-Peer Desktop Grids

Sven Schulz · Wolfgang Blochinger

Received: date / Accepted: date

Abstract SATCIETY is a distributed parallel satisfiability (SAT) solver which focuses on tackling the domain-specific problems inherent to one of the most challenging environments for parallel computing — Peer-to-Peer Desktop Grids. SATCIETY efficiently addresses issues related to resource volatility and heterogeneity, limited node and network capabilities, as well as non-uniform communication costs. This is achieved through a sophisticated distributed task pool execution model, problem size reduction through multi-stage SAT formula preprocessing, context-aware memory management, and adaptive topology-aware distributed dynamic learning. Despite the demanding conditions prevailing in Desktop Grids, SATCIETY achieves considerable speedups compared to state-of-the-art sequential SAT solvers.

Keywords SAT Solving · Desktop Grid · Peer-to-Peer · Distributed Systems

1 Introduction

Today's pervasiveness of information technology results in a plethora of exploitable computing power [1]. This has stimulated a new discipline of Grid research, called *Desktop Grid computing* [2]. Desktop Grid computing aims at harnessing idle resources of desktop computer systems for tackling resource intensive problems. Small-scale installations, e.g., comprising the workstations of a department [3], as well as large-scale, Internet-wide approaches [4] have been successfully implemented.

Desktop Grids differ significantly from traditional parallel systems. Particularly, the aggregated resources join and leave the Grid in an unpredictable manner. This phenomenon is called *volatility* [2,5,6] and is much more pronounced than in other kinds of parallel or distributed systems like compute clusters. The nodes of a Desktop Grid are primarily used for other purposes. This non-dedication considerably contributes to the observed volatility. Hence, a Desktop Grid system not only needs to handle occasional error conditions but must be explicitly tailored to cope with a constant flux in resource availability. Another major difficulty is heterogeneity: While cluster nodes are most often virtually identical, nodes of a Desktop Grid are different concerning hard- and software configuration. To complicate matters further, resource usage may be constrained by the host owner. Finally, Desktop Grids are typically operated over WANs, MANETs or the Internet. This causes non-uniform communication costs and often comes with limited connectivity. The latter is typically due to NAT devices and restrictive firewalls. Volatility, heterogeneity, and non-uniform communication costs turn Desktop Grids into one of the most challenging environments for distributed computing. Delivering sustained computing power in this setting poses enormous challenges to system and application designers.

Conceptually, Desktop Grids are either based on a Client/Server or on a Peer-to-Peer (P2P) interaction model. Prominent examples for Client/Server based platforms are Boinc [4] and Entropia [3]. JNGI [7] and P3

Sven Schulz
Institute of Parallel and Distributed Systems
University of Stuttgart
Tel.: +49-711-7816224
Fax: +49-711-7816424
E-mail: sven.schulz@ipvs.uni-stuttgart.de

Wolfgang Blochinger
Institute of Parallel and Distributed Systems
University of Stuttgart
Tel.: +49-711-78164928
Fax: +49-711-7816424
E-mail: wolfgang.blochinger@ipvs.uni-stuttgart.de

[8] by contrast are typical representatives of P2P platforms. The Client/Server approach represents a proven and well understood interaction model. Such systems have reached a considerable degree of maturity and stability. However, due to their centralized organization, parallel programming models requiring complex interaction patterns among the nodes cannot be realized efficiently. This considerably limits the range of applications which can be deployed on Client/Server based Desktop Grids. In contrast, P2P Desktop Grid computing permits direct interaction among arbitrary nodes by leveraging state-of-the-art concepts from the realm of distributed systems. Thus, the P2P approach lays the ground for advanced parallel programming models supporting non-trivial parallelism. However, there is little practical experience so far concerning the question which classes of applications can actually benefit from P2P Grid computing.

In this article, we deal with parallel Boolean satisfiability (SAT) solving on P2P Desktop Grids. Particularly, we present the architecture and design of SATCIETY, which is a parallel SAT solver implemented on top of our P2P Grid platform COHESION [9].

SAT is the problem of finding a variable assignment such that a given Boolean formula evaluates to TRUE, respectively to prove that no such assignment exists. SAT was the first problem shown to be NP-complete [10]. Besides this central role in theoretical computer science, many real world problems could have been tackled in recent years by encoding them as SAT instances. Prominent examples can be found in electronic design automation [11,12,13], artificial intelligence [14], scheduling [15], and cryptography [16]. Despite the tremendous improvements in SAT solving methods achieved in the last decade, there are still unsolved SAT problem instances in all major application fields. Particularly, the ever increasing complexity of chip designs is a source of extremely hard SAT problem instances which are far too complex to be solved by state-of-the-art sequential SAT solvers. In this situation, parallel computing is a promising option to enable further improvements. Our specific goal is to use the massive computational power of Desktop Grids to achieve a significant performance boost. In particular, we make the following contributions:

By combining existing and new methods from the realm of parallel and distributed systems, we were able to realize a distributed task pool execution model that is resilient to a high degree of volatility and forms the conceptual basis for executing non-trivial task-parallel applications.

Our work shows that highly optimized parallel search algorithms like SAT solving are prime examples of parallel applications that significantly take advantage of the unrestricted interaction patterns enabled by the P2P approach to Desktop Grid computing.

Moreover, by enabling parallel SAT solving on Desktop Grids, our article contributes to settling the question which classes of problems can profit from Desktop Grid computing. In particular, we demonstrate that not only embarrassingly parallel problems can be tackled by Desktop Grids.

The rest of this article is organized as follows: In Section 2 we give a brief account of state-of-the-art SAT solving methods and basic parallelization techniques. Subsequent to an overview of SATCIETY's architecture in Section 3, we give a detailed description of SATCIETY's inner workings in Sections 4–8. We report on performance measurements in Section 9 and discuss related work in Section 10. Finally, Section 11 concludes the article by summarizing our contributions and identifying directions for future research.

2 The SAT Problem

2.1 Basic Definitions

The Boolean satisfiability (SAT) problem asks whether one can find a variable assignment for a Boolean formula F such that F evaluates to TRUE.

We assume that F is given in *conjunctive normal form* (CNF). In CNF, a formula is composed of conjunctions ($\wedge$) of *clauses*. A clause is the injunction ($\vee$) of one or more *literals*, and a literal is a variable or the complement of a variable. By convention, variables are numbered consecutively and are represented as x_i with $i \in \{1, \ldots, n\}$. Note that all Boolean formulae can easily be transformed into the CNF representation.

Consider the following Boolean formula in CNF :

$$F = (x_1 \vee x_3) \wedge (x_2 \vee \overbrace{\overline{x_3}}^{literal}) \wedge \underbrace{(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})}_{clause} \wedge x_3.$$

The variable assignment $x_1 \rightarrow$ FALSE, $x_2 \rightarrow$ TRUE, $x_3 \rightarrow$ TRUE represents a satisfying assignment of F. For brevity, one uses literals which are implicitly assumed to be *true* to describe a variable assignment. Thus, the above assignment can be expressed as $(\overline{x_1}, x_2, x_3)$.

A fundamental property of a formula in CNF is that it is satisfiable iff in each clause at least one literal evaluates to TRUE. If for a clause all but one literal have already been assigned to FALSE, the remaining literal must be assigned to TRUE in order to satisfy the clause. Such clauses are called *unit clauses*. A situation when all literals of a clause are assigned to FALSE is called a *conflict*, and the clause is called a *conflicting clause*.

2.2 DPLL Based SAT Solving Algorithms

The original Davis-Putnam-Logemann-Loveland (DPLL) SAT solving algorithm [17,18] still serves as the algorithmic framework of modern complete[1] SAT solvers. However, in recent years sophisticated heuristics have been incorporated which are capable to significantly prune the search space for a wide spectrum of problem instances. Most beneficial advances have been achieved by employing *dynamic learning* and *conflict driven backtracking* [19].

Figure 1 outlines the overall structure of a modern DPLL based SAT solving algorithm (in an iterative form). We will restrict our discussion of the DPLL algorithm to a top-level treatment. A complete description can be found in [20] or [21].

```
while TRUE do
    if (decide()==VARIABLE_ASSIGNED) then
        while (propagate()==CONFLICT) do
            if (current_level==0) then
                return UNSAT;
            else
                new_level=analyze_conflict();
                back_track(new_level);
    else
        return SAT;
```

Fig. 1 Top-level structure of the DPLL algorithm with Dynamic Learning and Conflict Driven Backtracking

Basically, the DPLL algorithm is a search process with backtracking. Partial variable assignments are speculatively extended to find a satisfying assignment. The procedure `decide()` determines according to a heuristic [22,23] which unassigned variable should be chosen next to extend the current partial variable assignment. Each such decision is recorded on an *assignment stack* along with an associated *decision level*. The decision level of the first decision made is 1. The procedure `propagate()` infers additional assignments that are logical consequences of the current partial variable assignment using a technique called *unit propagation*: After making a new decision, some clauses may have become unit clauses, which imply new assignments as explained above. Such deduced assignments are called *implications*. They are also recorded on the assignment stack at the current decision level. Thus, the assignment stack tracks the current state of the search process. Unit propagation terminates when either no unit clauses exist or a conflict occurs. In the first case, a new decision is made starting the next decision level. In the second case, the conflict is analyzed and resolved by the procedure `analyze_conflict()`. Basically, it performs two tasks:

- **Dynamic Learning**: A new clause called *lemma* is constructed by analyzing the reasons for the current conflict. A lemma reflects a minimal subset of the current assignments that implies the conflict. When appended to the input formula, a lemma prevents the search process from reproducing the same conflict in other regions of the search space. Problem clauses and lemmas constitute the *clause database*. There are different schemes for deriving lemmas [24]. However, lemmas are always inferred by resolution and are logical consequences of the clause database. Thus adding a lemma to the input formula does not affect the correctness of the DPLL algorithm. For the same reason, lemmas can be safely removed from the clause database, e.g., for saving memory.
- **Conflict Driven Backtracking**: By construction, a lemma is initially a conflicting clause. The backtracking level is determined as the lowest level at which the lemma becomes a unit clause. Note that at this level the current conflict is also resolved. The procedure `back_track()` releases all assignments recorded on the assignment stack up to the computed backtracking level. The newly added lemma, which is now a unit clause, takes the search to a new direction.

When backtracking reaches decision level 0, the current lemma forces a variable assignment and possibly additional implications at level 0. Such variable assignments are called *top-level assignments*. Since top-level assignments do not depend on any decision, they are a necessary condition for the formula to be satisfied and are fixed for the rest of the search process. As a consequence, a conflict at decision level 0 (*top-level conflict*) cannot be resolved by releasing assignments. In that case the input formula is unsatisfiable. In contrast, if all variables have been assigned without a conflict, the input formula is satisfiable.

2.3 Basic Techniques for Parallel SAT Solving

2.3.1 Parallel Search Process

An important technique for the design of parallel algorithms which search in a space of possible solutions is *exploratory decomposition* [25]. The basic principle of this technique is to split up the search space into several disjoint subspaces to be treated in parallel.

[1] In contrast to *incomplete* solvers, *complete* solvers are able to prove unsatisfiability of a SAT formula.

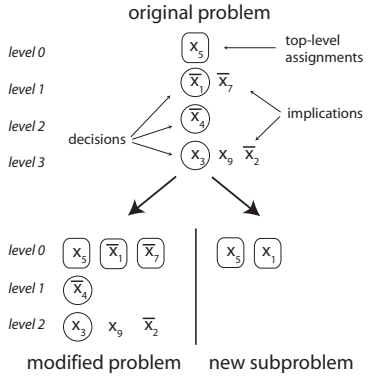


Fig. 2 Exploratory problem decomposition for parallel SAT solving.

We can employ the exploratory decomposition technique for parallel SAT solving by introducing a splitting operation on the assignment stack. Basically, out of the current assignment stack two stacks are generated, each representing a disjoint region of the search space (see Figure 2). One of them is actually a slight modification of the original assignment stack, such that the splitting operation can be carried out without restarting the solving process. The new assignment stack holds all variable assignments from level 0 of the original stack, since they are fixed for the whole search process. For splitting the search space we select the decision variable on decision level 1 of the original stack and include it in the complementary phase in the new assignment stack. Since for this variable now both phases are checked, we can fix the corresponding assignments by moving them to decision level 0 in both stacks. As a consequence, the implications of the selected decision can also be fixed in the original assignment stack. This splitting operation introduced by Zhang *et al.* [26] has become standard for parallel SAT solving on distributed memory architectures (see e.g. [27,28,29]).

While exploratory decomposition yields disjoint subproblems, their individual size cannot be predicted, since the effectiveness of the applied heuristics for pruning the search space can differ considerably for individual subproblems. This induces a high degree of irregularity of the parallel computation. Consequently, dynamic problem decomposition and dynamic load balancing is required in order to avoid significant processor idling. Thus, we must continuously apply the splitting operation and balance the resulting sub-problems among the processors.

The parallel algorithm terminates with the result UNSAT when the outcome of all generated subproblems is UNSAT. If the result of one subproblem is SAT the parallel algorithm terminates with the same result and the computation of all other subproblems is canceled.

2.3.2 Distributed Dynamic Learning

The dynamic learning process of modern SAT solvers relies on accumulated knowledge continuously deduced during the solving process. Employing exploratory decomposition techniques in a distributed-memory setting results in a (partially overlapping) partition of the clause database, consisting of several distributed clause databases. All clause databases comprise the problem clauses and lemmas which have been derived locally. Since dynamic learning can considerably prune the search space, it is crucial to exchange lemmas among the clause databases in order to exploit the full potential of this technique. This establishes a *distributed dynamic learning* process. It is orthogonal to the parallelization of the backtracking search (by exploratory decomposition) and specifically addresses the deduction part of modern SAT solving methods.

Exchanging lemmas synchronously among the clause databases (for example by an SPMD style all-to-all broadcast operation) causes significant processor idling due to the high irregularity of the solving process. Moreover, the total amount of deduced lemmas increases linearly with the number of processors. Thus, a total exchange approach doesn't scale. Consequently, for realizing an effective distributed learning process an asynchronous and selective communication method must by employed.

3 Overall Architecture

Satciety is realized as a three-tiered architecture consisting of the following tiers:

1. The core of Satciety is a **Distributed SAT solver** especially tailored for use in P2P Desktop Grid environments. It is built on top of our versatile P2P Grid system platform Cohesion and our network substrate for high-performance computing Orbweb. The parallel SAT solver implements dynamic problem decomposition based on a distributed task pool which is capable of handling volatility and random faults. For dynamic load balancing, tasks are interchanged in a compressed resource-friendly encoding. Termination detection is accomplished by using a replication-resilient variant of the fixed energy algorithm. The solver realizes a distributed dynamic learning process that adapts to bandwidth-utilization constraints and instance-specific lemma generation rates. To protect host systems from memory overload, Satciety uses a three-stage memory management approach that does not compromise completeness of the solver.

2. A J2EE **Application Server** constituting the middle tier of SATCIETY is responsible for queue management and job preprocessing. The latter includes SAT formula simplification through variable and clause elimination, transcoding to a space-efficient structure-preserving replacement of the commonly used DIMACS format, and efficient instance provisioning over a P2P file sharing protocol.
3. A **Web-based GUI** allows for real-time interaction with SATCIETY over a browser. Multiple users can concurrently submit, monitor, and control their SAT jobs.

The functionality of the first two tiers is described in more detail in the following sections: Section 4 quickly summarizes the features of COHESION and ORBWEB as far as they are relevant in the context of this article. Section 5 motivates the necessity for and breaks down the functionality of SATCIETY's distributed task pool. The anatomy of a SAT task and related optimizations are presented in Section 6. Section 7 delineates the concept of topology-aware adaptive lemma exchange. Section 8 is dedicated to the description of the preprocessing pipeline operated by the middle tier application server.

4 P2P Grid System Foundations

4.1 Cohesion

COHESION is our modular system platform for P2P Desktop Grid computing. It provides essential services for building sophisticated churn-resistant and fault-tolerant applications that are highly integrated with the host system to ensure an unobtrusive coexistence with user processes. An in-depth treatment of the system architecture can be found in [9].

COHESION provides group membership management [30], a failure detection service, and a highly configurable application container [31]. Group membership management is essential for almost all distributed algorithms as it provides the ability to organize the set of participating nodes into dynamic functional subsets and to make nodes within a group visible to each other according to an application-specific topology. Failure detection is orthogonal to group membership management. It allows to determine whether a node left a group intentionally or as a consequence of failure. *Membership views* [32] abstract from concrete group management and failure detection mechanisms and expose the current set of known non-faulty peers to the application. Thus, a membership view reflects the availability of group members. The application container of COHESION controls the lifecycle of an application based on arbitrarily complex combinations of environmental conditions, like system load, time of day, or mouse movement. As described in Sections 5 and 6, we leverage all these services to implement SATCIETY's distributed taskpool.

4.2 Orbweb

ORBWEB [33] is our network substrate for high-performance computing in Peer-to-Peer Grids. It is used as the underlay for COHESION and belongs to the class of unstructured hybrid P2P networks. In contrast to pure P2P, the hybrid approach is characterized by the fact that part of the network functionality is delegated to a small number of distinguished peers usually called *superpeers*. More precisely, ORBWEB delegates group management and failure detection to superpeers, thus allowing for rapid membership view updates, which is essential for achieving good efficiency for many distributed algorithms.

ORBWEB builds on and extends the open *eXtensible Messaging and Presence Protocol* (XMPP) [34]. It supports direct inter-peer connections that are established based on traffic pattern analysis. Furthermore, a probabilistic topology-aware decentralized groupcast implementation with superpeer resource usage that is constant with respect to the size of the group is provided. Improved protocol efficiency is achieved by a standardized binary encoding of XMPP messages. As substantiated by a detailed experimental analysis, these optimizations significantly improve the applicability, the performance, and the scalability of ORBWEB.

The key feature of ORBWEB with respect to SATCIETY is, that it offers fine-grained control over the composition of membership views via interchangeable *view managers*. As described in Section 7, SATCIETY deploys a custom view manager to realize topology-awareness for efficient lemma exchange.

5 Distributed Task Pool

Exploratory decomposition structures a parallel program execution into a set of interacting tasks. Each of these tasks consists of a sequence of operations that can include the creation of new tasks which may later be transferred to a different compute node for load-balancing. A *task pool* is a shared data structure to store and manage the tasks created during execution. All compute nodes have access to the task pool such that they can extract tasks for execution and enqueue newly created tasks in case the currently executing task dynamically creates new subtasks. A task pool can be or-

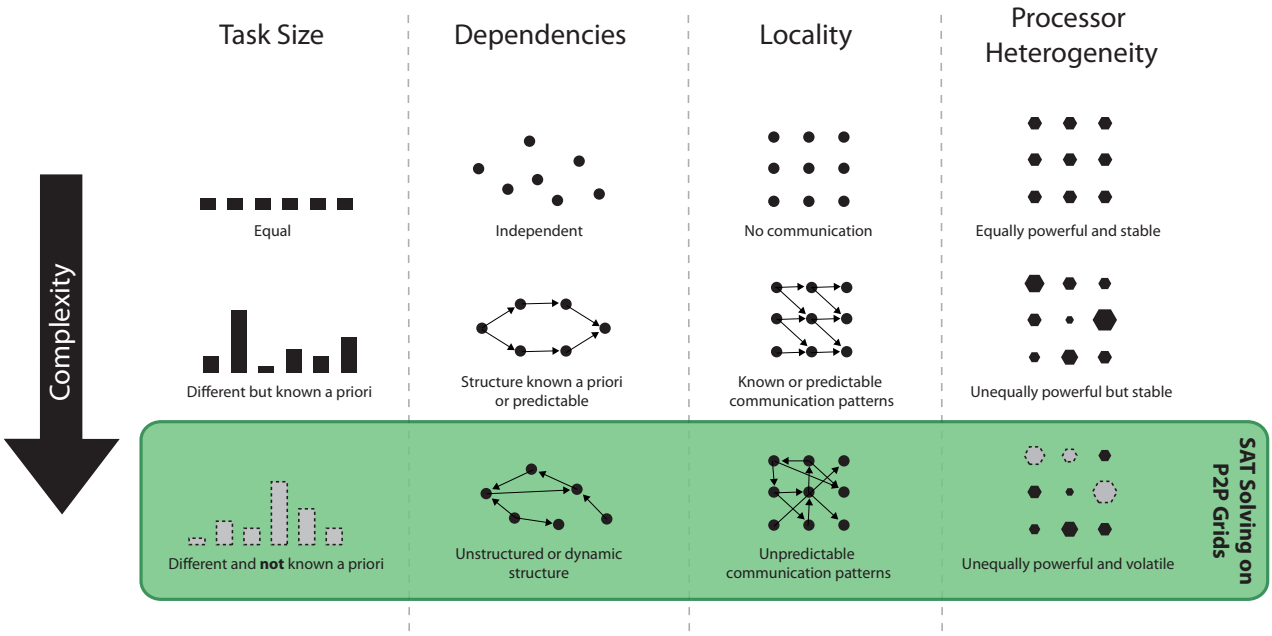


Fig. 3 Spectrum of problem and system properties for task-parallel applications.

ganized in a centralized or in a distributed manner. In a centralized approach — used by many existing state-of-the-art parallel SAT solvers (see Section 10) — a master node maintains a global task queue from which idle processors can fetch tasks. The queue is refilled with tasks that result from worker-side problem decomposition triggered by the master whenever the global task queue becomes empty. A major drawback of centralized pools is that the master node is a sequential bottleneck as tasks must be transferred from and to the master node. This limitation is especially impedimental in the case of SAT solving as a task description may become very large for real-world SAT instances with millions of variables. As the underlying communication platform supports direct Peer-to-Peer exchange of tasks, SATCIETY employs a distributed task pool model, in which a pool is located on every compute node. The downside of this decentralized approach is that the implementation of essential aspects of the control logic become much more challenging. These aspects are load balancing, fault tolerance, and termination detection. Each one is discussed in detail in the following sections.

5.1 Load Balancing

The selection of an appropriate load balancing strategy is governed by the properties of the problem to be solved and to the same extent by the environment the computation is executed in. These properties are task size, inter-task dependencies, communication locality and processor heterogeneity (see Figure 3). Parallel SAT solving on Desktop Grids is characterized by unpredictable task sizes, inter-task dependencies with a structure that changes dynamically, arbitrary communication patterns and a dynamically changing set of compute nodes potentially experiencing concurrent loads. Hence, it belongs to a very demanding class of task-parallel applications.

For this kind of applications Blumofe and Leiserson [35] have shown that a simple randomized algorithm is optimal with high probability. Well-known representatives of these algorithms are *random stealing* and *random pushing*. While with the former idle nodes pull or steal tasks from the queue of randomly selected neighbors, the latter is the opposite approach where nodes with excess tasks push some of them to randomly selected nodes. We use random stealing as it is demand-driven and thus avoids unnecessary transferal of the potentially large SAT tasks.

The algorithmic aspects of problem decomposition for parallel SAT solving have been discussed in Section 2.3. Pull-based load balancing and exploratory decomposition together solve the problems associated with performance heterogeneity across processors as more powerful hosts dynamically split off tasks from less powerful ones when they become idle.

In the context of load balancing another important question is, when to trigger task decomposition. In SATCIETY we prefer on demand task creation rather than

proactive problem decomposition. With *on-demand decomposition* we can limit undesirable growth of the effective search space caused by needless decomposition operations. As described in Section 2.2, a newly learned lemma prevents the search from making the same unprofitable work over and over again in other parts of the search space. By splitting tasks eagerly, the newly created task cannot profit from this extra knowledge, at least if it is transferred to another node for execution. Consequently, part of the search space will be traversed more often than necessary, resulting in the mentioned growth of the effective search space. Technically on-demand decomposition is realized by having the decomposition component listening for incoming balance requests emitted by remote load balancers. On arrival of such a request, the SAT solver is instructed to asynchronously split off and enqueue a new task that is finally dequeued, marshalled and sent to the requesting node by the load balancing component.

5.2 Fault Tolerance

As discussed initially typical Desktop Grid environments are highly volatile and exposed to increased node and network failure probabilities. Due to its high irregularity, parallel SAT solving results in tasks with execution times ranging from seconds to days. Relaunching the whole computation in the probable case of failure is no option. Thus, SATCIETY employs a fault tolerance algorithm that tracks tasks over their entire lifespan until they are finished or canceled. Using ORBWEB as the underlying communication substrate for COHESION, our fault tolerance algorithm assumes an environment that

1. ensures **atomic message transmission**, i.e., a message is delivered completely or not at all.
2. is **asynchronous**, as there is no bound on message delays, clock drift, and the time necessary to execute a step. As ORBWEB maintains only a limited number of direct peer-to-peer connections selected based on traffic analysis, messages may even get lost with small probability in case one of these connections is displaced and thus closed while a message is in transit.
3. belongs to the class of **crash-fault** systems, as crashed ORBWEB nodes recover with a new identity and thus logically never come back.
4. has a **perfect failure detector** [36]. In this context, *perfect* means that no process is suspected before it crashes (*strong accuracy*) and eventually every process that crashes is permanently suspected by every non-faulty process (*strong completeness*). This is accomplished by defining aliveness as being

P1a: When a task t_j is created on node p_i.
begin
  send UPDATE(t_j, **NULL**, p_i) to p_c
end

P1b: When a task t_j is finished on p_i.
begin
  send UPDATE(t_j, p_i, **NULL**) to p_c
end

P1c: When a task t_j is split on p_i creating a new task t_k.
begin
  send UPDATE(t_k, **NULL**, p_i) to p_c
  send UPDATE(t_j, p_i, p_i) to p_c
end

P2: When p_i receives a BALANCING-EVENT(t_k, p_j) indicating that task t_k is going to be transferred to p_j.
begin
  send UPDATE(t_k, p_i, p_j) to p_c
end

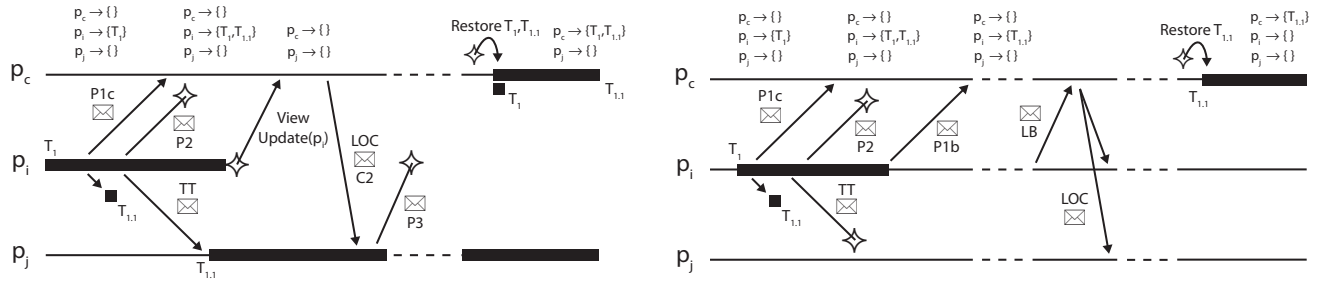
P3: When p_i receives a LOCATE(t_j) message.
begin
  if $t_j \in$ *local task queue* **then**
    send UPDATE(t_j, **NULL**, p_i) to p_c
  end
end

C1: When p_c receives an UPDATE(t_k, p_i, p_j).
begin
  if $p_i \neq$ **NULL then**
    CANCELRESTORATION(t_k)
    $location(t_k) =$ **NULL**
  end
  if $p_j \neq$ **NULL then**
    $location(t_k) = p_j$
    if $p_j \notin View(p_c)$ **then**
      SCHEDULERESTORATION(t_k, T)
    end
  end
end

C2: When p_c receives a VIEWUPDATE($p_j \notin View(p_c)$).
begin
  foreach t_i : $location(t_i) = p_j$ **do**
    send LOCATE(t_i) to all $p_i \in P$ by groupcast.
    SCHEDULERESTORATION(t_i, T)
  end
end

C3: When p_c receives a BALANCING-REQUEST(p_j) from p_j.
begin
  execute **C2**(p_j)
end

Fig. 4 SATCIETY's task tracking algorithm ($location(t_k)$ contains the current location of a task t_k, SCHEDULERESTORATION(t_i, T) (re-)schedules the restoration of task t_i to start after a configurable amount of time T on p_c, CANCELRESTORATION(t_i) cancels a previously scheduled restoration for a task t_i).



(a) Scenario where a task is duplicated.

(b) Scenario where rule **C3** is required for restoration of a lost task.

Fig. 5 Interesting task restoration scenarios for SATCIETY's fault tolerance scheme ($T_s \hat{=}$ Task with symbolic identifier s, $T_{s.t} \hat{=}$ subtask t of task T_s, TT $\hat{=}$ Task Transfer, LB $\hat{=}$ Load Balancing, LOC $\hat{=}$ Locate).

connected to the ORBWEB superpeer. As soon as the XMPP connection between a node p_i and the superpeer is closed by either side intentionally or caused by failure, the departure of p_i is communicated as a VIEWUPDATE(p_i) event to all nodes that are still alive, i.e., connected to the superpeer.

Figure 4 shows the algorithm as a set of event-driven actions. We assume that the system consists of a dynamic set of processors P. From the time when the initial task is submitted on the coordinator, there are eight events that may trigger an action. Three of them (**C1**-**C3**) are relevant to a stable coordinator node $p_c \in P$ which keeps record of the locations of all unprocessed tasks and is responsible for their restoration in case of failure. A change of the location of a task is communicated to the coordinator through UPDATE($task$, p_{source}, p_{target}) messages. While $p_{source} = null$ means that the specified task is created on node p_{target}, either as the first task of a new job or as the result of a splitting operation, $p_{target} = null$ means that the task has been processed on p_{source} and is thus removed from the task pool.

P1a is executed when the initial task is submitted and when a task is restored by the algorithm on node p_c. **P1b** is invoked when a task has been finished and thus is removed from the local and consequently from the distributed task pool. When a splitting operation creates a new task, **P1c** is executed, which actually consists of two individual update operations: First, an UPDATE message for the newly created task is sent to the coordinator. Second, the task from which the new task has been split off is updated. While omitting the latter step does not affect correctness of the algorithm, it may severely impact performance in case the parent task is lost subsequently, as then all the work – including that split off – has to be done again, which could be the whole job in the worst case. Note the fact that if the update for the original task is correctly reported but the UPDATE message for the new task gets lost, a crash of the splitting node would be unrecoverable. Thus, it is indispensable to guarantee that either both or none of the updates are performed. As ORBWEB guarantees that message delivery is atomic, this requirement can be satisfied easily by using a single message for transmitting both updates. **P2** is executed just before a task is transferred to a new node as part of a load balancing effort.

C1 is executed by the coordinator on receipt of an UPDATE message. There are three different possible scenarios: (a) The task is unknown, i.e., has been newly created. This is a result of some node p_j executing **P1a**. (b) The task is being migrated to a new node, which is signaled by some node p_j executing **P2**. (c) The task should be deleted as it has been finished by some node p_j consequently executing **P1b**. In every case the coordinator updates the location for the respective task accordingly.

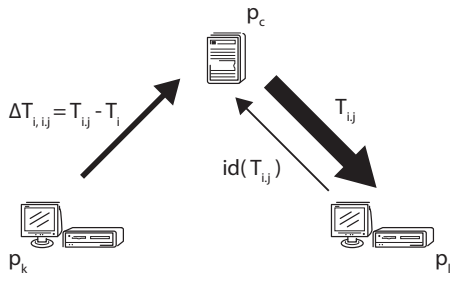
C2 is executed when a node departs either deliberately or by failure. SATCIETY relies on the group membership and fault detection services provided by the underlying COHESION platform to create the triggering VIEWUPDATE events. On receipt of such an event the coordinator might assume that the tasks located at the vanishing node are lost. However, this is not necessarily true as the underlying communication system is asynchronous, which means that an UPDATE message sent by the vanishing node on task transferal may be still in transit. Hence, the restoration of the task is delayed for a specified period T that is large compared to the average round-trip time in the underlying network, which can be in the order of seconds in typical networks spanned by Desktop Grids, like MANs or WANs. On receipt of a delayed UPDATE message the scheduled restoration is canceled as part of the execution of **C1**.

While losing tasks is effectively prevented by SATCIETY, task duplication is not. This is not critical as performing work twice does not affect correctness in

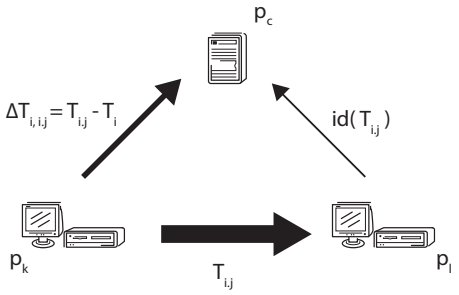
the context of SAT solving, however it affects performance. Figure 5a depicts a scenario where a task $T_{1.1}$ is duplicated. While the task transferal message from p_i to p_j containing $T_{1.1}$ reaches its destination, the update emitted to p_c by execution of rule (**P2**) gets lost. When p_i crashes subsequently, the coordinator incorrectly assumes $T_{1.1}$ was still at p_i and triggers its restoration. To reduce the probability of such *false positives*, the coordinator delays restoration of the task and emits a LOCATE request to all nodes in the system first. If the task is present in the task queue of a node, it responds with a corresponding UPDATE message (**P3**) so that the coordinator can update its location and cancel its restoration. If no node responds within a given period or the response gets lost as depicted in Figure 5a the coordinator assumes that the task is actually lost and restores it.

There are scenarios in which a lost task won't get restored without an additional rule **C3**. Consider the situation depicted in Figure 5b: while performing a load balancing operation both the UPDATE message sent by a source node p_i to the coordinator p_c in **P2** and the task transferal message carrying the task $T_{1.1}$ sent to the target node p_j get lost. When p_i remains stable, i.e. never leaves the grid, the coordinator would never try to restore the task as **C2** would never be executed. To prevent such situations SATCIETY exploits the fact that a node will emit balancing requests only if its local task queue is empty. Hence, the coordinator can infer that $T_{1.1}$ is not in p_i's task queue any more when receiving a balancing request from p_i. In case no new location has been reported for $T_{1.1}$, the coordinator assumes the task is lost and schedules its restoration in case a location attempt – performed to reduce false positives as described above – remains ineffective.

The actual structure of the UPDATE messages is omitted from the algorithm in Figure 4 for simplicity. In fact they never carry the full set of top-level assignments constituting a task (see Section 2.3.1): The initial task, which has an empty set of top-level assignments, is submitted on the coordinator allowing **P1a** to be performed locally without external communication. When the **P1b** (finished task), **P2** (transfer from one node to another), and **P3** are executed, the task has been sent to the coordinator previously. Thus, it is sufficient to transfer a unique identifier attached to each task on creation that can be used by coordinator to locate the associated task within the task location table. For the two remaining updates performed in **P1c**, SATCIETY exploits the fact that the top-level assignments for both tasks resulting from a splitting operation can be composed from the top-level assignments of the original task, the assignments done up to the moment the split



(a) Centralized task queue with implicit fault tolerance. All communication including task transferals is relayed by the coordinator.



(b) Decentralized task queue with task tracking-based fault tolerance. Bulk transfers are performed over P2P connections.

Fig. 6 Comparison of the data flow for a centralized and distributed task queues in case of a splitting operation for task T_i on node p_k with subsequent task transfer to and completion of task $T_{i.j}$ on another node p_l ($\Delta T_{i,i.j}$ is the delta between the tasks T_i and $T_{i.j}$, $id(T_{i.j})$ is the unique task identifier of task $T_{i.j}$).

is performed, and the split literal (cp. Figure 2). As the coordinator already knows the top-level assignments of the original task, it is not necessary to include them in the UPDATE messages. Together these measures significantly reduce the load on the coordinator that would otherwise become a bottleneck.

Figure 6 depicts the data flow after a splitting operation for a centralized task queue used by previous Desktop Grid SAT solvers [27] and for SATCIETY's fault-tolerant distributed task queue approach. In SATCIETY, the bulk of data is transferred in a peer-to-peer fashion eliminating the overhead and enhancing scalability compared to taking the indirection over the central task queue.

5.2.1 Deliberate Host Departure

Although the fault tolerance mechanism described above ensures correctness by restoring the tasks discarded when a node deliberately leaves the grid, the resulting inefficiency is unacceptable. In SATCIETY, the local task

pool is implemented as a COHESION application that is shutdown in a controlled way by the platform as soon as user and/or application defined conditions no longer hold [9]. As part of this shutdown sequence the tasks in the local task pool are offloaded to neighbor nodes that are selected at random. As SAT tasks are potentially large (see Section 6), this process may take a very long time and the user experience may be impaired. For that reason the offloading process is aborted after a configurable period of time. Those tasks that could not have been offloaded to other nodes are discarded and thus are effectively lost and subject to restoration by the fault tolerance mechanism. The same is true for tasks that are sent successfully but for which the receiving node departs or crashes before the task has been completely transmitted.

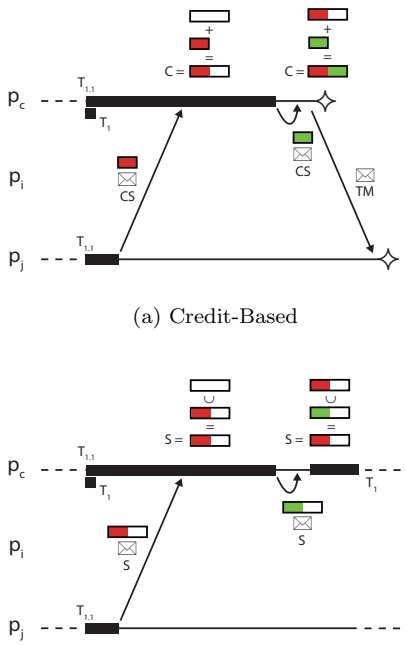
5.3 Termination Detection

In asynchronous distributed systems with message-based communication the detection of termination is a non-trivial problem as there is neither global time nor full knowledge of the global state available. A system is considered to be globally terminated, if every participating task is terminated and no potentially activating messages are in transit.

Today, there is a large body of research concerning termination detection algorithms [37,38]. Many of them make restrictive assumptions concerning the underlying model of computation. The most general algorithms consider diffusing computation models in which one process initiates the computation and dynamically launches subcomputations on remote nodes by sending task transferal messages. The resulting activation graph is a tree and is used in *parental responsibility algorithms* to detect termination by continuously tracking process lifecycles. In contrast, *wave-based* algorithms periodically propagate waves of control messages throughout the network.

More recent algorithms are applicable for completely asynchronous communication models where messages may arrive out-of-order or may be delayed for arbitrary but finite time [39,40,41,42]. While the most general of these algorithms support dynamic environments, many of them are not ready for use in real applications as they are restricted to scenarios where processors may be created but not destroyed [43,44,45] or require a node to participate in termination detection after it has been destroyed [46]. Both assumptions obviously do not hold for highly volatile Desktop Grids.

The algorithm by Mattern [47] is applicable for completely asynchronous systems and supports both process creation and destruction. Every task is associated



(a) Credit-Based

(b) Spectra

Fig. 7 Comparison of the behavior of Mattern's *credit-based* and SATCIETY's *spectra* termination detection algorithm in case a task is spuriously restored by SATCIETY's fault tolerance (e.g., after a scenario as depicted in Figure 5a). While the simple scalar arithmetic of the former is insufficient to handle credit share duplication correctly and hence announces termination prematurely, our algorithm tolerates such conditions as intervals additionally encode parent/child relations among the tasks.

with a certain amount of credit. Whenever a new task is created, half of the tasks credit is subducted and given to the newly created node. On termination of a task the associated credit is sent back to the initiating process. When the accumulated credit at the initiator equals the credit that was given to the initial task, the computation has terminated. Unfortunately, this credit-based algorithm is not resilient to task duplication. As both the original task as well as its duplicate carry the same credit, the invariant of the credit algorithm

$$\sum_{t_i \in T} C(t_i) = C(t_0),$$

is violated, where T is the set of tasks created so far, $C(t_i)$ is the credit associated with task t_i, and $C(t_0)$ is the credit assigned to the initial task t_0. This inevitably results in premature termination. This inadequacy is a direct consequence of using scalar values as credits (see Figure 7a).

To remove this vulnerability, we conceived a variant of the credit algorithm that uses credit intervals instead of scalar credits. As the credit algorithm is also known under the alias *fixed energy* termination detec-

tion algorithm and a collection of energy levels is called a *spectrum* in physics, we refer to our variant as the *spectra* termination algorithm: When a new task is created the spectrum of the original task is cut in half. While one spectrum remains with the original task the other is assigned to the newly created one. The initiator of the computation maintains a set of disjoint spectra and processes incoming spectra by computing the union of the new and the already available spectra. As depicted in Figure 7b, the effect of premature termination announcement caused by task duplication described above is prevented by using intervals.

An additional obstacle to reliable termination detection is the fact that messages may be delayed arbitrarily and may even get lost. Consider the case that upon completion of a task the UPDATE message send in rule **P1b** is delivered correctly, but the associated spectrum gets lost in transit. As the task won't be rescheduled by SATCIETY's fault-tolerance mechanism, the job's termination — albeit it has been completely processed — will never be detected. SATCIETY deals with this problem by piggybacking spectra on the UPDATE messages indicating task completion. Thus, either spectra delivery and removal from task tracking are performed together or not at all. As the latter results in redundant task restoration, SATCIETY retransmits the aforementioned UPDATE message until it is acknowledged by the coordinator.

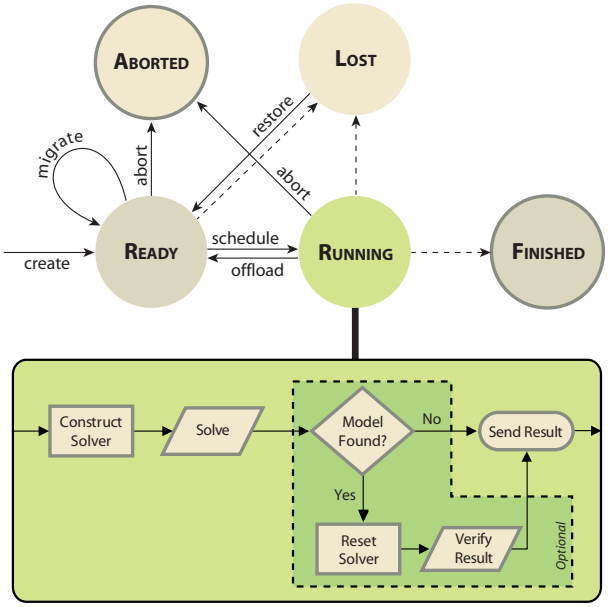


Fig. 8 Lifecycle of a SATCIETY task.

6 SAT Task Anatomy

The basic anatomy of a SATCIETY SAT task consists of:

1. A **reference to the input formula**, which is a *uniform resource locator* (URL) provided by the frontend as part of the provisioning process (see Section 8).
2. The potentially empty set of **top-level assignments** ϕ_0 defining the solver's initial state (see Section 2.3.1).
3. Additional **arguments** specified at job submission. This includes whether to suppress task decomposition in order to force the solver to operate in sequential mode, whether to perform lemma exchange, and a timeout to prevent the solver from running indefinitely long for extremely hard instances.

In addition to these core elements the components of the distributed task pool (see Section 5) can augment tasks by *attachments*. Examples for attachments are unique task identifiers used for fault tolerance, the drain address for spectrum messages sent by the spectra termination detector, and the node to deliver task- or job-related signals to, like termination when a node found a solution.

As depicted in Figure 8, a task's lifecycle consists of its creation by initial job submission or decomposition, possibly several migrations, execution and finally its completion. Tasks may get lost and eventually restored as described in Section 5, or aborted as a consequence of user-initiated job cancellation or expiration of the timeout specified on job submission. The transition from RUNNING to READY updates the top-level assignments part of the task according to the current state of the solver and is triggered on deliberate host departure (see Section 5.2.1). The workflow executed in the RUNNING state is depicted in the lower part of Figure 8 and basically consists of creating a solver instance for the given formula, executing the solver on the arguments as specified by the task, and sending the result back to the *gateway*, i.e., the peer where the job has been submitted. SATCIETY can be configured to execute an optional verification step (region surrounded by dashed line in Figure 8) in case a solution is found. Although this step is functionally unnecessary, it is a valuable tool for identifying bugs in a highly complex distributed solver implementation like SATCIETY. For this purpose the previously created solver instance is reset and reused. In case the verification process fails, an error signal is sent to the gateway.

6.1 Task compression

As the solving process evolves, the top-level assignments quickly become the largest part of a task. Its size is in the order of the number of variables which can easily exceed a million for real-world instances. Simply using integer arrays to encode the top-level assignments results in tasks sizes in the order of megabytes. This is prohibitive in the context of Desktop Grid applications. Thus, SATCIETY employs a more sophisticated two-stage strategy: First, we look at a compact binary encoding of the top-level assignments. For the set V of variables, we need

$$|Lit|_2 = \lceil \log_2 |V| \rceil + 1$$

bits to binary encode a literal and hence

$$|\phi_0|_{sparse} \approx |Lit|_2 \times |\phi_0|$$

bits to encode the top-level assignments by concatenating the encoded literals. We call this encoding *sparse*. In contrast a *dense* encoding is performed by encoding every literal between the lowest and the highest literal of the top-level assignments using 2 bits each. This results in a size of

$$|\phi_0|_{dense} \approx 2 \times (max\{i : lit_i \in \phi_0\} - min\{i : lit_i \in \phi_0\}).$$

SATCIETY's task encoder computes both $|\phi_0|_{sparse}$ and $|\phi_0|_{sparse}$ and encodes the top-level assignments using the encoding yielding the more compact representation. As a second step SATCIETY uses GZIP compression to further reduce the size of the task.

6.2 Memory Management

The DPLL algorithm with dynamic learning adds a new clause to the clause database whenever a conflict occurs. To avoid running out of memory, modern SAT solvers use heuristics to estimate the usefulness of clauses for the future solving process. Based on this estimation they decide which clauses should be deleted. One popular heuristic is to periodically delete those clauses that have not been occurred in a conflict clause for a certain time. To guarantee termination such solvers gradually increase periods between deletions. Hence, the solvers memory footprint is constantly growing, eventually resulting in the system running out of memory. In the case of SATCIETY this behavior is prohibitive for two reasons: First, with SATCIETY allocating a great deal of the overall system memory, other user processes are swapped by the operating system, which drastically reduces responsiveness in case the user reclaims the system. Second, the Java Virtual Machine is killed when no more memory is available, preventing the host from further participation in the parallel solving process. To circumvent these unacceptable scenarios, SATCIETY employs a three-stage memory management approach that enforces memory limits while preserving completeness. The stages are:

1. **Application Control**. SATCIETY leverages the fine-grained application lifecycle control of COHESION. SATCIETY is allowed to run only as long as the amount of free physical memory is above a given threshold $M_{Shutdown}$. In case free memory falls below this threshold, local tasks are off-loaded to other peers and the SATCIETY application is shutdown freeing all memory used by the solver.
2. **Stimulated Reduction**. When the amount of free physical memory drops below a threshold $M_{Stimulate} > M_{Shutdown}$ SATCIETY instructs the solver to reduce the clause database. This reduction is equivalent to reductions triggered by the clause removal heuristic described above and executed as part of the regular solving process.
3. **Forced Reduction**. Removing enough clauses to comply with memory constraints by stimulated reduction is not always possible, since clauses are locked when they are participating in the current backtracking branch by being the reason for a variable assignment [48]. In case free memory is below a threshold M_{Forced} with

   $$M_{Stimulate} > M_{Forced} > M_{Shutdown}$$

   after a stimulated reduction has been performed, SATCIETY backtracks to level 0 and triggers a stimulated reduction. Backtracking rewinds the assignment stack which unlocks additional clauses that may now be safely deleted by stimulated reduction.

To guarantee termination SATCIETY splits off a new task after performing a stimulated or forced reduction. As described in Section 2.3.1 splitting fixes the first decision variable on level 1 in the original task and adds the variable in the opposite phase to the assignment stack of the split off task. Thus, performing the split ensures progress and eventual termination.

7 Topology-Aware Distributed Dynamic Learning

Dynamic learning by conflict analysis has become standard for sequential SAT solvers and tremendously im-

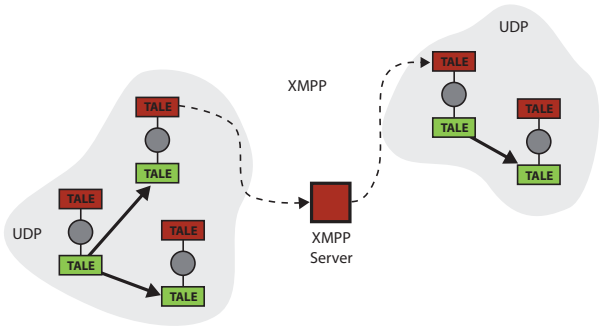


Fig. 9 SATCIETY's Topology-aware lemma exchange employs two ALEFs (see Figure 10) to realize higher exchange rates within (light green) than between network components (dark red)

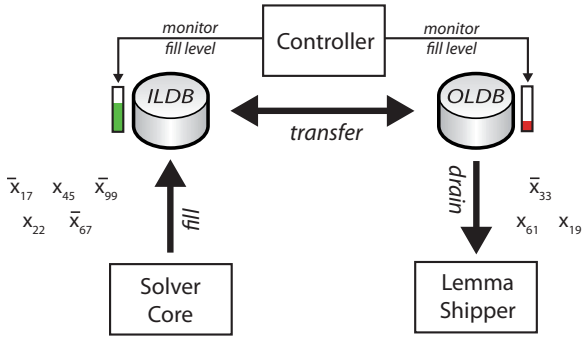


Fig. 10 SATCIETY's adaptive lemma exchange facility (ALEF) implements a demand-driven strategy that maintains a steady lemma output rate by dynamically regulating the lemma output of the solver core.

proves their performance. By exchanging lemmas between solver cores, a similar effect is aimed for parallel and distributed SAT solvers. Although a thorough theoretical investigation and comprehensive experimentations are still pending, the beneficial effect of sharing comparatively short lemmas is indisputable as they have high potential to help in pruning the search space while at the same time are cheap to transmit.

A distinguishing feature of Desktop Grid environments is the fact that in general a node is able to directly exchange messages with only a limited number of other nodes. While communicating with the rest of the system is often enabled by relaying, this limited (direct) connectivity introduces inhomogeneous communication costs that have to be reflected in application level communication patterns to avoid inefficiency and overload. Such protocols are called *topology-aware*.

In contrast to existing Grid SAT solvers [49,50], lemma exchange in SATCIETY is topology-aware. For this purpose, the system classifies each neighbor node — all nodes within the view of a node — as either direct or indirect. Neighbor classification is performed by leveraging a special view manager provided by ORBWEB that has been originally developed for topology-aware broadcast algorithms [33]. The view manager detects the components of the network, i.e., maximum sets of nodes with mutual direct neighbors. As depicted in Figure 9, lemmas are exchanged at a high rate within and at a lower rate between components. While SATCIETY uses UDP for the former because of its small overhead, lemmas exchanged between components are transmitted in-band over XMPP connections with messages relayed by the ORBWEB superpeer.

The number and average size of deduced lemmas is heavily dependent on the concrete SAT instance. Straightforward solutions using hard-coded size limits thus yield unsatisfactory results. Hence, SATCIETY makes use of an adaptive approach to ensure that a predefined exchange rate is sustained but not exceeded. This is of particular importance in the Desktop Grid context, as the user experience may be impaired by excessive communication. Figure 10 illustrates the function of the *Adaptive Lemma Exchange Facility* (ALEF) which is instantiated twice by SATCIETY to implement topology-awareness as described above. The control logic continuously monitors two lemma databases each with a fixed capacity. While the *inbound lemma database* (ILDB) is filled with lemmas produced by the solver core, the *outbound lemma database* (OLDB) is drained by the lemma shipper, which marshals extracted lemmas and sends them to the target node. Every time the OLDB becomes empty or the ILDB becomes full, the controller swaps the content of the databases. If both databases are constantly filled near their capacity limit, the lemma production rate is obviously too high. In this case the controller instructs the solver core to produce less lemmas by decreasing the maximum length of lemmas that are exported to the OLDB. Although the mechanism could have been implemented with a single DB filled and drained concurrently, the two database strategy decouples solver core and shipper, which is of particular importance as conflict clause generation is part of the main loop of solvers based on the DPLL algorithm. The database swap mentioned does not involve copying of the lemmas but is implemented by simply swapping references.

8 Instance Provisioning

SAT instances encoding real world problems are often very large, particularly in the context of formal verification. Figure 11 shows the cumulative histogram of file sizes of the DIMACS [51] encoded instances of the *SAT*

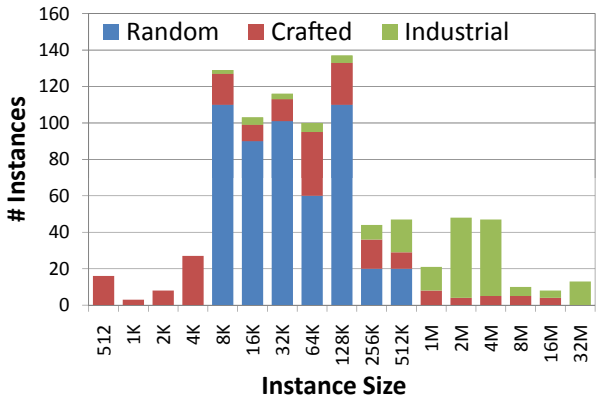


Fig. 11 Histogram of DIMACS encoded file sizes for 877 instances of the SAT Competition 2007.

Competition 2007 [52]. Most instances from the industrial category are larger than 512 KB with a cluster between 512 KB and 4 MB ranging up to over 40 MB. Instances from last year's *SAT Race* [53] were even up to 145 MB large. Although there is in general no correlation between the size of a formula and the runtime of the solving process, in practice, one can often observe such a relation in sequential SAT solving – at least for problems with similar structure. Hence, reducing the size of formulae is a promising approach to shorten solving times and thus has become subject of active research recently. In our case of parallel SAT solving in a Desktop Grid scenario, limited bandwidth is another important reason to keep instance encodings as compact as possible. For the largest formula from above (47 MB) and the grid comprised of 40 nodes used in our performance evaluation in Section 9), a total of approximately 2 GB of data has to be transmitted only to deliver the formula to the nodes. For these reasons SATCIETY employs an extensible preprocessing pipeline to ensure that formulae are delivered as fast as possible to the solver instances running on the participating compute nodes. This pipeline is installed and executed on the frontend node (see Section 3). It consists of five stages and can be extended easily. These stages are: decompression, simplification, transcoding, compression and P2P provisioning. Each stage maintains a job queue so that the stages can be executed in parallel for different instances. The decompression and compression stages are quite self-explanatory: They use the GZIP algorithm to decompress the incoming formula and to compress the transcoded formula respectively. The remaining stages are explained subsequently.

```
c SAT07-Contest Parameters:
c unif2p p=9 nbc2=228 nbc3=2052
c v=630 seed=702278147
p cnf 630 2280
-464 -204 0
-134 384 0
-456 446 0
...
-39 45 -225 0
295 516 337 0
```

Fig. 12 An example formula in DIMACS format from the random category of the SAT Competition 2007 consisting of a preamble (comments and problem line giving the number of clauses and variables) and the clauses in CNF. Negation of a variable is denoted with a minus character. Clauses are terminated with 0.

8.1 Formula Simplification

Recently, advanced preprocessing techniques have been introduced which are applied to the formula before starting the actual solving process. They focus on deriving unit clauses, implications and equivalent literals [54]. Their application on a given formula may result in a reduction of the number of literals, variables, and clauses, which generally yields smaller file sizes and may also reduce the runtime of the SAT solver. SATCIETY uses SATELITE by Eén and Biere [55] in the simplification stage of the preprocessing pipeline. As a lightweight preprocessor SATELITE does not add significant overhead to the solving process (see Section 9.3).

8.2 Transcoding to Binary CNF

DIMACS files are plain ASCII files structured as depicted in Figure 12. While textual data encodings do not suffer from byte order differences and thus are in principle well suited as a file format, their space efficiency is low. For example a literal 3.456.789 produces a 7 Byte text encoding, while a binary representation consumes only 22 Bit or 3 Byte if byte alignment is enforced. Thus, SATCIETY transcodes DIMACS encoded formulae to a structure preserving binary encoding we call *Bit-Packed Binary CNF* (BCNF).

SAT instances are very different with respect to the number of variables and the length of clauses: While a hard random instance may have only a couple of variables and rather long clauses, formulae resulting from real-world problems often have millions of variables and very short clauses. Thus, our BCNF encoder first looks at the number of variables (given in the DIMACS preamble) to compute the number of bits $|Lit|_{BCNF}$ necessary to encode a single literal, which is

$$|Lit|_{BCNF} = \lceil \log_2 |V| \rceil + 1,$$

where V is the set of variables. While clauses are terminated by the value 0 in DIMACS, BCNF writes a variable length header in front of each clause indicating the number of literals the clause consists of. This way memory can be allocated before reading the clause which helps avoid unnecessary copying. The first two bits of the header are used to indicate the length of the header resulting in a possible maximum clause length of 2^{8n-2} for an n-byte header. Note the fact that BCNF could have been designed to be even denser, if structural modifications like variable or clause reordering would have been applied. We have refrained from such optimizations for two reasons: First, downstream GZIP compression would have partially leveled out potential savings. Second, reordering heavily influences the runtime of SAT instances making performance comparisons unnecessarily difficult.

8.3 Peer-to-Peer Provisioning

After simplification and transcoding to BCNF the formula has to be delivered to the nodes of the Desktop Grid. As the estimation above illustrates this involves the transmission of huge amounts of data for large real-world instances. Even when SATCIETY's preprocessing efforts result in files half as large as the original, the frontend node still would be busy for minutes, which actually stalls the whole computation as all nodes are served concurrently throttling each and every transfer.

Instead of employing a custom solution, SATCIETY leverages BITTORRENT [56] for distributing large formulae. BITTORRENT is a peer-to-peer file sharing protocol which distributes the onus of uploading over all participants. A large body of research concerning all aspects of the protocol has been conducted. For a thorough analysis of its performance in heterogeneous systems – like Desktop Grids – see Liao et al. [57].

The BITTORRENT distribution mechanism is implemented as an additional stage of SATCIETY's solving pipeline. As demonstrated in Section 9.2.3 using BITTORRENT significantly reduces provisiong time for large instances and/or large Grids.

9 Performance Evaluation

To substantiate the effectiveness of our approach to parallel SAT solving in highly demanding Desktop Grid environments, we conducted extensive performance studies on a heterogeneous Desktop Grid. The methodology and testbed setup is described in the following section. The actual evaluation consists of two parts: First, we use a synthetic benchmark to assess the essential properties of SATCIETY's distributed task pool implementation, namely scalability and performance in the presence of volatility and faults. Second, we evaluate the application performance based on a set of challenging problems from a recent SAT competitive event.

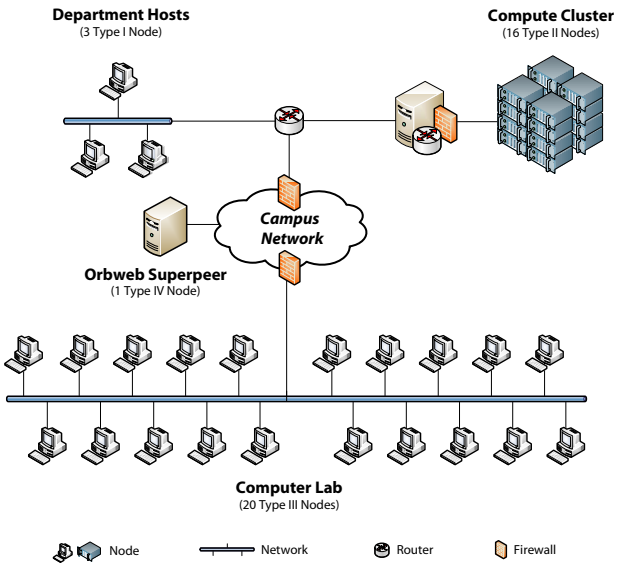


Fig. 13 Testbed Desktop Grid spanning 40 nodes and three networks

Type	Hardware CPU	Memory	Software OS	Kernel	RAPI
I	AMD® Athlon™ 64 X2 2 Cores @ 2.4GHz 512KB Cache / Core	3GB	Linux	2.6.22-14 (generic)	0.54
II	Intel® Xeon™ 2 Processors @ 2.67GHz 512KB Cache / Processor	2GB	Linux	2.6.22.9	0.29
III	Intel® Pentium™ D 2 Cores @ 3.40GHz 2048KB Cache / Core	2GB	Linux	2.6.23 (gentoo-r8)	0.51
IV	Intel® Core™ 2 Q6600 4 Cores @ 2.40GHz 2048KB Cache / Core	8GB	Linux	2.6.22-14 (server)	1.0

Fig. 14 Hardware and software configuration of the nodes of our testbed. We define the *Relative Application Performance Index* (RAPI) as $RAPI(p;I) := (1/|I|)\sum_{i\in I} T_{Seq}(p_{ref},i)/T_{Seq}(p,i)$, where I is a representative subset of the benchmark instances used in Section 9.3, $T_{Seq}(p,i)$ is the sequential runtime of instance i on host p and p_{ref} is the fastest host (Type IV).

9.1 Evaluation Methodology and Testbed Setup

In our case, testing on publicly available wide-area testbeds, e.g., PLANETLAB [58], is not feasible for two reasons: First, they operate on the public Internet and hence experience no network segmentation. Second, they are designed to experiment with I/O bound network applications and typically run many applications in parallel. Executing CPU bound applications consuming large amounts of CPU time over extended periods of time

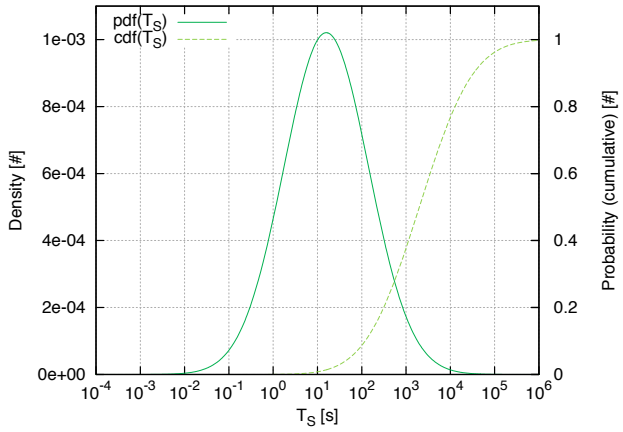


Fig. 15 Probability density (pdf) and cumulative distribution (cdf) functions for the log-normal session time distribution for $\mu = 7.6$ and $\sigma = 2.2$ (note the logarithmic scale of the x-axis).

would inevitably disrupt these applications, violating the usage policies of the testbeds.

Thus, we conducted our evaluation on a dedicated testbed consisting of 40 hosts distributed over three firewalled Fast Ethernet Local Area Networks (100 Mbit/s nominal bandwidth) connected by a campus network as depicted in Figure 13. Their hardware and software setup is summarized in Figure 14. All hosts are used as SATCIETY peers and a single machine additionally is configured to serve as an ORBWEB superpeer.

Wolski *et al.* have shown in [59] that machine availability in Desktop Grids is best described by a log-normal session time (T_S) distribution with the probability density function

$$pdf\left(T_S; \mu, \sigma\right) = \frac{1}{\sqrt{2\pi}\sigma T_S} e^{\frac{-\left(ln T_S - \mu\right)}{2\sigma^2}}; T_S > 0$$

with parameters $\mu = 7.6$ and $\sigma = 2.2$ (see Figure 15). We therefore configured SATCIETY nodes to exhibit random session times according to

$$T_S\left(\nu\right) = e^{7.6+2.2\nu} s$$

with ν being a random variate drawn from the normal distribution with mean 0 and standard deviation 1. T_S is log-normal distributed with mean $22471s$, standard deviation $251709s$ and the quartiles $453s$, $1998s$ (Median), and $8812s$. These values illustrate that the distribution is highly skewed towards low values. On the one hand, this means that there is considerable volatility despite the high mean session time. On the other hand, the high value of the upper quartile implies that we can find a stable node for performing the coordination tasks, i.e., termination detection and fault tolerance, with high probability.

Volatile nodes are *simulated* by having nodes join the SATCIETY group, participating in the computation for T_S seconds, leaving the group and rejoining with a new identity immediately after discarding the nodes internal state. Node failures are modeled by dropping tasks with a given probability P_{Error} when leaving the SATCIETY group. Unfortunately, to our knowledge there are no studies available yet that describe the reasons for node unavailability and quantify their respective share in total unavailability. Thus, we have to use an estimated value. Following our discussion in Section 5, we have chosen a supposedly high default error probability of $P_{Error} = 1\%$ meaning that on average 1 out of 100 node departures is attributed to node failure resulting in dropping the tasks currently located at the departing node.

All SATCIETY peers were run on SUN JRE 1.6.0_12 Java Virtual Machines. For setups consisting of more than 40 nodes, we configured each physical node to run up to four instances of SATCIETY in parallel. The memory limits were set to $M_{Stimulate} = 90\%$, $M_{Forced} = 95\%$, and $M_{Shutdown} = 98\%$ of the total physical system and/or process memory. Evaluation runs were repeated several times (30 times for the distributed task pool, ten times for provisioning, and three times for the application benchmarks). The presented confidence intervals are based on a 95% confidence level.

9.2 Distributed Taskpool

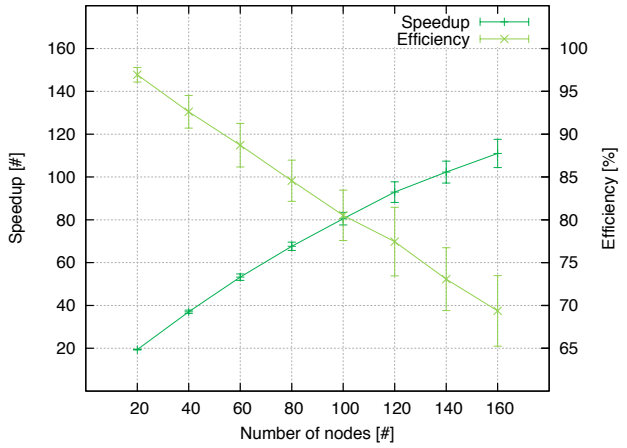
Parallel SAT solving is heavily influenced by algorithmic effects like work-anomalies [60]. To get a clear understanding of the performance opportunities and limits of our approach, we assess the performance of the underlying taskpool described in Section 5 using a synthetic benchmark that is not subject to these effects while still reflecting the irregular nature of parallel SAT solving: A task is defined by the number of seconds T a processor must wait to process the task. Splitting a task $Task_A$ is performed by subtracting a random fraction T_F of the remaining wait time T_R and creating a new task $Task_B$ for T_F:

$$Task_A\left\{T_R\right\} \overset{Split}{\rightarrow} \left(Task_{A'}\left\{T_R - T_F\right\}, Task_B\left\{T_F\right\}\right).$$

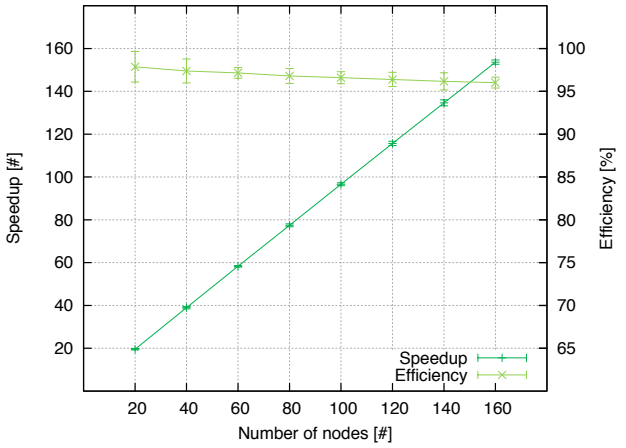
9.2.1 Scalability

To assess the scalability of the task pool, we use two standard metrics: *speedup* S and *parallel efficiency* E. They are defined as

$$S\left(N, p\right) = \frac{T_{Seq^*}\left(N\right)}{T_{Par}\left(N, p\right)}$$

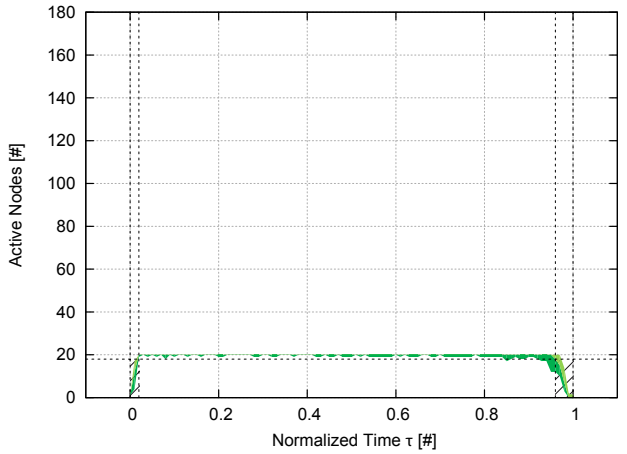


(a) Strong scalability for $T_{Initial} = 600s$

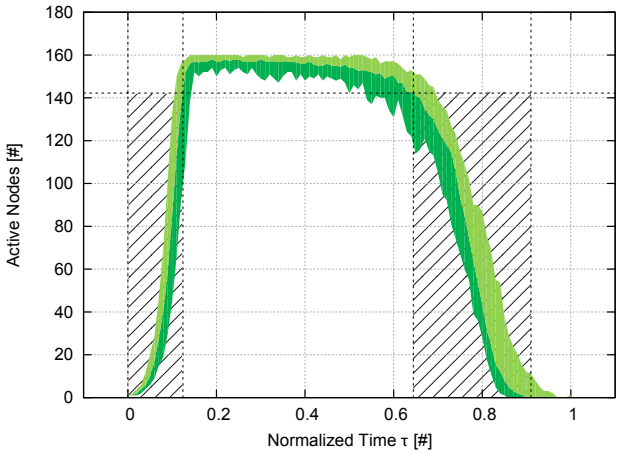


(b) Weak scalability for $T_{Initial} = p\,600s$

Fig. 16 Scalability of SATCIETY's distributed task pool for the synthetic benchmark application.



(a) Node activity over normalized time for $p = 20$ nodes



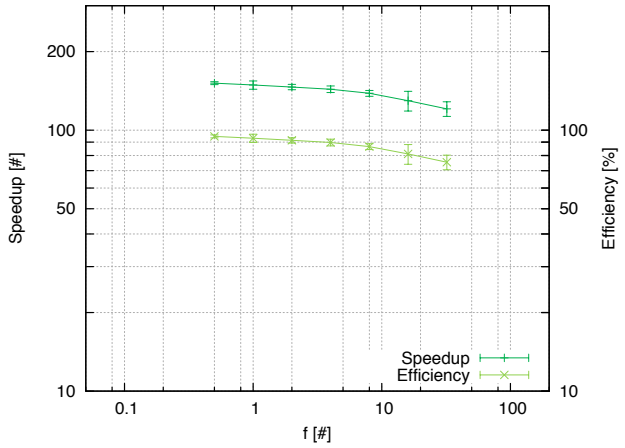
(b) Node activity over normalized time for $p = 160$ nodes

Fig. 17 Node activity over normalized time $\tau\,(t; N, p) = \frac{t}{T_{Par}(N,p)}$ for the synthetic benchmark application in 20 and 160 node setups for $T_{Initial} = 600s$. The figures show the minimum, mean, and maximum number of active nodes at normalized time τ within the dataset collected over 30 runs. Shaded areas — defined as times where the number of active nodes is below 90% of the number of available nodes — contribute to the parallel overhead of the computation.

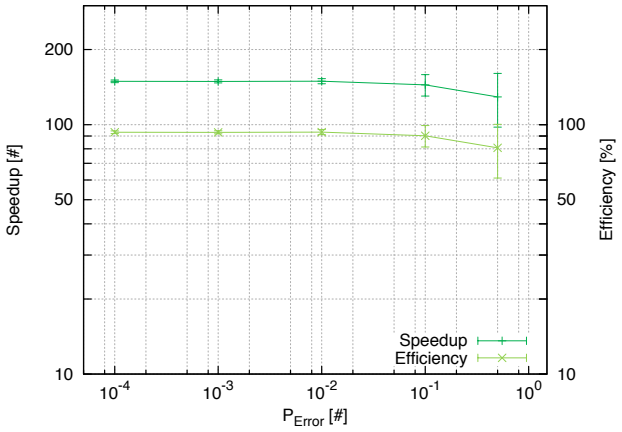
and

$$E\,(N,p) = \frac{1}{p} S\,(N,p)$$

where $T_{Par}\,(N,p)$ is the parallel runtime for a given problem size N on p processors. $T_{Seq^*}\,(N)$ denotes the runtime of the best (known) sequential algorithm for a given problem size N, which in our case is simply the initial overall wait time $T_{Initial}$. According to how N is selected one speaks of *strong* ($N \in O\,(1)$) and *weak scalability* ($N \in O\,(p)$). Figure 16a shows strong scalability metrics for $T_{Initial} = 600s$ and up to 160 non-volatile nodes. While the speedup is almost optimal for small node counts, it increasingly deviates from perfect speedups for increasing node counts resulting in a speedup of 111.0 ± 6.6 and a corresponding efficiency of roughly $69.4\% \pm 4.1\%$ for 160 nodes. The reason for this behavior is the increasing parallel overhead caused by idle nodes at the beginning and the end of the computation (see Figure 17). The time required to find an active node for work-stealing at a given point in time is proportional to the number of active nodes at that time. Thus, work diffusion at the beginning of the computation requires time $T_{Growth} \propto \log\,(p)$ until all nodes are active. For the same reason, the end of the computation is governed by exponential decay creating an analogue dependency $T_{Decay} \propto \log\,(p)$. Of course, the impact of these overheads on overall efficiency depends on the ratio between problem size and node count.



(a) Performance for varying mean session time $T_S^* = \frac{1}{f} T_s$



(b) Performance for varying failure probability P_{Error}

Fig. 18 Volatility tolerance of SATCIETY's distributed task pool

As expected, weak scalability ($T_{Initial} = p\,600s$) of the system is much better. As can be seen in Figure 16b, speedups grow linearly for increasing node count resulting in a slowly dropping parallel efficiency between $97.9\% \pm 1.8\%$ for 20 and $96.0\% \pm 0.6\%$ for 160 nodes.

9.2.2 Volatility Tolerance

Tolerating deliberate and failure induced node departures is one of the most important requirements for a Desktop Grid Computing system. Besides correctness, these factors may also severely impair performance. To substantiate that SATCIETY doesn't suffer from such inadequacies, we measured speedup and efficiency for the synthetic benchmark application under volatility.

Figure 18a depicts the impact of varying mean session time for a fixed error probability of 1% in a 160 node setup and a problem size of $N = 160 \cdot 600s = 96,000s$. The variation was done by using $T_S^* = \frac{1}{f} T_s$ with fixed $f > 0$ as the actual session time. For $f = 1$ — resulting in the session time distribution described in Section 9.1 — the speedup is 149.0 ± 5.5 and the efficiency is $93.1\% \pm 3.4\%$. The penalty when compared to the non-volatile setup from above is roughly 3%. For increasing f, speedups drop moderately. Even in the most demanding scenario evaluated here with a 32 times shorter mean session time, we still get a considerable speedup of 120.6 ± 7.8 or an efficiency of $75.4\% \pm 4.9\%$.

Figure 18b shows the performance penalties for different error probabilities P_{Error} in a 160 node setup with the real-world session time distribution ($f = 1$). For $P_{Error} = 1\%$, we see a speedup of 149.4 ± 3.5 and an efficiency of $93.3\% \pm 2.2\%$. Decreasing the error probabilities further results in nearly no further improvement. Noticeably, increasing P_{Error} to as much as 50% still yields an acceptable speedup of 129.2 ± 31.4. The large error is due to the fact that departures are comparatively infrequent so that the size T of the dropped task becomes decisive for the actual impact on efficiency.

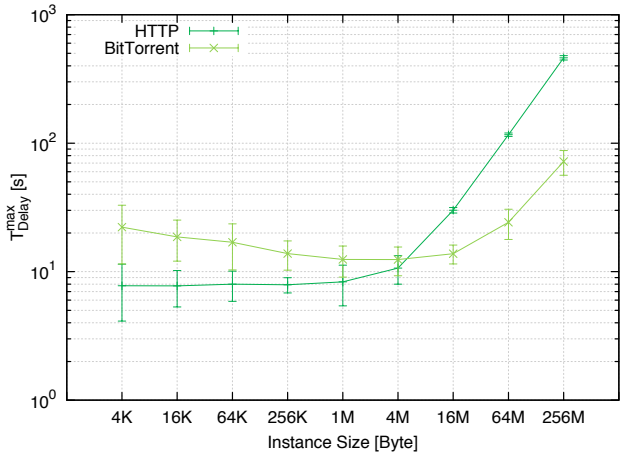


Fig. 19 Comparison of elapsed time T_{Delay}^{max} until an instance of given size is available on all nodes

Besides the dominating overhead associated with work repetition and restoration costs for lost tasks, a small fraction of the difference of roughly 3% in efficiency between the non-volatile (96%) and the volatile setup (93.1%/93.3%) can be attributed to the slightly smaller effective group size resulting from the small pause when nodes leave and rejoin the SATCIETY group.

9.2.3 Provisioning

As discussed in Section 8, SAT instances can be very large. Distributing the file from a single server inevitably becomes a limiting factor for achieving high efficiency when the system is scaled beyond a few dozen nodes. To substantiate this claim, we performed a comparison between server-based provisioning over HTTP and P2P provisioning over BitTorrent. Our implementation is based on the libtorrent library version 0.14.9 from *Rasterbar Software* [61] attached over the *Java Native Interface* (JNI).

Figure 19 shows the time until the instance is available on all nodes in a 40 node setup for varying instance sizes and both protocols. For files up to a size of 4 MB the overhead of scraping and peer-to-peer connection negotiation levels out the possible speedup of multi-sourced download resulting in relative differences between $\approx$ 185% for 4 KB and $\approx$ 16% for 4 MB instances in favor of server-based provisioning. For 16 MB instances and beyond P2P provisioning clearly outperforms server-based provisioning by increasing factors up to $\approx$ 3.8 for 256 MB instances. The overall amount of data delivered in this setup is 10 GB.

As the absolute difference between the two mechanisms is small for instance sizes up to 4 MB, the potential of an adaptive strategy, that employs the best mechanism for a given instance size, is limited. Furthermore, the transition point where P2P based provisioning first outperforms server-based provisioning would shift towards lower instance sizes for larger networks. Thus, Satciety always uses P2P provisioning regardless of instance size.

9.3 Parallel SAT Solving

Satciety provides an interface to plugin any DPLL-based solver that provides the ability to split on demand and that emits and is able to consume lemmas on the fly. For our tests, we used MiniSAT v1.14 [48] attached over JNI as Satciety's solver core. MiniSAT attained top rankings in SAT competitive events of the last years [62].

Motivated by the scalability results from the previous section, the benchmark suite used to evaluate Satciety's SAT solving performance consists of long-running benchmarks from all three categories of the SAT Competition 2007. We call an instance *long-running* when it has been solved by MiniSAT within the allowed time ($10,000s$ for industrial and $5,000s$ for crafted and random instances) but took at least $1,200s$. This criterion is met by 15 random, 23 crafted, and 17 industrial

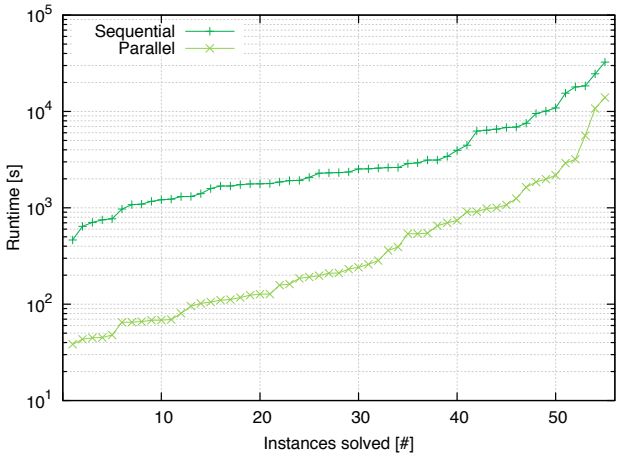


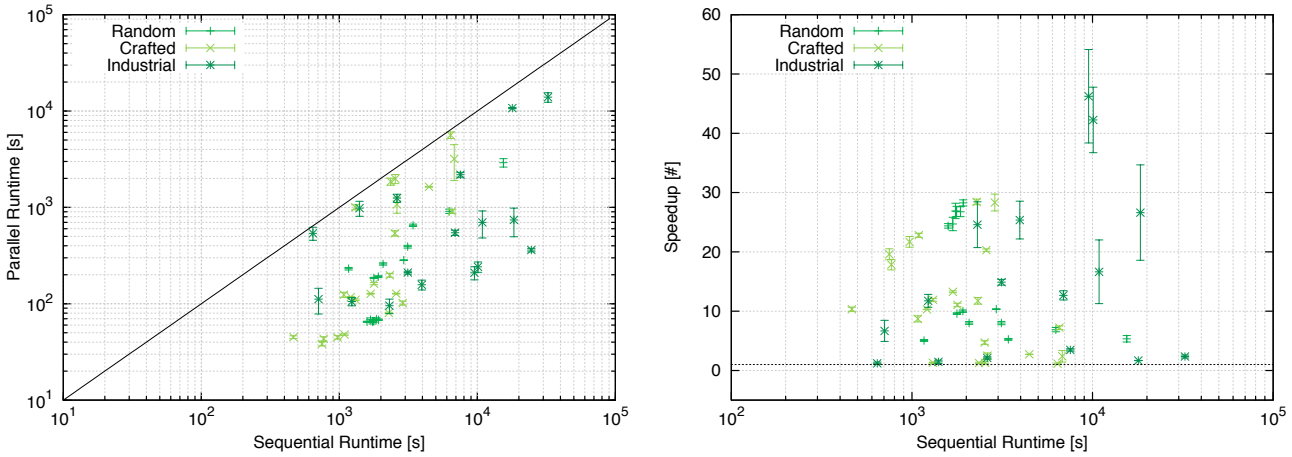
Fig. 20 Cactus Plot[3] comparing the performance of the sequential MiniSAT and Satciety

instances. Note that we restrict our evaluation to unsatisfiable instances. For parallel heuristic search methods like SAT solving, speedups obtained for satisfiable instances ultimately result from the parallelization of non-optimal sequential methods. (Note that the computation is terminated when a solution is found.) In the case of SAT it is impossible to find an optimal method for all instances. Thus, parallel processing of satisfiable instances is an important means for speeding up the solving process. However, performance results obtained for satisfiable instances are not suitable to investigate on the effectiveness of a specific parallel method.

Sequential runtimes were measured on the fastest machine among the testbed nodes (Type IV in Table 14). The coordinator performing fault-tolerance and termination detection was located on the same node. All other nodes were configured to use the real-world session time distribution as described above and an error probability of $P_{Error} = 1\%$. Although recent research [63] indicates that using (at least rapid) restarts may not be beneficial for unsatisfiable instances, disabling them would bias the results towards unsatisfiable instances. We thus use restarts for both the sequential and the parallel setups.

Table 1 shows the results of our evaluation. With the exception of four benchmarks, preprocessing through SatELite and transcoding to BCNF resulted in significant file size reduction of up to 81.4% and 30.3% on average. Preprocessing times (T_{pre}) are – with a single exception (`uts-l06-ipc5-h33-unknown`) – negligible as compared to both sequential (T_{seq}) and parallel

[3] *Cactus Plots* are traditionally used in SAT competitive events to compare SAT solver performance. It is a cumulative plot showing how many instances (x-axis) have been solved in time below or equal to a given runtime (y-axis).



(a) Head-to-head solver comparison with respect to runtime

(b) Speedup by sequential runtime

Fig. 21 Performance comparison of sequential MINISAT and SATCIETY

(T_{par}) solver runtimes. Thus, they do not significantly affect speedups.

As can be seen from Figure 20 and Figure 21a, SATCIETY clearly outperforms the sequential solver in all categories realizing significant average speedups of 15.1 ± 0.5 for random, 11.3 ± 0.4 for crafted, and 18.1 ± 2.5 for industrial instances (see Figure 21b). The average speedup over all categories is 14.5 ± 1.1. Note that the given speedup values are based on the sequential runtimes on the fastest machine with a RAPI value of 1.0 (cf. Table 14) while the mean RAPI value of the nodes in our testbed is 0.44. In this light, the performance of our parallel approach to SAT solving is even more impressive.

The variability of speedups, both across instances and across runs for the same instance, however is very pronounced. This decreased robustness is caused by work-anomalies, i.e., the total amount of work carried out by the parallel execution is different and in some cases considerably larger than for the sequential execution. This behavior shows that it is generally difficult to keep the effectiveness of the sequential heuristics in the corresponding parallel version at the same level for a wide range of different SAT instances. Work anomalies are typical for all known approaches to parallel SAT solving. They tend to become more pronounced for larger number of processors which represents a significant challenge for future research.

10 Related Work

In the following discussion of related work, we concentrate on approaches to parallel SAT solving that are designed for distributed-memory parallel architectures. For a further treatment of parallel SAT solving on shared-memory parallel hardware, we refer the reader to [64], [65], [66] and [67].

10.1 Parallel SAT Solving on Distributed-Memory Architectures

One of the first parallel SAT solvers was presented by Böhm and Speckenmeyer [68]. Their work especially investigates on efficient load balancing techniques for a d-dimensional mesh network-topology of a transputer. Another early approach to parallel SAT solving is Zhang's PSATO [26]. PSATO is a distributed parallel SAT solver targeted for networks of workstations. It introduced the *guiding path* technique for exploratory problem decomposition. This technique takes advantage of the decision heuristic of sequential solvers for splitting the search space. PSATO is based on external parallelization of the sequential solver SATO. The parallel solver PSatz by Jurkowiak *et al.* [29] is a parallel variant of the sequential solver Satz. PSatz employs a very similar approach to parallelization as PSATO, but uses work-stealing techniques for load-balancing. All parallel SAT solvers we discussed so far focus on the parallelization of the search process but don't establish a distributed learning process, which is crucial for exploiting the potential of modern SAT solving methods in parallel environments.

PaMiraXT and PaSAT are parallel SAT solvers that both establish a distributed learning process, but are based on contrary design principles:

PAMIRAXT by Schubert *et al.* [69] is a parallel SAT solver designed for networks of shared-memory parallel computers. It is based on a centralized Mas-

Instance	Preprocessing				Solving Runtime		Speedup
	L_I [Byte]	L_F [Byte]	δ_L [%]	T_{pre} [s]	T_{seq} [s]	T_{par} [s]	
Random Category							
unif2p-p0.7-v4500-c12015-S1349608788-18	78892	59607	24	0.74	15454.42	2912.38 ± 295.22	5.34 ± 0.54
unif2p-p0.8-v1665-c5178-S883553387-12	32368	25425	21	0.46	3418.74	652.04 ± 19.11	5.24 ± 0.15
unif2p-p0.8-v2035-c6328-S1022605561-16	40127	30140	25	0.50	1775.25	184.80 ± 2.74	9.58 ± 0.14
unif2p-p0.9-v630-c2280-S1071799860-07	13560	11103	18	0.35	1166.76	230.75 ± 6.90	5.05 ± 0.15
unif2p-p0.9-v810-c2932-S1275186626-09	17766	14178	20	0.37	6263.49	911.66 ± 48.60	6.88 ± 0.36
unif-k3-r4.26-v400-c1704-S1013535775-14	10007	7777	22	0.33	3127.13	391.11 ± 13.09	8.00 ± 0.27
unif-k3-r4.26-v400-c1704-S105499989-20	9982	7784	22	0.33	1914.97	191.20 ± 3.89	10.00 ± 0.20
unif-k3-r4.26-v400-c1704-S1671397883-11	10050	7800	22	0.33	2073.05	258.62 ± 7.33	8.01 ± 0.23
unif-k3-r4.26-v400-c1704-S1925680230-06	10013	7746	23	0.33	2928.86	282.92 ± 1.40	10.34 ± 0.05
unif-k5-r21.3-v90-c1917-S1380126410-13	14035	9930	29	0.37	1740.44	64.42 ± 3.17	26.91 ± 1.29
unif-k5-r21.3-v90-c1917-S1412274662-11	14042	9938	29	0.37	1585.32	64.67 ± 0.99	24.39 ± 0.37
unif-k5-r21.3-v90-c1917-S1414833579-15	13988	9925	29	0.37	1682.77	67.85 ± 3.03	24.71 ± 1.12
unif-k7-r89-v55-c4895-S1155123565-17	42501	34066	20	0.45	1923.19	67.62 ± 1.23	28.26 ± 0.51
unif-k7-r89-v55-c4895-S1215610276-02	42451	34125	20	0.44	1766.97	65.48 ± 2.13	26.83 ± 0.85
unif-k7-r89-v55-c4895-S145251364-05	42466	34120	20	0.44	1854.85	68.91 ± 2.13	26.77 ± 0.83
Crafted Category							
999999000001nw.sat05-447	130996	70639	46	0.95	748.70	37.38 ± 1.83	19.59 ± 0.92
connm-ue-csp-sat-n800-d0.02-s925928766.sat05-538	48848	46435	5	0.73	1077.33	123.32 ± 7.24	8.71 ± 0.52
contest03-hwb-n26-01-S1957858365.sat05-500	4713	3088	34	0.27	464.08	44.71 ± 1.49	10.33 ± 0.35
contest03-SGI_30_50_30_20_1-dir.sat05-439	274484	312259	-14	6.88	1679.13	120.21 ± 1.37	13.27 ± 0.14
contest03-SGI_30_50_30_20_3-dir.sat05-440	279307	311213	-11	4.71	2872.18	97.03 ± 4.95	28.32 ± 1.42
contest04-lksat-n1000-m6860-k4-l4-s1935114289.sat05-523	52960	39473	25	0.46	2319.05	197.33 ± 9.12	11.74 ± 0.56
hwb-n26-03-S540351185.sat05-490	4727	3109	34	0.27	769.86	42.97 ± 2.19	17.84 ± 0.89
hwb-n28-01-S136611085.sat05-491	5127	4112	20	0.28	1309.34	109.67 ± 2.16	11.91 ± 0.23
hwb-n28-02-S818962541.sat05-492	5088	4095	20	0.28	2584.33	127.11 ± 1.23	20.29 ± 0.20
linvrinv5.sat05-564	7502	6120	18	0.32	1090.34	47.50 ± 0.72	22.81 ± 0.34
mod2c-3cage-10-2.sat05-2567	3624	2641	27	0.23	2527.79	538.19 ± 33.68	4.71 ± 0.31
mod2c-3cage-10-3.sat05-2568	3644	2620	28	0.22	2616.97	1073.21 ± 204.98	2.50 ± 0.46
phnf-size10-exclusive-luckySeven.used-as.sat04-990.sat05-4196	6558136	6468952	1	50.47	6751.40	3134.40 ± 1289.16	2.39 ± 0.97
pmg-12.sat05-3940	3370	2709	20	0.23	2282.49	80.11 ± 1.32	28.42 ± 0.47
pyhala-braun-40-4-02.sat05-459	211234	71182	66	0.89	969.55	43.90 ± 1.85	21.69 ± 0.90
QG7a-gensys-icl001.sat05-3822	228575	168157	26	1.00	6395.19	5619.02 ± 443.29	1.14 ± 0.09
QG7-dead-dnd001.sat05-3419	102470	52390	49	0.58	1304.73	1002.32 ± 58.88	1.30 ± 0.07
QG7-dead-dnd002.sat05-3108	141168	64408	54	0.72	6550.16	908.05 ± 35.96	7.22 ± 0.29
QG7-gensys-icl100.sat05-3226	162219	100981	38	0.78	2539.24	1978.89 ± 213.54	1.29 ± 0.15
QG7-gensys-ukn003.sat05-3346	135088	84653	37	0.69	4464.59	1635.71 ± 15.29	2.73 ± 0.03
s101-100	812	562	31	0.17	1784.41	161.49 ± 4.14	11.04 ± 0.28
s97-100	777	546	30	0.17	1210.75	117.36 ± 0.43	10.30 ± 0.04
unsat-set-b-fclqcolor-10-07-09.sat05-1282	19664	21806	-11	0.46	2350.76	1854.89 ± 157.90	1.27 ± 0.11
Industrial Category							
AProVE07-08	68759	66808	3	0.90	1228.54	104.46 ± 10.29	11.74 ± 1.09
AProVE07-09	696516	693378	0	14.30	7522.14	2169.99 ± 127.73	3.46 ± 0.21
AProVE07-16	814813	378435	54	6.05	699.84	105.56 ± 33.29	6.68 ± 1.78
AProVE07-27	118729	100651	15	1.90	24591.93	358.88 ± 15.61	68.25 ± 2.98
cube-11-h13	6238423	5259684	16	318.06	17609.81	10396.53 ± 156.03	1.67 ± 0.02
dated-10-11-u	2806193	637494	77	14.49	10857.07	685.39 ± 218.52	16.63 ± 5.37
dated-10-13-u	3672073	871186	76	20.06	9496.21	189.41 ± 32.51	46.25 ± 7.88
dated-5-15-u	3148506	635904	80	16.04	3939.23	141.47 ± 18.50	25.36 ± 3.19
dated-5-17-u	3902323	725932	81	20.17	10072.31	221.30 ± 29.78	42.25 ± 5.53
emptyroom-4-h21	949908	748866	21	11.44	32460.37	13904.39 ± 1571.51	2.35 ± 0.25
eq.atree.braun.11	26538	20337	23	0.47	2302.96	94.93 ± 16.38	24.59 ± 3.86
manol-pipe-f9b	2495048	1069250	57	19.06	2595.25	1228.82 ± 120.50	2.11 ± 0.21
manol-pipe-f9n	2517249	1083411	57	19.36	1382.13	962.74 ± 174.24	1.46 ± 0.28
manol-pipe-g10nid	2995202	1254304	58	21.80	18389.17	718.16 ± 243.59	26.62 ± 8.04
sortnet-6-ipc5-h11	405533	174448	57	8.91	3121.68	201.56 ± 6.94	14.88 ± 0.48
total-10-13-u	4661995	1144413	75	25.70	6872.45	520.27 ± 34.48	12.67 ± 0.79
uts-l06-ipc5-h33-unknown	3496371	3625799	-4	209.05	433.79	329.46 ± 83.39	1.21 ± 0.19

Table 1 SATCIETY performance for long-running SAT Competition 2007 problems in a volatile Desktop Grid ($L_I \mathrel{\hat{=}}$ initial instance size, $L_F \mathrel{\hat{=}}$ final instance size after preprocessing, $\delta_L = 1 - (L_F/L_I) \mathrel{\hat{=}}$ instance size reduction, $T_{pre} \mathrel{\hat{=}}$ time spent on preprocessing, $T_{seq} \mathrel{\hat{=}}$ runtime of the sequential solver, $T_{par} \mathrel{\hat{=}}$ runtime of the SATCIETY solver). The index of dispersion is the (sample) standard deviation.

ter/Worker model, where the master is responsible for steering problem decomposition and load balancing. The master also serves as a hub for collecting and disseminating lemmas among the clients. Due to the completely centralized architecture, the scalability of this approach is limited. The authors present performance evaluations for a distributed environment consisting of 3 nodes with 8 cores in total.

The parallel SAT solver PaSAT by Blochinger *et al.* [70] is targeted for tightly coupled distributed memory architectures, like HPC clusters. It is based on a fully distributed task pool execution model for parallelizing the search process. Additionally, PaSAT establishes a distributed parallel learning process based on asynchronous communication: Each node dispatches a mobile agent which visits other nodes and gathers pertinent lemmas. The fully distributed problem decomposition and load balancing process enabled by the underlying distributed task pool, together with the asynchronous and selective dissemination of lemmas by mobile agents, ensures high scalability of the approach. The performance of PaSAT is evaluated in a distributed environment comprised of 24 nodes.

In contrast to SATCIETY's task pool, those of PaMiraXT and PaSAT both do not support volatile environments.

10.2 Parallel SAT Solving on Grids

ZetaSAT by Blochinger *et al.* [27] is a framework for parallel SAT solving on Desktop Grids. It is built on

top of the Client/Server based Desktop Grid platform ZetaGrid. Due to the limitations of this class of Desktop Grids (see Section 1), it uses a centralized task pool and does not communicate lemmas among the nodes.

In [50] Hyvärinen *et al.* present a distributed SAT solving method incorporating a limited form of dynamic learning which is tailored for Grids comprised of batch controlled resources, where individual jobs are not able to communicate directly. Their approach is based on competition parallelism where several randomized SAT solvers independently work on the same SAT instance until one finds a solution. For solver jobs which are terminated by the Grid scheduler because they exceeded their resource limits the lemmas deduced so far are communicated to the master node and stored in a central clause database. When additional solver jobs for the same instance are submitted some of these lemmas are selected by a heuristic and added to the initial clause databases.

GridSAT [71,49] by Chrabakh and Wolski is a parallel SAT Solver especially designed for Globus based Grids. The basic parallel procedure employs exploratory problem decomposition controlled by a dedicated master node. More precisely, the master node acts as a scheduler based on information delivered by external resource management services. It is also responsible for storing checkpoints. Lemmas are periodically exchanged between nodes. Thereby the maximum size of the lemmas which are selected for exchange is dynamically adjusted in order to adapt to the available network bandwidth. In GridSAT special attention is given to dynamically include batch controlled resources. When batch controlled resources become available, tasks are migrated from interactive nodes to these resources in order to exploit the additional computational power for the time allotted by the batch system.

While all discussed approaches are based on centralized control, SATCIETY is to the best of our knowledge the first parallel SAT solver for Grid environments that employs a decentralized execution model, which is – as shown in Section 9 – able to provide good performance and scalability even under a high degree of volatility and heterogeneity.

11 Conclusion

In this article, we have reported on SATCIETY, our parallel SAT solver for P2P Desktop Grids. SATCIETY is capable of solving problem instances used in current SAT competitions with significant speedups compared to state-of-the-art sequential SAT solvers. Achieving this high performance level on P2P Desktop Grids is considerably more demanding than in traditional parallel and distributed environments. SATCIETY effectively deals with resource volatility and heterogeneity in potentially large-scale Desktop Grids by means of a sophisticated distributed task pool implementation, various efforts to reduce problem and task size, scalable data provisioning, protective solver memory management, and adaptive topology-aware lemma exchange (see Figure 22).

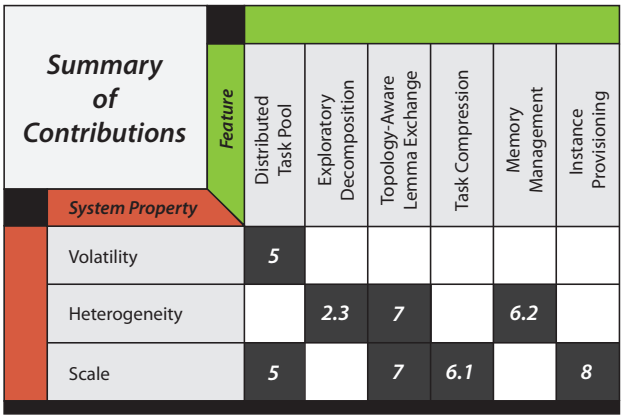


Fig. 22 Overview of the contributions made in this article. Numbers indicate the section where the respective feature is described.

Our work provides a solid foundation at the system level for achieving high scalability of parallel SAT solving (as well as for other problems based on heuristic search processes, e.g., discrete optimization). This is a crucial building block for enabling further research in parallel SAT solving which must address robustness in the first place. This requires a deeper understanding of the causes of work-anomalies which accompany the transformation of highly optimized sequential SAT algorithms into their parallel counterparts. The next step in our endeavor to tap the full potential of massively parallel SAT solving will be to incorporate additional techniques orthogonal to exploratory decomposition for exploiting parallelism including shared-memory data parallel techniques and hybrid approaches involving competition parallelism.

Acknowledgments

Sven Schulz is supported by Deutsche Forschungsgemeinschaft (DFG) under grant BL 941/1-2.

References

1. David P. Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In *Proc. of the*

Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006), pages 73–80, Singapore, 2006.
2. Derrick Kondo, Michela Taufer, Charles L. Brooks, Henri Casanova, and Andrew A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proc. of International Parallel and Distributed Processing Symposium*, Sante Fe, New Mexico, 2004.
3. Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *Journal of Parallel Distributed Computing*, 63:597–610, 2003.
4. David P. Anderson. BOINC: a system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Pittsburgh, USA, 2004.
5. R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pages 256–267, February 2003.
6. Rich Wolski, Neil Spring, and Jim Hayes. Predicting the cpu availability of time-shared unix systems on the computational grid. In *Proc. of the the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 105–112, Washington, DC, USA, 1999.
7. Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and Ilya Sharapov. Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. In *Proceedings of the Third International Workshop on Grid Computing (GRID '02)*, pages 1–12, London, UK, 2002. Springer-Verlag.
8. Yoshio Tanaka Kazuyuki Shudo and Satoshi Sekiguchi. P3: P2P-based middleware enabling transfer and aggregation of computational resources. In *Proc. Cluster Computing and Grid 2005 (Fifth Int'l Workshop on Global and Peer-to-Peer Computing)*, Cardiff, UK, 2005.
9. Sven Schulz, Wolfgang Blochinger, Markus Held, and Clemens Dangelmayr. COHESION - A microkernel based desktop grid platform for irregular task-parallel applications. *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Applications*, 24(5):354–370, 2008.
10. S. A. Cook. The complexity of theorem proving procedures. In *3rd Symp. on Theory of Computing*, pages 151–158. ACM press, 1971.
11. 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.
12. Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proc. of the 38th Conference on Design Automation Conference*, 2001.
13. P. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. Combinational test pattern generation using satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9):1167–1176, 1996.
14. Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proc. of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.
15. James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proc. of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1092–1097, Seattle, Washington, 1994. AAAI Press/MIT Press.
16. F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *Journal of Automated Reasoning*, 24(1-2):165–203, February 2000.
17. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
18. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
19. J. P. Marques-Silva and K. A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, 1996.
20. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proc. of 8th International Conference on Computer Aided Deduction(CADE 2002)*, Copenhagen, Denmark, 2002.
21. David G. Mitchell. A SAT solver primer. *EATCS Bulletin (The Logic in Computer Science Column)*, 85:112–133, February 2005.
22. J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.
23. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the 38th Design Automation Conference*, pages 530–535, Las Vegas, 2001.
24. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. of ICCAD*, San Jose, CA, 2001.
25. Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.
26. 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.
27. Wolfgang Blochinger, Wolfgang Westje, Wolfgang Küchlin, and Sebastian Wedeniwski. ZetaSAT – Boolean satisfiability solving on desktop grids. In *Proc. of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, volume 2, pages 1079–1086, Cardiff, UK, May 2005.
28. Wahid Chrabakh and Rich Wolski. GridSAT: A Chaff-based distributed SAT solver for the grid. In *Proc. of Supercomputing 03*, Phoenix, Arizona, USA, 2003.
29. Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.
30. Wolfgang Blochinger, Clemens Dangelmayr, and Sven Schulz. Aspect-oriented parallel discrete optimization on the Cohesion desktop grid platform. In *Proc. of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 49–56, Singapore, May 2006.
31. Sven Schulz, Wolfgang Blochinger, and Hannes Hannak. Capability-aware information aggregation in peer-to-peer grids – methods, architecture, and implementation. *Journal of Grid Computing*, 7(2):135–167, 2009.
32. Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massouli. Peer-to-Peer Membership Management for Gossip-based Protocols. *IEEE Trans. Comput.*, 52(2):139–149, 2003.
33. Sven Schulz, Wolfgang Blochinger, and Matthias Poths. Orbweb – a network substrate for peer-to-peer grid computing based on open standards. *Journal of Grid Computing*, 8(1):77–107, 2010.
34. The XMPP Software Foundation. http://www.xmpp.org. Last Accessed: March 2010.
35. Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

36. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
37. Jeff Matocha and Tracy Camp. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software*, 43(3):207–221, November 1998.
38. Ronald F. DeMara, Yili Tseng, and Abdel Ejnioui. Tiered algorithm for distributed process quiescence and termination detection. *IEEE Transactions on Parallel and Distributed Systems*, 18(11):1529–1538, 2007.
39. D. M. Dhamdhere, Sridhar R. Iyer, and E. Kishore Kumar Reddy. Distributed termination detection for dynamic systems. *Parallel Comput.*, 22(14):2025–2045, 1997.
40. Gideon Stupp. Stateless termination detection. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 163–172, London, UK, 2002. Springer-Verlag.
41. N. Mittal, S. Venkatesan, and S. Peri. Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies. In *Proc. 18th Ann. Conf. Distributed Computing (DISC '04)*, 2004.
42. Chengzhong Xu and Francis C. M. Lau. Efficient termination detection for loosely synchronous applications in multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 7(5):537–544, 1996.
43. Shimon Cohen and Daniel Lehmann. Dynamic systems and their distributed termination. In *PODC '82: Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 29–33, New York, NY, USA, 1982. ACM.
44. Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
45. Jayadev Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 4(1):37–43, 1982.
46. Ten-Hwang Lai. Termination detection for dynamically distributed systems with non-first-in-first-out communication. *J. Parallel Distrib. Comput.*, 3(4):577–599, 1986.
47. Friedemann Mattern. Global quiescence detection based on credit distribution and recovery. *Inf. Process. Lett.*, 30(4):195–200, 1989.
48. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003*, volume 2919 of *LNCS*, pages 502 – 518, Santa Margherita Ligure, Italy, 2003. Springer Verlag.
49. Wahid Chrabakh and Rich Wolski. Gridsat: a system for solving satisfiability problems using a computational grid. *Parallel Computing*, 32(9):660–687, 2006.
50. Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Incorporating learning in grid-based randomized SAT solving. In *Proceedings of the 13th international conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 247–261, Varna, Bulgaria, 2008.
51. Satisfiability - suggested format., May 1993. http://www.satlib.org/Benchmarks/SAT/satformat.ps. Last Accessed: March 2010.
52. The SAT Competition 2007 website. http://www.satcompetition.org/2007/. Last Accessed: March 2010.
53. The SAT Race 2008 website. http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/. Last Accessed: March 2010.
54. Anbulagan and John Slaney. Multiple preprocessing for systematic SAT solvers. In *Proceedings of the 6th International Workshop on the Implementation of Logics*, Phnom Penh, Cambodia, 2006.
55. Nicolas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *Proc. 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2005.
56. Bittorrent. http://www.bittorrent.com/protocol.html. Last Accessed: March 2010.
57. Wei-Cherng Liao, Fragkiskos Papadopoulos, and Konstantinos Psounis. Performance analysis of BitTorrent-like systems with heterogeneous users. *Performance Evaluation*, 64(9-12):876–891, 2007.
58. Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using planetlab for network research: myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, 2006.
59. R. Wolski, D. Nurmi, and J. Brevik. An analysis of availability distributions in Condor. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, March 2007.
60. Wolfgang Blochinger. Towards robustness in parallel SAT solving. In *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of the International Conference ParCo 2005)*, Malaga, Spain, September 2006.
61. Rasterbar Software website. http://www.rasterbar.com/. Last Accessed: March 2010.
62. SAT Competition. *http://www.satcompetition.org*.
63. Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Bning and Xishun Zhao, editors, *SAT*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008.
64. Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver: Design and implementation. In *3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004)*, London, U.K., 2004.
65. Daniel Singer and Alain Vagner. Parallel resolution of the satisfiability problem (SAT) with openmp and mpi. In *6th International Conference on Parallel Processing and Applied Mathematics, PPAM 2005*, pages 380–388, 2005.
66. M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT solving. In *In 12th Asia and South Pacific Design Automation Conference*, pages 926–931, 2007.
67. Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.
68. M. Boehm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.
69. Tobias Schubert, Matthew Lewis, and Bernd Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.
70. Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.
71. Wahid Chrabakh and Rich Wolski. Gridsat: Design and implementation of a computational grid application. *Journal of Grid Computing*, 5(2):177–193, 2006.