A Distribution Method for Solving SAT in Grids

Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä

Helsinki University of Technology, Laboratory for Theoretical Computer Science, P.O. Box 5400, FI-02015 TKK, Finland
{Antti.Hyvarinen,Tommi.Junttila,Ilkka.Niemela}@tkk.fi

Abstract. The emerging large-scale computational grid infrastructure is providing an interesting platform for massive distributed computations. In this paper the problem of exploiting such computational grids for solving challenging propositional satisfiability problem (SAT) instances is studied. When designing a distributed algorithm for a large loosely coupled computational grid, a number of grid specific problems need to be tackled including the heterogeneity of the resources, inherent communication delays, and high failure probabilities of grid jobs. In this work a novel distribution method for solving SAT problem instances, called scattering, is introduced. The key advantages of scattering are that it can be used in conjunction with any sequential SAT solver (including industrial black box solvers), the distribution heuristic is strictly separated from the heuristic used in sequential solving, and it requires no communication between processes solving subproblems but still allows coordination of such processes. An implementation of the method has been developed for NorduGrid, a large widely distributed production-level grid running in Scandinavia. The implementation has been benchmarked with test cases including random 3SAT and challenging industrial benchmarks used in previous SAT competitions.

1 Introduction

We study the *propositional satisfiability problem* (SAT) of determining whether a given propositional formula has a satisfying truth assignment. Decision methods for SAT and their implementation techniques have advanced considerably during the last decade and SAT based techniques have been applied successfully in several areas including planning [1,2], model checking of finite state systems [3,4], testing [5], hardware verification [6], VLSI-routing [7], and scheduling [8].

An interesting approach to boosting the applicability of SAT based problem solving is to exploit parallel and distributed computation to solve computationally challenging SAT problem instances. Improvements in networking, availability of networked clusters and, in particular, the emerging large scale computational grid infrastructure make this approach increasingly attractive. For example, the largest software project currently funded by the European Union is the EGEE project (Enabling Grids for E-sciencE; `http://public.eu-egee.org/`).

In this paper we study how to exploit the rapidly developing computational grid infrastructure in solving challenging SAT instances. Compared to more tightly coupled parallel and distributed computing architectures, computational grids have properties that need to be taken into account when designing distributed algorithms. In particular,

- the available resources can be quite heterogeneous in a grid,
- no shared memory is available and communication delays are significant, and
- individual jobs executed in grid nodes have non-negligible failure rates.

The goal is an approach where we can exploit the best available sequential SAT solving techniques and the (almost unlimited) computational resources that are becoming available through emerging computational grids. For this we are developing distributed SAT solving methods that satisfiy the following set of requirements.

- Any sequential SAT solver can be exploited in solving a (sub)problem with little (or preferably no) changes.
- If the used sequential SAT solvers are complete, that is, can decide whether the input formula is satisfiable or not, then the overall distributed method should be complete as well.
- The methods are usable in a wide variety of grid infrastructures.
- There is minimal (or no) communication between grid nodes during solving. This is important because in grids communication delays are significant and support for process-to-process communication is very limited due to security reasons.
- The methods are fault-tolerant and able to recover and maintain completeness even if a substantial number of grid processes fail.

Several parallel SAT solvers designed to work in a distributed networked environment have been described in the literature [9,10,11,12,13,14]. However, in these approaches it is not possible to exploit a chosen sequential SAT solver directly but they are based on developing a special purpose SAT solver for the distributed case. Moreover, distribution of work and load balancing are fairly tightly coupled with the decisions made in the SAT solver and a significant amount of inter-process communication is needed.

A straightforward way to satisfy the requirements above is to use an approach we call *Simple Distributed SAT* (SDSAT) where a SAT instance is solved by running a number of SAT solvers on the same instance as independent jobs and waiting until one of them solves the problem. If a randomized SAT solver is available (for example, `Satz-rand`, `WalkSAT`, or `AdaptiveNovelty+`), this solver can be used with different seeds. This approach has potential as results, for example, in [15,16] indicate.

However, an obvious weakness of SDSAT is that there is no cooperation between the independent jobs, that is, there is no way that progress and partial results obtained in one job can contribute to completing another job. This is a serious shortcoming especially in a grid environment because each process run in a grid needs to be given resource bounds (CPU time, memory) when the process is sent to be executed. Thus, the natural way of running SDSAT by starting jobs without resource bounds and waiting until one of them succeeds is not available in grids. Moreover, job management in a grid typically takes into account the resource requirements of a job, implying lower priority to jobs with substantial resource demands and, hence, longer delays.

In this paper we present a novel distributed SAT solving method called *scattering*, which satisfies the requirements above but still allows for cooperation. The basic idea is to divide a given SAT instance gradually to increasingly more constrained subproblems that are sent to the grid to be solved using practically any available SAT solver. While this is somewhat similar to SDSAT, there are substantial differences.

- The subproblems to be solved become increasingly constrained as the computation proceeds and, hence, computationally easier to solve.
- Learning techniques used in sequential solvers can be exploited when dividing a problem to subproblems.
- Jobs exceeding given resource bounds are interrupted and divided into more constrained and computationally manageable subparts.
- The division of a problem to subproblems is based on estimating the computational cost of the subproblems in order to achieve better load-balancing.

We have implemented the method for NorduGrid [17] (`http://www.nordugrid.org/`), a widely distributed Scandinavian computational grid consisting of mainly PC clusters running the Linux operating system.

The rest of the paper is structured as follows. Section 2 describes the scattering method for distributed SAT solving. Section 3 explains how the method has been implemented on NorduGrid and Sect. 4 reports experiments on the feasibility of the approach.

2 Algorithm

The goal is to develop an algorithm for solving challenging SAT instances on a variety of grids. In order to be able to employ a wide range of grid infrastructures, we make minimal assumptions regarding the *distributed execution environment* provided by the grid. Such an environment is assumed to offer a simple interface between the client sending *executions*, that is, executable programs together with their inputs, and the environment receiving and running the executions. The only functionalities available to the client are

- Send, which sends an execution to the environment,
- Monitor, which reports the state of the execution, and
- Receive, which returns the result of an execution.

It is straightforward to implement these functionalities in practically any grid environment. Notice that apart from the capability of monitoring the state of the executions, the environment is not assumed to support any communication from the client to the environment. We do not either assume that the executions are able to communicate directly with each other. In addition to defining the interface, we also assume that the distributed execution environment has some maximum amount of simultaneous executions it can hold. If this limit is reached, the environment is *saturated* and any new executions sent to the environment may fail without a result.

The executions are required to be autonomous in the sense that once the execution has been constructed by the client and sent to the environment, the execution cannot be further guided. The execution must finish when some condition given at its construction time is triggered, for example, when a given CPU time or memory limit is exceeded.

Such an environment is well suited to running the simple distributed SAT (SDSAT) scheme. However, SDSAT is not optimal for solving highly resource intensive problems in a grid because of the resource bounds and job management policies in grids explained in the introduction.

In order to address deficiencies of the SDSAT approach we have developed a distribution method called *scattering* for solving SAT instances in the simple distributed execution environment described above. The basic ideas underlying scattering are quite straightforward.

- A SAT problem instance is divided to a set of subproblems by adding new constraints (clauses) to the original problem to make the subproblems easier to solve.
- The division of a problem to subproblems (a scattering step) is done so that (i) if all subproblems have been solved, we get a solution to the original problem, and (ii) search spaces of the subproblems are disjoint.
- An execution of a subproblem is interrupted if a given resource bound is exceeded and the problem is divided further.

Next we explain (i) the basic scattering rule and the resulting scattering tree, (ii) the technique employed to generate subproblems with comparable estimated computational cost, (iii) the heuristic used to select constraints to be added, and (iv) methods for exploiting learning in scattering.

2.1 The Basic Scattering Rule and the Scattering Tree

The scattering rule constructs a parameterized number sf of scattered propositional formulas $F_1, \ldots, F_{sf}$ from a formula F such that

$$F_i = \begin{cases} F \wedge T_1 & \text{if } i = 1 \\ F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{i-1} \wedge T_i & \text{if } 1 < i < sf \\ F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{sf-1} & \text{if } i = sf, \end{cases} \tag{1}$$

where each T_i is a conjunction $l_1^i \wedge \cdots \wedge l_{d_i}^i$ of d_i literals selected by a heuristic method (explained in Sect. 2.3) and each d_i is selected to yield comparably sized subproblems (as described in Sect. 2.2). The expression $\neg T_i = \bar{l}_1^i \vee \cdots \vee \bar{l}_{d_i}^i$ is the negation of the conjunction T_i. Thus constructed propositional formulas have the properties that

- the disjunction $F_1 \vee \cdots \vee F_{sf}$ is logically equivalent to the formula F, and
- no two formulas F_i, F_j, $i \neq j$, share a satisfying truth assignment.

The idea is to solve a SAT instance F_r by performing a distributed search in a *scattering tree* where the root is F_r and the nodes are formulas obtained by the scattering rule so that the children of a node F are the scattered formulas $F_1, \ldots, F_{sf}$. The search is implemented using the distributed execution environment by sending formulas associated to nodes as jobs to be solved in the environment according to some search strategy. An example of a part of a scattering tree constructed with the scattering rule is given in Fig. 1.

A node of the tree is *computed satisfiable* if the computation of the corresponding job returns from the environment with this answer, or if at least one of the children is computed satisfiable. The node is *computed unsatisfiable* if the corresponding job returns with the answer unsatisfiable, or if all children have been computed unsatisfiable.

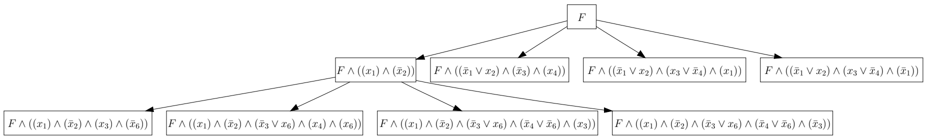

Fig. 1. A part of a scattering tree

The scattering tree is constructed incrementally while sending the jobs (formulas associated with the nodes of the tree) to the distributed execution environment until the root is computed either satisfiable or unsatisfiable.

The correctness of the computed answer follows from the following observations. Since scattered formulas of F are more constrained than F by (1), a satisfying truth assignment for one of the scattered formulas satisfies the formula F as well. Moreover, since the disjunction of the scattered formulas is logically equivalent to the formula F, if all scattered formulas are unsatisfiable, the formula F is also unsatisfiable. Furthermore, if a propositional formula is computed unsatisfiable, all possible scattered formulas, being more constrained, are also unsatisfiable.

2.2 Balancing the subproblems

The basic scattering step on F will fix d_i literals by the conjunction T_i on each scattered instance F_i, $1 \leq i \leq sf - 1$. The idea is to try to divide F to subproblems F_i that have comparable estimated computational costs. Let $t(F)$ denote the estimate of the time required to solve a formula F directly. Then the time required to solve each F_i should be comparable to $\frac{t(F)}{sf}$. Since the solution spaces of the instances are distinct by (1), we may assume that the solving times $t(F_j) = \frac{t(F)}{sf}, 1 \leq j < i$ of the previously constructed problems can be subtracted from the total solving time $t(F)$ to get the time required to solve the remaining problem. Since the time for the problem F_i should also be equal to $\frac{t(F)}{sf}$, a proportion r_i should be constructed from the remaining problem of which the run time is approximately $t(F) - (i-1)\frac{t(F)}{sf}$, yielding the equation

$$\frac{t(F)}{sf} = \left(t(F) - (i-1)\frac{t(F)}{sf}\right) r_i,$$

which, when solved for r_i, becomes

$$r_i = \frac{1}{sf - i + 1}.$$

As an approximation of the computational cost we assume the worst case behaviour, that is, $t(F) = \mathcal{O}(2^n)$ where n is the number of variables in F. If d_i is the number of literals fixed in a scattering step to obtain F_i, then $t(F_i) = \mathcal{O}(2^{n-d_i}) = \mathcal{O}(\frac{1}{2^{d_i}}2^n)$, when assuming direct simplification by unit propagation. Hence, the problem is reduced to that of finding d_i minimizing the difference $|\frac{1}{2^{d_i}} - r_i|$.

A balancing example for $sf = 7$ is given in Table 1. The first row shows values of r_i for $i = 1, \ldots, 7$, the second row shows the values of d_i minimizing the difference $|\frac{1}{2^{d_i}} - r_i|$, and the third row shows the resulting estimated fractions of the full problem.

Table 1. a balancing example for *sf* = 7

i	1	2	3	4	5	6	7
r_i	$\frac{1}{7}$	$\frac{1}{6}$	$\frac{1}{5}$	$\frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{2}$	1
d_i	3	3	2	2	2	1	0
Resulting	0.125	0.109	0.191	0.144	0.108	0.161	0.161

2.3 The Heuristic

The selection of literals for the scattered formulas in (1) is a heuristic process aiming at constructing scattered formulas which have comparable solving times and are more constrained than the formula from which they are scattered. In order to use existing techniques for literal selection, we have implemented the heuristic selection of scattering literals on top of a SAT solver implementation [18] similar to `zChaff` [19]. When selecting the d_i literals for the scattered formula F_i, the procedure takes as input $F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{i-1}$, runs the `zChaff` type algorithm until it reaches the decision level $d_i + 1$ and then selects as the literals $l_1^i, \ldots, l_{d_i}^i$ the d_i decision literals chosen on levels $1, \ldots, d_i$. More precisely this works as follows. After the initial unit propagation, a decision literal is chosen using a modification of the `zChaff` VSIDS heuristic explained below and propagated until either a conflict is derived (in which case the process will backtrack as indicated by the learned clause), or the algorithm has reached the decision level $d_i + 1$, where d_i decision literals have been chosen and successfully propagated[1].

Our modification of the original VSIDS heuristic [19] aims at finding variables that divide the remaining search space into parts of comparable size. The original VSIDS heuristic chooses greedily a literal with the best score. In the modification we use the same scoring function but try to choose literals that have a more balanced score. For this we rank each variable x with the scores of both literals x and $\neg x$ and choose the variable which has the best lower score of all variables, and finally select the literal corresponding to the higher score of this variable as the decision literal.

2.4 Learning in Scattering

On every path in the scattering tree, an ancestor of a formula is less constrained than the child. As a result, all clauses which are logical consequences of the ancestor are

[1] In case the literals cannot be selected, the problem is either unsatisfiable or a satisfiable truth assignment is found, and the scattering can be stopped.

also logical consequences of the child. Since most SAT solvers employ clause learning techniques, which construct logical consequences when the search arrives at a conflict, the clauses learned in the ancestor formulas might also be relevant to the child formulas. This suggests that some learned clauses should be selected for inclusion to the scattered formulas. However, learned clauses from one scattered formula F_i are not necessary logical consequences of another scattered formula F_j. Whereas the clauses from which the learned clauses are derived could be recorded, the effort might overweight the profit gained from the learned clauses.

In order to use the learned clauses in other formulas, we propose a scheme based on the scattering tree. Learned clauses from formula F can be included to the formula F' only if F is ancestor of F' in the scattering tree.

3 Implementation

We have implemented the scattering algorithm presented in Sect. 2 and the additional components required to apply the parallelization scheme in a computational grid. The implementation is named the *SATU (SAT Ubiquitous) distributed SAT solver*. SATU takes as input a propositional formula F, called the *original formula*, and outputs a satisfying truth assignment for F if the formula is shown to be satisfiable, an indication that the formula has no satisfying truth assignment if the formula is shown to be unsatisfiable, or a timeout if the result could not be determined before a given time limit.

The distributed SAT solver SATU consists of five distinct processes communicating with each other through network sockets. The overview of the architecture is presented in Fig. 2. The main functionality of the parallelization scheme described in Sect. 2 is implemented in Search and Scatter. In addition to these, the implementation also includes GridJM, which is an interface to the Grid, and a local satisfiability solver, Filter, acting as a preprocessor which aims at filtering easy jobs from being sent to the Grid. Finally, the implementation contains an intermediate layer SATQueue for communication between Search, Filter, and GridJM. The original formula is given to Search, which sends the formula to SATQueue and constructs the scattered formulas with Scatter. When a formula arrives at SATQueue, Filter tries to solve it until it is sent to GridJM which will finally deliver it to the Grid. The process Search receives the results and uses them to guide the search in the scattering tree. Depicted on the bottom of Fig. 2 are the executions, or jobs, consisting of a SAT solver and a scattered formula.

3.1 Scatter

The process Scatter is a subroutine used by Search. It takes as input a propositional formula and returns a list of scattered formulas. We assume that the construction of scattered formulas is a heuristic process, which, on average, gives better results in terms of even distribution of the run time when more time is given for the calculations relating to the heuristic. Since it is realistic to assume that the initiation of executions in the distributed computing environment takes some amount of time, this time should be used for heuristical calculations in Scatter. The architecture is designed so that a formula is sent early for scattering and the scattered formulas are collected only when resources

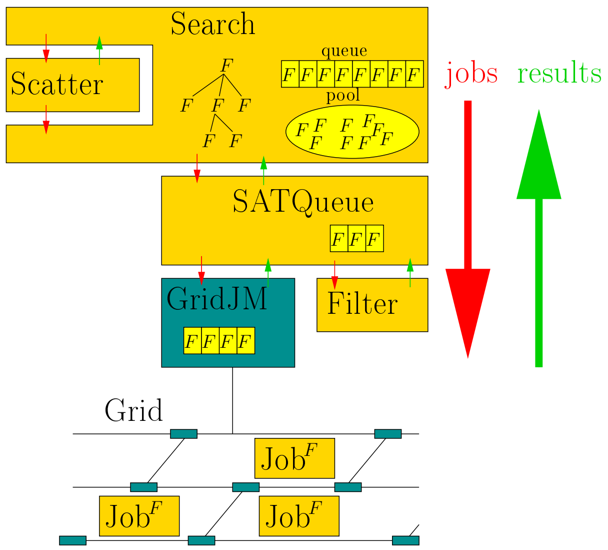

Fig. 2. the program architecture

for the distributed execution become available. This allows for some time for tuning the modified VSIDS heuristic in Scatter before proceeding to the basic scattering step. The tuning is implemented as a `zChaff` style SAT solving which updates the values of literals when new clauses are learned as a result of conflicts in the search. As a side effect, Scatter also learns new clauses, and the clauses can be passed on to the scattered formulas. The implementation supports a *maximum length for learned clauses* to pass. If Scatter finds the formula satisfiable or unsatisfiable during this preprocessing stage, it immediately reports the result to Search. This is an optimization, since the result would be reported by the time Search requests for scattered formulas. However, it was noted early on our experiments that on some easy problems, the time lost in waiting for the request was significant.

3.2 Search

The construction of the scattering tree in Search is implemented as a breadth first search. The search queue pictured in Fig. 2 keeps track of unsearched branches. The scattered formulas are placed into the pool, which is a buffer for delivering the formulas to Grid via SATQueue. The satisfiability of the original formula being solved is computed from the scattering tree maintained in Search. If an ancestor in the scattering tree of some formula in the queue is found unsatisfiable in the Grid, the formula is removed from the queue, significantly pruning the search. New scattered formulas are requested from Scatter only after the number of the formulas in the pool drops below a *pool limit* given as a parameter.

3.3 Communication to the Grid

The rest of the components, SATQueue, Filter and GridJM, deal with the interface towards the distributed execution environment. Main challenges in the interface are the communication delays related to sending the executions to the environment and receiving the results from the environment, and the recovery from the inevitable errors occurring in the environment. The delays make it costly to send executions which only run for a short period of time, and for this purpose the shortest jobs are filtered by the satisfiability solver running in Filter. The errors relating to the distributed environment affect significantly the run time. To lessen the effect, the errors are handled in GridJM by resubmitting the failed executions some fixed amount of times determined by the *resubmission count*. The communication between GridJM, Filter and Search passes through the intermediate process SATQueue.

The jobs sent to the Grid consist of a satisfiability solver and the problem instance to be solved. The solver will terminate in the Grid after a *job specific timeout* if it has not been able to show the instance satisfiable or unsatisfiable. There is a similar limit, the *job memory consumption limit*, for the memory consumption of the solver. GridJM requests jobs from Search until the number of jobs in the distributed computing environment reaches the *saturation limit*, given as a parameter to GridJM. Jobs arriving as replies to these requests are first placed in an incoming job queue. This queue is checked periodically for new jobs, which are then sent to the environment one at a time. When GridJM finds a correctly finished job by monitoring the grid, it forks a new process for receiving the results of the finished job. The receiving is done in parallel in the background. At the beginning of the solving of a difficult formula, it is common that most of the scattered formulas timeout in the distributed execution environment. As a result, the jobs finish approximately at the same time. To avoid this congestion, a whit of randomness is added to the timeout values.

4 Experimental Results

To study the effectiveness of the presented ideas, we tested SATU in NorduGrid [17], a production-level computational grid available in the Nordic countries. The cluster node types, that is, the different types of computers to which we were able to submit jobs during the benchmarking range from 450 MHz Pentium III to 2.6 GHz Pentium(R) 4 and 2.2 GHz AMD Opteron(tm) 248. Due to the dynamic nature of the grid, the set of available nodes changes constantly.

Below we present results of some preliminary tests. More test results are available in [20].

4.1 Scalability

To get some indication of the scalability of SATU with respect to the available resources from the computational grid, we tested it on a set of unsatisfiable 3SAT problems ranging from 350 to 390 variables. Each formula was created so that the clause to variable ratio is $4.258 + 58.26n^{-5/3}$, where n is the number of variables. This is the experimental crossover point of 3SAT [21]. In this test, SATU was run on a 500 MHz Pentium

III processor, except for Filter, which was running on a 2100 MHz AMD Athlon(tm) XP 2800+. The results were obtained with scattering factor $sf = 7$ and the *maximum length for learned clauses* of 10 literals in Scatter, *pool limit* of 8 in Search, *resubmission count* of one in GridJM and a *job memory consumption limit* of 1000 MB. The *job specific timeout* was randomly selected from the range between 50 and 70 minutes and the *timeout for SATU* was 12 hours. For each variable count we constructed a set of formulas and an amount of 3 to 6 unsatisfiable formulas were identified from each set. These formulas were solved with *saturation limit* ranging from 4 to 64. The results are presented in Fig. 3.

The initial tests suggest that the formula solving scales well up to the saturation limit of 16, after which the speedup grows more slowly. The behaviour is explained by the observation that the actual parallelism in the grid does not reach the saturation limit on larger values than 16. The effect of increasing parallelism seems more profound in the more difficult jobs. Especially the minimum speedup at problems with 350 variables is low compared to the speedups of formulas with more variables. This behaviour is expected, given that the longer run times of formulas will give more actual parallelism, as the sending delays do not dominate.

4.2 Comparison between SATU and SDSAT

In order to estimate the effectiveness of scattering we compared it to an idealized version of the simple distributed SAT method (SDSAT) where we ran a randomized version of the same SAT solver that is used in SATU multiple times and took the minimum run time in CPU seconds as the performance estimate. Of course, this is a very optimistic estimate because it does not include any overhead related to grid communication and job management or scheduling delays caused by other computations in the grid.

For the benchmarks, we used a selection of difficult propositional formulas from the SAT2002 competition [22], available at `http://www.satlib.org/`. These benchmarks were used to test also the GridSAT satisfiability solver [10]. The benchmarks were first computed using SATU with the same parameters as in Sect. 4.1, with the exception of GridJM running in a 1 GHz AMD Athlon(tm) processor, Filter running in the 500 MHz Pentium III, the *job specific timeout* being randomly selected from the range of 60 to 80 minutes, *timeout for SATU* set to 48 hours, and the *saturation limit* set to the constant 64. For SDSAT we used an instance specific timeout value which was the run time of SATU multiplied by three. To get the SDSAT results, each formula was run 64 times on `zChaff` version `Chaff 2004.11.15 simplified` which was compiled to use randomness. The memory limit for `zChaff` was set to 1024 MB. The SDSAT runs were done on a computer running AMD Athlon(tm) 64 Processor 3200+ at 2000 MHz.

We tried to locate the benchmarks used for GridSAT in [10] with no success. Hence, we used those benchmarks from `http://www.satlib.org/` whose names matched those reported in [10]. From these benchmarks SATU solved the following in less than 8 minutes: bart15, cache_05, cnt09, comb2, dp10u09, dp12s12, homer11, homer12, ip38, lisa20_1_a, w10_75. Results from the more difficult instances we have studied are presented in Table 2, in which we report SATU run times, the idealized SDSAT run time

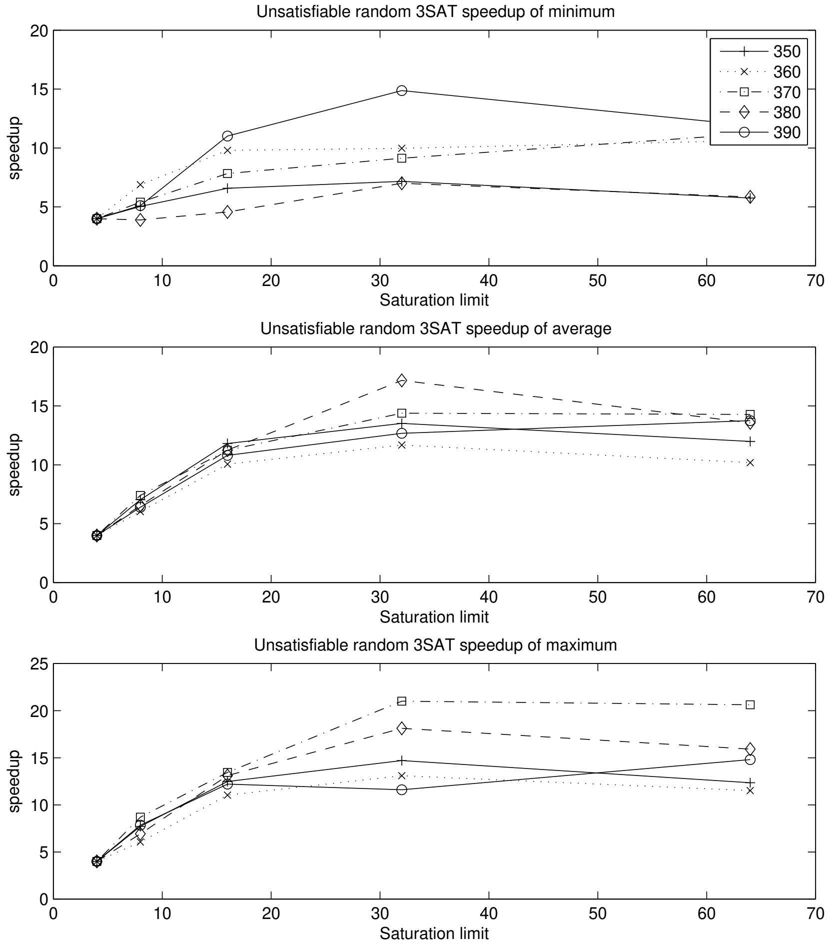

Fig. 3. scalability for unsatisfiable random 3SAT problems. Speedup compared to 4 grid nodes

estimate, and the number of SDSAT runs that had a shorter run time than that of SATU.

The results indicate that SATU performs surprising well in comparison to the minimal run time of the idealized SDSAT. Despite the communication and job management overhead and delays in scheduling caused by other jobs in the grid, SATU handles rather efficiently benchmarks where the idealized minimal run time is low. Some problems, notably Mat26, lisa21_3_a and vda_gr_rcs_w9, which seem to be consistently difficult for SDSAT are easier for SATU. This suggests the existence of formulas for which the fast determination of satisfiability using SDSAT is unlikely, and which would thus be difficult to solve in a grid environment using the basic SDSAT approach.

Table 2. comparison between SATU and SDSAT

Name	SATU (s)	SDSAT (s)	# SDSAT $\leq SATU$	Result
cnt10	1121	540	57	satisfiable
dp10u09	586	149	19	unsatisfiable
f2clk_40	8078	4831	34	unsatisfiable
lisa20_1_a	240	1	20	satisfiable
lisa21_3_a	1017	206	5	satisfiable
Mat26	1503	4151	0	unsatisfiable
vda_gr_rcs_w8	1160	1	5	satisfiable

4.3 The Unsolved Problems from SAT2005

In order to evaluate SATU on benchmarks that are challenging for current state-of-the-art sequential SAT solvers we used problem instances from the SAT2005 satisfiability solver competition (`http://www.satcompetition.org/2005`). An interesting class of problems are those formulas that were not solved by any of the participating solvers during the competition.

From the set of unsolved formulas, we took a set of instances which during the competition were tried with at least ten of the participating solvers but were solved by none. The results are given in Table 3. The parameters for SATU were the same as in Sect. 4.2. A total of seven problems from the nine for which the solving was tried were solved before the timeout of 48 hours.

Table 3. selected unsolved formulas from SAT2005 solver competition

Name	time (s)	Result
eulcbip-9-UNSAT	172800	timeout
gensys-ukn007	10701	unsatisfiable
gensys-ukn008	9422	unsatisfiable
2c-rand3bip-sat-230-1	1060	satisfiable
2c-rand3bip-sat-230-3	1365	satisfiable
2c-rand3bip-sat-240-2	3053	satisfiable
2c-rand3bip-sat-250-1	2264	satisfiable
mod2c-rand3bip-sat-250-3	2641	satisfiable
phnf-size10-exclusive-equilarge_m1	172800	timeout

5 Conclusions

The paper presents a novel method for distributed SAT solving, called scattering, describes how it is implemented in the SATU system on a production-level Scandinavian computational grid called NorduGrid, and studies the performance of SATU using benchmarks from previous SAT competitions. Scattering differs from other distributed

SAT solving methods [9,10,11,12,13,14] in a number of ways: (i) any SAT solver, including industrial black box solvers, can be used with no modifications, (ii) it has modest requirements for communication but still allows process coordination, (iii) when the solving of a problem needs to be distributed, it is possible to divide the problem to not just two but to an arbitrary number of subproblems, and (iv) in scattering, heuristics for dividing a problem to subproblems and for managing the overall distributed search are separated from the heuristic for solving individual subproblems.

Interesting topics of future work include the optimization of the scattering algorithm and the grid job management. Also a more thorough comparison between the SDSAT method and SATU might provide some useful hints on how to combine the benefits of both approaches.

References

1. Kautz, H., Selman, B.: Planning as satisfiability. In: ECAI 1992, John Wiley and Sons (1992) 359–363
2. Kautz, H., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. In: AAAI/IAAI 1996, AAAI Press (1996) 1194–1201
3. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
4. Bjesse, P., Leonard, T., Mokkedem, A.: Finding bugs in an Alpha microprocessor using satisfiability solvers. In: CAV 2001. Volume 2102 of LNCS., Springer (2001) 454–464
5. Larrabee, T.: Test pattern generation using Boolean satisfiability. IEEE Transactions on Computer-Aided Design **11**(1) (1992) 6–22
6. Biere, A., Kunz, W.: SAT and ATPG: Boolean engines for formal hardware verification. In: ICCAD 2002, ACM (2002) 782–785
7. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult instances of boolean satisfiability in the presence of symmetry. IEEE Transactions on Computer Aided Design **22**(9) (2003) 1117–1137
8. Zhang, H., Li, D., Shen, H.: A SAT based scheduler for tournament schedules. In: SAT 2004. (2004) Online proceedings at `http://www.satisfiability.org/SAT04/programme/index.html`.
9. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation **21**(4) (1996) 543–560
10. Chrabakh, W., Wolski, R.: GridSAT: A chaff-based distributed SAT solver for the grid. In: SC 2003, IEEE (2003)
11. Jurkowiak, B., Li, C., Utard, G.: A parallelization scheme based on work stealing for a class of SAT solvers. Journal of Automated Reasoning **34**(1) (2005) 73–101
12. Sinz, C., Blochinger, W., Küchlin, W.: PaSAT — Parallel SAT-checking with lemma exchange: Implementation and applications. In: SAT 2001. Volume 9 of Electronic Notes in Discrete Mathematics., Elsevier (2001) 12–13
13. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Journal of Parallel Computing **29**(7) (2003) 969–994
14. Forman, S., Segre, A.: NAGSAT: A randomized, complete, parallel solver for 3-SAT. In: SAT 2002. (2002) Online proceedings at `http://gauss.ececs.uc.edu/Conferences/SAT2002/sat2002list.html`.

15. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24**(1/2) (2000) 67–100
16. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1-2) (2001) 43–62
17. Eerola, P., Konya, B., Smirnova, O., Ekelöf, T., Ellert, M., Hansen, J.R., Nielsen, J.L., Wäänänen, A., Konstantinov, A., Ould-Saada, F.: Building a production grid in Scandinavia. IEEE Internet Computing **7**(4) (2003) 27–35
18. Zhang, L.: SAT-solving: From Davis-Putnam to Zchaff and beyond, lecture notes (2003) Available online at `http://research.microsoft.com/users/lintaoz/SATSolving/satsolving.htm`.
19. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC 2001, ACM (2001) 530–535
20. Hyvärinen, A.E.J.: SATU: A system for distributed propositional satisfiability checking in computational grids. Research Report A100, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland (2006)
21. Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in random 3-SAT. Artificial Intelligence **81**(1–2) (1996) 31–57
22. Simon, L., Berre, D.L., Hirsch, E.A.: The SAT2002 competition report. Annals of Mathematics and Artificial Intelligence **43**(1–4) (2005) 307–342