Incorporating Learning in Grid-Based Randomized SAT Solving

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

Helsinki University of Technology TKK
Department of Information and Computer Science
email: {Antti.Hyvarinen,Tommi.Junttila,Ilkka.Niemela}@tkk.fi

Abstract. Computational Grids provide a widely distributed computing environment suitable for randomized SAT solving. This paper develops techniques for incorporating learning, known to yield significant speed-ups in the sequential case, in such a distributed framework. The approach exploits existing state-of-the-art clause learning SAT solvers by embedding them with virtually no modifications. We show that for many industrial SAT instances, the expected run time can be decreased by carefully combining the learned clauses from the distributed solvers. We compare different parallel learning strategies by using a representative set of benchmarks, and exploit the results to devise an algorithm for learning-enhanced randomized SAT solving in Grid environments. Finally, we experiment with an implementation of the algorithm in a production level Grid and solve several problems which were not solved in the SAT 2007 solver competition.

1 Introduction

In this paper we consider solving hard propositional satisfiability (SAT) problems in a grid-like, widely distributed computing environment. One realistic example of this kind of an environment is NorduGrid (`http://www.nordugrid.org/`) that we use in some of the experiments of this paper. Compared to, for example, a cluster of locally connected computing elements (CEs), such a widely distributed, multi-party owned environment may pose several restrictions. First, communication with the computing elements (submitting jobs and retrieving their results) can take a significant amount of time due to the widely distributed nature of such an environment. In addition, job management (for example, finding available elements by communicating with the front-end machines) causes further delays when submitting jobs. Second, communication between the CEs is not necessarily allowed at all due to security reasons; typically, all the traffic to the CEs passes through a front-end machine and one can only submit jobs, query their status, and retrieve results. Third, in order to ensure fairness between multiple users, CEs can either impose strict resource limits for jobs (for example, the maximum running time is set to four hours) or prefer jobs with predefined resource limits in a way that makes running of jobs requiring unlimited resources very slow. Fourth, computing elements (and thus jobs) are more likely to crash because they are administrated by different parties; e.g. maintenance breaks of CEs are scheduled independently, and CEs can disappear from the environment if their owner decides to prioritize local use at some time.

In this paper we propose an approach called *Clause Learning Simple Distributed SAT* (CL-SDSAT) for solving *hard* SAT problems in such a widely distributed environment. The approach is based on running state-of-the-art, clause learning randomized SAT-solvers in a parallel environment until one of them solves the problem. In order to solve hard problems in the presence of resource limits imposed on jobs, the approach exploits the work done in *unsuccessful* jobs (i.e. those that exceeded the resource limits without finding a solution) by transferring some of the clauses learned by the solver back to the master process. When new jobs are submitted later, some of the learned clauses are passed to the jobs to constrain the search of the solver. This approach enables *parallel learning* techniques where, on one hand, learned clauses from multiple independent unsuccessful jobs are combined and, on the other hand, the clauses learned from such combinations are *cumulated*. The proposed approach can tolerate all the above mentioned restrictions related to the distributed environment as (i) a reasonable amount of data is transferred back to the master process only at the end of the execution of a job, (ii) the jobs do not communicate with each other, (iii) each job has predefined resource limits for the time and memory it is allowed to consume, and (iv) CE failures do not affect the correctness or completeness of the approach. In addition, only very modest modifications are required in a SAT solver in order to use it in the approach; thus it should be relatively easy for the approach to exploit the future improvements in clause learning SAT solvers.

Related Work. A major approach to distributed SAT solving is based on techniques relying on tight inter-process communication (for example, [1,2,3,4,5,6]). However, this approach is not directly applicable in grid environments we consider, where inter-process communication is often restricted and expensive. Methods for distributed SAT solving not based on inter-process communication have also been proposed [7,8,9]. The CL-SDSAT method developed in this work is similar to [7] as it does not involve search space partitioning like in [8,9] but it extends [7] by incorporating distributed clause learning. When the search space is partitioned, the methods are usually based on *Guiding Paths* [1,2,3]. Examples where clause learning is incorporated to the Guiding Path technique include [4,5] where the communication is performed between threads, [6] which is based on an MPI-implementation, and [10,11] where communication is performed in a more grid-like environment. The CL-SDSAT approach presented in this paper has the advantage of being able to use any clause learning SAT solver with no major changes. This is different from most approaches based on Guiding Paths and enables CL-SDSAT to exploit directly any future advances in SAT solver technology. Similar advantages are obtainable with approaches such as [8]. Distributed learning strategies are studied in [5,12,6]. In [5], the learning is based solely on the length of the clauses, whereas [6] considers several different techniques relating the learned clauses to the Guiding Paths of each solver. The learned clause distribution approaches of [6,5] exchange the learned clauses between the active jobs via a global master store dynamically, requiring modifications to SAT solvers and frequent communication. The distributed learning strategy in CL-SDSAT, on the other hand, cumulates and filters learned clauses over time. It only distributes and collects the clauses when jobs start and terminate due to resource limit, respectively, requiring much less communication and allowing jobs to have predefined resources limits.

The rest of the paper is structured as follows: Section 2 discusses the concepts used later in the presentation. In Sect. 3, basic ideas are introduced for the CL-SDSAT-framework. The issues related to the design of parallel learning are addressed in Sect. 4, and the developed ideas are implemented and evaluated in a production-level Grid environment in Sect. 5. Finally, the conclusions are presented in Sect. 6.

2 Preliminaries: Clause Learning, Randomization, and SDSAT

Most modern complete SAT-solvers, such as ZChaff [13] and MiniSAT [14] to name just two, are based on the Davis-Putnam-Logemann-Loveland (DPLL) depth-first search algorithm [15,16]. Modern implementations of the algorithm work in two alternating phases: an efficiently implementable *unit propagation*, and a heuristic selection of *decision variables* corresponding to branching in a depth-first search. To boost the search, *conflict driven clause learning* search space pruning techniques [17,18] are usually incorporated: whenever the solver reaches a conflict during the search, it analyzes the conflict and learns a new clause C that is a logical consequence of the original SAT instance $\mathcal{F}$. The solver then conjuncts the learned clause C with $\mathcal{F}$, guaranteeing that the search will not enter a similar conflict again. As C is a logical consequence of $\mathcal{F}$, a truth assignment for $\mathcal{F}$ satisfies $\mathcal{F}$ if and only if it satisfies $\mathcal{F} \wedge C$. Learned clauses usually decrease the number of decisions corresponding to the branches of the search. However, since a new clause is learned at each conflict, adding all of them into the instance $\mathcal{F}$ permanently would quickly exhaust the available memory and also slow down the unit propagation search space pruning routine forming the inner loop of the DPLL-algorithm. To avoid the exhaustion, the solvers periodically forget some of the learned clauses.

In addition to clause learning, most modern SAT solvers also apply search *restarts* and some form of *randomization* to avoid getting stuck at hard subproblems [19]. For instance, MiniSAT version 1.14 restarts the search periodically (the learned clauses are not discarded at restarts, though) and makes two percent of its branching decisions (pseudo)randomly. Despite restarts and randomness, the run times of a SAT solver can vary significantly on a single instance. As an example, observe the hundred samples based approximations of the cumulative run time distribution of the instances given in Figs. 2, 3 and 4 (the "base" plots, $q(t)$ is the probability that the instance is solved within t seconds): depending on the seed given to the pseudo-random number generator of MiniSAT 1.14, the run time varies from less than a hundred seconds to several thousands of seconds for some instances. This non-constant run time phenomenon can be exploited in a parallel environment. If we can run N randomized SAT solvers in parallel, the cumulative run time distribution $q(t)$ of an instance is improved to $q_N(t) = 1 - (1 - q(t))^N$: as an example, if $q(1000) = 0.5$ meaning that half of the runs end within 1000 seconds, then $q_8(1000) > 0.99$ and, thus, one obtains the solution almost certainly within 1000 seconds if eight parallel computing elements are available. For a more detailed analysis of running randomized SAT solvers in a parallel, distributed environment involving communication and other delays, see e.g. [7]. Although this simple strategy of running randomized SAT solvers in parallel, which we call the *Simple Distributed SAT solving* (SDSAT) approach, can reduce the expected time to solve an instance, it cannot reduce it below the minimum running time (i.e. the smallest t for which $q(t) > 0$). This is a serious drawback when solving *hard SAT problems* in a grid-like environment where the computing elements usually impose an upper limit for the computing time available for a single job. For example, if CEs only allow four hours of CPU time for each job, the basic SDSAT approach simply *cannot solve any problem with a longer minimum running time*. Notice that the "straightforward" approach of storing the memory image of a solver execution just before the time limit is reached and then continuing the execution in a new job in another CE is not a viable solution due to the amount of data that should be transferred between the jobs.

3 Clause-Learning Simple Distributed SAT Solving

The basic idea of the proposed *Clause-Learning Simple Distributed SAT* (CL-SDSAT) approach is relatively straightforward. A *master process* submits *jobs* consisting of a randomized SAT solver $\mathcal{S}$ and the SAT instance $\mathcal{F}$ to be solved into a grid-like distributed environment DE, which consists of computing elements (CEs) performing computations. If a job solves the problem (i.e. the satisfiability of $\mathcal{F}$ is decided) within the resource limits, the whole algorithm terminates with the solution. If the solution is not found in the job, some of the clauses the solver has learned during its search are transferred back to the master process. The master process maintains a database of such clauses, and whenever a new job is submitted, a subset of the clauses in the current database is conjuncted with $\mathcal{F}$ in the submitted SAT instance. The purpose of this is to ensure that the work done in unsuccessful jobs is not entirely wasted but can be used to prune the search in the subsequent jobs. Another advantage is that a form of parallel, history-dependent learning is enabled as clauses from several independent runs are collected together and cumulated over time. In the following we explain the proposed approach in more detail; the next sections then study different strategies for selecting the subset of learned clauses passed to the jobs and provide some experimental results.

The framework for the approach is presented in pseudo-code in Fig. 1. The learned clauses are collected to a database of clauses, denoted by *ClauseDB*, which is initially empty. The database is allowed to vary in size, but has an imposed maximum size, *MaxDBSize*. From this database, a subset of size at most *SubmSZ* is provided to each solver instance $\mathcal{S}$ together with the original SAT instance $\mathcal{F}$.

The main loop of the framework consists of two concurrent tasks: submitting new jobs to idle CEs in the distributed environment and receiving the results of the finished jobs.

The submission of new jobs is described on lines 2–4. If there are idle CEs in the environment (line 2), then a job $\langle \mathcal{S}, \mathcal{F} \cup \text{Choose}(\textit{ClauseDB}, \textit{SubmSZ}) \rangle$ is submitted to it (line 4).[1] The function Choose selects a subset of the clauses in the current database *ClauseDB* so that the size of the subset is at most *SubmSZ*.[2] The size of the subset is restricted for two reasons: transferring data in a widely distributed environment takes non-negligible time and, as mentioned in Sect. 2, having an excessive amount of learned

[1] As usual, we use the notation $\mathcal{F} \cup \mathcal{C}$ to denote the conjunction $\mathcal{F} \wedge \bigwedge_{C_i \in \mathcal{C}} C_i$.

[2] By the size of a set of clauses we mean here and in the following the sum of the number of the literals in the clauses in the set.

```
Input: F, a SAT instance; S, a randomized SAT solver
   let ClauseDB = ∅
   let MaxDBSize = M
   let SubmSZ = N
1  while (True):
2    if there are idle CEs in DE:
3      update SubmSZ
4      submit the job ⟨S, F ∪ Choose(ClauseDB, SubmSZ)⟩ to an idle CE
5    if ⟨result, C⟩ is received from DE:
6      if result is in {SAT, UNSAT}:
7        return result
8      else
9        update MaxDBSize
10       let ClauseDB = Merge(ClauseDB, C, MaxDBSize)
```

Fig. 1. A general framework for Grid-based randomized learning SAT solving

clauses can slow down the inner loop of the SAT solver $\mathcal{S}$. For the sake of completeness of the approach, the size limit *SubmSZ* may have to be increased during the search (line 3); this issue is discussed later at the end of this section.

The results received from the DE are handled on lines 5–10 with two cases.

- If the result is either SAT or UNSAT, the algorithm terminates with that result (line 7). The correctness of the result in this case, that is, the soundness of the framework, follows directly from the properties of learned clauses: a SAT instance $\mathcal{F} \cup \text{Choose}(\textit{ClauseDB}, \textit{SubmSZ})$ submitted to a CE is satisfiable if and only if the original instance $\mathcal{F}$ is satisfiable because all the clauses in $\text{Choose}(\textit{ClauseDB}, \textit{SubmSZ})$ are learned clauses and, thus, logical consequences of $\mathcal{F}$.
- If the problem was not solved in the job because the resource limits were exceeded, the clause database *ClauseDB* is updated with the set $\mathcal{C}$ of learned clauses returned from the job (line 10). The function Merge takes the old clause database *ClauseDB*, the new learned clauses $\mathcal{C}$, and the maximum size *MaxDBSize* of the database, and returns a new database $\textit{ClauseDB}' \subseteq \textit{ClauseDB} \cup \mathcal{C}$ of size at most *MaxDBSize*. For the sake of completeness (discussed below), it may become necessary to increase the maximum size of the clause database during the search (line 9).

When instantiating the framework into a concrete implementation, of special interest are the heuristics used in the two operators Choose and Merge. We will study some natural heuristics for these operators in the next section.

Completeness. If the resource limits for the jobs as well as the size limits for the clause database (*MaxDBSize*) and submitted learned clause sets (*SubmSZ*) are fixed, the framework is not complete (that is, there are SAT instances for which the framework does not terminate). One way to ensure completeness under fixed resource limits for jobs is to increase the parameters *MaxDBSize* and *SubmSZ* periodically during the search until a solution is found. Naturally, the Choose and Merge operators must use this increased space by returning clause sets of analogously increasing size. Observe the similarity to clause learning SAT solvers: they also usually increase the limit for the number of stored learned clauses gradually during the search.

4 Parallel Learning Strategies

The CL-SDSAT algorithm presented in Sect. 3 provides a framework for parallel learning which builds on five elements: (i) the size of the clause database, *MaxDBSize*, (ii) the number of learned clauses submitted in each job, *SubmSZ*, (iii) the operator Merge for updating the clause database, (iv) the operator Choose used for selecting the learned clauses for a job, and (v) the selection of learned clauses returned by an unsuccessful job.

A key issue is the case (iv), that is, the design of the operator Choose. The heuristic devised for the operator Choose determines to a large degree the choices in other elements (i), (iii), and (v). Four potential heuristics can be identified for Choose:

- $\text{Choose}_{\text{freq}}$ prefers the most common learned clauses. Such clauses are intuitively good since they are encountered in many jobs.
- $\text{Choose}_{\text{len}}$ prefers short learned clauses. Short clauses are potentially effective in pruning the search space.
- Choose_{123} returns only unary, binary, and ternary clauses. Such clauses are even more effective in pruning the search space but might be rare in practice in some cases.
- $\text{Choose}_{\text{rand}}$ returns a set of clauses which are randomly picked from the set of learned clauses so that each learned clause is returned with equal probability.

This section studies empirically the parallel learning strategies and the CL-SDSAT algorithm in Fig. 1 by (i) first studying the run time of a SAT instance $\mathcal{F}$ before and after including the learned clauses (line 4 of the algorithm), and (ii) then the cumulative effect of such inclusions after several rounds of the **while** loop.

All experiments in this section are performed using a representative set of benchmarks from the SAT 2007 Solver Competition, formed in three steps as follows. First, each of the instances in the industrial and crafted categories of the competition were attempted once with standard MiniSat v1.14 SAT solver, using a time limit of 8000 seconds, on a heterogeneous set of modern CPUs. Second, of these problems, the ones which were solved in more than 2000 seconds (but less than 8000 seconds) were collected for the third round. From the resulting set, a representative subset consisting of nine instances was formed so that there is at least one satisfiable and unsatisfiable instance from both the industrial and the crafted categories. The final benchmark set is presented in column Name of Table 1.

Heuristics for the Operator Choose. The comparison of the heuristics is performed in an idealized, zero-delay environment by running for each benchmark instance eight independent jobs which are all unsuccessful (do not return SAT or UNSAT) as for each job the run time limit is set to be 25% of the observed minimum run time (among hundred randomized runs) of the original instance. For these runs, MiniSat v1.14 was modified so that it accepts as input a seed for its internal search randomization procedure and a time limit, and upon reaching the time limit it outputs all learned clauses it is holding at that time. Based on these eight sets of learned clauses, say, $\mathcal{C}_1, \ldots, \mathcal{C}_8$, a *derived instance* is constructed by employing the respective Choose heuristic to select a subset of the learned clauses. The derived instance corresponds to the instance $\mathcal{F} \cup \text{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})$ of line 4 of the algorithm in Fig. 1, with *SubmSZ* set to 100,000 literals and $\mathit{ClauseDB} = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_8$. It should be noted that the size limitation of *ClauseDB* is ignored by effectively setting *MaxDBSize* to infinity in these experiments.

Table 1 gives an overview of the results by comparing the expected run times of the derived instances for several heuristics and the instance without additional learned clauses (in column Base). The expected run times are computed from fifty sample runs (hundred sample runs in column Base), all ran until a solution was found in a homogeneous computing environment consisting of Intel Xeon 5130 CPUs with 2GHz clock speed and 16 gigabytes of memory. The solver used in the experiments was the MiniSAT v1.14 SAT solver modified so that the solver accepts a seed for its internal search randomization procedure as input.

Table 1. Expected run times for a selection of benchmarks from SAT 2007 competition

Name	Base	$\text{Choose}_{\text{len}}$	$\text{Choose}_{\text{freq}}$	Choose_{123}	$\text{Choose}_{\text{rand}}$
`999999000001nc.shuffled--as.sat05-446`	2065	1436	1462	1431	1357
`AProVE07-16`	1563	1152	1133	1152	1647
`clqcolor-10-07-09.-shuffled-as.sat05-1258`	1900	1465	1447	1458	2016
`cube-11-h14-sat`	4832	5441	5245	5449	5164
`dated-10-13-s`	2278	1705	2069	1706	3283
`mizh-md5-48-5`	1659	1017	1436	1012	1565
`mod2-rand3bip-sat-250-3-.shuffled-as.sat05-2220`	1180	1205	1202	1202	1406
`mod2-rand3bip-sat-280-1-.sat05-2263.reshuffled-07`	2382	5231	5213	5209	5545
`vmpc_28.shuffled-as.-sat05-1957`	623	574	489	579	675
	18482	19226	19696	19198	22658

The results suggest that while the two length-based heuristics perform very similarly in this benchmark set, the frequency-based heuristic differs from these two. None of the three heuristics is consistently better than the other two. The total expected solving time, reported on the last row for each of the heuristics, confirms the similarity of the heuristics. Perhaps surprisingly, the results in column Base outperform others when only the sums are compared. It should be noted, however, that the effect is almost solely due to the instance `mod2-rand3bip-sat-280-1.sat05-`

-2263.reshuffled-07. Similarity of the results can be explained by the fact that short learned clauses are also frequent and usually of length one, two or three. The results also show that there are instances for which the addition of short learned clauses increases the expected solving time. For some problems, namely the two mod2-instances, the phenomenon can be explained by the structure of the instances: they are specially crafted to be difficult for the type of solvers that are being studied here [20]. Finally, the random heuristic is usually inferior to the other heuristics. This confirms that the performance of the CL-SDSAT algorithm can be enhanced by using some specific heuristics in the operator Choose.

The run times of certain instances are examined in detail by computing the distributions $q(t)$, giving the probability that the problem is solved in less than t seconds (or decisions). The effect of the parallel learning is further studied for these instances by using an additional heuristic, Single, which selects the shortest learned clauses from only a single job. The results are given in Figs. 2 to 5, where $\text{Choose}_{\text{rand}}$, Single, $\text{Choose}_{\text{freq}}$, Choose_{123}, and $\text{Choose}_{\text{len}}$ are denoted by random, single, frequency, 123, and length, respectively.

In Fig. 2, the instance benefits from the learned clauses independent of the heuristic, as can also be seen from the results in Table 1. Only the heuristic Single is not useful in decreasing the run time, indicating that the parallelism can be exploited. Figure 3 illustrates a run time distribution of an instance with a different type of a distribution, where the minimum run time is two orders of magnitude smaller than the maximum run time, as opposed to the less dramatic ten-fold difference between the minimum and maximum in Fig. 2. On this instance the effects of the heuristics are similar.

Table 1 suggests further that for some instances, the introduction of new learned clauses increases the expected run time. The phenomenon is further illustrated in Figs. 4 and 5. The run times, shown in Fig. 4, increase in general. However, Fig. 5 shows that the corresponding number of decisions decreases as new clauses are included. The added clauses decrease the number of decisions the solver has to make but in some cases they increase the overhead related to solving the problem, thus increasing the expected solving time.

As stated above, we conclude that $\text{Choose}_{\text{freq}}$, $\text{Choose}_{\text{len}}$ and Choose_{123} usually result in very similar run times. We will concentrate in our further analysis on the heuristic based on $\text{Choose}_{\text{len}}$ for two reasons. Firstly, the length-based heuristics are more efficiently implementable in smaller space, and secondly, $\text{Choose}_{\text{len}}$ is guaranteed to result in learned clauses and, thus, progress in the CL-SDSAT algorithm also in cases where Choose_{123} performs poorly when few unary, binary or ternary clauses are discovered during the search.

Cumulative Effect of Learned Clauses. The results in Table 1 and Fig. 4 clearly show that some instances do not benefit from learned clauses obtained in the early stages of the parallel learning. In the light of this result, it would be possible that some instances are not solvable using the CL-SDSAT algorithm. In this section we study this question on instances described in Table 1 and show that ultimately the introduction of learned clauses helps in reducing the run time of these instances. We restrict our study to instances where the minimum observed run time is large. This restriction follows from

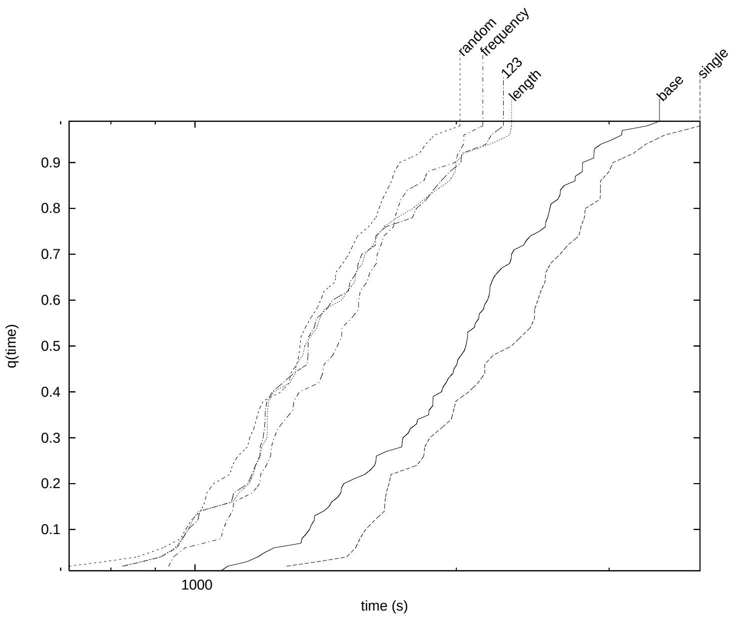

Fig. 2. Run time distributions for the derived instances of `999999000001nc.shuffled--as.sat05-446` corresponding to several heuristics

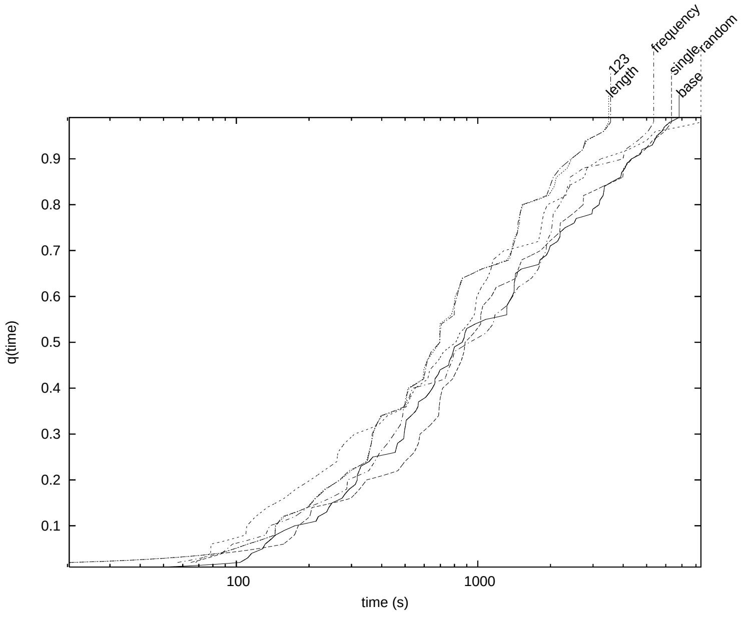

Fig. 3. Run time distributions for the derived instance of `mizh-md5-48-5` corresponding to several heuristics

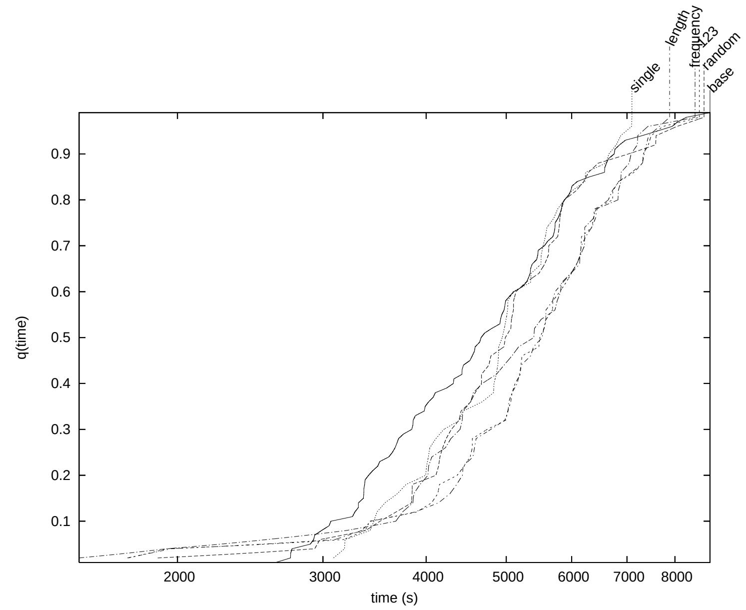

Fig. 4. Run time distributions for the derived instances of `cube-11-h14-sat` corresponding to several heuristics

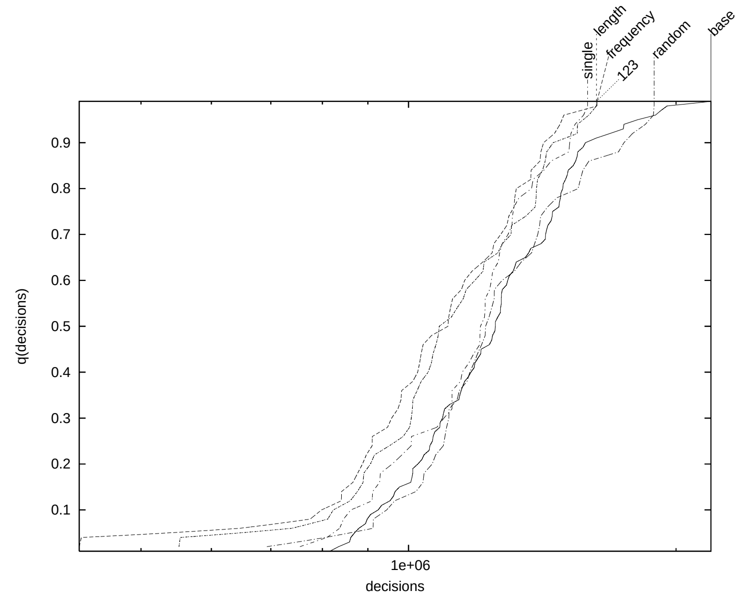

Fig. 5. Decision distribution for the derived instances of `cube-11-h14-sat` corresponding to several heuristics

the observation that if the minimum run time is small (which is the case, for example, for the two `mod2`-instances) the algorithm will solve the problem quickly without the learned clauses, and the question of how the learned clauses affect the run time of the instance is irrelevant as the problem is solved even before any learned clauses can be included to a job in the CL-SDSAT algorithm.

The effect of running the CL-SDSAT-algorithm for several steps is illustrated in Figs. 6, 7, and 8. The experiment is performed using the $\mathrm{Choose}_{\mathrm{len}}$-heuristic for both Choose and Merge to filter learned clauses, a distributed computing environment consisting of 32 CEs, and for each job a run time limit of 25% of the minimum observed run time of the original instance. In all jobs the parameters *MaxDBSize* and *SubmSZ* are set to 100,000 literals. From the resulting jobs twelve are selected evenly from the range between the initial job and the final successful job. The jobs are numbered in order they are created on line 4 of the CL-SDSAT-algorithm, and the numbers are shown in the figures which present the run time distributions of the corresponding submitted SAT instances (based on fifty sample runs). The base distribution for the instance without any learned clauses is included for reference. The results in Figs. 6, 7, and 8 show that the expected run time of an instance gradually decreases when learned clauses are cumulated and added to the instance using the given heuristic for filtering the learned clauses. Notice that this happens also for the instance `cube-11-h14-sat` (Fig. 6) for which combining learned clauses from eight independent jobs only increases the expected run time (recall Fig. 4).

To sum up, the controlled experiments in this section support the conclusion that the parallel learning strategy in the CL-SDSAT-algorithm decreases the expected run time of an instance as learned clauses are iteratively cumulated and filtered when the heuristic is chosen suitably.

5 Grid Implementation

The ideas developed in this work have been implemented in a prototype of the proposed CL-SDSAT framework. The prototype uses NorduGrid, a production level Grid, as the distributed environment, and MiniSAT version 1.14 (with modifiable pseudo-random number generator seed) as the randomized SAT solver. The job management in the Grid is handled by GridJM [21], and each job has a time limit of one hour and a memory limit of one gigabyte. The implementation uses the $\mathrm{Choose}_{\mathrm{len}}$-heuristic preferring the shortest clauses for parallel learning; this design choice is based on the results in Sect. 4 and the fact that length-based heuristics are more efficiently implementable than frequency-based in this framework. In the prototype, the requirements needed for guaranteeing completeness are ignored by simply using a fixed clause database size of 100,000 literals. Similarly, unsuccessful jobs do not return all their learned clauses but only the shortest ones that together have at most 100,000 literals.

For the benchmark problems we selected a set of hard SAT instances for which there was little or no a priori information about the run time distribution. Such problems are available from the SAT 2007 solver competition (`http://www.satcompetition.org/`), where some of the instances were not solved by any of the competing solvers within the time bounds (10,000 seconds for the industrial and 5,000 seconds for the

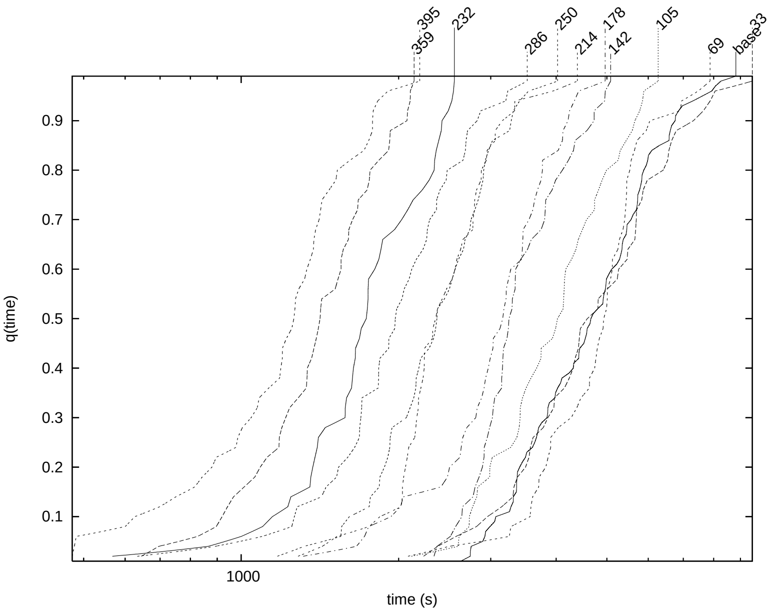

Fig. 6. Run time distributions corresponding to several jobs constructed by CL-SDSAT while solving `cube-11-h14-sat` with $\mathrm{Choose}_{\mathrm{len}}$

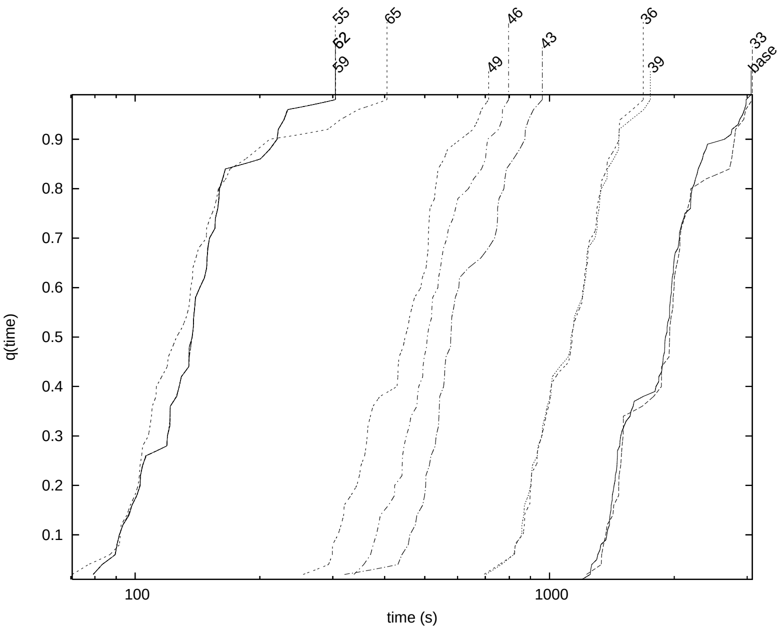

Fig. 7. Run time distributions corresponding to several jobs constructed by CL-SDSAT while solving `clqcolor-10-07-09.shuffled-as.sat05-1258`

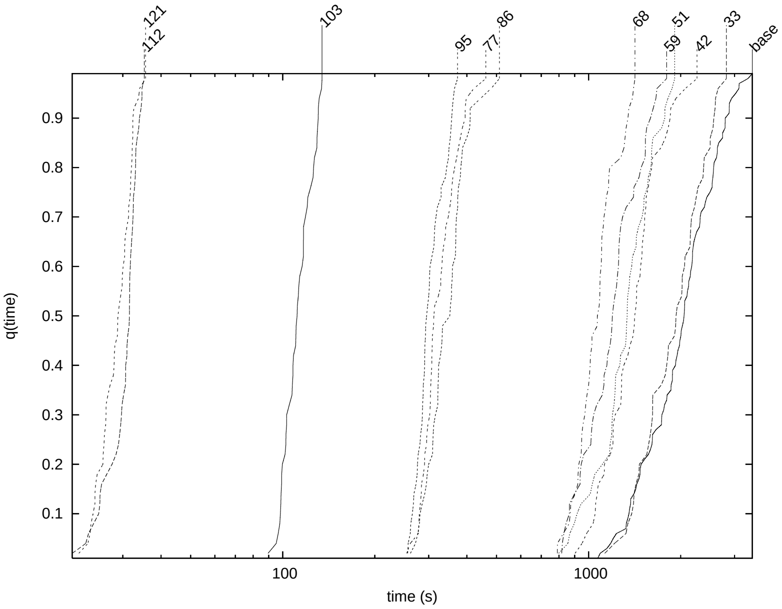

Fig. 8. Run time distributions corresponding to several jobs constructed by CL-SDSAT while solving `999999000001nc.shuffled-as.sat05-446`

crafted category). Table 2 presents the results of running the CL-SDSAT prototype on a subset of these unsolved problems as well as on some other problems which were not solved by MiniSAT in the competition. Each instance was run for at most thirty hours allowing the use of at least eight and at most 64 CEs simultaneously. Column MiniSAT also reports the run times of the sequential MiniSAT v1.14 with no time limit but the memory usage restricted to two gigabytes. The MiniSAT v1.14 runs were performed using dual CPU dual-core compute nodes (Intel Xeon 5130 2GHz) with at most four processes running simultaneously in each node. It should be noted that the exact run times reported in the column Grid in the table are dependent on factors such as the background load of the Grid environment and are, hence, difficult to reproduce.

Two phenomena can be observed from the results. Firstly, some problems, such as `vmpc_33`, are solved in less than one hour with the CL-SDSAT prototype and are, thus, clearly also solvable with the basic SDSAT method [7] with no parallel clause learning. Secondly, and more importantly, the prototype solves, with one hour time limit for each job, several problems which were not solved by *any* solver in SAT 2007 competition in 10,000 seconds. This suggests that the proposed framework with iterative parallel learning also works for very hard problems and causes the cumulative run time distribution to "shift leftwards" (recalling Figs. 6 to 8) as more learned clauses are seen. The other, in our opinion much more unlikely, explanation for this is that the problems have a very small but non-zero probability to be solved in less than one hour and, thus, would have been solved with the basic SDSAT method by using hundreds of parallel solvers.

Table 2. Wall clock times for some difficult instances from SAT 2007 competition solved in Grid and with standard MiniSAT 1.14. Memory outs are denoted by '*', time outs by '—'

Not solved by MiniSAT in SAT 2007			
Name	Type	Grid (s)	MiniSAT (s)
`ezfact64_5.sat05-452.reshuffled-07`	SAT	24956	65739
`vmpc_33`	SAT	1072	184928
`safe-50-h50-sat`	SAT	25551	*
`connm-ue-csp-sat-n800-d-0.02-s1542454144-.sat05-533.reshuffled-07`	SAT	8269	119724
Not solved by any solver in SAT 2007			
Name	Type	Grid (s)	MiniSAT (s)
`AProVE07-01`	UNSAT	14567	39627
`AProVE07-25`	UNSAT	89206	306634
`QG7a-gensys-ukn002.sat05--3842.reshuffled-07`	UNSAT	40239	127801
`vmpc_34`	SAT	38753	90827
`safe-50-h49-unsat`		—	*
`partial-10-13-s.cnf`	SAT	7144	*
`sortnet-8-ipc5-h19-sat`	SAT	95497	*
`dated-10-17-u`	UNSAT	—	105821
`eq.atree.braun.12.unsat`	UNSAT	13298	59229

6 Conclusions

We have proposed a new approach to solving hard satisfiability problems in a grid-like widely distributed parallel environment. The approach can tolerate the severe restrictions imposed on the jobs executed in such an environment, e.g., it requires no inter-node communication and is inherently fault-tolerant. The approach is based on combining (i) a natural method for solving SAT in parallel by independent randomized SAT solvers, and (ii) the powerful conflict driven clause learning technique employed in many modern, sequential, DPLL-style SAT-solvers. This combination results in a novel parallel and cumulative clause learning approach. We have experimentally (i) compared different heuristics for selecting the learned clauses that are dynamically stored during the process, and (ii) demonstrated that the process can gradually make the problem easier to solve. Preliminary experimental results carried out in a production level Grid indicate that the approach can indeed solve very hard SAT problems, including several that were not solved in the SAT 2007 competition by any solver. This suggests that the developed algorithm is also useful in practical environments.

Acknowledgments The authors wish to thank the anonymous reviewers for their valuable comments. The financial support of the Academy of Finland (projects 122399 and 112016), Helsinki Graduate School in Computer Science and Engineering, Jenny and Antti Wihuri Foundation, and Emil Aaltosen Säätiö is gratefully acknowledged.

References

1. Boehm, M., Speckenmeyer, E.: A fast parallel SAT-solver: Efficient workload balancing. Annals of Mathematics and Artificial Intelligence **17**(4-3) (1996) 381–400
2. 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
3. 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
4. Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel multithreaded satisfiability solver: Design and implementation. Electronic Notes in Theoretical Computer Science **128**(3) (2005) 75–90
5. 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
6. Schubert, T., Lewis, M., Becker, B.: PaMira — a parallel SAT solver with knowledge sharing. In: MVT'05, IEEE (2005) 29–36
7. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Strategies for solving SAT in grids by randomized search. In: 9th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2008). (2008) Accepted for publication.
8. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: A distribution method for solving SAT in grids. In: SAT 2006. Volume 4121 of LNCS., Springer (2006) 430–435
9. 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`.
10. Blochinger, W., Westje, W., Küchlin, W., Wedeniwski, S.: ZetaSAT – Boolean satisfiability solving on desktop grids. In: CCGrid 2005, IEEE (2005) 1079–1086
11. Chrabakh, W., Wolski, R.: GridSAT: A chaff-based distributed SAT solver for the grid. In: SC 2003, IEEE (2003)
12. Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., Tamura, N.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154**(16) (2006) 2291–2306
13. 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
14. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT 2003. Volume 2919 of LNCS., Springer (2003) 502–518
15. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7**(3) (1960) 201–215
16. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5**(7) (1962) 394–397
17. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
18. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD 2001, ACM (2001) 279–285
19. 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
20. Haanpää, H., Järvisalo, M., Kaski, P., Niemelä, I.: Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 27–64
21. Hyvärinen, A.E.J.: GridJM. A Computer Program. `http://www.tcs.hut.fi/~aehyvari/gridjm/`.