Journal on Satisfiability, Boolean Modeling and Computation 1 (2008) 1–xx

Incorporating Clause Learning in Grid-Based Randomized SAT Solving[*]

Antti E. J. Hyvärinen antti.hyvarinen@tkk.fi
Tommi Junttila tommi.junttila@tkk.fi
Ilkka Niemelä ilkka.niemela@tkk.fi
Department of Information and Computer Science,
Faculty of Information and Natural Sciences,
Helsinki University of Technology

Abstract

Computational Grids provide a widely distributed computing environment suitable for randomized SAT solving. This paper develops techniques for incorporating clause 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. The paper presents an algorithmic framework for learning-enhanced randomized SAT solving in Grid environments. With a substantial amount of controlled experiments it is demonstrated that this approach enables a form of clause learning which is not directly available in the underlying sequential SAT solver. Finally, an implementation of the algorithm is run in a production level Grid where it solves several problems 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 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

∗ This is an extended version of a conference paper [16].

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 basic idea is quite straightforward: a master process submits jobs consisting of a randomized state-of-the-art clause learning SAT solver and a SAT instance to a distributed computing environment until one of the jobs 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 a form of clause learning which is not directly available in the underlying sequential SAT solver: 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 relative 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.

We devise an algorithmic framework for implementing CL-SDSAT and present techniques for combining and cumulating learned clauses. Experimental results show that adding to a SAT instance cumulated learned clauses obtained in this way can affect considerably the run time distribution of a SAT solver on the instance and can reduce significantly the expected run time. This indicates that the techniques can be effective in solving hard SAT problems. In order to evaluate the potential of the CL-SDSAT approach we have implemented it in a real grid environment and tested it with SAT problems which were not solved in the SAT 2007 solver competition. The implementation is able to solve several of such problems in a realistic setting where the run times of individual jobs are severely restricted.

Related Work. A large part of work on distributed SAT solving is based on tight inter-process communication (for example, [2, 25, 19, 9, 24, 23]). Therefore, the results of these works are not directly applicable in grid environments where inter-process communication is often restricted and expensive. Methods for distributed SAT solving not based on inter-process communication have also been proposed [17, 15, 10]. The CL-SDSAT method developed in this work is similar to [17] as it does not involve search space partitioning like in [15, 10] but it extends [17] by incorporating distributed clause learning. When the search space is partitioned, the methods are usually based on *Guiding Paths* [2, 25, 19]. Clause learning and Guiding Paths are studied in [9, 24] where the communication is performed between threads, in [23] which is based on an MPI-implementation, and in [1, 4] 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 future advances in clause learning SAT solver technology probably in a very straightforward way. Similar advantages can be obtained with search space partitioning using approaches described in [15, 4]. Distributed learning strategies are studied in [24, 18, 23]. In [24], learning is based solely on the length of the clauses, whereas [23] considers several different techniques relating the learned clauses to the Guiding Paths of each solver. The learned clause distribution approaches of [23, 24] 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 distributes learned clauses only when jobs start and collects them when jobs terminate due to resource limits, requiring much less communication and allowing jobs to have predefined resources limits.

Outline. The rest of the paper is structured as follows. First, Sect. 2 reviews key concepts used in the paper including basic properties of modern clause learning SAT solvers and explains how randomizing such a solver leads to a straightforward distributed SAT algorithm that we call Simple Distributed SAT (SDSAT). Section 3 presents the proposed extension of this, the Clause Learning Simple Distributed SAT (CL-SDSAT) framework. The issues related to the design of parallel learning strategies 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

We first introduce some definitions and concepts needed in the rest of the work — formulas, SAT solvers, clause learning, randomization, and the Simple Distributed SAT Solving (SDSAT) approach.

Formulas, Satisfiability, and Simplification. A formula $\mathcal{F}$ in conjunctive normal form (CNF) is a conjunction of clauses, each clause C being a disjunction of literals, while a literal is either a Boolean variable v or its negation $\neg v$. Whenever convenient, we can treat a CNF formula as a set of clauses and a clause as a set of literals; for instance, the formula $\mathcal{F} = (x) \wedge (\neg x \vee \neg y) \wedge (y \vee \neg x \vee z) \wedge (x \vee v) \wedge (y \vee v \vee \neg w)$ can be written as $\{\{x\}, \{\neg x, \neg y\}, \{y, \neg x, z\}, \{x, v\}, \{y, v, \neg w\}\}$. As usual, given a negative literal $\neg v$, we identify $\neg\neg v$ with v. A clause with only one literal is called a *unit clause*.

A *truth assignment* α is a set of literals; α is *inconsistent* if $v, \neg v \in \alpha$ for some variable v and *consistent* otherwise. If $l \in \alpha$, then we say that l is true in α. Similarly, if $\neg l \in \alpha$, then l is false in α. If either $v \in \alpha$ or $\neg v \in \alpha$ (or both), then v is *assigned* in α. A consistent truth assignment α *satisfies a clause* C, denoted by $\alpha \vDash C$, if it makes at least one literal in the clause true, i.e. if $C \cap \alpha \neq \emptyset$. Furthermore, α *satisfies a formula* $\mathcal{F}$, denoted by $\alpha \vDash \mathcal{F}$, if it satisfies each clause $C \in \mathcal{F}$. A formula is satisfiable if there is a truth assignment satisfying it, and unsatisfiable otherwise. A formula $\mathcal{F}'$ is a *logical consequence* of a formula $\mathcal{F}$ if for each truth assignment α, $\alpha \vDash \mathcal{F}$ implies $\alpha \vDash \mathcal{F}'$. In such a case, $\mathcal{F}$ is satisfiable if and only if $\mathcal{F} \wedge \mathcal{F}'$ is. Two formulas are *logically equivalent* if they are logical consequences of each other, i.e. have the same satisfying truth assignments.

In this paper we will use two standard concepts in order to remove some redundancy in large clause sets. First, given a formula (i.e., a set of clauses) $\mathcal{F}$, the set $\mathit{UnitProp}(\mathcal{F})$ of *unit clauses implied by unit propagation* is the smallest set U of unit clauses such that

- if there is a unit clause (l) in $\mathcal{F}$, then $(l) \in U$, and
- if a clause $(l_1 \vee \ldots \vee l_n)$ is in $\mathcal{F}$ and for some $1 \leq j \leq n$, for all $1 \leq i \leq n, i \neq j$: $(\neg l_i) \in U$, then $(l_j) \in U$.

A key property of these literals is that a formula $\mathcal{F}$ is logically equivalent to the formula augmented with the implied unit clauses, i.e. $\mathcal{F} \cup U$. Furthermore, if $\mathit{UnitProp}(\mathcal{F})$ contains two inconsistent unit clauses (v) and $(\neg v)$, then $\mathcal{F}$ is unsatisfiable. Second, given a set U of unit clauses, a formula (i.e. a set of clauses) $\mathcal{F}$ can be *simplified* with respect to U by (i) removing all clauses that contain a literal appearing in U, and (ii) removing from all the remaining clauses all the literals whose negation appears in U. Formally, letting $\hat{U} = \{l \mid (l) \in U\}$ we define

$$\mathit{Simplify}(\mathcal{F}, U) = \left\{C \setminus \left\{\neg l \mid l \in \hat{U}\right\} \mid C \in \mathcal{F} \text{ and } C \cap \hat{U} = \emptyset\right\}. \tag{1}$$

The key property of this simplification is that the formulas $\mathcal{F} \cup U$ and $\mathit{Simplify}(\mathcal{F}, U) \cup U$ are logically equivalent. In particular, we will use the property that if $\mathcal{F}$ is a formula, $\mathcal{C}$ is a set of clauses that are logical consequences of $\mathcal{F}$, and $U = \mathit{UnitProp}(\mathcal{F} \cup \mathcal{C})$, then $\mathcal{F}$ and $\mathit{Simplify}(\mathcal{F}, U) \cup \mathit{Simplify}(\mathcal{C}, U) \cup U$ are logically equivalent.

Example 1. *Given a formula* $\mathcal{F} = (x) \wedge (\neg x \vee \neg y) \wedge (y \vee \neg x \vee z) \wedge (x \vee v) \wedge (y \vee v \vee \neg w)$, *the set of unit clauses implied by unit propagation is* $\mathit{UnitProp}(\mathcal{F}) = \{(x), (\neg y), (z)\}$, *and simplifying* $\mathcal{F}$ *with this set results in* $\mathit{Simplify}(\mathcal{F}, \mathit{UnitProp}(\mathcal{F})) = (v \vee \neg w)$.

SAT Solvers and Conflict-Driven Clause Learning. A SAT solver is a tool for finding a satisfying truth assignment for the formula given as input. The so-called complete SAT solvers are also able to determine when the formula has no satisfying truth assignment. Most modern complete SAT-solvers, such as ZChaff [22] and MiniSat [8] to name just two, are based on the Davis-Putnam-Logemann-Loveland (DPLL) depth-first search algorithm [6, 5]. In the DPLL algorithm, the space of all truth assignments is systematically searched until a satisfying truth assignment is found or it can be deduced that none exists. Basically, the search is performed by extending the current candidate assignment by (i) making decisions, i.e. setting a heuristically selected, currently unassigned variable to a value, and (ii) unit propagation, i.e. assigning all the literals that are implied by unit propagation under the formula and current assignment. When reaching a conflict, i.e, when the current truth assignment becomes inconsistent, the search backtracks to some earlier decision point, undoing all the assignments made since. For a tutorial on DPLL-based solvers, the reader is referred to e.g. [21].

Modern, complete SAT solvers usually incorporate *conflict driven clause learning* search space pruning techniques [20, 26] to boost the search. Whenever the solver reaches a conflict during the search, it analyzes the conflict and learns a new clause C that is a logical

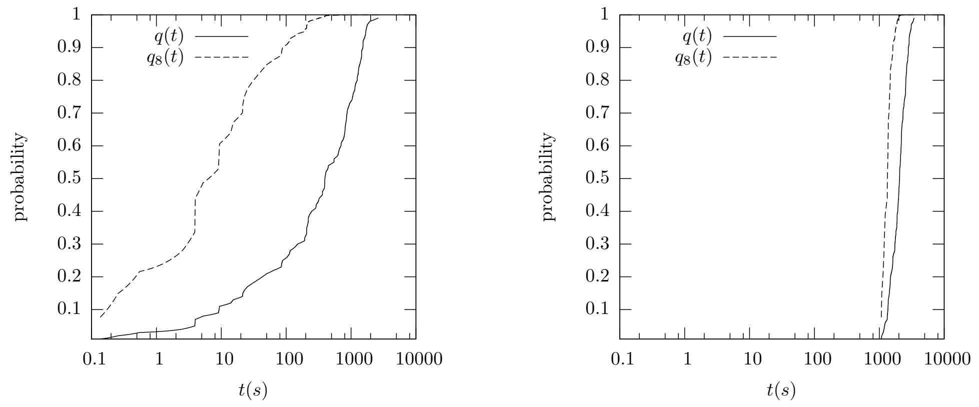

Figure 1. The run time distributions of two instances for single (the $q(t)$ plots) and eight (the $q_8(t)$ plots) randomized SAT solvers.

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. However, in order to remain complete, the solvers also periodically increase the amount of learned clauses kept in the memory.

Randomization and SDSAT. 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 [12]. For instance, MiniSat version 1.14 restarts the search periodically (all 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 approximation of the cumulative *run time distribution* $q(t)$ of an instance given in the left hand side plot of Fig. 1, where $q(t)$ is the probability that the instance is solved within t seconds: depending on the seed given to the pseudo-random number generator of MiniSat v1.14, the run time varies from less than a second to thousands of seconds.

This non-constant run time phenomenon can be exploited in a parallel environment. If N randomized SAT solvers are run 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. This approach can be surprisingly efficient. For example, the instance in the left hand side plot of Fig. 1 is solved within approximately 140 seconds with probability 0.5 when only one solver is used; however, when eight solvers are used, the instance is solved within approximately five seconds with probability 0.5 and within 140 seconds with probability almost one. In fact, the expected run times (approximated based on the hundred sample runs) are 623 seconds for one solver and 31 seconds (that is, around 20 times less) for eight parallel solvers. For a more detailed analysis of running randomized SAT solvers in a parallel, distributed environment involving communication and other delays, see [17].

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 run time (i.e. the smallest t for which $q(t) > 0$). For an example, observe the sequential run time distribution $q(t)$ of an another instance given in the right hand side plot of Fig. 1; the variation of the run time is significantly smaller and the instance seems to have no short run times. Consequently, running eight SAT solvers in parallel (the plot $q_8(t)$) does not reduce the expected run time significantly; in numbers, the (approximated) expected run time for this instance is 2,065 seconds with one solver and 1,334 seconds (i.e., only less than two times faster) for eight parallel solvers. Even more importantly, the *minimum run time stays the same irrespective of how many parallel solvers are employed*. 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 execution. For example, if the computing elements only allow four hours of CPU time for each execution, the basic SDSAT approach simply *cannot solve any problem with a longer minimum run time* because none of the SAT solvers running in parallel can solve the problem within that 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 another computing element is not a viable solution due to the amount of data that should be transferred between the computing elements.

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 r computing elements (CEs) performing computations dictated by the jobs. Each job occupies a CE for a time depending on the background load of the DE, the properties of the job, and the *resource limits* of the job which determine the amount of CPU time and memory the job can use on a CE.

If a job solves the problem (that is, the satisfiability of $\mathcal{F}$ is decided) within the resource limits, the CL-SDSAT 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. Given a clause database at a particular time, the jobs which are constructed by conjoining $\mathcal{F}$ with some of the clauses in the clause database are called *subsequent* to that clause database. Conversely, a job *precedes* a given clause database if the learned clauses of the job have been included to that clause database. The CL-SDSAT algorithm aims at pruning the search space of subsequent jobs by using the learned clauses

```
Input: $\mathcal{F}$, a SAT instance; $\mathcal{S}$, a randomized SAT solver; DE, the environment containing CEs
   let $\mathit{ClauseDB} = \emptyset$
   let $\mathit{MaxDBSize}$ be the initial size of the clause database
   let $\mathit{SubmSZ}$ be the initial size of the learned clauses submitted with the job
   let $U = \mathit{UnitProp}(\mathcal{F})$
1  while (True):
2     if there are idle CEs in DE:
3        update $\mathit{SubmSZ}$
4        submit the job $\langle \mathcal{S}, \mathcal{F} \cup U \cup \text{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ}) \rangle$ to an idle CE
5     if $\langle \text{result}, \mathcal{C} \rangle$ is received from DE:
6        if result is in $\{\text{SAT}, \text{UNSAT}\}$:
7           return result
8        else
9           update $\mathit{MaxDBSize}$
10          let $U = \mathit{UnitProp}(\mathcal{F} \cup U \cup \mathit{ClauseDB} \cup \mathcal{C})$
11          let $\mathit{ClauseDB} = \text{Merge}(U, \mathit{ClauseDB}, \mathcal{C}, \mathit{MaxDBSize})$
```

Figure 2. A general framework for CL-SDSAT

of preceding jobs. As the SAT solver in the jobs is randomized, the clauses returned by jobs subsequent to the same clause database usually differ. In the following we explain the proposed approach in more detail; the next sections then study different aspects of the approach using controlled experiments.

The framework for the approach is presented in pseudo-code in Fig. 2. The learned clauses are collected to an initially empty database of clauses, denoted by $\mathit{ClauseDB}$. The database is allowed to vary in size[1.] and its current maximum size is imposed by the variable $\mathit{MaxDBSize}$. From this database, a subset is provided to each job together with the original SAT instance $\mathcal{F}$ and the randomized SAT solver $\mathcal{S}$. The unit clauses U are stored separately, and are used for clause database simplification as explained below.

The main loop of the framework consists of two concurrent tasks: submitting subsequent jobs to idle CEs in the distributed environment and receiving the results of the finished jobs. The submitting of subsequent jobs is described on lines 2–4. If there are idle CEs in the environment (line 2), then a job $\langle \mathcal{S}, \mathcal{F} \cup U \cup \text{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ}) \rangle$ is submitted to one (line 4). The function Choose selects heuristically a subset of the clauses in the current database $\mathit{ClauseDB}$ so that the size of the subset is at most SubmSZ. Designing an effective heuristic for selecting learned clauses from the clause database is one of the key issues when instantiating the framework in Fig. 2. This problem is studied extensively in Sect. 4. The size of the selected 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 clauses can slow down the inner loop of the SAT solver. For the sake of extending the range of problems solvable with the approach, the size limit SubmSZ

1. 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.

may have to be increased during the search (line 3); this issue is discussed in the paragraph "On Completeness" below.

The results received from the DE are handled on lines 5–11 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 $\mathcal{F} \cup U \cup \text{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})$ submitted to a CE is satisfiable if and only if the original instance $\mathcal{F}$ is satisfiable because all the clauses in $U \cup \text{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})$ are (simplified) learned clauses and, thus, logical consequences of $\mathcal{F}$.

- If a job is unsuccessful, the clause database *ClauseDB* is updated with the set $\mathcal{C}$ of learned clauses returned from the job (lines 10 and 11). For the sake of being able to solve more difficult problems (see the "On Completeness" paragraph below), it may become necessary to increase the maximum size of the clause database during the search (line 9).

Maintaining the set U of unit clauses, detecting new ones with unit propagation, and using them to simplify the clause database is important for efficient implementation of the CL-SDSAT algorithm. Initially, the set U of unit clauses is obtained from the original instance $\mathcal{F}$. The set is monotonically increased on line 10 by applying unit propagation on the new and old learned clauses and the SAT instance. Then, on line 11, the set is used to simplify the clause database when new learned clauses are merged to it. This is done in the function Merge which takes the set of unit clauses U, the clause database *ClauseDB*, the new learned clauses $\mathcal{C}$, and the maximum clause database size *MaxDBSize* as input. It computes the simplified set of clauses, $S = \mathit{Simplify}(\mathit{ClauseDB} \cup \mathcal{C}, U)$, and returns a heuristically selected subset of S so that the size of the subset is at most *MaxDBSize*. As a consequence of restricting the maximum clause database size, some of the learned clauses can be discarded when computing the new clause database with the Merge function. This affects the range of problems solvable by the algorithm as discussed below and in Sect. 5.

When instantiating the framework into a concrete implementation, of special interest are the heuristics used in the two functions Choose and Merge. The next section analyzes key aspects of these functions.

On Completeness. As the resource limits for the jobs are fixed, the framework is obviously not complete (that is, there are SAT instances for which the framework does not terminate). For example, a SAT instance can be so large that it does not even fit in the available memory. In practice, this kind of incompleteness seems not to be of major concern. More practically relevant is however that the size limits for the clause database (*MaxDBSize*) and submitted learned clause sets (*SubmSZ*) also restrict the range of problems solvable with the framework. To extend this range, the parameters *MaxDBSize* and *SubmSZ* can be extended 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. Developing algorithms for deciding when and how much the size limits should be increased is left for future work.

Relation to SDSAT. Observe that when the clause database is empty, the CL-SDSAT algorithm performs exactly as the SDSAT algorithm. In practice, this happens at the beginning when all the, say N, jobs submitted so far are still running and have thus not yet returned any learned clauses. Therefore, when run in the same distributed environment with same resource limits, the CL-SDSAT algorithm can solve the instance at least as fast as SDSAT using N resources. In this sense CL-SDSAT subsumes SDSAT. The advantage of CL-SDSAT when compared to SDSAT is that, due to exploiting learned clauses, it can also solve problems that cannot be solved with SDSAT.

4. Analyzing the Key Aspects of CL-SDSAT

This section studies empirically key aspects of parallel learning strategies in the CL-SDSAT framework in Fig. 2. The study is divided into three parts. Part A experiments on four heuristics for selecting learned clauses to a subsequent job. This is done in a controlled setting involving one *round of learning* where a number of independent jobs are run with a randomized SAT solver on the same SAT instance and the resulting learned clauses are used to construct a *derived instance.* The four heuristics are compared by studying the run time distribution of a SAT solver on an instance and the corresponding derived instances where the learned clauses selected by the heuristic have been included. Part B studies how the run time distribution of the derived instance behaves as the number of computing elements is increased and, hence, the number of independent jobs in the round grows. Part C studies the cumulative effect of learned clauses by increasing the number of rounds.

Experimental Setting. All the experiments of this section were conducted in a controlled environment without background load, using Intel Xeon 5130 2GHz CPUs with 16GB of memory. The SAT solver used in the experiments is a modified version of MiniSat v.1.14 which accepts as input a seed for its internal random number generator to introduce randomness to the learned clauses and run time. It is also able to terminate the search when a given run time limit has been reached and to output the learned clauses held at that time. The value of *SubmSZ*, limiting the size of the learned clause set included to each problem, was kept relatively small (100,000 literals) throughout the experiments.

Selection of Benchmarks. The benchmark instances were selected from the benchmarks of the SAT 2007 Solver Competition so that it is possible to reliably evaluate the efficiency of the CL-SDSAT framework. In order to obtain reliable estimates on the performance of the framework, test runs need to be repeated and, hence, the chosen instances cannot be excessively difficult. To study the behavior of the CL-SDSAT framework, the instances should allow a SAT solver to produce a sufficient amount of learned clauses. In particular, instances which have short run times are likely to be solved fast by CL-SDSAT without any learning. This is because CL-SDSAT effectively subsumes SDSAT at the beginning of its execution when the clause database is empty and this leads to small expected run time for such instances [17].

These considerations led to the following three phase process of selecting the benchmark instances. In the first phase, an initial set of problems was selected such that it consisted of the publicly available instances which were solved by MiniSat in the 2007 competition, but required at least 2000 seconds for solving. This set consists of 53 problems. Second, each of these problems were solved 100 times using MiniSat v1.14 using different seeds, and only those for which the minimum run time was more than 1000 seconds were qualified. After this screening only one satisfiable instance remained, since all other satisfying instances had short run times. This set consists of 28 instances, and already constructing this set required more than half a CPU year. From the resulting set a representative subset consisting of 17 instances was formed so that a single random representative was selected from each instance family[2]. Table 1 reports the names together with the short labels used later in the experiments. The satisfiable instance is labeled `cube`.

Setting up Rounds of Learning. In the experiments the aim is to study the effect of heuristics for selecting learned clauses from the clause database after rounds of learning. In order to be able to use all selected benchmark instances, the run time distributions of the instances have to be taken into account when setting up rounds of learning and, in particular, the run time allowed for jobs in the round. Too long run times easily cause the instance to be solved during the round of learning. As the run time distributions of the instances differ considerably, in this controlled experiment a round of learning for an instance was implemented as follows. An approximation of the minimum run time of each instance is determined by solving it hundred times. The clause database is then constructed by running jobs for one fourth of this minimum run time and collecting the learned clauses. The clause database is constructed in this way in order to ensure that the instance is not solved while constructing the clause database.

A. Heuristics for the Function Choose. The criterion for selecting clauses in line 4 of Fig. 2 is central to the CL-SDSAT framework. In several related works, such as [20, 24], this criterion is the length of the clauses. Following this convention the actual implementation uses a criterion which prefers the short learned clauses. This criterion is called the $\text{Choose}_{\text{len}}$ heuristic. The approach is well justified in the CL-SDSAT framework: length based heuristic is efficiently implementable, and in this form guarantees the progress of the search because new clauses are included to the database as long as the database size limitation is not exceeded.

It is possible to vision other types of heuristics as well, which could be based on, for example, the number of occurrences of certain clauses in all learned clauses obtained from the unsuccessful jobs. Such heuristics are less straightforward to implement. For example, the heuristic $\text{Choose}_{\text{freq}}$, preferring the most commonly learned clauses in the preceding jobs, would require centralizing all learned clauses to a single place. This approach does not scale well, as it requires an excessive amount of memory when the number of clauses increases. To obtain an approximation of its efficiency, we experiment with the heuristic $\text{Choose}_{\text{freq}}$ on one round of learning; in this case the learned clauses still fit in a few gigabytes of storage space and can thus be fully analyzed.

The heuristic $\text{Choose}_{\text{len}}$ is also contrasted to two other heuristics: Choose_{123} which only considers clauses of length at most three and $\text{Choose}_{\text{rand}}$ which selects random clauses from the clause database. The former only selects a subset of the clauses selected by $\text{Choose}_{\text{len}}$, and helps to study the effect of longer clauses when contrasted to $\text{Choose}_{\text{len}}$. Since Choose_{123} does not consider clauses longer than three, it cannot guarantee progress in instances where

2. The instance families were identified based on the names of the instances, and each instance in a family had a similar run time distribution.

Table 1. Benchmarks instances from SAT 2007 competition and their short labels

Name	Label
`AProVE07-09`	`AProVE`
`contest03-SGI_30_50_30_20_3-dir.sat05-440.reshuffled-07`	`contest`
`cube-11-h14-sat`	`cube`
`dated-10-11-u`	`dated`
`emptyroom-4-h21-unsat`	`emptyroom`
`eq.atree.braun.11.unsat`	`atree`
`hwb-n28-02-S818962541.sat05-492.reshuffled-07`	`hwb`
`linvrinv5.sat05-564.reshuffled-07`	`linvrinv`
`manol-pipe-f9b`	`manol`
`mod2c-3cage-unsat-10-2.sat05-2567.reshuffled-07`	`mod2c`
`pmg-12-UNSAT.sat05-3940.reshuffled-07`	`pmg`
`pyhala-braun-unsat-40-4-02.sat05-459.reshuffled-07`	`pyhala`
`QG7-gensys-ukn003.sat05-3346.reshuffled-07`	`GQ7`
`s101-100`	`s101-100`
`sortnet-6-ipc5-h11-unsat`	`sortnet`
`total-10-13-u`	`total`
`unsat-set-b-fclqcolor-10-07-09.sat05-1282.reshuffled-07`	`fclqcolor`

short learned clauses are rare. Therefore it is not generic enough for the purposes we are considering. The heuristic $\text{Choose}_{\text{rand}}$ is useful as a reference, as all learned clauses are already carefully selected by the solver and it is not a priori clear if heuristics are required to obtain better results.

We repeat the definitions of the four heuristics below.

- $\text{Choose}_{\text{len}}$ prefers short learned clauses. Short clauses are potentially effective in pruning the search space.

- Choose_{123} returns only clauses of at most three literals. Such clauses are even more effective in pruning the search space but might be rare in practice in some cases.

- $\text{Choose}_{\text{freq}}$ prefers the most common learned clauses. Such clauses are intuitively good since they are encountered in many jobs. From equally frequent clauses, the shorter ones are preferred. As discussed above implementing this heuristic in the full CL-SDSAT framework seems prohibitively expensive.

- $\text{Choose}_{\text{rand}}$ 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.

The clause database is simplified before the heuristics are used for selecting the set of clauses, as discussed in Sect. 3. This has several subtle consequences: the length of a clause in the clause database may decrease as a result of simplification and if two longer clauses reduce to the same short clause, the frequency of the short clause increases. Also none of the heuristics need to return unit clauses, as unit clauses are included into the submitted instance separately.

The aim of the first experiment is to study the effect of including learned clauses to benchmark instances when performing one round of learning. A fixed number of independent jobs produce a large candidate set of learned clauses for each benchmark. The resulting sets of learned clauses are then simplified using the SAT instance and the possible unit clauses found either in the SAT instance or in the learned clauses, resulting in the clause database. A derived instance is constructed by employing the respective Choose heuristic to select a subset of the clause database (line 4 of the algorithm in Fig. 2), with *SubmSZ* set to 100,000 literals and number of computing elements $r = 8$. We note that the size limitation of *ClauseDB* is ignored by effectively setting *MaxDBSize* to infinity in these experiments.

Table 2 gives an overview of the results by comparing the expected run times over fifty runs of the derived instances when using the heuristics. For comparison, the table reports expected run times for the original instances without additional learned clauses (Base). The table also reports the expected number of decisions made by the SAT solver, which measures the expected size of the search space for the instance, below the run time. The lowest of these numbers on each row is printed in bold face.

Based on these results, the expected number of decisions is lowest when using the length-based heuristic (Choose$_{\text{len}}$), and the expected run time is lowest when only clauses of length two and three are considered (Choose$_{123}$). In Choose$_{\text{len}}$, the run time of the instance can be high, being often higher than when no clauses are included. The experiment allows us to conclude that while length is an efficient criterion for selecting clauses, the inclusion of longer clauses results in more overhead than what is gained by reduction of the size of the search space. On the other hand, using frequent clauses (Choose$_{\text{freq}}$) results also in good speed-up in expected run times. These clauses are at least as long as the clauses preferred by Choose$_{\text{len}}$, suggesting that carefully selected long clauses can be used to speed up the solving. The results from Choose$_{\text{rand}}$ show also reduction in both expected number of decisions and run time. The clauses returned by MiniSat seem to be good in reducing the number of decisions even when no particular heuristic is used in selecting the clauses. The comparison to other heuristics reveals though that an appropriate heuristic can significantly lower the expected run time of a derived instance.

The results on this benchmark set are surprisingly consistent: in 13 cases, Choose$_{\text{len}}$ results in lowest expected number of decisions, while in ten cases, Choose$_{123}$ results in lowest expected run time. Using eight sources of learned clauses, in every case at least one heuristic succeeds in reducing the expected time required to solve the instance compared to Base.

Also evident from the results is that Choose$_{\text{freq}}$ performs well when compared to Choose$_{\text{len}}$. The relatively good performance might be an indication that an approach based on clause frequencies has potential. However, further studies are required to determine if this actually is the case. If so, more work is needed to obtain an efficiently implementable realistic approximation of the Choose$_{\text{freq}}$ heuristic. In particular, the frequency of the clauses interacts interestingly with simplifications of the clause database. In some cases the same clause in its simplified form occurred more than fifty times, while the number of preceding jobs is only eight.

Table 2. Expected run times for a selection of benchmarks from the SAT 2007 competition

Name	Base	$\text{Choose}_{\text{len}}$	$\text{Choose}_{\text{freq}}$	Choose_{123}	$\text{Choose}_{\text{rand}}$
`AProVE`	4,016	**1,994**	2,616	2,264	3,393
	8,461,866	**4,388,463**	4,716,035	5,532,927	7,451,391
`atree`	3,096	2,967	2,152	**1,439**	2,481
	22,311,255	**7,831,761**	13,263,105	9,034,391	14,404,941
`contest`	1,432	**70**	485	211	343
	1,240,001	**165,721**	541,943	357,978	467,458
`cube`	4,832	**4,483**	4,939	4,888	5,294
	1,273,485	**967,851**	1,096,322	1,238,110	1,313,385
`dated`	9,889	2,037	**1,977**	2,187	5,240
	1,639,566	1,058,664	**998,103**	1,146,487	2,246,003
`emptyroom`	5,205	**1,498**	1,631	1,704	1,954
	1,885,355	**688,156**	813,027	853,642	1,052,777
`fclqcolor`	2,027	**1,153**	1,388	1,196	1,864
	41,172,989	**13,696,945**	29,946,390	25,945,033	26,103,961
`hwb`	4,654	14,128	5,001	**4,454**	10,211
	125,472,477	**68,950,042**	123,220,119	97,041,128	82,550,196
`linvrinv`	2,828	7,837	2,620	**2,518**	4,030
	40,917,769	**25,824,068**	37,369,017	36,283,860	32,008,217
`manol`	10,620	13,336	9,196	**7,120**	10,814
	4,954,967	5,308,314	4,328,594	**3,401,500**	5,101,791
`mod2c`	3,020	3,827	2,659	**2,496**	4,392
	271,766,780	**62,714,188**	221,568,484	195,269,018	87,430,751
`pmg`	4,268	9,372	4,189	**2,955**	7,876
	84,245,813	**40,690,352**	69,882,275	48,750,743	56,061,825
`pyhala`	2,641	887	1,086	**782**	1,348
	2,775,304	**1,001,999**	1,855,329	1,436,653	2,245,269
`QG7`	1,594	760	1,196	**513**	1,506
	6,799,632	**2,081,121**	5,256,338	2,737,811	5,436,088
`s101-100`	2,528	5,047	2,502	**2,428**	4,907
	170,749,796	47,196,913	167,440,762	166,645,578	**46,054,481**
`sortnet`	4,886	1,521	2,893	**1,507**	4,694
	2,743,833	**900,265**	1,842,295	980,166	2,607,584
`total`	3,279	1,296	**1,109**	1,695	1,722
	1,178,947	690,406	**682,302**	998,194	997,008
Sum	73,383	72,213	47,639	**40,357**	72,069
	789,589,835	**284,378,607**	684,820,440	597,653,219	373,533,126

Table 3. Minimum, expected and maximum run times for $\text{Choose}_{\text{len}}$ and different values of preceding jobs r

Label		0	16	32	48	64	80	96
`contest`	Min	1,080	6.58	0.70	0.53	0.41	0.17	0.16
	Exp	1,430	8.39	0.88	0.53	0.41	0.17	0.16
	Max	1,990	10.9	1.36	0.53	0.41	0.17	0.16
`fclqcolor`	Min	1,010	384	138	84.1	54.9	24.9	22.6
	Exp	2,030	595	281	161	102	55.9	51.1
	Max	3,650	1,450	529	297	187	120	162
`hwb`	Min	3,600	7,040	5,980	5,470	3,520	2,670	2,320
	Exp	4,650	8,770	7,880	6,710	4,680	3,390	3,000
	Max	6,190	11,300	9,990	9,050	6,590	4,780	4,100
`manol`	Min	1,510	1,390	924	516	589	584	576
	Exp	10,600	4,570	2,320	1,580	1,540	1,650	1,820
	Max	65,400	14,700	5,250	4,540	3,480	5,700	4,880

B: The Effect of Increasing the Number of CEs. In many realistic scenarios, the number of computing elements can be much higher than eight that was used in the experiments reported above and in Table 2. This corresponds to increasing the value of the parameter r, and should intuitively result in a decrease of the expected run time for the derived instance (that is, the instance including the learned clauses). We provide some experimental evidence supporting this intuition for the heuristic $\text{Choose}_{\text{len}}$ by showing how the expected run time of a job subsequent to a clause database after a single round of learning behaves when r is increased.

Starting from an empty database, we perform one round of learning by submitting r subsequent jobs. We ensure that the jobs all are unsuccessful by limiting the run time of each job again to 25% of the previously measured minimum run time. The resulting clauses are merged to clause database, which we denote by $\mathit{ClauseDB}_r$. While performing these experiments, we use the sizes $\mathit{MaxDBSize} = 10,000,000$ and $\mathit{SubmSZ} = 100,000$. As discussed above, the experiments use the $\text{Choose}_{\text{len}}$ heuristic. The benchmark instances are selected using the results of Table 2 as a criterion.

We show the run time distributions for instances `contest` (Fig. 3) and `manol` (Fig. 4), where the former seems to scale well using all heuristics, and the latter does not clearly benefit from any heuristic and is especially bad for $\text{Choose}_{\text{len}}$. For comparison, the figures also give the distributions for the number of decisions. Table 3 provides statistics for these and two other instances.

In all of the experiments, the number of decisions and the run time decrease when the amount of preceding jobs increases sufficiently. The instance `contest` becomes easy to solve relatively soon. As the typical run time reaches values less than one second, it becomes difficult to see if the added clauses help in solving the problem further. In contrast, some of the instances show a slowdown in the decrease of the run time. The slowdown is well

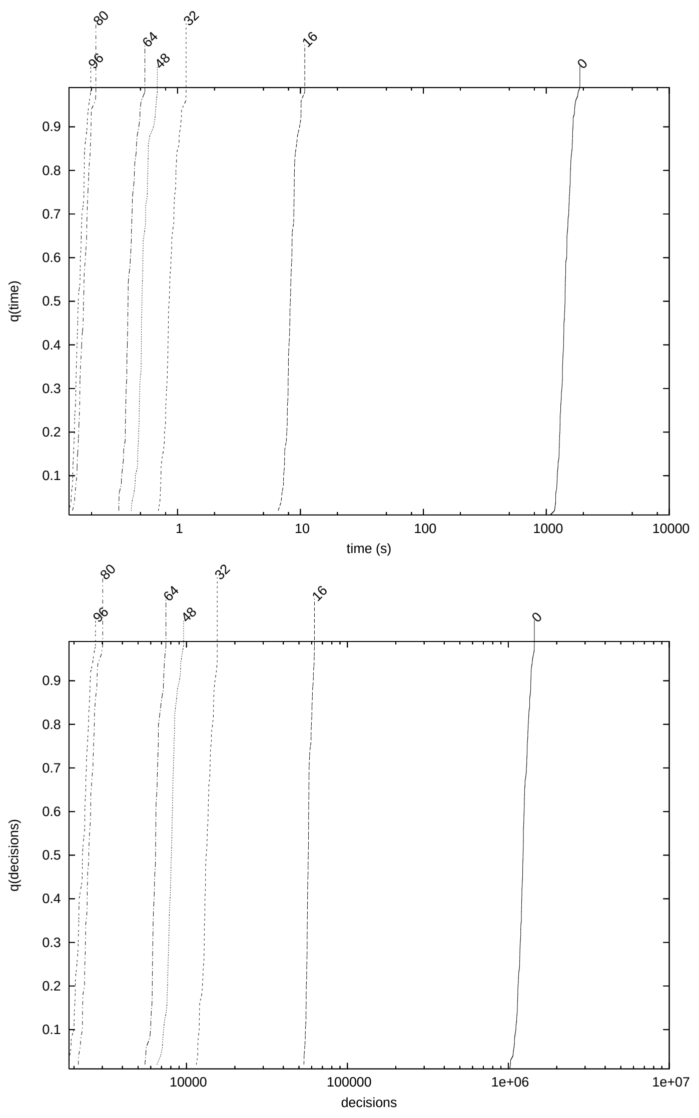

Figure 3. Run time and decision distribution for `contest` using $\text{Choose}_{\text{len}}$ and different values of preceding jobs r

illustrated by Fig. 4 for `manol`. The expected run time for the instance first decreases gradually from almost three hours to slightly over 25 minutes reaching the minimum when the number of preceding jobs is 64. Increasing the amount of preceding jobs does not help to decrease the expected run time, which indeed seems to slightly increase as the expected run time with 96 preceding jobs is almost five minutes higher compared to 64 preceding jobs.

The other two instances in Table 3 show a nearly consistent decrease in all statistics. Interestingly, the expected run time of `hwb` in Table 3 reaches that of the original instance (see Table 2) only when there are more than 64 preceding jobs.

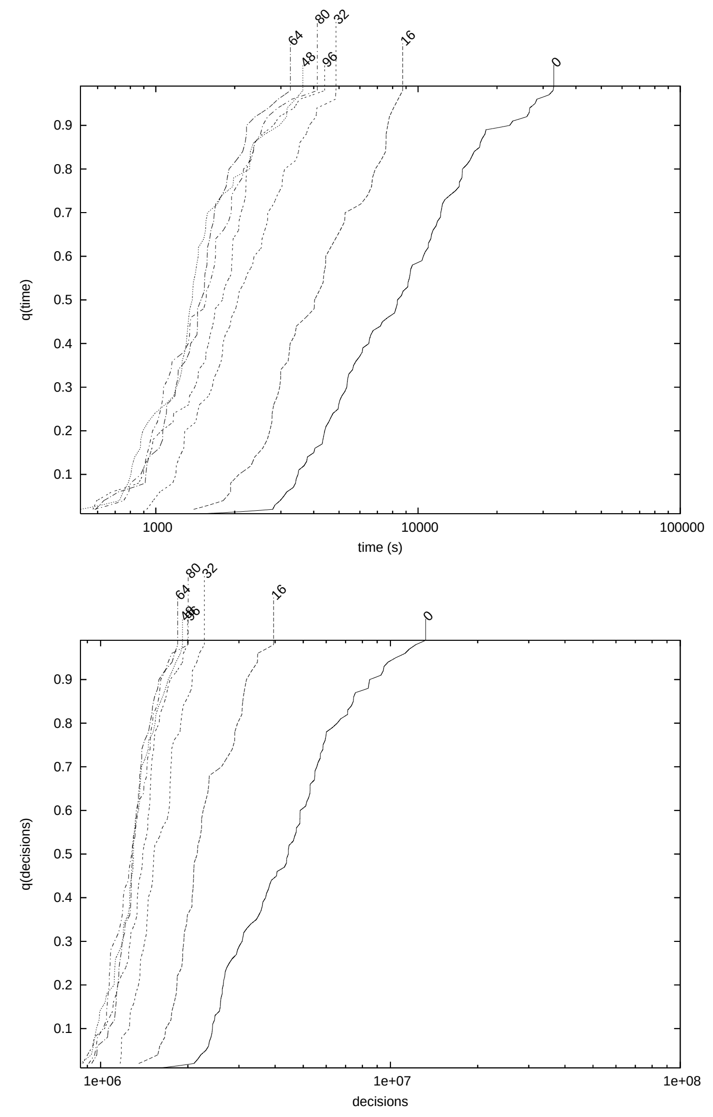

Figure 4. Run time and decision distribution for `manol` using $\text{Choose}_{\text{len}}$ and different values of preceding jobs r

We note briefly that the expected total time required to solve, for example, the derived instance `manol` when $r = 48$ is 1,580 seconds. Even when the time required to obtain the derived instance is taken into consideration $(0.25 \times 1,510 + 1,580)$, the problem can be solved five times faster than the original instance.

However, these results show that there are instances such that after a single round of learning in CL-SDSAT, even the minimum run time of the derived instance does not seem to become arbitrarily small no matter how much r is increased.

C: Cumulative Effect of Learned Clauses. As the previous experiment indicates, in many cases after a single round of learning the subsequent jobs do still exceed the time

Table 4. Minimum, expected (Exp) and maximum run times for different number of rounds i in jobs subsequent to $\mathit{ClauseDB}^i_{16}, 0 \leq i \leq 4$ and $\text{Choose}_{\text{len}}$

Label		0	1	2	3	4
`fclqcolor`	Min	1,010	384	5.15		
	Exp	2,030	595	9.10		
	Max	3,650	1,450	30.4		
`hwb`	Min	3,600	7,040	4,520	1,060	2.77
	Exp	4,650	8,770	6,000	1,350	3.16
	Max	6,190	11,300	7,880	1,900	3.87
`manol`	Min	1,510	1,390	618	76.9	
	Exp	10,600	4,570	1,350	157	
	Max	65,400	14,700	3,350	313	
`total`	Min	1,190	892	370	6.19	
	Exp	3,280	1,480	568	7.01	
	Max	8,530	2,020	830	8.41	

limitation imposed by the distributed environment. In the CL-SDSAT framework, the learned clauses are cumulated to overcome the problem. This means that not all jobs are subsequent to the same clause database, but the set of preceding jobs is allowed to grow arbitrarily as jobs are submitted. We study the effect of this by continuing the previous experiment as follows. As previously, we assume that each clause database includes the unit literals. Let $\mathit{ClauseDB}^0_r$ denote the empty clause database, and let $\mathit{ClauseDB}^{i+1}_r$ be the clause database after one round of learning from r SAT instances obtained from $\mathit{ClauseDB}^i_r$. The experiment studies the run time distribution of the job subsequent to $\mathit{ClauseDB}^i_r$ with fixed $r = 16$ as the number of rounds i increases. We impose a resource limit on the subsequent jobs so that their run time is at most 25% of the experimental minimum run time of the original instance. The process terminates when the instance is solved within this time. Other parameters are as in the previous experiment.

We report the results for the same problems as in the previous case, with the exception that `contest` is replaced by `total` since `contest` is easily solved in a job subsequent to $\mathit{ClauseDB}^1_{16}$. The results are shown in Fig. 5 and Table 4. We may compare the results illustrated for `manol` in Fig. 5 against those in Fig. 4 bearing in mind that each round corresponds to 16 jobs. When clauses are cumulated, the run times decrease at a consistent pace, as opposed to the slowdown illustrated in Fig. 4. Similar results are reported for the other instances in Table 4. For example, CL-SDSAT is able to overcome the increase in the run time of the derived instances of `hwb` relatively soon when the number of rounds increases.

The results support the hypothesis that hard instances with practical relevance can be made solvable within typical resource limits for individual jobs in realistic distributed environments considered in this work.

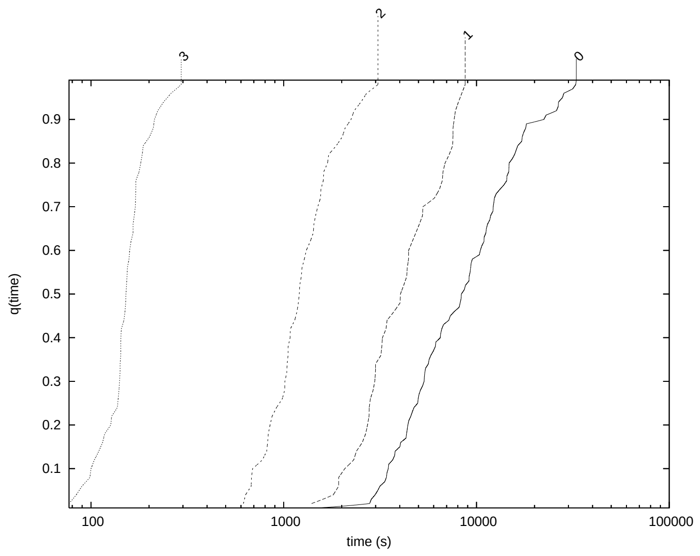

Figure 5. Run time distributions for jobs subsequent to $ClauseDB^0_{16}$, $ClauseDB^1_{16}$, $ClauseDB^2_{16}$ and $ClauseDB^3_{16}$ for `manol` and $\text{Choose}_{\text{len}}$

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 (`http://www.nordugrid.org/`), 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 [14], and each job has a time limit of one hour and a memory limit of one gigabyte. The implementation uses the heuristic $\text{Choose}_{\text{len}}$ preferring the shortest clauses for parallel learning; other heuristics discussed in Sect. 4 were not implemented in the prototype. In the experiments, the maximum size of the clause database is fixed to 1,000,000 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 crafted category). Table 5 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 three days allowing the use of 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 runs were performed using an Intel Xeon 5130 2GHz CPU. 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 therefore are 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 [17] with no need for the learning-enhanced techniques of CL-SDSAT. 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 CL-SDSAT framework also works for very hard problems and causes the run time distribution to "shift leftwards" (recalling Fig. 5 and the results from Table 4) as more learned clauses are cumulated. 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.

Some of the instances were not solved in the Grid within three days. The two instances the implementation was not able to solve suffered from the slow rate of change in the clause database. This in part was a result of the eventual high number of binary clauses in the clause database together with the property of the heuristic $\text{Choose}_{\text{len}}$ that it cannot differentiate clauses of the same length. When the clause database does not change, the subsequent jobs are similar to each other and the progress of the search is slow. This is of course a consequence of not increasing the size of the clause database as the search progresses. Implementing this feature is an interesting direction of future work. We also note that it is often possible to simplify binary clauses with sophisticated techniques [3]. However, experiments are required to determine whether such approaches are useful in this setting. In addition, it would be interesting to study how other formula simplification techniques such as [7, 13] could be applied to simplify the clause database.

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 compared different heuristics for selecting the learned clauses that are dynamically stored during the process, and demonstrated that the approach enables a form of clause learning that is not directly available in the underlying sequential clause learning SAT solver. 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.

Future Work. There are a number of interesting extensions of the current work. Preprocessing methods can applied in several stages of the framework: to simplify the original instance using, for example, methods discussed in [7], for compressing the learned clause database as suggested in Sect. 5, and in each individual job. In the experiments we have

Table 5. Wall clock times for some difficult instances from SAT 2007 competition solved in Grid and with standard MiniSat v1.14. Memory outs are denoted by '*', time outs by '—'

Solved by some solvers in SAT 2007 but not by MiniSat			
Name	Type	Grid (s)	MiniSat (s)
`ezfact64_5.sat05-452.reshuffled-07`	SAT	4,826	65,739
`vmpc_33`	SAT	669	184,928
`safe-50-h50-sat`	SAT	12,070	*
`connm-ue-csp-sat-n800-d-0.02-s1542454144-.sat05-533.reshuffled-07`	SAT	5,974	119,724
Not solved by any solver in SAT 2007			
Name	Type	Grid (s)	MiniSat (s)
`AProVE07-01`	UNSAT	13,780	39,627
`AProVE07-25`	UNSAT	94,974	306,634
`QG7a-gensys-ukn002.sat05--3842.reshuffled-07`	UNSAT	8,260	127,801
`vmpc_34`	SAT	3,925	90,827
`safe-50-h49-unsat`		—	*
`partial-10-13-s.cnf`	SAT	7,960	*
`sortnet-8-ipc5-h19-sat`		—	*
`dated-10-17-u`	UNSAT	11,747	105,821
`eq.atree.braun.12.unsat`	UNSAT	9,072	59,229

used MiniSat as the randomized clause learning solver. An interesting topic is to study whether similar results can be obtained if another clause learning solver is used. A natural next step would then be to consider algorithm portfolios [11] where a set of solvers is employed and solvers are varied in different jobs. There seems to be room for improving also the heuristics for selecting learned clauses. The results on the Choose$_{\mathrm{freq}}$ heuristic in Sect. 4 suggest that frequency based approaches could have potential. Another direction are activity based heuristics, for example, using learned clause activity weights as in MiniSat [8]. Although the goal of the work is to devise techniques for solving extremely hard SAT instances using computational grids, studying speed-up obtained by CL-SDSAT is an interesting topic that facilitates the comparison of CL-SDSAT to sequential solvers and to other distributed solving methods.

Acknowledgments

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, Emil Aaltosen Säätiö, and Technology Industries of Finland Centennial Foundation is gratefully acknowledged.

References

[1] W. Blochinger, W. Westje, W. Küchlin, and S. Wedeniwski. ZetaSAT – Boolean satisfiability solving on desktop grids. In *CCGrid 2005*, pages 1079–1086. IEEE, 2005.

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

[3] R. I. Brafman. A simplifier for propositional formulas with many binary formulas. *IEEE Transactions on Systems, Man, And Cybernetics—Part B: Cybernetics*, **34**(1):52–59, 2004.

[4] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based distributed SAT solver for the Grid. In *SC 2003*. IEEE, 2003.

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

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

[7] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT 2005*, **3569** of *LNCS*, pages 61–75. Springer, 2005.

[8] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2003*, **2919** of *LNCS*, pages 502–518. Springer, 2003.

[9] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science*, **128**(3):75–90, 2005.

[10] S. Forman and A. Segre. 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.

[11] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, **126**(1-2):43–62, 2001.

[12] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, **24**(1/2):67–100, 2000.

[13] H. Han and F. Somenzi. Alembic: An efficient algorithm for CNF preprocessing. In *DAC 2007*, pages 582–587. IEEE, 2007.

[14] A. E. J. Hyvärinen. GridJM. A computer program for managing grid jobs. http://www.tcs.hut.fi/~aehyvari/gridjm/.

[15] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. A distribution method for solving SAT in grids. In *SAT 2006*, **4121** of *LNCS*, pages 430–435. Springer, 2006.

[16] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Incorporating learning in grid-based randomized SAT solving. In *AIMSA 2008*, **5253** of *LNAI*, pages 247–261. Springer, 2008.

[17] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Strategies for solving SAT in grids by randomized search. In *AISC 2008*, **5144** of *LNAI*, pages 125–140. Springer, 2008.

[18] K. Inoue, T. Soh, S. Ueda, Y. Sasaura, M. Banbara, and N. Tamura. A competitive and cooperative approach to propositional satisfiability. *Discrete Applied Mathematics*, **154**(16):2291–2306, 2006.

[19] B. Jurkowiak, C. Li, and G. Utard. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning*, **34**(1):73–101, 2005.

[20] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, **48**(5):506–521, 1999.

[21] D. G. Mitchell. A SAT solver primer. *Bulletin of the EATCS*, **85**:112–132, 2005.

[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC 2001*, pages 530–535. ACM, 2001.

[23] T. Schubert, M. Lewis, and B. Becker. PaMira — a parallel SAT solver with knowledge sharing. In *MVT'05*, pages 29–36. IEEE, 2005.

[24] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT — Parallel SAT-checking with lemma exchange: Implementation and applications. In *SAT 2001*, **9** of *Electronic Notes in Discrete Mathematics*, pages 12–13. Elsevier, 2001.

[25] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, **21**(4):543–560, 1996.

[26] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD 2001*, pages 279–285. ACM, 2001.