Strategies for Solving SAT in Grids by Randomized Search

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. Grid computing offers a promising approach to solving challenging computational problems in an environment consisting of a large number of easily accessible resources. In this paper we develop strategies for solving collections of hard instances of the propositional satisfiability problem (SAT) with a randomized SAT solver run in a Grid. We study alternative strategies by using a simulation framework which is composed of (i) a grid model capturing the communication and management delays, and (ii) run-time distributions of a randomized solver, obtained by running a state-of-the-art SAT solver on a collection of hard instances. The results are experimentally validated in a production level Grid. When solving a single hard SAT instance, the results show that in practice only a relatively small amount of parallelism can be efficiently used; the speedup obtained by increasing parallelism thereafter is negligible. This observation leads to a novel strategy of using grid to solve collections of hard instances. Instead of solving instances one-by-one, the strategy aims at decreasing the overall solution time by applying an alternating distribution schedule.

1 Introduction

This paper considers techniques for solving challenging instances of the *propositional satisfiability* (SAT) problem with the aid of computational *Grids*. Such techniques are of particular interest firstly due to the increasing use of SAT based technologies in computer aided verification and other application areas, and secondly since Grids are nowadays offering large quantities of affordable computing power. The first phenomenon is a consequence of recent developments in SAT solvers which have dramatically improved the computational power of the solvers, whereas the second seems to be a major trend in high-performance computing.

Our goal in this paper is to develop techniques for exploiting the parallel computing resources provided by a Grid in a way that allows us to use state-of-the-art SAT solvers with no or only minor modifications. To do this, we use the *Simple Distributed SAT* (SDSAT) framework, whose basic version consists of simply running N *randomized SAT solvers* in parallel until one of them finds the solution. We consider extensions of the basic version obtained by incorporating different *restart strategies* and study their effects in a specifically built simulation environment. The simulation environment comprises of (i) a Grid model taking into account the inherent communication and management delays, and (ii) run time distributions of a state-of-the-art randomized SAT solver when applied on several hard SAT instances. We also validate some of the results and parameters of our Grid model by using a production level Grid called NorduGrid (see `http://www.nordugrid.org/`),

The key idea we exploit is that a complete SAT solver can be turned into a *randomized search procedure* (RSP) in a natural way by slightly modifying the heuristic function used in the solver. For example, MiniSAT [1] 1.14 makes by default 2% of its heuristic choices pseudo-randomly; thus a natural modification to turn MiniSAT into a RSP is to seed its pseudo-random number generator differently for each run. Such a randomized search procedure, when provided with an input x, is guaranteed to give a correct result $\mathrm{RSP}(x)$ when the computation of the procedure finishes. However, due to the randomization, the time required for computing $\mathrm{RSP}(x)$ is not known in advance but is described by a random variable $T_{\mathrm{RSP}(x)}$. The random variable $T_{\mathrm{RSP}(x)}$, and thus the run time of $\mathrm{RSP}(x)$, is completely characterized by its cumulative *run time distribution* function, $q_{\mathrm{RSP}(x)}(t)$, giving the probability that the computation will terminate before or at time t. This randomization of a SAT solver may sound counter-intuitive as one usually tries to remove all non-determinism in order to make runs reproducible to ease benchmarking and debugging. However, in the SDSAT framework as well as when employing restart strategies to a RSP (discussed below), the goal is to exploit the *short runs* (if any) in the distribution to decrease the *expected run time* of the overall system.

The expected run time of a randomized search procedure can often be substantially reduced by periodically restarting the procedure [2]. For example, assume that $T_{\mathrm{RSP}(x)} = 1\mathrm{s}$ with probability 0.3 and $T_{\mathrm{RSP}(x)} = 10\mathrm{s}$ with probability 0.7. Then the expected run time $\mathbb{E}(T_{\mathrm{RSP}(x)})$ is $0.3 \cdot 1\mathrm{s} + 0.7 \cdot 10\mathrm{s} = 7.3\mathrm{s}$. If the RSP is modified so that it restarts itself immediately after time $t = 1\mathrm{s}$, the expected run time becomes $\sum_{i=1}^{\infty} 0.7^{i-1} \cdot 0.3^{i} \cdot i\mathrm{s} \approx 3.3\mathrm{s}$. Such a modification, where the procedure is forced to start from the beginning after running t_1 seconds, then after t_2 seconds and so forth, is called a *restart strategy* $S = (t_1, t_2, \ldots)$ and the time t_i the i:th *restart limit*. When a restart strategy is employed to an RSP, the result is a randomized *algorithm* that also has a run time distribution and an expected run time. The restart strategy employed in the previous example is a special case of a *fixed restart strategy* $S^t = (t, t, \ldots)$ and the algorithm corresponding to the fixed restart strategy S^t employed on RSP is denoted by $\textsc{Fixed}_{t,\mathrm{RSP}}$ (or simply $\textsc{Fixed}_t$ when RSP is implicitly known). Fixed restart strategies are important in our analysis, since if $q_{\mathrm{RSP}(x)}(t)$ is known, then t can be chosen so that the expected run time of $\textsc{Fixed}_t(x)$ is the minimal among all the algorithms obtainable from $\mathrm{RSP}(x)$ by employing *any* restart strategy [3]. However, in practice $q_{\mathrm{RSP}(x)}(t)$ is not known: obtaining information about $q_{\mathrm{RSP}(x)}(t)$ in general requires solving $\mathrm{RSP}(x)$, which is the overall goal in many applications. To circumvent this problem, several *universal* restart strategies have been suggested [3,4]: they do not depend on the instance x and let the restart limits grow arbitrary large in order to preserve the completeness of the algorithm.

We first study the effect of applying several restart strategies on our benchmark set of hard SAT instances in the sequential setting. The results show that there are instances on which the optimal fixed restart strategy provides a substantial reduction in the expected run time. The two universal strategies considered can also reduce the expected run time on some instances but result in a bad performance on some others.

The reason is that the universal strategies can spend too much time in trying to find a short run; when an instance has none, all that time is wasted.

Based on results in the sequential case, we consider ways to parallelize restart strategies in the SDSAT framework and use our simulation model to benchmark them. The results give rise to two major observations. First, parallelism seems to be an effective "luck enhancer"; when randomized solvers are run in parallel, the probability that one of them finds a short run grows quite quickly. This seems to render elaborate restart strategies practically useless in the parallel setting as the simple approach with no restarts tends to provide quite good results consistently. The second observation is that only a relatively small amount of parallelism seems to be effectively exploitable; after a certain amount, adding more parallel solvers does not seem to give any significant performance gain. There seems to be two reasons for this: (i) the probability that a short run is found is already quite high with a smallish number of parallel solvers, and (ii) the delays in the Grid environment reduce the effect of restart strategies.

The above results suggest that when solving a *set of instances*, a good speedup is not obtained by solving them one-by-one in a Grid. Instead, the instances should be solved in parallel by reserving a smallish amount of computing resources for each instance. We validate this idea in Sect. 6 both with the simulation model and by using a production level Grid.

Related Work. Techniques for learning or adapting restart strategies to improve the aggregate performance on a given collection of instances are studied, e.g., in [5,6,7,8,9]. A closely related topic is the use of algorithm portfolios [10,11]. The idea is combined with clause learning in [12]. Parallel restart strategies are studied in [13], without considering the practical limitations of a Grid. Guiding path [14,15] is a technique for distributed SAT solving based on dynamic partitioning of the problem with new assumptions. Such methods combine also with clause learning [16]. The techniques in grid-like environments have been investigated, for example, in [17,18,19,20]. The guiding path method is further developed in [21]. A different algorithm is presented in [22].

In this paper we extend previous work in three crucial respects: (i) We take into account the limitations of practical Grid environments which involve strict resource bounds and significant latencies due to communication and job management. (ii) We require minimal changes to the SAT solvers, and the changes are almost totally independent of the underlying solver technology. (iii) We use realistic run time distributions of the randomized search procedure obtained experimentally by running a state-of-the-art SAT solver on a representative collection of SAT instances from the application domain.

2 Grid Environment

The paper develops techniques for using loosely coupled, widely distributed Grid environments for solving challenging SAT problems. From an abstract point of view a Grid environment can be seen as consisting of a collection of computing resources called *primitive computing elements* (PCEs). A PCE can execute a sequential program given its input, hence, in practice corresponding to a CPU. A user can submit a *job* (a sequential program together with its input) to the Grid which executes it on one of its PCEs and gives results back to the user.

Next we briefly described three key characteristics which play an important role when developing Grid applications and the algorithms in this paper: (i) jobs in Grids experience significant delays but (ii) the run time of a job typically affects the effect of delays and (iii) communication between jobs is very limited when compared to traditional multi-processor environments such as clusters.

(i) The entry point of a Grid environment is a set of queues accepting jobs. Each queue is associated with a set of *computing elements* (CEs) corresponding to a set of CPUs. A job starts executing when the queue system assigns the job to a CE. Several causes of delays can be identified. Firstly, the time required for the job to reach a CE after submission to the corresponding queue depends on the amount and types of previously submitted jobs still in the queue, and the remaining run times of the jobs currently executing in the CEs. Secondly, if the submission of a job involves transmitting a large amount of data, the amount of network bandwidth may greatly affect the delays [23]. Thirdly, the run time of a job in a CE depends on the load potentially placed by other jobs on the neighboring CPUs, as well as the types of the CPUs in the CE. Finally, it is possible that jobs disappear due to maintenance breaks or various random faults. Efficient job management in Grids is a non-trivial task and is typically handled by special tools. In those experiments of this paper that are run in NorduGrid, we use a fault-tolerant and efficient job management system called the Grid Job Manager (GridJM) [24].

(ii) Note that the different delays above seem to suggest that a job with limited run time could experience shorter delays. For instance, most queue systems support a mechanism called *reservation*, where a complicated task requesting a CE of several CPUs will force the queue system to start to reserve CPUs. In this case, no new jobs requesting a CE will be assigned from the queue, unless the run time of the job is short enough to finish before the time expected for the requested CE of several CPUs to become available. On the other hand, since the delays are experienced by each job, it would be preferable to submit sufficiently long running jobs so that the delays do not dominate the total run time. As a reasonable compromise, in the experiments in NorduGrid we use jobs where the run time is limited to one hour.

(iii) Since a Grid can be formed by several independent but collaborating organizations which decide to share the computing resources, it is common that two jobs submitted to the Grid are not guaranteed to be able to communicate with each other at all. For example, such limitations are typically posed by the networks of the organizations in NorduGrid used in the experiments and, therefore, in the algorithms developed in the paper we assume that jobs cannot communicate directly with each other.

3 Simulation Environment

Realistic Grid systems pose certain challenges for exact algorithm benchmarking, since both the delays and the run times vary, rendering the reproduction of results difficult. To overcome these challenges, we construct a simple Grid model based on the following components:

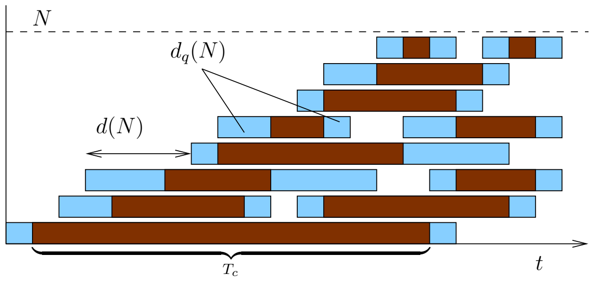


Fig. 1. A time line of an execution in Grid representing the number N of PCEs, queue delay $d_q(N)$, and the submit delay $d(N)$. In the example, the first job has executed the maximum allowed time T_c on a PCE.

(1) A unique central process M initiating new and monitoring old jobs, and a set of N PCEs receiving jobs from and reporting the results to M.
(2) An initiation delay describing the amount of time required to submit a job to the Grid. The delay $d(N)$ can be modeled as a random variable depending on the number of PCEs employed. The delay is executed by M and results in a bottleneck when initiating new computations.
(3) A queue delay is the sum of two components: the time spent queuing to the PCE, and the time spent receiving the results after the job has finished. The delay $d_q(N)$ can be modeled as a random variable depending on the number of PCEs employed. The delay is experienced by the job and does not form a bottleneck for submission.
(4) A maximum resource limit T_c describing the amount of time a PCE is allowed to execute before terminating a job and becoming ready to accept a new job.

We believe that this system provides a realistic model for distributed computing in Grids. (1) A central process managing jobs provides a natural synchronization mechanism. (2&3) Most such systems have a delay associated with the synchronization, and specifically shared distributed environments require certain communication in selecting the PCE to be employed. (4) Batch systems such as Grids usually limit the resources available to a single job, for example, to provide fairness in scheduling. The model does not directly consider the effect of various CPU models and the load on the CPUs on the run time. Such effects can be obtained by adjusting the queue delay and the resource limit accordingly.

We may study an application submitting jobs to the Grid through a central process M as a time line, illustrated in Fig. 1. The time advances to the right in the figure and the abstract PCEs can be seen as N bands placed on top of each other. The filled rectangles represent jobs, and the dark areas inside the jobs represents the CPU time, as opposed to the queuing delay. The time in the figure starts when the first job (the long rectangle at the bottom of the figure) is placed into a queue of a PCE. The second job is submitted immediately after this, and after the submission delay $d(N)$, reaches the queue. Meanwhile, the first job has reached the PCE, is executed in it, and finally the result is reported back to the central process after some queue delay.

When performing the actual simulations, we make the following simplifying assumptions on the model:

– submit delay $d(N) = d$ is constant for every PCE and does not depend on N, and
– queue delay $d_q(N) = d_q$ is constant for every PCE and does not depend on N.

If the effect of the number of PCEs is taken into account, the delays will increase since in practice the jobs will interfere with each other. This means that using the simplifying assumptions the resulting run time is underestimated and this error increases with the number of PCEs employed. Hence, the model with the simplifying assumptions gives overly optimistic results on speedups for larger numbers of PCEs which needs to be taken into consideration when evaluating the results. Nevertheless, these assumptions allow us to study the effect of delays in a simple yet reasonably realistic environment.

Run time distributions. As a representative collection of SAT instances we use a set of benchmarks from the SAT 2007 Competition (see `http://www.satcompetition.org/2007/`). The instances, with the full name, abbreviated name, and satisfiability, are listed below.

– mod2-rand3bip-sat-250-3.shuffled-as.sat05-2220, `mod2-250`, satisfiable.
– mod2-rand3bip-sat-280-1.sat05-2263.reshuffled-07, `mod2-280`, satisfiable.
– 999999000001nc.shuffled-as.sat05-446, `99999900`, unsatisfiable.
– clqcolor-10-07-09.shuffled-as.sat05-1258, `clqcolor`, unsatisfiable.
– cube-11-h14, `cube`, satisfiable.
– dated-10-13-s, `dated`, satisfiable.
– mizh-md5-48-5, `mizh-md5`, satisfiable.
– vmpc_28.shuffled-as.sat05-1957, `vmpc_28`, satisfiable.
– AProVE07-16, `AProVE07`, unsatisfiable.

The set covers both industrial and hand-crafted instances, having typical run time of thousands of seconds for a state-of-the-art SAT solver.

The SAT solver run time distributions are approximated by using a collection of samples for each instance. The samples are obtained by 100 separate randomized runs of a state-of-the-art SAT solver (MiniSAT version 1.14 with its pseudo-random number generator initialized differently for each run). Based on the randomized runs, we construct a distribution of run times with linear interpolation between the sample points, assuming probability 0 for runs shorter than the minimum sample and for runs longer than the maximum sample. We also studied the case with discrete distribution, but this did not significantly affect our results.

Table 1documents for each instance the abbreviated names and the SAT solver run times for minimum, fifth percentile, median, 95th percentile and maximum of the samples. We also provide the average of the samples, i.e., an approximation of the expected run time of the solver on the instance, in the RSP column. The columns OPTIMUM, t^*, LUBY and WALSH will be explained in Sections 4 and 5. At this point, of particular interest are the large dynamics in certain distributions, such as `vmpc_28` with over 19000-fold difference between minimum and maximum run time. We also provide the cumulative run time distributions for two of the test instances in Fig. 2. The distribution is the increasing graph $q(t)$. The horizontal lines in Figures 2(b) and 2(d) indicate the maximum and minimum run times of the instance and the vertical line indicates the

Table 1. Characteristics of the run times for the test instances

Instance	Min	5%	Median	95%	Max	RSP	OPTIMUM	t^*	LUBY	WALSH
`mod2-250`	40.16	97.16	1210	2675	3088	1181	1181	∞	2715	1510
`mod2-280`	9.184	55.71	1732	6611	7775	2382	918.4	9.184	1274	1718
`99999900`	1072	1204	2056	3101	3725	2065	2065	∞	25070	4560
`clqcolor`	1198	1300	1922	2955	4329	1900	1900	∞	23060	4158
`cube`	2629	2896	4708	7936	10049	4832	4832	∞	106200	18500
`dated`	10.09	46.53	803.0	12550	37930	2279	716.1	29.08	901.5	993.3
`mizh-md5`	49.76	128.7	861.7	5784	9489	1660	1236	899.3	3403	1471
`vmpc_28`	0.1370	3.905	394.7	1730	2720	623.3	12.71	0.2560	137.4	279.6
`AProVE07`	879.4	1071	1471	2713	2855	1564	1564	∞	17330	3381

maximum run time on the x-axis, i.e., the value of t where $q(t) = 1$. The remaining graphs will be explained in Sections 4 and 5.

It can be argued that 100 samples is not enough to give us a realistic view of the run time distribution of an instance. In order to estimate the magnitude of the error introduced to the finite distribution, we compare the distributions of `cube` with 100 samples and 1000 samples. The results are reported in the first two rows of Table 2. Even though the minimum run time decreases and the maximum run time increases, the distribution seems to remain relatively stable when increasing the number of samples. To have an impression on how, for example, a short run would affect the results, we inserted an artificial short sample and constructed the corresponding distribution. The resulting distribution has the same dynamics as the distribution of `vmpc_28`.

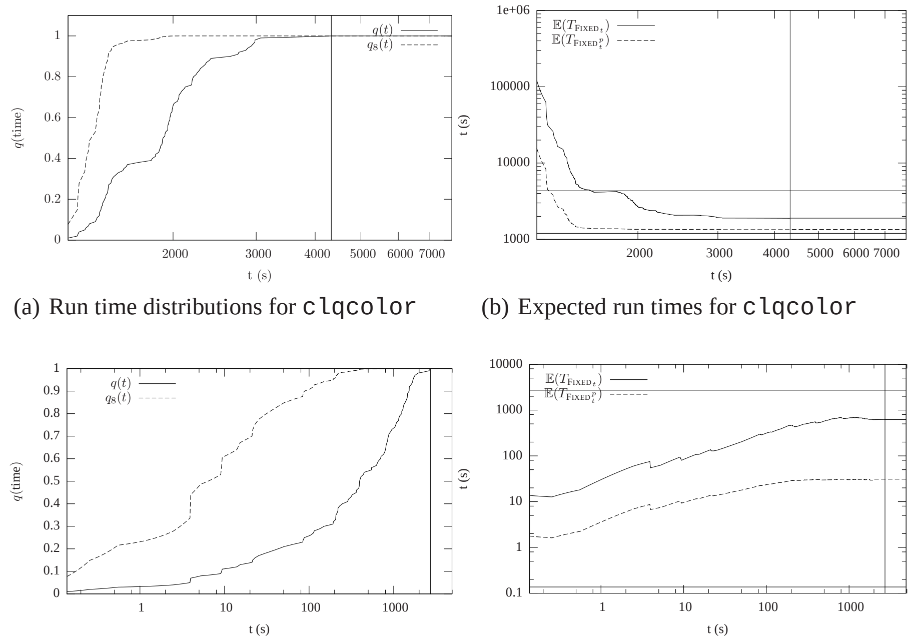
4 Restart Strategies in a Sequential Setting

Given a randomized search procedure RSP and a problem instance x, it is possible to associate a run time distribution $q_{\mathrm{RSP}(x)}(t)$ with the run time of $\mathrm{RSP}(x)$. Employing a restart strategy S on RSP results in a new algorithm with a potentially different run time distribution. In this section we discuss the effect of using several such algorithms on our collection of SAT instances by comparing the run time distributions $q_{\mathrm{RSP}(x)}(t)$ with the run time distributions of the new algorithms. We use the following restart strategies and corresponding algorithms:

– OPTIMUM. The fixed restart strategy S^t and the corresponding algorithm FIXED_t mentioned in Sect. 1 have the property that there is a restart limit t^* which is optimal

Table 2. Comparison of the distributions for cube with 100 samples (`cube`$_{100}$), 1000 samples (`cube`$_{1000}$), and a modified distribution with one artificial short run inserted (`cube`$_{1001m}$).

Instance	Min	5%	Median	95%	Max	RSP	OPTIMUM	t^*	LUBY	WALSH
`cube`$_{100}$	2629	2896	4661	7617	8821	4832	4832	∞	106200	18500
`cube`$_{1000}$	1441	2990	4914	7664	14051	5067	5067	∞	97360	31510
`cube`$_{1001m}$	0.7352	2990	4914	7647	14051	5061	725.9	0.735	5101	30280



(a) Run time distributions for `clqcolor` (b) Expected run times for `clqcolor`

(c) Run time distributions for `vmpc_28` (d) Expected run times for `vmpc_28`

Fig. 2. Run time distributions and expected run times for the instances `clqcolor` and `vmpc_28`

for a given RSP and instance x [3]. If the cumulative distribution function $q(t)$ of the instance is known, the optimal restart limit t^* may be determined by minimizing the expected run time $\mathbb{E}(T_{\textsc{Fixed}_{t}(x)})$ as a function of the restart limit t,

$$\mathbb{E}(T_{\textsc{Fixed}_{t}(x)}) = \frac{t - \int_{t'=0}^{t} q(t')dt'}{q(t)}, \tag{1}$$

i.e., $t^* = \text{argmin}(\mathbb{E}(T_{\textsc{Fixed}_{t}(x)}))$. Determining t^* can be done in our simulation environment but not usually in practice as the distribution $q(t)$ is typically not known.

– Luby. Luby et al. [3] define the universal strategy $S^{\text{L}} = (l(1), l(2), \ldots)$ where

$$l(i) = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1, k \in \mathbb{N} \\ l(i - 2^{k-1} + 1), & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

When the strategy S^{L} is employed on a RSP, the corresponding algorithm is called Luby. In [3] it is further shown that the expected run time of $\textsc{Luby}(x)$ is within a logarithmic factor from the expected run time of $\textsc{Optimum}(x)$ independently of x.

– Walsh. Another universal strategy is the strategy $S^{\text{W}} = (w(1), w(2), \ldots)$, where $w(i) = 2^{1.2i}$, presented in [4]. The strategy differs from S^{L}, for example, in the rate of growth. Clearly, the restart limits in S^{W} grow exponentially, whereas S^{L} grows only linearly with respect to i. The corresponding algorithm will be referred to as Walsh.

Table 1 compares the three algorithms against the run time of RSP. Column RSP reports the expected run time of $\mathrm{RSP}(x)$ for different instances x. Using the run time distribution $q_{\mathrm{RSP}(x)}(t)$, we computed the optimum restart limit t^* for each instance minimizing Eq. (1). The resulting expected run time is reported on column OPTIMUM and the corresponding restart limit in column t^*. The value ∞ is used to mark the cases when run times for $\textsc{Optimum}(x)$ and $\mathrm{RSP}(x)$ are equal. In this collection of instances, in five cases out of nine the expected run time of $\textsc{Optimum}(x)$ is equal to that of $\mathrm{RSP}(x)$. Some of the satisfiable instances, though not all, seem to profit from employing a fixed restart strategy with small restart limit. As an example, the expected run time for the algorithm $\textsc{Fixed}_t$ with input `vmpc_28`, is shown in Fig. 2(d) as a function of the restart limit t (graph labeled $\mathbb{E}(T_{\textsc{Fixed}_t})$). In other cases, the expected run times of algorithms with larger restart limits compare favorably to those with smaller restart limits. An example is shown in Fig. 2(b).

The results for the two universal strategies are shown in columns LUBY and WALSH of Table 1. Based on the results, it seems that in most cases the instances having $\mathbb{E}(T_{\textsc{Optimum}(x)}) \neq \mathbb{E}(T_{\mathrm{RSP}(x)})$ also profit of more complex strategies. We also note that LUBY performs very badly on many instances with a high minimum run time. This is a consequence of the slow growth of the restart limit in the strategy S^{L}. In general, the algorithm WALSH seems to offer a relatively robust approach, resulting in good speedup where such speedup would be obtainable with $\textsc{Fixed}_{t^*}$ given that t^* is known, and still performing usually well in cases where $\mathbb{E}(T_{\textsc{Optimum}(x)}) = \mathbb{E}(T_{\mathrm{RSP}(x)})$. This is a slightly surprising result, since to our knowledge no optimality result exists for the strategy S^{W}.

5 Parallel Solving of a Single Instance

In the previous section we discussed several restart strategies and resulting sequential algorithms when the strategies are employed to a RSP. In this section we develop a number of *parallel algorithms* for Grid environments based on the restart strategies. Here we consider a Grid environment as an efficient distributed system for running jobs. Hence, the algorithmic design boils down to approaches to constructing a sequence of jobs $j_1, j_2, \ldots$ to be submitted to the Grid for execution based on a RSP and a restart strategy. Since each job has a resource limit T_c limiting the execution time, we employ a *finite restart strategy* (discussed below) on the RSP which guarantees that the run time of the resulting algorithm is not more than T_c. Hence, each job j_i consists of the RSP, the input x to be solved and a finite restart strategy.

A *finite restart strategy* $S = (t_1, t_2, \ldots, t_n)$ is a finite sequence of restart limits which, when employed on a RSP, will terminate the resulting algorithm unless a solution is found by the end of the restart limit t_n. The *length* of the finite restart strategy S, denoted by $|S|$, is n. Given a restart strategy $S = (t_1, t_2, \ldots)$ and a resource limit T_c, we define an operator $\mathrm{finite}(S)$ for constructing finite restart strategies from S as

$$\mathrm{finite}(S) = \begin{cases} (T_c) & \text{if } t_1 > T_c \\ (t_1, t_2, \ldots, t_m) & \text{where } m \text{ maximizes } \sum_{i=1}^{m} t_i \leq T_c \text{ otherwise.} \end{cases}$$

For any restart strategy S, the run time of the algorithm obtained by employing $\mathrm{finite}(S)$ on a RSP is less than or equal to T_c.

The most intuitive way of constructing jobs from a restart strategy $S = (t_1, t_2, \ldots)$ is to assign the job j_i the restart strategy (t_i) for $i = 1, 2, \ldots$. In practice this approach performs very badly due to the high delays in actual Grid environments. Therefore, the parallel algorithms we propose are based on two general *schemes* for constructing a sequence of jobs, given a restart strategy S.

- *Straightforward scheme*. Given a restart strategy S for constructing jobs we define a sequence of restart strategies $S_1, S_2, \ldots$ in the following way: let $S_1 = S$ and given a strategy S_i, the restart strategy S_{i+1} is constructed from S_i by removing the first $|\mathrm{finite}(S_i)|$ restart limits from S_i. Given an environment with N PCEs, in the straightforward scheme jobs are constructed from the sequence $S_1, S_2, \ldots$ by assigning the restart strategy $\mathrm{finite}(S_1)$ for the jobs $j_1, \ldots, j_N$, then $\mathrm{finite}(S_2)$ for the jobs $j_{N+1}, \ldots, j_{2N}$ and so forth. This strategy is discussed in [13].
- *Faithful scheme*. In this scheme given a restart strategy S we construct the sequence $S_1, S_2, \ldots$ as above and then assign the job j_1 the restart strategy $\mathrm{finite}(S_1)$, the job j_2 the restart strategy $\mathrm{finite}(S_2)$, and so forth.

Parallel Algorithms. Given the randomized search procedure and the distributed environment, the parallel algorithm is uniquely determined by the used scheme (introduced above) and the restart strategy. Furthermore, for a fixed restart strategy, the straightforward and faithful schemes result in the same parallel restart strategy, and thus the same algorithm. We will discuss six parallel algorithms:

- The *maximum parallel algorithm* $\textsc{Fixed}^p_{T_c}$ is formed from the fixed restart strategy S^{T_c} and either straightforward or faithful scheme.
- The *optimal parallel algorithm* $\textsc{Fixed}^p_{t^*}$ is formed by finding a value t^* which minimizes the parallel run time distribution

$$\mathbb{E}(T_{\textsc{Fixed}^p_t(x)}) = \frac{t - \int_{t'=1}^{t} (1 - (1 - q(t'))^N) dt'}{1 - (1 - q(t))^N} \qquad (2)$$

  for $\mathrm{RSP}(x)$ with the run time distribution $q(t)$. Equation (2) is obtained from Eq. (1) by substituting $q(t)$ with the corresponding parallel distribution $1 - (1 - q(t))^N$. However, as shown in [13], there are run time distributions for which $\textsc{Fixed}^p_{t^*}$ does not result in minimum expected run time over all parallel algorithms.
- The *faithful parallel Luby and Walsh algorithms* $\textsc{Luby-f}^p$ and $\textsc{Walsh-f}^p$ are constructed by using the faithful scheme on the strategies S^{L} and S^{W}, respectively.
- The *straightforward parallel Luby and Walsh algorithms* $\textsc{Luby-s}^p$ and $\textsc{Walsh-s}^p$ are constructed by using the straightforward scheme on the strategies S^{L} and S^{W}, respectively.

Zero-Delay Parallel Environment. In this subsection we consider an idealized Grid environment captured by the Grid model, where we set the delays $d = d_q = 0$ and the resource limit $T_c = 3600$s. This provides us with a lower bound on the run times achievable in more realistic Grid environments.

Table 3. Results for different strategies and the zero-delay parallel environment

Instance	N	FIXED^p_{t*}	$\text{FIXED}^p_{T_c}$	LUBY-S^p	WALSH-S^p	LUBY-F^p	WALSH-F^p
`mod2-250`	16	105.7	116.2	334.2	177.5	171.8	114.0
	64	47.25	47.25	194.6	84.86	50.23	45.32
`mod2-280`	16	61.82	84.52	71.44	76.65	67.65	79.32
	64	19.36	21.55	22.29	25.69	21.44	24.58
`99999900`	16	1219	1219	14657	2910	1620	1238
	64	1097	1097	14530	2784	1213	1094
`clqcolor`	16	1293	1293	14730	2963	1553	1301
	64	1223	1223	14660	2899	1287	1224
`cube`	16	2891	2891	33600	6777	8105	2996
	64	2682	2682	33410	6570	3086	2687
`dated`	16	48.44	64.12	59.30	53.29	63.46	60.15
	64	15.89	16.33	15.92	16.05	14.69	19.26
`mizh-md5`	16	133.8	133.8	525.8	116.6	162.1	125.4
	64	73.23	73.23	259.2	126.1	84.53	81.76
`vmpc_28`	16	0.834	7.293	4.694	6.065	4.366	11.22
	64	0.251	0.539	0.6507	0.7994	0.6550	0.5003
`AProVE07`	16	1049	1049	11040	2285	1299	1064
	64	918.8	918.8	7823	1823	1056	915.4

We report the results for the maximum parallel algorithm in column $\text{FIXED}^p_{T_c}$ of Table 3 for 16 and 64 PCEs. For comparison, we also report on the column FIXED^p_{t*} the results when using the optimal parallel algorithm, in which case we use $T_c = \infty$.

The speedup is in most cases linear with respect to the added resources, and for `vmpc_28` even super-linear, for both $\text{FIXED}^p_{T_c}$ and FIXED^p_{t*}. For some instances, however, the speedup is negligible. It seems that there are certain distributions which do not allow for speedup when parallelized in this manner after a certain amount of PCEs has been reached. Two different examples of this phenomenon are closer studied in Figures 2(b) and 2(d) for $N = 1$ and $N = 8$. The graphs labeled $\mathbb{E}(T_{\text{FIXED}^p_t})$ in the figures are the expected run times of the algorithm FIXED^p_t with the respective instance as a function of the restart limit t. In Fig 2(b), the run time of the algorithm FIXED^p_t with large values of t is almost equal to that of the shortest sampled run (the lower horizontal line) which can also be seen from the run time distribution of the algorithm FIXED^p_t when $N = 8$, $q_8(t)$, in Fig 2(a). The situation is different in Fig 2(d), where the shortest run is much shorter than the expected run also when $N = 8$.

We also note that the difference between $\text{FIXED}^p_{T_c}$ and FIXED^p_{t*} becomes insignificant when N increases. The intuitive explanation for this is that the benefit of aggressive restarting can be obtained by running several solvers in parallel. The important consequence of the phenomenon is that with a large number of PCEs, the significance of the restart strategies decreases.

The remaining columns in Table 3 show the behavior of the strategies S^{L} and S^{W}. The results are obtained by simulating 100 runs of the parallel algorithms and reporting the mean time required to find the solution. The columns LUBY-S^p and WALSH-S^p correspond to the straightforward parallel restart strategy for S^{L} and S^{W}. This scheme

Table 4. Comparison of 64-PCE S^{L} and S^{W} with $f = 1.0$s, $f = 15.0$s, and $f = 100.0$s

	LUBY-F^p			WALSH-F^p		
Instance	$f = 1.0$	$f = 15.0$	$f = 100.0$	$f = 1.0$	$f = 15.0$	$f = 100.0$
mod2-250	68.09	50.23	47.19	46.54	48.85	48.55
mod2-280	34.71	21.44	20.16	23.82	21.69	18.55
99999900	1372	1213	1166	1093	1096	1105
clqcolor	1345	1287	1262	1220	1224	1222
cube	3950	3086	2977	2696	2688	2677
dated	28.43	14.69	18.71	18.74	15.85	18.57
mizh-md5	98.35	84.53	74.76	82.65	72.48	79.45
vmpc_28	0.5140	0.6550	0.6560	0.4717	0.5401	0.4870
AProVE07	1088	1056	992.2	930.2	936.2	914.76

has the benefit that small restart limits are attempted often. However, especially S^{L} suffers from the repeating of the short runs in cases where the smallest run time is high. The results corresponding to the faithful scheme are reported in columns LUBY-F^p and WALSH-F^p. In most cases the faithful scheme performs significantly better than the straightforward scheme, and when this is not the case, the difference is relatively small.

To further enhance the strategies S^{L} and S^{W}, we studied the effect of multiplying the restart limits of the strategies by a constant factor f in Table 4 for 64 PCEs. Based on these results, the factor does not seem to have a significant effect on the run times. The runs in Table 3 (as in Table 5) are measured with $f = 15.0$.

We study the effect of a larger sample base similar to the case in Table 2 in the zero-delay environment. The results are reported in Table 5. For this particular instance, the strategy FIXED$^p_{t*}$ is equal to the maximum strategy both when the amount of samples is 100 and 1000. In this case, when the number of samples is increased, the expected solving time decreases for most algorithms. There is no significant difference between WALSH-F^p and FIXED$^p_{T_c}$ whereas LUBY-F^p suffers from a larger number of short unsuccessful runs (even though not visible in Table 2, the distributions are significantly different when $t \leq T_c$; e.g. $q(3600\text{s}) \approx 0.24$ in the 100 samples distribution but only approximately 0.14 in the 1000 samples case). Since `cube` is a satisfiable instance, it is possible that there is a short run time for the randomized SAT solver. Since the 1000 samples did not reveal a short run time, it might be that the run is extremely improbable. To study the effect of such a short successful run we modify the distribution of `cube` to include a single short run. The resulting run times are given in the row la-

Table 5. Effect of additional samples on the zero-delay solving of `cube` with 64 PCEs

Instance	FIXED$^p_{t*}$	FIXED$^p_{T_c}$	LUBY-F^p	WALSH-F^p
$\texttt{cube}_{100}$	2682	2682	3086	2687
$\texttt{cube}_{1000}$	2364	2364	3760	2270
$\texttt{cube}_{1001m}$	11.86	2175	969.8	2185

beled `cube`$_{1001m}$. In this case, LUBY-F^p is better than FIXED$^p_{T_c}$ because of the higher probability of finding the short run.

Non-Zero Delay Parallel Environment. The simulation results from the parallel environment with zero submission delay and zero queuing delay provide some insight to how the parallelization method based on randomizing algorithms can perform on the benchmark set. However, realistic parallel environments in general, and Grid environments in particular, always include some overhead related to initializing the computations. As described in Sect. 3, we divide the delays into two categories: submit delay d and queue delay d_q. Typical values in NorduGrid are $d = 12$s and $d_q = 125$s. However, the two values seem to vary strongly. The simulated experiments are presented in Table 6 under the title “large delay”. All results are obtained by computing the mean run time over 100 samples using $T_c = 3600$s for the jobs.

The results show that almost always the maximum parallel algorithm FIXED$^p_{T_c}$ outperforms those based on universal restart strategies on these instances. It is worth noting that increasing the number of PCEs four-fold brings next to nothing in speedup, a consequence of the long queuing delays.

It is possible that the submission and queue delays are significantly shorter in, say, some other Grid environments. We simulate the effect of smaller delays by using submission delay $d = 5$s and queue delay $d_q = 30$s. The results are reported under the caption “small delay”. Even though the strategies S^{L} and S^{W} are now more competitive, their effectiveness still suffers from the high delays and it can be argued that the maximum timeout is a sufficient approximation of the optimum. The super-linear speedup observed in zero-delay environment cannot be observed in either of the delayed environments. For certain instances, such as `99999900` and `cube`, already a smallish number of parallel runs suffices to find a short run from the samples. As a result, obtainable speedup is small.

We confirm these results by repeating them for two instances in the NorduGrid Grid environment. We select two instances which according to the simulated results are illustrative examples on the techniques used in parallel solving. The instance `vmpc_28` shows super-linear speedup in simulations in zero-delay environments, but only a moderate speedup in delayed environments using the techniques we have studied. The instance `AProVE07`, on the other hand, has a less dynamic distribution in the simulations and yields no significant speedup at the transition from 16 to 64 PCEs even in the zero-delay environment. The results are presented in Table 7. The submission delays seem to be below the average delay of 12 seconds, but the results correspond approximately to the simulated results. No speedup seems to be achieved when the number of PCEs is increased.

6 Parallel Solving of a Set of Instances

In this section we propose an algorithm for solving a collection of SAT problems efficiently in a Grid environment based on the results on solving a single instance. The results indicate that (i) an increase in the number of PCEs does not result in a corresponding speedup when solving a single instance and (ii) for a large number of prob-

Table 6. Results for different strategies and delayed parallel environments. The two rows for each instance correspond to $N = 16$ (top) and $N = 64$ (bottom)

	small delay				large delay			
Instance	$\text{FIXED}^p_{t^*}$	$\text{FIXED}^p_{T_c}$	LUBY-F^p	WALSH-F^p	$\text{FIXED}^p_{t^*}$	$\text{FIXED}^p_{T_c}$	LUBY-F^p	WALSH-F^p
`mod2-250`	177.0	145.1	232.7	164.4	352.8	379.3	399.8	399.3
	161.5	157.7	182.7	133.5	364.4	355.7	422.7	350.4
`mod2-280`	125.8	159.0	137.4	150.7	306.8	331.0	321.4	350.0
	118.1	126.2	135.2	132.4	296.3	327.7	320.9	340.1
`99999900`	1242	1268	1672	1306	1431	1477	1984	1527
	1208	1246	1401	1253	1432	1485	1756	1490
`clqcolor`	1340	1353	1455	1378	1506	1525	1846	1577
	1328	1351	1448	1352	1508	1536	1777	1554
`cube`	2882	2960	9209	3067	3094	3117	9233	3195
	2792	2840	3489	2842	3050	3121	4159	3145
`dated`	112.1	140.5	138.2	126.1	272.2	323.8	281.6	312.1
	104.7	114.1	116.6	117.4	284.3	309.2	293.8	305.8
`mizh-md5`	181.4	190.4	268.7	199.4	352.6	391.2	445.0	395.3
	190.4	186.3	208.5	195.0	379.8	385.2	464.8	392.0
`vmpc_28`	43.27	67.35	62.70	65.49	155.7	206.7	198.4	214.0
	42.18	68.06	62.59	64.30	155.5	218.3	200.0	212.0
`AProVE07`	1073	1089	1313	1127	1262	1289	1569	1310
	1073	1065	1205	1061	1292	1299	1568	1300

Table 7. Experimental results in Grid for selected instances. Reported is the average over 10 runs using the strategy S^{T_c}.

Instance	PCEs	Time	d
`vmpc_28`	8	105.4	3.333
	16	125.7	7.668
	64	134.5	5.189

Instance	PCEs	Time	d
`AProVE07`	8	1624	5.917
	16	1574	9.714
	64	1271	8.555

lems to solve, a good speedup is not obtained by using all the resources for solving a single problem at a time, but rather by dedicating only a certain amount of PCEs for a single problem and solving multiple problems simultaneously instead. These observations lead to the following *locally-aided fair-share algorithm*: Given a collection of instances, the instances are sent for solving in a round-robin manner by using the maximum parallel algorithm $\text{FIXED}^p_{T_c}$. In addition, the problems are also solved locally at the same time using an algorithm similar to LUBY with the modified strategy $S^{\text{L},C} = (\min\{l(1), C\}, \min\{l(2), C\}, \ldots)$, where C is a maximum local run time constant, in a round-robin manner.

We provide experimental evidence that the proposed algorithm is efficient in a real Grid environment. For this experiment, we select 8 problems from our benchmark set of 9 problems and run them in parallel with 64 PCEs, reserving at most eight PCEs per problem. This enables us to compare the results of this experiment against a strategy where 64 PCEs are dedicated for a single instance at a time. We first exclude `cube` from the set of instances, since this problem is in the limit of solvable problems within 3600 seconds in our Grid environment, having expected run time of 4708 seconds in the simulation environment. The resulting run time for the full instance set is 1865 seconds. The sum of the simulated run times for these instances from Table 6 is 5916 seconds. This results in a speedup 3.17 compared to the strategy of using 64 PCEs per instance. When these results are compared against a simple strategy of running the problems on a single PCE with no delays, the speedup computed from the results of Table 1 is 7.32.

However, we note that the results can be significantly worse if a difficult instance, such as `cube`, is included in the set of problems to solve. We repeated the above experiment with 10 repetitions, now using 72 PCEs, resource limit $T_c = 7200$ seconds and including `cube` to the set of problems to solve. This resulted in a speedup of 1.76 with average solving time of 5136 seconds in the Grid environment compared to the expected solving time of 9037 seconds with long delays and 64 PCEs in Table 6. When these results are compared against a simple strategy of running the problems on a single PCE with no delays, the speedup is 3.60.

7 Conclusions

In this paper we have developed techniques for solving collections of hard SAT instance in a Grid using a randomized SAT solver. We have compared different approaches using a simulation framework consisting of a grid model capturing the communication and management delays, and a representative collection of run-time distributions of a randomized solver. The results are experimentally confirmed also in NorduGrid which is a European-wide distributed production level Grid. When solving a single hard SAT instance, the results show that in practice often (i) a relatively small number of parallel jobs suffices to increase the probability of finding a short run in the distribution to a significant level and (ii) the non-negligible delays in a Grid eliminate super linear speedups that could be obtained in an ideal environment without any delays. Hence, attempts to decrease the overall expected run time by using clever universal restart strategies or by finding optimal restart limits do not lead to significant improvements compared to using the resource limit implied by the Grid environment as the restart limit. These observations lead to a novel strategy of using Grid to solve collections of hard instances. Instead of solving instances one-by-one, the strategy aims at decreasing the overall solution time by applying an alternating distribution schedule.

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, and Jenny and Antti Wihuri Foundation is gratefully acknowledged.

References

1. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT 2003. Volume 2919 of LNCS., Springer (2003) 502–518

2. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Automated Reasoning **24**(1/2) (2000) 67–100
3. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett **47**(4) (1993) 173–180
4. Walsh, T.: Search in a small world. In: IJCAI, Morgan Kaufmann (1999) 1172–1177
5. Kautz, H.A., Horvitz, E., Ruan, Y., Gomes, C.P., Selman, B.: Dynamic restart policies. In: AAAI/IAAI. (2002) 674–681
6. Ruan, Y., Horvitz, E., Kautz, H.A.: Restart policies with dependence among runs: A dynamic programming approach. In: CP 2002, Proceedings. (2002) 573–586
7. Streeter, M., Golovin, D., Smith, S.F.: Restart schedules for ensembles of problem instances. In: AAAI, AAAI Press (2007) 1204–1210
8. Huang, J.: The effect of restarts on the efficiency of clause learning. In: IJCAI. (2007) 2318–2323
9. Wu, H., van Beek, P.: On universal restart strategies for backtracking search. In: CP. Volume 4741 of LNCS., Springer (2007)
10. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1-2) (2001) 43–62
11. Wu, H., van Beek, P.: On portfolios for backtracking search in the presence of deadlines. In: ICTAI. (2007) 231–238
12. Inoue, K., et al.: A competitive and cooperative approach to propositional satisfiability. Discrete Applied Mathematics **154**(16) (2006) 2291–2306
13. Luby, M., Ertel, W.: Optimal parallelization of Las Vegas algorithms. In: STACS. Volume 775 of LNCS., Springer (1994) 463–474
14. Boehm, M., Speckenmeyer, E.: A fast parallel SAT-solver: Efficient workload balancing. Annals of Mathematics and Artificial Intelligence **17**(4-3) (1996) 381–400
15. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. J. Symbolic Computation **21**(4) (1996) 543–560
16. Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel multithreaded satisfiability solver: Design and implementation. Electronic Notes in Theoretical Computer Science **128**(3) (2005) 75–90
17. Blochinger, W., Westje, W., Küchlin, W., Wedeniwski, S.: ZetaSAT – Boolean satisfiability solving on desktop grids. In: CCGrid 2005, IEEE (2005) 1079–1086
18. 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
19. 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
20. Chrabakh, W., Wolski, R.: GridSAT: A chaff-based distributed SAT solver for the grid. In: SC 2003, IEEE (2003)
21. 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
22. 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`.
23. Pitkanen, M.J., et al.: Using the grid for enhancing the performance of a medical image search engine. In: CBMS 2008, IEEE (2008) Accepted for publication.
24. Hyvärinen, A.E.J.: GridJM a Computer Program. `http://www.tcs.hut.fi/~aehyvari/gridjm/`.