Designing Scalable Parallel SAT Solvers

Antti E.J. Hyvärinen[1,⋆] and Norbert Manthey[2]

[1] Aalto University
Department of Information and Computer Science
P.O. Box 15400, FI-00076 AALTO, Finland
antti.hyvarinen@aalto.fi
[2] Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

Abstract. Solving instances of the propositional satisfiability problem (SAT) in parallel has received a significant amount of attention as the number of cores in a typical workstation is steadily increasing. With the increase of the number of cores, in particular the scalability of such approaches becomes essential for fully harnessing the potential of modern architectures. The best parallel SAT solvers have, until recently, been based on algorithm portfolios, while search-space partitioning approaches have been less successful. We prove, under certain natural assumptions on the partitioning function, that search-space partitioning can always result in an increased expected run time, justifying the success of the portfolio approaches. Furthermore, we give first controlled experiments showing that an approach combining elements from partitioning and portfolios scales better than either of the two approaches and succeeds in solving instances not solved in a recent solver competition.

1 Introduction

The satisfiability problem (SAT) of determining whether a given propositional formula has a satisfying truth assignment has been a target of intense research efforts due to its theoretical significance [1] and the numerous applications, such as scheduling [2], termination analysis [3], configuration [4], and bioinformatics [5], where SAT solvers have proven successful. Parallelism seems now to dominate the performance of future computer systems, as already current computers provide more than ten CPU cores. As a result, the research on how to parallelize SAT solvers for an increasing number of cores is of high practical relevance.

This paper uses rigorous analysis and experiments to find a novel explanation to the effects certain well-known parallelization techniques have on the expected run time of solving SAT instances. The time a SAT solver $\mathcal{S}$ requires to solve a given formula ϕ is known to be highly erratic and might vary significantly as the formula or the solver is modified even slightly. Hence, given a solver $\mathcal{S}$ and

⋆ Financially supported by the Academy of Finland under the Finnish Centre of Excellence in Computational Inference (COIN) and the project number 122399.

A. Cimatti and R. Sebastiani (Eds.): SAT 2012, LNCS 7317, pp. 214–227, 2012.

a formula ϕ, the run time is a random variable. The variance in run times has two important implications in parallel solving of formulas. Firstly, assume that the run time of $\mathcal{S}$ on ϕ is one second with probability 0.8 and ten seconds with probability 0.2. The expected run time of the solver $\mathcal{S}$ is then $1 \cdot 0.8 + 10 \cdot 0.2 = 2.8$ seconds. Running ten (randomized) solvers in parallel gives the solution in the expected time $(1 - 0.2^{10}) \cdot 1 + 0.2^{10} \cdot 10 \approx 1.000001$ seconds. Hence the use of this approach results in speed-up of 2.8. Secondly, assume there is a way of partitioning ϕ's search space into ten separately in parallel solvable, equally difficult parts. It is reasonable to assume that such a partitioning is not perfect so that the run time of each partition is, say, half of the original formula instead of tenth. Proving unsatisfiability of the formula would in this case require ensuring that there are no solutions in any of the partitions, and the expected run time with the partitioning approach is thus $0.8^{10} \cdot 1 \cdot 1/2 + (1 - 0.8^{10}) \cdot 10 \cdot 1/2 \approx 4.5$ seconds, resulting in speedup of 0.6. This artificial example provides some insight to why the *portfolio solvers*, corresponding to the first case, perform often better than the *search space partitioning solvers* which correspond to the second case. The portfolio approach provides a substantially better speed-up, while the partitioning approach results in fact in a higher expected run time than solving with the underlying solver $\mathcal{S}$.

In this paper we prove, under reasonable assumptions, that there is always a distribution which results in a similar increased expected run time as in the example above, unless the process of constructing partitions is *ideal* in the sense described later. Earlier it has been shown that by organizing the partitioning as a *partition tree* it can be guaranteed that not only the expected run time does not increase above that of $\mathcal{S}$, but that increasing the number of parallel resources never increases the expected run time [6]. We experimentally confirm this using realistic and comparable implementations by showing that the partition tree based *iterative partitioning approach* scales better than either the portfolio approach or the partitioning approach. The implementation is able to solve four instances that were not solved in the SAT Competition 2011.

The run times of randomized tree-based searches have been studied analytically both for sequential solving [7] and in parallel cases [8,9,10,11]. Our analytic discussion differs from these by studying unsatisfiability proofs with a model of partitioning function that is an extension of [11]. Much work has been invested in studying search space partitioning solvers [12,13,14,15] and algorithm portfolios [16,17]. In this work our aim is to build understanding between the two approaches by implementing similar systems in as comparable manner as possible, omitting for instance the most sophisticated clause sharing mechanisms [18,19,20]. The iterative partitioning approach discussed in this work is introduced in [21] and further developed in [6] and [22]. We extend these studies by implementing the approach for multi-core architectures instead of computational grids, which enables us to provide a much more reliable comparison of the iterative partitioning approach to the portfolio and partitioning approaches.

The work is organized as follows: Section 2 defines our notation. Section 3 presents the proof for increasing run times, and formalizes the three parallelization approaches. Section 4 provides the multi-core implementations of the approaches and discusses how clause sharing is implemented in them. The implementations are experimentally evaluated in Sect. 5, and conclusions are given in Sect. 6.

2 Preliminaries

Let V be a finite set of Boolean variables. The set of *literals* $\{x, \neg x \mid x \in V\}$ consists of negative and positive Boolean variables, a *clause* is a disjunction of literals and a *formula* (in conjunctive normal form) is a conjunction of clauses. Whenever convenient, we denote clauses by sets of literals and formulas by sets of clauses. Clauses of size one are called *unit clauses*. An *assignment* σ is a set of literals, is consistent if for no variable x both $x \in \sigma$ and $\neg x \in \sigma$, and is inconsistent otherwise. If an assignment σ does not contain a literal l for each variable $v \in V$, it is called partial. A consistent assignment σ satisfies a clause C if C contains a literal in σ, and satisfies a formula if it satisfies all its clauses. A formula is *satisfiable* if there is a consistent truth assignment satisfying it, and *unsatisfiable* otherwise. A formula ψ is a logical consequence of ϕ, denoted $\phi \models \psi$, if ψ is satisfied by all satisfying assignments of ϕ. The formulas are *logically equivalent*, denoted $\phi \equiv \psi$, if they are logical consequences of each other.

3 Parallel Solving Approaches

This work studies the parallel SAT solver designs that have recently proved successful. In particular, we will discuss

- the *Simple Parallel SAT Solving* (SPSAT) approach, which is a simplified variant of the portfolio approach;
- the *plain partitioning* approach, which again is a simplified version of the search space partitioning approaches such as those based on *guiding paths* [23]; and
- the *iterative partitioning* approach, again a simple approach where partitioning is recursive and the solving is attempted on the search spaces related to all recursive levels until satisfiability is proved.

In the following, we describe the approaches and the concepts related to them in more detail.

The SPSAT Approach is based on solving a given formula ϕ with several SAT solvers in parallel. As all solvers are working on the same instance, the solution is obtained from the solver finishing first. The underlying solvers are slightly randomized so that they correspond to a straightforward portfolio of seemingly infinite different algorithms. The approach is known to be efficient for a wide range of application originated formulas, in particular if they are satisfiable and have several solutions [11].

The Plain Partitioning Approach first divides the search space of the formula, and then solves the resulting partitions separately in parallel. The search space is divided using a *partitioning function* $P(\phi, n)$, which maps a formula ϕ to n *partitioning constraints* $\kappa_1 \ldots, \kappa_n$ such that (i) $\phi \equiv (\phi \wedge \kappa_1) \vee \ldots \vee (\phi \wedge \kappa_n)$, and (ii) $\kappa_i \wedge \kappa_j \wedge \phi$ is unsatisfiable when $i \neq j$. The formulas $\phi_i = \phi \wedge \kappa_i$, $1 \leq i \leq n$, are called the *derived formulas* of ϕ. The satisfiability of ϕ can be determined by either showing all derived formulas unsatisfiable or showing ϕ_i satisfiable for some $1 \leq i \leq n$. By (i), in the former case also ϕ is unsatisfiable, and in the latter case the assignment satisfying ϕ_i satisfies also ϕ.

The Iterative Partitioning Approach is based on solving a hierarchical *partition tree* in a breadth-first order. Given a formula ϕ, iteratively constructed derived formulas can be presented by a *partition tree* T_ϕ. Each node ν_i is labeled with a set of clauses $Co(\nu_i)$ so that the root ν_0 is labeled with $Co(\nu_0) = \phi$, and given a node ν_k and a rooted path $\nu_0, \ldots, \nu_{k-1}$ to its parent, the label of ν_k is $Co(\nu_k) = \kappa_i$, where κ_i is one of the constraints given by $P(\bigwedge_{j=0}^{k-1} Co(\nu_j), n)$. Each node ν_k with a rooted path $\nu_0, \ldots, \nu_k$ represents the formula $\phi_{\nu_k} = \bigwedge_{i=0}^{k} Co(\nu_i)$. Solving is attempted for each ϕ_{ν_k} in the tree in a breadth-first order. The approach terminates if a satisfying assignment is found, or all rooted paths to the leaves contain a node ν_j such that ϕ_{ν_j} is shown unsatisfiable. In practice, we also limit the run time of each solving attempt to ensure that a reasonably large portion of the search tree will be covered.

The Partitioning Function Model. A partitioning function $P(\phi, n)$ should produce derived formulas ϕ_i which are increasingly faster to solve as the number of derived formulas n increases. We will use an *efficiency function* to formalize how well P accomplishes this. Assume that the solver $\mathcal{S}$ performs with the same probability a given search that takes time t_ϕ in the formula ϕ but, due to the partitioning constraints, a shorter time t_{ϕ_i} in the derived formulas ϕ_i. The efficiency function $\epsilon(n)$ depends on the number n of derived formulas and gives the ratio of the two times, that is, $\epsilon(n) = t_\phi / t_{\phi_i}$.

We use a cumulative run time distribution $q_{\mathcal{S},\phi}(t)$ to describe the probability that a solver $\mathcal{S}$ determines the satisfiability of a formula ϕ in time t. This reasoning results in a model where, given a formula ϕ with the run time distribution $q_{\mathcal{S},\phi}(t)$ on a solver $\mathcal{S}$, the n derived formulas ϕ_i all have the distributions $q_{\mathcal{S},\phi_i}(t) = q_{\mathcal{S},\phi}(\epsilon(n)t)$.

We will only consider efficiency functions of the form $\epsilon(n) = n^\alpha$ where $0 \leq \alpha \leq 1$ is a constant depending on the partitioning function. The function satisfies the following natural properties:

(1) $1 \leq \epsilon(n) \leq n$,
(2) $\epsilon(n) \leq \epsilon(n+1)$, and
(3) $\epsilon(n)^p = \epsilon(n^p)$ for all $p \in \mathbb{N}$

The first condition states that the partitioning function should not make a particular search of S super-linearly faster or slow the search down. The second condition requires that the efficiency does not decrease as more derived formulas are

created. The last condition states that if a partitioning function $P(\phi, n)$ is used to produce n^p derived formulas recursively, the resulting efficiency must equal the efficiency of $P(\phi, n^p)$ where the derived formulas are all generated at once. Hence, given a partitioning function P with efficiency $\epsilon(n) = n^\alpha$, the cumulative run time distributions for the derived formulas ϕ_i of ϕ are

$$q_{\mathcal{S},\phi_i}(t) = q_{\mathcal{S},\phi}(n^\alpha t) \text{ for some } \alpha \text{ in the range } 0 \leq \alpha \leq 1, \tag{1}$$

where the partitioning function is called *ideal* if $\alpha = 1$, that is, $\epsilon(n) = n$.

3.1 Plain Partitioning Can Increase Expected Run Time

A run time distribution $q_{\mathcal{S},\phi}(t)$ for an unsatisfiable ϕ completely determines the run time distribution $q^n_{\text{Plain-Part}(\alpha),\phi}(t)$ for the plain partitioning approach with a partitioning function P with efficiency $\epsilon(n) = n^\alpha$. In particular, since the formula ϕ is shown unsatisfiable once all derived formulas have been shown unsatisfiable, by (1) we have

$$q^n_{\text{Plain-Part}(\alpha),\phi}(t) = q^n_{\mathcal{S},\phi_i}(t) = q^n_{\mathcal{S},\phi}(n^\alpha t). \tag{2}$$

In this section we are interested in studying the expected value of the random variables $T_{\mathcal{S},\phi}$ and $T^n_{\text{Plain-Part}(\alpha),\phi}$ describing the times required to solve ϕ with the solver $\mathcal{S}$ and the plain partitioning approach using n derived formulas, respectively. In particular, we wish to prove the somewhat surprising claim that for non-ideal partitioning functions there are distributions for unsatisfiable formulas such that the expected run time of the solver $\mathcal{S}$ is less than the expected run time of the plain partitioning approach, stated more formally as follows:

Proposition 1. *Let ϕ be unsatisfiable, $P(\phi, n)$ a partitioning function with efficiency $\epsilon(n) = n^\alpha$, and $\mathcal{S}$ a SAT solver. Then for sufficiently large n and every $0 \leq \alpha < 1$ there exists a distribution $q^n_{\mathcal{S},\phi}(t)$ such that the expected run time $\mathbb{E}T_{\mathcal{S},\phi}$ of $\mathcal{S}$ is lower than the expected run time $\mathbb{E}T^n_{Plain\text{-}Part(\alpha),\phi}$ of the plain partitioning approach.*

Proof. The family of distributions $q^n_{\mathcal{S},\phi}(t)$ we will use in the proof is

$$q^n_{\mathcal{S},\phi}(t) = \begin{cases} 0 & \text{if } t < t_1, \\ 1 - \frac{1}{n} & \text{if } t_1 \leq t < t_2, \textit{and} \\ 1 & \text{if } t \geq t_2, \end{cases} \tag{3}$$

where $t_1 < t_2$. Thus the probabilities that the formula is solved by $\mathcal{S}$ exactly in time t_1 is $1-1/n$ and exactly in time t_2 is $1/n$. The expected run time for a formula following the distribution is

$$\mathbb{E}T_{\mathcal{S},\phi} = (1 - \frac{1}{n})t_1 + \frac{1}{n}t_2. \tag{4}$$

The expected run time of the plain partitioning approach using the partition function $\epsilon(n) = n^\alpha$ can be derived by noting that all derived formulas need to be solved

before the result can be determined. This means that either all solvers are "lucky", and determine the unsatisfiability in time t_1/n^α, or at least one of the solvers runs for time t_2/n^α, which will then become the run time of the approach. This results in

$$\mathbb{E}T^n_{\text{Plain-Part}(\alpha),\phi} = \left(1 - \frac{1}{n}\right)^n \frac{t_1}{n^\alpha} + \left(1 - (1 - \frac{1}{n})^n\right) \frac{t_2}{n^\alpha}. \tag{5}$$

We claim that for every α, there are values for n, t_1 and t_2 such that $\mathbb{E}T_{\mathcal{S},\phi} < \mathbb{E}T^n_{\text{Plain-Part}(\alpha),\phi}$. Dividing both sides of the resulting inequality by t_2 and setting $k = t_1/t_2$ results in

$$(1 - \frac{1}{n})k + \frac{1}{n} < \frac{(1 - \frac{1}{n})^n}{n^\alpha}k + \frac{1 - (1 - \frac{1}{n})^n}{n^\alpha},$$

which can be reordered to

$$k\left((1 - \frac{1}{n}) - \frac{(1 - \frac{1}{n})^n}{n^\alpha}\right) < \frac{1 - (1 - \frac{1}{n})^n}{n^\alpha} - \frac{1}{n}.$$

We note that $(1 - \frac{1}{n}) > (1 - \frac{1}{n})^n/n^\alpha$ when $n \geq 2$, and therefore the left side of the inequality is positive and can be made arbitrarily small by setting k small. It remains to show that the right side of the inequality is positive for sufficiently large n, i.e.,

$$\frac{n - (1 - \frac{1}{n})^n n - n^\alpha}{n^{\alpha+1}} > 0.$$

Since $n^{\alpha+1}$ is always positive, we may simplify this and factor n from the nominator, resulting in

$$1 - (1 - \frac{1}{n})^n - n^{\alpha-1} > 0. \tag{6}$$

Noting that $\lim_{n\to\infty}(1 - \frac{1}{n})^n = \frac{1}{e} \approx 0.3$, and that $\lim_{n\to\infty} 1 - n^{\alpha-1} = 1$ if $\alpha < 1$, we get the desired result, that is, for sufficiently large n, there are values t_1 and t_2 such that $t_1 < t_2$ and $\mathbb{E}T_{\mathcal{S},\phi} < \mathbb{E}T^n_{\text{Plain-Part}(\alpha),\phi}$.

Note that the proof does not hold if the partitioning function is ideal, since the left hand side of the inequality (6) is negative if $\alpha = 1$. In fact, we have the following proposition proved in [11]:

Proposition 2. *Let $n \geq 1$, $\epsilon(n) = n^1 = n$ be the efficiency of an ideal partitioning function, and $q_T(t)$ be the run time distribution of an unsatisfiable formula ϕ with a randomized solver. Then $\mathbb{E}T^n_{Plain\text{-}Part(1),\phi} \geq \mathbb{E}T^{n+1}_{Plain\text{-}Part(1),\phi}$.*

The distribution $q^n_{\mathcal{S},\phi}(t)$ used in proof of Prop. 1 is clearly not a common distribution for any solver and unsatisfiable formula. Furthermore, many search space partitioning solvers are based on *guiding paths*, an approach designed to increase dynamically the number of derived formulas as the instance is being solved. Nevertheless we believe that the observation helps to understand the performance of parallel SAT solvers and thereby gives guidelines how to design better parallel solvers. To further evaluate the effect in practice, we compare the plain partitioning approach against the SPSAT approach and the iterative partitioning approach, both of which provably do not suffer from the increasing expected run times (see [11] and [24], respectively, for proofs).

4 Multi-core Implementations

The partitioning approaches discussed in Sect. 3 can be implemented in a relatively straightforward manner using the efficient off-the-shelf SAT solvers that are readily available. Our implementations use the POSIX threads library to enable multi-threaded computing. The SPSAT implementation is straightforward, and the plain partitioning approach can be seen as a special case of the iterative partitioning, where only the original formula is partitioned, and only the derived formulas are solved. Hence this section concentrates on describing the iterative partitioning approach. In interpreting the experimental results it is useful to keep in mind that the low-level SAT solving, corresponding to the underlying solver $\mathcal{S}$ in the analytical model, is performed by the same code in all three approach. This allows us to compare the results more reliably.

4.1 The Iterative Partitioning Approach

The iterative partitioning approach is implemented as a master-worker architecture, where the master maintains a tree of derived formulas and the workers both compute the partitioning function and run the underlying solvers. Communication is handled via shared memory and the locking primitives available from the library. The master thread takes care of the following tasks:

1. Maintaining the partition tree
2. Maintaining the queue of nodes to partition
3. Submitting partitioning tasks
4. Submitting solving tasks
5. Determining whether the search can be terminated

There are two kinds of workers: the *partitioner* and the *solver*. The maximum run times of both workers are limited. The partitioner takes as input a node ν_i in the partition tree and a number n, and produces the derived formulas computed by the partitioning function $P(\phi_{\nu_i}, n)$ upon reaching the time limit. The solver receives a node ν_i from the partition tree and tries to solve the corresponding formula ϕ_{ν_i}. At success, the solver returns either a satisfying truth assignment or concludes that ϕ_{ν_i} is unsatisfiable. Otherwise, if the run time limit is reached, no solution is returned and the corresponding node is marked *unknown*. Such nodes are subject to at most one partitioning and their satisfiability will be determined by attempting to solve recursively the formulas corresponding to child nodes.

A node ν_i is solved by first constructing the corresponding formula ϕ_{ν_i}. After successful solving of ϕ_{ν_i}, the master either updates the state of ν_i to *unsatisfiable* or receives the satisfying truth assignment depending on the outcome of the solver. Otherwise, if the solving of ϕ_{ν_i} failed due to a timeout, the state of ν_i remains *unknown*.

In case of receiving an unsatisfiable result on a node ν_i, the master checks the states of the sibling nodes. In case they all are already in the state *unsatisfiable*, also the parent of ν_i, if one exists, is marked *unsatisfiable*. This process is repeated

recursively upwards. This way a node in the tree is marked *unsatisfiable* if and only if all paths from the node to the leaves pass through a node corresponding to a formula that is shown unsatisfiable with the solver.

The partitioner implements the *vsids* scattering function [21], where the partitioning constraints are in general clauses consisting of literals with a high *vsids* [25] score.

4.2 Clause Learning

To keep the discussion and the results generalizable, the underlying solvers of the approaches are only allowed a limited form of learned clause sharing. In particular, the sharing of only unit clauses is allowed, since sharing longer clauses might have negative impact on the overall performance [20,22].

The SPSAT implementation synchronizes its units with a centralized database at every restart and when learning a new unit clause. This operation can be performed with no locks with a Compare-and-Swap instruction, and has no noticeable negative performance effect. Clause sharing is less straightforward in the partitioning approaches. A clause learned in one derived formula is not, in general, a logical consequence of another derived formula, and hence the learned clauses are not transferred between derived formulas. The iterative partitioning approach shares the unit clauses only "downwards", that is, clauses learned in a node are shared with the formulas in the subtree rooted at that node, by storing the units learned while solving a formula ϕ_{ν_i} to the constraints $Co(\nu_i)$. In plain partitioning, the unit clauses are similarly saved to the constraints of the derived formula from which they are learned, and are hence shared between two consecutive solvers if the first solver fails.

It is possible to maintain more complicated data structures which allow tracking to some extent from which constraints a given clause depends. This usually involves an overhead some times high enough to completely ruin the speed-up obtained from the parallelization [22]. For simplicity and to help in interpreting the comparison, such data structures are not implemented in our experiments.

5 Results

This section analyzes the performance of the SPSAT, the plain partitioning and the iterative partitioning approaches using the application category instances from the 2009 and 2011 SAT competitions[1]. We first compare the wall clock run time of each solving approach to that of the underlying solver, and then study the scalability of the approaches with four and 12 cores. We continue by comparing the plain and iterative partitioning approaches, by showing how the iterative partitioning approach scales when moving to a grid-based system, and finally report on solving the instances that were not solved in SAT competition 2011. The reliability of the results is addressed shortly by solving repeatedly certain randomly chosen instances.

[1] See `http://www.satcompetition.org/`

5.1 Experimental Setup

All three approaches use MiniSAT 2.2.0 [26] as the underlying solver[2]. We use preprocessing only in the last experiment. The experiments are run on a cluster consisting of nodes with two six-core AMD Opteron 2435 processors. Each instance was solved on an exclusively reserved computing node. The memory usage for each instance was limited to 30 GB and the duration to four hours of wall clock time. Each thread was allocated an equal amount of memory, that depended on the number of threads used. For instance, when running 12 threads, each thread had approximately 2.5 GB of memory. If the thread ran out of memory, the unit clauses learned by the thread were collected and, in case of the SPSAT and plain partitioning approaches, the thread was restarted with the same formula.

The measurement of memory usage is always an estimate, and therefore the system may nevertheless run out of memory resulting in an early termination of the search. The run time of each solver thread in the SPSAT and the plain partitioning approaches was limited to four hours, while run times of the solver threads in the iterative partitioning approach was limited to 2400 seconds wall clock time (however, the master thread still had the time limit of four hours).

The partitioning function used in the iterative partitioning approach constructed eight derived instances, that is, it was the function $P(\cdot, 8)$. The plain partitioning used the function $P(\cdot, 1000)$ to obtain roughly the same amount of formulas in total for both approaches. Increasing the number of derived formulas increases the probability that some of them are trivially unsatisfiable. In 50% of the 2009 benchmark formulas the number of non-trivial derived formulas was over 200, and in 25% of the formulas the number was over 600. In total 70 seconds were allocated for computing the partitioning function in both cases.

In most of the experiments, we illustrate the results with scatter plots with two solving approaches on the axis. Satisfiable instances are denoted by $\times$ and unsatisfiable instances by $\square$. The instances that timed out are plotted on the lines on the top and the right of the graphs, whereas the instances that ran out of memory despite the restart-forcing limitations are drawn at the edges of the graph. The dashed line in the figures correspond to the linear speedup.

5.2 Scalability of the Multi-core Implementation

Figure 1 shows scatter plots of the SPSAT approach, the iterative partitioning (*Iter-Part*) approach and the plain partitioning (*Plain-Part*) approach against the underlying solver. All three approaches are able to solve more formulas and are usually faster than MiniSAT 2.2.0. The SPSAT approach does not reach a linear speedup for unsatisfiable formulas, but works well for many of the satisfiable instances. The plain partitioning approach shows a noticeable slowdown for many of the instances where the run time is between hundred and thousand seconds. This could result from two factors; firstly, in multi-core computing the threads interfere between each other causing a slowdown. Secondly, as shown in Prop. 1, it is possible that the slowdown results from the shape of the distributions of the

[2] Solvers and data are available at `http://tools.computational-logic.org/`

original and the derived instances. Interestingly, the wall clock run time of plain partitioning for unsatisfiable instances is in 41 cases higher than that of the underlying solver, and lower in 69 cases (excluding instances not solved by both plain partitioning and the underlying solver). The corresponding numbers for iterative partitioning are substantially more convincing, 18 and 94.

As discussed in Sect. 3, the run time of a solver given a formula is essentially a random variable, and therefore a single run time pair on a single instance is inherently unreliable in comparing the performance of two algorithms. To estimate the quality of the results, we randomly selected ten unsatisfiable and ten satisfiable instances and repeated their solving with the iterative partitioning approach ten times. The average variation coefficient c_v, that is, the ratio of the standard deviation to the mean, itself averaged over the ten instances, is $c_v = 0.10$ for the unsatisfiable instances and $c_v = 0.31$ for the satisfiable instances.

5.3 Selecting a Scalable Algorithm

The use of more cores increases the memory access times and causes memory outs in the solvers as the data structures are replicated for each thread. A parallel solving approach should provide speed-up despite these adverse effects. Table 1 summarizes scalability using the instances that the approach solved both with four and 12 cores. In Fig. 2 we concentrate more on studying the run times of the unsatisfiable instances. The table distinguishes the results for satisfiable and unsatisfiable instances; the columns *slower* and *faster* denote the number of instances solved slower and faster, respectively, with 12 cores than with four cores. Hence if the number under *slower* is lower than the number under *faster*, this indicator shows that the approach scales. We also report the sum of the wall-clock run times for the approaches on the last four columns.

Based on the results we can make several interesting observations. Firstly, the wall-clock solving time for most instances in nearly all cases increases when the number of cores increases. The only exception is the iterative partitioning when solving satisfiable instances. Secondly, the total wall clock run time required to solve the instances decreases for almost all the approaches, here the exception being the SPSAT approach in unsatisfiable instances. The SPSAT approach scales

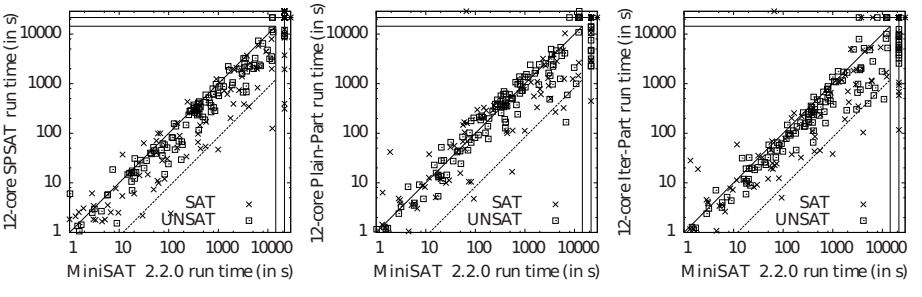

Fig. 1. The SPSAT, iterative and plain partitioning approaches with unit sharing

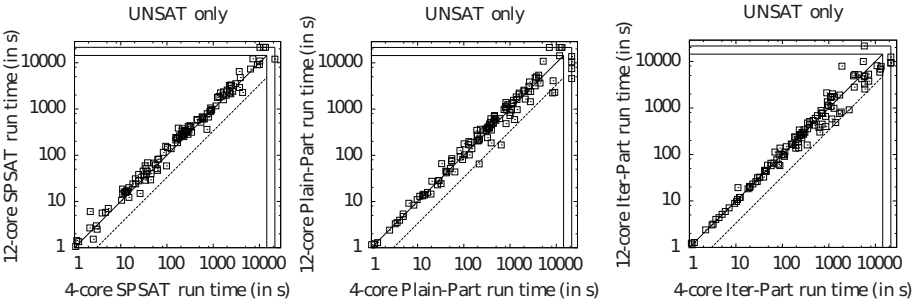

Fig. 2. The scalability of the parallel approaches

badly in unsatisfiable instances, while the plain and iterative partitioning approaches show better scalability, the iterative partitioning being clearly the best as it reduces the run time by 15% and almost never slows down the solving of an instance. As shown in Fig. 2, increasing the number of cores results in solving more instances in iterative partitioning, whereas in the other two approaches the number of solved instances either decreases or stays the same.

The scalability above suggest that the partitioning approaches scale better than SPSAT. Our three approaches are deliberately as simple as possible while still being interesting from the practical point of view, and hence none of the parallel solvers competing in the recent SAT competitions correspond exactly to any of the approaches. Nevertheless it is interesting to try to relate the observations here to the recent competitions. Of the four solvers competing in the 32 core track in 2011, three were variants of the SPSAT approach, one being implemented with the guiding path approach [23] related to the plain partitioning.

As can be seen in the comparisons in Figs. 1 and 2, the partitioning approaches are especially advantageous for the harder instances. We will next give some more insight into this. First, we show in left of Fig. 3 that the iterative partitioning approach compares favorably to the plain partitioning on unsatisfiable instances except for a handful of instances. The iterative partitioning approach also solves a significantly larger number of formulas than the plain partitioning approach. Again, there are two reasons for this. Firstly, in the light of the propositions 1 and 2 and results in [11], it is unlikely that the plain partitioning approach would obtain even close to linear speed-up. The iterative partitioning behaves analytically much

Table 1. Comparison on instances that the respective approaches could solve both with four and 12 cores. Column *slower* (resp. *faster*) denotes the number of instances solved slower (resp. faster) with 12 cores than with four cores.

	SAT		UNSAT		SAT runtime		UNSAT runtime	
Approach	slower	faster	slower	faster	4-core	12-core	4-core	12-core
SPSAT	**47**	36	**93**	23	61784	**57380**	**111152**	127462
Plain-Part	**45**	34	**77**	39	61681	**60934**	121432	**119925**
Iter-Part	33	**45**	**61**	58	53918	**50642**	153726	**131521**

Table 2. Instances not solved in SAT 2011 competition

Name	Solution	w/ preproc	w/o preproc
aes_32_4_keyfind_1	SAT	—	6299
gus-md5-12	UNSAT	4367	6022
rbcl_xits_09_UNKNOWN	UNSAT	—	9635
smtlib-qfbv-aigs-VS3-benchmark-S2-tseitin	UNSAT	4732	8163

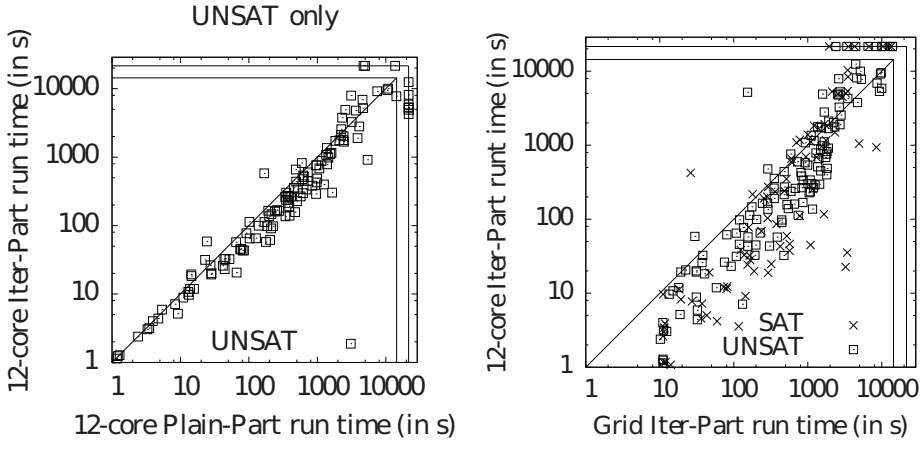

Fig. 3. Iterative partitioning in grid using at most 64 cores [22], and in multi-core approach using 12 cores

nicer [24]. Secondly, the iterative partitioning approach is able to adjust to the problem difficulty due to the dynamic construction of the partition tree.

We also give some insight into how the iterative partitioning approach scales beyond 12 cores in right of Fig. 3 by using a computing grid based implementation running on at most 64 cores. In this system the communication latencies are several orders of magnitude higher and the hardware is older than in the multi-core environment, rendering the results not directly comparable. We still note that increasing the number of cores helps in many hard unsatisfiable instances and results in solving roughly ten more instances.

Finally, we ran the iterative partitioning approach on the 2011 competition instances on 12 cores with and without the SatElite preprocessing techniques. The wall-clock run times are reported in Table 2. We only report the four instances that we could solve but were not solved by any solver in the competition.

6 Conclusions and Future Work

This work addresses some of the central questions in designing scalable parallel SAT solvers using a novel analysis based on a realistic model of search-space

partitioning and an efficient uniform implementation based on widely used techniques. The analysis shows that partitioning inherently involves a risk that the expected run time increases compared to sequential solving. An earlier result [24], showing that organizing the search spaces as a tree instead of a set avoids this problem, motivates the experimental comparison of these approaches as well as the widely used *portfolio* approach. Our results confirm that the partition tree based *iterative partitioning approach* performs well compared to the set-based *plain partitioning*, both of which perform better in the unsatisfiable formulas than the portfolio approach. Surprisingly, the iterative partitioning approach over-performs portfolio also in satisfiable formulas. Finally we demonstrate the performance of the iterative partitioning approach by solving four instances that could not be solved in the SAT competition 2011.

References

1. Cook, S.A.: The complexity of theorem-proving procedures. In: Proc. STOC 1971, pp. 151–158. ACM (1971)
2. Béjar, R., Manyà, F.: Solving the round robin problem using propositional logic. In: Proc. AAAI 2000, pp. 262–266. AAAI Press/MIT Press (2000)
3. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT Solving for Termination Analysis with Polynomial Interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
4. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Proc. ECAI 1992, pp. 359–363. John Wiley and Sons (1992)
5. Lynce, I., Marques-Silva, J.: SAT in Bioinformatics: Making the Case with Haplotype Inference. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 136–141. Springer, Heidelberg (2006)
6. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning SAT Instances for Distributed Solving. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 372–386. Springer, Heidelberg (2010)
7. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters 47(4), 173–180 (1993)
8. Rao, V.N., Kumar, V.: On the efficiency of parallel backtracking. IEEE Transactions on Parallel and Distributed Systems 4(4), 427–437 (1993)
9. Luby, M., Ertel, W.: Optimal Parallelization of Las Vegas Algorithms. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) STACS 1994. LNCS, vol. 775, pp. 463–474. Springer, Heidelberg (1994)
10. Segre, A.M., Forman, S.L., Resta, G., Wildenberg, A.: Nagging: A scalable fault-tolerant paradigm for distributed search. Artificial Intelligence 140(1/2), 71–106 (2002)
11. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning search spaces of a randomized search. Fundamenta Informaticae 107(2-3), 289–311 (2011)
12. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver — efficient workload balancing. Annals of Mathematics and Artificial Intelligence 17(3-4), 381–400 (1996)
13. Sinz, C., Blochinger, W., Küchlin, W.: PaSAT — parallel SAT-checking with lemma exchange: Implementation and applications. In: Proc. SAT 2001. Electronic Notes in Discrete Mathematics, vol. 9. Elsevier (2001)

14. Schubert, T., Lewis, M., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. Journal on Satisfiability, Boolean Modeling and Computation 6(4), 203–222 (2009)
15. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Proc. HVC 2001. Springer (2011) (to appear)
16. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
17. Hamadi, Y., Jabbour, S., Saïs, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 6(4), 245–262 (2009)
18. Hamadi, Y., Jabbour, S., Saïs, L.: Control-based clause sharing in parallel SAT solving. In: Proc. IJCAI 2009, pp. 499–504. IJCAI/AAAI (2009)
19. Guo, L., Hamadi, Y., Jabbour, S., Sais, L.: Diversification and Intensification in Parallel SAT Solving. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 252–265. Springer, Heidelberg (2010)
20. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Incorporating clause learning in grid-based randomized SAT solving. Journal on Satisfiability, Boolean Modeling and Computation 6, 223–244 (2009)
21. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: A Distribution Method for Solving SAT in Grids. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 430–435. Springer, Heidelberg (2006)
22. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 385–399. Springer, Heidelberg (2011)
23. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21(4-6), 543–560 (1996)
24. Hyvärinen, A.E.J.: Grid Based Propositional Satisfiability Solving. PhD thesis, Aalto University (2011)
25. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001, pp. 530–535. ACM (2001)
26. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)