Grid-Based SAT Solving with Iterative Partitioning and Clause Learning

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

Aalto University,
Department of Information and Computer Science,
PO Box 15400, FI-00076 AALTO, Finland
{Antti.Hyvarinen,Tommi.Junttila,Ilkka.Niemela}@aalto.fi

Abstract. This work studies the solving of challenging SAT problem instances in distributed computing environments that have massive amounts of parallel resources but place limits on individual computations. We present an abstract framework which extends a previously presented iterative partitioning approach with clause learning, a key technique applied in modern SAT solvers. In addition we present two techniques that alter the clause learning of modern SAT solvers to fit the framework. An implementation of the proposed framework is then analyzed experimentally using a well-known set of benchmark instances. The results are very encouraging. For example, the implementation is able to solve challenging SAT instances not solvable in reasonable time by state-of-the-art sequential and parallel SAT solvers.

1 Introduction

This work studies the solving of hard instances of the propositional satisfiability problem (SAT) using a massively parallel master-worker environment such as a grid or a cloud where several clusters are scattered around a large geographical area. Grids and clouds typically provide large amounts of computing power at a relatively low cost making them increasingly appealing for users.

This work considers a grid computing model where each worker executes a job for a limited amount of time and can communicate the results only to the master. The run time limits are typically quite low, in this work approximately one hour. Jobs with modest computing requirements are in many ways beneficial in practice. For example, a job requiring a single CPU core for a relatively short time can often be scheduled to a time slot unsuitable for jobs requiring several CPUs for an extended time period. Furthermore, should a job fail, e.g., due to a service break in a cluster, the cost of recovering from the failure is at most the duration of the job.

Most approaches to parallel SAT solving fall into the following two categories:

- In the *portfolio* approach the speed-up results from running slightly varied solvers with the same input simultaneously and obtaining the result from the first finishing solver (see, e.g., [10]). The idea generalizes to many related algorithms [18,12].
- In the *guiding path* approach the instance is constrained to several solution disjoint subproblems solved in parallel, usually aided with load balancing for dealing with unequally sized subproblems [25,27,22].

J. Lee (Ed.): CP 2011, LNCS 6876, pp. 385–399, 2011.
© Springer-Verlag Berlin Heidelberg 2011

It is not straightforward to implement approaches from either of the two categories in an environment which limits job run times, as they naturally assume unlimited run times.

This work discusses an iterative partitioning approach which scales to thousands of jobs. Even current grid middlewares with relatively high latencies allow us to run tens of jobs simultaneously. The approach was introduced in [13] and further developed in [15]. A job for solving an instance is first submitted to the parallel environment, and at the same time constrained to several solution disjoint formulas. These *derived formulas* are then also submitted and the constraining applied iteratively to each derived formula resulting in a recursively constructed *partition tree*. During the solving process the solvers learn clauses that are used to prune the search space of the instance they are solving [20]. This work extends [15] by taking the clauses produced by a timed out job and integrating them to the partition tree to constrain the search spaces of the subsequent jobs. The challenge in this is that the learned clauses may depend on the partitioning constraints. As a solution to this problem we present two ways of determining how the learned clauses depend on the constraints.

The resulting implementation is able to solve several challenging SAT instances, some of which were not solved in the SAT Competition 2009 (see `http://www.satcompetition.org/`).

The learning partition tree approach compares favorably to several state-of-the-art parallel shared memory SAT solvers as well as to grid-based approaches.

1.1 Related Work

The iterative partitioning approach differs from the portfolio approach in that the derived formulas become increasingly constrained and hopefully easier to solve deeper in the partition tree. The approach is closely related to divide-and-conquer approaches such as the guiding path. However, in the iterative partitioning approach the search is organized redundantly (see Sect. 3). While this might seem counter-intuitive, it has proved to be a surprisingly good strategy; for example, it can be shown that the redundancy in solving can help prevent some anomalies related to increasing expected run times in unsatisfiable formulas [16]. The constraints used for producing derived formulas in this work are not limited to unit clauses, but can instead be arbitrarily long formulas. We are not aware of guiding path implementations that would use such constraints. Most literature, to our knowledge, assumes unbounded run times for jobs, while in [13,14,15] the current authors have studied SAT solving in environments where maximum run time of a job is much lower than typical time required to solve an instance.

The guiding path type constraining dates at least back to [25,4]. Much work has been invested in finding "good" guiding paths (see, e.g. [6], as an improper construction results in the worst case in increased expected run times [16]. The partition tree approach followed in this work attempts to solve both a formula and all its derived formulas in parallel being therefore immune to the increase. The partitioning approaches used in this work are described in [13] and [15]. Similar ideas based on running the VSIDS heuristic [23] to produce good constraints are discussed in [21].

Guiding path based parallel SAT solvers for distributed computing environments have been implemented both without clause learning [17,27] and with different strategies for sharing the learned clauses [3,24,5]. The approach discussed in this work differs

from these by limiting the run time of jobs, by using the iterative partitioning approach for the basis of parallelism and by using a more general approach to constructing the derived formulas. In particular the latter strengthens the clauses as they need not be logical consequences of the original instance. Although this makes clause sharing tedious to implement efficiently, the approach performs well in experiments.

Algorithm portfolio based parallel solvers with clause sharing (see, e.g, [9,2]) work surprisingly well in shared memory environments. They have been adapted also to grid environments by the current authors as the CL-SDSAT framework [14] while the learning partition tree approach discussed in this work can be seen as an extension of CL-SDSAT.

2 Preliminaries

We assume the standard notations of satisfiability and basic knowledge conflict-driven clause learning (CDCL) SAT solvers (see e.g. [19]). Let V be a set of Boolean variables. The set $\{x, \neg x \mid x \in V\}$ is the set of *literals*, a *clause* is a set (disjunction) of literals and formula a conjunction of clauses. A formula is satisfiable if there is a set of literals τ such that for no $x \in V$ both $x, \neg x \in \tau$ and each clause contains a literal from τ. Such a τ is called a satisfying truth assignment, and in case one does not exist, the formula is unsatisfiable. Let ϕ and ψ be formulas. If all satisfying truth assignments of ψ satisfy ϕ, then ϕ is a *logical consequence* of ψ, denoted $\psi \models \phi$. If ψ and ϕ are satisfied by exactly the same truth assignments, they are logically equivalent and we write $\psi \equiv \phi$.

Many of the ideas in this work are based on the fact that a CDCL SAT solver, while solving a formula ϕ, produces during the search huge amounts of clauses C such that $\phi \models C$. The clauses are used in guiding the further search so that in general the size of the search space decreases. Typically one clause is learned each time the solver finds that a truth assignment does not satisfy the formula. Of particular interest are learned clauses containing a single literal l. Since l is true in any satisfying truth assignment of ϕ, its negation $\neg l$ can be removed from all clauses, resulting not only in smaller remaining search space but also in shorter clauses, decreased memory foot print and better cache performance.

3 Iterative Partitioning with Clause Learning

In this section, we first review and formalize the partition tree approach used, e.g., in [13]. We then extend the approach to allow the use of learned clauses, i.e., clauses that are logical consequences of the original formula or of the iteratively partitioned *derived formulas* in the partition tree, and discuss how they can be found and maintained in practice. Two implementation approaches that can make a modern CDCL SAT solver to produce such learned clauses are then described in the next section.

3.1 Partition Trees

The idea in the iterative partitioning approach is to construct a partition tree of formulas rooted at ϕ such that the satisfiability of ϕ can be deduced once a sufficient amount of the formulas in the tree have been solved.

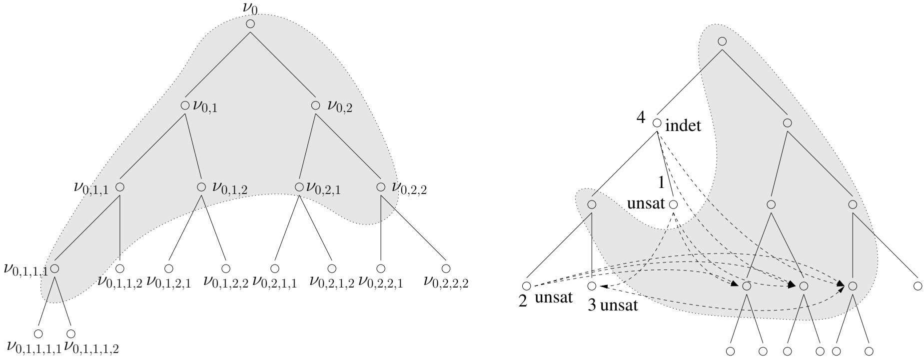

Fig. 1. The Iterative Partitioning approach. Nodes represent derived formulas, and the nodes in the shaded area are being solved simultaneously. Terminated jobs are marked either indet or unsat depending on whether they run out of resources or prove unsatisfiability, and annotated with the termination order (1 terminates first and 4 last). Some learned clauses from earlier terminated jobs can be transferred to the newly submitted jobs, illustrated by the dashed arrows. The tree is constructed in breadth-first order.

Given a formula ψ, the partitioning function computes the set of partitioning constraints $\mathcal{P}(\psi) = \{\Pi_1, \ldots, \Pi_n\}$ that, when conjoined with ψ, result in the formulas $\psi_i = \psi \wedge \Pi_i$ such that (i) $\psi \equiv \psi_1 \vee \ldots \vee \psi_n$, and (ii) $\psi_i \wedge \psi_j$ is unsatisfiable whenever $i \neq j$. A trivial way to get a partitioning function would be to select two variables a and b occurring in the formula ψ, and letting $\mathcal{P}(\psi) = \{a \wedge b, a \wedge \neg b, \neg a \wedge b, \neg a \wedge \neg b\}$.

In the following discussion we make a distinction between the nodes of the tree and the formulas representing the nodes. Formally, the partition tree $\mathcal{T}_\psi$ of a formula ψ is a rooted finite n-ary tree with the set of nodes $\mathcal{N}$. Each node $\nu \in \mathcal{N}$ is labeled with the partitioning constraint $constr(\nu)$ as follows:

1. The root node ν_0 is constrained with $constr(\nu_0) = \mathbf{true}$ (i.e. the empty conjunction).
2. Let ν_k be a node in the tree, $\nu_0\nu_1 \ldots \nu_k$ the path from the root node ν_0 to ν_k, and $\nu_{k,1}, \ldots, \nu_{k,n}$ the child nodes of ν_k. The constraints $constr(\nu_{k,i}) = \Pi_i$ are then obtained by computing the set $\mathcal{P}(\psi \wedge constr(\nu_0) \wedge \ldots \wedge constr(\nu_k)) = \{\Pi_1, \ldots, \Pi_n\}$.

Given a node ν_k, the formula $form(\nu_k) = \psi \wedge constr(\nu_0) \wedge \ldots \wedge constr(\nu_k)$ is the *derived formula at* ν_k. Based on the properties of partitioning functions it is evident that (i) if a derived formula is satisfiable, then so is the original formula ψ, and (ii) if the leaves of a sub-tree rooted at ψ are all unsatisfiable, then so is the formula ψ.

Example 1. Figure 1 illustrates how the partition tree is constructed on-the-fly in breadth-first order starting from the root using eight CPU cores in a grid and when the partition factor $n = 2$. In the left tree the derived formulas at nodes are sent to the environment to be solved (in parallel) with a SAT solver, and the nodes are further partitioned into child nodes at the same time. The derived formula at the root node ν_0 (i.e., the original formula ψ) is first sent to be solved in the environment; while it is being solved, the root

ν_0 is partitioned into nodes $\nu_{0,1}, \nu_{0,2}$ and these are sent to be solved in the environment; then partitioning is applied to $\nu_{0,1}$, $\nu_{0,2}$ and so on in the similar breadth-first manner, until finally the derived formulas in the shaded nodes are running simultaneously. If the derived formula at a node were found satisfiable, then the original formula would be declared satisfiable and the process would end. In the right tree, the nodes $\nu_{0,1,2}$ and $\nu_{0,1,1,1}$ are first solved, and shown unsatisfiable. We therefore know that all derived instances below these are unsatisfiable, and are therefore not submitted. Instead the nodes $\nu_{0,1,1,2}$ and $\nu_{0,2,1,1}$ are submitted to the newly freed cores. Later the node $\nu_{0,1,1,2}$ is shown unsatisfiable. We could now finish the solving of $\nu_{0,1,1}$ since we know it unsatisfiable. The solving is not terminated in the example as the clauses learned there might still prove useful in other parts of the partition tree, and instead the next node $\nu_{0,2,1,2}$ is submitted. Finally the node $\nu_{0,1}$ times out and the derived formula in node $\nu_{0,2,2,1}$ is submitted for solving.

3.2 Adding Clause Learning

Since its introduction and popularization in *Grasp* [20] and *zChaff* [23] solvers, *conflict driven clause learning* has been a major search space pruning technique applied in sequential SAT solvers. Basically, at each conflict reached during the search the SAT solver adds a new *learned clause* C that (i) is a logical consequence of the formula ϕ under consideration, and (ii) prevents similar conflicts from happening in future search. In this paper, our main goal is to exploit such learned clauses produced during *solving one derived formula* when *solving other derived formulas* in a partition tree. As a learned clause produced when solving a derived formula may depend on the partitioning constraints, it is not necessarily a logical consequence of some other derived formulas and cannot thus be used when solving those. We first give a very abstract framework of partitioning trees where arbitrary logical consequences can be incorporated and then discuss the current realization of the framework based on using learned clauses derived during the search tree construction.

Assume a node ν_k in the tree with the associated derived formula $\mathit{form}(\nu_k) = \psi \wedge \mathit{constr}(\nu_0) \wedge ... \wedge \mathit{constr}(\nu_k)$. A formula $\mathit{form}(\nu_k)'$ is a *simulating derived formula*, denoted by $\mathit{form}(\nu_k)' \sim \mathit{form}(\nu_k)$, if it is of form $\psi' \wedge \mathit{constr}(\nu_0) \wedge \Sigma(\nu_0) \wedge \mathit{constr}(\nu_1) \wedge \Sigma(\nu_1) \wedge ... \wedge \mathit{constr}(\nu_k) \wedge \Sigma(\nu_k)$ such that (i) $\psi' \equiv \psi$, and (ii) for each $0 \leq i \leq k$, $\psi \wedge \mathit{constr}(\nu_0) \wedge ... \wedge \mathit{constr}(\nu_i) \models \Sigma(\nu_i)$. That is, (i) the original formula ψ may be substituted with an equivalent one (in practice: simplified with additional information obtained during the tree construction) and (ii) "learned clauses" $\Sigma(\nu_i)$ can be added as long as they are logical consequences of the corresponding partitioning constraints. Now the second rule in the definition of the partition tree $\mathcal{T}_\psi$ can be replaced with

2'. Let ν_k be a node in the tree, $\nu_0\nu_1...\nu_k$ the path from the root node ν_0 to ν_k, and $\nu_{k,1}, \ldots, \nu_{k,n}$ the child nodes of ν_k. The constraints $\mathit{constr}(\nu_{k,i}) = \Pi_i$ are then obtained by computing the set $\mathcal{P}(\psi' \wedge \Sigma(\nu_0) \wedge \mathit{constr}(\nu_1) \wedge \Sigma(\nu_1) \wedge \ldots \wedge \mathit{constr}(\nu_k) \wedge \Sigma(\nu_k)) = \{\Pi_1, \ldots, \Pi_n\}$

Similarly, any simulating formula $\mathit{form}(\nu)' \sim \mathit{form}(\nu)$ at a node ν can be sent to be solved in the distributed computing environment instead of $\mathit{form}(\nu)$. Due to the

definition of simulating derived instances, the same construction algorithms and termination criteria can be applied as in the base case of partition trees without learning.

Let S be a CDCL solver. If a simulating derived formula $\phi = form(\nu_k)'$ is found satisfiable by S, then the original formula ψ is also satisfiable and the construction of the partition tree can be terminated. However, if S found ϕ unsatisfiable or S could not solve ϕ within the imposed resource limits, we would like to obtain new learned clauses to help in solving other nodes in the partitioning tree. That is, we would like S to produce new sets Σ'_i of clauses such that for each $0 \leq i \leq k$, $\psi \wedge constr(\nu_1) \wedge \ldots \wedge constr(\nu_i) \models \Sigma'_i$. Each new clause set Σ'_i can be used when solving any node having ν_i as its ancestor, since the constraints of ν_i are a subset of the constraints of its descendant by the rule 2'. Naturally, of particular interest are the *partitioning constraint independent learned clauses* Σ'_0 that can be used when solving any node in the tree; the transfer of these clauses is illustrated by the dashed arrows in the right tree of Fig. 1. Two techniques for obtaining such clauses are discussed in the next section.

In the experiments discussed in this work, we currently use two schemes for maintaining the sets of learned clauses. Firstly, we maintain a database of partitioning constraint independent learned clauses; when such new learned clauses are obtained from a job, they are inserted into the database. To limit the size of the database, only a fixed amount of clauses are kept in it; currently we prefer to keep the shortest learned clauses found. The found unit learned clauses are used to simplify the database. Secondly, we maintain for each node ν_k a limited set $\Sigma(\nu_k)$ of learned clauses specific to that node (i.e. $form(\nu_k) \models \Sigma(\nu_k)$). For the sake of space efficiency, these sets currently contain only unary learned clauses.

3.3 Partitioning Functions

This work considers two partitioning functions studied earlier in [15]. Both approaches take as input the formula $form(\nu_k)'$ and produce n partitioning constraints. The first, called *vsids* in this work, is based on running a SAT solver with the VSIDS branching heuristic [23] for a fixed amount of time (5 minutes in the experiments) and using the obtained heuristic values to pick literals l^i_j to construct the partitioning constraints

$$\Pi_i = \begin{cases} (l^1_1) \wedge \ldots \wedge (l^1_{d_1}) & \text{if } i = 1, \\ (\neg l^1_1 \vee \ldots \vee \neg l^1_{d_1}) \wedge \ldots \wedge (\neg l^{i-1}_1 \vee \ldots \vee \neg l^{i-1}_{d_{i-1}}) \wedge & \\ \quad (l^i_1) \wedge \ldots \wedge (l^i_{d_i}) & \text{if } 1 < i < n, \\ (\neg l^1_1 \vee \ldots \vee \neg l^1_{d_1}) \wedge \ldots \wedge (\neg l^{n-1}_1 \vee \ldots \vee \neg l^{n-1}_{d_{n-1}}) & \text{if } i = n. \end{cases}$$

The resulting constraints are not necessarily sets of unit clauses, but instead might contain clauses of length d_i. The value of d_i is selected so that the partitions have equal search spaces (see [15] for details).

The second partitioning function is based on the *unit propagation lookahead* [11], where the idea is to always branch on the most propagating literal. The *lookahead* partitioning function is analyzed further in [15], while in this work we chose not to produce the disjunctions as above, but instead to use constraints of unit clauses.

4 Learned Clause Tagging CDCL Solvers

We now study the problem of determining the "constraint dependency" of new learned clauses as discussed above. Assume a partition tree node ν_k with a simulating derived formula $\mathit{form}(\nu_k)' = \psi \wedge \Sigma(\nu_0) \wedge \mathit{constr}(\nu_1) \wedge \Sigma(\nu_1) \wedge \ldots \wedge \mathit{constr}(\nu_k) \wedge \Sigma(\nu_k)$, sent to the grid to be solved with a CDCL solver S, and that the solving terminates either due to exhausting the resource limits or to conclusion that $\mathit{form}(\nu_k)'$ is unsatisfiable. To further exploit the work done by the solver S, we would like the solver to give new learned clauses to help in solving other nodes in the partition tree. That is, we would like S to produce new sets $\Sigma(\nu_i)'$ of clauses such that $\psi \models \Sigma(\nu_0)'$ and, for each $1 \leq i \leq k$, $\psi \wedge \mathit{constr}(\nu_1) \wedge \ldots \wedge \mathit{constr}(\nu_i) \models \Sigma(\nu_i)'$. A clause set $\Sigma(\nu_i)'$ can always be used when solving any descendant node of ν_i; we are therefore particularly interested in the partitioning constraint independent learned clauses $\Sigma(\nu_0)'$ that can be used when solving any node in the tree.

In this section we describe two techniques that can be used to "tag" the learned clauses produced by a CDCL solver so that they can be classified as either partitioning constraint independent (i.e. belong to $\Sigma(\nu_0)'$) or belonging to a set $\Sigma(\nu_i)'$ for some $1 \leq i \leq k$.

4.1 Assumption-Based Learned Clause Tagging

Our first, more fine grained clause tagging technique uses the concept of *assumption variables* for tagging partition constraints. Each constraint Π is annotated with a newly introduced variable a with the conjunction $\Pi \vee \neg a$. The constraint Π is then enabled by setting a true. When learned clauses are deduced by the CDCL solver during its search, these special literals are "inherited" in learned clauses, thus also tagging which partition constraints the newly derived learned clause depended on. The assumption variable technique was introduced in [7] for dynamically adding and removing clauses in a formula between subsequent satisfiability tests in the context of bounded model checking; it has also been used in a folklore method for unsatisfiability core extraction [1].

Given a simulating derived formula $\mathit{form}(\nu_k)' = \psi' \wedge \Sigma(\nu_0) \wedge \mathit{constr}(\nu_1) \wedge \Sigma(\nu_1) \wedge \ldots \wedge \mathit{constr}(\nu_k) \wedge \Sigma(\nu_k)$, the idea in the assumption based tagging is that the CDCL solver considers the straightforward CNF translation of the formula $\mathit{form}(\nu_k)^\star = \psi' \wedge \Sigma(\nu_0) \wedge (\neg a_1 \vee (\mathit{constr}(\nu_1) \wedge \Sigma(\nu_1))) \wedge \ldots \wedge (\neg a_k \vee (\mathit{constr}(\nu_k) \wedge \Sigma(\nu_k)))$ instead, where $a_1, \ldots, a_k$ are k disjoint *assumption variables* not occurring in $\mathit{form}(\nu_k)'$. Thus, for each clause $C = (l_1 \vee \ldots \vee l_m)$ in a "constraint subformula" $\mathit{constr}(\nu_i) \wedge \Sigma(\nu_i)$ in $\mathit{form}(\nu_k)'$, there is a corresponding "a_i-triggered clause" $(\neg a_i \vee l_1 \vee \ldots \vee l_m)$ in $\mathit{form}(\nu_k)^\star$. Obviously, both $\mathit{form}(\nu_k)'$ and $\mathit{form}(\nu_k)$ are satisfiable if and only if $\mathit{form}(\nu_k)^\star \wedge (a_1) \wedge \ldots \wedge (a_k)$ is.

To deduce whether $\mathit{form}(\nu_k)^\star \wedge (a_1) \wedge \ldots \wedge (a_k)$ is satisfiable (and thus whether $\mathit{form}(\nu_k)$ is), the CDCL solver is now invoked on $\mathit{form}(\nu_k)^\star$ with a list of "assumptions" $[a_1, \ldots, a_k]$. That is, its branching heuristic is forced to always branch on these variables first and to make the assumption that their values are true; these assumptions activate the "a_i-triggered clauses". After the assumptions the search continues as usual in a CDCL solver; the beauty of this technique is that when a learned clause is deduced

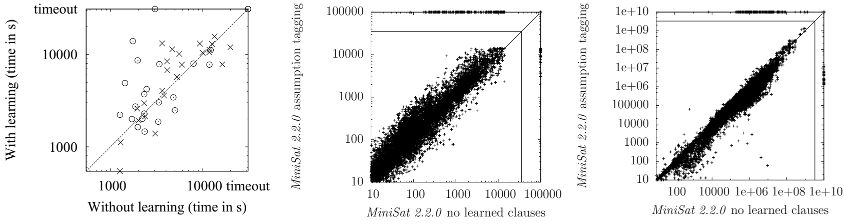

Fig. 2. The effect of learning in partition trees with assumption based tagging

during the search because a conflict was encountered, then the learned clause will include the literal $\neg a_i$ iff the deduction of the clause depended on any clause in the derived formula $constr(\nu_i) \wedge \Sigma(\nu_i)$ (this follows from [7]). Therefore, recalling that $\psi' \equiv \psi$ and $\psi' \wedge constr(\nu_1) \wedge \ldots \wedge constr(\nu_i) \models \Sigma(\nu_i)$ for each $1 \leq i \leq k$, we get for each learned clause $C = (\neg a_1 \vee \ldots \vee \neg a_j \vee l_1 \vee \ldots \vee l_m)$ where a_j is the assumption variable with the largest index occurring in C, that $\psi' \wedge \bigwedge_{0 \leq i \leq j} constr(\nu_i) \models (l_1 \vee \ldots \vee l_m)$. Thus the learned clause $(l_1 \vee \ldots \vee l_m)$ can be used in the learned clause set $\Sigma(\nu_j)$ in the simulating derived formula $form(\nu_n)$ of any child node of ν_j. The important special case is when $j = 0$ and thus $(l_1 \vee \ldots \vee l_m)$ can be used in the partitioning constraint independent learned clause set $\Sigma(\nu_0)$ in the simulating derived formula $form(\nu)$ of *any node* ν in the partition tree.

In addition to learned clause tagging, the assumption variables can also be used to deduce an *unsatisfiability level* $0 \leq U \leq k$ such that, if $form(\nu_k)^\star \wedge (a_1) \wedge \ldots \wedge (a_k)$ is unsatisfiable, then so is $\psi' \wedge \bigwedge_{1 \leq i \leq U} constr(\nu_i)$. This is because at the end of the search, when the solver deduces that not all the assumptions can be true at the same time (and thus $form(\nu_k)^\star \wedge (a_1) \wedge \ldots \wedge (a_k)$ is unsatisfiable), it can invoke a special form of conflict analysis that deduces on which assumptions this "final conflict" depended on; this is implemented, e.g., in the current version 2.2.0 of MiniSat solver. Thus if $U < k$, we know that already the derived formula $form(\nu_U)$ of the ancestor node ν_U of ν_k is unsatisfiable and can *backjump* to ν_{U-1} (or report that the original formula ψ is unsatisfiable if $U = 0$) and skip all the other children of ν_U. Unfortunately, our preliminary experiments show that such backjumping rarely occurs in real life benchmarks when a non-naive partitioning function is applied; it seems that the partitioning constraints imposed by the function are almost never totally irrelevant for the unsatisfiability proof found by the solver for $form(\nu_k)^\star \wedge (a_1) \wedge \ldots \wedge (a_k)$ (and thus for $form(\nu_k)'$). Thus, and due to space limits, we do not analyze this tree backjumping technique further in this paper.

In Fig. 2 we analyze the assumption-based tagging approach using 36 application category instances from SATCOMP-2009. The instances are selected so that most of them are challenging for modern SAT solvers; 19 of them were not solved by any solver in the competition. We attempted solving of all the instances with the partition tree approach both with and without learning, constructing the subproblems with *lookahead* and *vsids* partitioning functions. The leftmost figure is a scatter plot comparing the learning and non-learning partition tree approaches, where each point represents

an instance solved either with *lookahead* or *vsids* (marked $\circ$ and $\times$, respectively). Based on the results the learning, in fact, slows down the solving process compared to the approach without learning. The two rightmost figures illustrate the reason for this slow-down. Each point in the figures represents one job that was constructed while running the learning partition tree approach in the leftmost figure. Note that a single point in the leftmost figure might correspond to thousands of such points. The middle figure shows the run time of the jobs both with and without the learned clauses. The vertical axis is the run time with learned clauses solved with an assumption-based learned clause tagging solver, whereas the horizontal axis is the run time without learned clauses solved with an unaltered solver. Hence if a point is above the diagonal in the figure, the overhead caused by learning is not compensated by the reduction of the search space by the learned clauses. In particular it is interesting to note that the number of failures, shown as dots on the edges of the graph, is much higher when learning is used. Most of these result from memory exhaustion. The rightmost figure shows that the number of decisions made in the jobs decreases with learning. We may draw the conclusion that the solver with assumption based tagging consumes significantly more memory than the unaltered solver. This perhaps surprising result can be explained as follows. The conflict clauses deduced during the search can, for some real life formulas, contain large amounts of literals that (i) are implied by unit propagation after the assumption variables have been set to true, but (ii) are not (or have not been found to be by the solver) logical consequences of $form(\nu_k)^\star = \psi' \wedge \Sigma(\nu_0) \wedge (\neg a_1 \vee (constr(\nu_1) \wedge \Sigma(\nu_1))) \wedge \ldots \wedge (\neg a_k \vee (constr(\nu_k) \wedge \Sigma(\nu_k)))$. If the CDCL solver would consider $form(\nu_k)' = \psi' \wedge \Sigma(\nu_0) \wedge constr(\nu_1) \wedge \Sigma(\nu_1) \wedge \ldots \wedge constr(\nu_k) \wedge \Sigma(\nu_k)$ as a "flat formula" without assumption variables instead, such implied literals would not be included in conflict clauses as they are logical consequences of $form(\nu_k)'$. Such long clauses can consume excessive amounts of memory and also slow down the solver. We will shortly return to this perhaps surprising result in Example 2. The phenomenon has not, to our knowledge, been reported previously in parallel SAT solving, and we believe it plays a role also in the approaches based purely on guiding paths.

4.2 Flag-Based Learned Clause Tagging

To overcome the previously described challenge in assumption-based learned clause tagging, we describe here a light-weight version for clause learning similar to the one used in [26]. The intuition is to over-approximate the dependency of a learned clause from the constraints by flagging clauses which potentially depend on the assumptions.

Given a node ν_k with a simulating derived formula $form(\nu_k)' = \psi' \wedge \Sigma(\nu_0) \wedge constr(\nu_1) \wedge \Sigma(\nu_1) \wedge \ldots \wedge constr(\nu_k) \wedge \Sigma(\nu_k)$, our second clause tagging technique executes a CDCL solver "as is" on the formula except that in the beginning it marks the clauses in $\phi \wedge \Sigma(\nu_0)$ as "safe". Whenever a new learned clause (including learned unit clauses which are expressed as new "decision level 0" implied literals inside the solver) is derived, it is also marked "safe" if its derivation depended only on "safe" clauses. As a consequence, and recalling $\phi \models \Sigma(\nu_0)$, learned clauses marked "safe" are logical consequences of ϕ and can thus be included in the constraint-independent learned clause set $\Sigma(\nu_0)$ in any other node μ_m of the partition tree. The learned clauses not marked "safe" may depend on $constr(\nu_1) \wedge \Sigma(\nu_1) \wedge \ldots \wedge constr(\nu_k) \wedge \Sigma(\nu_k)$ and are thus

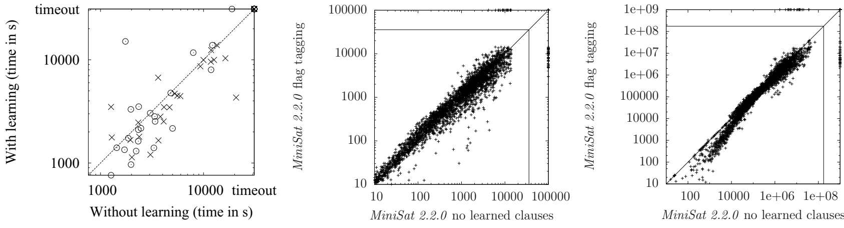

Fig. 3. The effect of learning in partition trees with flag based tagging

only guaranteed to be logical consequences of $\phi \wedge \mathit{constr}(\nu_0) \wedge \ldots \wedge \mathit{constr}(\nu_k)$; they can only be included in the constraint-dependent clause set $\Sigma(\nu_k)$ when considering any descendant node ν_l of ν_k.

This technique has the advantage of solving the above discussed "long clause problem" and adding only a very minimal overhead on the CDCL solver but, as shown below, the disadvantage that it can produce fewer constraint-independent learned clauses than the assumption based technique.

Figure 3 illustrates the effect of using flag-based learned clause tagging to the run time of the partition tree approach and to each job. We first note that based on the results in the leftmost graph using learned clauses seems to provide speed-up to the solving in most of the instances from the benchmark set. The per-job results in the two rightmost figures show that the previously observed failures are roughly equally common both with the flag-based solver with learned clauses and the unaltered solver without learned clauses. In particular the more difficult instances seem to be solved faster with the learned clauses (middle figure). The effect is also seen in the number of decisions (rightmost figure).

To shortly illustrate the two tagging techniques and to see that the assumption based one can produce more constraint-independent, although longer, learned clauses, consider the following simple example.

Example 2. Let the original formula ϕ include the clauses $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)$ and let $\mathit{constr}(\nu_1) = (\neg x_1)$ while $\Sigma(\nu_0) = \Sigma(\nu_1) = \emptyset$.

In the assumption based tagging, the solver will start with the formula ϕ extended with the assumption-encoded clause $(a_1 \vee \neg x_1)$. Making first the assumption branch on $\neg a_1$ and then the non-assumption branch on $\neg x_2$, the solver will learn the constraint-independent learned clause $(x_1 \vee x_2)$; note that $\neg x_1$ is not (necessarily) a logical consequence of ϕ and thus $(x_1 \vee x_2)$ is not simplified to x_2 by resolving on $\neg x_1$.

When using flag based tagging, the solver starts with an instance having the clauses $(x_1 \vee x_2 \vee x_3)^{\mathsf{s}} \wedge (x_2 \vee \neg x_3)^{\mathsf{s}} \wedge (\neg x_1)$ where the superscript s denotes a "safe" clause. Similarly branching on $\neg x_2$ results in a non-safe (and thus, correctly, constraint-dependent) learned clause (x_2); here $\neg x_1$ is trivially a logical consequence of the input formula $\phi \wedge \mathit{constr}(\nu_1)$ and thus $(x_1 \vee x_2)$ is simplified to x_2 by resolving on the non-safe unit clause $\neg x_1$.

5 The Main Experimental Results

This section compares our implementation of the iterative partitioning approach with clause learning to some other SAT solver implementations using all the 292 SATCOMP-2009 application benchmarks. These main experiments use the flag based learned clause tagging with the vsids heuristic, as it seems to perform slightly better than the lookahead heuristic (see the leftmost graph in Fig. 3). The remaining parameters, discussed in this section, are not particularly tuned for the benchmark set. They can be seen as reasonable guesses, but a closer study would likely reveal better values for these instances. The comparison uses the implementations below.

- *MiniSat 2.2.0*, a sequential SAT solver we have used as a basis for the grid-based approaches.
- *Part-Tree*, a grid-based iterative partitioning implementation without clause sharing [15].
- *Part-Tree-Learn*, a grid-based implementation of the learning iterative partitioning approach described in this work.
- *Cl-Sdsat*, a grid-based portfolio approach [14]. Learned clauses are collected from the timed-out jobs and used in subsequent jobs. Underlying solver is *MiniSat 2.2.0*.
- *ManySat 1.1* and *ManySat 1.5*, multi-core portfolio solvers using 4 cores [9].
- *Plingeling* 276, a multi-core portfolio solver which won the SAT-Race 2010. The results reported in this work are obtained with 12 cores [2].

The *Cl-Sdsat*, *Part-Tree*, and *Part-Tree-Learn* approaches use the M-grid environment currently consisting of nine clusters with CPUs purchased between 2006 and 2009. For more detailed information, see `http://wiki.hip.fi/gm-fi/`. The learned clauses transferred with the jobs are limited so that they contain in total 100 000 literals in these approaches. The other implementations were run using twelve-core AMD Opteron 2435 nodes from the year 2009.

As discussed before, each grid-based approach consists of a work flow of several relatively short lived jobs. The jobs enter the clusters through a batch queue system after a varying queuing time d_{queuing} which is affected, for example, by the background load of the grid. Figure 4 shows the cumulative distribution of the queuing time measured over 200 000 jobs between 2010 and 2011; time t is shown on the horizontal axis, and the vertical axis gives the probability that the queuing time d_{queuing} is at most t; median queuing time is approximately two minutes. The queuing time is included in the reported run times of the grid based approaches. Actual parallelism in the grid experiments depends on the queuing time and on the time required to construct the jobs, which in these experiments is at most 43 seconds per derived formula. As a result, the amount of parallelism varies typically between 8 and 60 simultaneously running jobs.

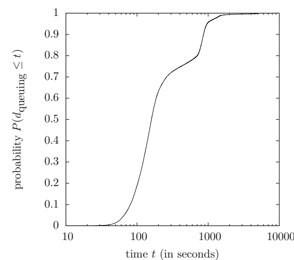

Fig. 4. Cumulative queuing time distribution in M-grid

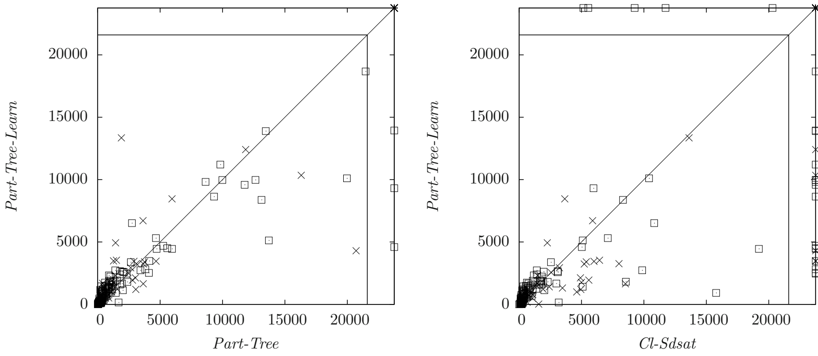

Fig. 5. Comparing *Part-Tree-Learn* to *Part-Tree* (left), and *Cl-Sdsat* (right). Satisfiable instances are marked with crosses (×) and unsatisfiable with boxes (□).

5.1 Comparing the Grid-Based Approaches

We first compare the grid-based *Part-Tree* and *Part-Tree-Learn* implementations in left of Fig. 5. The crosses (×) denote satisfiable and boxes (□) unsatisfiable instances from the SATCOMP-2009 benchmarks, and a mark below the diagonal means that *Part-Tree-Learn* performed better on that instance. An instance not solved in 6 hours is considered timed out. The 6 hour limit is marked on the graphs with the two inner lines on top and on the right of the graphs, and timed out instances are placed on the edges of the graph. The results suggest that learning may slow down the solving of easy instances but, as the run time of *Part-Tree* increases, the learned clauses decrease the run time of *Part-Tree-Learn*. The initial learned clauses are usually long and cause overhead in the search. The "quality" of the learned clauses improves as the search proceeds, which probably explains the speed-up for the more difficult instances.

The second experiment compares *Cl-Sdsat* against *Part-Tree-Learn*. The high number of time-outs for *Cl-Sdsat* compared to *Part-Tree-Learn* suggests that usually *Part-Tree-Learn* performs better than *Cl-Sdsat*. This is an interesting result, since most state-of-the-art parallel SAT solvers are based on a *Cl-Sdsat*-style portfolio approach, where several SAT solvers are running in parallel and sharing learned clauses. However, *Cl-Sdsat* is able to solve some instances not solved by *Part-Tree-Learn*, an indication that search space partitioning might not be the best solving approach for all instances.

5.2 Comparison to Parallel SAT Solvers

We now compare *Part-Tree-Learn* to three multi-core SAT solvers and the sequential *MiniSat 2.2.0* underlying *Part-Tree-Learn*. The jobs of *Part-Tree-Learn* ran in the grid with 2GB memory and approximately 1 hour time limit. All other solvers ran with 24GB memory and 6 hour time limit in relatively modern 12-core nodes so that no other process could cause, e.g., memory bus intereference with the solver. As the nodes are to our knowledge faster than the nodes in the grid and each job of *Part-Tree-Learn* experienced in addition the queue delay, we feel that the comparison should be relatively fair to our "competitors". The number of cores used by the other solvers is lower

Table 1. Instances that were not solved in SATCOMP-2009

Name	Type	*Plingeling*	*ManySat 1.1*	*ManySat 1.5*	*Part-Tree-Learn*	*Part-Tree*
9dlx_vliw_at_b_iq8	Unsat	3256.41	2950.52	**2750.39**	—	—
9dlx_vliw_at_b_iq9	Unsat	5164.00	4240.00	**3731.00**	—	—
AProVE07-25	Unsat	—	—	—	**9967.24**	9986.58
dated-5-19-u	Unsat	4465.00	11136.00	18080.00	**2522.40**	4104.30
eq.atree.braun.12.unsat	Unsat	—	—	—	**4691.99**	5247.13
eq.atree.braun.13.unsat	Unsat	—	—	—	**9972.47**	12644.24
gss-24-s100	Sat	2929.92	—	6575.00	3492.01	**1265.33**
gss-26-s100	Sat	18173.00	**1232.17**	—	10347.41	16308.65
gus-md5-14	Unsat	—	—	—	13890.05	**13466.18**
ndhf_xits_09_UNSAT	Unsat	—	—	—	**9583.10**	11769.23
rbcl_xits_09_UNKNOWN	Unsat	—	—	—	9818.59	**8643.21**
rpoc_xits_09_UNSAT	Unsat	—	—	—	**8635.29**	9319.52
sortnet-8-ipc5-h19-sat	Sat	**2699.62**	10785.00	7901.00	4303.93	20699.58
total-10-17-u	Unsat	**3672.00**	6392.00	10755.00	4447.26	5952.43
total-5-15-u	Unsat	—	—	—	**18670.33**	21467.79

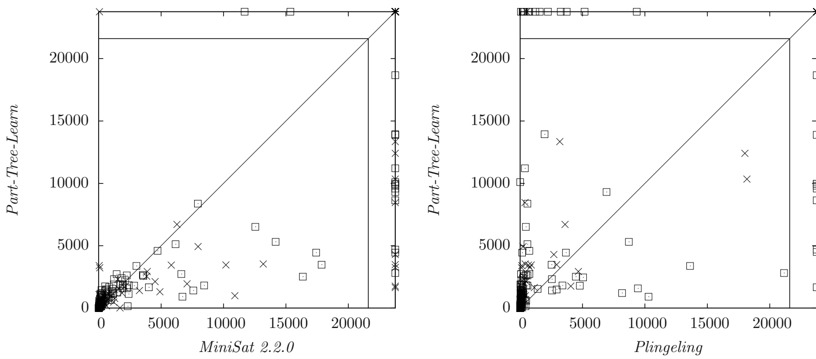

Fig. 6. Learning Partition Tree against *MiniSat 2.2.0* (left) and *Plingeling* (right)

than that used by *Part-Tree-Learn*, but we see this as an architectural limitation. For *Plingeling* we experimented with several numbers of cores. The 12-core configuration seemed to give the best result, whereas the default 4-core configuration was used for *ManySat 1.1* and *ManySat 1.5*. Of course, *MiniSat 2.2.0* was run with a single core.

Table 1 reports those of the 63 instances not solved in SATCOMP-2009 that were solved by at least one of the implementations in our experiments. The solvers *MiniSat 2.2.0* and *Cl-Sdsat* are omitted as they solved none of the unsolved instances. Based on the results, both *Part-Tree* and *Part-Tree-Learn* perform well on these hard instances, solving more instances than the other implementations. However, we do note that there are two instances from this benchmark set that *Part-Tree* and *Part-Tree-Learn* could not solve and three more where the solving time was lower in some other approach.

The *Part-Tree-Learn* approach is compared against *MiniSat 2.2.0* and *Plingeling* on the left and right hand side of Fig. 6, respectively, using the full set of application instances from SATCOMP-2009. We first note that *Part-Tree-Learn* performs in almost all more difficult instances significantly better than *MiniSat 2.2.0*. There are still some instances that cannot be solved with *Part-Tree-Learn*, an indication that the bounded job run times might limit the capabilities of *Part-Tree-Learn*. The comparison to the winner of SAT-Race 2010 *Plingeling* reveals that from the 292 instances, *Part-Tree-Learn*

could solve 227 and *Plingeling* 234. Based on the scatter plot there are several instances that are much faster solved with *Plingeling* although the number of cores available to *Plingeling* was lower. It is interesting to note that there are still many instances that solved quickly with *Part-Tree-Learn* which, based on the results in Table 1, are such that they are difficult for many other solvers competing in SATCOMP-2009. One could read the right-hand side plot of Fig. 6 so that if an instance can be solved, it is either solved quickly with *Plingeling* or *Part-Tree-Learn*. It is interesting, although beyond the scope of this work, to contemplate whether an implementation of *Part-Tree-Learn* based on *Plingeling* would indeed result in an even higher performance.

Finally we report that we could use *Part-Tree-Learn* to show unsatisfiable a challenge instance called aes-top-22-symmetryBreaking posed in a footnote of [8] in approximately 45 hours. Neither *Plingeling*, *ManySat 1.1*, *ManySat 1.5*, nor *MiniSat 2.2.0* could produce the result in three days, and this is indeed the fastest wall-clock time known computation of unsatisfiability for this instance.

6 Conclusions

This work introduces a new approach to solving hard SAT instances in a grid or cloud computing environment where a master sends jobs to workers having tight limits on their resources. The approach is based on partitioning iteratively a given formula to increasingly constrained derived formulas while maintaining a learned clause collection of heuristically increasing quality. Promising sets of learned clauses are selected from the collection for each job based on the constraints of the corresponding derived formula. Two techniques for clause learning in the workers are studied: the more fine-grained assumption-based tagging and the light-weight flag-based tagging. Two partition functions are used to produce the partitioning constraints.

The results indicate that the clause-learning partition tree approach compares favorably to state-of-the-art SAT solvers particularly in the most challenging SAT instances.

Acknowledgments. The authors are grateful for the financial support of the Academy of Finland (project 122399) and the valuable comments of the anonymous reviewers.

References

1. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Practical algorithms for unsatisfiability proof and core generation in SAT solvers. AI Communications 23(2-3), 145–157 (2010)
2. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University (2010)
3. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing 29(7), 969–994 (2003)
4. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver: Efficient workload balancing. Annals of Mathematics and Artificial Intelligence 17(4-3), 381–400 (1996)
5. Chrabakh, W., Wolski, R.: GridSAT: a system for solving satisfiability problems using a computational grid. Parallel Computing 32(9), 660–687 (2006)
6. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 226–241. Springer, Heidelberg (2009)

7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4) (2003)
8. Fuhs, C., Schneider-Kamp, P.: Synthesizing shortest linear straight-line programs over GF(2) using SAT. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 71–84. Springer, Heidelberg (2010)
9. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. Journal on Satisfiability, Boolean Modeling and Computation 6, 245–262 (2009)
10. Hamadi, Y., Jabbour, S., Sais, L.: Constrol-based clause sharing in parallel SAT solving. In: Proc. IJCAI 2009, pp. 499–504 (2009)
11. Heule, M., van Maaren, H.: Look-ahead based SAT solvers. In: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 155–184. IOS Press, Amsterdam (2009)
12. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
13. Hyvärinen, A.E.J., Junttila, T., 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)
14. 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)
15. 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)
16. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning search spaces of a randomized search. Fundamenta Informaticae 107(2-3), 289–311 (2011)
17. 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), 73–101 (2005)
18. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters 47(4), 173–180 (1993)
19. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability. IOS Press, Amsterdam (2009)
20. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
21. Martins, R., Manquinho, V., Lynce, I.: Improving search space splitting for parallel SAT solving. In: Proc. ICTAI 2010, pp. 336–343. IEEE Press, Los Alamitos (2010)
22. Michel, L., See, A., van Hentenryck, P.: Parallelizing constraint programs transparently. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 514–528. Springer, Heidelberg (2007)
23. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001, pp. 530–535. ACM, New York (2001)
24. Schubert, T., Lewis, M., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. Journal on Satisfiability, Boolean Modeling and Computation 6, 203–222 (2009)
25. Speckenmeyer, E., Monien, B., Vornberger, O.: Superlinear speedup for parallel backtracking. In: Houstis, E.N., Polychronopoulos, C.D., Papatheodorou, T.S. (eds.) ICS 1987. LNCS, vol. 297, pp. 985–993. Springer, Heidelberg (1988)
26. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: Proc. PDMC 2009. EPTCS, vol. 14, pp. 62–76 (2009)
27. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: A distributed propositional prover and its application to quasigroup problems. Journal of Symbolic Computation 21(4), 543–560 (1996)