Journal on Satisfiability, Boolean Modeling and Computation 6 (2009) 245-262

ManySAT: a Parallel SAT Solver

Youssef Hamadi youssefh@microsoft.com
Microsoft Research
7 J J Thomson Avenue, Cambridge CB3 0FB
United Kingdom

Said Jabbour jabbour@cril.fr
Lakhdar Sais sais@cril.fr
CRIL-CNRS, Université d'Artois
Rue Jean Souvraz SP18, F-62307 Lens Cedex
France

Abstract

In this paper, ManySAT a new portfolio-based parallel SAT solver is thoroughly described. The design of ManySAT benefits from the main weaknesses of modern SAT solvers: their sensitivity to parameter tuning and their lack of robustness. ManySAT uses a portfolio of complementary sequential algorithms obtained through careful variations of the standard DPLL algorithm. Additionally, each sequential algorithm shares clauses to improve the overall performance of the whole system. This contrasts with most of the parallel SAT solvers generally designed using the divide-and-conquer paradigm. Experiments on many industrial SAT instances, and the first rank obtained by ManySAT in the parallel track of the 2008 SAT-Race clearly show the potential of our design philosophy.

KEYWORDS: *parallel search, dynamic restarts, extended clause learning*

Submitted November 2008; revised May 2009; published June 2009

1. Introduction

In addition to the traditional hardware and software verification domains, SAT solvers are gaining popularity in new domains. For instance they are also used for general theorem proving and computational biology [11, 9]. This widespread adoption is the result of the efficiency gains made during the last decade [36]. Indeed, many industrial problems with hundreds of thousands of variables and millions of clauses are now solved within a few minutes. This impressive progress can be related to both the algorithmic improvements and the ability of SAT solvers to exploit the hidden structures[1] of a practical problem.

However, many new applications with instances of increasing size and complexity are coming to challenge modern solvers, while at the same time, it becomes clear that the gains traditionally given by low level algorithmic adjustments are gone. As a result, a large number of industrial instances from the last competitions remain challenging for all the available SAT solvers. Fortunately, the previous comes at a time where the generalization

1. By structure, we understand the dependencies between variables, which can often appear through functional constraints. One particular example being the well known notion of back-doors.

of multicore hardware gives parallel processing capabilities to standard PCs. While in general it is important for existing applications to exploit new hardwares, for SAT solvers, this becomes crucial.

Many parallel SAT solvers have been previously proposed. Most of them are based on the divide-and-conquer principle (see Section 5). They either divide the search space using for example guiding paths or the formula itself using decomposition techniques. The main problem behind these approaches rises in the difficulty to get workload balanced between the different processor units or workstations. Another drawback of these approaches rises in the fact that for a given large SAT instance with hundreds of thousands of variables it is very difficult to find the most relevant set of variables to divide the search space.

In this paper, we detail ManySAT, a new parallel SAT solver, winner of the 2008 Sat-Race[2.]. The design of ManySAT takes advantage of the main weakness of modern solvers: their sensitivity to parameter tuning. For instance, changing the parameters related to the restart strategy or to the variable selection heuristic can completely change the performance of a solver on a particular problem class. In a multicore context, we can easily take advantage of this lack of robustness by designing a portfolio which will run different incarnations of a sequential solvers on the same instance. Each solver would exploit a particular parameter set and their combination should represent a set of orthogonal yet complementary strategies. Moreover, individual solvers could perform knowledge exchange in order to improve the performance of the system beyond the performance of its individual components.

In the following, we report and discuss the design decisions taken while developing ManySAT. In section 2 we present some technical background about DPLL search, modern SAT solvers and multicore architectures. Section 3 describes our different design choices. Section 4 evaluates our solver on a large set of industrial benchmarks. Section 5 relates our work to previous works, and section 6 gives a general conclusion and points out some important perspectives.

2. Technical background

In this section, we first recall the basis of the most commonly used DPLL search procedure. Then, we introduce some computational features of modern SAT solvers. Finally, a brief description of multicore based architectures is given.

2.1 DPLL search

Most of the state of the art SAT solvers are simply based on the Davis, Putnam, Logemann and Loveland procedure, commonly called DPLL [10]. DPLL is a backtrack search procedure; at each node of the search tree, a *decision* literal is chosen according to some branching heuristics. Its assignment to one of the two possible values (true or false) is followed by an *inference* step that deduces and propagates some forced literal assignments such as unit and monotone literals. The assigned literals (decision literal and the propagated ones) are labeled with the same decision level starting from 1 and increased at each decision (or branching) until finding a model or reaching a conflict. In the first case, the formula is answered to be satisfiable, whereas in the second case, we backtrack to the last

2. http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/index.html

decision level and assign the opposite value to the last decision literal. After backtracking, some variables are unassigned, and the current decision level is decreased accordingly. The formula is answered to be unsatisfiable when a backtrack to level 0 occurs. Many improvements have been proposed over the years to enhance this basic procedure, leading now to what is commonly called modern SAT solvers. We also mention that, some look-ahead based improvements are at the basis of other kind of DPLL SAT solvers (e.g. Satz [28], kcnfs [12], march-dl [22]) particularly efficient on hard random and crafted SAT categories.

2.2 Modern SAT solvers

Modern SAT solvers [31, 14], are based on classical DPLL search procedure [10] combined with (i) restart policies [19, 25], (ii) activity-based variable selection heuristics (VSIDS-like) [31], and (iii) clause learning [30]. The interaction of these three components being performed through efficient data structures (e.g., watched literals [31]). All the state-of-the-art SAT solvers are based on a variation in these three important components.

Modern SAT solvers are especially efficient with "structured" SAT instances coming from industrial applications. VSIDS and other variants of activity-based heuristics [6], on the other hand, were introduced to avoid thrashing and to focus the search: when dealing with instances of large size, these heuristics direct the search to the most constrained parts of the formula. Restarts and VSIDS play complementary roles since the first component reorder assumptions and compacts the assumptions stack while the second allows for more intensification. Conflict Driven Clause Learning (CDCL) is the third component, leading to non-chronological backtracking. In CDCL a central data-structure is the *implication graph*, which records the partial assignment that is under construction together with its implications [30]. Each time a dead end is encountered (say at level i) a conflict clause or nogood is learnt due to a bottom up traversal of the implication graph. This traversal is also used to update the activity of related variables, allowing VSIDS to always select the most active variable as the new decision point. The learnt conflict clause, called asserting clause, is added to the learnt data base and the algorithm backtracks non chronologically to level $j < i$.

Progress saving is another interesting improvement, initially introduced in [16] it was recently presented in the Rsat solver [33]. It can be seen as a new selection strategy of the literal polarity. More precisely, each time a backtrack occurs from level i to level j, the literal polarity of the literals assigned between the two levels are saved. Then, such polarity is used in subsequent search tree. This can be seen as a partial component caching technique that avoids solving some components multiple times.

Modern SAT solvers can now handle propositional satisfiability problems with hundreds of thousands of variables or more. However, it is now recognised (see the recent SAT competitions) that the performances of the modern SAT solvers evolve in a marginal way. More precisely, on the industrial benchmarks category usually proposed to the annual SAT-Race and/or SAT-Competitions, many instances remain open (not solved by any solver within a reasonable amount of time). These problems which cannot be solved even using a 3 hours time limit are clearly challenging to all the available SAT solvers. Consequently, new approaches are clearly needed to solve these challenging industrial problems.

2.3 Multicore architectures

We can abstract a multicore architecture as a set of processing units which communicate through a shared memory. In theory, access to the memory is uniform, i.e., can be done simultaneously. Practically, the use of cache mechanisms in processing units creates coherence problems which can slow down the memory accesses.

Our work is built on this shared-memory model. The communication between the DPLLs solvers of a portfolio is organized through lockless queues that contain the lemmas that a particular core wants to exchange.

3. ManySAT: a parallel SAT solver

ManySAT is a DPLL-engine which includes all the classical features like two-watched-literal, unit propagation, activity-based decision heuristics, lemma deletion strategies, and clause learning. In addition to the classical first-UIP scheme [40], it incorporates a new technique which extends the implication graph used during conflict-analysis to exploit the satisfied clauses of a formula [1]. In the following, we describe and motivate for a set of important parameters, our design choices.

3.1 Restart policies

Restart policies represent an important component of modern SAT solvers. Contrary to the common belief, restarts are not used to eliminate the heavy tailed phenomena [19, 18] since after restarting SAT solvers dive in the part of the search space that they just left. In SAT, restarts policies are used to compact the assignment stack and improve the order of assumptions.

Different restart policies have been previously presented. Most of them are static, and the cutoff value follows different evolution scheme (e.g. arithmetic, geometric, Luby). To ensure the completeness of the SAT solver, in all these restarts policies, the cutoff value in terms of the number of conflicts increases over the time. The performance of these different policies clearly depends on the considered SAT instances. More generally, rapid restarts (e.g. Luby) perform well on industrial instances, however on hard SAT instances slow restarts are more suitable. Generally, it is hard to say in advance which policy should be used on which problem class [23].

Our objective was to use complementary restart policies to define the restart cutoff x_i.

We decided to use the well known Luby policy [29], and a classical geometric policy, $x_i = 1.5 \times x_{i-1}$ with $x_1 = 100$ [14]. The Luby policy was used with a unit factor set to 512. In addition, we decided to introduce two new policies. A very slow arithmetic one, $x_i = x_{i-1} + 16000$ with $x_1 = 16000$, and a new dynamic one.

3.1.1 New Dynamic Restart Policy

The early work on dynamic restart policy goes back to 2008. Based on the observation that frequent restarts significantly improve the performance of SAT solvers on industrial instance, Armin Biere presents in [2] a novel adaptive restart policy that measures the "agility" of the search process dynamically, which in turn is used to control the restart

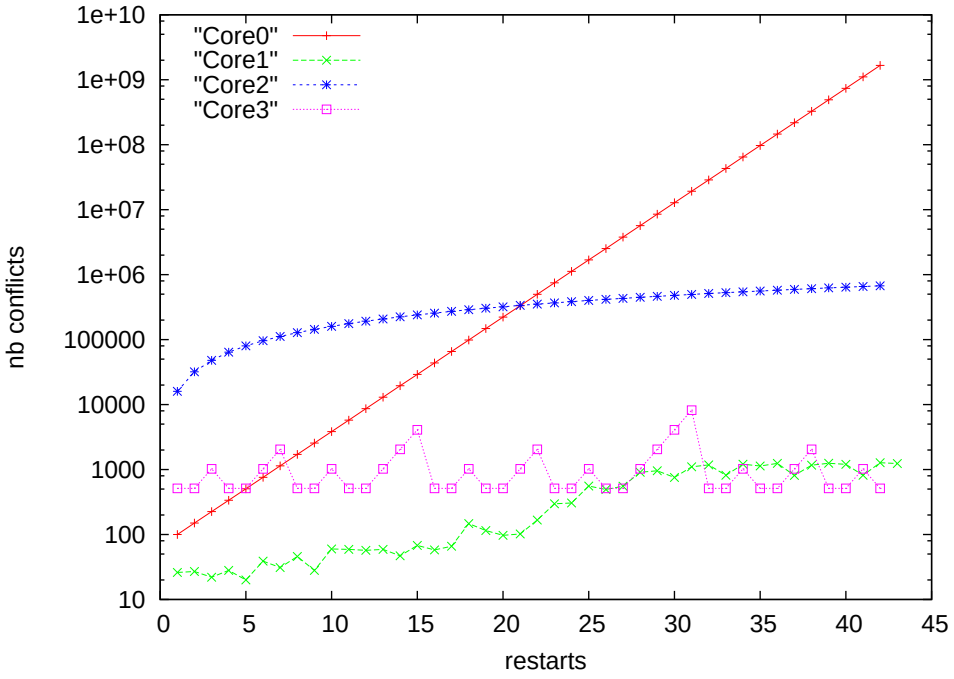

Figure 1. Restart strategies

frequency. The agility measures the average number of recently flipped assignments. Low agility enforces frequent restarts, while high agility tends to prohibit restarts.

In [35], the authors propose to apply restarts according to measures local to each branch. More precisely, for each decision level d a counter $c(d)$ of the number of conflicts encountered under the decision level d is maintained. When backtracking to the decision level d occurs, if the value $c(d)$ is greater than a given threshold, the algorithm restarts.

Considering CDCL-based SAT solvers, it is now widely admitted that restarts are an important component when dealing with industrial SAT instances, whereas on crafted and random instances they play a marginal role. More precisely, on industrial (respectively crafted) category, rapid (respectively long) restarts are more appropriate. It is important to note that on hard SAT instances, learning is useless. Indeed, on such instances, conflict analysis generally leads to a learnt clause which includes at least one literal from the level just before the current conflict level. In other words the search algorithm usually backjumps to the level preceding that of the current conflict. For example, if we consider the well known Pigeon-hole problem, learning from conflicts will produce a clause which includes at least one literal from each level. It is also obvious on this example, that learning does not achieve important backjumps in the search tree. The algorithm usually carries out a chronological backtracking.

In the following, we define a new dynamic restart policy based on the evolution of the average size of backjumps. First, such information is a good indicator of the decision errors made during search. Secondly, it can be seen as an interesting measure of the relative hardness of the instance. Our new policy is designed in such a way that, for high (respectively

low) fluctuation of the average size of backjumps (between the current and the previous restart), it delivers a low (respectively high) cutoff value. In other words, the cutoff value of the next restart depends on the average size of backjumps observed during the two previous and consecutive runs. We define it as, $x_1 = 100, x_2 = 100$, and $x_{i+1} = \frac{\alpha}{y_i} \times |cos(1 - r_i)|$, $i \geq 2$ where $\alpha = 1200$, y_i represents the average size of backjumps at restart i, $r_i = \frac{y_{i-1}}{y_i}$ if $y_{i-1} < y_i$, $r_i = \frac{y_i}{y_{i-1}}$ otherwise. The cutoff value x_i is minimal when the ratio between the average size of jumps between the two previous and consecutive runs is equal to one.

From the figure 1, we can observe that the cutoff value in terms of the number of conflicts is low in the first restarts and high at the last ones. This mean that the fluctuation between two consecutive restarts is more important at the beginning of the resolution process. Indeed, the activity of the variables is not sufficiently accurate in the first restarts, and the sub-problem on which the search focuses is not sufficiently circumscribed.

The dynamic restart policy, presented in this section is implemented in the first version of ManySAT [20] presented at the parallel track of the SAT Race 2008.

3.2 Heuristic

We decided to increase the random noise associated to the VSIDS heuristic [31] of core 0 since its restart policy is the slowest one. Indeed, that core tends to intensify the search, and slightly increasing the random noise allows us to introduce more diversification.

3.3 Polarity

Each time a variable is chosen, one needs to decide if such a variable might be assigned true (positive polarity) or false (negative polarity). Different kinds of polarity have been defined. For example, Minisat usually chooses the negative polarity, whereas Rsat uses progress saving. More precisely, each time a backtrack occurs, the polarity of the assigned variables between the conflict and the backjumping level are saved. If one of these variables is chosen again its saved polarity is preferred. In CDCL based solvers, the chosen polarity might have a direct impact on the learnt clauses and on the performance of the solver.

The polarity of the core 0 is defined according to the number of occurrences of each literal in the learnt data base. Each time a learnt clause is generated, the number of occurrences of each literal is increased by one. Then to maintain a more constrained learnt data base, the polarity of l is set to $true$ when $\#occ(l)$ is greater than $\#occ(\neg l)$; and to $false$ otherwise. For example by setting the polarity of l to $true$, we bias the occurrence of its negation $\neg l$ in the next learnt clauses.

This approach tends to balance the polarity of each literal in the learnt data base. By doing so, we increase the number of possible resolvents between the learnt clauses. If the relevance of a given resolvent is defined as the number of steps needed to derive it, then a resolvent between two learnt clauses might lead to more relevant clauses in the data base.

As the restart strategy in core 0 tends to intensify the search, it is important to maintain a learnt data base of better quality. However, for rapid restarts as in the core 1 and 3, progress saving is most suitable in order to save the work accomplished. For the core 2, we decided to apply a complementary polarity ($false$ by default as in Minisat).

3.4 Learning

Learning is another important component which is crucial for the efficiency of modern SAT solvers. Most of the known solvers use similar CDCL approaches associated with the first UIP (Unique Implication Point) scheme.

In our parallel SAT solver ManySAT, we used a new learning scheme obtained using an extension of the classical implication graph [1]. This new notion considers additional arcs, called inverse arcs. These are obtained by taking into account the satisfied clauses of the formula, which are usually ignored by classical conflict analysis. The new arcs present in our extended graph allow us to detect that even some decision literals admit a reason, something which is ignored when using classical implication graphs. As a result, the size of the backjumps is often increased.

Let us illustrate this new extended conflict analysis using a simple example. We assume that the reader is familiar with classical CDCL scheme used in modern SAT solvers (see [30, 31, 1]).

Let $\mathcal{F}$ be a CNF formula and ρ a partial assignment given below : $\mathcal{F} \supseteq \{c_1, \ldots, c_9\}$

(c_1)	$x_6 \vee \neg x_{11} \vee \neg x_{12}$	(c_2)	$\neg x_{11} \vee x_{13} \vee x_{16}$	(c_3)	$x_{12} \vee \neg x_{16} \vee \neg x_2$
(c_4)	$\neg x_4 \vee x_2 \vee \neg x_{10}$	(c_5)	$\neg x_8 \vee x_{10} \vee x_1$	(c_6)	$x_{10} \vee x_3$
(c_7)	$x_{10} \vee \neg x_5$	(c_8)	$x_{17} \vee \neg x_1 \vee \neg x_3 \vee x_5 \vee x_{18}$	(c_9)	$\neg x_3 \vee \neg x_{19} \vee \neg x_{18}$

$\rho = \{\langle \ldots \neg x_6^1 \ldots \neg x_{17}^1 \rangle \langle (x_8^2) \ldots \neg x_{13}^2 \ldots \rangle \langle (x_4^3) \ldots x_{19}^3 \ldots \rangle \ldots \langle (x_{11}^5) \ldots \rangle\}$. The sub-sequence $\langle (x_8^2) \ldots \neg x_{13}^2 \ldots \rangle$ of ρ expresses the set of literals assigned at level 2 with the decision literal mentioned in parenthesis and the set of propagated literals (e.g. $\neg x_{13}$). The current decision level is 5. The classical implication graph $\mathcal{G}_{\mathcal{F}}^{\rho}$ associated to $\mathcal{F}$ and ρ is shown in Figure 2 with only the plain arcs.

In the sequel, $\eta[x, c_i, c_j]$ denotes the *resolvent* between a clause c_i containing the literal x and c_j a clause containing the literal $\neg x$. In other words $\eta[x, c_i, c_j] = c_i \cup c_j \backslash \{x, \neg x\}$. Also a clause c subsume a clause c' iff $c \subseteq c'$.

The traversal of the graph $\mathcal{G}_{\mathcal{F}}^{\rho}$ allows us to generate three asserting clauses corresponding to the three possible UIPs (see figure 2). Let us illustrate the such resolution process leading to the first asserting clause Δ_1 corresponding to the first UIP.

- $\sigma_1 = \eta[x_{18}, c_8, c_9] = (x_{17}^1 \vee \neg x_1^5 \vee \neg x_3^5 \vee x_5^5 \vee \neg x_{19}^3)$
- $\sigma_2 = \eta[x_1, \sigma_1, c_5] = (x_{17}^1 \vee \neg x_3^5 \vee x_5^5 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$
- $\sigma_3 = \eta[x_5, \sigma_2, c_7] = (x_{17}^1 \vee \neg x_3^5 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$
- $\sigma_4 = \eta[x_3, \sigma_3, c_6] = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee x_{10}^5)$

As we can see, σ_4 gives us a first asserting clause (that we'll also name Δ_1) because all of its literals are assigned before the current level except one (x_{10}) which is assigned a the current level 5. The intermediate clauses σ_1, σ_2 and σ_3 contain more than one literal of the current decision level 5, and $\neg x_{10}$ is a first UIP. If we continue such a resolution process, we obtain the two additional asserting clauses $\Delta_2 = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee \neg x_4^3 \vee x_2^5)$, corresponding to a second UIP $\neg x_2^5$; and $\Delta_3 = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_8^2 \vee \neg x_4^3 \vee x_{13}^2 \vee x_6^1 \vee \neg x_{11}^5)$,

corresponding respectively to a 3rd UIP ($\neg x_{11}^5$) which is the last UIP since it corresponds to the last decision literal in the partial assignment.

In modern SAT solvers, clauses containing a literal x that is implied at the current level are essentially ignored by the propagation. More precisely, because the solver does not maintain the information whether a given clause is satisfied or not, a clause containing x may occasionally be considered by the propagation, but only when another literal y of the clause becomes false. When this happens the solver typically skips the clause. However, in cases where x is true *and all the other literals are false*, an "arc" was revealed for free that could as well be used to extend the graph. Such arcs are those we exploit in our proposed extension.

To explain further the idea behind our extension, let us consider, again, the formula $\mathcal{F}$ and the partial assignments given in the previous example. We define a new formula $\mathcal{F}'$ as follow : $\mathcal{F}' \supseteq \{c_1, \ldots, c_9\} \cup \{c_{10}, c_{11}, c_{12}\}$ where $c_{10} = (\neg x_{19} \vee x_8), c_{11} = (x_{19} \vee x_{10})$ and $c_{12} = (\neg x_{17} \vee x_{10})$

The three added clauses are satisfied under the instantiation ρ. c_{10} is satisfied by x_8 assigned at level 2, c_{11} is satisfied by x_{19} at level 3, and c_{12} is satisfied by $\neg x_{17}$ at level 1. This is shown in the extended implication graph (see Figure 2) by the doted edges. Let us now illustrate the usefulness of our proposed extension. Let us consider again the the asserting clause Δ_1 corresponding to the classical first UIP. We can generate the following strong asserting clause: $c_{13} = \eta[x_8, \Delta_1, c_{10}] = (x_{17}^1 \vee \neg x_{19}^3 \vee x_{10}^5)$, $c_{14} = \eta[x_{19}, c_{13}, c_{11}] = (x_{17}^1 \vee x_{10}^5)$ and $\Delta_1^s = \eta[x_{17}, c_{14}, c_{12}] = x_{10}^5$. In this case we backtrack to the level 0 and we assign x_{10} to *true*.

As we can see Δ_1^s subsumes Δ_1. If we continue the process we also obtain other strong asserting clauses $\Delta_2^s = (\neg x_4^3 \vee x_2^5)$ and $\Delta_3^s = (\neg x_4^3 \vee x_{13}^2 \vee x_6^1 \vee \neg x_{11}^5)$ which subsume respectively Δ_2 and Δ_3.

This first illustration gives us a new way to minimize the size of the asserting clauses.

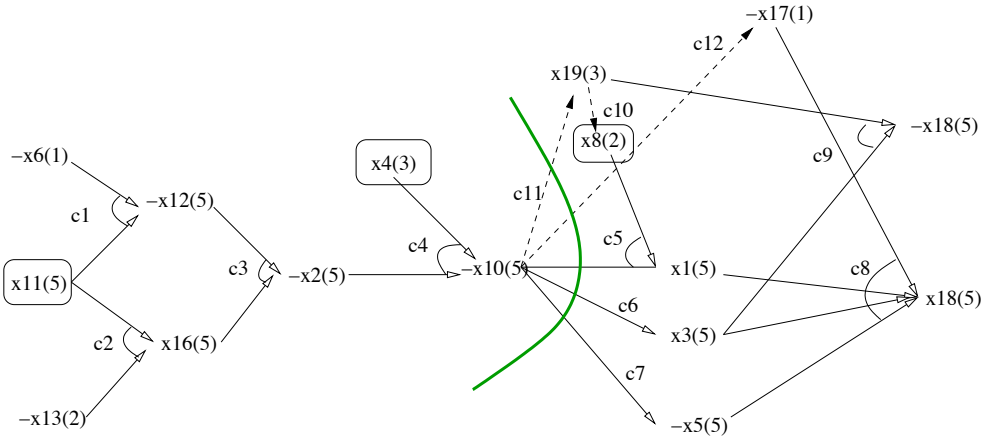

Figure 2. Implication graph / extended implication graph

Let us now explain briefly how the extra arcs can be computed. Usually unit propagation does not keep track of implications from the satisfiable sub-formula. In this extension the

new implications (deductions) are considered. For instance in the previous example, when we deduce x_{19} at level 3, we "rediscover" the deduction x_8 (which was a choice (decision literal) at level 2). Our proposal keeps track of these re-discoveries.

Our approach makes an original use of inverses arcs to back-jump farther, i.e. to improve the back-jumping level of the classical asserting clauses. It works in three steps. In the first step (1) : an asserting clause, say $\sigma_1 = (\neg x^1 \vee \neg y^3 \vee \neg z^7 \vee \neg a^9)$ is learnt using the usual learning scheme where 9 is the current decision level. As $\rho(\sigma_1) = false$, usually we backtrack to level 7. In the second step (2): our approach aims to eliminate the literal $\neg z^7$ from σ_1 using the new arcs of the extended graph. Let us explain this second and new processing. Let $c = (z^7 \vee \neg u^2 \vee \neg v^9)$ such that $\rho(z) = true$, $\rho(u) = true$ and $\rho(v) = true$. The clause c is an inverse arc i.e. the literal z assigned at level 7 is implied by the two literals u and v respectively assigned at level 2 and 9. From c and σ_1, a new clause $\sigma_2 = \eta[z, c, \sigma_1] = (\neg x^1 \vee \neg u^2 \vee \neg y^3 \vee \neg v^9 \vee \neg a^9)$ is generated. We can remark that the new clause σ_2 contains two literals from the current decision level 9. In the third step (3), using classical learning, one can search from σ_2 for another asserting clause σ_3 with only one literal from the current decision level. Let us note that the new asserting clause σ_3 might be worse in terms of back-jumping level. To avoid this main drawback, the inverse arc c is chosen if the two following conditions are satisfied : i) the literals of c assigned at the current level (v^9) has been already visited during the first step and ii) all the other literals of c are assigned before the level 7 i.e. level of z. In this case, we guaranty that the new asserting clause achieve better back-jumping.

This new learning scheme is integrated on the SAT solvers of the cores 0 and 3.

3.5 Clause sharing

Each core exchanges a learnt clause if its size is less or equal to 8. This decision is based on extensive tests with representative industrial instances. Figure 3 (respectively Figure 4) shows for different limits e the performance of ManySAT on instances taken from the SAT-Race 2008 (respectively SAT-Competition 2007). We can observe that on each set of benchmarks a limit size of 8 gives the best overall performance.

The communication between the solvers of the portfolio is organized through lockless queues which contain the lemmas that a particular core wants to exchange.

Each core imports unit-clauses when it reaches level 0 (e.g., after a restart). These important clauses correspond to the removal of Boolean variables, and therefore are more easily enforced at the top level of the tree.

All the other clauses are imported on the fly, i.e., after each decision. Several cases have to be handled for the integration of a foreign clause c:

- c is false in the current context. In this case, conflict-analysis has to start, allowing the search process to backjump. This is clearly the most interesting case.
- c is unit in the current context. The clause can be used to enforce more unit propagation, allowing the process to reach a smaller fix-point or a conflict.
- c is satisfied by the current context. It has to be watched. To exploit such a clause in the near future, we consider two literals assigned at the highest levels.

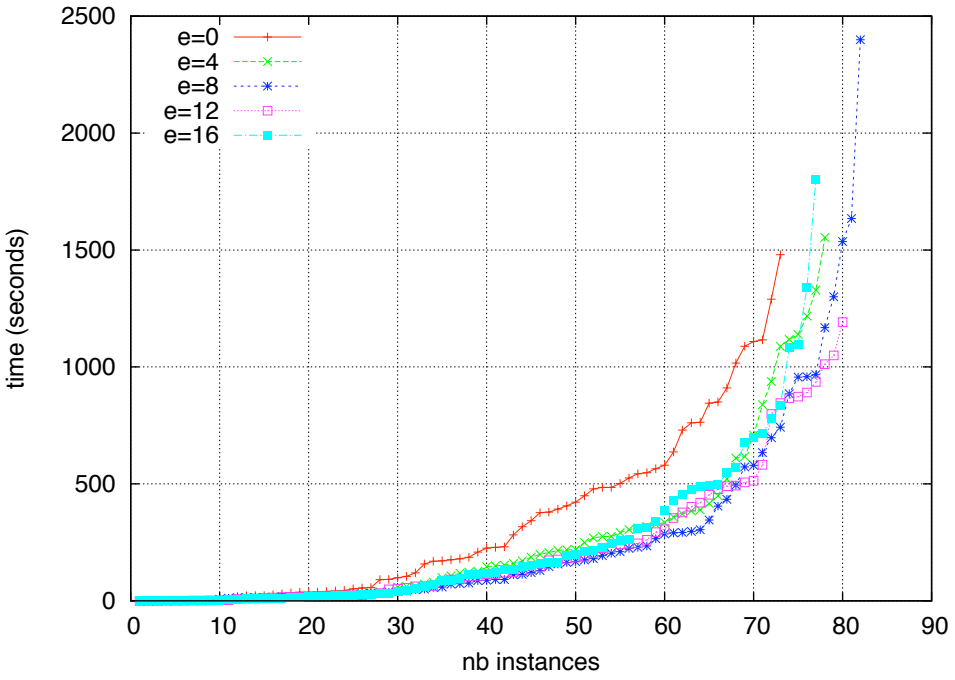

Figure 3. SAT-Race 2008: different limits for clause sharing

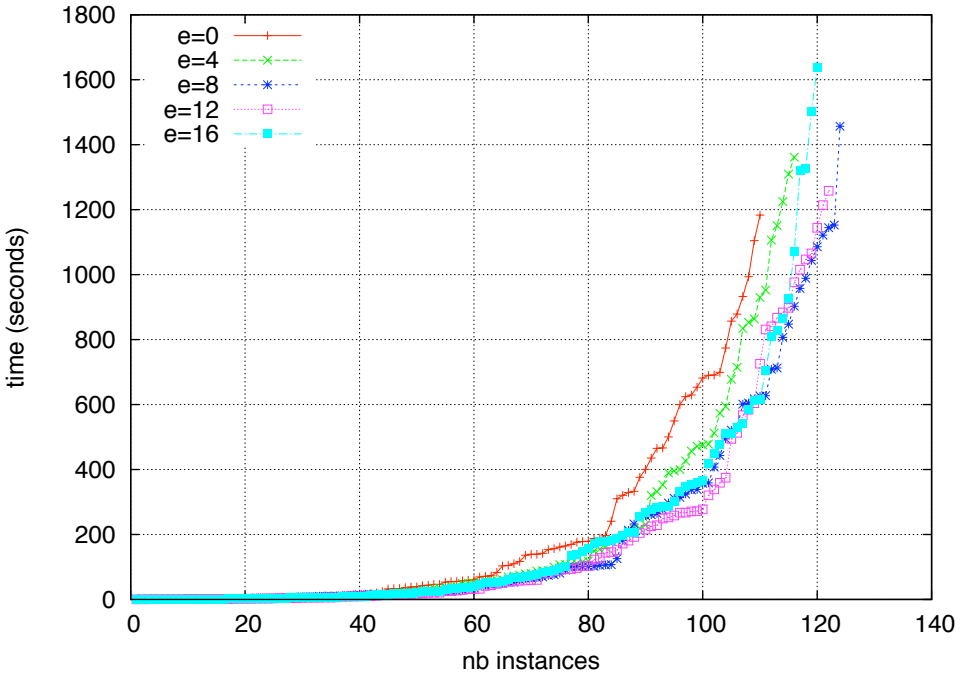

Figure 4. SAT-Competition 2007: different limits for clause sharing

- otherwise, c has to be watched. In this last case, the first two unassigned literals are watched.

The following example illustrates the different cases mentioned above.
Let $\mathcal{F}$ be a CNF formula and $\rho = \{\langle \dots \neg x_6^1 \dots \neg x_{17}^1 \rangle \langle (x_8^2) \dots \neg x_{13}^2 \dots \rangle \langle (x_4^3) \dots x_{19}^3 \dots \rangle \dots \langle (x_{11}^5) \neg x_{12}^5, x_{16}^5, \neg x_2^5, \dots, \neg x_{10}^5, x_1^5, \dots, x_{18}^5 \rangle\}$ a partial assignment. To make the shared clause c exploitable in a near future, it might be watched in a certain way. Suppose that,

- $c = (x_{17}^1 \vee \neg x_{19}^3 \vee x_{10}^5) \in \mathcal{F}$. The clause c is false and the two literals $\neg x_{19}^3$ and x_{10}^5 are watched.
- $c = (x_{17}^1 \vee \neg x_{19}^3 \vee x_{30}) \in \mathcal{F}$. The clause c is unit and the two literals $\neg x_{19}^3$ and x_{30} are watched;
- $c = (x_{17}^1 \vee \neg x_{19}^3 \vee \neg x_{10}^5) \in \mathcal{F}$. We watch the last satisfied literal $\neg x_{10}$ and another literal with the highest level from the remaining ones.
- $c = (x_{25} \vee \neg x_{34} \neg x_{29}) \in \mathcal{F}$. We watch any two literals from c.

3.6 Summary

Table 1 summarizes the choices made for the different solvers of the ManySAT portfolio. For each solver (core), we mention the restart policy, the heuristic, the polarity, the learning scheme and the size of shared clauses.

Table 1. ManySAT: different strategies

Strategies	**Core 0**	**Core 1**	**Core 2**	**Core 3**
Restart	Geometric $x_1 = 100$ $x_i = 1.5 \times x_{i-1}$	Dynamic (Fast) $x_1 = 100, x_2 = 100$ $x_i = f(y_{i-1}, y_i), i > 2$ if $y_{i-1} < y_i$ $f(y_{i-1}, y_i) = \frac{\alpha}{y_i} \times \lvert cos(1 - \frac{y_{i-1}}{y_i}) \rvert$ else $f(y_{i-1}, y_i) = \frac{\alpha}{y_i} \times \lvert cos(1 - \frac{y_i}{y_{i-1}}) \rvert$ $\alpha = 1200$	Arithmetic $x_1 = 16000$ $x_i = x_{i-1} + 16000$	Luby 512
Heuristic	VSIDS (3% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)	VSIDS (2% rand.)
Polarity	if $\#occ(l) > \#occ(\neg l)$ $l = true$ else $l = false$	Progress saving	false	Progress saving
Learning	CDCL (extended [1])	CDCL	CDCL	CDCL (extended [1])
Cl. sharing	size ≤ 8	size ≤ 8	size ≤ 8	size ≤ 8

4. Evaluation

4.1 Performance against a sequential algorithm

ManySAT was built on top of Minisat 2.02 [14]. SatElite was applied systematically by each core as a pre-processor [13]. In all the figures, instances solved by Satellite in the pre-processing step are not included. In this section, we evaluate the performance of the solver

on a large set of industrial problems. Figure 5, shows the improvement of performances provided by our solver when opposed to the sequential solver Minisat 2.02 on the problems of the Sat-Race 2008. It shows the performance of ManySAT running with respectively 1,2,3 and 4 cores. When more than one core is used, clause sharing is done up to clause size 8.

We can see that even the sequential version of ManySAT (single core) outperforms Minisat 2.02. This simply means that our design choices for core 1 represent a good combination to put in a sequential solver. Interestingly, with each new core, the performance increases both in speed and number of problems solved. This is the result of the diversification of the search but also the fact that clause sharing quickly boosts these independent search processes.

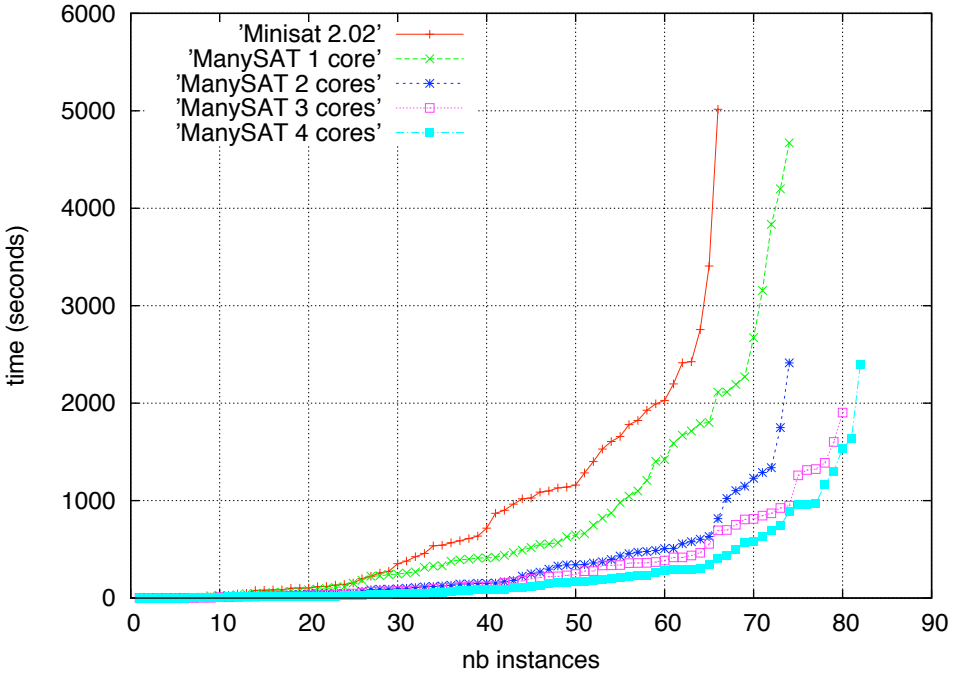

Figure 5. SAT-Race 2008: ManySAT e=8, m=1..4 against Minisat 2.02

4.2 Performance against other parallel SAT solvers

We report here the official results of the 2008 Sat-Race. They can be downloaded from the competition website[3]. They demonstrate the performance of ManySAT as opposed to other parallel SAT solvers. These tests were done on 2x Dual-Core Intel Xeon 5150 running at 2.66 GHz, with a timeout set to 900 seconds.

The Table 2 shows the number of problems (out of 100) solved before the time limit for ManySAT, pMinisat [8], and MiraXT [27] - these solvers are described in the next section. We can see that ManySAT solves 5 more problems than pMinisat, which solves 12

3. http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/

more problems than MiraXT. Interestingly, the performance of our method is well balanced between SAT and UNSAT problems.

Table 2. SAT-Race 2008: comparative performance (number of problems solved)

	ManySAT	**pMinisat**	**MiraXT**
SAT	45	44	43
UNSAT	45	41	30

Table 3. SAT-Race 2008: parallel solvers against the best sequential solver (Minisat 2.1)

	ManySAT	**pMinisat**	**MiraXT**
Average speed-up	6.02	3.10	1.83
by SAT/UNSAT	8.84 /3.14	4.00/2.18	1.85/1.81
Minimal speed-up	0.25	0.34	0.04
by SAT/UNSAT	0.25/0.76	0.34/0.46	0.04/0.74
Maximal speed-up	250.17	26.47	7.56
by SAT/UNSAT	250.17/4.74	26.47/10.57	7.56/4.26

Table 3 shows the speed-up provided by these parallel SAT algorithms as opposed to the best sequential algorithm of the Sat-Race 2008, Minisat 2.1. We can see that on average, ManySAT is able to provide a superlinear speed-up of 6.02. It is the only solver able of such performance. The second best provides on average a speed-up of 3.10, far from linear. When we consider the minimal speed-up we can see that the performance of the first two solvers is pretty similar. They decrease the performance against the best sequential solver of the 2008 Sat-Race by up to a factor 4, while the third solver decreases the performance by a factor 25. Finally, the maximal speed-up is given by ManySAT which can be up to 250 times faster than Minisat 2.1. These detailed results show that the performance of the parallel solvers is usually better on SAT problems than on UNSAT ones.

Table 4. SAT-Race 2008: runtime variation of parallel solvers

	ManySAT	**pMinisat**	**MiraXT**
Average variation	13.7%	14.7%	15.2%
by SAT/UNSAT	22.2%/5.5%	23.1%/5.7%	19.5%/9.7%

It is well known that parallel search is not deterministic. Table 4 gives the average runtime variation of each parallel solver. ManySAT exhibits a lower variation than the other techniques, but the small differences between the solvers do not allow us to draw any definitive conclusion.

5. Previous work

We present here the most noticeable approaches related to parallel SAT solving.

PSATO [38] is based on the SATO (SAtisfiability Testing Optimized) sequential solver [39]. Like SATO, it uses a *trie* data structure to represent clauses. PSATO uses the notion of *guiding-paths* to divide the search space of a problem. These paths are represented by a set of unit clauses added to the original formula. The parallel exploration is organized in a master/slave model. The master organizes the work by addressing guiding-paths to workers which have no interaction with each others. The first worker to finish stops the system. The balancing of the work is organized by the master.

In [24] a parallelization scheme for a class of SAT solvers based on the DPLL procedure is presented. The scheme uses a dynamic load-balancing mechanism based on work-stealing techniques to deal with the irregularity of SAT problems. PSatz is the parallel version of the well known Satz solver.

Gradsat [7] is based on zChaff. It uses a master-slave model and the notion of guiding-paths to split the search space and to dynamically spread the load between clients. Learned clauses are exchanged between all clients if they are smaller than a predefined limit on the number of literals. A client incorporates a foreign clause when it backtracks to level 1 (top-level).

In [3], the authors use an architecture similar to Gradsat. However, a client incorporates a foreign clause if it is not subsumed by the current guiding-path constraints. Practically, clause sharing is implemented by *mobile-agents*. This approach is supposed to scale well on computational grids.

Nagsat [15] is a parallel SAT solver which exploits the heavy-tailed distribution of random 3-SAT instances. It implements *nagging*, a notion taken from the DALI theorem prover. Nagging involves a master and a set of clients called *naggers*. In Nagsat, the master runs a standard DPLL algorithm with a static variable ordering. When a nagger becomes idle, it requests a *nagpoint* which corresponds to the current state of the master. Upon receiving a nagpoint, it applies a transformation (e.g., a change in the ordering or the remaining variables), and begins its own search on the corresponding subproblem.

In [4], the input formula is dynamically divided into disjoint subformulas. Each subformula is solved by a sequential SAT-solver running on a particular processor. The algorithm uses optimized data structures to modify Boolean formulas. Additionally workload balancing algorithms are used to achieve a uniform distribution of workload among the processors.

MiraXT [27], is designed for shared memory multiprocessors systems. It uses a divide-and-conquer approach where threads share a unique clause database which represents the original and the learnt clauses. When a new clause is learnt by a thread, it uses a lock to safely update the common database. Read access can be done in parallel.

PMSat uses a master-slave scenario to implement a classical divide-and-conquer search [17]. The user of the solver can select among several partinioning heuristics. Learnt clauses are shared between workers, and can also be used to stop efforts related to search spaces that have been proven irrelevant. PMSat runs on networks of computer through an MPI implementation.

In [8], the authors use a standard divide-and-conquer approach based on guiding-paths. However, it exploits the knowledge on these paths to improve clause sharing. Indeed, clauses

can be large with respect to some static limit, but when considered with the knowledge of the guiding path of a particular thread, a clause can become small and therefore highly relevant. This allows pMiniSat to extend the sharing of clauses since a large clause can become small in another search context.

In [34], the authors use a portfolio of distributed backtracking algorithms to solve distributed constraint satisfaction problems. They let different search algorithms run in parallel, give hints to each other and compete for being the first to finish and deliver the solution. This approach has inspired our ManySAT solver.

6. Conclusion

We have presented ManySAT, a portfolio-based parallel SAT solver which advantageously exploits multicore architectures. ManySAT is based on an understanding of the main weakness of modern sequential SAT solvers, their sensitivity to parameter tuning and their lack of robustness. As a result, ManySAT uses a portfolio of complementary sequential algorithms, and let them cooperate in order to improve further the overall performance. This design philosophy of ManySAT clearly contrasts with most of the well known parallel SAT solvers. The good performance obtained by ManySAT on industrial SAT instances clearly suggests that the portfolio based approach is more interesting than the traditional divide-and-conquer based one.

In this paper, we also proposed a new and efficient dynamic restart policy which exploits relevant measures of the search tree and a new polarity strategy for literal assignment. While developing ManySAT we learned a lot on the trade offs related to having (dis)similar search strategies in a portfolio. In the future we are going to consider this aspect in order to improve the benefit of cooperation in parallel SAT.

ManySAT has been already extended to integrate dynamic clause sharing policies as described in [21]. This work presents two innovative policies to dynamically adjust the size of shared clauses between any pair of processing units. The first one controls the overall number of exchanged clauses whereas the second additionally exploits the relevance quality of shared clauses. Experimental results show important improvements when compared to the optimal static size policy described in here ($e = 8$).

Even if we truly believe that portfolio-based approached should be mixed with divide-and-conquer ones as soon as the number of processing units is significant, the question of scalability of the ManySAT portfolio approach has to be asked. As stated here, our four-cores portfolio was carefully crafted in order to mix complementary strategies. If ManySAT could be run on dozens of computing units, what would be the performance? We have considered this question in a more general context in [5]. This work presents the first study on scalability of constraint solving on 100 processors and beyond. It proposes techniques that are simple to apply and shows empirically that they scale surprisingly well. It proves that portfolio-based approaches can also scale-up to several dozens of processors.

Finally, as stated in the introduction, SAT is now applied to other domains. One domain which particularly benefits from the recent advances in SAT is Satisfiability Modulo Theory [32]. There, our ManySAT approach has been integrated to the Z3 SMT solver [11], allowing it to achieve impressive speed-ups on several classes of problems [37].

References

[1] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In Kleine Büning and Zhao [26], pages 21–27.

[2] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In Kleine Büning and Zhao [26], pages 28–33.

[3] W. Blochinger, C. Sinz, and W. Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, **29**(7):969–994, 2003.

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

[5] L. Bordeaux, Y. Hamadi, and H. Samulowitz. Experiments with massively parallel constraint solving. In *IJCAI 2009, Proceedings of the 21th International Joint Conference on Artificial Intelligence, to appear*, 2009.

[6] L. Brisoux, E. Grégoire, and L. Sais. Improving backtrack search for sat by means of redundancy. In *Foundations of Intelligent Systems, 11th International Symposium, ISMIS '99*, **1609** of *Lecture Notes in Computer Science*, pages 301–309. Springer, 1999.

[7] W. Chrabakh and R. Wolski. GrADSAT: A parallel sat solver for the grid. Technical report, UCSB Computer Science Technical Report Number 2003-05, 2003.

[8] G. Chu and P. J. Stuckey. Pminisat: a parallelization of minisat 2.0. Technical report, Sat-race 2008, solver description, 2008.

[9] F. Corblin, L. Bordeaux, Y. Hamadi, E. Fanchon, and L. Trilling. A sat-based approach to decipher gene regulatory networks. In *Integrative Post-Genomics, RIAMS, Lyon*, 2007.

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

[11] L. Mendonça de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, **4963** of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[12] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'01*, pages 248–253, 2001.

[13] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, SAT'05*, **3569** of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[14] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, **2919** of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[15] S. L. Forman and A. M. Segre. Nagsat: A randomized, complete, parallel solver for 3-sat. sat2002. In *Proceedings of Theory and Applications of Satisfiability Testing, SAT'02*, pages 236–243, 2002.

[16] D. Frost and R. Dechter. In search of the best constraint satisfaction search. In *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94*, pages 301–306, 1994.

[17] L. Gil, P. Flores, and L. M. Silveira. PMSat: a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, **6**:71–98, 2008.

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

[19] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.

[20] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: solver description. Technical Report MSR-TR-2008-83, Microsoft Research, may 2008.

[21] Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. In *IJCAI 2009, Proceedings of the 21th International Joint Conference on Artificial Intelligence, to appear*, 2009.

[22] M. J. H. Heule and H. van Maaren. March dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:47–59, 2006.

[23] J. Huang. The effect of restarts on the efficiency of clause learning. In M. M. Veloso, editor, *IJCAI*, pages 2318–2323, 2007.

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

[25] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman. Dynamic restart policies. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 674–682, 2002.

[26] H. Kleine Büning and X. Zhao, editors. *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, **4996** of *Lecture Notes in Computer Science*. Springer, 2008.

[27] M. Lewis, T. Schubert, and B. Becker. Multithreaded sat solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.

[28] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'97*, pages 366–371, 1997.

[29] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, **47**:173–180, 1993.

[30] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[31] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[32] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(). *J. ACM*, **53**(6):937–977, 2006.

[33] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. P. Marques-Silva and K. A. Sakallah, editors, *Theory and Applications of Satisfiability Testing, SAT'07*, **4501** of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[34] G. Ringwelski and Y. Hamadi. Boosting distributed constraint satisfaction. In P. van Beek, editor, *International Conference on Principles and Practice of Constraint Programming, CP'05*, **3709** of *Lecture Notes in Computer Science*, pages 549–562. Springer, 2005.

[35] V. Ryvchin and O. Strichman. Local restarts. In Kleine Büning and Zhao [26], pages 271–276.

[36] L. Sais, editor. *Problème SAT : Progrès et Défis*. Hermes Publishing Ltd, Londres, 2008. http://www.lavoisier.fr/notice/fr423832.html.

[37] C. Wintersteiger, Y. Hamadi, and L. de. Moura. A concurrent portfolio approach to SMT solving. In *CAV 2009, Proceedings of the Twenty-one International Conference on Computer Verification, to appear*, 2009.

[38] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, **21**:543–560, 1996.

[39] H. Zhang and M. E. Stickel. Implementing the davis-putnam algorithm by tries. Technical report, Artificial Intelligence Center, SRI International, Menlo, 1994.

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