Diversification and Intensification in Parallel SAT Solving

Long Guo[1], Youssef Hamadi[2,3], Said Jabbour[4], and Lakhdar Sais[1]

[1] Université Lille-Nord de France
CRIL - CNRS UMR 8188
Artois, F-62307 Lens
{guo, sais}@cril.fr

[2] Microsoft Research
7 J J Thomson Avenue
Cambridge, United Kingdom

[3] LIX École Polytechnique
F-91128 Palaiseau, France
youssefh@microsoft.com

[4] INRIA-Microsoft Research Joint Centre
28 rue Jean Rostand
91893 Orsay Cedex, France
said.jabbour@inria.fr

Abstract. In this paper, we explore the two well-known principles of diversification and intensification in portfolio-based parallel SAT solving. These dual concepts play an important role in several search algorithms including local search, and appear to be a key point in modern parallel SAT solvers. To study their trade-off, we define two roles for the computational units. Some of them classified as *Masters* perform an original search strategy, ensuring diversification. The remaining units, classified as *Slaves* are there to intensify their master's strategy. Several important questions have to be answered. The first one is what information should be given to a slave in order to intensify a given search effort? The second one is, how often, a subordinated unit has to receive such information? Finally, the question of finding the number of subordinated units and their connections with the search efforts has to be answered. Our results lead to an original intensification strategy which outperforms the best parallel SAT solver ManySAT, and solves some open SAT instances.

Keywords: Satisfiability, SAT and CSP, Search

1 Introduction

In addition to the traditional hardware and software verification fields, SAT solvers are gaining popularity in new domains. For instance they are also used for general theorem proving and computational biology. This widespread adoption is the result of the efficiency gains made during the last decade [1]. Indeed, industrial instances with hundred of thousand 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 to the ability of SAT solvers to exploit the hidden structures[1] of such instances. However, new applications are always more challenging with instances of increasing size and complexity, while the gains traditionally given by low level algorithmic adjustments are now stalling. As a result, a large number of industrial instances from the last competitions remain challenging for all the available SAT solvers. Fortunately, this last challenge comes at a time where the generalization of multicore hardware gives parallel processing capabilities to standard PCs. While in general it is important for existing applications to exploit these 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 (e.g. [2]). They generally divide the search space using the well known guiding-path concept [3]. The main problems behind these approaches rise in the difficulty to get workload balanced between the different processing units and in finding the best guiding path. Also, splitting the search tree using guiding paths leads to the exploration of unrelated parts of the search space and reduces the benefit of clauses sharing. Portfolio-based parallel SAT solving has been recently introduced [4]. It avoids the previous problem by letting several differentiated DPLL engines compete and cooperate to be the first to solve a given instance. Each solver works on the original formula, and search spaces are not split or decomposed anymore. To be efficient, the portfolio has to use diversified search engines with clauses sharing. The key point remains in finding such strategies while maintaing clauses exchange of higher quality. In ManySAT [4], the state-of-the-art portfolio-based parallel SAT solver, such diversification is obtained by a careful combination of different restarts policies, literals polarity assignment, and learning schemes. These differentiated search strategies enhanced with clause sharing aim to explore the search space with less possible redundancies. The first rank obtained by ManySAT on the parallel track of the 2008 SAT Race and 2009 SAT competition demonstrates that portfolio-based parallel approaches clearly outperform the divide-and-conquer based ones.

However, when clause sharing is added, diversification has to be restricted in order to maximize the impact of a foreign clause whose relevance is more important in a similar or related search effort. Despite the efficiency of ManySAT, the question of finding the best portfolio of diversified strategies while maintaining a hight quality of exchange remains very challenging. Indeed, two orthogonal (respectively close) strategies might reduce (respectively increase) the impact of clause sharing. Therefore, a challenging question is to maintain a good and relevant "distance" between the parts of the search space explored by the different search units which is equivalent to the finding of a good diversification and intensification tradeoff. Indeed, intensification (respectively diversification) directs the search to the same (respectively different) parts of the search space. This question heavily depends on the problem instance. On hard ones it might be more convenient to direct the search towards building the same and common proof (intensification), whereas on easy ones diversifying it might be the way towards finding a short proof.

[1] By structure, we understand the dependencies between variables, which can often appear through Boolean functions. One particular example being the well known notion of backdoors.

Taking this in mind, we propose to study the diversification/intensification tradeoff in a parallel SAT portfolio. We define two roles for the computational units. Some of them classified as *Masters* perform an original search strategy, ensuring diversification. The remaining ones, classified as *Slaves* are there to intensify their master's strategy. Doing so, several important questions have to be answered. The first one is what information should be given to a unit in order to intensify a given search effort? The second one is, how often, a subordinated unit has to receive such information? Finally, the question of finding the number of subordinated units along their connections with original search efforts has to be answered. In other words, we need to determine the best Masters/Slaves division and hierarchy i.e. topology.

In the following, Section two describes the internals of modern SAT solvers, and the architecture of a portfolio-based parallel SAT engine. Section three studies the best way to intensify a given search strategy. Section four, considers the different diversification/intensification tradeoffs in a portfolio. Section five, presents our experimental results. Finally, before the general conclusion, section six presents the related works.

2 Technical Background

In this section, we first introduce the most salient computational features of modern SAT solvers. Then, we describe a typical portfolio based parallel SAT solver.

2.1 Modern SAT Solvers

Modern SAT solvers [5, 6], are based on classical DPLL search procedure [7] combined with (i) restart policies [8, 9], (ii) activity-based variable selection heuristics (VSIDS-like) [5], and (iii) clause learning [10]. The interaction of these three components being performed through efficient data structures (e.g., Watched literals [5]).

Modern SAT solvers are especially efficient with "structured" SAT instances coming from industrial applications. On these problems, Gomes et al. [11] have identified a heavy tailed phenomenon, i.e., different variable orderings often lead to dramatic differences in solving time. This explains the introduction of restart policies in modern SAT solvers, which attempt to discover a good variable ordering. VSIDS and other variants of activity-based heuristics [12], 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. VSIDS and restarts are two important and connected components since the first increase the activities of the variables involved in conflicts while the second allows the solver to reorder the decision stack according to these activities. Conflict Driven Clause Learning (CDCL) is the third component, leading to non-chronological backtracking. In CDCL a central data-structure is the *implication graph* [10], which records the partial assignment under construction made of the successive *decision literals* (chosen variable with either positive or negative *polarity*) with their propagations. Each time a conflict is encountered (say at level i) a *conflict clause* or nogood is learnt thanks to a bottom up traversal of the implication graph. Such a traversal can be seen as a resolution derivation starting from the two implications of the conflicting variable. The next resolvent is generated, from the previous one and another clause from the implication graph. Such linear resolution derivation stops when the current resolvent $(\alpha \vee a)$, contains only one literal a from the current conflict level, called an *asserting literal*. The node in the graph labeled with $\neg a$ is called the *first Unique Implication Point* (first-UIP). This traversal or resolution process 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 $(\alpha \vee a)$, called *asserting clause*, is added to the learnt data base and the algorithm backtracks non chronologically to level $j < i$.

Modern SAT solvers can now handle propositional satisfiability problems with hundreds of thousands of variables or more. However, it is now recognized (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 Races and/or SAT Competitions, many instances remain open (not solved by any solver within a reasonable amount of time). Consequently, new approaches are clearly needed to solve these challenging industrial problems.

2.2 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 [13], it incorporates a new technique which extends the implication graph used during conflict-analysis to exploit the satisfied clauses of a formula [14]. Unlike other parallel SAT solvers, ManySAT does not implement a divide-and-conquer strategy based on some dynamic partitioning of the search space. On the contrary, it uses a portfolio philosophy which lets several sequential DPLLs compete and cooperate to be the first to solve the common instance. These DPLLs are differentiated in many ways. They use different and complementary restart strategies, VSIDS, polarity heuristics, and learning schemes. Additionally, all the DPLLs are exchanging learnt clauses up to some size limit.

As ManySAT finished first during the 2008 SAT Race and 2009 SAT Competition (parallel track - industrial category), we conducted our experimental comparison using this state-of-the-art parallel SAT solver.

3 Towards a Good Intensification Strategy

In this section, we first determine the relevant knowledge to be passed from a Master to a Slave in order to intensify the search. Secondly, we address the frequency of such directed intensification.

To this end, we consider a simple system with two computing units, respectively a Master (M) and a Slave (S) (see Figure 1). The role of the Master is to invoke the Slave for search intensification (dashed arrow in Figure 1). By intensification we mean that the slave would explore ”differently” around the search space explored by the Master. Consequently, the clauses learnt by the Master and the Slave are relevant to each other and shared in both direction (plain line in Figure 1).

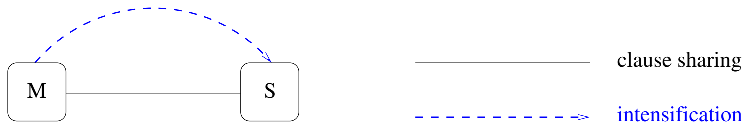

Fig. 1. Intensification topology

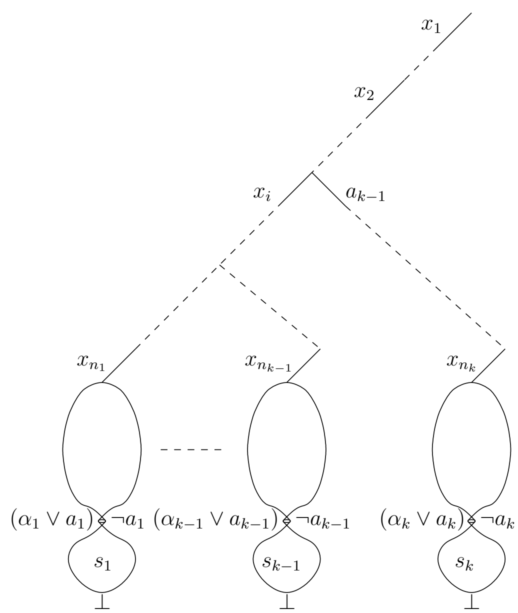

Fig. 2. A partial view of the Master search tree : conflicts branches and implication graphs

To explore differently around a given search effort, several kind of knowledge can be considered. Suppose that the Master is currently at a given state $S_M = (\mathcal{F}, \mathcal{D}_M, \Gamma_M)$, where $\mathcal{F}$ is the original SAT instance, $\mathcal{D}_M$ the set of decision literals, and Γ_M the set of learnt clauses (learnt database). In the following, from a given state S_M, we derive three different knowledge characterizing the Master search effort.

We use Figure 2, to illustrate such knowledge. It represents a current state S_M corresponding to the branch leading to the last conflict k. The decisions made in the last branch are x_1, x_2,. . ., x_{n_k}. The boxes give a partial view of the implication graph obtained on the last k conflicts derived after the assignment of the last decisions x_{n_k}, $x_{n_{k-1}}$, . . ., and x_{n_1}. The learnt clauses are respectively $(\alpha_k \vee a_k)$, $(\alpha_{k-1} \vee a_{k-1})$, . . ., and $(\alpha_1 \vee a_1)$ where a_k, a_{k-1}, . . ., and a_1 are the asserting literals corresponding to the first-UIP $\neg a_k$, $\neg a_{k-1}$, . . ., and $\neg a_1$.

Decision list The first kind of knowledge characterizing the Master search effort uses the current set of decisions $\mathcal{D}_M$ (in short $decision\ list$). Using such decisions, the Slave can build the whole or a subset of the current partial assignment of the Master depending if all the asserting clauses generated by M on the current branch are passed to S. Since the activity of the variables are not passed to the Slave, it shall explore the same area in a different way.

Asserting set The second one, uses the sequence $A_M =< a_k, a_{k-1}, \ldots, a_1 >$ (in short $asserting\ set$) of the Master asserting literals associated to the k clauses learnt before the current state S_M. The sequence is ordered from the latest to the oldest conflict. By branching on the ordered sequence A_M using the same polarity, the Slave is able to construct a partial assignment involving the most recent asserting literals learnt from the Master unit. Let us recall that an asserting literal a_i is part of the Master learnt clause $(\alpha \vee a_i)$. As the Slave branches on a_i, future conflicts analysis involving a_i, might lead to learnt clauses containing $\neg a_i$. More generally, invoking the Slave using A_M pushes it to learn more relevant clauses, connected by resolution (contains complementary literals) to the most recent clauses learned by M. This is clearly an intensification process, as the clauses learnt by S involve the most important literals of M, and lead in some way to a more constructive resolution proof thanks to the complementary shared literals between M's learnt clauses, and the future clauses that will be learnt by S.

Conflict sets The last one, uses the sequence of ordered sets $C_M =< s_k, s_{k-1}, \ldots, s_1 >$ of literals collected during the Master conflict analysis (in short $conflict\ sets$). The set s_k represents the set of literals collected during the last conflict analysis. More precisely, the literals in s_k correspond to the nodes of the implication graph located between the conflict node and the the first-UIP node $\neg a_k$ (see Figure 2). Moreover, the set s_k includes a literal of the conflicting variable and the literal labeling the first-UIP node $\neg a_k$. It can be defined as $s_k =< y_{k_1}, y_{k_2}, \ldots, y_{k_m} >$, where y_{k_1} corresponds to the literal of the first-UIP node $\neg a_k$ and y_{k_m} to the literal of the conflict variable as it appears in the current partial assignment. The aim of considering this sequence of sets is to intensify the search by directing S around the same conflicts. Let us note that the activity of the variables appearing in the conflict sets are those updated during conflict analysis. One can use the most active variables of the Masters to direct the search of the Slaves. However, exploiting such kind of knowledge leads to redundant search between the Masters and Slaves i.e. the Masters and Slaves tends to reproduce the same search.

We can first remark that, the sequence A_M and C_M might contain redundant literals (the same literal occurs several times). As the Slave S assign such literals according to the defined ordering, S chooses the next unassigned literal in the ordering. For the first one, and as mentioned above, the Master invoke the Slave using the decision list D_M together with the set of asserting clauses learnt on the current branch in order to build the same partial assignment.

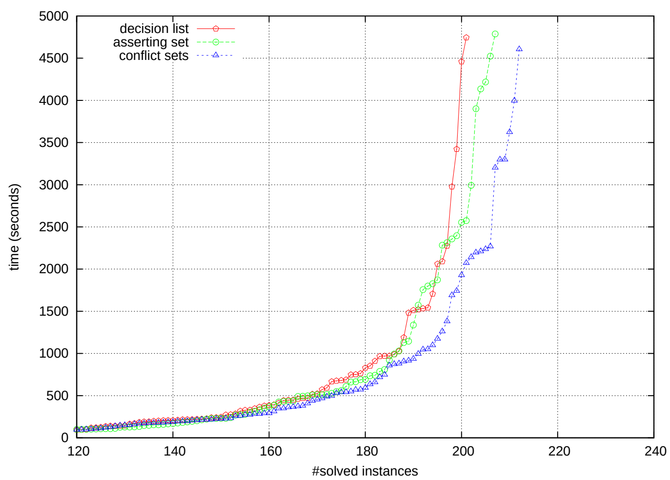

Fig. 3. Three intensification strategies

To compare the relevance of the previously defined intensification strategies, we conducted the following experiments on the whole set of instances (292 instances) from the industrial category of the 2009 SAT Competition. We use ManySAT with two computing units (see figure 1) sharing clauses of size less or equal to 8. The Master M invokes the Slave S at each restart and transmits at the same time the intensification knowledge. For the Master M we used a rapid restart strategy. It is widely admitted that rapid restarts lead to better learning [15] or to learnt clauses of small width [16]. Additionally, rapid restarts provide frequent intensification of the Slave leading to a tight synchronization of the search efforts.

Let us note that, the Slave do not implement any restart strategy. It restarts when invoked by the Master. For the Master, we use in this experiments the rapid and dynamic restart policy introduced in [4].

The Figure 3, shows the experimental comparison using the above three intensification strategies ($decision\ list$, $asserting\ set$, and $conflict\ sets$). It presents the cumulated time results i.e. the number of instances (x-axis) solved under a given amount of time in seconds (y-axis). As we can observe, directing the search using $conflict\ sets$ gives the best results. The number of solved instances using the $decision\ list$, $asserting\ set$ and $conflict\ sets$ are 201, 207 and 212 respectively. In the rest of this paper, we use $conflict\ sets$ as the intensification strategy.

4 Towards a Good Search Tradeoff

This section explores the diversification and intensification tradeoff. We are using the ManySAT architecture which is represented by a clique of four computational units interacting through clause sharing [4] up to size 8. As ManySAT finished first during the 2008 SAT Race and 2009 SAT Competition (parallel track - industrial category), we are testing our intensification technique against a state-of-the-art solver. These units represent a fully diversified set of strategies. In order to add some intensification, we propose to extend this architecture and to partition the units between Masters and Slaves. If we allow a Slave to intensify its own search effort through another Slave, we have a total of seven possible configurations. They are presented in Figure 4. In this Figure, dotted lines represents the Master/Slave relationships. Note that when a unit has to provide intensification directives to several Slaves, it alternates its guidance between them, i.e., round-table. Moreover, when a configuration contains chain(s) of Slaves, (see (d), (f), and (g) in Figure 4), the intensification of a Slave of level i is triggered by the Slave of level $i - 1$.

These configurations represent all the possible diversification and intensification tradeoffs which can be implemented on top of the ManySAT architecture. We recall that ManySAT exploit diversified search strategies on each core [4]. In ManySAT, the different cores (or processing units) are ordered according to their overall performance from the best (core 0) to the least best (core 3). The performance of the different cores are taken from the results obtained by ManySAT during the last SAT 2009 competition and corresponds to the number of instances solved by each core. In the different topologies of Figure 4, the core 0, core 1, core 2 and core 3 corresponds to the processing unit at the bottom left, bottom right, top right and top left boxes respectively. Naturally, in our experiments, we allocate in priority the best strategies of ManySAT to Masters and the least performant ones to Slaves. This rational choice avoids to consider all the possible symmetric topologies that can be obtained by simple rotations.

The following section explores their respective performances and compare them to the original ManySAT solver.

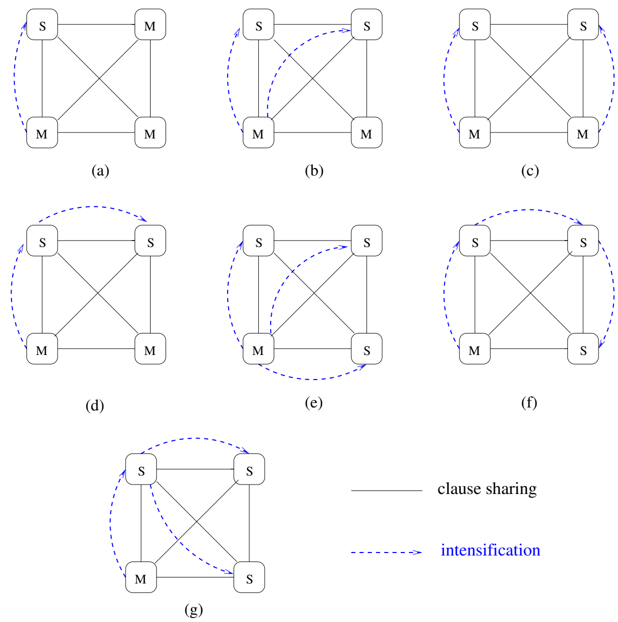

Fig. 4. Diversification/Intensification topologies

5 Experiments

Our tests were done on Intel Xeon quadcore machines with 32GB of RAM running at 2.66 Ghz. For each instance, we used a timeout of 4 hours of CPU time which corresponds to a 1 hour timeout per computational unit (core). Our Master/Slave roles and their different configurations were implemented on top of the original ManySAT. This solver was also used as a baseline for comparison. We used the $conflict\ sets$ intensification strategy.

We used the 292 industrial instances of the 2009 SAT competition to compare our different algorithms.

Method	**# SAT**	**# UNSAT**	**Total**	**Tot. time (sc.)**	**Avg. time**
ManySAT	87	125	212	329378	1128
Topo. (a)	86 (7)	133 (49)	219 (56)	311590	1067
Topo. (b)	84 (28)	130 (73)	214 (101)	324800	1112
Topo. (c)	**89** (23)	132 (74)	**221 (97)**	**307345**	**1052**
Topo. (d)	87 (25)	132 (67)	219 (92)	315537	1080
Topo. (e)	86 (45)	131 (109)	217 (154)	323208	1106
Topo. (f)	82 (44)	128 (102)	210 (146)	339677	1163
Topo. (g)	80 (45)	127 (107)	207 (152)	343800	1177

Table 1. 2009 SAT Competition, Industrials: overall results

The Table 1 summarizes our results. The first column presents the method, i.e., the original ManySAT (first line) or ManySAT extended with one of our seven diversification/intensification topology (see Figure 4). In the second column, the first number represents the overall number of SAT instances solved by the associated method, the second number (in parenthesis) gives the number of instances found SAT by a Slave. The third column gives similar information for UNSAT problems. The column four, gives the overall number of instances solved, again the parenthesis gives the number solved by one of the Slaves. To alleviate the effects of unpredictable threads scheduling, each instance was solved three times and we take the average as the time needed for solving a given instance. The average is calculated using the 1 hour timeout when an instance is not solved at a given run. Finally, the last two columns give respectively, the total time (cumulated), and the average time in seconds calculated over the overall set of 292 instances.

This Table shows that the vast majority of our topology-based extensions are superior to the original ManySAT. This algorithm solves 212 problems whereas the best topology (c) solves 221. Remarkably, all the topologies are able to solve more UNSAT problems than ManySAT. This unsurprisingly shows that adding intensification, is more beneficial on this last category of problems. Indeed, our intensification strategy increases the relevance of the learnt clauses exchanged between masters and slaves, since unsatisfiable instances are mainly solved by resolution, improving the quality of the learnt clauses increases the performances on UNSAT problems.

When we compare the results achieved by our different topologies. It seems that balancing the tradeoff between 2 Slaves and 2 Masters works better (topo. b, c, and d). Among them, balancing the slaves to the masters gives the most efficient results i.e., topology c.

Instance	**Status**	**ManySAT**	**Topology (c)**
9dlx_vliw_at_b_iq1	UNSAT	87.3	**10.6**
9dlx_vliw_at_b_iq2	UNSAT	226.3	**27.1**
9dlx_vliw_at_b_iq3	UNSAT	602.8	**103.2**
9dlx_vliw_at_b_iq4	UNSAT	1132	**163.5**
9dlx_vliw_at_b_iq5	UNSAT	2428	**313.1**
9dlx_vliw_at_b_iq6	UNSAT	–	**735.6**
9dlx_vliw_at_b_iq7	UNSAT	–	**991**
9dlx_vliw_at_b_iq8	**UNSAT**	–	**1822.7**
9dlx_vliw_at_b_iq9	**UNSAT**	–	**2670.1**
velev-pipe-sat-1.0-b10	SAT	4.4	**3.6**
velev-engi-uns-1.0-4nd	UNSAT	5	**4.9**
velev-live-uns-2.0-ebuf	UNSAT	6.7	**6.8**
velev-pipe-sat-1.0-b7	SAT	48.3	**6.2**
velev-pipe-o-uns-1.1-6	UNSAT	65.2	**30.8**
velev-pipe-o-uns-1.0-7	UNSAT	149.9	**118.2**
velev-pipe-uns-1.0-8	UNSAT	274.5	**82.7**
velev-vliw-uns-4.0-9C1	UNSAT	297.2	**235.4**
velev-vliw-uns-4.0-9-i1	UNSAT	–	**1311.6**
goldb-heqc-term1mul	UNSAT	23.8	**4.3**
goldb-heqc-i10mul	UNSAT	36.3	**23.5**
goldb-heqc-alu4mul	UNSAT	49.9	**40.9**
goldb-heqc-dalumul	UNSAT	384.1	**33.6**
goldb-heqc-frg1mul	UNSAT	2606	**83.1**
goldb-heqc-x1mul	UNSAT	–	**246.9**

Table 2. 2009 SAT Competition, Industrials: time (s) results on three families

The Table 2 highlights the results achieved by our best topology (c) against ManySAT on three complete families of problems. We can see that our best topology outperforms ManySAT on all these problems. Let us mention that we have not found families where ManySAT dominates our best topology (c). Even more importantly, our algorithm allowed the resolution of two open instances (9dlx_vliw_at_b_iq8, and 9dlx_vliw_at_b_iq9), proved UNSAT for the first time.

The Figure 5, presents cumulated time results for ManySAT and for our best topology on the whole set of problems. On small time limit (less than 10 minutes), the algorithms have the same behavior. On the other hand, when more time is allowed, the new technique exhibits an important improvement, and solves 9 more instances.

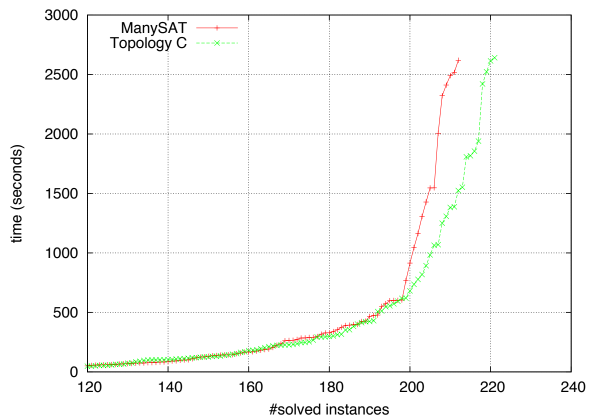

Fig. 5. 2009 SAT Competition, Industrials: cumulated time

Finally, it is important to note that in the last SAT 2009 competition no sequential or parallel SAT solver has been able to reach such number of solved instances. In SAT 2009 competition, all the solvers are allowed a time limit of about 3 hours (10 000 seconds) for a given instance. The tests were done on a Intel Xeon machines with 2GB of RAM and 3.2 Ghz. The Virtual Best Solver (VBS) solved 229 instances (91 SAT and 138 UNSAT). VBS is a theoretical construction which returns the best answer provided by one of the submitted solver. An instance is solved by VBS if it is solved by at least one of the submitted solvers. Another way to look at it is to consider this VBS as a solver which would run all other solvers in parallel, bringing together all the solvers strengths. This VBS is essentially the same notion as State Of The Art (SOTA) solver defined in [17]. From the description above, we can measure that the performance of our proposed approach is very close to those of VBS.

6 Related Works

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

In [18] 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 [19] 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).

[20] uses 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.

In [21], 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 [2], 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 [22]. The user of the solver can select among several partitioning 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.

[23] uses 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 [24] a SAT Solver c-sat, a parallelization of MiniSat using MPI is presented. It employs a layered master-worker architecture, where the masters handle lemma exchange, deletion of redundant lemmas and the dynamic partitioning of search trees, while the workers do search using different decision heuristics and random number seeds.

In [25] a new switching criterion based on the evenness or unevenness of the distribution of variable weights is presented. The proposed hybrid local search algorithm combines intensification and diversification by switching between two different heuristics using this criterion.

Other portfolio-based solvers have been proposed in the sequential context, such as Satzilla [26] or cpHydra [27], they mainly based on running several solvers on a set of training instances in order to determine the most appropriate solver to solve a given instance. They clearly differ from parallel portfolio based solvers, where all the solvers of the portfolio are run in parallel and clauses are shared between them.

7 Conclusion

We have explored the two well-known principles of diversification and intensification in portfolio-based parallel SAT solving. These dual concepts play an important role in several search algorithms including local search, and appear to be a key point in modern parallel SAT solvers. To study their tradeoff, we defined two roles for the computational units. Some of them classified as *Masters* perform an original search strategy, ensuring diversification. The remaining units, classified as *Slaves* are there to intensify their master's strategy.

Several important questions have been addressed. The first one is what information should be given to a slave in order to intensify a given search effort? It appeared that passing the set of literals found during previous conflict analysis gives the best results. This strategy aims at directing the slave towards conflicts highly related to the master's conflicts, allowing masters and slaves to share highly relevant clauses.

The second one is, how often, a subordinated unit has to receive such information? We have decided to exploit the restart policy of a master to refresh the information given to its slave(s). As shown in other works, rapid restarts lead to better learning [15] or to learnt clauses of small width [16]. Therefore, a rapid restarts strategy on the master node reinforces the interests of the clauses shared with its slaves. In our context it allows frequent intensification of a Slave leading to a tight synchronization of the search efforts.

Finally, the question of finding the number of subordinated units along their connections with the search efforts had to be answered. Our tests have shown that balancing the set of nodes between Masters and Slaves roles, and balancing the slaves to the masters gives the best results. In particular, our best topology solves 9 more industrial instances than the actual best solver, ManySAT. The results have also demonstrated the relative performance of the intensification strategy on UNSAT problems. Remarkably, our new strategy was able to close the 9dlx_vliw_at_b_iq* family by finding the proofs of unsatisfiability for two open instances.

As future work, we would like to dynamically adapt the topology and roles in a portfolio based on the perceived hardness of a given instance. This should benefit to hard UNSAT proofs were several units could be used for intensification, and at the same time, could preserve performances on difficult SAT problems where intensification is less needed. A second interesting path for future research concerns the integration of control based clause sharing [28] in this context. The third issue is to address scalability, one of the most important challenge in parallel SAT solving. The framework proposed in this paper is better suited to achieve this goal. Indeed, as the intensification leads to clause sharing of better quality, we can allow such exchange between a Master and its Slave only. This will reduce the number of shared clauses while maintaining the overall performance. Other measures of clause-quality need to be defined in order to reduce the global number of exchanged clauses. Finally, we plan to extend our proposed framework for solving other problems around SAT including constraint satisfaction problems.

References

1. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
2. Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Multithreaded sat solving. In *ASP-DAC*, pages 926–931, 2007.
3. Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.
4. Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation - JSAT*, 6:245–262, 2009.
5. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, pages 530–535, 2001.
6. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.
7. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
8. Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.
9. Henry A. Kautz, Eric Horvitz, Yongshao Ruan, Carla P. Gomes, and Bart Selman. Dynamic restart policies. In *AAAI/IAAI*, pages 674–681, 2002.
10. Joao P. Marques-Silva and Karem 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.
11. Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, 24(1/2):67–100, 2000.
12. Laure Brisoux, Éric Grégoire, and Lakhdar Sais. Improving backtrack search for SAT by means of redundancy. In *Foundations of Intelligent Systems, 11th International Symposium, ISMIS '99*, volume 1609 of *Lecture Notes in Computer Science*, pages 301–309. Springer, 1999.
13. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.
14. Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In *Theory and Applications of Satisfiability Testing, SAT'2008*, pages 21–27, 2008.
15. Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing, SAT'2008*, pages 28–33, 2008.
16. Knot Pipatsrisawat and Adnan Darwiche. Width-based restart policies for clause-learning satisfiability solvers. In *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, pages 341–355, June 2009.

17. Geoff Sutcliffe and Christian B. Suttner. Evaluating general purpose automated theorem proving systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.
18. Bernard Jurkowiak, Chu Min Li, and Gil Utard. A parallelization scheme based on work stealing for a class of sat solvers. *Journal of Automated Reasoning*, 34(1):73–101, 2005.
19. Wahid Chrabakh and Rich Wolski. GrADSAT: A parallel sat solver for the grid. Technical report, UCSB Computer Science Technical Report Number 2003-05, 2003.
20. Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.
21. Max Böhm and Ewald Speckenmeyer. A fast parallel sat-solver - efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3-4):381–400, 1996.
22. Luis Gil, Paulo Flores, and Luis Miguel Silveira. PMSat: a parallel version of minisat. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.
23. Geoffrey Chu and Peter J. Stuckey. Pminisat: a parallelization of minisat 2.0. Technical report, Sat-race 2008, solver description, 2008.
24. Kei Ohmura and Kazunori Ueda. c-sat: A parallel sat solver for clusters. In *In proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing - SAT'2009*, pages 524–537, 2009.
25. Wanxia Wei, Chu Min Li, and Harry Zhang. A switching criterion for intensification and diversification in local search for sat. *Journal on Satisfiability, Boolean Modeling and Computation - JSAT*, 4(2-4):219–237, 2008.
26. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.
27. Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of the 19th Irish conference on artificial intelligence and cognitive science (AICS 2008)*, 2008.
28. Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pages 499–504, 2009.