Control-based Clause Sharing in Parallel SAT Solving

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

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

Abstract

Conflict driven clause learning, one of the most important component of modern SAT solvers, is also recognized as very important in parallel SAT solving. Indeed, it allows clause sharing between multiple processing units working on related (sub-)problems. However, without limitation, sharing clauses might lead to an exponential blow up in communication or to the sharing of irrelevant clauses. This paper, proposes two innovative policies to dynamically adjust the size of shared clauses between any pair of processing units. The first approach controls the overall number of exchanged clauses whereas the second additionally exploits the relevance quality of shared clauses. Experimental results show important improvements of the state-of-the-art parallel SAT solver.

1 Introduction

The recent successes of SAT solvers in traditional hardware and software applications have extended their applicability to important new domains. Today, they represent essential low level reasoning components used in general theorem proving, computational biology, AI, etc. This popularity gain is related to their breakthrough on real world instances involving million of clauses and hundred of thousands of variables. These solvers, called modern SAT solvers [Moskewicz *et al.*, 2001; Eén and Sörensson, 2003], are based on a nice combination of (i) clause learning [Marques-Silva and Sakallah, 1996; Moskewicz *et al.*, 2001], (ii) activity-based heuristics [Moskewicz *et al.*, 2001], and (iii) restart policies [Gomes *et al.*, 1998] enhanced with efficient data structures (e.g. watched-literals [Moskewicz *et al.*, 2001]). This architecture is now standard, and today only minor improvements have been observed (cf. SAT competitions). Therefore, it seems difficult to bet on others orders of magnitude gains without a radical algorithmic breakthrough.

Fortunately, the recent generalization of multicore hardware gives parallel processing capabilities to standard PCs. This represents a real opportunity for SAT researchers which can now consider parallel SAT solving as an obvious way toward substantial efficiency gains.

Recent works on parallel SAT are all based on the modern SAT architecture [Moskewicz *et al.*, 2001], and therefore systematically exploit clause learning as an easy way to extend the cooperation between processing units. When a unit learns a new clause, it can share it with all other units in order to prune their search spaces. Unfortunately, since the number of potential conflicts is exponential, the systematic sharing of learnt clauses is not practically feasible. The solution is to exchange up to some predefined size limit. This has the advantage of reducing the overhead of the cooperation while focusing the exchange on short clauses, recognized as more powerful in term of search pruning.

In this work, our goal is to improve the clause sharing scheme of modern parallel SAT solvers. Indeed, the approach based on some predefined size limit has several flaws. The first and most apparent being that an overestimated value might induce a very large cooperation overhead, while an underestimated one might completely inhibit the cooperation. The second flaw comes from the observation that the size of learnt clauses tends to increase over time (see section 3.1), leading to an eventual halt of the cooperation. The third flaw is related to the internal dynamic of modern solvers which tend to focus on particular subproblems thanks to the activity/restart mechanisms. In parallel SAT, this can lead two search processes toward completely different subproblems where clause sharing becomes pointless.

We propose a dynamic clause sharing policy which uses pairwise size limits to control the exchange between any pair of processing units. Initially, high limits are used to enforce the cooperation, and allow pairwise exchanges. On a regular basis, each unit considers the number of foreign clauses received from other units. If this number is below/above a predefined threshold, the pairwise limits are increased/decreased. This mechanism allows the system to maintain a throughput. It addresses the flaws one and two. To address the last flaw related to the poor relevance of the shared clauses, we extend our policy to integrate the quality of the exchanges. Each unit evaluates the quality of the received clauses, and the control is able to selectively increase/decrease the pairwise limits based on the underlying quality of the recently communicated clauses. The rationale being that the information recently received from a particular source is qualitatively linked to the information which could be received from it in the very near future. The evolution of the pairwise limits w.r.t., the throughput or quality criterion follows an AIMD (Additive-Increase-Multiplicative-Decrease) feedback control-based algorithm [Chiu and Jain, 1989].

The paper is organized as follows. After some preliminaries (section 2), our dynamic control-based clause sharing policies are motivated and presented in section 3. The section 4 presents extensive experimental evaluation of our policies as opposed to a standard static one. The section 5 details previous parallel SAT works. Finally, we conclude by providing some interesting future paths of research.

2 Technical background

In this section, we introduce the computational features of modern SAT solvers. Then, we briefly describe the principle of the AIMD feedback control-based algorithm usually applied to solve TCP congestion control problems.

2.1 Computational features of modern SAT solvers

Most of the state of the art SAT solvers are based on the Davis, Putnam, Logemann and Loveland procedure, commonly called DPLL [Davis *et al.*, 1962]. DPLL is a backtrack search procedure; at each node of the search tree, a decision literal is chosen according to some branching heuristic. Its assignment to one of the two possible values (true or false) is followed by an inference step that deduces and propagates some forced unit literal assignments. 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 (or a dead end). In the first case, the formula is answered to be satisfiable, whereas in the second case, we backtrack to the last decision level and assign the remaining 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 backtracking to level 0 occurs. In addition to this basic scheme, modern SAT solvers use additional important component such as restart policy, conflict driven clause learning and activity based heuristics. Let us give some details on these last two important features. First, to learn from conflict, they maintain a central data-structure, the implication graph, which records the partial assignment that is under construction together with its implications. When a dead end occurs, a conflict clause (called asserting clause) is generated by resolution following a bottom-up traversal of the implication graph. The learning process stops, when a conflict clause containing only one literal from the current decision level is generated. Such a conflict clause (or learnt clause) expresses that such a literal is implied at a previous level. Modern SAT solvers backtrack to the implication level and assign that literal to true. Let us mention, that the activity of each variable encountered during such resolution process is increased. The variable with greatest activity is selected to be assigned next.

As the number of learnt clauses can grow exponentially, even in the sequential case, modern SAT solvers regularly reduce the data-base of learnt clauses. In the SAT solver Minisat [Eén and Sörensson, 2003], such a reduction (called reduceDB) is achieved as follows. When the size of the learnt data base exceeds a given upper bound (B), it is reduced by half. The set of deleted clauses corresponds to the less active ones. Initially, B is set to $\frac{1}{3} \times |\mathcal{F}|$) where $|\mathcal{F}|$ is the number of clauses in the original formula $\mathcal{F}$. At each restart, B is increased by 10%.

2.2 AIMD feedback control based algorithm

The Additive Increase/Multiplicative Decrease (AIMD) algorithm is a feedback control algorithm used in TCP congestion avoidance. The problem solved by AIMD is to guess the communication bandwidth available between two communicating nodes. The algorithm performs successive probes, increasing the communication rate w linearly as long as no packet loss is observed, and decreasing it exponentially when a loss is encountered. More precisely, the evolution of w is defined by the following $AIMD(a, b)$ formula:

- $w = w - a \times w$, if loss is detected
- $w = w + \frac{b}{w}$, otherwise

Different proposals have been made in order to prevent congestion in communication networks based on different numbers for a and b. Today, AIMD is the major component of TCP's congestion avoidance and control [Jacobson, 1988]. On probe of network bandwidth increasing too quickly will overshoot limits (underlying capacities). On notice of congestion, decreasing too slowly will not be reactive enough.

In the context of clause sharing, our control policies want to achieve a particular throughput or a particular throughput of maximum quality. Since any increase in the size limit can potentially generate a very large number of new clauses, AIMD's slow increase can help us to avoid a quick overshoot of the throughput. Similarly, in case of overshooting, aggressive decrease can help us to quickly reduce clause sharing by a very large amount.

3 Control-based clause sharing in parallel SAT solving

3.1 Motivation

To motivate further our proposed framework, we conducted a simple experiment using the standard Minisat algorithm [Eén and Sörensson, 2003]. In figure 1 we show the evolution of the percentage of learnt clauses of size less than or equal to 8 on a particular family of 16 industrial instances *AProVE07_**. This limit represents the default static clause sharing limit the ManySAT parallel solver [Hamadi *et al.*, 2009].

This percentage is computed every 10000 conflicts, and as it can be observed, it decreases over time[1]. Initially, nearly 17% of the learnt clauses could be exchanged but as search goes on, this percentage falls below 4%. Our observation is very general and can be performed on different instances with similar or different clause size limits. It illustrates the second flaw reported above: the size of learnt clauses tends to increase over time. Consequently, in a parallel SAT setting, any static limit might lead to an halt of the clause sharing process. Therefore, if one wants to maintain a quantity of exchange

[1]The regular small raises are the result of the cyclic reduction of the learnt base through reduceDB.

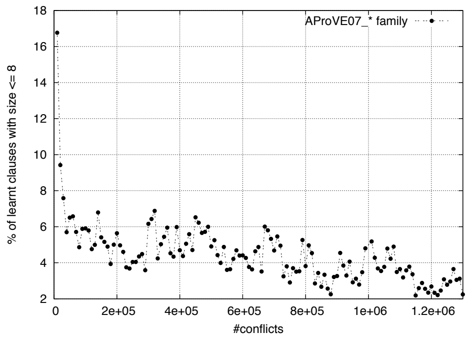


Figure 1: Evolution of the percentage of learnt-clauses with size ≤ 8

over time, there does not exist an optimal *static* policy for that. This clearly shows the importance of a dynamic control clause sharing policy.

3.2 Throughput and quality based control policies

In this section, we describe our dynamic control-based clause sharing policies which control the exchange between any pair of processing units through dynamic pairwise size limits.

The first policy controls the throughput of clause sharing. Each unit considers the number of foreign clauses received from other units. If this number is below/above a predefined throughput-threshold, the pairwise limits are all increased/decreased using an AIMD feedback algorithm. The second policy is an extension of the previous one. It introduces a measure of the quality of foreign clauses. With this information, the increase/decrease of the pairwise limits become proportional to the underlying quality of the clauses shared by each unit. The first (respectively second) policy allows the system to maintain a throughput (respectively throughput of better quality).

We consider a parallel SAT solver with n different processing units. Each unit u_i corresponds to a SAT solver with clause learning capabilities. Each solver can either work on a subspace of the original instance as in divide-and-conquer techniques, or on the full problem, as in ManySAT (see details in sections 5 and 4.1). We assume that these different units communicate through a shared memory (as in multicore architectures).

In our control strategy, we consider a control-time sequence as a set of steps t_k with $t_0 = 0$ and $t_k = t_{k-1} + \alpha$ where α is a constant representing the time window defined in term of number of conflicts. The step t_k of a given unit u_i corresponds to the conflict number $k \times \alpha$ encountered by the solver associated to u_i. In the sequel, when there is no ambiguity, we sometimes note t_k simply k. Then, each unit u_i can be defined as a sequence of states $S_i^k = (\mathcal{F}, \Delta_i^k, R_i^k)$, where $\mathcal{F}$ is a CNF formula, Δ_i^k the set of its proper learnt clauses and R_i^k the set of foreign clauses received from the other units between two consecutive steps $k-1$ and k. The different units achieve pairwise exchange using pairwise limits. Between two consecutive steps $k-1$ and k, a given unit u_i receives from all the other remaining units u_j where $0 \leq j < n$ and $j \neq i$ a set of learnt clauses $\Delta_{j\to i}^k$ of length less or equal to a size limit $e_{j\to i}^k$ i.e., $\Delta_{j\to i}^k = \{c \in \Delta_j^k / \, |c| \leq e_{j\to i}^k\}$. Then, the set R_i^k can be formally defined as $\cup_{0\leq j<n, j\neq i}\Delta_{j\to i}^k$.

Using a fixed throughput threshold T of shared clauses, we describe our control-based policies which allow each unit u_i to guide the evolution of the size limit $e_{j\to i}$ using an AIMD feedback mechanism.

Throughput based control

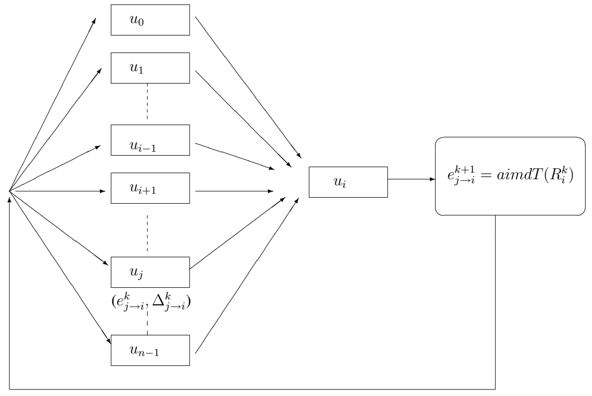
As illustrated in figure 2, at step k a given unit u_i checks whether the throughput is exceeded or not. if $|R_i^k| < T$ (respectively $|R_i^k| > T$) the size limit $e_{j\to i}^{k+1}$ is additively increased (respectively multiplicatively decreased). More formally, the upper bound $e_{j\to i}^{k+1}$ on the size of clauses that a solver j shares with the solver i between k and $k+1$ are changed using the following AIMD function:

$aimdT(R_i^k)\{$

$\forall j | 0 \leq j < n, j \neq i$

$$e_{j\to i}^{k+1} = \begin{cases} e_{j\to i}^k + \frac{b}{e_{j\to i}^k}, if(|R_i^k| < T) \\ e_{j\to i}^k - a \times e_{j\to i}^k, if(|R_i^k| > T) \end{cases}$$

$\}$ where a and b are positive constants.



u_0
u_1
u_{i-1}
u_{i+1}
u_j
$(e_{j\to i}^k; \Delta_{j\to i}^k)$
u_{n-1}
u_i
$e_{j\to i}^{k+1} = aimdT(R_i^k)$

Figure 2: Throughput based control policy

Throughput and quality based control

In this policy, to control the throughput of a given unit u_i, we introduce a quality measure $Q_{j\to i}^k$ (see definition 1) to estimate the relative quality of the clauses received by u_i from u_j. In the throughput and quality based control policy, the evolution of the size limit $e_{j\to i}^k$ is related to the estimated quality.

Our quality measure is defined using the activity of the variables at the basis of VSIDS heuristic [Moskewicz *et al.*, 2001] another important component of modern SAT solvers. The variables with greatest activity represent those involved in most of the (recent)-conflicts. Indeed, when a conflict occurs, the activity of the variables whose literals appear in the clauses encountered during the generation of a learnt clause are updated. The most active variables are those related to the current part of the search space. Consequently, our quality measure exploits these activities to quantify the relevance of a clause learnt by unit u_j to the current state of a given unit u_i. To define our quality measure, suppose that, at any time of the search process, we have $\mathcal{A}_i^{max}$ the current maximal activity of u_i's variables, and $\mathcal{A}_i(x)$ the current activity of a given variable x.

Definition 1 (Quality) *Let c be a clause and $\mathcal{L}_{\mathcal{A}_i}(c) = \{x/x \in c$ s.t. $\mathcal{A}_i(x) \geq \frac{\mathcal{A}_i^{max}}{2}\}$ the set of active literals of c with respect to unit u_i. We define $\mathcal{P}_{j\to i}^k = \{c/c \in \Delta_{j\to i}^k$ s.t. $|\mathcal{L}_{\mathcal{A}_i}(c)| \geq Q\}$ be the set of clauses received by i from j between steps $k-1$ and k with at least Q active literals. We define the quality of clauses sent by u_j to u_i at a given step k as $Q_{j\to i}^k = \frac{|\mathcal{P}_{j\to i}^k|+1}{|\Delta_{j\to i}^k|+1}$*

Our throughput and quality based control policy change the upper bound $e_{j\to i}^{k+1}$ on the size of clauses that a solver j shares with the solver i between k and $k+1$ using the following AIMD function:

$aimdTQ(R_i^k)\{$
$\forall j | 0 \leq j < n, j \neq i$

$$e_{j\to i}^{k+1} = \begin{cases} e_{j\to i}^k + (\frac{Q_{j\to i}^k}{100}) \times \frac{b}{e_{j\to i}^k}, if(|R_i^k| < T) \\ e_{j\to i}^k - (1 - \frac{Q_{j\to i}^k}{100}) \times a \times e_{j\to i}^k, if(|R_i^k| > T) \end{cases}$$

$\}$ where a and b are positive constants.

As shown by the AIMD function of the throughput and quality based control policy, the adjustment of the size limit depends on the quality of shared clauses. Indeed, as it can be seen from the above formula, when the exchange quality between u_j and u_i ($Q_{j\to i}^k$) tends to 100% (respectively 0%), then the increase in the limit size tends to be maximal (respectively minimal) while the decrease tends to be minimal (respectively maximal). Our aim in this second policy is to maintain a throughput of good quality. The rationale being that the information recently received from a particular source is qualitatively linked to the information which could be received from it in the very near future.

4 Evaluation

4.1 The parallel SAT solver

Our policies were implemented and tested on top of the ManySAT parallel SAT solver [Hamadi *et al.*, 2009] which won the parallel track of the 2008 SAT-Race[2]. ManySAT 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, it incorporates a new technique which extends the classical implication graph used during conflict-analysis to exploit the satisfied clauses of a formula [Audemard *et al.*, 2008].

Unlike others parallel SAT solvers, ManySAT does not implement a divide-and-conquer strategy based on some dynamic partitioning of the search space. At 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 and polarity heuristics, and learning schemes. Additionally, all the DPLLs are exchanging learnt clauses up to some static size limit.

[2]http://baldur.iti.uka.de/sat-race-2008/

4.2 Experiments

Our tests were done on Intel Xeon Quadcore machines with 16GB of RAM running at 2.3Ghz. We used a timeout of 1500 seconds for each problem. ManySAT was used with 4 DPLLs strategies each one running on a particular core (unit). To alleviate the effects of unpredictable threads scheduling, each problem was solved three times and the average was taken.

Our dynamic clause sharing policies were added to ManySAT and compared against ManySAT with its default static policy *ManySAT e=8* which exchanges clauses up to size 8. Remark that since each pairwise limit is read by a unit, and updated by another one, our proposal can be integrated without any lock.

We have selected $a = 0.125, b = 8$ for aimdT and aimdTQ, associated to a time window of $\alpha = 10000$ conflicts. The throughput T is set to $\frac{\alpha}{2}$ and the upper bound Q on the number of active literals per clause c is set to $\frac{|c|}{3}$ (see definition 1). Each pairwise limit $e_{j\to i}$ was initialized to 8.

4.3 Industrial problems

The Table 1 presents the results on the 100 industrial problems of the 2008 SAT-Race. The problem set contains families with several instances or individual instances.

From left to right we present, the family/instance name, the number of instances per family. Results associated to the standard ManySAT, with the number of problems solved before timeout, and the associated average runtime. The right part reports results for the two dynamic policies. For each dynamic policy we provide $\bar{e}$, the average of the $e_{j\to i}$ observed during the computation. The last row provides for each method, the total number of problems solved, and the cumulated runtime. For the dynamic policies, it also presents the average of the $\bar{e}$ values.

At that point we have to stress that the static policy ($e = 8$) is optimal in the way that it gives the best average performance on this set of problems. We can observe that the static policy solves 83 problems while the dynamic policies aimdT and aimdTQ solve respectively 86 and 89 problems. Except on the *ibm_** and *manol_** families, the dynamic policies always exhibit a runtime better or equivalent to the static one. Unsurprisingly, when the runtime is significant but does not drastically improve over the static policy, the values of $\bar{e}$ are often close to 8, i.e., equivalent to the static size limit. When we consider the last row, we can see that the aimdT is faster than the aimdTQ. However, this last policy solves more problems. We can explain this as follows. The quality-based policy intensifies the search by favoring the exchange of clauses related to the current exploration of each unit. This intensification leads to the resolution of more difficult problems. However, it increases the runtime on easier instances where a more diversified search is often more beneficial. Overall these results are very good since our dynamic policies are able to outperform the best possible static tuning.

		ManySAT e=8		ManySAT aimdT			ManySAT aimdTQ		
family/instance	#inst	#Solved	time(s)	#Solved	time(s)	$\bar{e}$	#Solved	time(s)	$\bar{e}$
ibm_*	20	19	**204**	19	218	7	19	286	6
manol_*	10	10	**117**	10	**117**	8	10	205	7
mizh_*	10	6	762	7	746	6	**10**	**441**	5
post_*	10	9	325	9	**316**	7	9	375	7
velev_*	10	8	585	8	**448**	5	8	517	7
een_*	5	5	2	5	2	8	5	2	7
simon_*	5	5	111	5	84	10	5	**59**	9
bmc_*	4	4	7	4	7	7	4	**6**	9
gold_*	4	1	1160	1	**1103**	12	1	1159	12
anbul_*	3	2	742	**3**	**211**	11	3	689	11
babic_*	3	3	2	3	2	8	3	2	8
schup_*	3	3	129	3	**120**	5	3	160	5
fuhs_*	2	2	90	2	**59**	11	2	77	10
grieu_*	2	1	783	1	**750**	8	1	**750**	8
narain_*	2	1	786	1	**776**	8	1	792	8
palac_*	2	2	20	2	**8**	3	2	54	7
aloul-chnl11-13	1	0	1500	0	1500	11	0	1500	10
jarvi-eq-atree-9	1	1	70	1	69	25	1	**43**	17
marijn-philips	1	0	1500	**1**	1133	34	1	**1132**	29
maris-s03-gripper11	1	1	11	1	11	10	1	11	8
vange-col-abb313gpia-9-c	1	0	1500	0	1500	12	0	1500	12
Total/(average)	100	83	10406	86	9180	(10.28)	89	9760	(9.61)

Table 1: SAT-Race 2008, industrial problems

4.4 Crafted problems

We present here results on the crafted category (201 problems) of the 2007 SAT-competition. These problems are hand made and many of them are designed to beat all the existing SAT solvers. It contains for example Quasi-group instances, forced random SAT instances, counting, ordering and pebbling instances, social golfer problems, etc.

The scatter plot (in log scale) given in figure 3 (left hand side) illustrates the comparative results of the static and dynamic throughput version of ManySAT. The x-axis (resp. y-axis) corresponds to the CPU time tx (resp. ty) obtained by ManySAT e=8 (resp. ManySAT aimdT). Each dot with (tx, ty) coordinates corresponds to a SAT instance. Dots below (resp. above) the diagonal indicate that ManySAT aimdT is faster (resp. slower) than ManySAT e=8. The results clearly exhibit that the throughput based policies outperform the static policy on the crafted category. These improvements are illustrated in the figure 3 (right hand side) which shows the time in seconds needed to solve a given number of instances ($\#instances$). We can observe that both aimdT ($\bar{e} = 23.14$ with a peak at 94) and aimdTQ ($\bar{e} = 26.17$ with a peak at 102) solve 7 more problems than the static policy. Like for the previous problem category, aimdT remains faster than aimdTQ. We can explain this as follows. It seems that since by definition, these problems do not have a structure which can be advantageously exploited by an intensification process, the higher diversification provided by aimdT allows better performances.

4.5 The dynamic of $\bar{e}$

It is interesting to consider the evolution of $\bar{e}$, the average of the $e_{j\rightarrow i}$ observed during the computation. In Figure 1 it was shown on the 16 instances of the *AProVE07_** family that the size of learnt clauses was increasing over time.

We present in figure 4 the evolution of $\bar{e}$ with ManySAT aimdT on the same family. The evolution is given for each unit (core0 to core3), and the average of the units is also presented (average). We can see that our dynamic policy overcomes the disappearing of “small” clauses by the incremental raising of the pairwise limits. It presents a typical saw tooth behavior that represents the probe for throughput T. This Figure and our results on the industrial and crafted problems show that the evolution of the pairwise limits if not bounded, does not reach unacceptable levels.

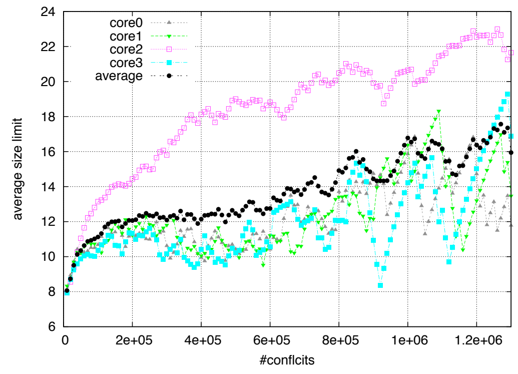


Figure 4: ManySAT aimdT : $\bar{e}$ on the AProVE07_* family

5 Previous works

We do not present parallel SAT solvers which predate the introduction of the modern DPLL architecture since these works did not use clause sharing.

The first parallel SAT solver based on a modern DPLL is Gradsat [Chrabakh and Wolski, 2003] which extends the zChaff solver with a master-slave model, and implements *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. Additionally, learned clauses are exchanged between all clients if they are smaller than a predefined limit on the number of literals.

[Blochinger *et al.*, 2003] uses an architecture similar to

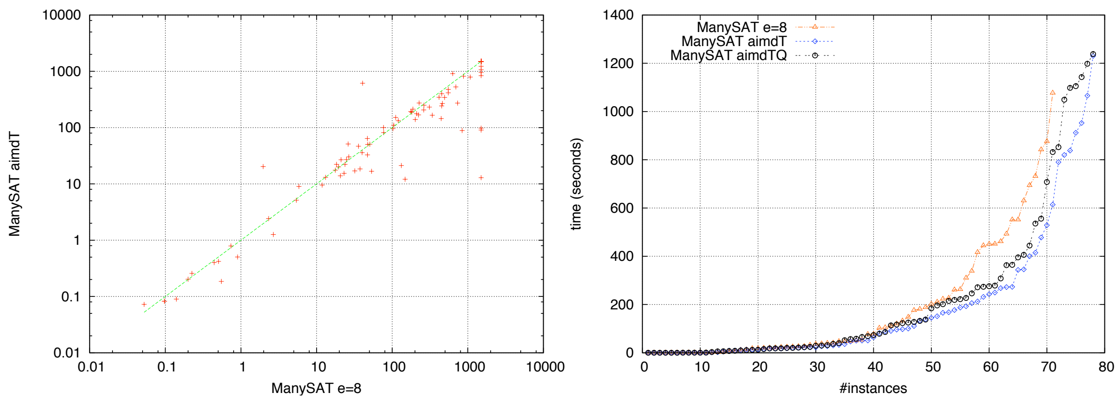


Figure 3: SAT-Competition 2007, crafted problems

Gradsat. However, a client incorporates a foreign clause if it is not subsumed by the current guiding-path constraints.

MiraXT is designed for shared memory multiprocessors systems [Lewis *et al.*, 2007]. It uses a divide and conquer approach where threads share a unique clause database which represents the original and the learnt clauses. Therefore in this system all the learnt clauses are shared.

pMiniSat uses a standard divide-and-conquer approach based on guiding-paths [Chu and Stuckey, 2008]. It exploits these paths to improve clause sharing. 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.

Let us remark that pMiniSat and MiraXT were respectively ranked second and third of the 2008 SAT-Race while ManySAT (presented in section 4.1) finished first.

6 Conclusion

We have presented two dynamic clause sharing policies for parallel SAT solving. They use an AIMD feedback control-based algorithm to dynamically adjust the size of shared clauses between any pair of processing units. Our first policy maintains an overall number of exchanged clauses (throughput) whereas the second additionally exploits the relevance quality of shared clauses. These policies have been devised as an efficient answer to the various flaws of the classical static size limit policy. The experimental results comparing our proposed dynamic policies against the static policy show important improvements for the state-of the-art parallel SAT solver ManySAT. It allows this solver to solve 6 more industrial and 7 more crafted problems. Our proposed framework opens interesting perspectives. For example, the design of new relevant quality measures for clause sharing is of great importance. It could benefit to sequential solver to improve their learnt base reduction strategy and as demonstrated by this work have an important impact in parallel SAT solving.

References

[Audemard *et al.*, 2008] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *Proc. of SAT*, pages 21–27, 2008.

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

[Chiu and Jain, 1989] D-M Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comp. Networks*, 17:1–14, 1989.

[Chrabakh and Wolski, 2003] W. Chrabakh and R. Wolski. GrADSAT: A parallel sat solver for the grid. Technical report, UCSB CS TR N. 2003-05, 2003.

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

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

[Eén and Sörensson, 2003] N. Eén and N. Sörensson. An extensible sat-solver. In *Proc. of SAT'03*, pages 502–518, 2003.

[Gomes *et al.*, 1998] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. AAAI'98*, pages 431–437, 1998.

[Hamadi *et al.*, 2009] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, to appear, 2009.

[Jacobson, 1988] V. Jacobson. Congestion avoidance and control. In *Proc. SIGCOMM '88*, pages 314–329, 1988.

[Lewis *et al.*, 2007] M. Lewis, T. Schubert, and B. Becker. Multithreaded sat solving. In *Proc. ASP DAC'07*, 2007.

[Marques-Silva and Sakallah, 1996] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proc. of IEEE/ACM CAD'96*, pages 220–227, 1996.

[Moskewicz *et al.*, 2001] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC'01*, pages 530–535, 2001.