Improving SAT Solver Efficiency Using a Multi-Core Approach

Ricardo Marques, Luis Guerra e Silva, Paulo Flores, L. Miguel Silveira
INESC-ID / IST - TU Lisbon
Rua Alves Redol 9, 1000-029 Lisbon, Portugal
{*rsm,lgs,pff,lms*}*@algos.inesc-id.pt*

Abstract

Many practical problems in multiple fields can be converted to a SAT problem, or a sequence of SAT problems, such that their solution immediately implies a solution to the original problem. Despite the enormous progress achieved over the last decade in the development of SAT solvers, there is strong demand for higher algorithm efficiency to solve harder and larger problems. The widespread availability of multi-core, shared memory parallel environments provides an opportunity for such improvements. In this paper we present our results on improving the effectiveness of standard SAT solvers on such architectures, through a portfolio approach. Multiple instances of the same basic solver using different heuristic strategies, for search-space exploration and problem analysis, share information and cooperate towards the solution of a given problem. Results from the application of our methodology to known problems from SAT competitions show relevant improvements over the state of the art and yield the promise of further advances.

1 Introduction

Over the last decade interest on Propositional Satisfiability (SAT) and SAT solvers has increased manifold. The SAT problem has long been one of the most studied problems in computer science since it was the first problem proven to be NP-complete. Nowadays, due to the enormous advances in computational SAT solvers, the practical importance of the SAT problem has extended to a wide range of disciplines. Many problems in many fields of science are now actively seeking a reformulation that will allow them to be amenable to analysis by state of the art SAT solvers.

One of the main reasons for this increased interest in SAT is the considerable efficiency improvement that SAT solvers have undergone in the past decade. Many real-life, industrial problems, with hundreds of thousands of variables and millions of clauses, are routinely solved within a few minutes by off the shelf, state of the art, SAT solvers. This impressive progress can be traced back to a substantial research investment in the topic, which has led to remarkable algorithmic improvements as well as significant progress in the ability of SAT solvers to exploit the hidden structure of many practical problems. However, this added capability is continuously challenged by emerging applications which bring to the fold problems and systems of increasing size and complexity. As a consequence, in spite of the remarkable gains we have seen in the area, there are still many problems that remain very challenging and unsolved by even the best solvers. Perhaps the main cause for this situation is that the large, steady algorithmic improvements that were made available in the last decade with the introduction of powerful techniques such as non-chronological backtracking, restarts, improvements in decision heuristics, etc, seem to have slowed down. Improvements to SAT solvers nowadays appear to be more incremental, sometimes problem-related. Faced with this situation, researchers have started to look elsewhere for ways to continue improving the efficiency of SAT solvers and the widespread availability of parallel computing platforms provided renewed opportunities to achieve this goal.

The generalization of multi-core processors, as well as the availability of fairly standard clustering software, provided access for the common user to parallel computing environments and has opened up new opportunities for improvement in many areas. In this context, many parallel SAT solvers have been proposed and SAT competitions now routinely include a parallel track. The main goal of parallel SAT solvers is to be able to solve problems faster. Several metrics can be used to quantify the resulting improvements, including speedup and efficiency. The basic obstacles to improved efficiency are generally the same ones as encountered by other parallel implementations, namely load balancing and robustness, i.e. the ability to sustain similar efficiency over a large range of problems. An enticing feature of parallelization in search-based problems is that super-linear speedups are achievable if one is lucky to search the right region of the search space. Often, one is satisfied with the ability to demonstrate speedup by solving problems faster and doing so in a robust manner over a large range of problems. Combining such robustness with the elusive possibility of large gains is clearly a goal worth pursuing.

In this paper we present PMCSAT, a portfolio-based multi-threaded, multi-core SAT solver which exploits a different approach to resource utilization. The general strategy pursued in PMCSAT is not entirely novel and has in fact been previously proposed in other SAT solvers (Hamadi, Jabbour, and Sais 2009). The idea is to launch multiple instances of the same (or different) solvers, sometimes called

a portfolio, with different parameter configurations, which cooperate to a certain degree by sharing relevant information when searching for a solution. This approach has the advantage of minimizing the dependence of current SAT solvers on specific parameter configurations chosen to regulate their heuristic behavior, namely the decision process on the choice of variables, on when and how to restart, on how to backtrack, etc. Instead of attempting to guess the optimal parameter configuration that better leads to the problem solution, we exploit multiple configurations in parallel, enforce some level of cooperation between them, and hope that one of them, with the help of the shared information, might find a solution faster. The set of parameter configurations chosen should be such that they represent complementary strategies. Each solver instance will attempt to find a solution to the problem or prove that no solution exists. To do so, it will use the information it gathers plus the information gathered and shared by others, which are concurrently attempting to find the same solution.

The remainder of this paper is organized as follows. Section 2 reviews core SAT solver techniques and summarizes several representative parallelization strategies previously presented. Section 3 details the proposed multi-core SAT approach. Section 4 presents experimental results, including comparisons with serial and other parallel implementations. Finally, Section 5 presents a few concluding remarks.

2 Background

The SAT problem consists in determining if there exists an assignment to the variables of a propositional logic formula such that the formula becomes satisfied. A problem to be handled by a SAT solver is usually specified in a conjunctive normal form (CNF) formula of propositional logic. A CNF formula is represented using Boolean variables, which can take the values 0 (*false*) or 1 (*true*). A clause is a disjunction (OR) of literals, which are either a variable or its complement. A CNF formula is a conjunction (AND) of clauses.

Basic SAT solvers are based on the *Davis-Putnam-Loveland-Logemann* (DPLL) algorithm (Davis, Logemann, and Loveland 1962), which improves over the simple assign, test and backtrack algorithm by using two simple rules at each search step: unit propagation and pure literal elimination. The unit propagation occurs when a clause contains only a single unassigned literal (unit clause). In order to satisfy the unit clause, no choice is necessary, since the value to be assigned to the variable is the value necessary to make the literal *true*. The pure literal elimination consists in determining if a propositional variable occurs with only one polarity in the formula. Such literals can always be assigned in a way that makes all clauses containing them *true*. Thus, these clauses no longer constrain the search and can be deleted.

During the search process a conflict can arise when both Boolean values have been tested on a given variable and, in both cases, the formula is not satisfied. In this situation the algorithm backtracks to a prior decision level, where some variable has yet to toggle its value. The idea of diagnosing the reason of the conflict and learning from it led to the *conflict-driven clause learning* (CDCL) algorithm (Silva and Sakallah 1996). Resolving the conflict implies the generation of new clauses that are learned. These learned clauses are added to the original propositional formula. This enables non-chronological backtracking, avoiding large parts of the search space if no solution is deemed to exist there. For this reason the CDCL algorithm is very effective and is the basis of most modern SAT solvers.

2.1 SAT Solver Techniques

In the following we provide a brief overview on the core techniques employed in modern SAT solvers.

Decision Heuristics Decision heuristics play a key role in the efficiency of SAT algorithms, since they determine which areas of the search space get explored first. A well chosen sequence of decisions may instantly yield a solution, while a poorly chosen one may require the entire search space to be explored before a solution is reached. Older SAT solvers would employ *static* decision heuristics, where variable selection was only based on the problem structure. Modern SAT solvers use *dynamic* decision heuristics, where variable selection is not only based on the problem structure, but also on the current search state. The most relevant of such decision heuristics is Variable State Independent Decaying Sum (VSIDS), introduced by CHAFF (Moskewicz et al. 2001), whereby decision variables are ordered based on their *activity*. Each variable has an associated activity, which is increased whenever that variable occurs in a recorded conflict clause. VSIDS, and similar activity-based heuristics, avoid scattering the search, by directing it to the most constrained parts of the formula. These techniques are particularly effective when dealing with large problems.

Non-Chronological Backtracking and Clause Recording When a conflict is identified, backtracking needs to be performed. *Chronological* backtracking simply undoes the previous decision, and associated implications, resuming the search afterwards. On the other hand, *non-chronological* backtracking, introduced by GRASP (Silva and Sakallah 1996), can undo several decisions, if they are deemed to be involved in the conflict. When a conflict is identified, a diagnosis procedure is executed, which builds a *conflict clause* encoding the origin of the conflict. That clause is recorded (learned), i.e. added to the problem. Backtracking is then performed, possibly undoing several decisions, until the newly-added conflict clause becomes unit (with only one free literal). While the immediate purpose of learned clauses is to drive non-chronological backtracking, they also enable future conflicts to show up earlier, thus significantly improving performance. However, clause recording slows down propagation, since more clauses must be analyzed and must therefore be carefully monitored. Therefore, modern SAT solvers periodically remove a number of learned clauses, deemed irrelevant by some heuristic.

Watched Literals Since any SAT algorithm relies extensively on accessing and manipulating large amounts of information, its data structures are of paramount importance for its overall performance. The single most effective improvement on the data structures of SAT algorithms was the introduction of *watched literals*, as proposed by

CHAFF (Moskewicz et al. 2001). During propagation, only unit clauses (with only one free literal) can be used to imply new variable assignments. For each clause, two free literals are selected to be watched. When a watched literal becomes false, the corresponding clause is analyzed to check whether it has become unit or if a new free literal should be watched instead of the previous one. When no other literal can be chosen to be watched this means that the clause is unit and that the remaining free literal is the other watched literal. This technique provides an efficient method to assess whether a clause has become unit or not, and to determine its free literal. One interesting advantage of this technique is that it does not require the watched literals associated with each clause to be changed when backtrack is performed.

Restarts SAT algorithms can exhibit a large variability in the time required to solve any particular problem instance. Indeed, huge performance differences can be observed when using different decision heuristics. This behavior was studied by (Gomes, Selman, and Kautz 1998) where it was noted that the runtime distributions for backtrack search SAT algorithms are characterized by heavy tails. Heavy tail behavior implies that, most often, the search can get stuck in a region of the search space. The introduction of *restarts* (Kautz et al. 2002) was proposed as a method of minimizing the effects of this problem. The restart strategy consists of defining a threshold value in the number of backtracks, and aborting a given run and starting a new run whenever that threshold value is reached. Randomization must also be incorporated into the decision heuristics, to avoid the same sequence of decisions to be repeated on every run. In order to preserve the completeness of the algorithm, the backtrack threshold value must be increased after every restart, thus enabling the entire search space to be explored, after a certain number of restarts. Restarts and activity-based decision heuristics are complementary, since the first one moves the search to a new region of the search space, while the second one enables the search to be focused in that new region.

2.2 Parallel Approaches

The usage of parallel computing environments, is a promising approach to speed-up the search for a solution, when compared to sequential SAT solvers. Moreover, parallel solvers should also be able to solve larger and more challenging problems (industrial problems) for which sequential SAT solvers are not able to find a solution in a reasonable time. Parallel implementations of SAT solvers can be divided in two main categories: cooperative and competitive SAT solvers. In the former, the search space is divided and each computational unit (either a core, a processor or a computer) searches for a solution in their subset of the search space. Often the workload balance between the different units is difficult to ensure. In the latter, each computational unit tries to solve the same SAT instance, but using alternative search paths. This is achieved by assigning different algorithms to each unit and/or using the same algorithm but with a different set of configuration parameters (portfolios). For this reason, this latter category is often called portfolio SAT-solution. In both categories the computational units can work collaboratively by sharing information about learned clauses to speed-up the search process. This requires communication between the processing units that may introduce some overhead. Deciding which clauses to share and when to share them, may have a significant impact on the time that a parallel SAT solver takes to find a solution.

Among all the parallel SAT solvers that have been developed over the past decade we present here some of the most noticeable approaches. A more complete overview of parallel SAT solvers can be found in (Martins, Manquinho, and Lynce 2010) or (Holldolber et al. 2011).

The PMSAT (Gil, Flores, and Silveira 2008) solver is based on MINISAT (En and Srensson 2004), which is a sequential SAT solver that implements most of the techniques used on advanced solvers. The PMSAT solver run on a cluster of computers (grid) using the Message Passing Interface (MPI) for communication and it is based on the master-slave approach with a fixed number of slaves. The search space is divided by the master using several partition heuristics. Each slave searches for a solution in a subset of the space and shares the selected learned clauses when reporting the solution to the master. Load balancing is implicility implemented by providing enough tasks to the slaves.

The C-SAT (Ohmura and Ueda 2009) SAT solver is also a master-slave parallelization approach of MINISAT based on MPI. The master is responsible for clause sharing and dynamic partitioning of the search space. The slaves work on subsets of the search space using different heuristics and random number seeds. Therefore, this solver, which runs on a network of computers, combines the search space splitting with a portfolio of algorithms for each subspace.

The PAMIRAXT (T. Schubert 2009) solver is another parallel SAT solver that follows the master-slave model and can run on any cluster of computers. Each slave is based on the MIRAXT (Schubert, Lewis, and Becker 2005) solver, which is a thread-based solver. The MIRAXT uses a divide-and-conquer approach where all threads share a unique clause database that includes learned clauses.

The YSAT (Feldmana, Dershowitz, and Hanna 2005) SAT solver is a shared memory solver that synchronizes the list of available tasks to minimize the number of idle threads. In this solver the formula and the tasks queue are globally accessible but the learned clauses are local to each thread. Due to the synchronization involved, the overall performance of the entire solver degrades as the number of cores increases.

MANYSAT (Hamadi, Jabbour, and Sais 2009) is a portfolio-based multithreaded solver that won the parallel track of the SAT Race 2008 (Sinz and et al 2008). Since then portfolio solvers became popular. In MANYSAT, built on top of MINISAT, there are four parallel instances with different restart, decision and learning heuristics. Additionally, sequential instance share clauses, with a given threshold size, to improve the overall system performance.

The SAT4J// (Martins, Manquinho, and Lynce 2010) solver is a hybrid solver that mixes competitive and cooperative search, but without clause sharing. The solver, implemented in Java, starts with a portfolio approach (weak portfolio) where a VSIDS-based heuristic determines the

variables with the highest activity. The solver then splits the search space based on such variables and, after a given number of iterations, it switches back to a portfolio approach (full portfolio).

Cooperative SAT solvers, through search space splitting, are one of the most used techniques to implement parallel SAT solvers. Although, recently there has been an increasing interest on the portfolio approaches, which have been shown to achieve very good performance (Hamadi, Jabbour, and Sais 2009), (Ohmura and Ueda 2009).

3 Multi-core SAT Solver: PMCSAT

PMCSAT is a MultiCore SAT solver based on portfolios. The solver uses multiple threads (eight currently) that explore the search space independently, following different paths, due to the way each thread is configured. However, this is not just a purely competitive solver because the threads cooperate by sharing the learned clauses resulting from conflict analysis. The underlying solver running on each thread is based on the MINISAT (En and Srensson 2004) sequential SAT solver, v2.2.0. The solver was however modified to support clause sharing and the ability to implement different heuristic schemes. In the following we briefly describe a few strategies that were adopted in the implementation of PMCSAT.

3.1 Decision Heuristics

Heuristics are used on SAT solvers for selecting the next variable to be assigned, and the corresponding value, when no further propagation can be done. Although some randomness is incorporated into most heuristics, we would like to keep a tight control over the search space explored by each thread. Therefore, we introduce the notion of variable priority. Initially variables are partitioned into sets, and each set is assigned a given priority. Variables in sets with higher priority are selected and assigned first. Within a given set, the well proven VSIDS heuristic is used to select variables.

In order to ensure that each thread follows divergent search paths, we defined distinct priority assignment schemes, one for each thread of the PMCSAT solver. Table 1 describes the eight priority schemes that were used. Note that, for most industrial SAT instances we can take advantage of the fact that the variables appear in the CNF file in a particular order, which is not random, but related to the problem structure.

Table 1: Priority assignment schemes for each thread.

Thread #	Variable Priority Assignment Scheme
0	All the variables have the same priority, therefore this thread mimics the original VSIDS heuristic.
1	The first half of the variables read from the file have higher priority than the second half.
2	The second half of the variables read from the file have higher priority than the first half.
3	The priority is sequentially decreased as the variables are read from the file.
4	The priority is sequentially increased as the variables are read from the file.
5	The priority is assigned randomly for each variable read from the file.
6	The priority is sequentially increased as the variables are read from the file, but it has a random component which can yield a priority increase of up-to 5x.
7	The priority is sequentially increased as the variables are read from the file, but it has a random component which can yield a priority increase of up-to 10x.

3.2 Lockless Clause Sharing

It is well established that clauses learned as a result of conflict analyses are vital to speed up the search process. In a parallel solver, the information learned from a conflict in one particular thread can be very useful to other threads, in order to prevent the same conflict to take place. Therefore, clause sharing between threads was implemented in PMCSAT. We limit the size of the clauses to be shared, to avoid the overhead of copying large clauses, which may contain very little relevant information. In (Hamadi, Jabbour, and Sais 2009), the authors show that the best overall performance is achieved with a maximum size of 8 literals per clause. Our experiments seems to corroborate these results and therefore we used the same limit. To reduce the communication overhead introduced by clause sharing, and its overall impact in performance, we designed data structures that eliminate the need for read and write locks. These structures are stored in shared memory, which is shared among all threads. We will consider that shared clauses are sent by a *source* thread and received by a *target* thread. As illustrated in Figure 1, each source thread owns a set of queues, one for each target thread, where the clauses to be shared are inserted. While this flexible structure enables sharing different clauses with different threads, we will restrict ourselves to sharing the same clauses with every thread. Therefore, every thread is a source thread and their target threads are all the others. On each queue, the *lastWrite* pointer marks the last clause to be inserted. The *lastWrite* pointer is only (atomically) written by the source thread, but can be read by each target thread. On the other hand, the *lastRead* pointer which marks the last clause received by the target thread, is only manipulated by each target thread. This data structure eliminates the need for a locking mechanism, since *lastRead* is only manipulated by one thread and even though *lastWrite* is read and written by different threads, the reading thread does not have to read its latest value. Clause sharing can occur after a conflict analysis.

4 Evaluation Results

In this section we present preliminary results from applying PMCSAT to a slew of problems gathered from a recent SAT race (Sinz and et al 2008). We pay particular attention

Table 2: Evaluation Results

Instance	Sol.	#Vars	#Clauses	MINISAT (sec)	PMCSAT (sec)	Winning Thread	Speedup PMCSAT vs no sh	Speedup PMCSAT vs MINISAT	MANYSAT (sec)	PLINGELING (sec)	Speedup PMCSAT vs MANYSAT	Speedup PMCSAT vs PLINGELING
cmu-bmc-barrel6	U	2306	8931	1.32	0.46	0,3,5,6	2.04	2.87	0.47	0.22	1.02	0.47
cmu-bmc-longmult13	U	6565	20438	26.27	7.12	0,1,4	3.06	3.69	7.38	12.78	1.04	1.79
cmu-bmc-longmult15	U	7807	24298	15.60	6.30	0,1	2.08	2.48	5.23	10.84	0.83	1.72
ibm-2002-11r1-k45	S	156626	633125	38.19	4.39	0,1,6	8.70	8.70	21.14	22.47	4.82	5.12
ibm-2002-18r-k90	S	175216	717086	102.30	56.71	1,6	1.80	1.80	82.33	71.51	1.45	1.26
ibm-2002-20r-k75	S	151202	619733	166.08	128.10	0	1.30	1.30	107.25	52.55	0.84	0.41
ibm-2002-22r-k60	U	208590	845248	716.53	554.06	0,2	1.29	1.29	187.18	118.68	0.34	0.21
ibm-2002-22r-k75	S	191166	793646	251.07	8.37	1,6	20.38	30.00	112.03	87.37	13.38	10.44
ibm-2002-22r-k80	S	203961	846921	159.03	19.74	1	2.90	8.06	133.66	119.83	6.77	6.07
ibm-2002-23r-k90	S	222291	922916	680.85	234.78	0	2.90	2.90	158.94	212.08	0.68	0.90
ibm-2002-24r3-k100	U	148043	545315	202.90	104.05	0	1.44	1.95	95.15	74.52	0.91	0.72
ibm-2002-30r-k85	S	181484	888663	850.73	823.43	0	1.03	1.03	248.12	224.22	0.30	0.27
ibm-2004-1_11-k80	S	262808	1023506	145.46	90.96	0,1	1.43	1.60	126.74	51.12	1.39	0.56
ibm-2004-23-k100	S	207606	847320	837.61	62.91	3	11.08	13.31	177.85	89.17	2.83	1.42
ibm-2004-23-k80	S	165606	672840	232.34	16.31	3	14.25	14.25	87.87	147.61	5.39	9.05
ibm-2004-29-k25	U	17494	74526	98.99	34.64	0	2.86	2.86	24.05	27.43	0.69	0.79
mizh-md5-47-3	S	65604	234719	265.53	21.20	0,1	12.53	12.53	68.23	55.71	3.22	2.63
mizh-md5-47-4	S	65604	234811	87.00	17.85	0,1	2.23	4.87	334.92	61.24	18.76	3.43
mizh-md5-47-5	S	65604	235061	563.58	53.09	1	2.44	10.62	56.83	34.39	1.07	0.65
mizh-md5-48-5	S	66892	240181	312.49	30.16	0	6.29	10.36	367.53	42.29	12.19	1.40
mizh-sha0-35-3	S	48689	173748	29.36	5.52	1,3	1.09	5.32	36.72	14.50	6.65	2.63
mizh-sha0-35-4	S	48689	173757	262.22	17.55	1	3.89	14.94	47.27	21.75	2.69	1.24
mizh-sha0-36-1	S	50073	179811	353.95	10.46	1,3	8.51	33.84	339.40	42.22	32.45	4.04
mizh-sha0-36-4	S	50073	179989	217.09	131.42	3	1.65	1.65	733.75	22.26	5.58	0.17
velev-engi-uns-1.0-4nd	U	7000	67553	10.83	4.89	0,1	2.09	2.21	7.25	14.86	1.48	3.04
velev-fvp-sat-3.0-b18	S	35853	1012240	27.05	3.03	0,1	8.94	8.94	2.86	4.59	0.95	1.52
velev-npe-1.0-9dlx-b71	S	889302	14582952	190.91	15.49	1	2.42	12.32	268.84	37.84	17.36	2.44
velev-vliw-sat-4.0-b4	S	520721	13348116	72.90	9.03	0,1,6,7	5.32	8.07	30.98	72.51	3.43	8.03
velev-vliw-sat-4.0-b8	S	521179	13378616	101.53	21.55	0,1	1.49	4.71	42.41	60.81	1.97	2.82
velev-vliw-uns-2.0-iq1	U	24604	261472	435.23	41.77	2,4	6.14	10.42	789.34	17.52	18.90	0.42
velev-vliw-uns-2.0-iq2	U	44095	542252	TO	416.14	2,4	2.31	NA	TO	303.33	NA	0.73
avg. table set (30 inst.)	–	155385	1790335	251.34	211.23	–	4.82	8.19	158.43	69.19	5.81	2.53
avg. full set (78 inst.)	–	168240	1119370	309.48	128.87	–	3.24	37.45	182.56	102.04	9.47	2.48

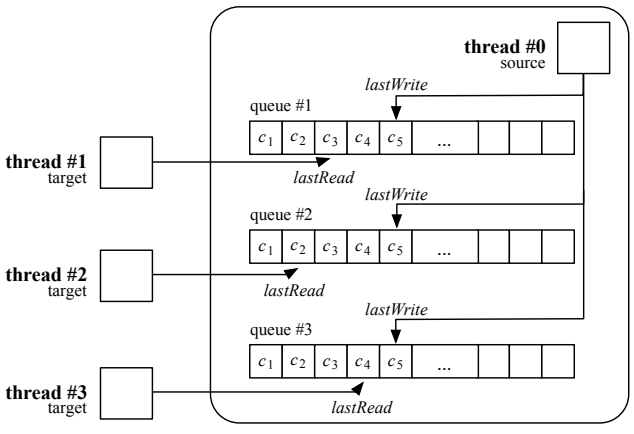

Figure 1: Data structures to share learned clauses.

to those originated from circuit examples, given their practical relevance in an industrial setting, which are shown in Table 2. These include problems which are both known to be SAT or UNSAT. We run additional examples which, for lack of space, are omitted but are accounted for on the averages shown in the last line of Table 2. All experiments were conducted on a machine with a Dual Intel Xeon Quad Core processor at 2.33GHz, 24GB of RAM and running Fedora Linux, release 17. In order, the columns of the table provide information about the problems chosen followed by runtimes for MINISAT (En and Srensson 2004) and PMCSAT using clause sharing. Also shown in the table is the information of which thread(s) of PMCSAT first found a solution for each of the problems. When sharing of learned clauses is turned off, the comparison reduces to determining which of the strategies, described in Table 1, is more appropriate to a given problem or set of problems. In essence in this case all threads are instances of the basic MiniSAT solver with the parameter configurations described. In this case, it turns out that the standard MiniSAT performs quite well but many other strategies do equally well, including random picking of variables. When sharing is turned on, the

scenario changes considerably and picking chunks of variables from the top or bottom of the order seems to do quite well on many occasions.

Next we show speedup computations to attest the potential gains of our solver. First we compare PMCSAT with clause sharing versus non-clause sharing. When clause sharing is turned off we are really testing the appropriateness of the strategies in Table 1. When clause sharing is on, we are measuring the advantages of cooperation between threads. The advantages of clause sharing seem obvious: using information from other threads, which are exploring problem structure elsewhere in the search space, provides relevant information and speeds up problem solution. However the advantage of this approach is also offset by the cost of doing the sharing (both preparing clauses for sharing, as well as using clauses originally from other threads). For this reason in certain problems little speedup is obtained. Next we compare PMCSAT with clause sharing versus the serial MINISAT. In general the speedups are interesting, with the average speedup larger than 8 (excellent efficiency) for the instances shown and an even larger speedup (over 37) for all instances. This really shows how sensitive problem solution is to variable ordering and to which portion of the space is searched. Next we show runtimes for MANYSAT (Hamadi, Jabbour, and Sais 2009) and PLINGELING (Biere 2010), solvers using approaches similar to PMCSAT, followed by the speedup comparisons. Again the results are quite interesting with good average speedups reported. These speedups clearly indicate that, as expected, there is substantial structure to be found on circuit descriptions and variable ordering. If this structural information can be obtained or guessed from the circuit itself, then a better ordering of variables during SAT solution can lead to relevant gains in problem solution. Overall, the results, seem very promising and justify further investment in adding new approaches to obtain further speedups.

5 Conclusion

The widespread availability of multi-core, shared memory parallel environments provides an opportunity for boosting the effectiveness of SAT solution. In this paper we presented a portfolio-based multi-core SAT solver to improve solution efficiency and capacity. Multiple instances of the same basic solver, using different heuristic strategies for search-space exploration and problem analysis, share conflict information towards the solution of a given problem. Results from the application of our methodology to known problems from SAT competitions, with practical importance, show relevant improvements over the state of the art and yield the promise of further advances.

Acknowledgment

This work was partially supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project "ParSat: Parallel Satisfiability Algorithms and its Applications" (PTDC/EIA-EIA/103532/2008) and project PEst-OE/EEI/LA0021/2011.

References

Biere, A. 2010. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report 10/1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Austria.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A Machine Program for Theorem-Proving. *Communications of the ACM* 5(7):394–397.

En, N., and Srensson, N. 2004. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing* 2919:502–518.

Feldmana, Y.; Dershowitz, N.; and Hanna, Z. 2005. Parallel multithreaded satisfiability solver: Design and implementation. In *Proceedings of International Workshop on Parallel and Distributed Methods in Verification (PDMC)*, volume 128 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, 75–90.

Gil, L.; Flores, P.; and Silveira, L. M. 2008. PMSat: a parallel version of MiniSAT. *JSAT* 6:71–98.

Gomes, C. P.; Selman, B.; and Kautz, H. 1998. Boosting Combinatorial Search Through Randomization. In *Proceedings of AAAI*, 431–437.

Hamadi, Y.; Jabbour, S.; and Sais, L. 2009. ManySAT: A Parallel SAT Solver. *JSAT* 6.

Holldolber, S.; Manthey, N.; Nguyen, V. H.; Stecklina, J.; and Steinke, P. 2011. A short overview on modern parallel sat-solvers. In *International Conference on Advanced Computer Science and Information System (ICACSIS)*, 201–206.

Kautz, H.; Horvitz, E.; Ruan, Y.; Gomes, C.; and Selman, B. 2002. Dynamic Restart Policies. In *Proceedings of AAAI*, 674–681.

Martins, R.; Manquinho, V.; and Lynce, I. 2010. Improving search space splitting for parallel sat solving. In *Proceedings of ICTAI*, 336–343.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC*, 530–535.

Ohmura, K., and Ueda, K. 2009. c-sat: A Parallel SAT Solver for Clusters. In Kullmann, O., ed., *Theory and Applications of Satisfiability Testing*, volume 5584 of *LNCS*. Springer Berlin / Heidelberg. 524–537.

Schubert, T.; Lewis, M.; and Becker, B. 2005. Pamira - a parallel sat solver with knowledge sharing. In *Proceedings of International Workshop on Microprocessor Test and Verification*, 29–36. IEEE Computer Society.

Silva, J. P. M., and Sakallah, K. A. 1996. GRASP: A New Search Algorithm for Satisfiability. In *Proceedings of ICCAD*, 220–227.

Sinz, C., and et al. 2008. SAT Race 2008. http://baldur.iti.uka.de/sat-race-2008/index.html. Accessed on May 2012.

T. Schubert, M. Lewis, B. B. 2009. Pamiraxt: Parallel sat solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation* 6:203–222.