A More Efficient Parallel Unit Propagation

Norbert Manthey

KRR Report 11-04

A More Efficient Parallel Unit Propagation

Norbert Manthey

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert@janeway.inf.tu-dresden.de

Abstract. This work extends early work about parallel unit propagation in SAT solvers. Since CSP solvers use SAT solvers as backend, parallel solving ships with parallel SAT solvers. The parallelization is based on splitting the formula into partitions. Each thread will propagate all literals only on its private partition and communicates the found implied literals with the other threads without using locks. The prototype implementation [12] mentioned some weaknesses that have been analyzed and improved in this work. The absence of load balancing is tackled by introducing two load balancing partition functions for the formula. The first method is based on the number of clauses per partition and results in a moderate speedup of 7 %. Distributing clauses based on the number of occurrences per variable results in a completely different search path and increases the number of solved instances from 61 to 67. The analysis of the implementations is based on five runs on the SAT Race 2010 benchmark per configuration. Still, there is no common way to compare parallel solvers, so that a pessimistic approach has been chosen in this work.

1 Introduction

The satisfiability problem(SAT) is an intensely studied problem in Computer Science. Due to the power of SAT solvers applications like planning, scheduling or cryptography ([10,1,16]), are solved in the domain of SAT. Furthermore, finite domain CSP solvers use SAT solvers as back-end and solve a CSP problem after they translated it into a SAT problem [17].

The introduction of the multi core architecture and the reduced increasing the CPU frequency force developers of SAT solvers to create parallel systems. A modern CPU has 4 to 6 cores and provides simultaneous multi threading. These cores need to be provided with work. Most parallel solvers follow a portfolio approach [6]. With this approach, the number of parallel running solvers is limited by the memory bandwidth. The number of future processors will increase further [3] and also GPUs yield highly parallel computation power.

In this paper the most time consuming part of the sequential SAT solving algorithm CDCL [15], namely the unit propagation (UP) is parallelized by extending the research of [12]. The algorithm splits the clause database into partitions so that each processor can work on its private clauses. The parallelization of a SAT solver also enables CSP solver to be parallelized, if they use the SAT solver as back end.

The improved algorithm is implemented in the CDCL solver *riss*[1][13] and the performance measurements use the benchmark of the SAT Race 2010. The parallel implementation has been run several times to get an average result. For comparing the results, a pessimistic evaluation is used, because the parallelization is fine grained and therefore it behaves randomly due to race conditions.

The parallel unit propagation is a technique that is orthogonal to all recently used parallelizations. The portfolio approach runs several solvers in parallel and clauses are exchanged. Search tree splitting creates sub formulas out of the original formula and solves each sub formula with a solver. Finally, also the variables can be split into sets and assignments have to be found for each set, such that finally the whole problem is either satisfied or unsatisfied by again using multiple solvers. The solvers that are used in all the three approaches can be extended by the parallel unit propagation and thus can be speed up the solving process further.

This paper is consequently structured in the following way. Important details of the parallelized UP are given in Section 2. The improvements for the parallelization are introduced in Section 3. Section 4 will focus on the results of the experiments. Finally, a conclusion and future work are given in Section 5.

2 Preliminaries

Specifying a SAT problem is done in conjunctive normal form (CNF). The description of the problem is given by a set of n propositional variables that are represented by natural numbers starting with 1. These variables can occur in literals positively or negatively, e.g. 2 respectively $\neg 2$. In addition to the variables, a problem is specified by a set of clauses F where a clause is a disjunction of literals. A clause is denoted by using square brackets, for example $[\neg 1, 2, \neg 3]$. The set of clauses is written by using angle brackets $F = \langle [1], [\neg 2] \rangle$. To solve a SAT problem, a mapping from the set of variables to true or false has to be found such that for every clause at least one of its literals is satisfied.

This work focuses on CDCL solvers [15], that is an extension of the DPLL procedure [4]. The main part of the runtime, about 80 %, of this algorithm is spent on unit propagation (UP) [7]. In the sequel it is assumed that the reader is familiar with the CDCL algorithm and the Two-Watched-Literal scheme. More details can be found in [2,14]. The definitions for reason clause and conflict clause are taken from [12] and also the requirements for UP will be used.

2.1 The Original Algorithm

The parallelization in [12] takes advantage of multiple cores without introducing twice as many additional memory accesses. Other recent systems, e.g. [11,6],

[1] The source code is available at `http://gitorious.org/riss`.

that use the portfolio approach copy most clauses or introduce expensive memory indirection. The parallel UP also takes advantage of the shared memory architecture by communicating via the lower cache hierarchy levels. Furthermore, no locks are needed to implement this algorithm. The work in [12] has already shown that this technique does not scale beyond 2 cores. Still, there are improvements that can be applied to the presented methods, although the algorithm itself is P-complete [9].

Number of new implied literals	Ratio
2	13 %
4	4 %

Fig. 1: Sequential unit propagation implementation

Measurements based on the SAT Competition 2009 application benchmark have shown that in praxis each literal that is propagated finds more than a single literal, that is added to the propagation queue. Table 1 shows the related data. If a literal is propagated, at least 2 more literals are found in 13 % of all propagation. For at least 4 new literals this number decreases to 4 %, that are already included in the 13 %. Since the new literals are implied by a reason clause, these literals can be found in parallel by searching the clauses in parallel.

```
0:traditionalPropagate(){
1:  C = 0;
2:  while( not myQueue.processed() ){
3:    l = myQueue.dequeue(); // keep on queue, remember last processed element
4:    C = propagate( l );    // enqueues implied literals
5:    if( C != 0 ) break;
6:  }
7:  return C;
8:}
```

Fig. 2: Sequential unit propagation implementation

Figure 2 shows the unit propagation, as it is implemented in state-of-the-art SAT solvers. Furthermore, there is only one queue, trail and assignment in the traditional solver. This scheme is also applied per thread in the parallel UP. The propagation uses the queue *myQueue* for literals that still need to be propagated. The initial queue contains only the decision literal. Afterwards, the presented algorithm is executed. As long as there are literals in the *myQueue* (line 2), these literals are propagated. In the traditional method, processed literals are removed from the queue when they are propagated. For the parallel version, each thread never removes literals from *myQueue* during propagation (line 3). Each thread simply stores the index of the last propagated literal. While propagating (line 4) a literal, further implied literals might be found. These literals are added to their queue and the reason clause is stored. If a conflict clause has been found, the propagation is interrupted and the conflict clause is returned (line 7). By returning 0, the method indicates that the propagation reached a fix point.

```
0:propagate(){
1: while( (not all finished) and (no conflict signaled)){
2:   C = traditionalPropagate();
3:   if( C != 0 ){ signalConflict( C ); break; }
4:   check all other threads for new literals;
5: }
6: if( (master) and (conflict not from master)){
7:  updateMaster();
8: }
9:}
```

Fig. 3: Parallel unit propagation implementation

For the parallel solver additional work has to be done. During the initialization of the solver, the clause database is split into partitions using a function *Assign: clause → partition* that assigns each clause to a partition. Each partition is assigned a thread T_i. If the number of threads is n than there are n partitions. A thread is only allowed to work on its own partition. The CDCL algorithm is executed by a single master thread that is allowed to read all clauses. For UP the slave threads are waked up from a sleep state. Additionally to the propagation itself, the communication among the threads has to be handled. The parallelized propagation, which is presented in Figure 3, is based on the traditional method. Now each thread executes the same algorithm: As long as there is any thread that still has to propagate literals and there is no conflict, all threads will not leave the propagation loop (line 1). Afterwards, the traditional propagation with all its consequences is executed on the threads partition. If a conflict occurred, this conflict is reported to all threads and the propagation is stopped. Otherwise, the private literal queue *myQueue* is now filled with implied literals that have been found by other threads (line 4). This can be done cheaply be reading all the literals of the queue of the other threads. Each literals is only enqueued once per thread. If all threads propagated all literals, the propagation reached a fix point. In case of a conflict, the trail of the master thread is updated (line 6-8). More details and a sketch of a correctness proof of this algorithm can be found in [12]. There are some weaknesses of the algorithm that have also been given:

1. There is a waiting time between the propagating threads
2. There is no load balancing of the learned clauses, especially after removals
3. The thread management uses about 2 % of the runtime
4. The approach does not scale beyond 2 cores

These first three weaknesses are tackled by introducing several heuristics to assign clauses and by improving the implementation of the thread management.

2.2 Parallel Algorithm Evaluation

Comparing parallel algorithms is very hard, because the individual runs cannot be reproduced due to race conditions. In [5] the portfolio approach has been extended to be reproducible by introducing synchronization points. This method cannot be applied to the parallel UP, because it would result in an almost sequential propagation process and all the benefits of the parallel execution would be lost. The parallel UP behaves more like a randomized sequential solver. Therefore, methods for randomized algorithms might be used.

In [8] the runtime distribution of randomized complete sequential solvers has been discussed. The analysis is based on multiple runs of the algorithm. The used randomization is based on the random seed that is used for random decisions.

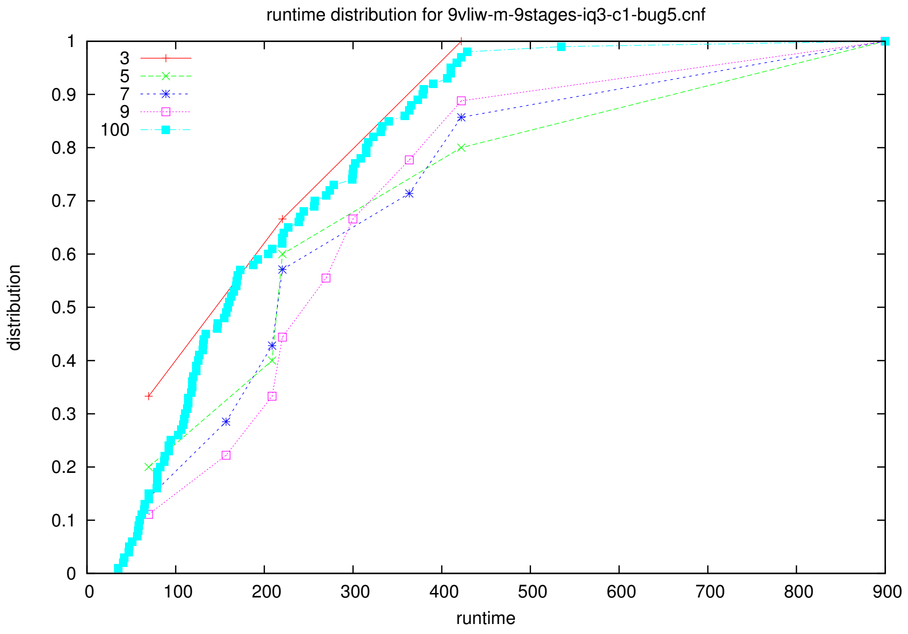

Fig. 4: Runtime distribution for 9vliw_m_9stages_iq3_C1_bug5.cnf, sampled by 100 runs

Evaluating parallel solvers can be done by looking at their runtime distribution as well. The instance 9vliw_m_9stages_iq3_c1_bug5.cnf from the SAT 2009 application benchmark is solved as an example and the distribution is given in Figure 4. Usually, the runtime of an instance is bounded by a t_{min} and t_{max}.

For the given instance, t_{min} is 30 second and t_{max} is 900 seconds, because on of the runs for this instance timed out. Within these bounds, each time is hit with a certain probability. Using the distribution, it can be stated that that the instance can be solved within a certain time by giving a probability. For the example, it can be stated that with 80 % the given instance can be solved within 320 seconds. By comparing these distributions one can compare the robustness of a parallelization on the one hand and on the other hand a solver clearly has a higher performance than another, if its distribution dominates the distribution of the other solver. Unfortunately, these distributions cannot be calculated by analyzing the solving algorithm and the instance. To obtain a distribution, the randomized solver has to be executed multiple times. From a practical point of view and to save resources, the number of runs should also be limited without introducing a big error.

Different clauses can be learned, since the order of reason clauses and propagated literals can be very different. From the first different learned clause, the search space that is examined by the solver will be different. Afterwards, the solving time cannot be predicted anymore. Both, super-linear speedups and slowdowns can be experienced. Thus, comparing a single run of the two algorithms is not appropriate.

Since it is not possible to run each configuration very often on each instance, due to limited computational resources, the distribution per instance is be sampled. Comparing two solvers on a given instance, the two distributions should be compared. To reduce the number of runs, only the median could be considered, because this value has the highest change to be hit. For a better comparison, a certain threshold probability should be introduced. Whenever a configuration is faster than another in many cases, this configuration should be considered the better configuration. If no clear winner can be determined, the according instance should be counted as a draw. By using this technique, the resulting comparison should yield better results than comparing only a small number of runs.

For the final comparison of our parallel algorithms we compare the highest runtime the algorithm would reach in 80 % of its executions. This percentage is a trade-off between not using too much resources for the experiments and giving a comparison that holds for most of the cases. The presented comparison is a pessimistic way, because it does not take into account that for 70 % of the executions algorithm A is faster than algorithm B, if algorithm A times out in the remaining 20 %. Of course, using another value can lead to different outcomes of the algorithm[2]. We claim, that using this value yields a smaller error than simply using the average of all the performed runs. Still, a study on the comparison and evaluation of parallel algorithms has to be done.

[2] All measurements of each configuration are provided at to repeat the measurement with another threshold at `http://www.ki.inf.tu-dresden.de/~norbert/paperdata/PMCS2011.html`

3 Parallel Unit Propagation

The improvements of the parallel UP are presented in the order the weaknesses have been presented in Section 2. Afterwards, the comparison method per thread is presented. Finally, results for the improvements are given in the next Section 4.

3.1 Introducing Load Balancing

The used *Assign* function in [12] simply distributes alternating. When a removal is executed, the size of the partitions can become unbalanced and the time to execute propagation might differ per thread. To avoid the waiting times of single threads that wait for others to finish their propagation, load balancing is introduced. The first approach is to spread learned clauses into partitions so that each partition has the similar size. The function $Assign_{size}$ applies this method.

Another method is to spread the clauses according to its literals. The clause is assigned the partition that has the least number of clauses with the literals of the clause. The motivation of this approach is to balance the execution times of the propagations for each literal better. The according function is called $Assign_{count}$.

3.2 Introducing Spinlocks

If other parts of the CDCL algorithm than UP are executed, the slave threads are not needed, because the other algorithms are not yet parallelized. Their waiting has been implemented by using semaphores. With this decision, the CPU time of the solver can be reduced, because waiting thread are set to a sleep state. For entering and leaving the sleep state, system calls need to be executed. The analysis in Section 4 showed, that 1.7 % of the execution time is spend in system calls. For 4 threads this number increases to almost 5 %. The system calls can be avoided by using spin locks. A waiting thread will not go to a sleep state but waits to continue its execution by waiting for a variable taking a certain value. The configuration *Busy* uses spin locks instead of semaphores for the thread management.

3.3 Avoiding Execution Overhead

Additionally to the weaknesses in Section 2.1, the parallel UP might execute unnecessary propagation steps, because a conflict has been already found by another thread. The first thread will still execute its private propagation until fixpoint. Therefore, in the traditional propagate method (Figure 2) a check can be added, that stops the algorithm in case of a conflict before the next literal is propagated. Thus, no unnecessary literals needs to be propagated. Another weakness is the recognition of conflicts. Assume the master thread has already propagated a variable with positive polarity. Without any synchronization, a slave thread might find that this variable has also to be propagated with negative polarity. This conflict will recognized only when the two threads perform

Configuration		CPU time	Wall time	Idle time	UP	System	Solved
Original	avg.	758.27	465.76	15.77	63.37	7.61	61
	median	189.22	151.19	1492	72	1.44	
$Assign_{size}$	avg.	696.83	433.29	47.72	62.63		60
	media	175.62	138.26	1232.5	71.5		
$Assign_{count}$	avg.	1028.88	598.86	294.78	68.98		67
	median	190.07	152.8	9371	76		
Fast	avg.	859.01	501.52	32.83	64.1		60
	median	229.60	157.61	2832	75		
Busy	avg.	834.35	465.90	16626.21	62.41	0.44	61
	median	194.59	153.34	1201	70	0.276017	

Table 1: Comparing measurements of parallel UP configurations

synchronization. If a slave thread checks the assignment of the master, the conflict can be found earlier and the propagation is also stopped faster. Checking all assignments of the slave threads is more expensive. Therefore, only a single check with the assignment of the master has been chosen. The name of the configuration, that implements these two combinations, is called *Fast*.

4 Experiments

The SAT solver *riss* [13] with the implemented parallel UP has been used for experiments and has been extended. The same settings as in [12] have been applied for running the original parallel UP. All the improvements have been also implemented into the solver. For the comparison the benchmark of the SAT Race 2010 has been used and a timeout of 3600 seconds has been applied. The used cluster operates on an AMD Opteron 285 CPU with 2.66 GHz and 2 GB main memory and thus has not the same power as the CPU that has been used for the original experiments. As suggested in Section 2, the 4^{th} slowest out of 5 runs is used for the comparison to compare the 80 %-threshold of the runtime. All the configurations are analyzed by using 2 threads.

The main results of the experiments are presented in Table 1. The table shows the configuration and gives the average and median value for the solved instances for the CPU time, the wall clock time and the system time (System) in seconds. It furthermore provides the idle time in seconds. This time measures the time that is spend by a thread that waits for the other thread to finish its execution or to find a new implied literal. The UP value gives the percentage of the time that is spend for UP during executing the search algorithm.

4.1 Selecting an Assign Function

The function *Assign* is initially set to spread original clauses and learned clauses alternating. Both the $Assign_{size}$ and $Assign_{count}$ function are compared against the sequential algorithm and the original parallel UP.

Comparing the solved instances, the clear winner is the $Assign_{count}$ balancing method. The $Assign_{size}$ can solve one instance less than the original configuration, but the average runtime decreases by 7 %.

The aim of the new load balancing procedures was to reduce the idle time. In average, balancing based on the number of clauses per partition does not seem to pay off. In fact, even more idle time is introduced. The $Assign_{count}$ method has an even worse picture. In average, a third of the CPU time is spend for waiting. Even the overall time of unit propagation increases by 4 % from 72 % to 76 %. For the load balancing with the $Assign_{size}$ function, this ratio decreases only slightly, but not significantly. Still, the idle time is higher.

A summary of this comparison can be drawn: The changed load balancing scheme can influence the direction of the search in the search tree dramatically. Although the performance of UP itself has been decreased, the $Assign_{size}$ function was able to solve more instances than the original version of the parallel UP. So far it is unclear, which effect the parallelized UP has to the performance of the SAT solver at all, because not only UP is improved but also the search tree is changed. By using load balancing with the $Assign_{size}$ method, 7 % runtime improvement can be gained.

4.2 Comparing Locking Techniques

Since the parallel UP reports 1.5 % system time, the thread management might be improve by using spin locks instead of semaphores. The original algorithm has been altered accordingly. Table 1 shows the comparison of these two synchronization techniques. In the measurements of this work, the percentage of the system time is 1.6 % for the original implementation and 0.07 % if spin-locks are used. The very small value shows that the implementation of the locks cannot be improved further. If spin locks are used, the overall performance of the solver does not increase significantly and the number of solved instances stays the same. The ratio of UP time to search time is slightly less than in the original implementation. This effect can be explained by the reduced latency. If semaphores are used, the slave threads are waked up sequentially. By using spin locks, each thread can start working immediately after new work has been given to it. A drawback of the spin locks is, that more CPU time is used than for semaphores, because even if the conflict analysis is performed by the master thread, the slave threads are running. The CPU time is still not twice the wall clock time, because before the search is started the preprocessor is executed sequentially. Thus, there is a part of the wall clock time that is only counted once for the CPU time.

4.3 Conflict Detection Approaches

The last improvement of this work is to be able to detect conflicts faster than in [12]. The configuration *Fast* has been compared with the original algorithm and the results are also presented in Table 1. It can be seen, that the number of solved instances differs only by one to the original approach. Both the CPU time and the wall clock time are higher than in the original algorithm. The average and the median of the percentage of UP to search and the idle time increased also slightly. One of the reasons for this behavior might be the implementation of the fast conflict detection. Since the slave thread reads the assignment of the master thread during each enqueue operation, this information has to be passed from one core of the CPU to another one by following a coherency protocol. Executing this protocol is time consuming. Thus, if the two threads are working on the same part of the assignment, additional waiting time is introduced, that has not been analyzed in detail so far. At the moment, this fast conflict detection cannot be regarded as an improvement.

5 Conclusion and Future Work

In this paper an extension of the parallel unit propagation is presented. The aim is to speed up the solving process further by tackling the weaknesses of the first algorithm. The main issue, load balancing, of the previous work was solved by introducing two schemes that split the formula into two partitions. The aim is to provide each thread with a set of clauses such that the communication among the threads is reduced. This work proposed two algorithms for this task. The first is based on the number of clauses in each partition and results in 7 % runtime improvement. The second partition function balances the clauses based on the variables of the clauses. Surprisingly, this method results in a better performance by solving 6 more instances although it decreases the performance of unit propagation. By chance, this algorithm seems to guide the search into a good direction for the selected benchmark. More detailed experiments should be carried out for this configuration.

Using spin locks instead of semaphores to suspend slave threads during the execution of conflict analysis reduces the time that is spend during operating system calls almost to zero seconds. However, the CPU time is increased due to busy waiting and no real improvement in the runtime is made. Thus, semaphores are still a good choice to reduce the CPU time that is needed to execute the algorithm. This result also indicates, that several parallel unit propagation solvers in a portfolio system do not need twice as many cores because the slave threads are not executed with 100 % load and thus cores could be shared.

Further improvements have been suggested by trying to reduce idle times with a faster global conflict detection. This approach failed and has to be analyzed further in more detail. The reason for the slower unit propagation performance can be the sharing of a single cache line among several cores in a CPU. A resource analysis of the parallel unit propagation and also of a portfolio based parallelization of the solver is considered future work.

As further future work, the presented approach will be combined with the current most used parallelization approach: the portfolio approach. The fact that using semaphores is already efficient points into this direction. Provided with a solver that can solve SAT instances in parallel and uses more than 4 cores, its utilization will be analyzed and improved for modern and future memory architectures. Scalability plays a huge role and is the main aim of our current research. Developing a SAT solver that can also be executed on massively parallel architecture is the next big goal of our work.

To reach this goal we also need to find a way to evaluate parallel algorithms with a non-deterministic behavior and to be able to state the significance of empirical results. To the knowledge of the author, there is no proposal how to study this kind of algorithms.

Acknowledgment The author would like to thank Holger Hoos for suggesting the distribution sampling to analyze the runtime behavior of the parallel unit propagation.

References

1. R. Béjar and F. Manyà. Solving the round robin problem using propositional logic. In *Procs. 17th National Conf. on Artificial Intelligence and 12th Conf. on Innovative Applications of Artificial Intelligence*, 2000.
2. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2009.
3. Intel Corporation. Intels Teraflops Research Chip. `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`, 2010.
4. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
5. Youssef Hamadi, Said Jabbour, Cdric Piette, and Lakhdar Sas. Deterministic parallel dpll: System description. In *Pragmatics of SAT(POS'11)*, jun 2011.
6. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.
7. Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware sat solvers. In Christian Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin / Heidelberg, 2010.
8. Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Strategies for solving SAT in Grids by randomized search. In Serge Autexier, John Campbell, Julio Rubio, Volker Sorge, Masakazu Suzuki, and Freek Wiedijk, editors, *AISC 2008*, volume 5144 of *Lecture Notes in Artificial Intelligence*, pages 125–140. Springer, 2008.
9. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *AI*, 45(3):275–286, Oct 1990.
10. H. Kautz and B. Selman. Planning as satisfiability. In *Procs. 10th European Conference on Artificial Intelligence*, 1992.
11. Stephan Kottler and Michael Kaufmann. Sartagnan - a parallel portfolio sat solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.
12. Norbert Manthey. Parallel SAT Solving - Using More Cores. In *Pragmatics of SAT(POS'11)*, 2011.
13. Norbert Manthey. Solver Submission of riss 1.0 to the SAT Competition 2011. Technical Report 1, Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, January 2011.
14. Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2004.

15. João P. Marques Silva and Karem A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.
16. Mate Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In *Pragmatics of SAT*, Edinburgh, Scotland, UK, July 2010.
17. Naoyuki Tamura and Mutsunori Banbara. Sugar: A csp to sat translator based on order encoding. `www.cril.univ-artois.fr/CPAI06/descriptionSolvers/Sugar.pdf`.