TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Artificial Intelligence Knowledge Representation and Reasoning

Parallel SAT Solving - Using More Cores

Norbert Manthey

KRR Report 11-02

Mail to
Technische Universität Dresden
01062 Dresden

Bulk mail to
Technische Universität Dresden
Helmholtzstr. 10
01069 Dresden

Office
Room 2006
Nöthnitzer Straße 46
01187 Dresden

Internet
http://www.wv.inf.tu-dresden.de

Parallel SAT Solving - Using More Cores

Norbert Manthey

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

Abstract. A parallelization approach for SAT solving is presented. Common parallel portfolio approaches seem to stagnate at four parallel solvers and cannot compete with the growing number of cores in the next generation of CPUs. The presented algorithm provides an opportunity to use additional cores by parallelizing the unit propagation. In average, unit propagation consumes 80% of the solvers runtime and thus provides a high potential to be parallelized. This load can be distributed among cores by splitting the clause database into partitions. The presented algorithm uses only a single lock for synchronizing conflicts. The parallelization is implemented into *riss*. A speedup of 1.57 can be reached by using two threads on the SAT Race 2010 benchmark in the best case, however, the performance of the parallelization is higher, because the sequential algorithm can solve only 48 instances within the timeout, whereas two threads can solve at least 61 out of 100 instances. Experiments on the SAT Competition 2009 Application benchmark showed that using four threads does not increase the solvers power. Still, by applying the presented technique to an existing solver, the number of used cores can be doubled.

1 Introduction

The satisfiability problem(SAT) is an intensely studied problem in Computer Science. Due to the power of SAT solvers, numerous applications, e.g. planning, scheduling or cryptography ([12,3,19]), are solved in the domain of SAT.

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 [9]. With this approach, the number of parallel running solvers is limited by the memory bandwidth, so that the number of used cores is also limited although there might be much more cores available in the near future [7].

In this paper a new techniques is presented which is able to extend existing solutions. The most time consuming part of the sequential SAT solving algorithm CDCL [17], namely the unit propagation (UP), uses 80% of the solvers runtime and is the component of the solver that provides the biggest potential to be parallelized [10]. The suggested parallelization is based on distributing the clause database into partitions with the result that each processor can work on its

private clauses. This approach can be combined with all existing parallel solvers, because it does not interfere with current parallelization techniques.

The algorithm is implemented in the CDCL solver *riss* [13] and its performance measurements have been done by using the benchmark of the SAT Race 2010 and a subset of the SAT Competition 2009 Application benchmark. The SAT Race instances and setup has been used to study the number of solved instances within a certain timeout, and the speedup on commonly solved instances. The parallel implementation has been run several times to get an average result. The SAT Competition benchmark has been used to analyze the algorithm in more detail, and to find weaknesses in the implementation.

The most notably result is the speedup of the algorithm. By combining the best runs, a speedup of 1.57 can be reached by using 2 processors. This value is very close to the theoretical optimum of 1.66, specified by Amdahl's law [2], which predicts the maximal possible speedup if a certain part of an algorithm is parallelized. Using Amdahl's law is not accurate, because parallelizing UP can provide a super linear speedup due to a changed search path.

From a practical point of view, the user of a SAT solver would solve a certain instance only once. The measured speedup for this scenario is 1.28. However, the speedup is only calculated for commonly solved instances. The performance of the parallelization is much higher, because it is able to solve 61 instances, where the sequential solver can solve only 48 instances. For a set of instances with medium difficulty, the speedup of the parallelization is close to 1 and only a few more instances can be solved on average. The used implementation has potential to be improved, because by using two threads, 2% of the runtime is spend for thread management (6% for four threads).

This paper is consequently structured in the following way. Important details of the sequential CDCL algorithm are given in Section 2. The parallelization is introduced in Section 3. Section 4 will focus on the results of the experiments. Finally, related work is discussed in Section 5 and a conclusion and future work are given in Section 6.

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.

For industrial benchmarks the CDCL [17] algorithm, which is an extension of the DPLL [8] procedure, is used. The main part of the runtime of this algorithm is spent on unit propagation (UP) [10], although an efficient scheme, namely

the Two-Watched-Literal unit propagation [14] is used already. 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 [4,16].

Watching the raising number of cores, any computing platform provides the ability to run parallel programs. An advantage of a multi core architecture is the fast communication among the cores. This communication is usually done via a fast accessible shared memory, e.g. the L2 cache. Furthermore, all the data from one core can be accessed by another core without the first core taking actively part in the communication. In contrast to Graphic Processing Units (GPUs), a multi core CPU has the ability to execute instructions on a core independently from the instructions on the other cores. These two properties are exploited in the parallelization of UP. Since UP uses most of the runtime [10], it provides the highest speedup potential if it is executed in parallel. The number of memory accesses is not multiplied, and thus the load on the memory bus is reduced compared to other parallelization techniques. Running independent parts of the solver on multi core CPUs enables the propagation to split the clause database into partitions. Consequently, each core can work on its private partition. The fast communication among the cores is exploited to share information about implied literals and the current conflict. This sharing is done lazily without blocking other cores. In the following, a multi core CPU is assumed to have n cores. On each core exactly one thread T_i $(1 \leq i \leq n)$ will be executed.

The CDCL algorithm has several requirements for the UP, namely:

Requirement 1 *The closure of literals that are implied by all the current decisions has to be propagated.*

Requirement 2 *The order of variable assignments is stored in the trail.*

Requirement 3 *The reason clause is given for all assignments (compare Definition 1 below) and is called reason.*

Requirement 4 *A conflict clause fulfills Definition 2 below. A conflict clause is called conflict.*

Requirement 1 ensures that all clauses are checked before a model is accepted to ensure completeness. Requirement 2 is necessary for modern implementations of the conflict analysis that use the order of the variable assignments to generate a 1st-UIP clause with linear resolution steps [16,22]. Furthermore, to generate a learned clause, the reason clause for an assigned variable is used for resolution (Requirement 3). Requirement 4 is needed to ensure that only learned clauses are generated from unsatisfied clauses with respect to the current assignment. In the sequel, the assignment, the trail and the reason information per assignments are called *search data*.

Definition 1 *A reason is a clause with a single satisfied literal and no unassigned literal with respect to the current search state. All unsatisfied literals have been assigned before the satisfied literal was assigned.*

Definition 2 *A conflict is a clause that contains only unsatisfied literals with respect to the current assignment.*

To get familiar with the data structures, an example is given with the formula $F = \langle[\neg 1, 2], [\neg 2, 3], [\neg 1, 4], [\neg 4, 5]\rangle$. The clauses are referred to by their indices; C_i represents the i-th clause in the formula. Let the algorithm choose the decision 1. Literal 1 will be enqueued to the propagation queue and the *propagate*()-method is called to process the watch list for this literal with the clauses $[\neg 1, 2]$ and $[\neg 1, 4]$. Both 2 and 4 will be enqueued with the corresponding reasons. Now the next literal, namely 2, is read from the queue and propagated. Literal 3 is enqueued with the reason C_2. The next literal on the propagation queue is 4 and it enqueues literal 5 with the reason C_4. Finally, the literals 3 and 5 are processed. Since their lists are empty reaches a fix point. The final trail is $\mathit{trail} = [1, 2, 4, 3, 5]$ (left to right).

3 Parallel Unit Propagation

The idea to parallelize UP is to split the clause database into partitions, such that each thread T_i gets a partition P_i. For a partition P_i, T_i is the only thread that has write access. Furthermore, all threads have their private watch lists, propagation queue, trail and reason information. Again, the owner of these data structures is the sole writer. It is assumed, that the implementations of these data structures except the watch lists allow multiple read accesses, even if the owner adds more data to them in the same moment. For simplicity of description, the implementation details of these structures are neglected. It is only important that reading and writing a single 32 bit data word is executed atomically on modern x86 architectures. Since only UP is parallelized, there is a single thread T_1, which executes the CDCL algorithm, including conflict analysis, decisions, restart and removal. The thread T_1 could also be seen as the master thread. The remaining threads $T_2 \ldots T_n$ could be referred to as slaves, because they are dependent on the execution of T_1. They wait for T_1 until UP has to be executed. To reduce power consumption, these threads sleep instead of performing busy waiting. The work during UP is spread among all threads, so that the separation in master and slave threads is not accurate. The major idea behind the proposed algorithm is that sequential unit propagation needs also to touch most of the clauses that are propagated using the parallelization and thus the same amount of memory accesses will be performed.

3.1 Sequential Execution

First, the new unit propagation is introduced for a single thread T_1, before the parallel variant is presented. The algorithm is visualized in Fig. 1 where all the colored boxes are the relevant steps for a sequential propagation. Since there is no other thread, the interaction among the threads is dropped and the fix point is reached after the queue has been processed without finding a conflict.

For a single thread there is no initial work to do. The whole clause database is handled by T_1. The search data for the CDCL algorithm is also maintained by T_1. The algorithm starts with enqueuing a literal that can come from either a

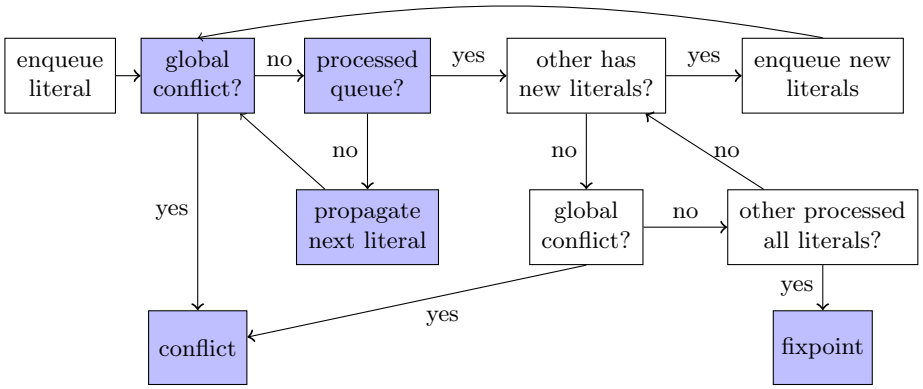

Fig. 1: Algorithm for a parallel propagate implementation

decision or a learned clause that is unit. Afterwards, the occurrence of a conflict is checked in order to return this conflict and stop the UP. Initially, there is no conflict. If no conflict has been found, the next element of the propagation queue will be propagated according to the Two-Watched-Literal scheme. During this step, more implied literals can be found. Finding an implied literal results in the following steps, which are executed in the specified order:

1. the corresponding variable is assigned such that this literal is satisfied
2. the reason clause is stored for this variable
3. the literal is added to the trail
4. the literal is added to the propagation queue

These steps ensure that the Requirements 2 and 3 are met. The loop of checking for the next literal and propagating the literal is executed until a conflict is detected, or until all enqueued literals have been propagated. A found conflict fulfills Definition 2 with respect to the assignment of T_1. Thus, Requirement 4 is also not violated. Requirement 1 cannot be seen from the propagation only. It can be explained incrementally. At the beginning of the search, there is only the first decision in the propagation queue. By propagating this decision, all implied literals will be found and also propagated because the whole propagation queue is processed. Finally, the first decision and all implied literals are stored on the trail and a fix point is reached. The next decision can be seen as applying it as first decision to the reduct of the formula with respect to the current trail. These steps can be repeated iteratively.

After UP has reached a fix point or a conflict, the propagation queue will be cleared, such that enqueuing the next decision can be done without having old literals in the queue. Since T_1 works on the search data that is also used during the other parts of the CDCL algorithm, no more work is necessary.

3.2 Parallel Execution

For a multi processor system with n cores, there will be threads T_1 to T_n. T_1 executes the whole CDCL algorithm where all the other threads work only during

UP. Therefore, T_1 works on the shared search data. Each of the remaining threads T_2 to T_n gets its private copy of the search data. During the initialization of the solver the clause database needs to be distributed by using a function *Assign:* $clause \rightarrow thread$ that assigns each clause C_j to a thread T_i. This function is also used to distribute learned clauses among the threads.

After the initialization, the propagation can be executed according to the whole algorithm shown in Fig. 1. Enqueuing a literal has to be done for each private propagation queue. Afterwards, all other slave threads are woken up and start executing the UP algorithm similarly to T_1. If any of the threads finds a conflict, the propagation stops and the conflict is processed further by T_1 in the conflict analysis. This case is explained in more detail in the next section. As long as there is no conflict, the propagation follows the sequential algorithm. Each thread propagates all implied literals of the enqueued decision with respect to its clause database partition. As in the sequential case, it can be shown that the four requirements of the CDCL algorithm hold for all threads.

Finding all Implied Literals A critical requirement for the correctness of this UP algorithm is to find the closure of implied literals, because certain literals might be implied only by clauses of T_i whereas the clauses of T_j imply other literals. To ensure, that T_i also enqueues and propagates all implied literals that have been found by T_j, T_i needs to check whether T_j found new literals. If there are new literals that have to be propagated, T_i needs to enqueue these literals with their reason information. To preserve the properties of reason clauses and to fulfill the Requirements 2 and 3, enqueuing the new literals has to be done exactly in the order T_j has found them[1]. To ensure the closure, each thread has to check each other thread for new literals that need to be propagated.

If T_i does not find new literals, it has to wait for all other threads to complete propagation. This waiting is implemented by a spin lock. Only if all threads shared and propagated all implied literals, propagation can be stopped. Otherwise, T_i might decide to stop propagating and T_j finds a new implied literal $\neg 5$ during propagating 2. In this case, T_i cannot check its clause partition for further literals that are implied by $\neg 5$. Furthermore, T_j might find a conflict during propagating literal 2 while T_i already signaled that there is no conflict and has stopped propagation.

Since all new literals are enqueued by each thread and since all threads wait for each other until all literals have been propagated, Requirement 1 is met. Propagating a single literal in a thread did not change from the sequential execution, so that the Requirements 2 and 3 are also fulfilled. Since T_i always adds literals from T_j in the order the literals have been found by T_j, they can also be used for the 1st-UIP conflict analysis. Naturally, the order of the literals of T_i and T_j can be different.

After a fix point is reached, or a conflict is found, all the threads except T_1 return to the sleep state until the next propagation. T_1 continues the algorithm with either the next decision or the conflict analysis.

[1] There might be alternatives, which are more complex than chosen heuristic.

Handling a Global Conflict The last Requirement 4 is not always met, since a conflict C can be found by thread T_2, although C contains unassigned literals with respect to the assignment of T_1. If C was given to the conflict analysis, the learned clause would not be a 1st-UIP clause. Therefore, the search data of T_1 needs to be extended, if another thread T_i finds the conflict. This extension is done by adding all implied literals from T_i to T_1 according to the four steps in Section 3.1. As soon as T_1 has enqueued all literals from T_i, the conflict C fulfills Definition 2 with respect to the search data of T_1. If enqueuing a literal, e.g. 7, found by T_i fails, another conflict clause C' is found, namely the reason of 7. This clause is a conflict clause, because all its literals except 7 are already unsatisfied, such that C' became the reason for the assignment of 7 in T_i and because of the synchronization also for T_1. In this case, 7 is unsatisfied with respect to T_1 as well. All other literals of C' are now unsatisfied with respect to T_1, because T_1 enqueued and assigned already all the other literals.

As soon as a conflict has been found by T_1 or when all literals of T_i have been enqueued and assigned, the propagation can be stopped and the propagation queues can be cleared again. Afterwards, T_1 can continue with the CDCL algorithm and run the conflict analysis, because the returned conflict is always a conflict with respect to T_1.

$$F = \langle [\neg 1, 2], [\neg 1, 4], [\neg 2, 3], [\neg 2, 5], [\neg 4, \neg 5] \rangle$$

T_1: $[\neg 1, 2], [\neg 2, 3], [\neg 4, \neg 5]$ $\qquad$ T_2: $[\neg 1, 4], [\neg 2, 5]$

T_1:	Reason	-	C_1	C_3		C_2	C_5					C_4
	Queue	1	2	3	sync	4	$\neg 5$				sync	$\lightning 5$
	Step	0	1		2	3	4	5	6	7	8	9
T_2:	Queue	1	4					sync	2 3 $\neg 5$	$\lightning 5$		
	Reason	-	C_2						C_1 C_3 C_5	C_4		

Fig. 2: Example for running UP in parallel

The example in Fig. 2 shows the parallel algorithm. The formula has five clauses to which we refer with their index. It is assumed that no variables are assigned. The algorithm decides to set variable 1 to true. This decision is enqueued in the two queues in step 0. In step 1 both threads propagate this decision on their private clauses. Found implied literals are also added to the queues and propagated. The master thread finds the literals 2 and 3 with the corresponding reason clauses and the slave thread finds literal 4 with C_2. In the next step the master thread synchronizes its queue with the slave's queue. The new literal 4 is added with its reason clause C_2. In step 4, this literal is propagated and the implied literal $\neg 5$ is also added to the private queue of the master thread with its reason clause C_5. Now the slave updates its queue and adds the literals 2, 3 and $\neg 5$. In step 7 the slave thread tries to propagate the new elements of its unit queue. While propagating the literal 2, clause C_4 requires literal 5 to be true. Since the complement is already assigned, C_4 is recognized as conflict

and set as global conflict. The master thread stops waiting for new clauses and updates the shared search data. Since the master already enqueued all literals that have been enqueued by the slave thread, C_4 is also unsatisfied with respect to the shared search data. Finally, the parallel propagation is stopped and the algorithm proceeds with conflict analysis.

3.3 Implementation Details

The most critical point in a parallel algorithm is blocking another threads' execution. The implementation of the presented algorithm needs only a single lock. This lock manages the write access on the only shared variable, namely the global conflict indicator. The remaining communication is lock free. It is not wait free, because blocking a single thread results in waiting for this thread by all the other threads to ensure that all threads propagate all implied variables.

The implementation is based on shared memory systems. A thread can read from propagation queues of other threads to update its search data. Whenever the size of a queue is increased, the element can always be read by other threads, because the element is written before updating the size. A literal is never removed from a queue, so that all threads can always read all literals that have been or are in this queue. If a thread T_i wants to synchronize its propagation queue with the new literals of T_j, it can simply read the size of T_j's queue and read all the literals and the corresponding reason information. This implementation is possible, because the write order to memory is not changed with respect to the algorithm. As stated in the four enqueue steps in Section 3.1, the reason information is written before the propagation queue is updated, so that the needed data is always available, before the queue size indicates new data.

The read on shared memory is also applied to check the state of other threads. If T_i wants to check whether all threads have already propagated all literals, it checks whether the size of all propagation queues is the same and whether all threads propagated all literal as indicated by their queue indexes.

4 Experiments

The parallel unit propagation has been implemented into the SAT solver *riss* [13]. The evaluation has been done in two stages. The UP algorithm used in [13] treats binary clauses specially. This is not done for the parallel UP. Instead, binary clauses are stored implicitly in the watch list [6]. The resource utilization of *riss* has been studied in [10]. Adding another thread that executes UP increases the load on the memory architecture. To reduce this load, the introduced prefetching has been disabled.

4.1 Instantiation of the Algorithm

The function *Assign* is set such that it distributes the clauses of the original formula in an alternating fashion. Learned clauses are also spread this way. As

Configuration	Seq1	D1	D2	D3	D4	D5	Q1	Q2	Q3	Q4	Q5
Threads	1	2 (*Dual*)					4 (*Quad*)				
Solved instances	38	41	38	38	38	40	40	38	39	41	39
Avg. user time (s)	1737	1565	1630	1589	1700	1596	1668	1634	1657	1714	1678
Avg. CPU time (s)	1737	2672	2776	2688	2927	2743	5196	5131	5136	5327	5177
Any instances	38	45					47				
All instances	38	30					27				
User time Std.Dev.	0%	32.98%					31.27%				
Propagations*	5.48	4.69	4.41	4.86	5.76	4.71	5.24	5.13	4.85	5.22	4.92
UP time	79%	78%	78%	76%	78%	78%	78%	79%	77%	78%	76%
Common	-	27					23				
All speedup	1	1.06					1.11				
Any speedup	1	1.10					1.05				
System time	0%	1.5%					5.32%				

Table 1: Scalability Experiment Summary; *in million

long as no removal is scheduled, this simple scheme should reach a good load balancing. After several removals it might be the case, that T_i has to propagate only original clauses and T_j still has all its learned clauses. This unbalanced load could be removed by re-distributing some clauses. There might be several approaches for the instantiation of the *Assign* function. For simplicity, the alternating scheme has been kept although load balancing might perform better.

The variable assignment order highly influences the performance of the search, because it determines the order of resolution steps during conflict analysis. Thus, different clauses are learned and the search continues in a different part of the search space. These different clauses and the different search space complicate the comparison of two parallel executions and can result in super linear speedup. To obtain a better result, parallel configurations are run several times.

4.2 Measuring Scalability

For finding a good number of threads for the algorithm, a subset of instances of the SAT Competition 2009 Application benchmark has been selected. The 49 instances, which could be solved between 15 and 120 minutes by either *riss*, the sequential algorithm or the parallel algorithm with two threads have been selected. For these instances, a configuration *Dual* with two thread and a configuration *Quad* with four threads have been tested with a timeout of 60 minutes. To overcome the repeatability problem of parallel programs, each configuration has been executed five times, resulting in runs *D1*, ..., *D5* with two threads and *Q1*, ..., *Q5* with four threads. Many measurements have been collected and are given in Table 1.[2]. The experiments have been executed on a cluster with AMD Opteron 285 CPUs from 2006.

The first comparison is done on the number of solved instances. On average, both parallel configurations can solve at least one instance more than the se-

[2] More details per instance can be found at `http://www.ki.inf.tu-dresden.de/~norbert/paperdata/POS2011.html`.

quential solver. The expected reduction of the user time (wall clock) by using another thread does not occur. Instead, the parallel configurations use almost as much time as Seq1. Comparing the time that has been really used for computation (CPU time), the values are not twice as much as the CPU time of Seq1 nor the wall clock time of the configuration itself, because during conflict analysis or preprocessing, only a single core is used. Comparing the number of instances, that could be solved by any solver with a certain configuration, it can be seen, that using more cores results in the higher number, namely 38, 45 and 47 with one, two and four threads. However, comparing the number of instances that have been solved by all solvers with the same configuration, the picture is the other way around. The effect can be explained with the high variance, namely a third of the average runtime of the parallel solvers.

Since the algorithm parallelizes UP, the number of method calls is also presented. Seq1 call this method most often, which leaves room for more discussion whether the parallel execution has a beneficial propagation order. By looking at the ratio of the time, the solvers spend for UP, it can be seen that the parallelization does not improve the performance of UP significantly. There is no relation between the number of calls and the relative time consumption. To calculate the speedup, only the mutually solved instances can be taken into account. Computing the speedup for instances that could be solved by all parallel configurations (All speedup), the value 1.06 for two threads is not much lower than 1.11 for four threads. For instances, that could be solved by at least one configuration (Any speedup), the difference is the other way around. Using two threads (1.1) outperforms using four threads (1.05). A reason for this effect could be the ratio the parallel solver spend in the system to manage the threads. On average, *Dual* spends 1.5% of its runtime in the system without proceeding with the algorithm but just waking up threads or stopping them again. This time almost quadruples to 5.32% for *Quad*. Based on this measurement it can be concluded that there will not be much more performance gain by adding more cores, if the implementation of the algorithm is not improved, for example by spin locks. Since the differences between using two or four cores are not significant, we claim that two threads are the most efficient instantiation for the algorithm on current computing platforms with the current implementation. Thus, a combination with other parallel approaches can double the number of used cores.

4.3 Measuring Performance

For comparing the system to the most recent competition, the remaining used heuristics of the solver are set to the same settings as used for the SAT Race 2010. The benchmark of the SAT Race 2010 with 100 instances is used to measure the performance of the parallel algorithm. The results are not comparable to the results of *riss* in the SAT Race, because the algorithm of the unit propagation has been changed. The used computing system is similar to the one used in the SAT Race. The CPU is an Intel Core 2 Quad Q9450 with 6MB L2 Cache and 4GB main memory. The timeout for the experiments has been set to 900 seconds. To compare the performance of the introduced approach, the algorithm

Configuration	Seq1	D1	D2	D3	Median
Solved instances	48	61	64	68	83
Average runtime	191.721	219.941	193.019	215.212	239.66

Table 2: Experiment Summary

Solved instances	Average speedup	Maximal speedup	D1 speedup	Median speedup
41	1.091	1.578	1.28	1.3

Table 3: Speedup on solved instances

is run with one thread and with two threads, because Section 4.2 showed that additional threads do not lead to major improvements. The repeated runs of the settings are limited because we do not have a cluster with these CPUs available.

For solving a single instance, the most interesting measurement is the runtime. However, a threshold is introduced to cut off instances that would take too long. Thus, there is another measurement, namely the number of instances that can be solved within a specified timeout. A user of the solver will only see the performance of a single run, because it is sufficient to find a solution for a given instance once. Therefore, this result needs to be treated specially.

Table 2 shows the runtime of the configurations. *Median* has been introduced to represent the median of the three configurations. Due to race conditions, differences in the parallel execution occur. The sequential configuration solves 48 instances whereas any parallel configuration solves at least 13 more instances. Usually, the average runtime increases with the number of solved instances, because more difficult instances have been solved. Surprisingly, D2 has solved more instances than D1 with a lower average runtime. This effect can be explained with the high standard deviation of the parallel runs on the same instance.

For analyzing speedup and efficiency, only instances that could be solved by all configurations can be used. The results are given in Table 3. All configurations solved 41 instances. Seq1 can solve 7 instances that have not been solved by all parallel solvers. On the other hand, another 20 instances can be solved by using a second thread. The average speedup on mutually solved instances is very close to 1 so that the efficiency is 0.5. Still, the overall efficiency of the presented approach is higher, because the parallel algorithm solved more instances. If always the fastest parallel run would occur, the average speedup would increase to 1.57. This is very close to the upper bound of 1.66 that is specified by Amdahl's law [2], if no super linear speedup could be obtained. Using only the first of the three runs, as it would be done in practise, results in a speedup of 1.28. The median solver can solve 45 instances that can also be solved by Seq1. The speedup of this solver is 1.3 and shows, that the parallelization yields a performance boost. Since unit propagation is an algorithm that is P-complete, it is hard to find a scalable paralellization with an efficiency close to 1. By tuning the *Assign* function and by introducing dynamic load balancing there is room to improve the presented work. Still, parallelizing UP helps to exploit a higher number of cores than dropping this technique.

5 Related Work

There are several approaches for solving SAT in parallel. The most commonly used approach during recent competitions is the portfolio based approach: Several configurations of a SAT solver are chosen and run in parallel. Achieving that the combination is faster than the best solver, learned clauses are shared among configurations. A well known implementation of this approach is ManySAT [9].

Dividing the search space based on guiding path is done in e.g. Pminisat [1] and c-sat [15]. After splitting, workers process the sub spaces and share information. Most of these approaches are implemented for computing clusters such that many CPUs can be used. Although this approach scales well with respect to the number of CPUs, its performance does not necessarily increase, because after splitting the resulting sub formulas are not always easier to solve. Furthermore, it is hard to prove unsatisfiability if one of the workers does not finish its work, for example because of a timeout.

To overcome these two issues [11] splits the search space as well, but also solves the original formula. This separation is repeated for all nodes in the tree that is used to split the search space. Thus, the parallel solver is at least as fast as solving the formula sequentially. Furthermore, if only a single worker fails to prove unsatisfiability it might be still possible to prove unsatisfiability based on the results of the original formula. Sharing clauses has not been enabled yet.

Similarly, the search space is split and explored by multiple threads in [21], where each thread that works on a node of the tree can either explore the whole subtree, apply UP once, or simply compute the next branching variable. The work steps are more fine grained, than in [11].

Another splitting technique has been used in [18]. The variables of the formula are split into two sets. The third set of variables is the intersection of these two sets. By finding all models for the first two sets it is possible to find a combination, that also satisfies the third set. A drawback of this solution is, that more than one model for the subsets has to be found.

6 Conclusion and Future Work

In this paper a new approach to parallelize SAT solving by splitting the clause database is presented. The aim is to speed up the solving process by utilizing more processing units. The parallelization is based on independent unit propagation among several processors. The main problems are to keep the order of propagated literals consistent and to keep soundness. To overcome these issues, the algorithm restricts the synchronization order and stops propagation only if all processors reached a fix point after propagating all implied literals.

The benchmark of the SAT Race 2010 has been used for experiments. The parallelization can solve at least 61 instances whereas the sequential run can solve only 48 instances. The measured speedup of 1.28 is a good starting point for further development. The presented algorithm is a powerful extension for recent parallelizations of SAT, because it can be integrated into common parallelizations, e.g. the portfolio and the search space splitting approach. On a

subset of the SAT Competition 2009 Application benchmark it has been shown that using more than 2 thread does not lead to additional improvements. Still, existing approaches can be extended by the presented technique to use twice as many cores as at the moment.

Finding a good distribution for the clause database is considered a hard task. The aim is to provide each thread with a set of clauses such that the communication among the threads can be reduced. The shared search data could be improved by the data that is found in all threads. One example for this improvement could be the selection of a shorter reason clause. Removing learned clauses should also result in load balancing with the result that all threads maintain a similar amount of clauses. These steps are considered future work.

Developing a state-of-the-art SAT solver is not easy if all the modern techniques are taken into account, e.g. [20,16]. Tools to test the solver can be used to find bugs in the solver [5]. However, having a proven SAT solver would be much more convenient, because in this case only the small extensions would have to be proved. Developing extensions would be much easier. It would also be possible to prove the correct behavior of the presented parallel unit propagation formally and thus in a way that is easier to understand.

More future work needs to be done by analyzing and improving the resource utilization of parallel SAT solvers. The presented approach has also to be combined with the current most used approach: the portfolio approach. Provided with a solver that can efficiently solve SAT instances in parallel, its utilization can be analyzed and improved for modern memory architectures. Scalability plays a huge role and is the main aim of our current research. We know that we will not reach a good scalability by only parallelizing UP, but it is a first nice extension for existing approaches.

Acknowledgment The author would like to thank Julian Stecklina for many hints and suggestions concerning the implementation. Furthermore, fruitful discussions in the ICCL SAT group at TU Dresden have been very helpful.

References

1. Chu, Geoffrey and Stuckey, Peter J. and Harwood, Aaron . Pminisat - a parallelization of minisat 2.0. `http://baldur.iti.uka.de/sat-race-2008/descriptions/solver_32.pdf`, 2008.
2. Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.
3. 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.
4. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
5. Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In Ofer Strichman and Stefan Szeider, editors,

SAT, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.
6. G. Chu, A. Harwood, and P. J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *JSAT*, 6:99–120, 2009.
7. Intel Corporation. Intels Teraflops Research Chip. `http://download.intel.com/pressroom/kits/Teraflops/Teraflops_Research_Chip_Overview.pdf`, 2010.
8. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
9. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.
10. 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.
11. Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 372–386, Berlin, Heidelberg, 2010. Springer-Verlag.
12. H. Kautz and B. Selman. Planning as satisfiability. In *Procs. 10th European Conference on Artificial Intelligence*, 1992.
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. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, pages 530–535, 2001.
15. Kei Ohmura and Kazunori Ueda. c-sat: A parallel sat solver for clusters. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 524–537. Springer, 2009.
16. Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2004.
17. 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.
18. Daniel Singer and Anthony Monnet. Jack-sat: a new parallel scheme to solve the satisfiability problem (sat) based on join-and-check. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 249–258, Berlin, Heidelberg, 2008. Springer-Verlag.
19. Mate Soos. Enhanced gaussian elimination in DPLL-based SAT solvers. In *Pragmatics of SAT*, Edinburgh, Scotland, UK, July 2010.
20. Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 237–243, Berlin, Heidelberg, 2009. Springer-Verlag.
21. Pascal Vander-Swalmen, Michaël Krajecki, and Gilles Dequen. Automatic parallel sat solving using mtss. In *High Performance Computing and Simulation*, Leipzig, June 2009. IEEE.
22. Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.