Parallel SAT Solving in Bounded Model Checking*

Erika Ábrahám[1,3], Tobias Schubert[1], Bernd Becker[1],
Martin Fränzle[2], and Christian Herde[2]

[1] Albert-Ludwigs-Universität Freiburg, Germany
[2] Carl von Ossietzky Universität Oldenburg, Germany
[3] RWTH Aachen, Germany

Abstract. Bounded Model Checking (BMC) is an incremental refutation technique to search for counterexamples of increasing length. The existence of a counterexample of a fixed length is expressed by a first-order logic formula that is checked for satisfiability using a suitable solver.

We apply communicating parallel solvers to check satisfiability of the BMC formulae. In contrast to other parallel solving techniques, our method does not parallelize the satisfiability check of a single formula, but the parallel solvers work on formulae for different counterexample lengths. We adapt the method of constraint sharing and replication of Shtrichman, originally developed for sequential BMC, to the parallel setting. Since the learning mechanism is now parallelized, it is not obvious whether there is a benefit from the concepts of Shtrichman in the parallel setting. We demonstrate on a number of benchmarks that adequate communication between the parallel solvers yields the desired results.

1 Introduction

The term *Bounded Model Checking* [16,8] refers to symbolic analysis techniques checking finite unravelings of transition systems for satisfaction of a formal specification. While originally being confined to a refutation technique based on an incremental search for counterexamples of increasing length, there are now several extensions to recognize fixed points and allow system verification (see e.g. [12]). Basically, given a system together with a specification, the existence of counterexamples of increasing length $k = 0, 1, \ldots$ is expressed by first-order formulae φ_k that are checked for satisfiability by a solver suitable for the underlying logic. For discrete systems a SAT solver is used, while the analysis of linear hybrid automata, for example, requires the application of a combined SAT-LP solver. Some popular solvers are, e.g., zChaff [24], BerkMin [15], MiniSAT [13], HySat [14], MathSAT [5], CVC Lite [6], and ICS [10].

Given the high computational cost of checking large BMC instances and driven by the advent of affordable multiprocessor machines, research recently focusses on the development of *parallel* BMC techniques, too. The main line of research applies parallel solvers to the *same* BMC instance, that means, the solvers work on the satisfiability

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

L. Brim et al. (Eds.): FMICS and PDMC 2006, LNCS 4346, pp. 301–315, 2007.

check of the same formula φ_k. Thereby, the overall search space is divided into disjoint parts which are then treated by the involved processes.

Parallel SAT algorithms can be traced back to at least 1994, where Böhm and Speckenmeyer presented an approach for a transputer system with up to 256 processors [9]. In subsequent years, a number of more advanced implementations have been developed. Two of the most powerful distributed SAT solvers nowadays are PaSAT [27] and PaMira [25]. Both tools use many of the latest improvements in sequential SAT solving, e.g., conflict-driven learning combined with non-chronological backtracking, various efficient decision heuristics, and zChaff's concept of watched literals. Additionally, PaMira employs *Early Conflict Detection BCP* and *Implication Queue Sorting* [20]. Both features together result in a significantly reduced number of clauses the BCP stage has to deal with, and by this substantially increase the overall performance.

PaSAT and PaMira also support the exchange of information about the problem instance under consideration, usually encoded as *conflict clauses*. In traditional parallel SAT solvers the processes independently generate conflict clauses for their own usage (in [27] also referred to as lemmas). Every conflict clause, generated by a conflicting assignment of the variables, is a piece of information that the corresponding process has learnt about the problem and that might be helpful to cut off parts of the search space. If the solvers share their knowledge, consisting of their conflict clauses, then this information enables them to avoid descending into parts of the search tree that have already been proven to be unsatisfiable by other solvers. If a solver, receiving a conflict clause, is currently analyzing such an unsatisfiable sub-tree then it can immediately stop its analysis of the current part of the search tree. Thus exchanging conflict clauses is helpful in increasing the performance of the overall system.

In this work we introduce a different kind of parallelization of BMC: instead of applying a distributed SAT solver to a single BMC instance, we do concurrently address the satisfiability check for *different* counterexample lengths through parallel solvers.

In a naive setting, without relating the SAT-checks of different BMC instances, such a parallelization would immediately provide ideal, *linear* speedup. Solvers optimized towards BMC do, however, exploit constraints of earlier SAT-checks to aggressively prune the search space of the subsequent ones (see e.g. [26,14,2]). These pruning techniques are developed for sequential execution. Our primary goal is to preserve linear speedup even when parallelizing such optimized BMC engines.

The BMC formulae φ_k for different ks describe similar problems, i.e., the formulae have some common sub-formulae. We make use of this fact and let the parallel solvers exchange information in the form of conflict clauses. However, knowledge sharing is not as simple as before: In parallel SAT solvers like PaSAT or PaMira, as described above, the solvers communicate conflict clauses in order to help each other with information which part of the state space does not need to be searched through. Without modification, this method does not work in our case when the different solvers have to solve different problems: a conflict clause, generated by some solver, may result from clauses that some of the other solvers do not have in the clause set they have to satisfy and thus those other solvers must not make use of that conflict clause.

In [14,2] we dealt with *constraint sharing and replication (CSR)* in the style of Shtrichman [26]. Recall, that sequential BMC defines a sequence of SAT problems

$\varphi_0, \varphi_1, \ldots$, which are checked sequentially one after the other. CSR can be seen as a method of communicating at the interface between the different BMC SAT problems: the conflict clauses after the SAT check of one problem are analyzed and used to prune the check of the following BMC problems when possible. Constraint sharing re-uses those conflict clauses whose generation involved only clauses that are part of the next SAT problem, too. Constraint replication shifts conflict clauses in time: if all clauses involved in the generation of a conflict clause are present in the next SAT check with the same variables but at different computation depth, i.e., with different indices, then we can insert that conflict clause after renaming the variables accordingly. Constraint replication can also be applied on-the-fly. In that case, shifted copies of new conflict clauses are added immediately after conflict resolution, when possible.

In the context of the AVACS project [1] we are interested in the analysis of linear hybrid automata. Linear hybrid automata are transition systems with mixed discrete-continuous behavior. Additionally to the discrete part, time passes while control stays in the locations, and the values of the real-valued variables evolve continuously according to some linear flows.[1] Consequently, the BMC formulae for linear hybrid automata are not only Boolean combinations of Boolean variables, but additionally of some linear constraints over the real-valued variables.

In [14,2] we showed that CSR speeds up the satisfiability checks remarkably not only for discrete systems (e.g. circuits) but also for linear hybrid automata. Our experience shows that in the mixed discrete-continuous case the search in the real domain, involving some LP solving techniques, is very time-consuming. Thus, for the hybrid case CSR is especially important to make use of the conflicts found in the real domain.

Now, if we start several SAT solvers running in parallel and solving BMC problems for different counterexample lengths *independently using CSR*, the expected speedup of CSR plus the linear speedup due to parallelization will not be reached. The reason is the following: In sequential BMC with CSR, each BMC instance is solved completely before the next check starts. As CSR re-uses the conflict clauses, the forthcoming check will not run into the same conflicts again. However, in the parallel case, if the different solvers do not communicate, they may find the same conflict independently, thus wasting time.

Even if the solvers communicate, yielding constraint sharing, we need to apply constraint replication to the communicated clauses immediately after communication, in order to get the same speedup as in the sequential case for each of the solvers. Without immediate constraint replication, the different solvers often find the same problem at different time instances. This entails on the one hand unnecessarily finding the same conflict twice, and on the other hand increased constraint propagation time: constraint replication at the beginning of the SAT checks may produce lots of subsumed clauses.

In this paper we integrate the standard parallel SAT-solving paradigm and CSR in the context of BMC. We parallelize the learning algorithm using communicating SAT solvers such that we can keep the speedup of the sequential case for each of the parallel solvers and at the same time, due to parallelization, the experiments show an additional linear speedup.

[1] The linearity of the flows allows us to express the BMC formulae in a decidable logic for which efficient solvers are available.

The rest of the paper is structured as follows: Section 2 deals with BMC and SAT solving, and Section 3 describes our parallelization technique. In Section 4 we present the experimental results, and finally we draw conclusions in Section 5.

2 Bounded Model Checking

We first give a short review of bounded model checking [16,8] and briefly describe how state-of-the-art SAT solvers check satisfiability of propositional formulae. In this paper we restrict ourselves to safety properties; for lifeness properties see e.g. [8].

2.1 Encoding Finite Transition Systems

Given a finite transition system, its initial condition and transition relation can be described by propositional formulae $Init(s)$ and $Trans_t(s, s')$ for all $t \in T$ with T the set of transitions, where s and s' explicitly denote the free variables occurring in the given formulae: $s = (v_0, \ldots, v_m)$ contains all variables and $s' = (v'_0, \ldots, v'_m)$ copies of them in order to describe the target valuation after a transition.

Let $Safe(s)$ be a propositional formula describing a safety property of the system. Counterexamples of a fixed length k, i.e., runs of length k violating the property $Safe$ in their final state, can be described by the following formula:

$$\varphi_k(s_0, \ldots, s_k) = Init(s_0) \wedge \left(\bigwedge_{i=0,\ldots,k-1} \bigvee_{t \in T} Trans_t(s_i, s_{i+1}) \right) \wedge \neg Safe(s_k) \; .$$

Starting with $k = 0$ and iteratively increasing $k \in \mathbb{N}$, BMC checks whether the BMC instances $\varphi_0, \varphi_1, \varphi_2, \ldots$ are satisfiable. The algorithm terminates at depth k if φ_k is satisfiable, i.e., an unsafe state is reachable from an initial state in k steps.

2.2 Satisfiability Checking

The formulae φ_k describing counterexamples of length k are checked for satisfiability by a traditional SAT solver.

First, the Boolean formula is transformed into a *conjunctive normal form* (CNF). In order to keep the formula as small as possible, auxiliary Boolean variables are used to build the CNF [30]. A formula in CNF-form is a conjunction of *clauses*, while each clause is the disjunction of *literals*. We distinguish between positive literals being Boolean variables, and negative literals being negated ones.

In order to satisfy the formula, each of the clauses must be satisfied, i.e., at least one literal in each clause must be true. The SAT solver *assigns values* to the variables in an iterative manner. After each *decision*, i.e., free choice of an assignment, the solver *propagates* the assignment by searching for *unit-clauses*, i.e., clauses in that all literals but one are already false; the remaining literal is implied to be true, since otherwise the clause would not be satisfied.

If two unit-clauses imply different values for the same variable, a *conflict* occurs. In this case a conflict analysis can take place which results in *nonchronological backtracking* and *conflict learning* [22,33]. Intuitively, the solver applies resolution to some

unit-clauses, using the implication tree, and inserts a new *conflict clause* thereby strengthening the problem constraints and restricting the state space for further search.

For performing the experiments of Section 4, we developed our own SAT solver, which – from a top level point of view – works quite similarly to zChaff [24] and BerkMin [15]. While not being as optimized as other state-of-the-art solvers, it incorporates most of the algorithms employed by modern SAT engines to accelerate the search process, like conflict-driven learning, non-chronological backtracking, and watched literals. The development of our own solver was necessary for our experiments, since there is no parallel solver available which supports constraints over the reals, as necessary for checking hybrid automata.

Our tool exploits the concept of lazy theorem proving [7] to provide a decision procedure for LinSAT formulae, i.e. CNFs where the atoms can be both propositional variables and linear inequations over the reals. It tightly integrates a Davis-Putnam style SAT solver with a linear programming (LP) routine, combining the virtues of both methods: LP adds the capability of solving large conjunctive systems of linear inequalities over the reals, whereas the SAT solver accounts for fast Boolean search and efficient handling of disjunctions (see e.g. [32,5,7,11]). The basic idea of the integration is to build a Boolean abstraction of the hybrid problem by replacing each non-propositional constraint occuring in the input formula by a fresh auxiliary Boolean variable. The SAT solver checks the satisfiability of a Boolean abstraction, while the LP solver checks the consistency of the assignments in the real domain.

2.3 Symmetries of BMC Problems

The formulae of BMC problems have a special structure: they describe computations, starting from an initial state, executing k transition steps, and leading to a state violating the specification. Accordingly, the set of clauses generated by the SAT solver can be grouped into clauses describing (1) the initial condition (*I-clauses*), (2) one of the transitions (*T-clauses*), and (3) the violation of the specification (*S-clauses*). Furthermore, the T-clauses can be grouped into k groups describing the k computation steps. Those k T-clause groups describe the same transition relation, but at different time points. That means, they are actually the same up to variable renaming. For example, some BMC problem for counterexample length $k = 3$ could be represented by a clause set like this:[2]

I-clauses	*T-clauses*	*S-clauses*
$(x_0 \vee y_0), \ldots$	$(x_0 \vee y_1 \vee \overline{z}_0), \ldots, (x_1 \vee \overline{y}_1 \vee z_0)$ $(x_1 \vee y_2 \vee \overline{z}_1), \ldots, (x_2 \vee \overline{y}_2 \vee z_1)$ $(x_2 \vee y_3 \vee \overline{z}_2), \ldots, (x_3 \vee \overline{y}_3 \vee z_2)$	$(y_3 \vee z_3), \ldots$

We say that a T-clause describing the ith transition step is a $\mathrm{T}_{[i-1,i]}$-clause, since it involves state-vector components with indices $i-1$ to i; we call $i-1$ the *lower boundary* and i the *upper boundary* of the clause. Similarly, I-clauses are also called $\mathrm{I}_{[0,0]}$-clauses

[2] The value of a Boolean variable v in the ith state of the computation is denoted by v_i, the value of its negation by $\overline{v}_i$.

and S-clauses in iteration k also $S_{[k,k]}$-clauses. We also write $T_{[i,\leq j]}$ when being unspecific about the upper boundary $i \leq j' \leq j$; we use similar notation for I- and S-clauses and lower boundaries. Furthermore we say that we *shift* a clause by d meaning that we replace each variable index i by $i + d$.

2.4 Constraint Sharing and Replication

Usually, the conflict clauses learned during the SAT check of a BMC instance φ_k get removed before the satisfiability check of the next BMC instance φ_{k+1}. However, they can also be partially re-used in the style of Shtrichman [26], thereby excluding search paths from the SAT search already before the search starts: If a conflict clause is the result of a resolution applied to clauses that are present also in the next BMC iteration, then the same resolution could be applied in the new setting, too, and thus we can keep those conflict clauses. Furthermore, if all clauses used for resolution to generate a conflict clause are present with a shifted instance, then the same resolution could be made using the shifted instances. Accordingly, we distinguish between the following conflict clause types:

- $I_{[0,j]}$*-conflict-clauses* are the result of resolution applied to $I_{[0,\leq j]}$- and possibly $T_{[\geq 0,\leq j]}$-(conflict-)clauses. They can be re-used without any modification in all iterations $k' \geq j$.
- $S_{[i,k]}$*-conflict-clauses* are the result of resolution applied to $S_{[\geq i,k]}$- and possibly $T_{[\geq i,\leq k]}$-(conflict-)clauses. They can be re-used in all iterations $k' \geq k - i$ when shifted by $k' - k$.
- $T_{[i,j]}$*-conflict-clauses* are the result of resolution applied to (conflict) clauses of type $T_{[\geq i,\leq j]}$. They can be inserted in iteration $k' \geq j - i$ in all instances shifted by $-i, \ldots, k' - j$.
- *IS-conflict-clauses* are the result of resolution applied to (conflict) clauses of both types I and S. They cannot be re-used in other iterations.

In a sequential setting, a single solver is used to check all the BMC formulae for incremental counterexample lengths. Thereby, the conflict clauses can be re-used in the above manner: before each iteration k, the conflict clauses generated in the iterations less than k are analyzed and adapted to the depth k. Alternatively, T-conflict-clauses can also be replicated on-the-fly directly after their generation, within the width of the current BMC instance.

2.5 Extension to Linear Hybrid Automata

The previously presented approach can be naturally extended to BMC of linear hybrid automata. *Hybrid automata* [3,18] are a formal model to describe systems with combined discrete and continuous behavior. We consider the class of *linear hybrid automata*, whose behavior can be described by Boolean combinations of linear (in)equations over real-valued variables.

Applying BMC, counterexamples of a linear hybrid automaton can be encoded similarly to that of a finite transition system. In the hybrid case the underlying logic is the existential fragment of the first-order logic over $(\mathbb{R}, +, <, 0, 1)$, i.e., formulae are the

Boolean combinations of (in)equations over linear terms using real-valued variables. The satisfiability check of those formulae is done by a combined SAT-LP solver. For a detailed description of the encodings, the satisfiability checks, and for optimizations see [2].

3 Parallel BMC

We are going to transfer the BMC technique into a parallel setting. Assume n solvers running in parallel, where each solver checks a different BMC instance of the same system for satisfiability. An additional master process makes the book-keeping of the BMC problems already checked and assigns the unresolved instances to the distributed solvers. When becoming idle due to completion of its previous instance, each solver asks the master process which counterexample length to check next. Starting at 0, the master distributes the problem instances in the order of increasing unraveling depth.

After receiving the first BMC instance to check, each solver generates the corresponding clause set and starts the satisfiability check.

If one of the solver runs into a conflict, it generates a conflict clause, which is then sent to the master process. In addition to the literals constituting the clause, the solver sends the conflict's type and boundaries. For example, the sequence of literals of a $\mathrm{T}_{[i,j]}$-conflict-clause is augmented with the information that it is a T-clause with boundaries i to j. IS-conflict-clauses are not sent, since they cannot be re-used in other iterations. After sending, the solver replicates the conflict clause, if possible, and the SAT algorithm continues. During replication, we remember which clause is the highest instance of the conflict, i.e., the one shifted by the highest value. To avoid multiple copies, only the highest instance will be shifted in future.

In order to benefit from conflict clauses generated by other solvers, each solver frequently fetches from the master new conflict clauses sent by the other solvers. Note that solvers receiving some clauses do actually process a BMC instance of another length than the sending solver does. Therefore, a solver checking iteration k may receive a $\mathrm{T}_{[i,j]}$-conflict-clause with $j - i > k$. In this case, the solver must not yet make use of the clause, but may of course memorize it for later use when it attacks a larger BMC instance.

Thus each receiving solver checks whether it can currently make use of the received conflict clause. If so, the clause and possible replications get inserted into the solver's clause set. S-clauses get shifted into the right position before insertion. Again, when replicating T-clauses, the highest instance is marked. It is important that received conflict clauses are replicated on-the-fly directly after their reception, and not later before the next satisfiability check: If different shifted instances of the same conflict are found or received, the solver would replicate all of them before the next check, resulting in multiple copies of the same clauses. That would increase propagation time.[3] Replicating on-the-fly alleviates this problem, since all processes insert all possible shifted instances within a small time frame, such that the probability that two solvers find the same conflict in the same or in a different instance is significantly reduced (see Section 4 for some experimental results).

[3] One could also employ subsumption checks to avoid this effect.

Otherwise, if the width of the conflict is too large for the current iteration of the receiver, the clause is inserted as a *silent* clause: though it is syntactically stored, it will not influence the current SAT check.

Before a solver starts a new iteration, it adds possible new replications of the already active T-conflict-clauses, adapts the S-conflict-clauses, and deletes all IS-conflict-clauses. Note that, in order to reduce subsumption, only the highest instances get shifted with all possible positive values, and the new highest instances get marked. Besides that, the solver checks which silent clauses may be activated and eventually replicated. In the above example, the solver may make use of the previously silent $\mathrm{T}_{[i,j]}$-conflict-clause when the new iteration k' that the solver is going to check is at least $j - i$. In this case the conflict clause shifted by $-i, \ldots, k' - j$ can be added to the clause set.

The above mechanisms aim at preserving linear speedup from parallelization even if constraint sharing is used, a mechanism that suits the sequential world better. Communication between the solvers has the role of knowledge transfer, such that none of the solvers has disadvantages from the fact that it does not compute each BMC instance incrementally, but skips over some that get computed by the others. In the next section we show by means of benchmarks that without exchanging conflict clauses, the effect of constraint sharing and replication in the parallel case does not yield the same speedup as in the sequential setting.

4 Experimental Results

We implemented a prototype SAT solver which works mainly as described in Section 2.2. For communication we use MPICH2 [17], an implementation of the Massage Passing Interface (MPI) standard. It is worth to mention that our approach works well only if communication is *very fast*. To satisfy this requirement, we designed the communication as simple as possible. It is very important that a process does not loose time by checking incoming messages if no message is there, or that it does not have to wait long when sending. Direct synchronous communication between the solvers seems to be disadvantageous, since the processes have to wait for each other. MPICH supports buffered, i.e., asynchronous communication, only in a restricted manner: there is a fast algorithm to check whether there have arrived some messages along some channels, but it can happen that a process must wait when sending if the buffer is full. For that reason, we introduce an additional master process acting as a communication hub and providing sufficient buffer capacity (see also [19,23] for such master-solutions). The master receives messages from the solvers, buffers copies for all other solvers, and forwards them when the solvers are ready to receive. This way communication can proceed without long waiting times. Thus we have one more process, but the solvers themselves need negligible time for communication (for experimental results see Figure 3).

For our experiments we used a network of 4 computers each having two AMD Opteron(tm) 250/252 processors with 2400-2600 MHz, 1024 kB L2 cache, and between 4 and 16 GB of main memory. We measured the relative performance of 1, 2, and 5 parallel solvers supported by a master process. We performed experiments with communicating solvers using constraint sharing and replication. To test the effectiveness

of these algorithmic enhancements, we also measured the running times when the solvers do not use constraint sharing and replication or when no communication takes place.

The running times (except in Figure 3) are given as the CPU times per processor for each BMC iteration. For each iteration, the CPU time per processor is computed as the runtime of that instance divided by the number of parallel solvers. Thus the depicted values correspond to the system time: the sum of the values for the iterations 0 to k is approximately the system time up to iteration k.

The running times contain the CPU times for the SAT checks including the communication times as well as the times needed for the generation of the BMC problems for the different counterexample lengths. We have run each experiment 4 times and show the average results below.

To give an example of a discrete benchmark, we applied BMC to check invariants of `UsbPhy` (Universal Serial Bus), taken from the VIS benchmark suite [28,31]. As for hybrid automata, we applied BMC to Fischer's mutual exclusion protocol [21] for 2, 3, and for 4 processes. The specification states the mutual exclusion property, i.e., that at each time point there is at most one process in its critical section. The Railroad Crossing [18] is a further hybrid benchmark. It consists of 3 parallel automata modeling a train, a railroad crossing gate, and a controller. The specification requires that the gate is always fully closed when the train is close to the railroad crossing. Further hybrid benchmarks are a model of an elastic approach to train distance control and a model of a Renault Clio 1.9 DTI RXE, equipped with a simple cruise controller as reported in [29].[4] To test the behavior also for deep counterexample search, we have chosen invariant safety properties, i.e., there exists no counterexample for the systems used, but for the elastic train train control at depth 22.

Figure 1 shows results for different settings for Fischer's protocol for 4 processes. Figure 1 a) motivates CSR by comparing the running times for a single solver with and without CSR. For all the benchmarks applied, CSR leads to substantially shorter running times.

Figure 1 b) shows what happens when parallelizing the solver with applying CSR but without communicating information between the solvers. I.e., CSR is only applied locally on a per-solver basis. The speedup due to parallelization is very small; the running times are sometimes even longer for the parallelized version than for the sequential setting. The reason is illustrated in Figure 1 c) by listing the number of conflicts in each iteration. Without communication, the number of conflicts may increase when employing more solvers. When a solver has computed a problem instance k and starts to compute a new instance $k' > k$, then during the computation of the new instance k' it will find conflicts that already occurred in other solvers computing instances between k and k'. However, since the solver computing k' is not informed about those conflicts, it may run into the same conflicts again.

To complete the picture, Figure 1 d) compares the running times when using CSR with and without communication between the solvers. Figure 1 e) shows the

[4] These two hybrid benchmarks are very complex in the real domain, i.e., the satisfiability check of their BMC instances requires a massive usage of the LP-solver. This yields, at least for deep BMC instances, long running times even for only a few conflicts.

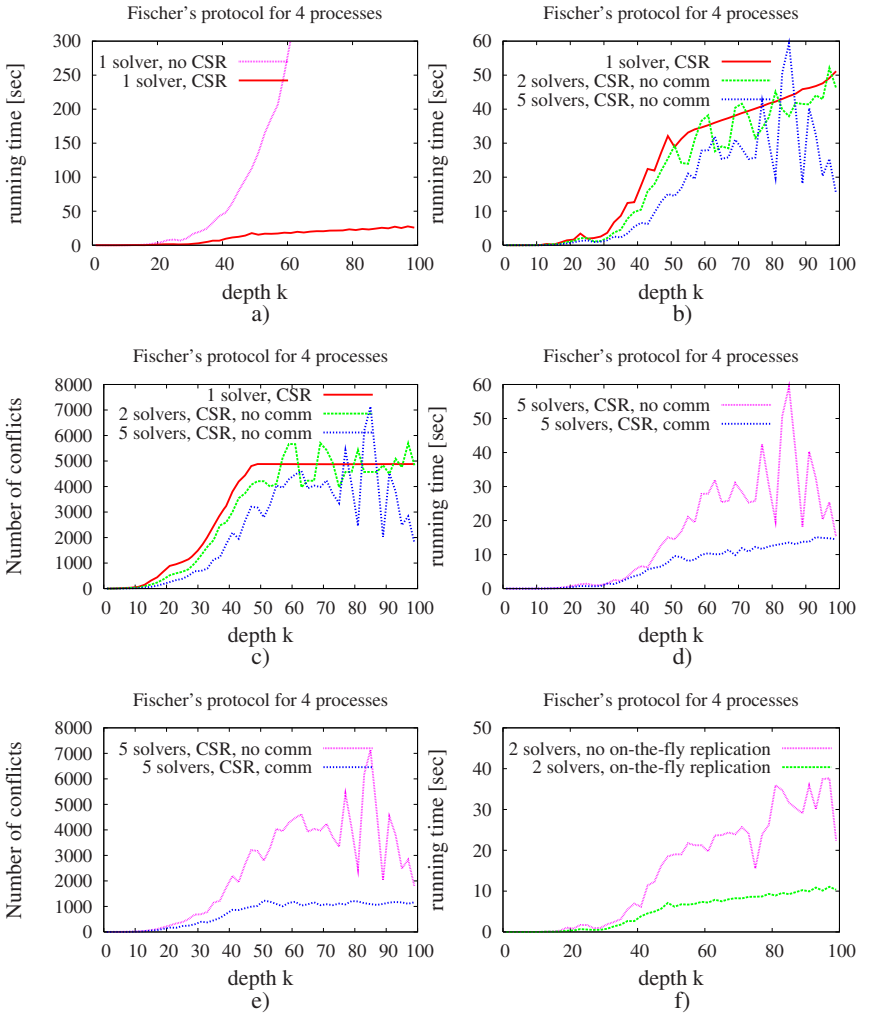

Fig. 1. CPU times per processor and iteration in different settings for Fischer's protocol for 4 processes

corresponding number of conflicts for each iteration. Communication between the solvers massively reduces the increment of the conflicts with a growing number of solvers. We observe that communication becomes more and more important when increasing the number of solvers, since the number of iterations that are skipped over by a solver after finishing instance k and before starting a new instance k' increases, and so the number of conflict clauses the solver did not get informed about. The last diagram f) of Figure 1 shows for two communicating solver using CSR the effect when not applying on-the-fly constraint replication, but replicating only at the beginning of a new SAT check.

Figure 2 shows the running times for the different benchmarks, when applying parallelized CSR and communication. We obtain, that the running times of the sequential

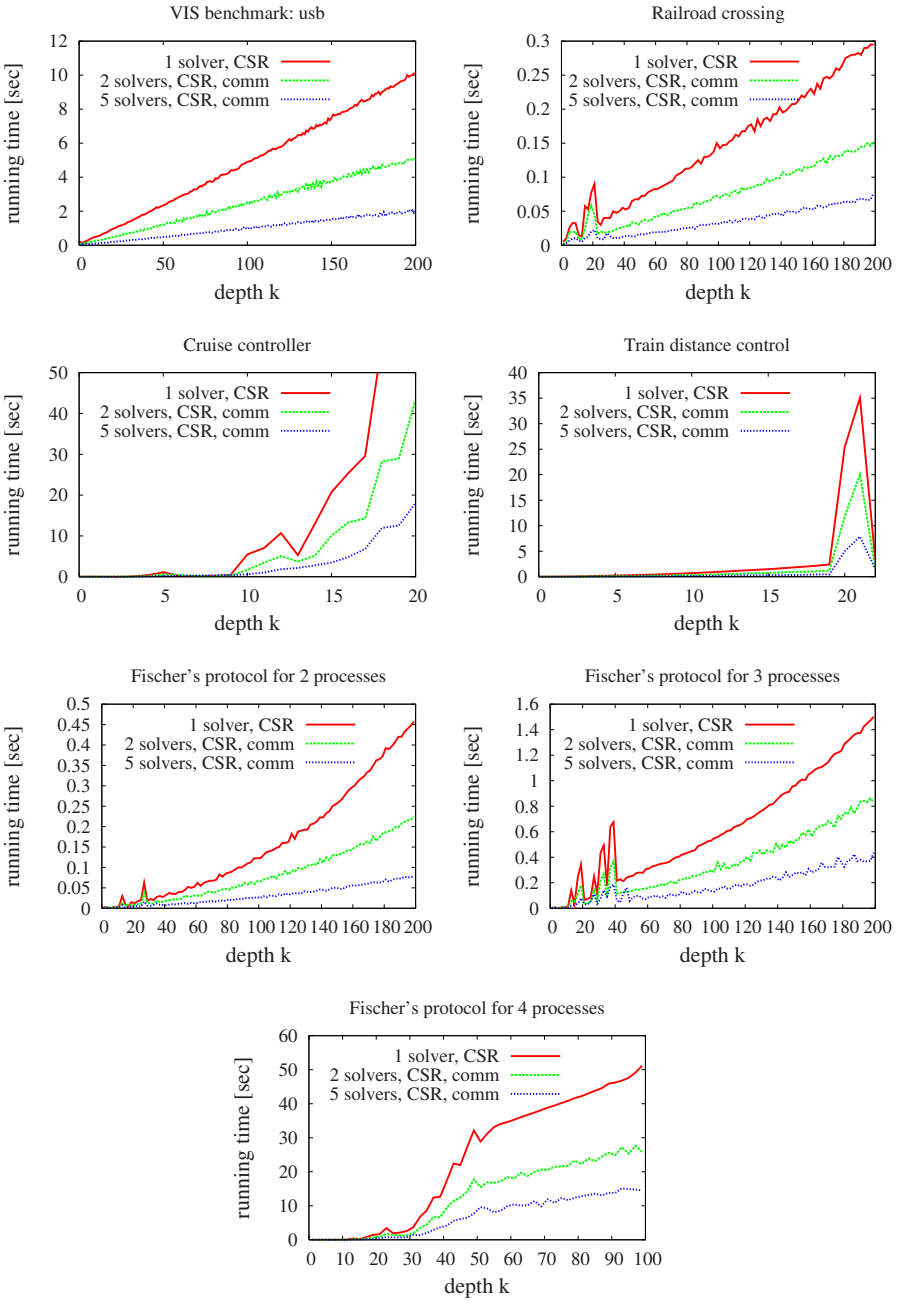

Fig. 2. CPU times per processor and iteration for different benchmarks with CSR and with communication

Bench-mark	depth k	M	1 solver		2 solvers		5 solvers		10 solvers	
			Time	C	Time	C	Time	C	Time	C
usb	200	1	973 (0)	13	487 (0)	13	201 (0)	13	99 (0)	13
		2	990 (0)	1	491 (0)	1	197 (0)	1	100 (0)	1
		3	– (–)	–	504 (< 1)	1	201 (< 1)	1	103 (< 1)	1
Railroad crossing	200	1	798 (0)	72	399 (0)	72	165 (0)	72	81 (0)	72
		2	14 (0)	3	8 (0)	5	7 (0)	10	10 (0)	17
		3	– (–)	–	7 (< 1)	3	3 (< 1)	4	1 (< 1)	6
Cruise controller	20	1	2254(0)	798	1129(0)	798	454 (0)	798	225 (0)	798
		2	317 (0)	262	278 (0)	275	183 (0)	387	148 (0)	582
		3	– (–)	–	158 (< 1)	240	67 (< 1)	235	37 (< 1)	240
Train distance control	22	1	120 (0)	9	62 (0)	9	27 (0)	9	13 (0)	9
		2	85 (0)	8	50 (0)	8	22 (0)	9	10 (0)	10
		3	– (–)	–	49 (< 1)	7	20 (< 1)	8	10 (< 1)	9
Fischer 2	200	1	242 (0)	258	121 (0)	258	47 (0)	258	23 (0)	258
		2	16 (0)	6	9 (0)	10	5 (0)	24	3 (0)	46
		3	– (–)	–	8 (< 1)	6	3 (< 1)	5	1 (< 1)	6
Fischer 3	200	1	5688(0)	10830	2815(0)	10830	1093(0)	10830	552 (0)	10830
		2	63 (0)	265	39 (0)	517	27 (0)	1222	24 (0)	2323
		3	– (–)	–	35 (1)	330	17 (2)	360	8 (3)	297
Fischer 4	100	1	*nn* (*nn*)	*nn*	*nn* (*nn*)	*nn*	3945(0)	43514	1971(0)	43514
		2	1186(0)	3268	1054(0)	6086	787 (0)	11944	820 (0)	23045
		3	– (–)	–	637 (11)	3289	341 (15)	3658	277 (9)	4259

Fig. 3. Total running times. Notations: Time=total SAT check time for all iterations up to depth k (average communication time per solver in brackets) in seconds, C=average number of conflicts per iteration, M=method. We distinguish the methods (1) without CSR and without communication, (2) with CSR but without communication, and (3) with CSR and with communication.

solver using CSR, which is already substantially shorter than without CSR, can be further improved by a linear factor when using our parallel approach with communication.

Finally, Figure 3 shows for each benchmark the total running times, i.e., the sum of the running times for all instances, up to the shown depth. We distinguish the methods (1) without CSR, (2) with CSR but without communication, and (3) with CSR and with communication. Besides the running times, the average total communication time per solver is given in brackets. Note, that the communication is very fast; for most benchmarks it amounts to less than 1 second. Additionally, the average number of conflicts per iteration is listed. Again, the results are presented for 1, 2, and for 5 solvers. To give an impression of the behavior for more massive parallelism, we also list the results for 10 parallel solvers. However, since we had only 8 CPUs but 11 processes (10 solvers and a master), the listed running times, i.e., the average CPU times per solver, do not correspond to the real system times observed, but state an estimated running time for 11 processors calculated based on measurements stemming from a time-sharing execution.

Since one single solver cannot communicate, the corresponding fields are not filled. By *nn* we denote that the computation was timed out because the total running time reached the timeout threshold of 10000 seconds.

To conclude the results, the experiments show that communication between parallel solvers using CSR yields an additional linear speedup to the improvement of CSR. Without communication, the effect of CSR gets worse with an increasing number of solvers.

5 Conclusions

In this paper we introduced a parallel SAT solving technique for bounded model checking. The parallelization is different from existing approaches: instead of solving a single problem instance using parallel solvers, we let different solvers solve different BMC instances. In order to speed up the search, we apply constraint sharing and replication and let the solvers communicate the conflict clauses found during their SAT checks. The experiments performed show that the positive effect of constraint sharing and replication can only be preserved under parallelization if the solvers communicate the conflict clauses among themselves. With communication, the full advantage of sequential CSR can be maintained in the concurrent setting, yielding linear speedup from parallelization.

The efficiency of other parallelization techniques strongly depends on how the state space gets split. The experiments show that our parallelization technique is stable: Each BMC instance is solved by a single solver, and thus we do not have to split the state space. With parallelized CSR we obtain a remarkable and consistent speedup for all benchmarks used. As for future work, we will compare our results with the application of other parallel SAT-solvers.

Furthermore, motivated by the positive findings obtained using our approach, we will combine both kinds of parallelization: on the one hand, instead of having a single solver checking each BMC instance, we will use a group of solvers for this purpose that work in parallel on the same problem by sharing the search space, similarly to PaSAT and PaMira. On the other hand, we will parallelize those solver groups as described in this paper to check for existence of counterexamples of different lengths. This combined approach we will build on the HySAT solver [14] which is more efficient than the prototype solver employed for these experiments.

Acknowledgements. We thank Marc Herbstritt for supplying us with benchmarks. We thank Henrik Bohnenkamp, Peter Schneider-Kamp, and Ivan Zapreev for their valueable comments on the paper.

References

1. AVACS: Automatic Verification and Analysis of Complex Systems. `http://www.avacs.org`.
2. E. Ábrahám, B. Becker, F. Klaedke, and M. Steffen. Optimizing bounded model checking for linear hybrid systems. In R. Cousot, editor, *Proc. of VMCAI'05*, volume 3385 of *LNCS*, pages 396–412. Springer-Verlag, 2005.
3. R. Alur, C. Courcoubetis, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
4. R. Alur and D. A. Peled, editors. *Procceedings of the 16th International Conference on Computer Aided Verification, CAV'04, Boston, MA, USA, July 13-17, 2004*, volume 3114 of *LNCS*. Springer-Verlag, 2004.
5. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In A. Voronkov, editor, *Proc. of CADE'02*, volume 2392 of *LNAI*. Springer-Verlag, 2002.

6. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Alur and Peled [4], pages 515–518.
7. C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Proc. of CAV'02*, 2002.
8. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.
9. M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver – Efficient Workload Balancing. *Annals of Mathematics and Artificial Intelligence*, 17(3–4):381–400, 1996.
10. L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In Alur and Peled [4], pages 162–174.
11. L. de Moura, H. Rueß, J. Rushby, and N. Shankar. Embedded deduction with ICS. In B. Martin, editor, *Proc. of HCSS'03*, 2003.
12. L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. J. Hunt and F. Somenzi, editors, *Proc. of CAV'03*, number 2725 in LNCS, pages 14–26. Springer-Verlag, 2003.
13. N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proc. of SAT'03*, volume 2919 of *LNCS*, pages 502–518. Springer-Verlag, 2003.
14. M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *ENTCS*, 133:119–137, 2005.
15. E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Proc. of DATE'02*, pages 142–149, 2002.
16. J. F. Groote, J. W. C. Koorn, and S. F. M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Proc. of Compass'95*, pages 57–68. National Institute of Standards and Technology, 1995.
17. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
18. T. Henzinger. The theory of hybrid automata. In *Proc. of LICS'96*, pages 278–292. IEEE, Computer Society Press, 1996.
19. F. Holmén, M. Leucker, and M. Lindström. UppDMC – a distributed model checker for fragments of the μ-calculus. In L. Brim and M. Leucker, editors, *Proc. of PDMC'04*, volume 128/3 of *Electronic Notes in Computer Science*. Elsevier Science Publishers, 2004.
20. M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.
21. N. Lynch. *Distributed Algorithms*. Kaufmann Publishers, 1996.
22. J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
23. I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in Eddy. In *Proc. of SPIN'06*, pages 108–125, 2006.
24. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Yang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of DAC'01*, pages 530–535, 2001.
25. T. Schubert, M. Lewis, and B. Becker. PaMira – a Parallel SAT Solver with Knowledge Sharing. In *6th International Workshop on Microprocessor Test and Verification*, 2005.
26. O. Shtrichman. Accelerating bounded model checking of safety formulas. *Formal Methods in System Design*, 24(1):5–24, 2004.
27. C. Sinz, W. Blochinger, and W. Küchlin. PaSAT – Parallel SAT-Checking with Lemma Exchange: Implementation and Applications. In *Proc. of LICS'01*, 2001.
28. The VIS Group. VIS: A system for verification and synthesis. In *Proc. of CAV'96*, volume 1102 of *LNCS*, pages 428–432. Springer-Verlag, 1996.

29. F. D. Torrisi. *Modeling and Reach-Set Computation for Analysis and Optimal Control of Discrete Hybrid Automata*. Doctoral dissertation, ETH Zürich, 2003.
30. G. Tseitin. On the complexity of derivations in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logics*. 1968.
31. VIS Benchmark Suite. `http://vlsi.colorado.edu/~vis`.
32. S. A. Wolfman and D. S. Weld. The LPSAT engine & its application to resource planning. In T. Dean, editor, *Proc. of 16th International Joint Conference on Artificial Intelligence*, pages 310–315, 1999.
33. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.