Parallelizing A Symbolic Compositional Model-Checking Algorithm

Ariel Cohen, Kedar S. Namjoshi[1⋆], Yaniv Sa'ar[2],
Lenore D. Zuck[3⋆⋆], and Katya I. Kisyova[3]

[1] Bell Labs, Alcatel-Lucent, Murray Hill, NJ. Email: kedar@research.bell-labs.com
[2] Weizmann Institute of Science, Rehovot, Israel. Email: yaniv.saar@weizmann.ac.il
[3] University of Illinois at Chicago, Chicago, IL. Emails: {lenore,kkisyo2}@cs.uic.edu

Abstract. We describe a parallel, symbolic, model-checking algorithm, built around a compositional reasoning method. The method constructs a collection of per-process (i.e., local) invariants, which together imply a desired global safety property. The local invariant computation is a simultaneous fixpoint evaluation, which easily lends itself to parallelization. Moreover, locality of reasoning helps limit both the frequency and the amount of cross-thread synchronization, leading to good parallel performance. Experimental results show that the parallelized computation can achieve substantial speed-up, with reasonably small memory overhead.

1 Introduction

The verification of concurrent programs remains an difficult task, in spite of numerous advances in model checking methods. The main difficulty is state explosion: the verification question is PSPACE-hard in the number of components. In practice, this means that the size of the reachable state space can be exponential in the number of processes.

Compositional reasoning and other abstraction approaches can ameliorate the effects of state explosion. In this work, we point out that compositional reasoning is also particularly amenable to parallelization. In compositional reasoning, each process is analyzed separately, and the information exchanged between processes is limited by the localized nature of the analysis. Both factors are crucial to effective parallelization.

To the best of our knowledge, this is the first work to explore parallel model checking based on automatic compositional analysis. Prior approaches to parallelization (see Section 5) use algorithms which compute the exact set of reachable states. The compositional algorithm, however, generally computes an over-approximation of the reachable state set–one which suffices to prove the desired property.

⋆ Kedar Namjoshi's research was supported, in part, by National Science Foundation grant CCR-0341658.

⋆⋆ This material was based on work supported by the National Science Foundation, while Lenore Zuck was working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this article are those of the author and do not necessarily reflect the views of the National Science Foundation.

The algorithm we consider is a mechanization of the classical Owicki-Gries compositional method [29]. The model is that of *asynchronously*-composed processes, communicating through *shared-memory*. The algorithm constructs a "local proof", which is a collection of per-process assertions, $\{\theta_i\}$, whose conjunction (i.e., $\theta_1 \wedge \theta_2 \ldots \wedge \theta_N$) is guaranteed to be an inductive whole-program invariant. This vector of local assertions is called a *split-invariant*, as the program invariant is in this conjunctive form. In previous work [6–9], we have shown that this algorithm often out-performs the standard reachability-based method of verifying safety properties.

The computation of the strongest split invariant is a simultaneous fixpoint computation over the vector $(\theta_1, \theta_2, \ldots, \theta_N)$. In the simplest setting, each thread of a multi-threaded implementation is responsible for computing one component of the fixpoint. The interaction between thread i and another thread j is limited to communicating the effect that the transitions of processes i have on the *shared* program state.

While it is easy to see how to parallelize the fixpoint computation, an actual implementation with BDDs is not straightforward. The BDD data structure is naturally "entangled". Standard BDD libraries are not thread-safe. We show that one can exploit the locality of the reasoning, and use independent non-thread-safe BDD stores, one per thread.

The algorithm has been implemented using JTLV [32], a Java-based framework for developing verification algorithms. The experimental results are very encouraging, on several (parameterized) protocols, the parallel algorithm demonstrates speedup ranging from 5 to nearly 7.5 on a system with 8 cores, with a small memory overhead.

The extension of the local reasoning computation to liveness properties given in [7, 8] is also easily parallelizable. In a nutshell, the liveness algorithm first computes the strongest split invariant, followed by an independent analysis of each component process. The second step is trivially parallelized.

To summarize, we view the main contribution of this work is in proposing and evaluating the use of compositional reasoning as a basis for parallel model checking. As local reasoning is itself often more efficient than a global reachability computation, parallelization offers a multiplicative improvement over sequential reachability analysis. While our implementation and experiments use finite-state protocols and BDDs, the algorithmic ideas are more general, and apply also to non-finite domain representations, such as those used in static program analysis.

An abbreviated description of this work will be presented at the EC2 workshop, associated with CAV 2010 [10].

2 Background

This section introduces split invariance and gives the simultaneous fixpoint formulation of the strongest split invariant. Some of this material is taken from [27], and is repeated here for convenience.

Definition 0 *A* program *is given by a tuple (V, I, T), where V is a set of (typed) variables, $I(V)$ is a predicate over V defining an initial condition, and $T(V, V')$ is a predicate defining a transition condition, where V' is a fresh set of variables in 1-1 correspondence with V.*

The semantics of a program is given by a *transition system*: a triple (S, S_0, R), where S is the state domain defined by the Cartesian product of the domains of variables in V, $S_0 = \{s \,:\, I(s)\}$, and $R = \{(s,t) \,:\, T(s,t)\}$. T is assumed to be left-total, i.e., every state has a successor. A *state predicate* (also called "assertion") is a Boolean-valued expression over the program variables. The truth value of a predicate at a state is defined as usual, by induction on formula structure. The expression $w(s)$ represents the value of variable w in state s.

Definition 1 *The* asynchronous composition *of processes $\{P_i\}$, written as $||_i P_i$, is the program $P = (V, I, T)$, where the components are defined as follows. Let $V = \bigcup_i V_i$ and $I = \bigwedge_i I_i$. The* shared variables*, denoted X, are those that belong to $V_i \cap V_j$, for some distinct pair (i, j). The* local variables *of process P_i, denoted L_i, are the variables in V_i that are not shared (i.e., $L_i = V_i \setminus X$). We assume, for simplicity that, for each i, it is true that $V_i = L_i \bigcup X$. The set of local variables is $L = \bigcup_i L_i$.*
Let $\hat{T}_i = T_i(V_i, V_i') \wedge (\forall j \colon j \neq i \colon pres(L_j))$, where $pres(W)$ is short for $\bigwedge_{w \in W}(w' = w)$. Thus, $\hat{T}_i$ behaves like T_i, but leaves local variables of other processes unchanged. The transition relation of the composition, T, is defined as $\bigvee_i \hat{T}_i$.

Notation In what follows, we use a notation introduced by Dijkstra and Scholten [13]. Sets of program states are represented by first-order formulas. Existential quantification of a formula ξ by a set of variables X is denoted as $(\exists X \,:\, \xi)$ The notation $[\xi]$ stands for "ξ is valid". The successor operation is denoted by sp (for strongest postcondition): $sp(\tau, \xi)$ represents the set of states reachable in one τ-transition[4] from states in ξ.

Inductiveness and Invariance A state predicate is an *invariant* of program $M = (V, I, T)$ if it holds at all reachable states of the transition system defined by M. It is an *inductive invariant* for M if (1) it includes all initial states (i.e., $[I \;\Rightarrow\; \xi]$), and (2) it is preserved by program transitions (i.e., $[sp(T, \xi) \;\Rightarrow\; \xi]$). An inductive invariant ξ is *adequate* to prove the invariance of a state predicate φ if it implies φ.

Local Reasoning and Split Invariants Consider an N-process composition $P = ||_k P_k$. To reason locally about P, we restrict the shape of invariance assertions to a special form. A *local* assertion is one that is based on the variables of a single process, say V_i (equivalently, on X and L_i). A vector of local assertions, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$, is called a *split assertion*. A split assertion θ is a *split invariant* if the conjunction of the components, i.e., $\bigwedge_k \theta_k$, is an inductive invariant for P.

[4] This can be represented by the formula $unprime(\exists V : \tau(V, V') \wedge \xi(V))$, where the $unprime$ operator replaces each next-state variable x' with its current-state counterpart x.

Definition 2 ***(Summary Transition)*** *For a split assertion θ and process k, the summary transition for process i, denoted $\overline{T}_k(X, X')$, is defined as $(\exists L_k, L'_k : T_k \wedge \theta_k)$. This captures the effect of the transition relation T_k of process k on the shared variables X, from states satisfying θ_k.*

Split Invariance as a Fixpoint As shown in [27], the split-invariance constraints can be simplified into the equivalent set of constraints below, making use of locality. For process i, sp_i is the strongest post-condition operator for component P_i; i.e., $sp_i(\tau, \xi) = unprime_i(\exists V_i : \tau \wedge \xi)$. By definition, the result of sp_i is a local assertion (on V_i). For each process index i:

1. **[initiality]** θ_i should include all initial states of process P_i: $[(\exists L \setminus L_i : I) \Rightarrow \theta_i]$
2. **[step]** θ_i must be closed under local transitions of P_i: $[sp_i(T_i, \theta_i) \Rightarrow \theta_i]$
3. **[non-interference]** θ_i must be closed under transitions by processes other than P_i. For all k different from i, $[sp_i(\overline{T}_k \wedge pres(L_i), \theta_i) \Rightarrow \theta_i]$

Calculating the Strongest Split Invariant By monotonicity of the left-hand sides of these constraints and the Knaster-Tarski theorem, there is a least vector solution, which is the least fixpoint, denoted by $\theta^* = (\theta^*_1, \ldots, \theta^*_N)$. For each i, the i'th component of θ^* is a local assertion on V_i; thus, the least solution is also the strongest split-invariant. Hence, from [27], a global property φ is invariant for program P if $[(\bigwedge_i : \theta^*_i) \Rightarrow \varphi]$ holds. The least fixpoint can be computed by the standard Knaster-Tarski approximation sequence, as shown in Fig. 1. The calculation can be optimized by computing only the change to each θ_i, in a manner similar to the use of a frontier set in the standard reachability algorithm.

forall i: θ_i := $(\exists L \setminus L_i : I)$; `/* initialize` θ_i `by (1) */`
while (fixpoint is not reached){
  `/* compute summary transitions */`
  forall i: $\overline{T_i}$:= $(\exists L_k, L'_k : T_k \wedge \theta_k)$;
  `/* compute states reachable by (2) and (3) */`
  forall i: $\theta_i := \theta_i \vee sp_i(T_i, \theta_i) \vee (\bigvee_{k \neq i} sp_i(\overline{T}_k \wedge pres(L_i), \theta_i))$;
}

Fig. 1. Outline of the sequential split invariance computation.

Completeness of Local Reasoning A split invariant is a restricted class of formula; hence, the local reasoning method may fail to prove a property—the induced global invariant may be too weak. As shown by Owicki and Gries [29] and Lamport [24], this can always be remedied by adding shared auxiliary variables to the program, whose sole purpose is to expose more of the local state of the processes. Heuristics for automatically deriving such auxiliary variables were presented in [6, 7]. In this paper, we focus on the split invariance calculation, with auxiliary variables already added, if necessary.

3 Parallelizing The Split-Invariance Calculation

This section describes how to parallelize the simultaneous fixpoint calculation of the strongest split invariant and provides a generic algorithm outline. We discuss BDD implementation issues and heuristics.

3.1 Parallelizing the Least Fixpoint Evaluation

The operations required to evaluate the simultaneous fixpoint (conjunction, disjunction, quantification, etc.) can be carried out by standard BDD manipulation for finite variable domains. From the chaotic iteration theorem [12], the least vector fixpoint can be obtained by *any fair schedule* of the operations. This theorem is central to the parallelization, as it allows the computation for θ_i to be carried out at a different rate than that of θ_j, for $j \neq i$. Hence, as pointed out in [11], the computations can be carried out on distinct processors with very loose synchronization.

The parallel algorithm is outlined in Fig. 2. For simplicity, it is assumed that each component of the fixpoint is computed by a separate thread. The algorithm is described for thread i, which is responsible for component θ_i. It corresponds to the fixpoint evaluation schedule where θ_i is initialized according to (1); the sp_i operations in (2) and (3) are iterated until θ_i stabilizes (this generates states reachable from actions of process P_i and the shared effects of other processes); only then is the effect of process P_i calculated, and broadcast to all other processes. This is repeated until global convergence.

Constraint (3) forces communication and synchronization between the various threads. By definition, the summary transition, $\overline{T}_k$, represents the effect on the shared state of transitions taken by process P_k from the set of states satisfying θ_k. This term is periodically evaluated at thread k, using its current value for θ_k, and the result is broadcast to all other threads (as shown in Fig. 2 for thread i). The broadcast can be carried out through a (virtual) communication topology. The reception of such broadcasts is left implicit in the algorithm description.

3.2 BDD implementation issues

Implementing the computation in Fig. 2 with BDD techniques gives rise to issues concerning synchronization and memory locality, which we discuss below.

Currently available BDD implementations are not thread-safe and require substantial modification to be so [35]. A sequential BDD store can be made thread-safe at a coarse-grain level by a global lock acquired prior to each BDD operation; however, this is prohibitively expensive. Moreover, even a single thread-safe BDD store may have locality issues. The summary transition terms broadcast by each thread are accessed by multiple threads, which requires synchronization. BDDs representing "mixed" terms (i.e., terms such as θ_i that depend on both X and L_i) are accessed by a single thread, and do not require synchronization. Using a single (thread-safe) BDD store to represent

$\theta_i := (\exists L \setminus L_i : I)$; /* initialize θ_i by (1) */
forall $k : k \neq i : \overline{T}(k) := \mathit{false}$;
while (not globally converged){
  while (θ_i does not stabilize){
    /* compute states reachable by (2) and (3) */
    $\theta_i := \theta_i \vee sp_i(T_i, \theta_i) \vee (\bigvee_{k \neq i} sp_i(\overline{T}(k) \wedge pres(L_i), \theta_i))$
  }
  /* broadcast this process' summary */
  asynchronously broadcast $\overline{T}(i) = (\exists L_i, L_i' : T_i \wedge \theta_i)$;
}

Fig. 2. Outline of the computation for Thread i. Vector $\overline{T}$ represents summary transitions. A secondary thread (not shown) is used to receive updates for $\overline{T}$ from other threads via the broadcast operation.

both types of BDDs uniformly could result in mixed BDDs from distinct threads being mapped to the same unique table bucket, which unnecessarily synchronizes accesses to these BDDs. (This scenario is similar to "false sharing", which arises when variables local to distinct threads are mapped to the same cache line.)

Our current implementation has multiple (non-thread-safe) BDD stores, one for each thread. From the structure of the local reasoning computation, the θ_i term does not refer to L_j, for $j \neq i$, so that it is not necessary to pick a single total ordering of the local variables. The BDD stores have to agree on the ordering of the shared variables X (or incur a cost to translate between distinct orders). With this structure, there is a certain amount of *replication* amongst the BDD stores: if local BDDs from distinct threads have a common term (necessarily over X, X'), the BDD for this term is replicated. Another potential issue is the cost of *copying* summary transition BDDs (the $\{\overline{T}_k\}$ terms) between stores to implement the broadcast operation from Fig. 2. In many of our experiments, we did not observe a serious effect from either replication or copying, but the degree to which this is an issue depends on the amount of shared state in the protocol. This is examined in more detail in Section 4.

3.3 Implementation Decisions

Perhaps the most important decision is that of the topology of the implementation, the most obvious ones being a clique, a star, and a tree. While intuitively it seems that the topology of the threads should mimic the topology implemented protocol, it turns out that often this is not the case. For example, while the underlying topology of Szymanski's mutual exclusion algorithm [37] is a clique, our experiments show that an implementation with a star topology is more advantageous.

The simplified description given above allocates one thread to each component of the vector. This can be bad for performance if too many threads are created. (For good utilization, the number of threads should be approximately the number of cores.) An alternative is to let each system thread represent several processes. Hence, each thread

or core is responsible for computing several components of the split-invariant vector. The BDDs for these components are managed by a single, per-thread BDD store.

Large BDD *caches* can improve performance by avoiding recomputation of previously computed BDD nodes. In our experimental results, we present the performance of the sequential and parallel algorithms on the cache size which gave the best results for the type of algorithm.

4 Experiments and Results

We compared the parallel algorithm with the sequential split invariant algorithm, which was shown in [9] to often have order-of-magnitude improvements in run-time over monolithic model checking. All experiments reported here were conducted on a dual-quad-core AMD Opteron (8 cores total), with $1.1GHz$ clock-speed and $512KB$ cache processors, and a total of $32G$ RAM. Both versions were implemented in Java, using JTLV (*Java Temporal Logic enVironment*) [32], a BDD-based framework for developing verification algorithms. JTLV provides a common Java API to several BDD libraries. We used a native JAVA BDD package, based on BUDDY, which is supplied in JAVABDD.

For our testbed, we used four known algorithms, three mutual exclusion protocols and a cache coherence protocol, representing parameterized systems with various number of shared variables, amount of synchronization, and complexity of transition relation.

We tested an optimized sequential implementation, and the parallel algorithm with different numbers of processing cores (2/4/8). The tests were done on several instantiations of each protocol. All the results reported here refer to the optimal execution we obtained. For each parameterized system, we measured, for each instantiation, the number of BDD nodes in the sequential and in the parallel case and the increment (if any) caused by the latter. We then compare, for each instantiation, the speedup obtained, and the efficiency of the parallel implementation measured as

$$\frac{\text{speedup}}{\#\text{ of active cores}} = \frac{\text{sequential time}}{\text{parallel time} \times \#\text{ of active cores}}$$

where "# of active cores" is the minimum between the number of threads and the number of processing cores available.

4.1 The Examples

Unless noted, code for examples can be found in [1].

Mutual Exclusion with Semaphores MUXSEM is a simple parameterized mutual exclusion protocol, which uses a semaphore to coordinate accesses to the critical region. Multiple processes from the protocol were mapped to a single thread. For $N =$

$512, 1024, 1536$ we used 32 threads, and 64 threads for $N = 2048$. The broadcast operation is implemented by the central thread, which disjuncts transitions from multiple threads before forwarding them, thus reducing the number of messages (while increasing their complexity). The property we verified is that of mutual exclusion. Table 1 shows the number of BDD nodes for each instantiation. Table 2 shows the speedup and efficiency obtained. Note that the number of BDD nodes is roughly the same for the sequential and parallel implementations.

	Sequential	**Parallel**	
N	**number of BDD nodes**	**number of BDDs nodes**	**BDD inc.**
512	19.5M	19.8M	1%
1024	81.0M	82.0M	1%
1536	219.0M	221.0M	1%
2048	335.0M	342.0M	2%

Table 1. Number of BDD nodes for MUXSEM

	sequential	**2 cores**			**4 cores**			**8 cores**		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
512	27	16	1.68	0.84	8.3	3.25	0.81	4.8	5.6	0.70
1024	117	65.8	1.77	0.88	34.8	3.3	0.82	19.2	6.1	0.76
1536	360	203	1.77	0.88	112	3.2	0.80	65	5.5	0.69
2048	561	314	1.80	0.90	165	3.4	0.85	92	6.1	0.76

Table 2. Test results for MUXSEM

Szymanski's protocol [37] SZYMANSKI is a more complex mutual exclusion protocol where communication is achieved by shared distributed (single-writer multiple-readers) variables. We verified the mutual exclusion property for both sequential and parallel implementations for $N = 6, 7, 8, 9$ on 2-, 4- and 8-core machines. The results when applying a star topology are provided in Table 3 and Table 4. As can be seen from the tables, the efficiency obtained is similar to that obtained for MUXSEM, however, there is an increase in the number of BDD nodes required for the parallel implementation (that is correlated with the size of the instantiation).

German 2004 German's original cache coherence protocol (see, e.g., [31]), consists of a central controller called $Home$ and N clients that coordinate with $Home$ for shared and exclusive access to a shared variable. In a tutorial at FMCAD'04, German introduced a more involved version of the protocol, that became known as "German 04". A

	Sequential	Parallel	
N	number of BDD nodes	number of BDDs nodes	BDD inc.
6	4.8M	6.9M	43%
7	16.1M	23M	42%
8	49M	73M	48%
9	141M	216M	53%

Table 3. Number of BDD nodes for SZYMANSKI

	sequential	2 cores			4 cores			8 cores		
N	Time	Time	Speedup	Eff.	Time	Speedup	Eff.	Time	Speedup	Eff.
6	20.5	11.6	1.76	0.88	6.5	3.15	0.78	4.4	4.65	0.78
7	130	73.5	1.76	0.88	41	3.17	0.79	23.7	5.48	0.78
8	564	302	1.87	0.93	163	3.46	0.86	93	6.06	0.76
9	2896	1362	2.12	1.06	739	3.91	0.97	492	5.88	0.73

Table 4. Test results for SZYMANSKI

description of the protocol is in [16]. The new protocol differs from its predecessor by allowing each process to be both a home of some cache lines, and a client for all cache lines. It also allows for message queues, and a "send/receive" cycle for each process. Our modeling of the protocol is based on the one of [31]. We do not deal with the message queues and rather assume that each channel can hold at most one message, and model the channels as shared variables. We also simplified the protocol by assuming that each process is a home for a single cache line, though we can easily remove this assumption. Finally, we replace the send/receive cycles by non-determinism; our safety proof implies that of the more detailed version.

The protocol is defined by $||_{i=1}^{N} P[i]$ where each process $P[i]$ is itself a parallel composition of N homes (one for each client it serves) and N clients (one for each home), which we denote by $Home[i][1], \ldots, Home[i][N]$ and $Client[i][1], \ldots, Client[i][N]$, that is,

$$P[i] \;::\; \mathop{||}_{j=1}^{N} Home[i][j] \quad || \quad \mathop{||}_{j=1}^{N} Client[i][j]$$

The property we wish to verify for this system is that of *coherence* by which there cannot be two clients, one holding a shared access to a cache line and the other holding, simultaneously, an exclusive access to the same cache line.

The system thus consists of a parallel composition of N^2 subsystems, and is equivalent to the system $||_{i=1}^{N} Q[i]$ where clients are grouped with the homes they refer to.

$$Q[i] \;::\; \mathop{||}_{j=1}^{N} Home[i][j] \quad || \quad \mathop{||}_{j=1}^{N} Client[j][i]$$

In the $P[i]$ processes, home and clients share no variables, while a home and its clients on other threads share only communication channels. The advantage of this refactor-

ing is that the $Q[i]$'s do not communicate with one another – they each consist of a home process and the fragments of clients that communicate with it. We thus apply the analysis where each thread models a single $Q[i]$. We used a star topology where the central thread is used only for communication and does not model any process. The experimental results are in Table 5.

We omit the table of the number of the BDD nodes since they are the same for both the sequential and the parallel case, with one important exception: The sequential version state exploded with $N = 12$ (BUDDY allows up to 429M BDD nodes), while the parallel version did not, and used 718M BDD nodes, split among the threads. Note that for a few instances the efficiency exceeds 1! We believe that this may be due to better cache utilization due to the multiple BDD stores.

	sequential	**2 cores**			**4 cores**			**8 cores**		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
8	185	78	2.37	1.19	44	4.20	1.05	31	5.96	0.74
9	489	234	2.08	1.04	126	3.88	0.97	76	6.40	0.80
10	1076	511	2.10	1.05	268	4.00	1.00	164	6.56	0.82
11	2867	1310	2.18	1.09	691	4.14	1.03	385	7.44	0.93
12	over BDD limit	3505	-	-	1819	-	-	1013	-	-

Table 5. Test results for German's cache coherence protocol

Peterson's Mutual Exclusion Protocol [30] This protocol uses both shared arrays and distributed shared variables. We proved its mutual exclusion property. This is a particularly interesting examples, because of the high number of shared variables (see Subsection 4.2 for elaboration on this point). Yet, we obtained significant speedup (see Table 7) for all cases but for $N = 4$ on an 8-core machine, with rather small increment in the number of BDD nodes (see Table 6).

	Sequential	**Parallel**	
N	**number of BDD nodes**	**number of BDDs nodes**	**BDD inc.**
4	266k	301k	13%
5	2M	2.4M	20%
6	15M	21M	40%

Table 6. Number of BDD nodes for PETERSON's

	sequential	2 cores			4 cores			8 cores		
N	**Time**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**	**Time**	**Speedup**	**Eff.**
4	0.7	0.7	1.00	0.50	0.7	1.00	0.25	0.6	1.16	0.29
5	6.1	4.5	1.35	0.67	2.5	2.44	0.61	1.9	3.20	0.64
6	123	63	1.95	0.97	36	3.41	0.85	22.5	5.46	0.91

Table 7. Test results for PETERSON's

4.2 Comments and Observations

Letting each thread have its own BDD store and distributing the BDD nodes among the store resulted in only a slight increase in the total number of BDD nodes, and at times accommodated larger instantiations than allowed by the sequential counterpart.

Roughly speaking, we believe that what is happening here has to do with "locality" – the more restricted a process is to its local environment, the less BDD nodes and the faster runtime. In addition, the number of shared variables may also play a role in the results:

In MUXSEM, there are two shared variables (and no distributed shared variables), one that is finitary (as a matter of fact, boolean) and the other that takes on values in the range $[1..N]$, thus there are $O(\log N)$ shared variables with relatively simple access. In SZYMANSKI there are no shared variables, but each process has a finitary distributed shared variable. Thus, there are $O(N)$ shared variables, each can be written by a single process and read by all. Indeed, the efficiency obtained for this case is somewhat worse than that obtained for MUXSEM. For German, after we manipulated the processes, we obtained no sharing, and, consequently, high efficiency.

PETERSON has the most complex structure of variables each process has a distributed shared variable that can have values in the range $[1..N]$, and there is a shared array $[1..N] \mapsto [0..N]$. In fact, the arrays are not stratified (see [1]). There are $O(N \log N)$ shared variables, however, a conclusion from our promising results is that the interaction among them is rather localized.

Another issue, mentioned in Subsection 3.2, is that the multiple-store implementation incurs costs due to replication and copying of BDD's. We instrumented the code to measure speedup costs due to copying of BDD's across threads. The results for 8 cores are shown in Table 8. (The structure of GERMAN's protocol implies that there is no copying required.) From this table, it is clear that the copying cost is low, but also that there is a strong correlation between the copying cost and the efficiency of the parallel algorithm on a given protocol. Thus, providing a way to do the broadcast without requiring copying of BDD's would further improve the performance of the parallel algorithm.

5 Related Work and Conclusions

We compare our approach with earlier work on partitioned BDD representations and parallel model checking.

N	Threads	Copy (sec)	Algorithm (sec)	Total (sec)	Copy/Total
MUXSEM:					
512	32	0	4.8	4.8	0.00
1024	32	0.3	18.9	19.2	0.01
1536	32	0.8	64.2	65	0.01
2048	64	2.4	91.7	92	0.02
SZYMANSKI:					
6	6	1.0	3.4	4.4	0.22
7	7	3.5	21	23.7	0.15
8	8	11.3	81.7	93	0.12
9	9	32	460	492	0.06
PETERSON'S:					
4	4	0.10	0.50	0.6	0.16
5	5	0.28	1.62	1.9	0.14
6	6	1.60	20.9	22.5	0.07

Table 8. Results showing copying time

It is known that partitioned representation of the reachable states, and of transition relations, can significantly speed up a reachability computation. Examples include (implicit) conjunctive partitioning of transition relations [2] and reachability sets [22], overlapping projections [17], approximate traversal [4, 33], and OBDDs partitioned according to window functions [28]. These representations, and others, have been used to split up the work of reachability in parallel (distributed) algorithms [3, 18, 19], as well as in parallel (shared-memory) algorithms [14, 23, 34, 35].

A significant point of difference with these methods is that, instead of computing the exact set of reachable states, the local reasoning method computes an over-approximation in the form of a split invariant. The form of the split invariant requires "looser" connections between the BDDs in the split invariance vector—the connections are, by definition, only on the portions of the BDDs which represent shared variables.

The Machine-by-Machine and Frame-By-Frame traversals and their variants [5, 26] perform approximate reachability in a synchronous computation model. The results of LMBM can be tighted by using overlapping projections [17]. In particular, the LMBM method in [26] has similarities to the split-invariance computation. At each fixpoint step of LMBM, θ_i is updated using the image of $(\bigwedge_k : \theta_k)$ by T_i, treating non-P_i variables as unconstrained. This is qualitatively weaker than the steps (2) and (3) in Section 2, which account for interference by other processes. Of course, the underlying models differ; but applying LMBM to an encoding of asynchronous computation in the synchronous model would result in weaker results than split-invariance.

To the best of our knowledge, this work represents the first parallel model checking method based on compositional reasoning. Intuitively, compositional reasoning has the advantage of more localized computation over non-compositional reasoning. Moreover, the local reasoning algorithm can often succeed in proving a property without computing the exact reachability set, and automated heuristics can be applied for choosing the

auxiliary variables necessary for completeness [6, 7]. The locality of the computation makes it easier to parallelize, and results in locality in BDD operations. The experiments justify this intuition by showing significant speedup over an optimized sequential computation of split invariance.

In most cases, the memory overhead of our implementation is small. As explained in Section 4, this overhead is correlated with the size and the usage of the shared variable space. The small overhead of the compositional approach can be contrasted with the parallel (exact) symbolic reachability computation on asynchronous programs in [14, 25], where the parallelized algorithms showed excessive memory overhead, between 2 and 20 times the memory required by the sequential algorithm.

Parallel versions of explicit-state model checking algorithms have been developed for the Murφ and SPIN model checkers [20, 21, 36]. These algorithms compute the exact reachability set (under partial-order reductions), and are different in that crucial respect from the local computations described here. When cast in explicit-state terms, the split invariance calculation is precisely the "thread-modular" algorithm described by Flanagan and Qadeer in [15].

In terms of future work, several directions open up. One is to investigate whether a single (thread-safe) BDD store can provide better performance than the current multiple-store implementation. Another is to experiment with a distributed-memory implementation of this method. Yet another is to design parallel algorithms for computing split invariance with explicit state representations. We also plan to incorporate the parallel algorithm into SPLIT [9].

References

1. Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, volume 2102 of *LNCS*, pages 221–234, 2001.
2. Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transistion relations. In *VLSI*, 1991.
3. Gianpiero Cabodi, Paolo Camurati, Antonio Lioy, Massimo Poncino, and Stefano Quer. A parallel approach to symbolic traversal based on set partitioning. In *CHARME*, pages 167–184, 1997.
4. Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Improved reachability analysis of large finite state machines. In *ICCAD*, pages 354–360, 1996.
5. Hyunwoo Cho, Gary D. Hachtel, Enrico Macii, Bernard Plessier, and Fabio Somenzi. Algorithms for approximate fsm traversal based on state space decomposition. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 15(12):1465–1478, 1996.
6. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *CAV*, volume 4590 of *LNCS*, pages 55–67. Springer, 2007.
7. A. Cohen and K. S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV*, volume 5123 of *LNCS*, pages 149–161. Springer, 2008.
8. A. Cohen, K. S. Namjoshi, and Y. Sa'ar. A dash of fairness for compositional reasoning. In *CAV*, 2010.
9. A. Cohen, K. S. Namjoshi, and Y. Sa'ar. Split: A compositional LTL verifier. In *CAV*, 2010.

10. Ariel Cohen, Kedar S. Namjoshi, Yaniv Sa'ar, Lenore D. Zuck, and Katya I. Kisyova. Model checking in bits and pieces. In *EC2 Workshop, CAV 2010*, 2010.
11. P. Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France, Sep. 1977.
12. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence & Programming Languages*, Rochester, NY, ACM SIGPLAN Not. 12(8):1–12, August 1977.
13. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer Verlag, 1990.
14. Jonathan Ezekiel, Gerald Lüttgen, and Gianfranco Ciardo. Parallelising symbolic state-space generators. In *CAV*, volume 4590 of *LNCS*, pages 268–280. Springer, 2007.
15. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224, 2003.
16. S. German and G. Janssen. A tutorial example of a cache memory protocol and RTL implementation. Technical Report RC23958 (W0605-092), IBM, 5 2006.
17. Shankar G. Govindaraju, David L. Dill, Alan J. Hu, and Mark Horowitz. Approximate reachability with bdds using overlapping projections. In *DAC*, pages 451–456, 1998.
18. Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, volume 3725 of *LNCS*, pages 129–145, 2005.
19. Orna Grumberg, Tamir Heyman, and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.
20. Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.
21. Gerard J. Holzmann and Dragan Bosnacki. Multi-core model checking with SPIN. In *IPDPS*, pages 1–8. IEEE, 2007.
22. A. J. Hu and D. L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *CAV*, volume 697 of *LNCS*, pages 3–14, 1993.
23. Subramanian K. Iyer, Debashis Sahoo, E. Allen Emerson, and Jawahar Jain. On partitioning and symbolic model checking. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(5):780–788, 2006.
24. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2), 1977.
25. G. Lüttgen. Parallelising Symbolic State-Space Generators: Frustration & Hope. Seminar on Distributed Verification and Grid Computing. Schloss Dagstuhl, Germany, August 2008. at `http://www-users.cs.york.ac.uk/~luettgen/presentations`.
26. In-Ho Moon, James H. Kukula, Thomas R. Shiple, and Fabio Somenzi. Least fixpoint approximations for reachability analysis. In *ICCAD*, pages 41–44, 1999.
27. K. S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, volume 4349 of *LNCS*, 2007.
28. Amit Narayan, Adrian J. Isles, Jawahar Jain, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-robdds. In *ICCAD*, pages 388–393, 1997.
29. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
30. Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
31. A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In TACAS'01, pages 82–97. LNCS 2031, 2001.

32. A. Pnueli, Y. Saar, and L. D. Zuck. Jtlv : A framework for developing verification algorithms. In *CAV*, 2010. web: `http://jtlv.sourceforge.net/`.
33. Kavita Ravi and Fabio Somenzi. High-density reachability analysis. In *ICCAD*, pages 154–158, 1995.
34. Debashis Sahoo, Jawahar Jain, Subramanian K. Iyer, and David L. Dill. A new reachability algorithm for symmetric multi-processor architecture. In *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 26–38. Springer, 2005.
35. Debashis Sahoo, Jawahar Jain, Subramanian K. Iyer, David L. Dill, and E. Allen Emerson. Multi-threaded reachability. In *DAC*, pages 467–470. ACM, 2005.
36. Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.
37. B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proc. 1988 International Conference on Supercomputing Systems*, pages 621–626, St. Malo, France, 1988.