Parallel symbolic state-space exploration is difficult, but what is the alternative?*

Gianfranco Ciardo Yang Zhao Xiaoqing Jin

Department of Computer Science and Engineering
University of California, Riverside

{ciardo,zhaoy,jinx}@cs.ucr.edu

State-space exploration is an essential first step in many modeling and analysis problems. Its goal is to find and store all the states reachable from the initial state(s) of a discrete-state high-level model described, for example, using pseudocode or Petri nets. The state space can then be used to answer important questions, such as "Is there a dead state?" and "Can variable *n* ever become negative?", or as the starting point for sophisticated investigations expressed in temporal logic.

Unfortunately, the state space is often so large that ordinary explicit data structures and sequential algorithms simply cannot cope, prompting the exploration of parallel or symbolic approaches. The former uses multiple processors, from simple networks of workstations to expensive shared-memory supercomputers or, more recently, powerful multicore workstations, to satisfy the large memory and runtime requirements. The latter uses decision diagrams to compactly encode the large structured sets and relations manipulated during state-space generation.

Both approaches have merits and limitations. Parallel explicit state-space generation is challenging, but close to linear speedup can be achieved, thus its scalability can be quite good; however, the analysis is ultimately and obviously limited by the amount of memory and number of processors available overall. Symbolic methods rely on the heuristic properties of decision diagrams, which can encode many, but by no means all, functions over a structured and exponentially large domain in polynomial space; here the pitfalls are subtler, as the performance of symbolic approaches can vary widely depending on the particular class of decision diagram chosen, on the order in which the variables describing the state are considered, and on many obscure algorithmic parameters.

In this paper, we survey both approaches. Observing that symbolic approaches are often enormously more efficient than explicit ones for many practical models (although it is rarely obvious *a priori* whether this will be the case on a particular application), we argue for the need to parallelize symbolic state-space generation algorithms, so that we can realize the advantage of both approaches. Unfortunately, this is a very challenging endeavor, as the most efficient symbolic algorithm, Saturation, is inherently sequential. We conclude by discussing challenges, efforts, and promising directions toward this goal.

1 Introduction

Model checking was introduced almost three decades ago and has gradually been adopted in industrial applications. State-space generation forms the base of safety checking and the first step towards more complex temporal property checking. In some scenarios, such as VLSI circuits, the potential state space is finite; on the other hand, high-level models such as Petri Nets may have an infinite state space. Furthermore, even when a given Petri Net is bounded, a finite bound on the potential state space is usually not known a priori. Hence, state-space generation is an essential and interesting problem.

*Work supported in part by the National Science Foundation under grant CCF-0848463.

L. Brim and J. van de Pol (Eds.): 8th International Workshop
on Parallel and Distributed Methods in verifiCation 2009 (PDMC'09)
EPTCS 14, 2009, pp. 1–17, doi:10.4204/EPTCS.14.1

The most important metrics to evaluate the effectiveness of state-space generation are *memory consumption* and *run time*. These two metrics are often closely related, but ultimately reflect different complexity aspects. Considering run time, state-space generation can be expressed as a fixpoint iteration and, for models with very large diameter (maximum distance from an initial state to any reachable state), this iteration can be very time consuming. Considering memory consumption, the state space of complex models is often too large to fit in main memory, or even in secondary storage. Even when the latter suffices, large memory consumption leads to frequent swaps between main and virtual memory, with negative effects on the run time. Fortunately, processors and memory are becoming cheaper at each new generation, so that multi-core processors and multi-processor systems provide larger computational resources at the same or lower price. The memory bottleneck can be relieved using more memory per workstation and multiple workstations, but tackling the long run times is more difficult. While multi-processor systems enable the execution of multiple tasks in parallel, it is hard to achieve a speedup linear in the number of processors. The difficulty lies in the need to find enough parallel tasks to fully exploit the available processors. We believe that speedup will be a fundamental concern for future research on parallel formal verification algorithms.

1.1 Problem setting and notation

If we ignore the particular high-level formalism used to express our system, we are interested in studying a *discrete state model* fully specified by:

- A set of states, or *potential state space* $\mathcal{X}_{pot}$, which describes the "type" of the states.
- A set of *initial states* $\mathcal{X}_{init} \subseteq \mathcal{X}_{pot}$ from which the system behavior can evolve. Often, there is a single initial state, $\mathcal{X}_{init} = \{\mathbf{i}_{init}\}$.
- A *next-state function* $\mathcal{N} : \mathcal{X}_{pot} \to 2^{\mathcal{X}_{pot}}$, which describes the states to which a system can move in one step. This function can naturally be extended to sets of states, $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i}\in\mathcal{X}} \mathcal{N}(\mathbf{i})$.

We observe that the model is *nondeterministic*, unless, for all states $\mathbf{i} \in \mathcal{X}_{pot}$, $|\mathcal{N}(\mathbf{i})| \leq 1$, and we say that state $\mathbf{i}$ is *absorbing* (or a *trap*, or *dead*, or a *sink*) if $\mathcal{N}(\mathbf{i}) = \emptyset$. In the literature, a *transition relation* is sometimes defined instead of the next-state function, but the two carry exactly the same information: the pair of states $(\mathbf{i},\mathbf{j})$ is in the transition relation iff $\mathbf{j} \in \mathcal{N}(\mathbf{i})$. Then, we assume that the state is *structured*:

- $\mathcal{X}_{pot} = \mathcal{X}_L \times \cdots \times \mathcal{X}_1 = \times_{L\geq k\geq 1} \mathcal{X}_k$, so that a (global) state is of the form $\mathbf{i} = (i_L, ..., i_1)$, and $\mathcal{X}_k$ is the (discrete) *local state space* for submodel k or the *local domain* for state variable x_k.

The techniques we consider assume that $\mathcal{X}_{pot}$ is finite, thus $\mathcal{X}_k$ must be finite as well, and we can map it to $\{0,1,\ldots,n_k-1\}$. If n_k is *unknown a priori*, we can initially let $\mathcal{X}_k = \mathbb{N}$, the set of natural numbers, and discover the value of n_k later, as we explore the model. Finally, we assume *asynchronous* behavior, that is, there is a set $\mathcal{E}$ of *events* defining a *disjunctively-partitioned* next-state function:

- For each event $\alpha \in \mathcal{E}$, $\mathcal{N}_\alpha : \mathcal{X}_{pot} \to 2^{\mathcal{X}_{pot}}$. State $\mathbf{j}$ can be reached by *firing* α in state $\mathbf{i}$ iff $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$.
- The overall next-state function is the union of the functions for each event, $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha\in\mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$.
- We say that event α is *enabled* in $\mathbf{i}$ iff $\mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset$, otherwise we say it is *disabled*.

The main goal of our study is then to generate and store the (*reachable*, or *actual*) *state space* $\mathcal{X}_{rch}$ of the model, that is, the smallest subset of $\mathcal{X}_{pot}$ containing $\mathcal{X}_{init}$ and satisfying:

- The *recursive definition* $\mathbf{i} \in \mathcal{X}_{rch} \wedge \mathbf{j} \in \mathcal{N}(\mathbf{i}) \Rightarrow \mathbf{j} \in \mathcal{X}_{rch}$.
- Or, equivalently, the *fixpoint equation* $\mathcal{X} = \mathcal{X} \cup \mathcal{N}(\mathcal{X})$.

The most obvious way to think of the state space is as the limit of the expression

$$\mathscr{X}_{init} \cup \mathscr{N}(\mathscr{X}_{init}) \cup \mathscr{N}(\mathscr{N}(\mathscr{X}_{init})) \cup \mathscr{N}(\mathscr{N}(\mathscr{N}(\mathscr{X}_{init}))) \cup \cdots$$

but we stress that, while this expression suggests a breadth-first iteration where states at distance d from $\mathscr{X}_{init}$ are discovered after exactly d iterations (i.e., after d applications of the next-state functions $\mathscr{N}$), this is neither implied by the definition nor it is necessarily the most efficient way to build $\mathscr{X}_{rch}$.

Beyond state-space generation, more advanced analyses can be performed on a discrete-state model. For example, in the temporal logic CTL [21, 33], the operators EX, EU, and EG are *complete*, that is, they can be used to express any CTL operator through complementation, conjunction, and disjunction. If $\mathscr{A}$ and $\mathscr{B}$ are the sets of states satisfying CTL formulae a and b, respectively, the set of states satisfying EXa is $\mathscr{X}_{\mathsf{EX}a} = \{\mathbf{i} : \exists \mathbf{j} \in \mathscr{A} \cap \mathscr{N}(\mathbf{i})\}$, thus $\mathscr{X}_{\mathsf{EX}a} = \mathscr{N}^{-1}(\mathscr{A})$, where $\mathscr{N}^{-1}$ is the *backward state function*, i.e., $\mathscr{N}^{-1}(\mathscr{X}) = \{\mathbf{i} : \exists \mathbf{j} \in \mathscr{X}\ (\mathbf{j} \in \mathscr{N}(\mathbf{i}))\}$. The set of states satisfying EaUb is instead

$$\mathscr{X}_{\mathsf{E}a\mathsf{U}b} = \left\{\mathbf{i}^{(0)} : \exists d \geq 0 \left(\forall c \in \{0,...,d-1\} \left(\mathbf{i}^{(c)} \in \mathscr{A} \wedge \exists \mathbf{i}^{(c+1)} \in \mathscr{N}(\mathbf{i}^{(c)})\right) \wedge \mathbf{i}^{(d)} \in \mathscr{B}\right)\right\}$$

and can be characterized as the *smallest* solution of the fixpoint equation $\mathscr{X} \subseteq \mathscr{B} \cup (\mathscr{A} \cap \mathscr{N}^{-1}(\mathscr{X}))$. Analogously, the set of states satisfying EGa is

$$\mathscr{X}_{\mathsf{EG}a} = \left\{\mathbf{i}^{(0)} : \forall d \geq 0 \left(\forall c \in \{0,...,d-1\} \left(\mathbf{i}^{(c)} \in \mathscr{A} \wedge \exists \mathbf{i}^{(c+1)} \in \mathscr{N}(\mathbf{i}^{(c)})\right)\right)\right\}$$

and can be characterized as the *largest* solution of the fixpoint equation $\mathscr{X} \supseteq \mathscr{A} \cap \mathscr{N}^{-1}(\mathscr{X})$.

We focus on state-space generation, but many of the problems faced are analogous to those for the computation of these more complex fixpoints, and many of the possible solutions are applicable to them.

1.2 Explicit vs. implicit techniques, which one should we parallelize?

Symbolic model checking [8] is undoubtedly a significant breakthrough in formal verification. Instead of representing states explicitly, symbolic approaches exploit advanced data structures to encode and manipulate entire sets of states at once. Paired with binary decision diagrams (BDDs) [7], symbolic model checkers are able to handle enormous state spaces. At the same time, several questions remain open for BDD-based symbolic model checking. One of the most challenging ones is that the evolution of the BDDs being manipulated is quite unpredictable during the fixpoint iterations, and it is their *peak size*, often many orders of magnitude larger than the final result being sought, that may exceed the available memory and cause the program to fail, making symbolic algorithms brittle and subtle. To alleviate this problem, much research has been devoted to *static* or *dynamic variable ordering*, *quantification scheduling*, and *BDD partitioning*, to name a few. After two decades, BDD-based symbolic techniques have become mainstream for the verification of synchronous systems, such as VLSI circuits.

However, state-space generation for asynchronous models such as Petri nets, communicating sequential processes, and process algebras appears more challenging. Although gradually replaced in synchronous systems, explicit techniques are still competitive for asynchronous systems. Such techniques take advantage of the locality and symmetry properties widely enjoyed by asynchronous systems. Partial order reduction [24, 43] and symmetry reduction [10] have been successfully implemented in explicit model checkers to reduce the number of states that must be explored and stored, thus the run time. These approaches explore and store only a "representative" subset of the reachable states, but are nevertheless able to answer the same questions as an exhaustive search. Moreover, low-level memory reduction techniques such as hash compaction are widely employed to further reduce memory requirements.

```
SymbolicBfsSsGen(X_init, N) is
 1  Y ← X_init;                                  • known states
 2  U ← X_init;                                  • unexplored states
 3  while U ≠ ∅ do
 4      Z ← N(U);                                • potentially new states
 5      U ← Z \ Y;                               • truly new states
 6      Y ← Y ∪ U;
 7  return Y;
```

Figure 1: The traditional breadth-first symbolic state-space generation.

The need to choose between explicit and implicit techniques, then, arises mainly for the verification and analysis of asynchronous systems, as many factors can reduce the effectiveness of symbolic algorithms when applied to asynchronous models. Traditional BDD techniques can efficiently encode complex synchronous transition relations, but they are often not as compact when applied to asynchronous events, even if disjunctive partitioning [9] often helps. Analogously, *image computation*, the base of symbolic state-space generation, is often expensive for asynchronous systems. Thus, tools like SPIN [28], a model checker with advanced explicit techniques, have achieved wide acceptance and success in industrial protocol verification. Furthermore, as shown in the following sections, the parallelization of explicit algorithms is much more successful than that of symbolic ones, in the sense that explicit techniques can achieve almost linear speedup when devising parallel implementations for them.

While it appears that the parallelization of explicit techniques is more promising than that of symbolic ones, this paper argues that symbolic techniques have nevertheless often the best chance to shine, if not in speedup, certainly in ultimate performance. The most traditional approach to symbolic state-space generation is shown in Fig. 1, where all sets, i.e., $\mathcal{Y}$, $\mathcal{U}$, and $\mathcal{Z}$, and relations, i.e., $\mathcal{N}$, are encoded using BDDs (if the local state variable x_k is not boolean, we can use $\lceil \log_2 n_k \rceil$ boolean variables for it, or we can use MDDs, in particular *extensible* MDDs [45] if n_k is not known a priori; in the following, we simply use the term DD). This is essentially a symbolic breadth-first exploration, where iteration d discovers all states at distance d from $\mathcal{X}_{init}$. While this approach is often very effective, our confidence in the appropriateness of symbolic techniques mainly comes from an even better algorithm: *Saturation*.

Initially defined for asynchronous models satisfying *Kronecker-consistency* [17], where, for each event α, $\mathcal{N}_\alpha$ is the conjunction of L "local" functions $\mathcal{N}_{\alpha,k} : \mathcal{X}_k \to 2^{\mathcal{X}_k}$, for $L \geq k \geq 1$, Saturation has later been extended to a fully general setting where each $\mathcal{N}_\alpha$ is the conjunction of some number of functions, each depending and affecting a subset of the state variables [20]. The main idea behind Saturation is that this disjunctive-then-conjunctive decomposition highlights the *locality* of each event α, so that we can define $Top(\alpha)$ to be the highest local state variable on which the enabling or the effect of α depends, or which α affects. Then, we build the DD encoding $\mathcal{X}_{init}$ and saturate its nodes bottom up, applying, for k from 1 to L, all events α with $Top(\alpha) = k$ to it, exhaustively, until no more states are discovered, with the proviso that, whenever saturating a DD node p at level k causes the creation of a DD node q at a level below k, we saturate node q before completing the saturation of the higher node p. Thus, when saturated, DD node p encodes a fixpoint with respect to $\mathcal{N}_{\leq k} = \bigcup_{\alpha : Top(\alpha) \leq k} \mathcal{N}_\alpha$ and, when we saturate the root node at level L, we have the entire state space, that is, the fixpoint of $\mathcal{X} = \mathcal{X} \cup \mathcal{N}(\mathcal{X})$. The result is a much more efficient state-space generation algorithm often requiring many orders of magnitude less memory and run time than symbolic breadth-first iterations. Indeed, we have applied the Saturation approach also to CTL model checking [47], distance function computation [19], and timed reachability in integer-timed models [44], but here we will limit our discussion to state-space generation.

1.3 Classification of previous work on parallel state-space analysis

Modern technology offers new parallel platforms for both explicit and implicit techniques. In general, there are two methodologies for parallelization: data decomposition and functional decomposition. Data decomposition distributes the data to be processed *across* parallel tasks, each of which executes on a different workstation. Functional decomposition exploits the parallelism *within* a function computed by an application, allowing the distribution of computation over multiple processors or cores.

Parallel DD-based algorithms have been developed for a variety of platforms: shared memory multiprocessor or multi-core systems [41], network of workstations (NOW) [34], distributed shared memory (DSM) architectures [37], single-instruction-multiple-data (SIMD) and multiple-instruction-multiple-data (MIMD) architectures [23], and vector processors [36]. According to the nature of the state-space generation algorithm, we can generally classify the implementation platform into two categories: distributed-memory architecture vs. shared-memory architecture. The former, e.g., NOW or PC clusters, has the advantage of possessing abundant resources to handle large systems. The latter, e.g., multiprocessor multi-core systems, is becoming the predominant technology trend. We omit detailed discussion of these platforms, and focus on their characteristic features and challenges. For distributed-memory systems, the main considerations are how to distribute data, maintain load balance, and reduce communication overhead and latency. For shared-memory systems, the mechanism employed to guarantee mutually exclusive access to a memory region, load balance, and task scheduling are instead paramount. Most literature over the last 25 years has been devoted to algorithms on distributed-memory systems due to their ability to overcome memory constraints and to the pervasiveness of computer networks. The advent of inexpensive memory and multi-core systems has reignited interest in shared-memory systems.

With respect to an orthogonal classification that takes into consideration not the hardware architecture but the type of data structure employed by the state-space generation algorithm, two main approaches exist. Traditional explicit state space generation approach, such as the one used in the model-checking tool SPIN [28], enumerates and explores each state one by one, and was first parallelized in [16, 35, 40]. The other approach, DD-based state-space generation, is behind all commonly used symbolic model-checking tools, such as SMV [30] and SMART [18].

The remainder of this paper is organized as follows. Section 2 focuses on the explicit distributed-memory approaches proposed in [2, 3, 32, 40] and the explicit shared-memory methods presented in [4, 5, 29]. Section 3 surveys parallel symbolic approaches for distributed-memory architectures [1, 11, 12, 25, 26, 27, 31, 41, 42, 46], as well as the shared-memory approach of [22]. Section 4 discusses some promising directions for further research on optimizing the parallelization of symbolic state-space generation algorithms.

2 Parallelizing explicit state-space generation

Most work in parallel explicit state-space generation has focused on distributed-memory approaches that utilize inexpensive NOWs, although shared memory approaches have also been explored.

2.1 Distributed-memory approaches for explicit state-space generation

As memory consumption is the main bottleneck for explicit techniques, being able to exploit the overall storage available on a NOW is an appealing idea. Most approaches along these lines follow the general framework of Fig. 2. Assuming that there are N workstations indexed with an identifier w ranging from 1 to N, a function $\lambda : \mathscr{X}_{pot} \to \{1, ..., N\}$ is used to partition the potential state space $\mathscr{X}_{pot}$ into N classes, that

```
DistributedExplicitStateSpaceGeneration(w, 𝒳_init, 𝒩, λ) is
 1    foreach i ∈ 𝒳_init do
 2        if λ(i) = w then
 3            𝒰_w ← 𝒰_w ∪ {i};
 4            𝒴_w ← 𝒴_w ∪ {i};
 5    repeat forever
 6        if 𝒰_w = ∅ then
 7            𝒰_w ← GetStatesSentToMeFromOthers(w);      • distributed termination detection...
 8            if 𝒰_w = ∅ then                            • ...returns an empty set if it is time to terminate
 9                return 𝒴_w;
10        i ← ChooseOneStateFrom(𝒰_w);
11        𝒰_w ← 𝒰_w \ {i};
12        foreach j ∈ 𝒩(i) do
13            if λ(j) ≠ w then                           • if j does not belong to you...
14                SendStateToWorkstation(j, λ(j));       • ...send j to its owner
15            else if j ∉ 𝒰_w ∪ 𝒴_w then                  • if j belongs to you and is a new state...
16                𝒴_w ← 𝒴_w ∪ {i};                       • ...add it to your states...
17                𝒰_w ← 𝒰_w ∪ {j};                       • ...and remember to explore j later
```

Figure 2: General framework for distributed explicit state-space generation.

is, assign an "owner" workstation to each state. Then, each workstation performs essentially the same steps as those required for sequential explicit state-space generation, except that each state **j** reachable from the state **i** currently being explored is checked first to see if it belongs to a different workstation, in which case **j** is sent to it, not processed locally.

In this framework, several important factors affect performance. First and foremost, the state-space partitioning function λ has great impact on both workload and memory balance. Second and almost as important, the communication overhead and latency must be taken into account when deciding how and when to exchange states. Specifically, we need to decide whether states belonging to other workstations are sent immediately after being discovered, or if they are buffered into larger messages and, in this case, how large the message buffers should be. Then, we need to decide the frequency at which a workstation checks for incoming states sent to it, as waiting too long to receive these states can cause incoming buffers to grow too large, possibly with many duplicate states in them. The goal of tuning these parameters is then to keep all workstations busy most of the time, while attempting to achieve similar proportions of memory usage in each workstation and minimizing the number of message exchanges.

Of course, these factors might be in conflict. Obviously, a partitioning function where all states belong to one workstation has no communication at all, but the worst workload and memory balance. More interestingly, a perfect hash function for λ will instead achieve excellent workload and memory balance, but also maximize communication for state exchanges, as the probability that any state **j** reachable from **i** belongs to the same workstation as **i** is only $1/N$. Thus, a good choice of λ should still achieve a good workload and memory balance, but at the same time guarantee that most state-to-state transitions remain within the same workstation, thus require no communication. A hash function is often used to define λ, and an approach to achieve a good compromise was discussed in [16] for the case of Petri nets, where, by hashing the state on just a few of its components (the number of tokens in only a few of the Petri net places), we ensure that any Petri net transition not affecting those places will result in states belonging to the same workstation. However, even employing this idea, it is possible to define λ so that the mapping of *reachable* states (as opposed to *potential* states) is highly uneven.

We proposed a completely different way to define λ in [35], by organizing the states in a search

tree (a data structure commonly used in explicit state-space exploration anyway), in such a way that the top few levels of the tree are duplicated in each workstation, while the subtrees at the lower levels are mapped to individual workstations with no duplication. If the shared portion of the tree has $M \gg N$ terminals, each corresponding to a non-shared subtree, the approach simply requires to associate the index of the owner workstation to each of these terminals. As the search progresses, each workstation keeps track of the sizes of the subtrees it owns, and, if a workstation is overloaded, it can restore memory balance by reassigning some of its subtrees to light-loaded workstations. The overhead to rebalance is clearly much lower than with a hash function, which requires to reallocate all the states discovered so far if it is modified. Fig. 3 illustrates this idea, where the states actually stored by workstation w, for $= 1,2,3,4$, are shown in the four quadrants. When workstation 1, 2, or 4 searches for state **s**, with $\lambda(\mathbf{s}) = 3$, it reaches the leaf storing state **k** and learns that workstation 3 owns the subtree rooted at **k**. When workstation 3 searches for state **s**, it instead reaches the node storing state **k** and continues the search, until it finds **s** in that subtree, as in the figure, or determines that state **s** must be inserted in that subtree. The shared top portion of the tree represents a small overhead in practice, since a value for M of the order of 10 to 100 times N is large enough in practice, and this is negligible with respect to the number of reachable states in practical applications. Only minimal synchronization is required, initially to agree on the structure of the top portion of the search tree and the assignment of its leaves to workstations, and, occasionally during state-space generation, to agree on a different subtree-to-workstation mapping, and broadcast this change. The asynchronous communication between pair of workstations is then limited, as before, to sending newly found states determined to belong to another workstation (which again can be reduced if the top portion of the search tree is such that the search is determined by just a few local state components), and to exchange subtrees when load balancing is required.

Both [16], which requires the user to explicitly provide a partitioning function, and the more automated tree-based approach of [35] achieve close to linear speedups, as well as excellent memory load balance on conventional distributed memory architectures.

In [40], the explicit model checker Murϕ is parallelized on a NOW. With the support of a fast message passing scheme, *active messages*, each process runs asynchronously without global synchronization. A universal hash function is used to determine the workstation to which a state belongs to and the property of this hash function guarantees that states are evenly distributed among the workstations. The parallel version of Murϕ achieves close to linear speedup. Also the tool SPIN was parallelized [32], but the focus was not on speedup, rather on the ability to handle large models otherwise intractable. In this work, communication becomes a dominant factor compared to the time to compute successor states. To minimize communication, the partition function λ depends on one state component. As already shown in [16], this reduces cross-transitions between processes. The parallel version of SPIN retains the most important memory and complexity reduction techniques employed by the sequential version.

Beyond state space generation, [3] proposed a distributed algorithm for LTL model checking, building upon a parallel algorithm for *accepting cycle detection* in Büchi automata [2]. Sequential solutions to this problem rely on depth first search (DFS), which is hard to parallelize. The basic idea of this work is then to detect *back-level edges*, i.e., edges $(\mathbf{i},\mathbf{j})$ where the distance of state **i** from the initial state is greater than the distance of state **j** from the initial state. Parallel breadth first search (BFS) is employed to detect back-level edges. After each BFS step, workstations synchronize and detect back-level edges. DFS is then employed on each workstation in parallel to find cycles. Techniques are employed to reduce state revisiting, and partial order reduction can be combined with this distributed algorithm. This parallel scheme falls into the basic framework discussed at the beginning of this section, showing that this framework is applicable to not only reachability analysis, but also to more complicated model checking.

In conclusion, explicit distributed-memory algorithms mainly focus on how to maintain load balance,

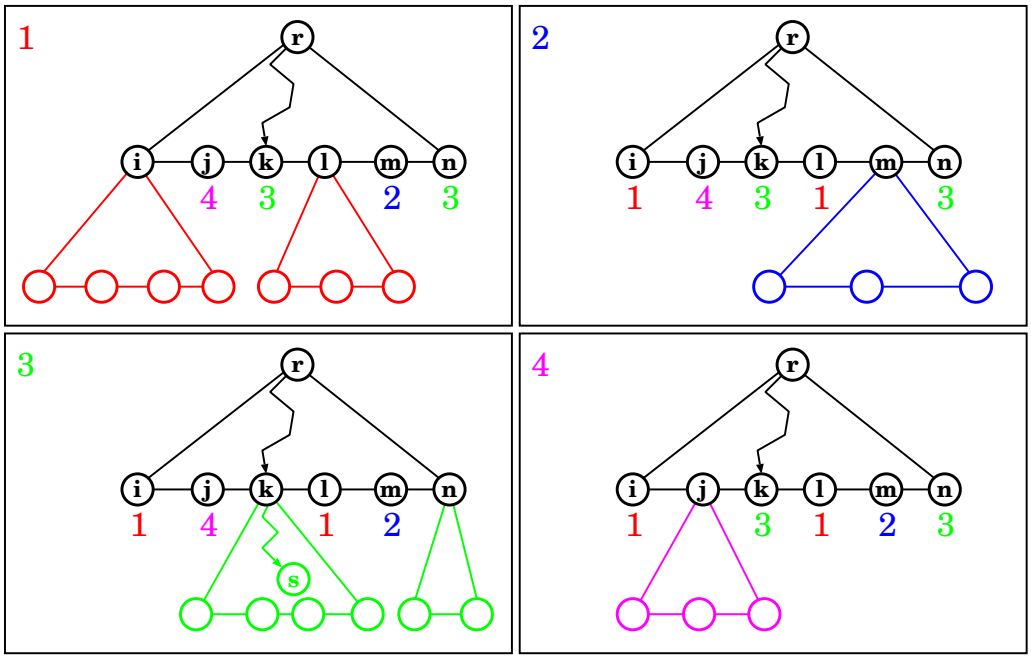

Figure 3: A tree-based partitioning function ($N = 4$, $M = 6$).

using static strategies such as a good hash function or our partially-shared search tree, and try to minimize communication by defining a partition function that exploits knowledge of the state structure.

2.2 Shared-memory approaches for explicit state-space generation

With respect to explicit shared-memory approaches, [4, 5] "port" the parallel algorithm from distributed-memory to shared-memory architecture in conjunction with various techniques to improve real-world performance for LTL model checking. First, lightweight threads play the role of workstations in the NOW framework, and mutual exclusion techniques are used to prevent data races. Then, a two-level lock algorithm is used to reduce synchronization overhead, essentially changing the lock granularity of the data structure. Furthermore, FIFO queues handle message passing between threads, to reduce communication overhead, and are also employed to solve memory allocation issues. Experimental results show that the implementation scales up to 16 cores and has better performance than the MPI version. [4] concludes the main bottleneck is the state generator and proposes in future work to balance the performance of state generator for better scalability of the entire algorithm.

Another relevant work is [29], which provides another algorithm for reachability analysis in the context of CTL* model checking. A work stealing two-queue structure is used to dynamically maintain load balance during state exploration, with low synchronization overhead. Each process has an unbounded shared queuue and a bounded private queue to store unexplored states. A process is allowed to add and remove states in its own queues, but it can also remove states from the shared queue of other processes, thus it can steal another process's work instead of going idle. Of course, shared queues must be guarded by a lock, introducing some synchronization overhead in exchange for good load balance. Experimental results show almost linear speedups up to a 12 to 16 processors, depending on the state space size; beyond that, the benefits of using more processors are offset by synchronization and scheduling overheads.

This observation leads us to the second part of paper: explicit approaches for state-space generation exhibit good scalability on distributed and parallel systems, but only up to some point, beyond which further increasing their performance becomes very difficult. Since symbolic approaches tend to work very well even just in sequential implementations, why not attempt to parallelize them?

3 Parallelizing symbolic state space generation

For symbolic approaches, the first and foremost problem is to identify a set of parallel "tasks", which is a challenge as most symbolic operations are recursive and inherently sequential. Still, there is a large amount of research attacking this problem from different angles. Parallelism can be realized through low-level symbolic operations, such as logic conjunction or disjunction operations in DDs, or through high-level algorithms, such as BFS or Saturation. On distributed-memory systems, symbolic DD structures can be stored in "vertical" or "horizontal" partitions, as we will discuss in Section 3.1.

3.1 Distributed-memory approaches for symbolic state-space generation

One way to achieve parallelization for symbolic techniques is designing parallel BDD libraries for a NOW environment. [34, 38, 42] discuss how to store and manipulate BDDs on a NOW. These approaches to achieve parallelism lie on low-level DD operations and, if they succeed in providing an interface similar to that provided by an ordinary sequential DD library, they can be applied to traditional DD-based algorithms without requiring them to be rewritten. As these packages are mostly applied to benchmark synchronous circuits, their performance on asynchronous models is unknown.

At a higher level, parallelism can be attained through a state-space partitioning approach similar to the one we discussed in Section 2.1. This symbolic strategy, which we call *vertical partitioning*, assumes a partition of the potential state space $\mathscr{X}_{pot}$ into a set of N "windows" $\{\mathscr{W}_1,\ldots,\mathscr{W}_N\}$. These are in fact exactly analogous to the function λ required for the distributed explicit approach: we can simply think of $\mathscr{W}_w$ as $\{\mathbf{i}\in\mathscr{X}_{pot} : \lambda(\mathbf{i}) = w\}$. In practice, we require that the sets $\mathscr{W}_w$ be easily encoded as DDs, thus they usually depend on just a few variables (interestingly, this also bears similarity with the requirements for a good choice of λ). The approach, shown in Fig. 4, is similar in spirit to the explicit one of Fig. 2, except all operations are performed symbolically on DDs, not on individual states. When workstation w explores its set of unexplored states $\mathscr{U}_w$ by applying the next-state function $\mathscr{N}$, it finds both states that it owns (states in $\mathscr{W}_w$) and states that belong to other workstations w' (states in $\mathscr{W}_{w'}$); the latter are sent to the appropriate workstations, encoded as DDs.

Just as in the explicit case, the choice of partition is critical, and even more difficult:

- A balanced partition of potential states $\mathscr{X}_{pot}$ does not imply a balanced partition of reachable states $\mathscr{X}_{rch}$, yet, the slicing windows are defined on the potential state space.
- A balanced partition of the reachable states $\mathscr{X}_{rch}$ does not imply a balanced number of DD nodes, since the number of states encoded by a DD is not directly related to the number of DD nodes.
- Even if $\{\mathscr{W}_1,\ldots,\mathscr{W}_N\}$ are a partition of potential states, thus result in a partition of the reachable states, this does not imply absence of DD node duplication (top of Fig. 5). Indeed, it is obvious that the minimum amount of DD node duplication will occur when the DD is a tree, which is generally the worst case for the application of symbolic approaches (bottom of Fig. 5).

In summary, the goal should be to minimize the sizes of the DDs managed by the N workstations (i.e., minimize the size of the largest DD, or the the sum of the DD sizes), but the vertical partitioning approach, being after all based on partitioning *states* and not *DD nodes*, might fail to achieve this goal.

```
DistributedVerticalBfsSymbolicStateSpaceGeneration(w, 𝒳_init, 𝒩, 𝒵_1, ..., 𝒵_N) is
 1  𝒴_w ← 𝒳_init ∩ 𝒵_w;                                  • known states
 2  𝒰_w ← 𝒳_init ∩ 𝒵_w;                                  • unexplored states
 3  if  𝒰_w = ∅  then
 4      𝒰_w ← GetBddsSentToMeFromOthers(w);              • distributed termination detection...
 5      if 𝒰_w = ∅ then                                  • ...returns an empty set if it is time to terminate
 6          return 𝒴_w;
 7      𝒫_w ← 𝒩(𝒰_w);                                    • potentially new states
 8      𝒰_w ← (𝒫_w ∩ 𝒵_w) \ 𝒴_w;                          • truly new states belonging to w
 9      𝒴_w ← 𝒴_w ∪ 𝒰_w;
10      foreach v ∈ {1,...,w−1,w+1,...,N} do             • distribute (potentially) new states owned by others
11          if 𝒫_w ∩ 𝒵_v ≠ ∅ then
12              SendBddToWorkstation(𝒫_w ∩ 𝒵_v, v);
```

Figure 4: Distributed symbolic state-space generation using vertical slicing.

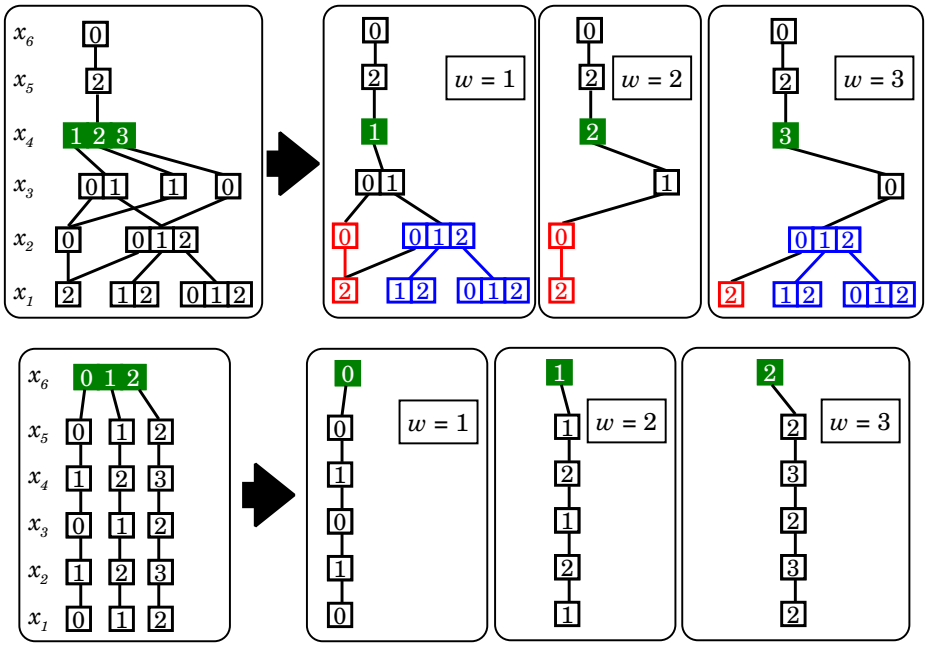

Figure 5: Problems with vertical partitioning.

Nevertheless, some success was exhibited with this approach, through intelligent dynamic workload balancing, which essentially reduces to intelligent choices of windows (and reassessments of these choices). In [27], a slicing method is proposed to achieve balanced slices that, with the help of a "cost function", also keep the number of duplicated DD nodes low after a re-slicing. In [26] a more advanced strategy is proposed to reallocate the task to free process when necessary and release a process when its work load is small. This *work-efficient* approach attempts to adaptively minimize the number of workstations employed at any one point during the fixed point iterations: we start with one workstation and begin state-space generation, then we periodically reassess the situation and, if all workstations have an excessive memory load, we increase N and use a finer slicing, or, if they all have a very light memory load, we decrease N and use a coarser slicing. This optimizes workstation utilization and reduces communication, so it is a good idea (indeed, it could and should be employed also in conjunction with the horizontal partitioning we described next). However, this is just a confirmation that achieving true speedup through parallelization is hard for symbolic methods. Maximizing utilization is not the goal, otherwise we would simply just use one workstation! In practice, a fairly large number of workstations might be available, whether we use them or not, so the real goal is reducing run time for a given memory footprint, or reducing the memory footprint for a given runtime, which is much harder.

Further improvement can be achieved using asynchronous DD exchanges. [26] observed that "the fact that the reachability computation is synchronized in a step-by-step fashion has a major impact on the computation time". Due to the need to exchange non-owned states, processes synchronize after each image computation, and the slowest one determines the speed of the overall computation. To overcome this drawback, a fully asynchronous distributed algorithm is proposed in [25], where processes do not synchronize at each iteration, but instead run concurrently without waiting for each other. The classic two-phase Dijkstra algorithm is then adapted to this framework, where the number of processes can vary dynamically. An "early split" is introduced to utilize free processes and achieve speedup. Compared to the previous approach of [26], improvements of up to a factor of ten are reported for some large circuits.

We now move to consider distributed approaches for the Saturation algorithm. Saturation executes in a node-wise fashion, instead of using heavy global breadth-first iterations, but this also means that Saturation follows a strict order of when firing events on nodes: a node is saturated only after all of its children have been saturated. This policy is quite efficient but difficult to parallelize.

In [11] we adopted *horizontal partitioning* [38] to distribute the Saturation algorithm. Assuming that the number of levels L is at least as large as the number of workstations N, and hopefully much larger, MDDs nodes are distributed to workstations according to their level: workstation w owns a contiguous range of levels $\mathscr{L}_w = \{mytop_w, ..., mybot_w\}$, so that $\{\mathscr{L}_N, ..., \mathscr{L}_1\}$ constitute a partition of the set of levels $\{L, ..., 1\}$ (see Fig. 6). Since Saturation fires event α starting at level $Top(\alpha)$, such an arrangement allows the appropriate workstation to start firing an event, and, if the recursive firing reaches a boundary level, the workstation simply issues a request to continue the operation in the workstation responsible for the next set of levels below, and goes idle, waiting for a reply. The use of quasi-reduced [31] MDDs simplifies the implementation, since it naturally allows us to associate a unique table UT_k and an operation cache OC_k to each level k. Then, workstation w stores and manages $\{UT_{mytop_w}, ..., UT_{mybot_w}\}$ and $\{OC_{mytop_w+1}, ..., OC_{mybot_w+1}\}$. The advantage of horizontal partitioning is clear: absolutely no duplication of nodes or cache entries. Furthermore, memory balance simply requires to reallocate levels by changing boundaries across neighboring workstations and moving the corresponding nodes and cache entries, and it is easy to calculate what the new memory load will be due to such an exchange before performing it. However, as stated, this approach is completely sequential: at any one time, exactly one workstation is performing work, while the others are idle, waiting for results or to start the saturation of nodes at their levels. Thus, any speedup is due to being able to exploit the overall memory of a NOW,

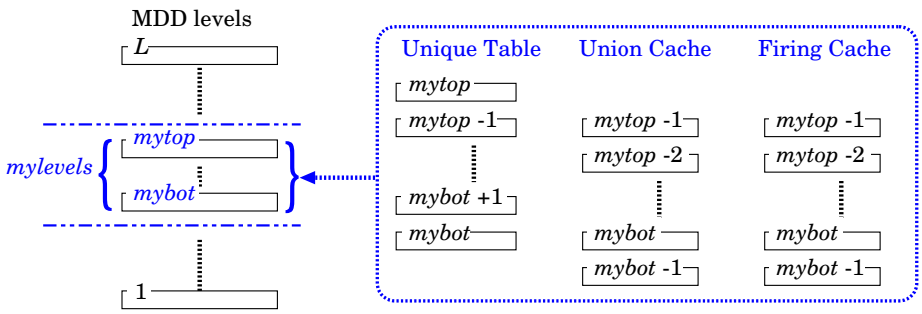

Figure 6: Horizontal partition used in distributed Saturation.

and is observed only when comparing with sequential Saturation running with insufficient memory.

To achieve a true speedup within this horizontal-partitioning Saturation framework, [12] employs *speculative computation*, by using workstations' idle time to perform potentially useful firings. More precisely, we compute the relational product of a node p at level k in the L-level MDD of the current $\mathcal{X}_{reach}$ and a node r at level k of the $2L$-level MDD encoding $\mathcal{N}_\alpha$, where $Top(\alpha) > k$. If, later on, the firing of α on a node at level $Top(\alpha)$ reaches node p, the speculation pays off, as we simply retrieve the result (computed using idle workstation time!) from the cache. However, excessive speculation can easily lead to memory overload, as the unique table and the operation cache may end up containing many useless entries that will not be needed in future. To reduce this problem, [13] associates a *firing pattern* (the set of events non-speculatively fired on a node so far) to each MDD node, and computes a *score* for each speculation to reflect how likely the speculation of an event on a node is to move it toward another pattern. Furthermore, the score can take into account *pattern popularity*, i.e., how many nodes have a particular pattern. With respect to [12], the results in [13] show how patterns can be used to avoid excessive speculation, so that, when speculation does not help speedup the computation, at least it does not harm memory much. Overall, a speedup of up to a factor of two is observed in many models using $N = 8$ workstations, with moderate increase in memory consumption (i.e., a workstation might use up to $1.9/8$ of the memory required when $N = 1$, while, without speculation, the horizontal partitioning uses only little over $1/8$ of the memory required when $N = 1$, thus achieves almost perfect memory balance and no memory overhead, but no speedup at all, actually a slowdown due to communication).

3.2 Shared-memory approaches for symbolic state-space generation

With shared-memory, the main concern shifts from the communication overhead of coarse-granularity processes to the locking and mutual exclusion requirements of fine grained processes. Consider a call $RelProd(p,r)$, that is, a relational product call reaching nodes p and r, both associated with variable x_k, as shown in Fig. 7. The recursion will issue the calls $RelProd(p[0],r[0][0])$, $RelProd(p[0],r[0][1])$, $RelProd(p[1],r[1][2])$, $RelProd(p[2],r[2][1])$, $RelProd(p[2],r[2][2])$, and $RelProd(p[3],r[3][3])$. These can be issued in any order; indeed, they can be run in parallel, if enough cores or processors are available.

However, DDs are not trees, they hopefully contain many recombining paths. Thus, multiple recursive calls may reach the same argument pairs. In a sequential approach, we use the cache to avoid repeating computations. In a shared-memory approach, in addition, we need to use some locking mechanism to avoid all redundant computations; one possible approach is for a process to first insert its *intention* to build a result in the cache, by immediately inserting in the cache a dummy value before initiating a

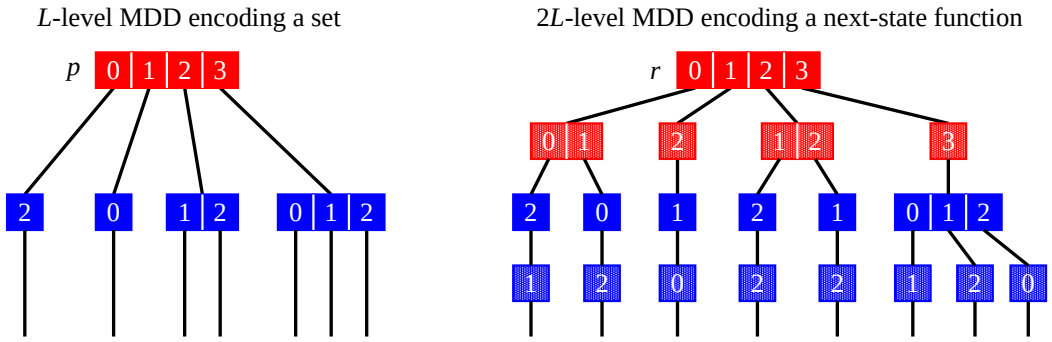

Figure 7: Potential fine-grained parallelism in a relational product computation.

computation, to be substituted by the actual result value later, once it has been computed. Concurrent cache lookups by other processes needing the same result retrieve either this dummy value from the cache (the processes will know that the value is already being computed and will be available soon) or the actual result (as in the sequential case). Of course, processes must issue locks on the unique table and the operation cache; we can avoid excessive serialization by partitioning the unique table and cache (for example by levels), so that the unit of memory being locked is finer, reducing blocking probability.

This works, and can indeed achieve reasonable speedup for symbolic breadth-first state-space generation. However, since Saturation tends to be enormously more efficient than breadth-first iterations, we should take this approach to parallelize Saturation. Ironically, exploring this opportunity led us to further speedup sequential Saturation first [15]. One of the reasons for the efficiency of Saturation is its extensive use of *chaining* [39]: if events α and β can be fired on the set of states $\mathscr{X}$,

$$\mathscr{X} \cup \mathscr{N}_\alpha(\mathscr{X}) \cup \mathscr{N}_\beta(\mathscr{X}) \;\subseteq\; \mathscr{X} \cup \mathscr{N}_\alpha(\mathscr{X}) \cup \mathscr{N}_\beta(\mathscr{X} \cup \mathscr{N}_\alpha(\mathscr{X}))$$

(for the inclusion to be strict, α must add new states that enable β). Then, chaining was proposed as heuristic that looks at the system structure (Petri net, circuit) to derive a good event order so that firing "help compound each other's effect".

In [15], we applied this idea by considering not the structure of the high-level model, but of the MDD itself, when performing a relational product. Let r_k be the MDD encoding $\bigcup_{Top(\alpha)=k} \mathscr{N}_\alpha$. To saturate p at level k, we build its *dynamic transition graph*

$$G_p = (\mathscr{X}_k, \mathscr{T}_p) \quad \text{where} \quad \mathscr{T}_p = \{(i,j) \in \mathscr{X}_k^2 : p[i] \neq \mathbf{0} \wedge r_k[i][j] \neq \mathbf{0} \wedge p[j] \neq \mathbf{1}\}.$$

If the dynamic transition graph has a path from i to j but not from j to i, then we should not issue the call $RelProd(p[j], r_k[j][\cdot])$ until the calls $RelProd(p[\cdot]r_k[\cdot][i])$ have converged (i.e., they cannot add more states). This is not a heuristic, it is *guaranteed* to be optimal

Unfortunately, this observation does not suffice to provide us with a total order on the firings, since the dynamic transition graph may contain strongly-connected components. To "break cycles", we define the *fullness* of node p as $\phi(p) =$ "number of paths encoded by p"$/|\mathscr{X}_k \times \cdots \times \mathscr{X}_1|$, then, under a *uniform distribution assumption*, adding the result of the call $RelProd(p[i], r_k[i][j])$ increases $\phi(p[j])$ by

$$\phi_\Delta \approx |\mathscr{X}_k \times \cdots \times \mathscr{X}_1| \cdot \phi(p[i]) \cdot \phi(r_k[i][j]) \cdot (1 - \phi(p[j]))$$

in expectation. Thus, we call first $RelProd(p[i], r_k[i][j])$ for the pair (i,j) maximizing ϕ_Δ. This heuristic was shown to work quite well in practice, resulting in consistently better run time (up to 4×) and mem-

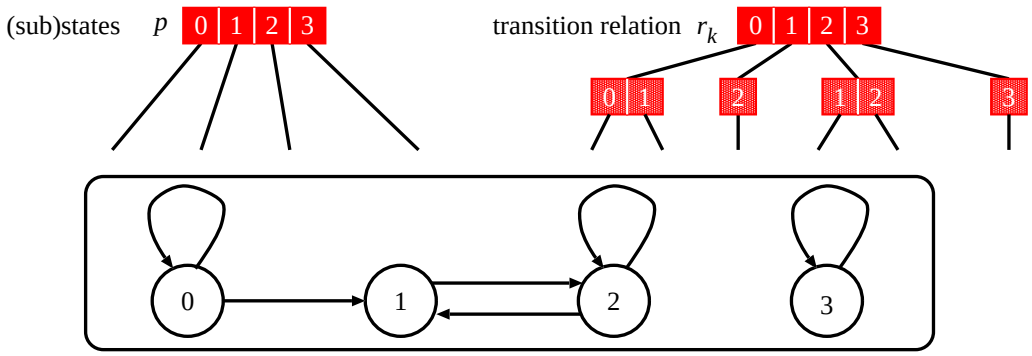

Figure 8: The dynamic firing graph in a relational product computation.

ory (up to 3×) than previous (sequential) implementations of Saturation. Furthermore, the overhead to maintain fullness data and to build and update G_p when saturating p was shown to be negligible.

Unfortunately (from a parallelization perspective), this result improves sequential Saturation by imposing a strict order on the *RelProd* calls. This fine-grained chaining heuristics can help us understand what happens when parallelizing Saturation. If G_p contains no path from i to j and no path from j to i, we can perform some firing in parallel, out of i and out of j, for example (of course, we must use fine locks since the DD is a DAG, not a tree). If G_p contains a path from i to j but not from j to i, we know that we should perform the firing from i on the path to j before firing any event on j, otherwise we may hurt chaining. If G_p contains a path from i to j and a path from j to i, we know we need to break the cycle (even in the sequential case) and may hurt chaining, but parallelization can further hurt chaining.

This was experimentally verified in a Cilk [6] implementation at York University [22] running on a shared-memory multicore processor computer system. We achieved some speedup on some models on a four-core machine, but also experienced substantial slowdowns on many models, when parallelization hurts chaining. Thus, this approach is likely not scalable in the number of cores for practical models.

One intriguing possibility that needs further investigation is that we can obviously fire in parallel on different nodes at the same level. However, it remains to be seen how often this situation can be exploited in practice using Saturation, especially in the common case where $\mathcal{X}_{init}$ contains a single state.

4 Challenges and future goals

As we argued at the beginning of this paper, achieving good speedups is the central goal for future work on parallel symbolic state-space generation and formal verification. From the above discussion, we can summarize the challenge in the following points:

- Finding appropriate workload partitioning.
- Minimizing the synchronization overhead, especially due to global synchronizations.
- Devising efficient mutual exclusive schemes in DD operations.

The first point derives from the fact that DDs are brittle in time and memory consumption during their computation, so that balanced workloads across processes are hard to achieve and maintain. From previous work, we can observe that distributed algorithms achieving good speedups are mainly asynchronous,

while those with global synchronizations are often not as competitive. On the shared-memory side, symbolic algorithms are memory intensive, and frequent accesses to lock-protected data can greatly reduce the potential parallelism.

We believe that the parallelization of Saturation is still a promising, albeit challenging, work. The main open question is how to find more tasks that can be executed in parallel to achieve true speedup. The reported results in [22], which are still far from satisfying, show that there is a subtle trade-off between the parallelism and the level-wise firing order of Saturation. Naïve parallelization of event firings will likely not lead to a faster algorithm, as the order of firing has enormous impact on the performance. Rather, exploring how to extract all the possible parallelism, both at a coarse and at a fine granularity, while respecting the partial order of operations required by the Saturation approach, is likely to offer the greatest payback. In addition, the speculative firing ideas used in [12, 13, 14] might still be helpful to provide further parallelism, by exploiting idle processor or core time.

References

[1] Prakash Arunachalam, Craig Chase & Dinos Moundanos (1996): *Distributed binary decision diagrams for verification of large circuits*. In: *Proc. Int. Conference on Computer Design (ICCD)*. IEEE Comp. Soc. Press, Austin, TX, pp. 365–370.

[2] Jiri Barnat, Lubos Brim & Jakub Chaloupka (2003): *Parallel breadth-first search LTL model-checking*. In: *In 18th IEEE International Conference on Automated Software Engineering (ASE'03)*. IEEE Comp. Soc. Press, pp. 106–115.

[3] Jiri Barnat, Lubos Brim & Jakub Chaloupka (2005): *From distributed memory cycle detection to parallel LTL model checking*. *Electronic Notes in Theoretical Computer Science* 133, pp. 21–39.

[4] Jiri Barnat, Lubos Brim & Petr Rockai (2007): *Scalable multi-core LTL model-checking*. In: *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings, Lecture Notes in Computer Science* 4595. Springer-Verlag, pp. 187–203.

[5] Jiri Barnat, Lubos Brim & Petr Ročkai (2008): *DiVinE Multi-Core — A Parallel LTL Model-Checker*. In: *ATVA '08: Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*. Springer, Seoul, Korea, pp. 234–239.

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall & Y Zhou (1995): *Cilk: An efficient multithreaded runtime system*. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*. ACM Press, pp. 207–216.

[7] Randy E. Bryant (1986): *Graph-based algorithms for boolean function manipulation*. *IEEE Transactions on Computers* 35(8), pp. 677–691.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill & L. J. Hwang (1992): *Symbolic model checking:* 10^{20} *states and beyond*. *Information and Computation* 98, pp. 142–170.

[9] Jerry R. Burch, Edmund M. Clarke & David E. Long (1991): *Symbolic model checking with partitioned transition relations*. In: A. Halaas & P.B. Denyer, editors: *Int. Conference on Very Large Scale Integration*. IFIP Transactions, North-Holland, Edinburgh, Scotland, pp. 49–58.

[10] G. Chiola & Giuliana Franceschinis (1991): *A structural colour simplification in well-formed coloured nets*. In: *Proc. 4th Int. Workshop on Petri Nets and Performance Models (PNPM'91)*. IEEE Comp. Soc. Press, Melbourne, Australia, pp. 144–153.

[11] Ming-Ying Chung & Gianfranco Ciardo (2004): *Saturation NOW*. In: Giuliana Franceschinis, Joost-Pieter Katoen & Murray Woodside, editors: *Proc. Quantitative Evaluation of SysTems (QEST)*. IEEE Comp. Soc. Press, Enschede, The Netherlands, pp. 272–281.

[12] Ming-Ying Chung & Gianfranco Ciardo (2005): *A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation*. In: Leucker Martin & Jaco van de Pol, editors:

Workshop on Parallel and Distributed Model Checking (PDMC), ENTCS. Elsevier, Lisbon, Portugal, pp. 65–79.

[13] Ming-Ying Chung & Gianfranco Ciardo (2006): *A dynamic firing speculation to speedup distributed symbolic state-space generation*. In: Arnold L. Rosenberg, editor: *Proc. International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Comp. Soc. Press, Rhodes, Greece. (electronic proceeding).

[14] Ming-Ying Chung & Gianfranco Ciardo (2009): *Speculative image computation for distributed symbolic reachability analysis*. *Journal of Logic and Computation* .

[15] Ming-Ying Chung, Gianfranco Ciardo & Andy Jinqing Yu (2006): *A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis*. In: Susanne Graf & Wenhui Zhang, editors: *Proc. 4th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, LNCS 4218. Springer-Verlag, Beijing, China, pp. 51–66.

[16] Gianfranco Ciardo, Joshua Gluckman & David Nicol (1998): *Distributed state-space generation of discrete-state stochastic models*. *INFORMS J. Comp.* 10(1), pp. 82–93.

[17] Gianfranco Ciardo, Gerald Lüttgen & Radu Siminiceanu (2001): *Saturation: An efficient iteration strategy for symbolic state space generation*. In: Tiziana Margaria & Wang Yi, editors: *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2031. Springer-Verlag, Genova, Italy, pp. 328–342.

[18] Gianfranco Ciardo, Andrew S. Miner & Min Wan (2009): *Advanced features in SMART: the Stochastic Model checking Analyzer for Reliability and Timing*. *ACM SIGMETRICS Perf. Eval. Rev.* 36(4), pp. 58–63.

[19] Gianfranco Ciardo & Radu Siminiceanu (2002): *Using edge-valued decision diagrams for symbolic generation of shortest paths*. In: Mark D. Aagaard & John W. O'Leary, editors: *Proc. Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, LNCS 2517. Springer-Verlag, Portland, OR, USA, pp. 256–273.

[20] Gianfranco Ciardo & Andy Jinqing Yu (2005): *Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning*. In: Dominique Borrione & Wolfgang Paul, editors: *Proc. CHARME*, LNCS 3725. Springer-Verlag, Saarbrücken, Germany, pp. 146–161.

[21] E. M. Clarke & E. A. Emerson (1981): *Design and synthesis of synchronization skeletons using branching time temporal logic*. In: *Proc. IBM Workshop on Logics of Programs*, LNCS 131. Springer-Verlag, London, UK, pp. 52–71.

[22] Jonathan Ezekiel, Gerald Lüttgen & Gianfranco Ciardo (2007): *Parallelising symbolic state-space generators*. In: Werner Damm & Holger Hermanns, editors: *Computer Aided Verification (CAV'07)*, LNCS 4590. Springer-Verlag, Berlin, Germany, pp. 268–280.

[23] S. Gai, M. Rebaudengo & M. Sonza Reorda (1995): *A data parallel algorithm for boolean function manipulation*. In: Joel Saltz & Dennis Gannon, editors: *Frontiers of Massively Parallel Scientific Computation (FMPSC)*. National Aeronautics and Space Administration (NASA), IEEE Comp. Soc. Press, pp. 28–36.

[24] Patrice Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag.

[25] Orna Grumberg, Tamir Heyman, Nili Ifergan & Assaf Schuster (2005): *Achieving Speedups in Distributed Symbolic Reachability Analysis Through Asynchronous Computation*. In: Dominique Borrione & Wolfgang Paul, editors: *Proc. Correct Hardware Design and Verification Methods (CHARME), LNCS* 3725. Springer-Verlag, Saarbrücken, Germany, pp. 129–145.

[26] Orna Grumberg, Tamir Heyman & Assaf Schuster (2003): *A work-efficient distributed algorithm for reachability analysis*. In: Jr. Warren A. Hunt & Fabio Somenzi, editors: *Computer Aided Verification (CAV)*. Boulder, CO, USA, pp. 54–66.

[27] Tamir Heyman, Danny Geist, Orna Grumberg & Assaf Schuster (2002): *A scalable parallel algorithm for reachability analysis of very large circuits*. *Formal Methods in System Design* 21(3), pp. 317–338.

[28] Gerard J. Holzmann (2003): *The SPIN Model Checker*. Addison-Wesley.

[29] Cornelia P. Inggs & Howard Barringer (2002): *Effective state exploration for model checking on a shared memory architecture*. In: *Workshop on Parallel and Distributed Model Checking (PDMC'02)*, ENTCS 68/4.

Brno, Czech Republic.

[30] K. L. McMillan (1992): *The SMV system, symbolic model checking - an approach*. Technical Report CMU-CS-92-131, Carnegie Mellon University.

[31] Shinji Kimura & Edmund M. Clarke (1990): *A parallel algorithm for constructing binary decision diagrams*. In: *Proc. Int. Conf. on Computer Design (ICCD)*. IEEE Comp. Soc. Press, Cambridge, MA, pp. 220–223.

[32] Flavio Lerda & Riccardo Sisto (1999): *Distributed-memory model checking with SPIN*. In: *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. Springer-Verlag, London, UK, pp. 22–39.

[33] K. L. McMillan (1993): *Symbolic Model Checking*. Kluwer.

[34] Kim Milvang-Jensen & Alan J. Hu (1998): *BDDNOW: A parallel BDD package*. In: Ganesh Gopalakrishnan & Phillip J. Windley, editors: *Proc. International Confrence on Formal Methods in Computer-Aided Design (FMCAD), LNCS* 1522. Springer-Verlag, Palo Alto, California, USA, pp. 501–507.

[35] David Nicol & Gianfranco Ciardo (1997): *Automated parallelization of discrete state-space generation*. *J. Par. and Distr. Comp.* 47, pp. 153–167.

[36] Hiroyuki Ochi, Nagisa Ishiura & Shuzo Yajima (1991): *Breadth-first manipulation of SBDD of Boolean functions for vector processing*. In: *DAC '91: Proceedings of the 28th ACM/IEEE Design Automation Conference*. ACM, New York, NY, USA, pp. 413–416.

[37] Y. Parasuram, E. Stabler & Shiu-Kai Chin (1994): *Parallel implementation of BDD algorithms using a distributed shared memory*. In: *The 27th Hawaii International Conference on System Sciences (HICSS'94)*, 1. IEEE Comp. Soc. Press, Maui, HI, USA, pp. 16–25.

[38] R. K. Ranjan, J. V. Snaghavi, R. K. Brayton & A. Sangiovanni-Vincentelli (1996): *Binary decision diagrams on network of workstations*. In: *Proc. Int. Conference on Computer Design (ICCD)*. IEEE Comp. Soc. Press, Austin, TX, pp. 358–364.

[39] O. Roig, J. Cortadella & E. Pastor (1995): *Verification of asynchronous circuits by BDD-based model checking of Petri nets*. In: Giorgio De Michelis & Michel Diaz, editors: *Proc. 16th Int. Conf. on Applications and Theory of Petri Nets*, LNCS 935. Springer-Verlag, Turin, Italy, pp. 374–391.

[40] Ulrich Stern & David L. Dill (1997): *Parallelizing the Murϕ verifier*. In: Orna Grumberg, editor: *Proc. International Conference on Computer Aided Verification (CAV), LNCS* 1254. Springer-Verlag, Haifa, Israel, pp. 256–278.

[41] Anthony L. Stornetta (1995): *Implementation of an Efficient Parallel BDD Package*. Master's thesis, University of California, Santa Barbara.

[42] Anthony L. Stornetta & Forrest Brewer (1996): *Implementation of an efficient parallel BDD package*. In: *Proc. Design Automation Conference (DAC)*. ACM Press, Las Vegas, NV, pp. 641–644.

[43] A. Valmari (1991): *A stubborn attack on the state explosion problem*. In: *CAV '90*. Springer-Verlag, pp. 156–165.

[44] Min Wan & Gianfranco Ciardo (2009): *Symbolic reachability analysis of integer timed Petri nets*. In: M. Nielsen et al., editors: *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404. Springer-Verlag, Špindlerův Mlýn, Czech Republic, pp. 595–608.

[45] Min Wan & Gianfranco Ciardo (2009): *Symbolic state-space generation of asynchronous systems using extensible decision diagrams*. In: M. Nielsen et al., editors: *Proc. 35th Int. Conf. Current Trends in Theory and Practice of Computer Science (SOFSEM)*, LNCS 5404. Springer-Verlag, Špindlerův Mlýn, Czech Republic, pp. 582–594.

[46] Bwolen Yang & David R. O'Hallaron (1997): *Parallel breadth-first BDD construction*. In: *6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, SIGPLAN Notices 32(7). ACM Press, Las Vegas, NV, USA, pp. 145–156.

[47] Yang Zhao & Gianfranco Ciardo (2009): *Symbolic CTL model checking of asynchronous systems using constrained saturation*. In: Zhiming Liu & Anders P. Ravn, editors: *Proc. 7th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, LNCS 5799. Springer-Verlag, Macao SAR, China, pp. 368–381.