A General Lock-Free Algorithm for Parallel State Space Construction

Rodrigo Tacla Saad, Silvano Dal Zilio, Bernard Berthomieu

Parallel State Space Construction for NUMA architecture *

Rodrigo T. Saad, Silvano Dal Zilio, Bernard Berthomieu and François Vernadat

CNRS; LAAS;
7, avenue du Colonel Roche, F-31077 Toulouse – France
Université de Toulouse;
UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
{rsaad, dalzilio, bernard, francois}@laas.fr

Abstract. Verification via model-checking is a very demanding activity in terms of computational resources. While there are still gains to be expected from algorithmic methods, it is necessary to take advantage of the advances in computer hardware to tackle bigger models. Recent improvements in this area take the form of multiprocessor and multicore architectures with access to large memory space.
We address the problem of generating the state space of finite-state transition systems; often a preliminary step for model-checking. We propose a novel algorithm for enumerative state space construction targeted at Non-Uniform Memory Access (NUMA), that is multiprocessor architectures where the latency and bandwidth characteristics of memory actions depend on the processor or memory region being accessed. Our approach relies on the use of a shared Bloom filter to coordinate the state space exploration distributed among several processors. The goal is to limit undesired synchronizations and increase the locality of memory access. Bloom filters have already been applied for the probabilistic verification of system; they are compact data structure used to encode sets, but in a way that false positive are possible, while false negative are not. We circumvent this limitation and propose an original multiphase algorithm to perform exhaustive, deterministic, state space generations.

1 Introduction

Verification via model-checking is a very demanding activity in terms of computational resources. While there are still gains to be expected from algorithmic methods, it is necessary to take advantage of the advances in computer hardware to tackle bigger models. Obviously, the use of a parallel architecture is helpful to cut the time needed to check a model because it divides the computation over several processing units instead of one. What is even more important, though, is the possibility to access a large amount of fast-access memory. Indeed, model-checking is very space-consuming and cannot realistically make use of disk-based memory storage.

* This work has been supported by the French AESE project Topcased and by region Midi-Pyrénées

We address the problem of generating the state space of finite-state transition systems, often a preliminary step for model-checking. We propose a novel algorithm for enumerative state space construction targeted at Non-Uniform Memory Access (NUMA), that is multiprocessor architectures where the latency and bandwidth characteristics of memory actions depend on the processor or memory region being accessed. Practically, this means that the shared, addressable memory space is divided into several regions, reachable through physically different buses. To give actual figures regarding our algorithm, we have tested our approach on a high-end server configured with 8 dual core opteron processors, equipped with 208GB of RAM memory. (It is unlikely that commercially viable, single processor computers with that amount of RAM could be available in the close future.) For the experiments detailed in this paper, where we work with an explicit representation of the state space, this configuration allows us to work with models generating more than 500 millions states and to divide the computation time by a factor of about 6 to 8.

The basic idea behind a state space construction algorithm is pretty simple: take a state that has not been explored (a fresh state); compute its successors and check if they have already been found before; iterate. A key point is to use an efficient data structure for storing the set of generated states and for testing membership in this set. Our approach relies on the use of a shared Bloom Filter to implement this membership test and to coordinate the exploration distributed among several processors. We take advantage of the fast response time and space efficiency of Bloom filter in order to limit undesired synchronizations and increase the locality of memory access. Bloom filters have already been applied for the probabilistic verification of systems; they are compact data structure used to encode sets, but in a way that false positive are possible, while false negative are not. We circumvent this limitation and propose an original multiphase algorithm to perform exhaustive, deterministic, state space generations. In the first phase (*exploration*), the algorithm is guided by the Bloom filter until we run out of states to explore. During this phase, states found by a processor are stored locally in two AVL trees: one for states that, according to the Bloom filter, have already been generated by another processor; another for fresh states. Since the Bloom filter may, in rare cases, falsely report that a state has already been visited (what is called a false positive), we need to give a special treatments to these *collision* states. This is done in the consecutive phase (*collision resolution*) that takes care of collisions among possible false positive. The algorithm concludes with a *termination detection* phase when there are no more states to explore and no collisions.

The rest of this paper is organized as follows. In Section 2 we review related work and give a brief introduction on Bloom Filters. Section 3 give the details of our algorithms. In Section 4, we examine experiments performed on a set of typical benchmarks. We conclude with ideas for future extensions of our work.

2 Related Work

There is already a large body of work addressing the problem of parallelizing and distributing state space construction. Several solutions have been proposed that are each tailored to a particular type of parallel and distributed architecture. The vast majority of these solutions adopt a common approach, that could be labeled as "homogeneous" parallelism, which follows a Single Program Multiple Data (SPMD) programming style, such that each processor performs the same steps concurrently. (To the best of our knowledge, the only work following the Single Instruction Multiple Data(SIMD) model is [5].) A drawback of the SPMD model, which is commonly used to accomplish coarse-grained parallelism, is that data and computations should be explicitly assigned to each processor. It is therefore necessary to set up an efficient load-balancing mechanism to improve the speedup of the implementation.

Slicing Functions and the Work Stealing Paradigm. A common approach to assign work and data is to partition the state space into several chunks, one for each processor available, through a slicing function. This scheme is more generally applied on distributed memory systems, where solutions mostly differ by the nature of the slicing function, i.e. static or dynamic. Several of the mechanisms proposed for distributed architectures [1, 6, 8, 9, 13, 14] rely on slicing functions and differ basically by the nature of this function in order to provide both locality and balance. Balance can be measured as spatial or temporal balance: spatial balance means that each processor will receive an equal amount of states; temporal balance means that each processor will be busy most of the time. Locality measures the fact that states which are "related" during the computation should be assigned to nearby processes (typically, the successors of a state should be handled by the same processor). Locality is desired to reduce communication overheads.

In contrast with distributed memory systems, shared memory systems abstract away from the need to explicitly pass messages between processors. As a consequence, mechanisms proposed for these systems do not require a slicing function to assign states to processors, since they can be all shared. However, for ensuring data consistency, shared memory systems incur synchronization overheads on operations that perform concurrent access to the memory. Consequently, solutions developed for shared memory systems often rely on a pool of "local memory", assigned to each processor, along with customized synchronization mechanisms to guarantee a consistent access to a shared data structure that stores the bulk of the state space. In this context, to achieve high degree of parallelism, the goal is to keep to a minimum the part of global data that is locked for mutual exclusion. Allmaier et al. [1] were among the first to implement a parallel state space construction algorithm for shared memory systems. In their design, states are stored in a balanced-tree and the data consistency problem is solved by using locks together with a "splitting-in-advance" scheme to reduce the contention on data locks. In [10], the authors propose a parallel

algorithm for state exploration based on a *work stealing* scheduling paradigm to provide dynamic load balancing without a blocking phase. The idea is that underemployed processors attempt to "steal" work from other processors. In this paper, processing nodes store pending states (states that have potentially unprocessed successors) in two local queues: a private queues for states they will process; a shared queue for states that can be appropriated by other nodes. A global hash table is used to store already visited states. Every time the private queue of a process is empty, it has to acquire a lock to check over its own shared queue for a pending state. If no state is found, the processor starts searching through all other shared queues until it finds a nonempty queue or finds that all shared queues are empty. With reference to the storage data, unlike [1], this work implements a hash table without any mutual exclusions locks to synchronize access. The authors emphasize that the duplication caused by the lack of a locking strategy is not relevant compared to the parallel computation power available.

Bloom Filter in Model Checking Applications. Explicit (or enumerative) model checking suffers from the well known state explosion problem. This problem has direct implication on the choice of the data structure to store the state space since the amount of memory required depends on the number of reachable states. When the state space is too large, it may be interesting to store states in a probabilistic data structure in order to spare memory space. In this context, probabilistic means that testing the data structure for membership returns the "correct result" with some (hopefully high) probability. Obviously, the drawback of this approach is that it is not possible to have full confidence on the outcome of model checking, since the actual state space may not be completely explored. Nonetheless, a "probabilistic verification result" may still be helpful to find errors in a model and some model-checking tools provide this facility. Usually, probabilistic model checkers use one of two data structures, *compacted hash* and *Bloom Filter*. Choosing the right data structure depends on a priori knowledge on the state space [7]: when the state space size is known, the best choice is the compact hash, otherwise a Bloom filter may result in a better coverage of the state space.

A *Bloom filter* is a space-efficient data structure for encoding set membership that is very popular in database and network applications. General theoretical results on Bloom filters are given in [3], while [7] focus more on their use for probabilistic verification. Bloom filters support two operations: insertion of an element in the set and test that an element is in the set. A filter B of size n is implemented as a vector of n bits and is associated with a series of k independent hash function $(h_i)_{i\in 1..k}$ with image in the interval $1..n$. An empty set is represented by a vector with all bits set to 0. Insertion of the element x in B is performed by setting the bits $h_i(x)$ of the vector to 1 for all i in $1..k$. Reciprocally, to query wether an element y is in B, we test that the bits $(h_i(y))_{i\in 1..k}$ are all set to 1 in the vector. If it is not the case, then we are sure that y is not in the set encoded by B. If all these bits are set to 1, then we

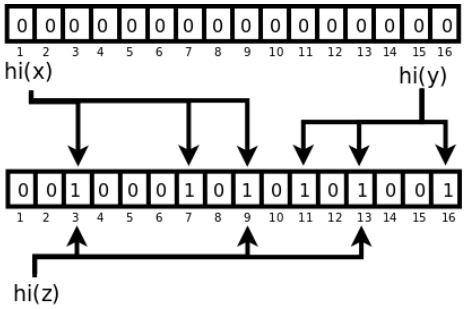


Fig. 1. Illustration of some operations on a Bloom filter.

only have a probabilistic result: in the case where y is actually not in the set, we say we have a *false positive*. The probability of false positive is a function of the size, n, number of hash functions, k, and number of elements inserted so far. Hence the parameters n and k should be carefully chosen in an implementation. Figure 1 illustrates insertion and query operations on a Bloom filter with size $n = 16$ and $k = 3$. Starting from an empty set (above), we show the result after the insertion of two elements, x and y. Element z is an example of false positive.

3 Description of the Parallel Algorithm

Our algorithm elaborates on the work-stealing paradigm and the "homogeneous" parallelization approach introduced in the previous section. Work is distributed homogeneously between processors and each processor handles its own local view of the state space. This allows us to take into account the locality constraint imposed by the NUMA architecture.

Coordination between the processors is based on a shared Bloom filter used to test wether a state has (potentially) already been visited by some of the processors. All states are stored locally in two AVL trees; more details about these data structures are given in Section 3.1. In Section 3.2, we discuss the work-sharing techniques used in our algorithm. Indeed, the processors may share work in two manners, either a passive or an active way. The active way is the work-stealing paradigm we already mentioned, that is triggered when a processor runs out of work. We add a passive way of sharing, that is when a given processor explicitly wakes up a sleeping processor in order to share some work. We use these two techniques alternately according to the amount of work in the system. To conclude the section, we discuss the three phases of our algorithm.

In the remainder of the text, we assume that there are N processors and that each processor is given a unique *id*, which is an integer in the interval $0..N-1$.

3.1 Shared and Local Data

Our objective is to design a solution adapted to NUMA architectures. Hence, in addition to the common difficulties related to shared memory architecture, like ensuring data consistency and reducing contention on shared data access, we should also consider the variations in latency between access to different memory regions. To improve locality, states generated by a processor are stored in one of two possible local AVL trees, the *state tree* or the *collision tree*. This corresponds to one of the two following cases. Assume that processor i generates a new state s. If a query on the Bloom filter answers that s has not been visited before, the processor may continue generating new states from s. In this case we add s to the state tree of processor i. If the query is negative, we add s to the collision tree. States in the collision tree will undergo a special treatment to take into account possible false positives.

Each processor also manages two stacks of unexplored states for work-sharing: one for local work; the other for sharing work with idle processors. Finally, in order to detect termination, we also manage a shared vector that stores the current state of processors (either idle or busy). Figure 2 illustrates the shared and local data structures used in the algorithm.

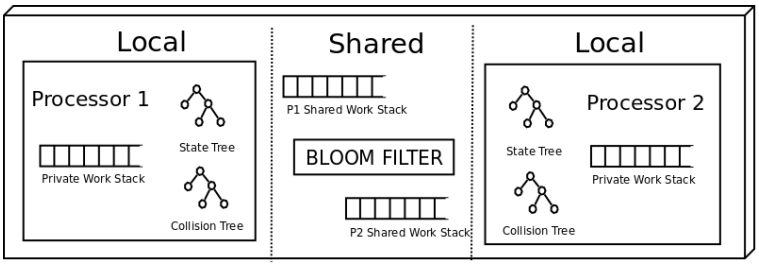


Fig. 2. Shared and Private Data Model Scheme.

3.2 Work-Sharing Techniques

Our algorithm relies on two different work-sharing techniques to balance the working load between processors. We use these mechanisms alternately during the exploration phase in accordance with the processor occupancy. First, we use an *active* technique very similar to the work-stealing paradigm of [10]. This mechanism uses two stacks: a private stack that holds all states that should be worked upon; a shared stack for states that can be borrowed by idle processors. The shared stack is protected by a lock to take care of concurrent access. The second technique can be described as *passive* and has the benefit to avoid useless synchronization and contention caused by the active technique. In the passive mode, an idle processor waits for a wake-up signal from another processor willing

to give away some work instead of polling other shared stacks. The shift between the passive and active modes is governed by two parameters:

- the private minimum workload (*pr_work_load*), which defines the minimal charge of work that should be kept private. The processor will share work only if the charge in its private stack is larger than *pr_work_load*;
- the share workload (*sh_work_load*), which defines the ratio of work that should be added in the shared stack if the load in the private stack is larger than *pr_work_load*.

Our implementation of the work-stealing paradigm differs from [10] by its use of unbounded shared stacks and the *sh_work_load* parameter.

3.3 Different Phases of the Algorithm

As mentioned before, our solution makes use of a shared Bloom filter to test whether a state may have already been discovered before. To overcome the problem with false positives, our algorithm iterates between an exploration phase and a collision resolution phase before concluding with a termination detection phase.

The exploration phase takes great advantage of the strong points of a multiprocessor architecture because the shared space is small and all work is done locally. On the opposite, the collision resolution phase put a lot of stress on the architecture: each processor has to compare the elements in its collision tree with the state tree of all the other processors. As a consequence, the goal is to favor the exploration phase and to reduce the number of iterations. Figure 3 shows the characteristic timeline of phase alternations that we are aiming at. Since iterations are directly related to the probability of false positive, it is important to correctly dimension the Bloom filter. In our experiments, we typically observe less than 3 iterations.

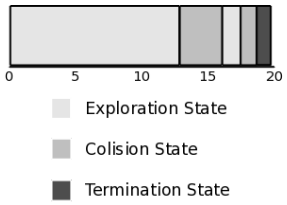


Fig. 3. Timeline of states alternation.

In the remaining of this section, we define each phase of our algorithm using pseudo-code. Variable *SS* indicates the current phase of the algorithm. The data structures used in the algorithm is composed of shared and local elements. Shared variables are: (1) the Bloom Filter *BF*, used to test whether a state had already been discovered or not; (2) the bitvector *V*, that stores the state of the processor (0 for idle and 1 for busy); and (3) the shared stacks *Shared_Stack*[0], ...,

Shared_Stack[N-1]. Processor-local variables are the private stack, *private_stack*, of unexplored states and the two local AVL: *state_tree*, to store states discovered by this processor; and *collision_tree*, to store potential false positive.

Exploration. The exploration phase proceeds until no new states can be added to the Bloom filer *BF*. During the exploration, all states appointed by *BF* as already discovered are stored locally in the *collision_tree* . On the opposite, all newly discovered states are stored locally in the *state_tree*. Computation switches to the collision resolution phase when all processors are idle and there is at least one non-empty local *collision_tree*. After a complete iteration, not resolved collisions (false positive) are marked with a special tag because they bypass the *BF* membership test at this phase. More information about not resolved collisions is presented later.

```
while SS == Exploration and at least one process is busy do
  while private_stack is not empty do
    s := pop(private_stack) ;
    if s is not in BF or s is marked with a special tag then
      search_and_insert s into state_tree ;
      let s1, .. , sj, ..., sn = successors(s) where
        j = shared_work_load x n
        if size(private_stack) > private_work_load then
          // Share a percentage of new work
          if some processor is sleeping then wake him up endif
            // Protected action by locks
            insert s1, ..., sj        in my shared_stack
            insert sj+1, ..., sn  in my private_stack
        else
            insert s1, ..., sn    in my private_state
        endif
      endlet
    else search_and_insert s into collision_tree
    endif
  endwhile
  // private stack empty
  if my shared stack is not empty then
    transfer work from my shared stack to private_stack
  else
    look for a non empty shared_stack to transfer work ;
    if all shared_stacks empty and at least one processor busy
    then enter into sleep mode
    endif
  endif
endwhile
// Everybody is idle
// Protected action by locks
SS := Collision Resolution ;
wake up all processors and enter Collision Resolution phase
```

Collision Resolution. The search for collisions (the same state generated in two distinct processors) is done concurrently by each processor through the synchronization of its *collision_tree* with every non-empty *state_tree*. Since states are already sorted (states can be lexicographically sorted and are stored in an AVL), collisions can be efficiently resolved by comparing all trees as lexicographic ordered lists starting by the leftmost state of each tree. The advantage of this approach is that if a colliding state s is smaller than a given state of a *state_tree*, no more states of this *state_tree* need to be compared with s. During this synchronization, a state from the collision tree that is in the state tree of another processor, say P_i, can be safely omitted: it is a "real" collision and it will be eventually processed by P_i. If the state does not appear in any state trees then its presence in the collision tree is the result of a false positive in the Bloom filter. As a consequence, it will be directly inserted into the private stack of the processor to be expanded during the following exploration phase. We will also mark this state with a special tag to avoid testing him against the Bloom filter a second time. For this reason, if more than one processor find the same false positive, it will result in duplicated states in state space.

```
leftmost[0..N] :=  leftmost states from state_tree [0..N] ;
not_larger[0..N] := {true,...,true} ;
found := false ;
collision :=  leftmost state from collision_tree ;
while collision is not empty do
  forall i in 0..N do
    if not_larger[i] then
      if collision is smaller than leftmost[i] then
        // No more comparisons for this collision
        not_larger[i]:=false
      elsif collision is larger than leftmost[i] then
        leftmost[i] := next ordered element from state_tree[i]
      else // collision == leftmost[i]
        found := true
      endif
    endif
  endfor
  if found is false then
    insert collision into private_stack and marked as an special state
  endif
  collision := next ordered element from collision_tree
endwhile
// No more collision to resolve
if private_stack is not empty then
  // Protected action by locks
  SS := Exploration
endif
if one processor is still busy then
  enter into sleep mode
else
```

```
    wake up every processor ;
    if SS == Exploration then enter Exploration phase
    else enter Termination Detection phase endif
  endif
```

Termination Detection. This phase is responsible for checking if the state space construction should end. Termination detection performs a simple test on the states of the processor and consumes no resources. Assume we arrive in the termination detection phase from the exploration phase. We can finish the construction if the collision_tree in all processors are empty. In the case we arrive in this phase from the collision resolution phase. Then we can finish the construction if the private_stack of all processors are empty.

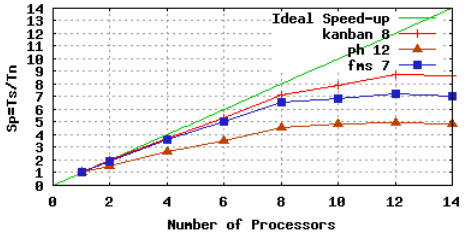
4 Experiments

We implemented our algorithm using the C language with Pthreads [4] for concurrency and the Hoard Library [2] for parallel memory allocation. We used an off-the-shelf implementation for Bloom Filter, namely the Bob Jenkins hash function [11]. Experimental results presented in this section were obtained on a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208GB of RAM memory, running the Solaris 10 operating system. When not specified, we worked with a 512MB Bloom filter ($n = 4.10^9$ bits) and 6 chained hash-functions ($k = 6$).

The finite state systems chosen for our benchmarks are classical examples of Petri Nets taken from [12]. Together with the perennial Dining Philosophers, we also study the examples of the Flexible Manufacturing System (FMS) and the Kanban System, where the first one is parameterized by the number of subnets and the two following ones by the weights in their initial marking. We give several results detailing the performance of our implementation. While speedup is the obvious criteria when dealing with parallel algorithm, we also study the memory tradeoff of our approach and report on experiments carried out to choose the dimension of the Bloom filter.

Speedup. The speedup achieved by a parallel algorithm is measured by dividing the time spent using only one processor (the sequential time, T_s) by the time spent using N processors (T_N). Figure 4 gives the observed speedup of our algorithm when generating the state space for 12 philosophers, FMS 8 and Kanban 8 with different number of processors. Clearly, the algorithm is very dependent on the "degree of concurrency" of the model: it is not necessary to use lots of processors for a model with few concurrent actions. This is an inherent limitation of parallel state space construction algorithm.

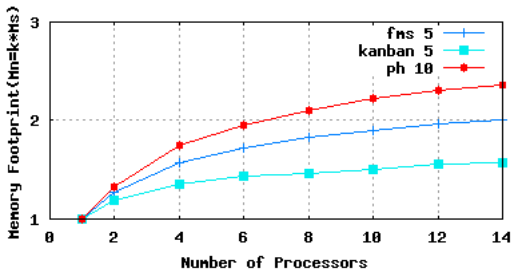
Memory tradeoff. A parallel algorithm often trades additional memory space for better execution time. Figure 5 gives results on the memory used for 10



	speedup		
# proc.	philosopher 12 (3.10^7 states)	FMS 8 (6.10^7 states)	Kanban 8 (13.10^7 states)
2	1.5	1.9	2
4	2.7	3.6	3.7
6	3.6	5	5.3
8	4.6	6.6	7.1
10	4.9	6.9	7.9
12	5	7.21	8.7
14	4.8	7.03	8.7

Fig. 4. Speedup analysis for 12 philosophers, FMS 8 and Kanban 8.

philosophers, FMS 5 and Kanban 5. The graph shows the ratio between the memory actually used and the memory needed by the sequential algorithm. The table concentrates on the FMS 5 example and shows that the increase in the memory footprint is related to the increased number of *collision nodes* (see Section 3.1). The intuition behind these numbers is quite simple: due to the strong symmetry of the example, if we add more processors, we increase the probability of different processors finding the same state, that is the probability of creating a collision node. As seen in the experiments, our algorithm may require twice as much memory than the sequential algorithm in the worst case (14 processors on an example with lot of symmetries). This illustrates the necessity to tune the parameters of our algorithm. The outcome of the experiments presented here were achieved choosing a 512MB Bloom filter and a value of 70% for the share work load, which seems to give consistently good results on many examples. The following sections illustrates the tuning of these two parameters.



	Flexible Manufacture System 5 (29.10^5 states)	
# proc.	# collision tree nodes	total memory (MB)
1	–	906
2	24.10^5	1148
4	50.10^5	1425
6	63.10^5	1559
8	72.10^5	1653
10	79.10^5	1716
12	85.10^5	1777
14	89.10^5	1820

Fig. 5. Memory tradeoff analysis for 10 philosophers, FMS 5 and Kanban 5.

Work Sharing Policy. The sharing of work between processors in our algorithm is parameterized by the value of *sh_work_load*, the share work load (see Section 3.2), that defines the ratio of work that should be inserted inside the shared stack. In all the experiments that we have performed, the best results (speedup) were achieved using a value between 50% and 70%. Figure 6 presents the speedup obtained for different values of this parameter when running on the Kanban 7 system with 14 processors.

sh_work_load (ratio)	time (s)	speedup	# duplicated states
10%	2058	1	0
30%	331	6.2	42
50%	246	8.3	62
70%	252	8.1	76
90%	326	6.3	69

Fig. 6. Impact of the *sh_work_load* parameter on the Kanban 7 example with 14 processors (41.10^6 states, 450.10^7 transitions).

Size of the Bloom Filter. The *exploration* phase of our algorithm ends when, according to the Bloom filter, there is no more state to be explored. Hence, depending on the size of the Bloom filter, our algorithm may prematurely start its *collision resolution* phase due to the high rate of false positives. Consequently, it will result in a higher number of duplicated states. Figure 7 shows the results obtained by executing the Kanban 7 model with three different sizes for the Bloom filter. The table also shows the ratio of time spent in the *exploration* and *collision resolution* phases. We can observe that the speedup degrades when the ratio of time spent in the collision resolution phase increases.

5 Conclusions and Future Work

We propose a new algorithm for parallel state space construction targeted at NUMA architecture. We use a Bloom filter for the shared data structure and define a multiphase algorithm to obtain an exhaustive, deterministic result.

In the context of our experiments, we worked more specifically with system described by Petri Nets. Nonetheless, our algorithm is quite general and could be applied to different formalisms for describing finite transition systems (or finite abstractions of infinite-state models): we only require a simple way to represent states and a function to generate successors. While we provide an implementation that work with an explicit representation of states, our algorithm can be applied alongside traditional optimisations for reducing the state space size, such as

# proc.	Bloom size (MB)	time (s)	speedup	# duplicated states	exploration ratio	collision ratio
2	128	973	2.1	12031	97%	3%
	256	1005	2	331	96%	4%
	512	986	2.1	12	97%	3%
4	128	564	3.6	20067	93%	7%
	256	582	3.5	521	92%	8%
	512	566	3.6	23	93%	7%
6	128	394	5.2	23308	89%	11%
	256	395	5.2	686	90%	10%
	512	385	5.3	30	89%	11%
8	128	311	6.6	26177	84%	16%
	256	315	6.5	765	82%	18%
	512	313	6.5	38	84%	16%
10	128	267	7	31217	87%	13%
	256	271	7	795	79%	21%
	512	264	7	63	77%	23%
12	128	249	8.2	29604	71%	29%
	256	253	8.1	896	68%	32%
	512	254	8.1	66	69%	31%
14	128	257	8	32535	59%	41%
	256	243	8.4	902	63%	37%
	512	247	8.3	88	63%	37%

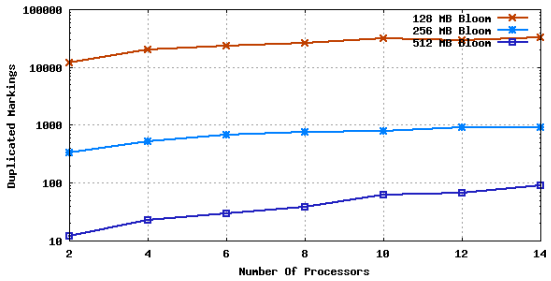


Fig. 7. Impact of the Bloom filter size on the Kanban 7 example.

symbolic approaches or partial-order techniques. Our algorithm takes a black-box approach and is orthogonal to the representation details of the state space.

The experiments conducted with the preliminary implementation of our algorithm shows promising speedups on a set of typical benchmarks. While the performance of the algorithm depends on the "geometry" of its input, e.g. its concurrency degree, we have consistently obtained good results. For example, we have routinely observed speedup of 8 using 10 processors while using less than the double of the memory footprint required by a straightforward sequential implementation.

Experiments carried out to find the appropriate size for the Bloom filter have shown that the performance of our approach is also impacted by the time spent in the the collision resolution phase of our algorithm, that is by the number of duplicated states. We have shown empirically that the best results are achieved when at least two thirds of the computation time is spent in the exploration phase. As a consequence, for the examples studied in this paper, it is impractical to scale our solution to more than 16 processors. For this reason, we are currently studying an asynchronous version of our algorithm in which each processor would asynchronously alternate between exploration and collision resolution phases without blocking each other.

References

[1] Allmaier, S., Kowarschik, M., Horton, G.: State space construction and steady-state solution of GSPNs on a shared-memory multiprocessor. In: Workshop on Petri Nets and Performance Models (1997)

[2] Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A scalable memory allocator for multithreaded applications. ACM SIGPLAN Notices 35(11) (2000)

[3] Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4) (2004)

[4] Butenhof, D.: Programming with POSIX threads. Addison-Wesley (1997)

[5] Caselli, S., Conte, G., Bonardi, F., Fontanesi, M.: Experiences on SIMD massively parallel GSPN analysis. In: Computer Performance Evaluation Modelling Techniques and Tools. LNCS, vol. 794. Springer (1994)

[6] Ciardo, G., Gluckman, J., Nicol, D.: Distributed state space generation of discrete-state stochastic models. INFORMS Journal on Computing 10(1) (1998)

[7] Dillinger, P., Manolios, P.: Bloom filters in probabilistic verification. In: Formal Methods in Computer-Aided Design. LNCS, vol. 3312. Springer (2004)

[8] Flavio Lerda, R.S.: Distributed-memory model checking with spin. In: Theoretical and Practical Aspects of SPIN Model Checking. Springer (1999)

[9] Garavel, H., Mateescu, R., Smarandache, I.: Parallel State Space Construction for Model-Checking. In: SPIN workshop on Model checking of software. LNCS, vol. 2057 (2001)

[10] Inggs, C.P., Barringer, H.: Effective state exploration for model checking on a shared memory architecture. In: Parallel and Distributed Model Checking. ENTCS, vol. 68(4) (2002)

[11] Jenkins, B.: Hash Functions. "Algorithm Alley". Dr Dobb's Journal (1997)

[12] Miner, A., Ciardo, G.: Efficient reachability set generation and storage using decision diagrams. In: Application and Theory of Petri Nets. LNCS, vol. 1639. Springer (1999)

[13] Petcu, D.: Parallel explicit state reachability analysis and state space construction. In: Symposium on Parallel and Distributed Computing. IEEE (2003)

[14] Stern, U., Dill, D.: Parallelizing the Murϕ verifier. In: Computer Aided Verification. LNCS, vol. 1254. Springer (1997)