A General Lock-Free Algorithm for Parallel State Space Construction

Rodrigo T. Saad and Silvano Dal Zilio and Bernard Berthomieu
CNRS; LAAS; 7 ave. Colonel Roche, F-31077 Toulouse, France
Université de Toulouse; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
{rsaad, dalzilio, bernard}@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 improvements, 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 shared memory systems. Our approach relies on the use of two data structures: a shared Bloom filter to coordinate the state space exploration distributed among several processors and local dictionaries to store the states. The goal is to limit synchronization overheads and to increase the locality of memory access without having to make constant use of locks to ensure data integrity.

Bloom filters have already been applied for the probabilistic verification of systems; they are compact data structures used to encode sets, but in a way that false positives are possible, while false negatives are not. We circumvent this limitation and propose an original multiphase algorithm to perform exhaustive, deterministic, state space generations. We assess the performance of our algorithm on different benchmarks and compare our results with the solution proposed by Inggs and Barringer.

I. 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. Another motive, important as well, is the possibility to access a large amount of fast-access memory.

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 shared memory systems, that are multiprocessor architectures where the memory space can be accessed by all processors. The basic idea behind such algorithms 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. With a shared memory architecture, the state space is distributed among all processors and additional efforts are required to ensure data integrity. This is generally obtained through the use of low-level concurrency control mechanism such as locks and barriers.

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

Our approach is based on two data structures: a lock-free, shared Bloom filter to coordinate the data distribution; and local dictionaries – we use AVL trees in our implementation – to explicitly store the data. 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. The use of a Bloom filter avoids requiring a critical section when writing on local state spaces without sacrificing data integrity. The benefits of this design is better scalability on the number of processors and less contention on memory access. Bloom filters have already been applied for the probabilistic verification of systems; they are compact data structures 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; the other 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 II we review related work and give a brief introduction on Bloom Filters. Section III gives the details of our algorithms. In Section IV, we examine experiments performed on a set of typical benchmarks. We also compare our results with the solution proposed by Inggs and Barringer [11], which uses a lockless hash-table to store the states. 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. 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 5.10^8 states and to obtain absolute speed-ups of factor 10 – using the 16 cores available on this computer. We conclude with ideas for future extensions of our work.

II. 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 not following this approach is [6], using SIMD model. 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.

A. 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, 7, 9, 10, 14, 15] 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 concurrent balanced-tree. Data consistency is solved by using locks together with a "splitting-in-advance" scheme to reduce the contention. In [11], 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. The data structure – the dictionary – used to store already visited states is a global hash-table. Unlike [1], access to the dictionary is not protected by locks, hence it is not possible to ensure data integrity. (See [2] for a discussion on using shared hash-tables for model-checking.)

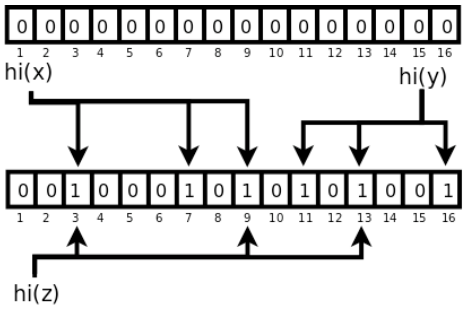


Figure 1. Illustration of some operations on a Bloom filter.

B. 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 [8]: 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 [4], while [8] 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 whether 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 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.

C. This Work

Our contribution is an algorithm that relies on a novel way to distribute the state space. Our algorithm dynamically assign states to processors, in the same way we dynamically assign work to processors using a work-stealing strategy. This differs from hash partitions based solutions [1, 7, 9, 10, 14, 15], where states are assigned statically.

States are stored in local dictionaries, while a shared Bloom filter is used to avoid the need for locks. In a local dictionary, only the processor that owns the dictionary can write on it (but concurrent read access are allowed). Hence, compared to the work in [11], based on a lockless hash-table, we can guarantee data integrity and, in particular, that no state will be lost due to concurrent writes. Another benefit of local dictionaries is to take the most advantage of memory affinity and to improve spatial balance. Our algorithm is quite general and can be easily adapted to accept different kind of data structures for local dictionaries – we have, for example, already experimented both AVL trees and hash-tables.

III. 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.

Coordination between the processors is based on a lockless shared Bloom filter used to test whether 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 III-A. In Section III-B, 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 $0..N-1$.

A. Shared and Local Data

Our objective is to design a solution adapted to typical shared memory architectures. This means that, in addition to the common difficulties related to shared memory architecture (ensuring data consistency; reducing contention on shared data access; ...), we should also consider the case of Non-Uniform Memory Access architectures (NUMA), where the latency and bandwidth characteristics of memory accesses depend on the processor or memory region being accessed. 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 positive – state s may have been visited before – 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: a *local stack* for storing its private work and a *shared stack* for sharing work with idle processors. Access to the shared stack are protected by locks to prevent different processors from requesting the same work. Finally, a shared vector is used to store the current state of processors (either idle or busy) in order to detect termination. Figure 2 illustrates the shared and local data structures used in the algorithm.

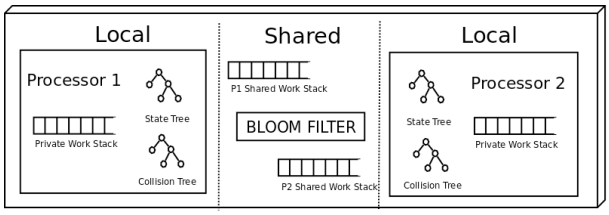


Figure 2. Shared and Private Data Model Scheme.

B. 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 [11]. 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 [11] by its use of unbounded shared stacks and the *sh_work_load* parameter.

C. 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 fewer than 3 iterations.

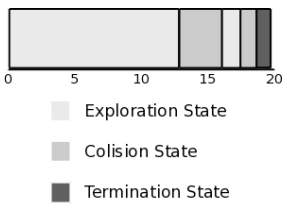


Figure 3. Timeline of states alternation.

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 are 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 positives.

1) Exploration: We give the pseudo-code related to the exploration phase below. The exploration phase proceeds until no new states can be added to the Bloom Filter *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*. Although concurrent accesses to *BF* may seldom result in extra work (state duplication), this is negligible compared to the gain in performance due to the use of a lock-free data structure. 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, unresolved collisions (false positive) are specially tagged as a means to bypass the *BF* membership test at this phase. We give more details on unresolved collisions in the description of the next phase.

```
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_stack
        endif
      endlet
    else if s is not in state_tree
        then search_and_insert s
               into collision_tree endif
    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
```

2) Collision Resolution: The search for collisions (the same state generated in two distinct processors) is done concurrently by each processor through the comparison of its *collision_tree* with the *state_tree* of every other processors. This operation can be implemented efficiently. Indeed, since all these data structures are sorted (we use AVL trees for storing states), collisions can be efficiently resolved by comparing trees as 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 the collision resolution, a state found in the state tree of another processor, say P_i, can be safely deleted from *collision_tree*: it is a "real" collision and it is currently processed by P_i. If the state does not appear in the state tree of another processor then the state 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 not(found) then
    insert collision into private_stack
        and mark as a 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
```

3) 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.

IV. Experiments

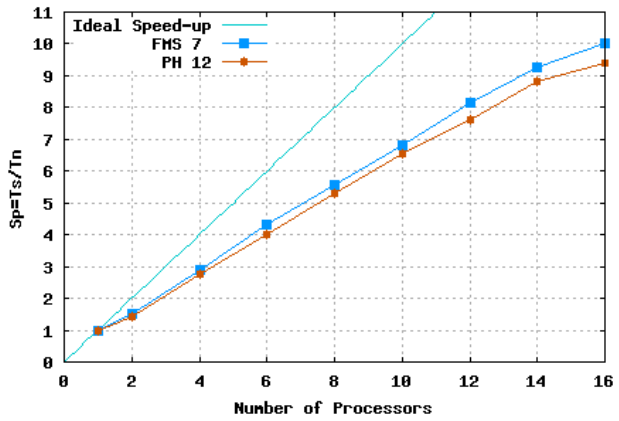
We implemented our algorithm using the C language with Pthreads [5] for concurrency and the Hoard Library [3] for parallel memory allocation. We developed our own library for Bloom filters with support for concurrent insertion. The library makes use of Bob Jenkins's hash function [12]. Experimental results 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. We worked with a 512MB Bloom filter ($n = 4.10^9$ bits) and 6 chained hash-functions ($k = 6$). These parameters are dimensioned for examples of up to 5.10^8 states, with a small rate ($\approx 2\%$) of false positives. In practice, this means that users do not need to adjust any parameter of the tool before using it.

The finite state systems chosen for our benchmarks are classical examples of Petri Nets taken from [13]. 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.

A. Speedup

Figure 4 gives the observed speedup of our algorithm when generating the state space for 12 philosophers (PH 12) and FMS 7 with a different number of processors. We give the relative speedup, measured as the ratio between the execution time using N processors (T_N) and the time of the same algorithm on one processor.

Figure 5 depicts the system occupancy rate, throughout the duration of the state space computation, for the PH 12 model using all the available processors. The occupancy rate measures the utilization of the machine CPUs. The figure shows high occupancy rate[1] ($\approx 92\%$) for our algorithm, except for a small interval that corresponds to the transition between the exploration and the collision resolution phases.



# proc.	speedup			
	FMS 7 (6.10^7 states)		PH 12 (3.10^7 states)	
	Time (s)	Speedup	Time (s)	Speedup
1	3132	–	2046	–
2	2048	1.52	1414	1.44
4	1075	2.91	744	2.75
6	723	4.33	510	4.01
8	564	5.55	386	5.30
10	460	6.80	312	6.55
12	384	8.15	269	7.60
14	338	9.26	232	8.81
16	313	10	218	9.38

Figure 4. Speedup analysis for PH 12 and FMS 7 models.

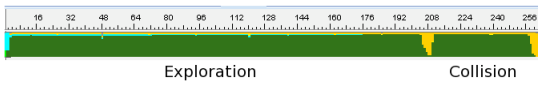


Figure 5. Occupancy rate for PH 12 with 16 processors.

B. Time-Memory Tradeoff

Figure 6 gives results on the number of *collision nodes* (see Section III-A) used on the FMS 7 and PH 12 examples. We also give the amount of memory required for the collisions tree. The results show an increase of the memory footprint when the number of processors increase. 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 shown with the experiments, the number of collisions generated by our algorithm may be 8 times greater than the number of states in the worst case (16 processors). What is observed is a general tradeoff between memory space and computation time that is often found in parallel algorithms. It should be noted that the use of traditional optimizations, such as partial-order or symmetry reduction techniques, will reduce the number of collisions.

# proc.	FMS 7 (6.10^7 states)		PH 12 (3.10^7 states)	
	# collision tree nodes	Ex. M[a] (GB)	# collision tree nodes	Ex. M[b] (GB)
2	19.10^7	3.8	13.10^7	6.5
4	29.10^7	5.8	20.10^7	10
6	32.10^7	6.4	23.10^7	11.5
8	34.10^7	6.8	24.10^7	12
10	35.10^7	7.0	25.10^7	12.5
12	36.10^7	7.2	25.10^7	12.5
14	37.10^7	7.4	26.10^7	13
16	37.10^7	7.4	26.10^7	13

[a]Extra Memory Estimation = $collisions * 20$
[b]Extra Memory Estimation = $collisions * 50$

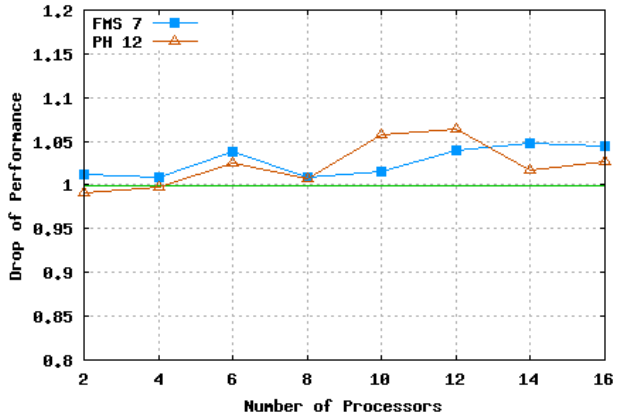
Figure 6. Collision analysis for FMS 7.

A parallel algorithm often trades additional memory space for better execution time. Nevertheless, it is very important to maintain this additional memory usage at an acceptable level. In our case, this means limiting the number of collisions. A straightforward way to deal with this problem is to force the early start of the *collision resolution* phase as soon as one of the processors reaches a given threshold of collisions states. We can compare this strategy with familiar memory management techniques, such as garbage collection. The choice of the good value for the threshold is a tradeoff between execution time and memory usage.

Figure 7 gives results, for the PH 12 and FMS 7 examples using a threshold value of 10^7 states. We observe for the FMS 7, in the worst case, a drop of performance below 10% with an average extra memory capped at 3.2 GB. With this strategy, the maximal extra memory required by our algorithm is given by the formula $N \times Th \times SS$, where N is the number of processors used; Th is the threshold (10^7 states in this case); and SS is the size of the state representation (20 bytes for FMS 7). The results given in Figure 7 shows a small gain of performance with 2 and 4 processors. This behavior can be explained by the use of a NUMA architecture. Indeed, with few processors, using less memory results in better processor affinity.

We study the impact of the threshold value on the overall performance in Figure 8. The figure depicts the relative variation of performance for different values of the threshold for the FMS 7 and PH 12 models. For both models, the experiments show that threshold values above 4.10^6 lead to almost no penalty: we observe a drop of performance below 10% using 16 processors. The last column in the table of

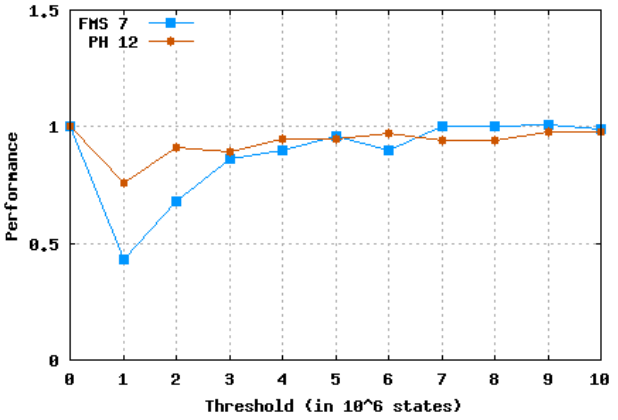
[1]The slightly time difference is a consequence of the overhead generated by the profiling tool.



# proc.	speedup					
	FMS 7 (6.10^7 states)			PH 12 (3.10^7 states)		
	Time(s)	Speedup	Ex.M(GB)[a]	Time(s)	Speedup	Ex.M(GB)[b]
2	2072	1.51	0.4	1401	1.46	1.0
4	1085	2.88	0.8	742	2.75	2.0
6	750	4.17	1.2	523	3.91	3.0
8	569	5.50	1.6	389	5.25	4.0
10	467	6.70	2.0	330	6.20	5.0
12	399	7.84	2.4	286	7.15	6.0
14	354	8.84	2.8	236	8.66	7.0
16	327	9.57	3.2	224	9.13	8.0

[a]Extra Memory Estimation = $N * 10^7 * 20$
[b]Extra Memory Estimation = $N * 10^7 * 50$

Figure 7. Speedup analysis for PH 12 and FMS 7 with memory recycling.



# Threshold	speedup					
	FMS 7 (6.10^7 states)			PH 12 (3.10^7 states)		
	Time(s)	Gain	Exp. – Col.	Time(s)	Gain	Exp. – Col.
∞	334	1	.76 – .24	216	1	.87 – .13
1.10^6	762	.43	.37 – .63	284	.76	.58 – .42
2.10^6	488	.68	.56 – .44	237	.91	.71 – .29
3.10^6	384	.86	.64 – .36	242	.89	.72 – .28
4.10^6	369	.90	.64 – .36	226	.95	.77 – .23
5.10^6	346	.96	.68 – .32	226	.95	.79 – .21
6.10^6	370	.90	.68 – .32	222	.97	.80 – .20
7.10^6	331	1.00	.77 – .23	228	.94	.81 – .19
8.10^6	332	1.00	.75 – .25	229	.94	.82 – .18
9.10^6	328	1.01	.73 – .27	219	.98	.87 – .13
10.10^6	337	.99	.76 – .24	219	.98	.85 – .15

Figure 8. Threshold analysis for PH 12 and FMS 7.

Figure 8, labelled "Exp. – Col.", splits the total execution time into the time spent in the exploration and collision resolution phases. The results show an inverse correlation between the ratio of times spent in these two phases and the overall performance: we observe that the speedup decreases when this ratio increases. The intuition behind these numbers is quite simple; with smaller thresholds, there are not enough "newly discovered states" during the exploration phase to compensate for the time spent during the collision phase. As a matter of fact, we observe good time ratio between exploration and collision phases in the experiments without memory recycling (threshold value of ∞). For instance, for both models, a threshold of 10^7 gives almost the same profile than using no memory recycling.

C. Comparison

We conclude this section on experimental results with a comparison with previously existing algorithms. It has proved difficult to port available implementations on the configuration used for our experiment. As a result, we developed our own implementation of some algorithms described in the literature. In this section, we turn our attention to an algorithm proposed by Inggs and Barringer [11]. We consider a simple variant based on a lockless shared table to store the states (in this variant, hash collisions are resolved using a chained list). The choice of a lockless shared table result in better performances, since it alleviates most of the synchronizations costs. Nonetheless this choice does not ensure data integrity, unlike with our proposed solution.

Figure 9 gives a comparison between two different implementations of our algorithm, AVL and Table, and our implementation of Inggs and Barringer algorithm [11], Lockless. AVL stands for the straightforward implementation described in Section III, using AVL Trees as local dictionaries. Table is the same algorithm where AVL trees have been replaced by local hash-tables. The AVL implementation proved slower than the two other solutions. This result is not enough to dismiss the use of AVL. Indeed, while the high algorithmic cost associated to this data structure is an handicap, the choice of AVL has also some benefits. For instance, the use of a sorted data structure in AVL simplifies the *collision resolution* phase, where the state in each local collision tree should be compared against all the other collision trees; this may make the *AVL* solution faster when there are many collisions (hence it could be superior when the number of processors increase). Finally, we intend to use our proposed algorithm for the analysis of timed systems, in which the cost of computing a new state is much higher than in the examples presented here ; this will increase considerably the exploration times, by the same amount in all versions, making their performances very similar on average.

The results show that our implementations (AVL and Table) are significantly slower than Lockless. We do not consider this result as discouraging as the difference in performance can be

traced to our main objective, that is to ensure that no state is lost during the exploration. Our solutions could be improved in a number of ways, for example using low-level optimizations, such as local caches. We used the exact same algorithm than the one defined in Section III for our experiments, without any low-level optimizations, in order to precisely study the impact of each of our choices. In parallel with this work, we also started to investigate other variants of our algorithm and to test with optimized implementations. Most particularly, we are currently testing an asynchronous version of our algorithm where each processor can asynchronously alternate between the exploration and collision resolution phases, without blocking.

	Execution Time (s) with 16 processors		
Model	AVL	Table	Lockless
Kanban 9 38.10^7 states	2547	1319	535
FMS 8 24.10^7 states	2003	953	330
PH 13 14.10^7 states	1306	836	227

Legend for the algorithm name abbreviations
AVL Local AVL Trees as dictionaries
Table Local Hash Tables as dictionaries
Lockless Lockless Table for shared storage

Figure 9. Comparison of Different Implementations.

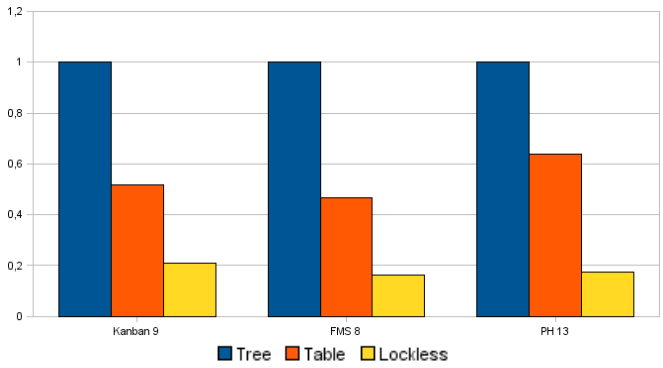


Figure 10. Normalized Execution Time.

V. Conclusion

We propose a new algorithm for parallel state space construction on shared memory systems. Our approach takes into account *spatial balance* by dynamically assigning states to processors and managing, as much as possible, states locally. For this reason, our algorithm is adapted to a non-uniform memory architecture. This approach complements the work-stealing strategy – that is also used in our algorithm – that fosters *temporal balance* by dynamically assigning work to processors. Another innovation resides in the use of Bloom filters – a data structure associated with probabilistic verification – to perform a complete exploration of the state space. 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 works with an explicit representation of states, our algorithm can be applied alongside traditional optimizations for reducing the state space size, such as partial-order and state compression 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 show promising speedups on a set of typical benchmarks. While the performance of the algorithm depends on the "geometry" of its input – for instance its concurrency degree – we have consistently obtained good results. For example, we routinely observe efficiency values[2] over 70% while keeping the extra memory needed with our algorithm at a constant level.

We consider several directions for future works. First, we are studying an asynchronous version of our algorithm in which each processor would asynchronously alternate between exploration and collision resolution phases without blocking each other. A further step, that follows the same direction, is to solve possible collisions on-the-fly. We are already experimenting with these ideas and we expect to reach performances (both in terms of time and memory) close to the one observed with our implementation of the "unsafe" Lockless Table algorithm. Another possible direction of work is to derive, from our current exhaustive algorithm, a *probabilistic state space construction algorithm*. In this context, the adjective probabilistic stands for an algorithm that builds an underapproximation of the global state space, with a very high probability of building the exact state space – by very high, we mean a probability of failure less than 10^{-30}. The idea, basically, is to leave out the collision-resolution phase and to use an enhanced Bloom Filter data-structure where only potential false positives are stored. Finally, we plan to experiment more thoroughly on the effect of combining state compression techniques with our approach.

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] Barnat, J., Rockai, P.: Shared hash tables in parallel model checking. Electronic Notes in Theoretical Computer Science 198(1) (2008), proceedings of the 6th Inter-

[2]Efficiency is computed as the ratio between speedup, T_N, and the number N of processors.

national Workshop on Parallel and Distributed Methods in Verification (PDMC 2007)
[3] Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A scalable memory allocator for multithreaded applications. ACM SIGPLAN Notices 35(11) (2000)
[4] Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4) (2004)
[5] Butenhof, D.: Programming with POSIX threads. Addison-Wesley (1997)
[6] 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)
[7] Ciardo, G., Gluckman, J., Nicol, D.: Distributed state space generation of discrete-state stochastic models. INFORMS Journal on Computing 10(1) (1998)
[8] Dillinger, P., Manolios, P.: Bloom filters in probabilistic verification. In: Formal Methods in Computer-Aided Design. LNCS, vol. 3312. Springer (2004)
[9] Flavio Lerda, R.S.: Distributed-memory model checking with spin. In: Theoretical and Practical Aspects of SPIN Model Checking. Springer (1999)
[10] 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)
[11] Inggs, C.P., Barringer, H.: Effective state exploration for model checking on a shared memory architecture. In: Parallel and Distributed Model Checking. Electronic Notes in Theoretical Computer Science, vol. 68(4) (2002)
[12] Jenkins, B.: Hash Functions. "Algorithm Alley". Dr Dobb's Journal (1997)
[13] 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)
[14] Petcu, D.: Parallel explicit state reachability analysis and state space construction. In: Symposium on Parallel and Distributed Computing. IEEE (2003)
[15] Stern, U., Dill, D.: Parallelizing the Murϕ verifier. In: Computer Aided Verification. LNCS, vol. 1254. Springer (1997)