DOI 10.12694/scpe.v17i2.1161
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 129–141. http://www.scpe.org

A PARALLEL ALGORITHM FOR THE STATE SPACE EXPLORATION

LAMIA ALLAL*, GHALEM BELALEM†, PHILIPPE DHAUSSY‡, AND CIPRIAN TEODOROV§

Abstract. Model checking has long been used as a means of verification of formal specifications. This is a verification technique of dynamic systems that explores all possible states of the system. It determines whether the given system satisfies its specification. This technique suffers from the state explosion problem when traversing all possible states of systems. Parallel and/or distributed approaches are used to cope with the state space explosion problem. In this article, we propose a synchronized parallel algorithm of exploration based on a fixed number of threads. We present many experiments for a comparison between our parallel approach and the algorithm proposed for a parallel exploration in $SPIN$. We show by an experimental study that our parallel approach gives encouraging results.

Key words: Model checking, state explosion problem, parallel exploration, memory space, reachability graph, parallel algorithms, execution time, software verification.

AMS subject classifications. 68Q60, 68W10

1. Introduction.

Verification by model checking is an automatic verification technique, to verify that a system satisfies a given specification. This is a commonly used tool in situations, where, it is essential to certify the proper functioning of a system. Thus, the model checking tools are widely used in high-tech industries, for verification of electronic circuits, or even in aeronautics, to ensure safety of embedded systems. Any algorithm of model checking is based on two steps: (1) exploration of reachable states of the system; and (2) verification of the specifications in this state space. These two steps can sometimes be performed simultaneously, which is called on the fly model checking. Exploration is a computing process which determines a sequence of actions, making it possible to achieve a desired goal. A good exploration means, the achieving and the storage of a large number of states without exceeding the available memory resources [1] and in finite time. The state space can be described by an initial state and a set of transitions. A succession of states produced by actions forms a path within the state space [2, 3].

In the case of realistic examples, the number of states can be enormous. For example, in an n-bit counter, the number of states is exponential in the number of bits (2^n). Reachability analysis is limited by the state explosion problem [4, 5, 6]. This problem occurs, when the state space to be explored is large and cannot be explored by the algorithms for lack of capacity memory resources, or an important time because the memory space needed to carry out exploration is higher than the memory space contained in the machine. Many researches have been done to fight the state explosion problem, by taking advantage of a distributed environment, by increasing the computational power and using large available memory [27].

In this article, we are interested in the analysis of execution time needed to carry out exploration. During the reachability analysis, it is essential to take into account the concept of time because, even by distributing the states graph on multiple machines (at Amazon or another provider) will not settle the problem, due to the exponential growth of the number of system-states, so we need in this case to treat the temporal explosion. We present a comparative study between two algorithms for a parallel exploration. For that, we used four models: Peterson [7], Dining philosophers [8], Producer-consumer [9] and counters.

Outline of the paper: the article is divided into 7 sections, the second section presents some works which offer solutions to the state explosion problem. Section 3 presents model checking and reachability analysis. Section 4 is devoted to the definition of the synchronized parallel algorithm (SPA). The fifth section presents the parallel algorithm for state exploration in $SPIN$. The sixth section is devoted to the experiments performed for a comparison of the SPA algorithm and the algorithm proposed for $SPIN$ model checker. Section 7 presents a comparison between two parallel approaches for reachability analysis.

*Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 Ahmed Ben Bella, Oran, Algeria (allal.lamia@gmail.com).

†Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 Ahmed Ben Bella, Oran, Algeria (ghalem1dz@gmail.com).

‡Lab-STICC UMR CNRS 6285 ENSTA Bretagne, Brest, France (philippe.dhaussy@ensta-bretagne.fr).

§Lab-STICC UMR CNRS 6285 ENSTA Bretagne, Brest, France (ciprian.teodorov@ensta-bretagne.fr).

2. Model checking and reachability. Model Checking is a verification technique, based on the exhaustive state space exploration of systems, in searching of behaviors that do not verify its specification. A model checker can be seen as a black box, which accepts as input a system as well as a property expressed on this system and returns an answer, indicating if the property is checked or not. When verifying properties, through explicit-state model checking, all the possible behaviors of the system are enumerated and the properties are checked. The algorithms implemented include two phases, a construction of the state space then an exploration of this state space in searching of errors. The state space is represented as a graph which, describes all the possible evolutions of the system. Nodes of the graph represent the states of the system and the arcs represent the transitions between these states [2, 3]. The reachability analysis consists on the exploration of models state by state, each state and all its successors are stored in memory. Exploration finishes when all the states are visited. An exploration algorithm, with each step of its execution, can either visit a new node, or an explored node. *Fig. 2.1* gives the organizing chart of the basic sequential reachability for performing a breadth first search.

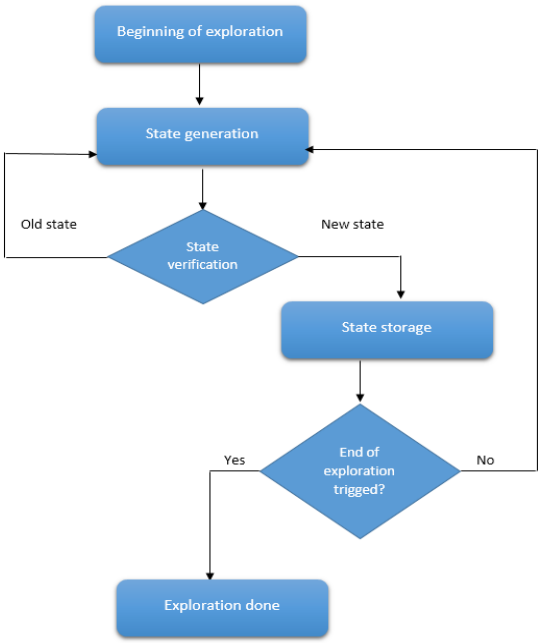

FIG. 2.1. *Sequential Reachability using a breadth first search*

3. Related work. Many solutions have been proposed for the state explosion problem. In this section, we present five solutions. Each one of them is running on a different architecture (distributed, parallel, sequential). These solutions are based on methods (states compression, partial order reduction, bit state hashing) and different data structures. Each solution aims to improve the performances in execution time and memory capacity.

In [10], the authors present a new solution to the state explosion problem. The approach is based on the concept of scalability. When checking systems by model checking, this problem can occur if the models are large. The authors propose to optimize the model checker Divine [10], so that the network can be scaled up in number of nodes, which can check larger systems. This is the main advantage of this method. In previous experiments, Divine was evaluated on a small number of nodes, it was soon determined that performance, of several Divine algorithms did not scale well, with a high number of nodes, because of the number of messages that pass through the network. The optimizations focus on improving two distributed algorithms (OWCTY and MAP) included in Divine, which allow for a good exploration of the model. Each node will process a part

of state space. A major drawback of this approach, is the number of messages exchanged through the network because large messages are sent in priority which could cause an overload on the network.

The solution described in [11] allows a distribution of state space exploration, during the verification of models by model checking using $SPIN$ [12, 13]. Each node has a set V, that contains the explored states, and a queue U, to store the unvisited states. An advantage of this approach, is that the proposed algorithm is compatible with 3 methods for state space exploration (states of compression, partial order reduction, state bit hashing). A drawback of this method is the absence of a considerable gain for each model explored.

The approach proposed in [14] is based on a sequential algorithm. Its objective is the storage of states in their compressed form (this constitutes the advantage of this method), only the difference between the previous state and the following state is stored. The first generated state (initial state) is stored in an explicit way, the other states are stored in a compressed form in hash tables. The states are decompressed, to verify if a state was already visited or not, for that, it is necessary to add the most recent changes for each state, until a state stored in explicit form is reached. The disadvantage of this method, is the backtracking function, which represents an overload because the execution time can increase quickly.

The solution presented in [15] is based on an algorithm executed in parallel by multiple processors. The authors presented a model checker called PMC (Parallel Model Checker), to verify properties written in CTL* [16] that combines both of language Computation Tree Logic (CTL) [18] and Linear Temporal Logic (LTL) [17]. This model checker verifies, models whose behavior is represented as and/or trees. The proposed solution is inspired by the parallel algorithm, for pure reachability analysis [19]. Inggs & Barringer [15], presented a parallelization of processing model checking using a shared memory architecture. PMC uses a dynamic load balancing technique. Each process has its own unbounded private stack and bounded shared stack for storing work items. Shared stacks are protected by locks to synchronize read and write access to it. During model checking, processes have to interact with each other, via the shared memory, to divide new work items and to avoid duplication of work. The advantage of this method is the presence of private and shared stacks for each processor.

Inggs & Barringer [15] presented the results for three types of experiments. The drawback of this approach, is the use of a shared memory, which requires synchronization between processes.

In [20], the authors propose a parallel algorithm for the construction of state space. The architecture used is a shared memory multiprocessor architecture. States are stored in the local hash tables (lockless hash table). This constitutes the advantage of this method. Each processor has a private stack and a shared stack. Distribution and coordination of states between the processors is made through a location table, which contains the list of states that have been visited. It is used to dynamically allocate works on the processors. Its primary role is to return true, if the state was visited and the number of process running, false otherwise. Collisions may occur, if the key returned by the hash function, is the same for both states. This constitutes the drawback of this approach.

4. Synchronized parallel algorithm (SPA). A parallel machine is essentially a set of processors that cooperate and communicate. A parallel algorithm runs on a parallel computer. The instructions are executed simultaneously, which can lead to a considerable gain in execution time. An important task in a parallel approach, is the assignment of work to threads, to have a load balancing between threads.

The Synchronized parallel algorithm (SPA), is based on the use of a fixed number of threads and uses 2 sets of states: K and $Q[i]$. Set K, is the set of visited states, this set is shared by all the threads, then, the access is synchronized. Successors' states are stored in Queue $Q[i]$, where i varies from 1 to N (N is the number of threads). $Q[i]$ is a FIFO queue where each thread i processes states in $Q[i]$. The size of this queue is unlimited. Each pointer $Q[i]$, points to a linked list of nodes (value, link to the next node). Each time a state is generated, a thread will be randomly selected and the state will be stored in its list (we can't have at the same time, removal and adding operations). Random function was used, to load balancing states on subsets $Q[i]$. Synchronization between different threads, is performed on the lists of each thread during states adding, or states removal. We have a fixed number of threads, so we need to be able to determine when all states have been reached, to stop the exploration.

Fig. 4.1 shows the threads synchronization on the Queue $Q[i]$ containing the states to be processed. At the first step, thread 1 generates the initial state. For each next state, a thread is generated randomly and the state

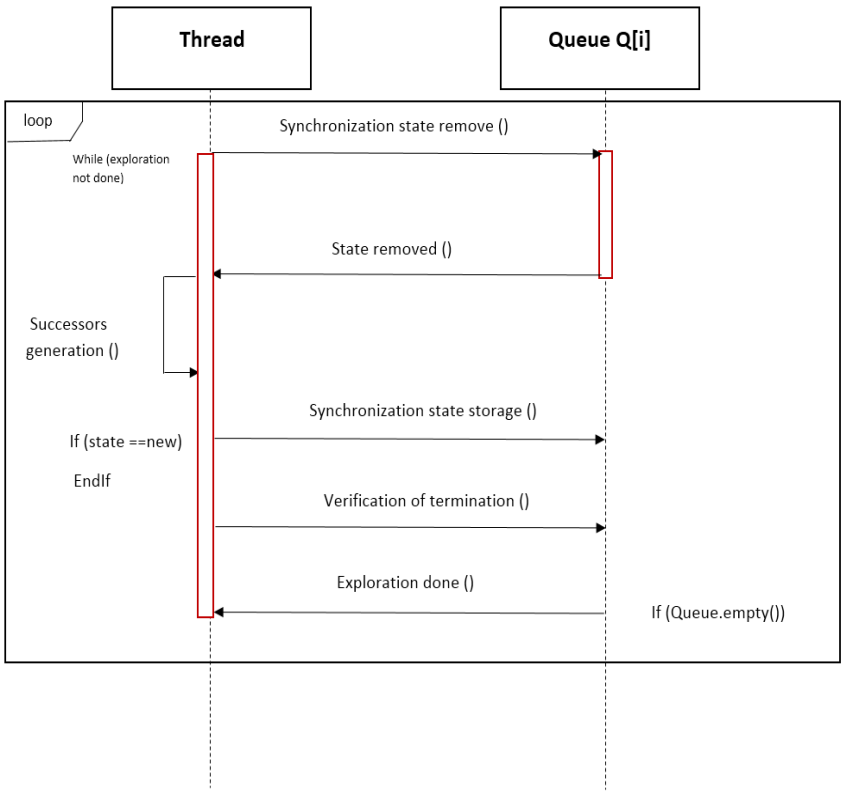

FIG. 4.1. *Threads Synchronization on the queue* $Q[i]$

is stored in its list. After this step, each thread i where $Q[i]$ is not empty, will remove a state from the queue by the synchronization state remove function, only one thread can execute this function at the same time (critical section). The successors of this state are generated by the successors generation function. After that, a test is realized on each next state, if the state is new, it is stored randomly in the queue $Q[k]$ of the thread k, by executing the state storage function, otherwise it goes to the next state. Termination verification is triggered, to check whether, termination of exploration has been reached, by the verification of termination function. If the queue $Q[i]$ is empty, exploration ends, otherwise the process will be repeated as long as the queue is not empty.

The algorithm of parallel exploration is presented in what follows.

At the beginning of the exploration, the initial state is generated, then its successors are observed. At this time, for each new configuration, a thread is chosen randomly (line 9 of *Algorithm 1*). Data used are shared between threads, synchronization are made on the set k containing all visited states and on the queue $Q[i]$ for processing current states.

To explain the exploration process, we have taken an example of counters (*Fig. 4.2*). The representation of a counter that is incremented and decremented, is carried out using a deterministic automaton consisting of a single state (both initial and terminal). The state s is incremented up to N, s can be decremented at each step down to 0, which represents the initial state. If we add another counter, there could be a state vector with 2 cells, therefore, with N counters, we will have a state vector composed of N cells, corresponding to n automaton (*Fig. 4.3*). For example, by fixing the specific value to 5 and having 6 counters, the total number of states to be explored is equal to 46656 $((the\_specific\_value + 1)^{the\_number\_of\_counters})$. The number 1 corresponds to the minimum value 0. *Fig. 4.3* presents a part of the reachability graph of 6 counters that can be incremented or decremented on each step. For each explored state, a set of generated successors is generated. The first step, is the generation of initial state (0 0 0 0 0 0) by thread 1. From the initial state, six states are discovered, because

Algorithm 1 Synchronized Parallel Algorithm (SPA)

```
1: exploration_done ← false
2: For each (w: 1..N)
3: do
4:     for (each s in 1 .. N) do
5:         (Synchronized) delete s from Q[w]
6:         for (each successor s of s) do
7:             if ((Synchronized) s not in K) then
8:                 (Synchronized) add s to K
9:                 w= choose Random 1 .. N
10:                (Synchronized) add s to Q[w]
11:            end if
12:        end for
13:    end for
14:    if (w==1) then
15:        if (all Q[1..N] == NULL) then
16:            done ← true
17:        end if
18:    end if
19: while !exploration_done
```

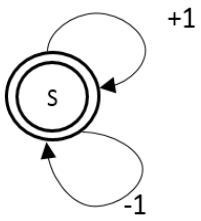

FIG. 4.2. *Automaton of a counter that increments and decrements*

at each transition, a counter can be incremented. A transition from one state to another occurs, when a counter is incremented or decremented. The initial state is stored in the set K, because the state is new and in the Queue $Q[1]$, to explore its next configurations (states). Thereafter, all its successors are visited and stored in their turn in both sets, to do that, we need to choose randomly, a thread x for each successor c. Each next configuration c is stored in $Q[x]$. The exploration stops when the queue $Q[i]$ is empty, i varies from 1 to N.

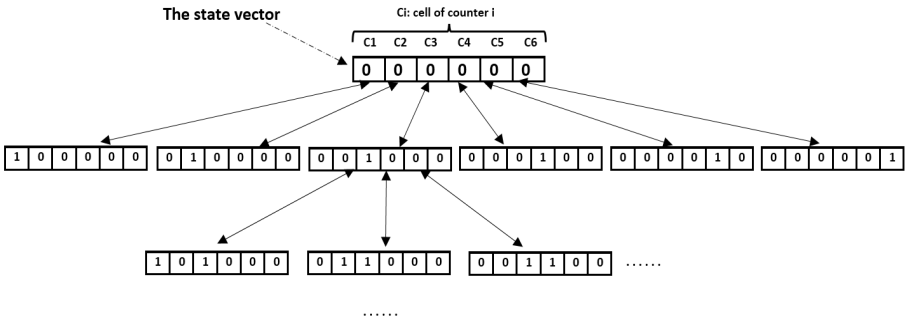

FIG. 4.3. *A part of Reachability graph of 6 counters incremented up to 5 and decremented down to 0*

5. Parallel exploration in SPIN. $SPIN$ is a tool for verification and simulation of concurrent systems. To be studied, a concurrent system is first described in Promela (Process Meta Language), the $SPIN$ verification language [26]. The algorithm (*Algorithm 2*) proposed in [22], is based on the use of a three dimensional queue $Q[t][i][j]$ for storage of states, whose successors have not been observed yet. It is composed of 3 parameters: t, i and j. The parameter t, is varied from 0 to 1, it allows states to pass from current states to future states. At each step of exploration, all states from $Q[t][i][j]$ are processed and their successors are stored in $Q[1-t][i][j]$, corresponding to the configurations that will be observed at the next step. A lockless hachtable [23], was used in order to avoid waits between different threads. An important task in the algorithm proposed in [22], is to determine when all states have been explored to stop exploration.

Algorithm 2 Parallel Exploration Algorithm in $SPIN$

```
1: done ← false
2: t ← 0
3: Search (i: 1..N)
4: do
5:     for (each state in 1..N) do
6:         Delete s from Q[t][i][j]
7:         for (Each next configuration c of s) do
8:             if ¬ (S.Contains(c)) then
9:                 S.add(c)
10:                k ← Choose Random from 1..N
11:                add state to Q[1 − t][k][i]
12:            end if
13:        end for
14:    end for
15:    Wait()
16:    if (i==1) then
17:        wait until all threads are idle
18:        if (all Q[1 − t][i][j] == NULL) then
19:            done ← true
20:        else
21:            Notify all threads
22:            t ← 1-t
23:        end if
24:    end if
25: while !done
```

In this algorithm, the parameters i and j vary from 1 to N, where N is the number of threads. The size of this queue is unlimited. The current states to be generated are treated from $Q[t][i][j]$ and successors are add to $Q[1-t][k][i]$ with k, a thread selected randomly from N. The conceptual difference between the SPA and $SPIN$ algorithm is that, in $SPIN$, a three dimensional queue is used, which consumes more memory space. SPA algorithm is based on the use of a one dimensional queue. A thread that ends exploring its existing states, waits for all threads to finish their treatment, to pass to future states. In the SPA algorithm, there is no waiting between threads, threads wait when adding or removing state in (from) the queue.

6. Experimental study. In this article, we presented a parallel approach for state space exploration. We carried out four experimental studies on 4 different models, 3 models from BEEM database [21] and a counter model. The objective is to make a comparison between two parallel approaches. This comparison is based on the execution time of the exploration step. The experiments are performed by varying some metrics. We study the behavior of these approaches by experiments. The experiments were performed on an i7 machine with 8 cores, it operates at a frequency of 2 MHz, with 16 GB of physical memory. We have implemented both parallel algorithms using java platform [25]. We realized the specification of these models in Java. We have fixed the

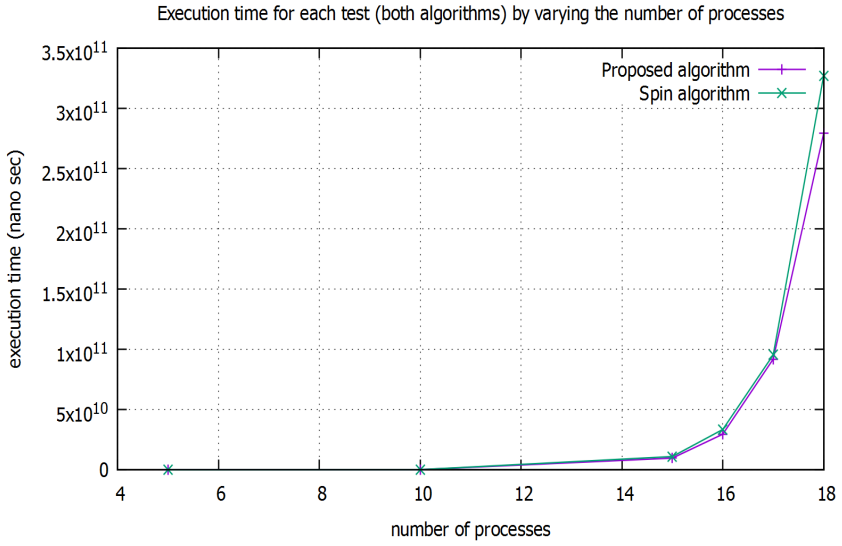

FIG. 6.1. *Execution time (both parallel approaches) by varying the number of processes*

TABLE 6.1
Execution time, configurations number, and processes for both algorithms

Processes	**Configurations**	**Exe. time SPA algo (seconds)**	**Exe. time $SPIN$ algo (seconds)**
5	352	0.017	0.022
10	47104	0.20	0.21
15	3473408	9	11
16	7929856	29	33
17	17956864	91	95
18	40370176	279	327

number of threads at each experiment. The launch of threads has been realized using the class thread. The number of states is given at the end of each experiment by the set K (set of visited states). The size of this set is equal to the number of states. No information is calculated or communicated to each state, because, the purpose of reachability analysis, is to load all the states in memory. The number of threads to perform exploration is the same for both algorithms on each experiment.

6.1. Experiment 1: Peterson model. Peterson's algorithm [7], is a mutual exclusion algorithm for concurrent programming. This algorithm is based on an active wait approach. It consists of two parts: the input into the critical section and the output from it. We have made a parallel comparison between, our approach, and the parallel algorithm developed in $SPIN$ model checker [22], using Peterson model [7].

We made 6 different tests, by varying the number of concurrent processes (5, 10, 15, 16, 17, 18), and estimated execution time of reachability analysis for each test (see *Fig. 6.1* and *Table 6.1*). The number of states varies from 352 to 40370176, regarding to the number of processes accessing a critical section. At each test, we fixed the number of created threads (process machine) for a parallel execution (from 2 to 8). At the last test (with 18 processes accessing a critical section), we used 8 threads to fully exploit all machine resources. From the results provided by this experiment, SPA algorithm shows better performance in execution time (*Fig. 6.1*). The comparison was made on the execution time estimated to perform experimentations. In the algorithm presented in [22], whenever a thread finishes processing its tasks (lists of current states), it waits for other threads, therefore, it takes more time because it is based on the processing of states step by step. Having 18 processes accessing a critical section, the synchronized parallel algorithm shows better performance, because, for each process added (process in critical section), more configurations will be observed, because, the state

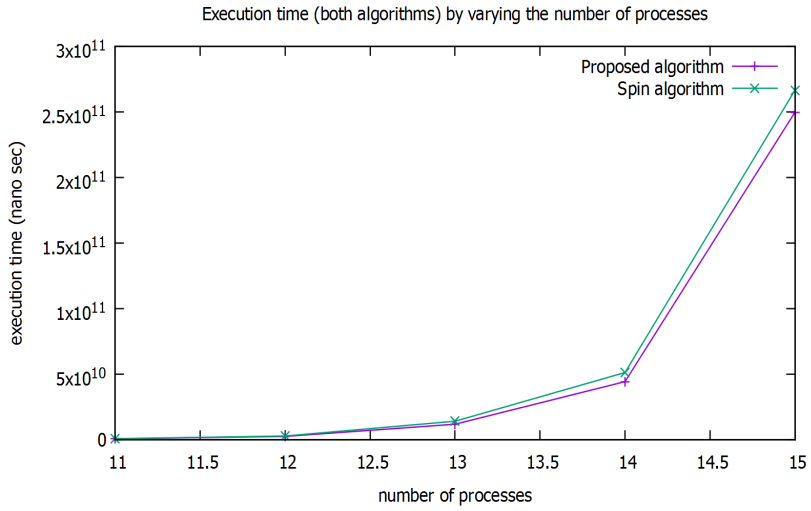

FIG. 6.2. *Execution time (both parallel approaches) by varying the number of processes*

TABLE 6.2
Execution time, configurations number, and processes for both algorithms

Processes	**Configurations**	**Exe. time** SPA **algo (seconds)**	**Exe. time** $SPIN$ **algo (seconds)**
11	393660	0.86	0.88
12	1240029	2	3
13	3897234	11	14
14	12223143	44	51
15	38263752	249	266

vector will change. The state vector, is an array, where each cell corresponds to the identifier of a process in critical section.

To interpret these results in execution time, we calculated the gain (in percentage) obtained by our proposed algorithm from each experience (21.45, 3.31, 12.10, 11.75, 3.64, 14.63). From these results, we estimate the average gain obtained from all tests, using Peterson model specification, $T_{average\_peterson} = 11.15\%$.

6.2. Experiment 2: Dining philosophers model. The dining philosophers model [8], is used to solve the synchronization problems between different processes. we realized the specification of this model in Java, and then, we performed an exploration on this model using both parallel algorithms. The result about execution time estimated, is given in *Fig. 6.2* and *Table 6.2*. We realized the experiments on the same machine, we have varied the number of processes in critical section from 11 to 15, with a step of 1 and we measured at every experiment, the execution time estimated during exploration. The number of states increases regarding to the number of concurrent processes, all possible case are exploited (all possible states are explored).

Having 15 concurrent processes, the number of configurations (states) is about 38.263.752. We can notice a performance gain, using the proposed approach compared to the parallel exploration in $SPIN$ model checker. The average gain obtained using our algorithm is about $9.90\% \simeq 10\%$. Concerning the model specification and reachability analysis, our algorithm shows better performance compared to $SPIN$ algorithm. The gain increases by increasing the number of processes in critical section, therefore, the proposed algorithm scales well.

6.3. Experiment 3: Producers consumer model. The Producers/Consumer model [9], is a classic example of two synchronization processes, one that produces information he deposited in a limited buffer size, and one that removes one by one to consume. We have done the specification of this model, using several producers and one consumer. We realized 5 tests, by varying the number of producers and the table size, in which data are stored. The comparison is made on execution time (*Fig. 6.3* and *Table 6.3*). In the first test,

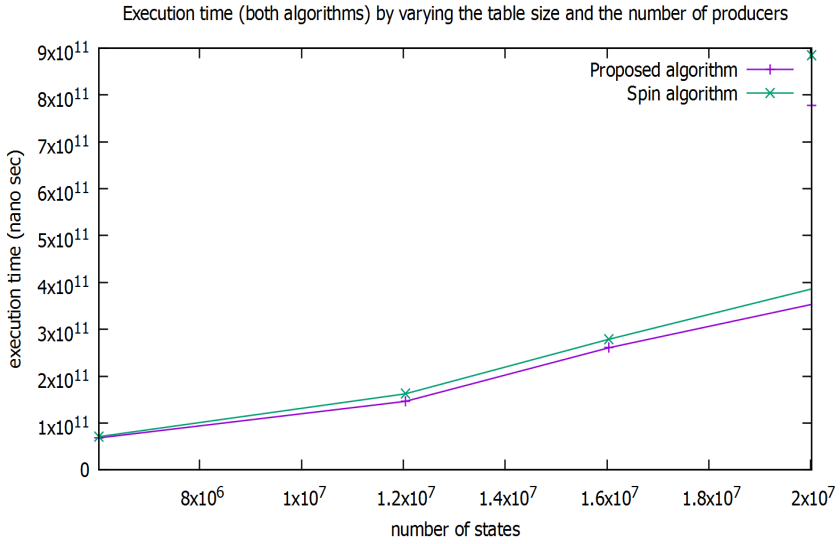

FIG. 6.3. *Execution time (both parallel approaches) by varying the table size and the number of producers*

TABLE 6.3
Execution time, and configurations number for both algorithms

Configurations	**Exe. time** SPA **algo (seconds)**	**Exe. time** $SPIN$ **algo (seconds)**
6020000	68	70
12040001	146	162
16040001	260	278
30060001	586 ($\simeq 10min$)	656 ($\simeq 11min$)
20020001	777 ($\simeq 13(min)$)	885 ($\simeq 15(min)$)

we used 600 producers and the table size was fixed at 10000 cells. The reachability graph corresponding to this experiment is composed of 6.020.000 states (configurations). The experiments and the number of states at each test, are given in *Table 6.4*.

TABLE 6.4
Experiments parameters and the total number of states obtained

Producers	**Table size (cells)**	**Number of states after reachability analysis**
600	10000	6.020.000
600	200000	12.040.001
800	20000	16.040.001
1000	30000	30.060.001
2000	20000	20.020.001

Comparing the results given by both approaches, we note that our algorithm shows better results in execution time, compared to the parallel exploration algorithm in $SPIN$ in each experiment. The average gain obtained by performing a comparison by SPA algorithm, is estimated by 8.62%.

6.4. Experiment 4: Counters model. We have made a parallel comparison between our approach, and the parallel algorithm developed in $SPIN$ model checker [22], using counters model. We used 5 counters, incremented from 0 to a maximum value, and made 6 tests. In the first one, the counters are incremented from 0 to 22. At each test, we incremented a maximum value by a step of 3 (from 0 to 22, from 0 to 25, ...). The result, is shown in *Fig. 6.4* and *Table 6.5*.

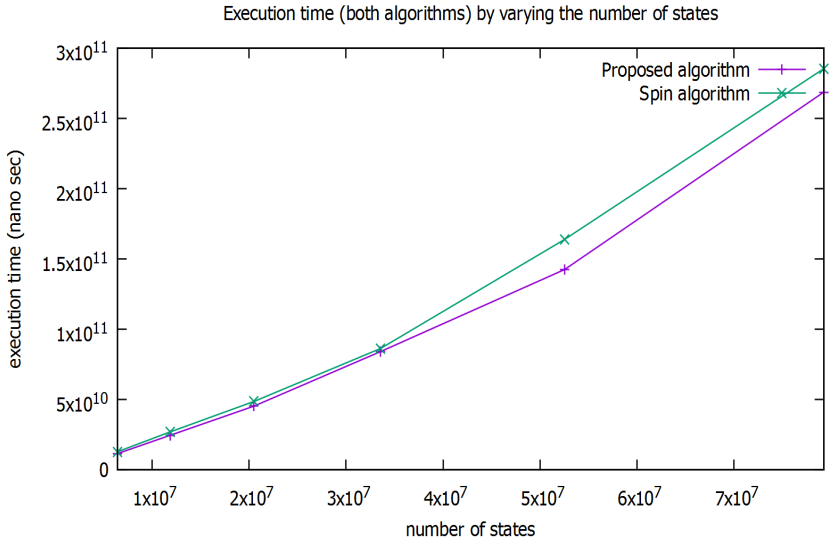

FIG. 6.4. *Execution time (both parallel approaches) by varying the number of configurations*

TABLE 6.5
Execution time, configurations number, and max_value for both algorithms

Max_value	**Configurations**	**Exe. time SPA algo (seconds)**	**Exe. time $SPIN$ algo (seconds)**
22	6436343	11	13
25	11881376	24	27
28	20511149	45	48
31	33554432	84	86
34	52521875	142	164
37	79235168	268	285

To calculate the number of states, we need the number of counters and the maximum value of counter. Then, this number is calculated by: $(max\_value + 1)^{number\_of\_counters}$. Regarding to the experiment shown in *Fig. 6.4*, our algorithm gives better results, by increasing the maximum value of counters. Both parallel approaches show significant results, when the number of configurations is high, with a gain provided by the synchronized parallel algorithm.

Despite the gain of 10%, we can notice at each experiment, that the gap between both curves (SPA and $SPIN$), becomes important when the number of processes increases, which predicts that these curves move further for larger number of processes, and therefore, the gain obtained by our approach will be more significant.

7. Positioning our parallel approach and Discussion. To position our parallel algorithm, with the integrated parallel algorithm in $SPIN$, which is very used in the exploration of states using model checking techniques, we studied the conceptual difference between both algorithms (SPA and $SPIN$), and the complexity of these two algorithms [24].

Conceptual difference between both algorithms: We realized in the previous section, a series of experiments. The SPA algorithm showed in each one, better results. This is due to the fact, that in the $SPIN$ approach, several loops are executed each time we alternate between current and future states: having N threads, for each thread i, all $Q[t][i][j]$ are explored (j varies from 1 to N). The treatment of visited states, for this thread, ends only after all queues are empty. The process is the same for other threads. In the proposed approach, each thread i manages its queue $Q[i]$, so there is no time lost. In $SPIN$, at each step (alternation between current and future states), a thread that ends its processing, waits for all other threads. A time is lost at this level, which represents the major drawback of this algorithm ($SPIN$ algorithm), especially if the wait is

TABLE 8.1
Data structures specified in the SPA algorithm

Data structures	**Memory space used**
exploration_done	32 bits
pointer queue $Q[i]$	32 bits
Linked list (queue $Q[i]$)	(32 + 32) * s bits
K (set of reached states)	(32 + 32) * t bits (key and data)

long. Waiting in the SPA algorithm are in the access to the queue $Q[i]$ for removal, or addition of states. This time is smaller, compared to the wait between threads in $SPIN$. This is demonstrated by the results obtained.

Study of the algorithmic complexity: an algorithm, is a sequence of actions performed from an initial state, to a final state in a finite time. We study the complexity to predict the execution time of an algorithm, and to compare two algorithms performing the same treatments. The complexity of an algorithm, is determined through a description of the behavior of algorithms. The complexity of an algorithm can be evaluated in time (speed), and in space. In this article, we focus on the study of the execution time. We conducted a study of the complexity, for both parallel algorithms: SPA algorithm, and the algorithm integrated in $SPIN$, for this, we have defined execution time for each type of instruction:

- ae: state assignment
- ce: comparison of states
- s: number of next states per state
- q: maximum number of states in $Q[i]$ or M
- p: number of threads
- w: waiting time per thread (idle)

Taking each operation, we estimate the time needed to achieve the reachability analysis, by each algorithm:

- **complexity of the synchronized parallel algorithm "SPA" (*Algorithm 1*, section 4)**: complexity of the proposed algorithm C_{App} is estimated by:

$$C_{App} = ae + p(ae + s.q(ce + 3.ae)) + ce + p.ce)) = O(qp) \tag{7.1}$$

- **complexity of the parallel algorithm proposed in [22] (*Algorithm 2*, section 5)**: complexity C_{SPIN} is estimated by:

$$C_{SPIN} = 2.ae + p(ae + s.p.q(ce + 2.ae)) + w(p-1) + q.ce.ae = O(qp^2) \tag{7.2}$$

According to these complexities obtained by equations (7.1) and (7.2), we can notice that our algorithm has order of $O(qp)$ time complexity, the complexity of the algorithm proposed in [22] is around the square, estimated to $O(qp^2)$. In conclusion, we can say and confirm that our proposed algorithm, for the exploration of states, can be used to explore a large number of states, in a linear time.

8. Memory space used in reachability analysis. Model checking, is a technique based on verification of the correctness of a system, with respect to a desired behavior and properties. Exploration consists on exploring each state (we have to iterate algorithm many times), therefore, having systems composed of large number of states, that tend to grow exponentially, in the number of its processes and variables, this case leads to a state space exploration for large systems. In this section, we analyzed the memory space used in both algorithms, by counting the total number of data structures used.

- **Memory space used in the synchronized parallel algorithm "SPA" (*Algorithm 1*, section 4)**: An integer value, is specified in the source code of any program as a sequence of digits. Usually, variables are stored on 32 bits, therefore, to analyze the used memory space, we have to count the number of data structures declared in the algorithm. The data structures specified in the algorithm, are listed in *Table 8.1.* exploration_done is a boolean variable, it is stored on 32 bits, we have N pointers of queue $Q[i]$, with N referring to the number of threads and each $Q[i]$ (i varies from 1 to N), points to a liked list of states. A number of next states is unknown. We have N linked lists, and each one contains nodes,

TABLE 8.2
Data structures specified in the parallel algorithm in SPIN

Data structures	**Memory space used**
done	32 bits
t	32 bits
pointer queue $Q[i]$	32 bits
list of points (thread's pointer to other threads)	32 * N bits
Linked list (for one thread)	(32 + 32) * s * N bits
S (set of reached states)	(32 + 32) * t bits (key and data)

each node contains two fields, an integer value and a link to the next node. Therefore, the memory used for states storage in the queue $Q[i]$ is (32 + 32) * s * N (s is the maximum number of states in a list of a thread). A set K contains explored states.
The size of K is determined by: (32 + 32) * t, with t the total number of states explored by reachability analysis. By having these information, we can estimate the state space storage, used by our approach, expressed in bits: $Memory\_used_{SPA\_app}$ = 32 + (32 * N) + ((32 + 32) * s * N) + ((32 + 32) * t) = $32(1 + N + (2 * s * N) + (t * 2)) = 32(1 + N(1 + (2 * s) + (t * 2))$.

- **Memory space used in the parallel algorithm in $SPIN$ (*Algorithm 2*, section 5)**: In the parallel algorithm developed in $SPIN$ model checker, 2 sets are specified, $Q[i]$ (3 dimensional queue) and S (set of reached states). The data structures specified in the algorithm are listed in *Table 8.2*. done is a boolean variable that indicates whether all states have been reached or not. The variable t, is stored on 32 bits. There are N pointers of queue $Q[i]$, each one points to a list of pointers (N pointers), linking to linked lists. Each thread, maintains N lists of states, the memory space used for states storage in the queue $Q[i]$ is (32+32) * s * N (s is the maximum number of states in a list of a thread). The memory space, is allocated for storage of current and future states, therefore, space is allocated for $Q[0][i][j]$ and $Q[1][i][j]$.
The size of S is determined by: (32 + 32) * t (t is the total number of states explored). We can estimate the state space storage used by the parallel algorithm proposed in [22] in bits: $Memory\_used_{algo\_SPIN}$ = 32 + 32 + 2 (32 * N) + 2 (32 * N * N) + (2* N) *((32 + 32) * s * N) + ((32 + 32) * t) = $32(1 + 1 + (2 * N) + (2 * N^2) + 4 * s * N^2 + (t * 2)) = 32(2 + N(2 + (2 * N) + (4 * s * N)) + (t * 2))$.

A less memory space used is necessary to fight the state explosion problem. From this analysis, we can conclude that our algorithm uses less memory than the parallel algorithm developed in $SPIN$.

9. Conclusion and future work. Model checking is a set of an automatic verification techniques of temporal properties on reactive systems. It takes as input, a system of transitions and a formula from some temporal logic, and answers if the abstraction satisfies or not the formula. This technique suffers from the state explosion problem, where systems become too large. In this article, we have proposed a parallel approach to the state space exploration. We realized many experiments, for a comparison between our algorithm, and the algorithm proposed in [22]. As first experiment, we measured the performance of both parallel algorithms, using Peterson model [7] then we compared the results. We showed that our approach gives better results. In the second experiment, we measured performances in terms of execution time, obtained by both parallel algorithms, using Dining Philosophers model [8], we calculated and noticed that our approach produces better results. In the third experiment, we made a comparison between both parallel algorithms, using Producer-Consumer model [9], we noticed an improvement of performance in execution time, reported by our approach. In the last experiment, we used counters model, and concluded that our approach gives better results. We work on the execution of experiments on a distributed environment, to improve performance, we are about to conduct tests, using models observed in the article.

REFERENCES

[1] N. ABED, S. TRIPAKIS, AND J. M. VINCENT, *Resource-Aware Verification Using Randomized Exploration of Large State Spaces*, Model Checking Software. In Proceedings of the 15th International SPIN Workshop, August 10-12, 2008, Los

Angeles, CA, USA, Proceedings, pp. 214–231.
[2] M. C. Boukala, and L. Petrucci, *Towards distributed verification of petri nets properties*, In Proceedings of the First international conference on Verification and Evaluation of Computer and Communication Systems VECoS '07, May, 2007, Algiers, Algeria, pp. 13–24.
[3] S. Christensen, L. M. Kristensen and T. Mailund, *A Sweep-Line Method for State Space Exploration*, In Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '2001, April, 2001, Genova, Italy, pp. 450–464.
[4] E. M. Clarke, W. Klieber, M. Novček, and P. Zuliani, *Model Checking and the State Explosion Problem*, In Proceedings of the 8th Laser Summer School on Software Engineering, September, 2011, Elba Island, Italy, pp. 1–30.
[5] N. Guan, Z. Gu, W. Yi, and G. Yu, *Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs*, In Proceedings of the 14th Asia and South Pacific Design Automation Conference, ASP-DAC '09, January, 2009, Pacifico Yokohama, Yokohama, Japan, pp. 715–720.
[6] R. Pelánek, *Fighting State Space Explosion: Review and Evaluation*, In Proceedings of the 13th on Formal Methods for Industrial Critical Systems, FMICS '08, September, 2008, L'Aquila, Italy, pp. 37–52.
[7] G. L. Peterson, *Myths About the Mutual Exclusion Problem*, Inf. Process. Lett., 12(3): 115–116, 1981.
[8] K. M. Chandy, and J. Misra, *The Drinking Philosophers Problem*, ACM Trans. Program. Lang. Syst., 6(4): 632–646, 1984.
[9] K. Jeffay, *The Real-Time Producer/Consumer Paradigm: A paradigm for the construction of efficient, predictable real-time systems*, In Proc. ACM/SIGAPP Symp. on Applied Computing, 1993, pp. 3796–804.
[10] K. Verstoep and H. E. Bal and J. Barnat and L. Brim, *Efficient Large-Scale Model Checking**, IEEE International Parallel and Distributed Processing Symposium, IPDPS'09, May, 2009, Rome, Italy, pp. 1–12.
[11] F. Lerda, and R. Sisto, *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, July, 1999, Trento, Italy, pp. 22–39.
[12] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, 23(5): 279–295, 1997.
[13] P. Dhaussy, J. C. Roger, and F. Boniol, *Reducing State Explosion with Context Modeling for Model-Checking*, In Proceedings of the 13th International Symposium on High Assurance Systems Engineering, November, 2011, Boca Raton, Florida, USA, 1990, pp. 130–137.
[14] A. Mukherjee, Z. Tari, and P. Bertok, *Memory Efficient State-space Analysis in Software Model-checking*, In Proceedings of the Thirty-Third Australasian Conference on Computer Science, ACSC '10, January, 2010, Brisbane, Australia, pp. 23–32.
[15] C. P. Inggs, and H. Barringer, *CTL* Model Checking on a Shared-memory Architecture*, Formal Methods in System Design, 29(2): 135–155, 2006.
[16] E. A. Emerson, and J. Y. Halpern, *"Sometimes" and "not never" revisited: on branching versus linear time temporal logic*, Journal of the ACM (JACM) - The MIT Press scientific computation series, 33(1), pp. 151–178, 1986.
[17] A. Pnueli, *The temporal logic of programs*, In Proceedings of the 18th Annual Symposium on Foundations of Computer Science, October, 1977, Providence, Rhode Island, USA, pp. 46–57.
[18] E. M. Clarke, and E. A. Emerson, *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic*, In Logic of Programs, Workshop, May, 1981, pp. 52–71.
[19] T. Heyman, D. Geist, O. Grumberg, and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, In Proceedings of the 12th international Conference on computer aided verification, CAV2000, July, 2000, Chicago, Illinois, USA, pp. 20–35.
[20] R. T. Saad, S. D. Zilio, and B. Berthomieu, *Mixed Shared-Distributed Hash Tables Approaches for Parallel State Space Construction*, In Proceedings of the 10th International Symposium on Parallel and Distributed Computing, ISPDC '11, July, 2011, Cluj-Napoca, Romania, pp. 9–16.
[21] R. Pelánek, *BEEM: Benchmarks for explicit model checkers*, In Proceedings of the 14th International Spin Workshop on Model Checking Software, Springer Verlag, LNCS Vol. 4595, 2007, pp. 263–267.
[22] G. J. Holzmann, *Parallelizing the Spin Model Checker*, In Proceedings of the 19th International Conference on Model Checking Software, SPIN'12, 2012, Oxford, UK, pp. 155–171.
[23] C. P. Inggs, and H. Barringer, *Effective State Exploration for Model Checking on a Shared Memory Architecture*, Electr. Notes Theor. Comput. Sci., No. 4, 2002, pp. 605–620.
[24] A. M.C.A. Koster and M. Tieves, *Network design with compression: complexity and algorithms*, INFORMS Computing Society Conference (INFORMS ICS), 2015.
[25] K. Arnold, and J. Gosling, *The Java Programming Language (2Nd Ed.)*, INFORMS Computing Society Conference (INFORMS ICS), ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.
[26] G. J. Holzmann, *The Model Checker SPIN*, IEEE Trans. Softw. Eng., 23(5): 279–295, May, 1997.
[27] M. C. Boukala, and L. Petrucci, *Distributed CTL Model-Checking and counterexample search*, International Journal of Critical Computer-Based Systems (IJCCBS), 3(1), January, 2012.

Edited by: Roman Trobec
Received: December 21, 2015
Accepted: March 31, 2016