DE GRUYTER
OPEN

BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **16**, No 1
Sofia • 2016 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.1515/cait-2016-0001

Sequential and Parallel Algorithms for the State Space Exploration

Lamia Allal[1], *Ghalem Belalem*[1], *Philippe Dhaussy*[2], *Ciprian Teodorov*[2]

[1]*Dept. Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 Ahmed Ben Bella, Oran, Algeria*
[2]*Lab-STICC UMR CNRS 6285 ENSTA Bretagne, Brest, France*
Emails: allal.lamia@gmail.com ghalem1dz@gmail.com philippe.dhaussy@ensta-bretagne.fr ciprian.teodorov@ensta-bretagne.fr

Abstract: *In this article, we are interested in the exploration part of model checking which consists in traversing all the possible states of a system. We propose two approaches to exploration, parallel and sequential. We present a comparison between our parallel approach and the parallel algorithm proposed in SPIN.*

Keywords: *Model checking, state explosion problem, sequential exploration, parallel exploration.*

1. Introduction

Model checking is a set of 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. Exploration is a computing process which determines a sequence of actions making it possible to achieve a desired goal. A good exploration means the achievement and the storage of a large number of states of a system without exceeding the available memory resources [1]. 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].

Model checking techniques suffer from a major problem known as the state explosion problem [4, 7, 16]. This problem occurs when the state space to be explored is large and cannot be explored by the algorithms for a 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.

In this article, we are interested in the time needed to carry out exploration (execution time), for that, we present a comparative study between two types of exploration: sequential and parallel, where the comparison is carried out over the execution time of each experimentation. The model used is a counter incremented from 1 to *N* and decremented to 0, with *N* a parameter fixed at the experimentations.

This article is divided into eight sections: Section 2 is devoted to the model checking and to the state space exploration; Section 3 presents some works which offer solutions to the state explosion problem. Sections 4 and 5 are devoted to the definition of the sequential and parallel approaches proposed. Section 6 is devoted to the experiments performed for a comparison between sequential and parallel approaches. Section 7 presents the parallel algorithm for state exploration in SPIN.

2. Model checking and the state space exploration

Model checking is a verification technique based on the exhaustive state space exploration of systems in search 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. The implemented algorithms 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 advantage of using model checking is the accuracy of the answer obtained [12]. The exploration phase consists on visiting state by state, each state and all its successors are stored in memory. Exploration ends 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.

3. Related work

Many solutions have been proposed for the state explosion problem [7, 12]. In this section, we present three solutions. Each one of them is running on a different architecture (distributed, parallel, and sequential). These solutions are based on different methods and data structures. Each solution aims to improve the performances in execution time and memory capacity. The approach proposed in [14] allows a distribution in the exploration of states when checking a model by the model checker SPIN [6, 8]. The treatment is performed by exploiting a network of machines. Each node holds a set of states to be treated. The authors present an algorithm composed of three functions: (I) Start, (II) Visit, (III) DFV (Depth First visit). Each node holds a set *V*, containing the visited states and a queue *U* for the storage of the unexplored states. At the initialization step, each process executes the Start procedure in order to identify the number of the processor that will treat the initial state. With each time a successor is generated, a checking is carried out on the index of the node. If the processor *i* treated the state *e*, *i* generates the successors of *e* and checks if it must visit them or not. That is performed by a partition function. Exploration finishes when all the queues of nodes are empty. The disadvantage of this method lies in the choice of the partition function which should enable achieving a uniform distribution of states in terms of memory space, tine execution and messages transferred across the network.

The approach proposed in [15] is based on a sequential algorithm. Its objective is the storage of states in their compressed form, 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 approach proposed in [17] is based on a parallel algorithm. The authors are based on model checking on a shared memory multiprocessors architecture. The solution is based on a bloom filter [17] to indicate if a state was visited or not. A problem occurs when the bloom filter may return a false result, it can make a false positive when the state has not been seen, causing collisions.

4. Sequential algorithm proposed

An algorithm corresponds to a succession of states, and transitions between these states, the idea is that the states correspond to instantaneous descriptions of the algorithm. An exploration algorithm is defined mainly by three parameters: a condition stop of the exploration, a function of selection of next states to explore, starting from a state or a set of states already explored and stored in memory, and a function of actualization which defined the storage of new states in memory. The execution of two sequences is known as sequential when one starts only when the other one is finished. In a sequential approach, exploration is carried out by one thread only, the time expressed compared to the number of states generated during exploration can be huge if this number is very high.

In this section, we present an algorithm for a sequential exploration of state space, we explore a model defined by a counter incremented from 1 up to *N* and decremented to 0. We carried out an exploration of this model by considering all possible states of the system while varying the number of counters as well as the parameter *N*. In a sequential exploration, the states are visited one following the other one, the first visited state is the initial state, thereafter, the successors of this state are generated, then stored in memory. This process stops when all the states are treated.

The algorithm of sequential exploration is presented as

Algorithm 1. Sequential exploration algorithm

```
Step 1.  $S \leftarrow S_0$
Step 2.  $M \leftarrow S_0$
Step 3.  while ¬ (M.isEmpty ()) do
Step 4.      X ← M.dequeue ()
Step 5.      Successors = X.GetSuccessors ()
Step 6.      for (State K: Successors) do
Step 7.          if ¬ (S.Contains (K)) then
Step 8.              S.Add (K)
Step 9.              M.Add (K)
Step 10.         end if
Step 11.     end for
Step 12. end while
```

Here S_0 represents the initial state, *S* defines the whole of the already explored states and *M* the whole of the states whose successors were not observed yet. As long as the set *M* is non-empty (Step 3 of Algorithm 1) the states are treated, Step 6 indicates that for each next state of the visited one, a checking is carried out on the set *S* to know if the state is old or new. Fig. 1 presents the steps of a sequential exploration. Only one process carries out the treatment since the initial state until the end of exploration. This process is repetitive, it is composed of five principal actions which are:

- Generation of the initial state (Step 1);
- Verification of all states to explore: if this set is not empty go to Step 2 (Fig. 1) else the exploration ends (Step 3);
- Generation of next states;
- Storage of new states in two sets (*S* and *M*);
- End of the exploration.

To better illustrate this approach, we have defined an example composed of two counters incremented up to 3 and decrement to 0.

Fig. 2 shows the accessibility graph of this example made up of all possible states. At the beginning of the exploration, both counters are set to zero. All possible states are generated from each configuration (state). In this example, there are 16 configurations. The exploration is carried out by a single thread. The configuration (0, 0) represents the initial state, it is stored in two data structures, a hash table *S* that contains all observed states and a queue *M* containing the states whose successors have not been processed yet. In the first step, both counters are initialized to zero. Thereafter, their next configurations are observed, the counters can be incremented, then we can have two possible states (0, 1) and (1, 0). Both states are new, they are stored into *S* and *M*. At each step, a state is taken from the queue *M* and its successors are visited. From the state (1, 0), we obtained three states – (2, 0), (1, 1) and (0, 0). The process is the same, a generated state is stored in memory if and only if it is not contained in *S*. States (2, 0) and (1, 1) are new, they are added to *S* and *M*. This process is repeated until *M* is empty. In this example the exploration ends when the configuration (3, 3) has been observed.

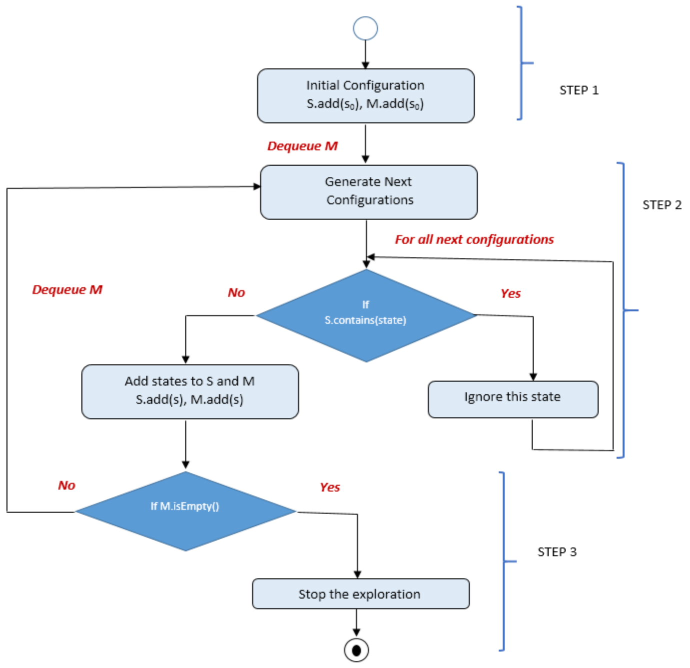


Fig. 1. Sequential exploration steps

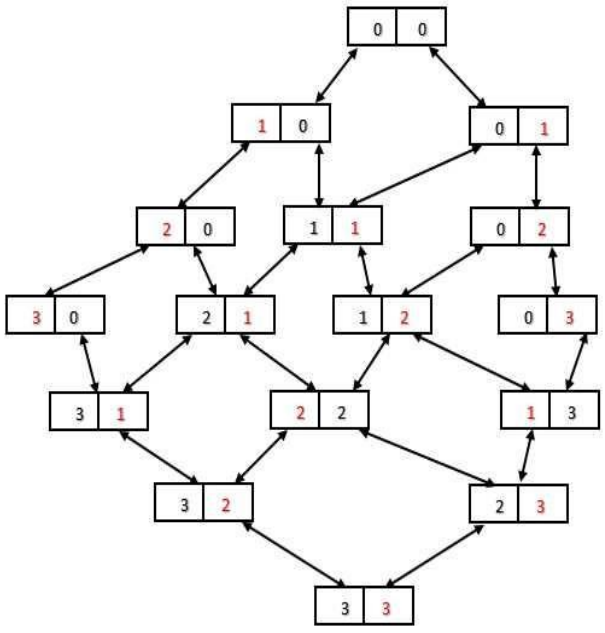


Fig. 2. Reachability graph

5. Parallel algorithm proposed

A parallel algorithm runs on a parallel computer to solve a given problem. Calculations of a sequential program are decomposed into tasks and each one is assigned to a process. 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 processes to have a load balancing between threads so that all processors or cores has the same load or almost to treat. In our approach, we used the framework executor [5] where each process performs a function returning all visited states, the objective is to be able to easily choose which tasks will be executed (Fig. 3), in what order and how many tasks can be run in parallel.

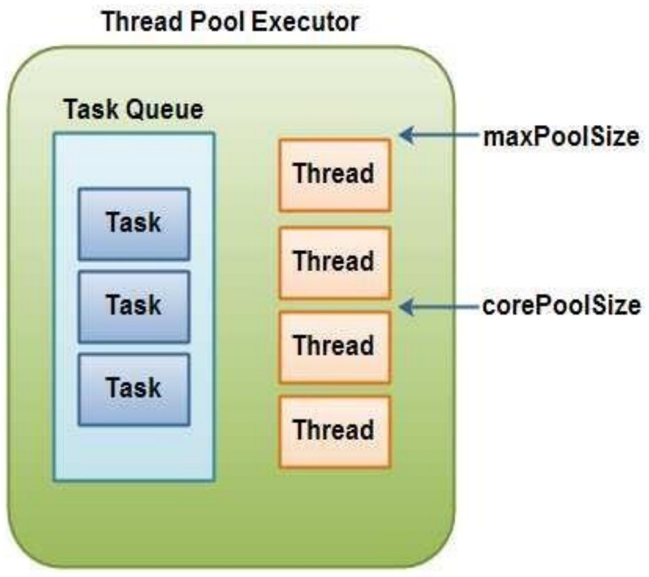


Fig. 3. Main components of the framework executor

Fig. 4 shows the execution steps of the parallel approach proposed. The treatment begins with the initial state generation, it is stored in the HashSet S containing all visited states and in the queue M that contains the states whose successors have not been observed yet. Subsequently, the initial state is removed from the queue and all its next states are generated, then for each one, a process is launched and this state is assigned, this means that the accessibility graph of this configuration is created by the thread. Each process simultaneously processes a set of configurations according to the state assigned initially. Each thread will execute the sequential code produced by a single process. The HashSet S is protected, the processes are synchronized. At the end of the exploration, each process returns the set of observed states, which means that S is built by all threads. The algorithm of parallel exploration is presented in Fig. 4.

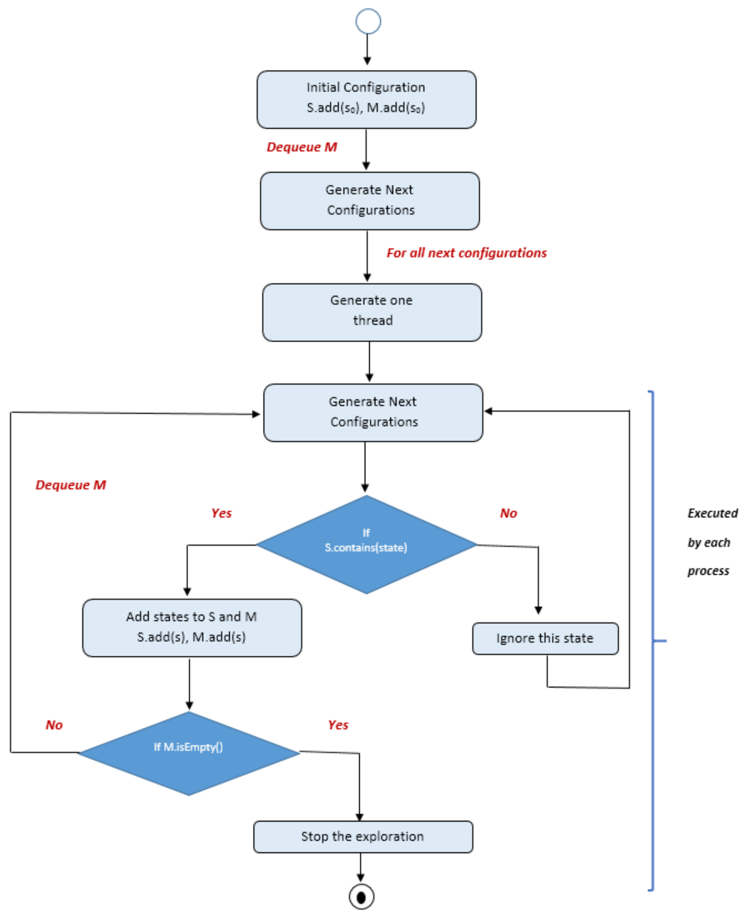


Fig. 4. Parallel exploration steps

In the parallel approach, each thread executes the sequential code described in the previous section. At the beginning of exploration, the initial state is generated, then its next states are observed. At this time, for each new configuration, a process is started (Step 6 of Algorithm 2). The latter is active until all states have been treated. Each thread executes a function for the state space exploration, therefore, data used are local, except for the HashSet *S* whose access is shared.

Taking the same example as above composed of two counters which are incremented up to 3 and decrements to 0, the initial state is generated first and stored in the sets *S* and *M* , thereafter its successors (1, 0) and (0, 1) are visited and stored in their turn in both sets, after that, two threads are started and each one will execute the exploration code using a queue *M* containing local states whose successors have not been observed.

The exploration stops when both local queues reserved for the generated threads are empty. Fig. 5 shows the reachability graph generated by the parallel algorithm. From this figure, we notice that both processes generate the state (1, 1), this state is treated by a single process that will explore all its successor states.

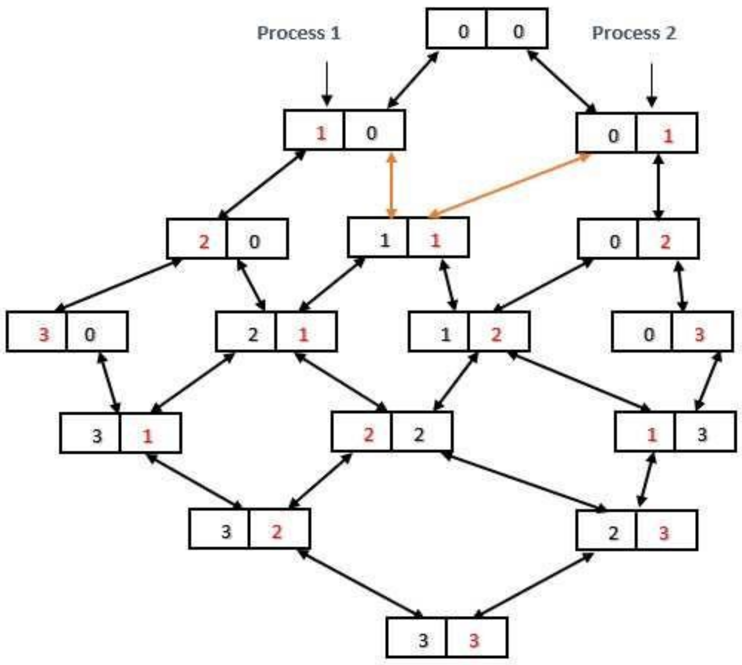


Fig. 5. Reachability graph

Algorithm 2. Parallel exploration algorithm

```
Step 1.  $S \leftarrow S_0$
Step 2.  $M \leftarrow S_0$
Step 3.  $X \leftarrow M$.dequeue ()
Step 4. for allNextStates do
Step 5.      Tasks.add (work)
Step 6.      Executor.Submit (Tasks)
Step 7. end for
Step 8.  while $\neg$ ($M.isEmpty$ ()) do
Step 9.      $X \leftarrow M$.dequeue ()
Step 10.     Successors $\leftarrow X$.GetSuccessors ()
Step 11.     for (State $K$: Successors) do
Step 12.         if $\neg$ ($S$.Contains ($K$)) then
Step 13.             $S$.add ($K$)
Step 14.             $M$.add ($K$)
Step 15.         end if
Step 16.     end for
Step 17.  end while
```

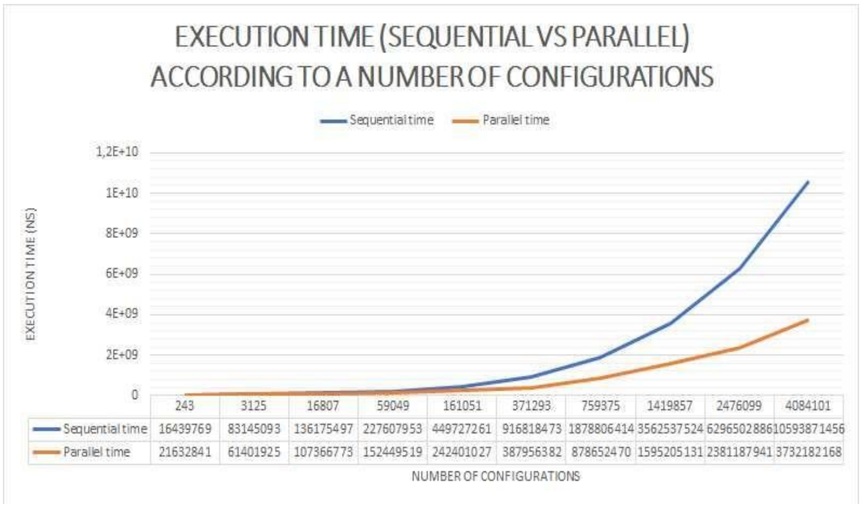
6. Experimental study

In this article, we proposed two approaches for sequential and parallel exploration. We carried out three experimental studies, the objective is to make a comparison between sequential and parallel approaches and the proposed algorithm for a parallel exploration in SPIN. This comparison is based on the execution time. The experiments are performed by varying the number of configurations. 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 algorithms (sequential and parallel).

6.1. Sequential approach vs parallel approach

To study the performance of both approaches (sequential and parallel) in execution time, we varied two parameters: the number of counters and the maximum value of the counter, called *V*. The metric considered is the running time representing the time elapsed to perform the exploration.

Fig. 6 shows the results of an experiment using five counters and varying the parameter *V* from 2 up to 20. To interpret these results, we divided this experience in two parts.



	243	3125	16807	59049	161051	371293	759375	1419857	2476099	4084101
Sequential time	16439769	83145093	136175497	227607953	449727261	916818473	1878806414	3562537524	6296502886	10593871456
Parallel time	21632841	61401925	107366773	152449519	242401027	387956382	878652470	1595205131	2381187941	3732182168

Fig. 6. Execution time (sequential vs parallel) by varying the number of configurations

Fig. 7 shows the results of exploration using a reduced number of configurations. The sequential algorithm produced better results than the parallel algorithm when *V* is equal to 2, because for a small number of configurations, multiple threads are created. The process generation takes a lot of time which explains the result shown in Fig. 7. The parallel approach shows better results from 3125 configurations. The average gain of the parallel approach in the experiment 1 against the sequential approach is 10%. This gain increases by changing the parameter *V*. According to Fig. 8, where *V* is high, the number of configurations increases. In this experiment, the average gain of the parallel approach is 17%. We can conclude that the sequential approach doesn't scale and is blocking for a high number of configurations. The execution time is expressed in nanoseconds.

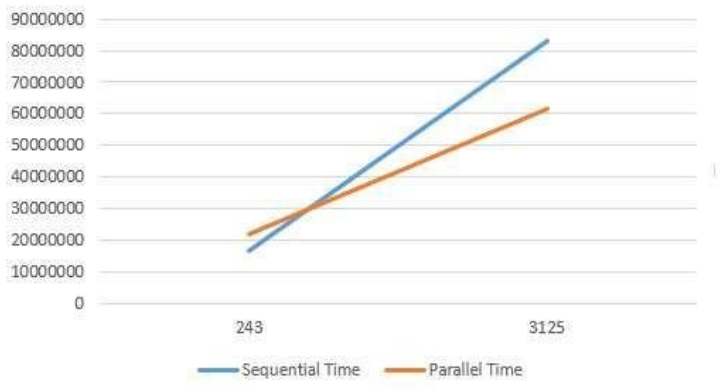


Fig. 7. Experience 1

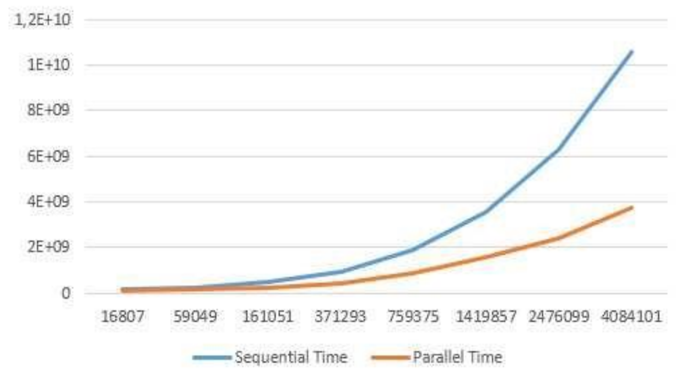
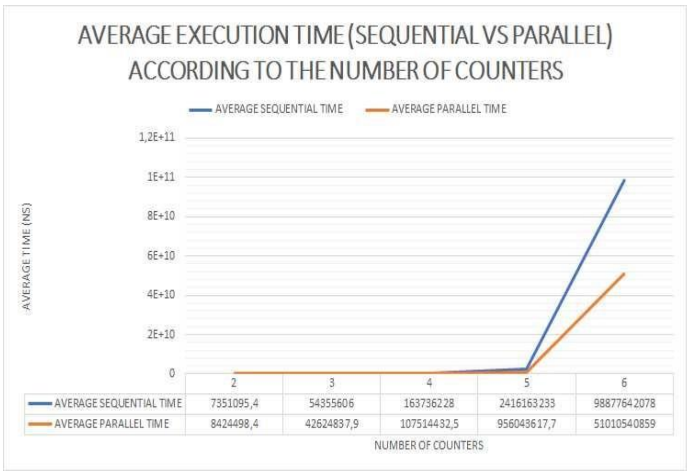


Fig. 8. Experience 2

Fig. 9 shows the average execution time for both approaches by varying the number of counters. This number varies from 2 up to 6. Five experiments were performed by setting the number of counters (from 2 up to 6) and by varying the parameter *V* from 2 up to 20. This experience shows the average execution time returned for each exploration. This figure shows that the parallel approach gives better performance compared to the sequential approach. This gain is fixed at 10%. For a large number of configurations, sharing tasks between processes is necessary to save time.

A parallel program is divided into several sequential tasks running simultaneously. When tasks are big, the use of parallelism is the best solution.



	2	3	4	5	6
AVERAGE SEQUENTIAL TIME	7351095,4	54355606	163736228	2416163233	98877642078
AVERAGE PARALLEL TIME	8424498,4	42624837,9	107514432,5	956043617,7	51010540859

Fig. 9. Execution time (sequential vs parallel) by varying the number of counters

6.2. Parallel approach vs SPIN

We have made a parallel comparison between our approach and the parallel algorithm developed for SPIN model checker [9]. The algorithm (Algorithm 3) proposed in [9] is based on the use of a three dimensional queue $Q[t][i][j]$ for the storage of states whose successors have not been observed yet. The parameter i is varied from 0 up to 1, it allows states to pass from current 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 configuration that will be observed at the next step. The algorithm was designed to avoid locks when accessing in mutual exclusion. The set S in which visited states are stored is protected to prevent simultaneous access. A lockless hachtable [10] was used in order to avoid waits between different threads. An important task in the algorithm proposed in [9] is to determine when all states have been explored to stop exploration.

In the algorithm, the parameters i and j vary from 1 up to N, where N is the number of threads. The size of this queue is unlimited. Each time a state is generated, a check is made on a set S: if the state is new, it is stored in S and added to the queue Q by randomly choosing a thread that will handle it.

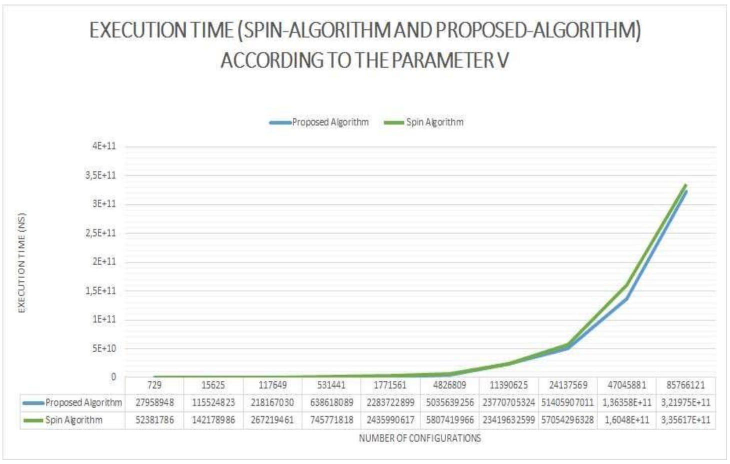
Algorithm 3. Parallel exploration algorithm in SPIN

```
Step 1.  $done \leftarrow false$
Step 2.  $t \leftarrow 0$
Step 3.  Search ($i$: 1..$N$)
Step 4.  for ($j$=1; $j$<=$N$; $j$++) do
Step 5.      Delete s from Q[$t$][$i$][$j$]
Step 6.      for (Each next configuration $c$ of $s$) do
Step 7.          if $\neg$ (Contains($c$)) then
Step 8.              $S$.add($c$)
Step 9.              $k \leftarrow$ Choose Random from 1… $N$
Step 10.             add state to $Q$ [1–$t$] [$k$] [$i$]
Step 11.         end if
Step 12.     end for
Step 13. end for
Step 14. Wait ()
Step 15. if (i==1) then
Step 16.     wait until all threads are idle
Step 17.     if (all Q [1–t] [i] [j] == NULL) then
Step 18.         done ← true
Step 19.     else
Step 20.         Notify all threads
Step 21.         t ← 1–t
Step 22.     end if
Step 23. end if
```

The current states to be visited are treated from $Q[t][i][j]$ and next states are stored in

$Q[1-t][k][i]$ with k, a randomly selected process from N .

We compared our parallel algorithm with the parallel algorithm developed for SPIN model checker [9] using the same example as before. We conducted experiments on the same machine. We used six counters, the parameter V varies from 2 up to 20 (Fig. 10). To interpret these results, we calculated the gain (in percentage) obtained from each experience, this gain is presented in Fig. 11. We note that the proposed parallel algorithm shows better performance than the parallel algorithm proposed for SPIN for a number of configurations varying from 729 up to 85 766 121. The average gain in response time obtained by our parallel approach is about 1.5% compared to the parallel approach proposed for SPIN. So compared to the sequential approach where the number of configurations influenced on the final result, the proposed parallel algorithm follows the same pace as the proposed algorithm for SPIN an improvement in execution time at each experience. In the algorithm presented in [9], whenever a thread finishes processing its tasks, it waits for other processes, therefore, this algorithm is based on the processing of states step by step.



	729	15625	117649	531441	1771561	4826809	11390625	24137569	47045881	85766121
Proposed Algorithm	27958948	115524823	218167030	638618089	2283722899	5035639256	23770705324	51405907011	1,36358E+11	3,21975E+11
Spin Algorithm	52381786	142178986	267219461	745771818	2435990617	5807419966	23419632599	57054296328	1,6048E+11	3,35617E+11

Fig. 10. Execution time (both parallel approaches) by varying the number of configurations

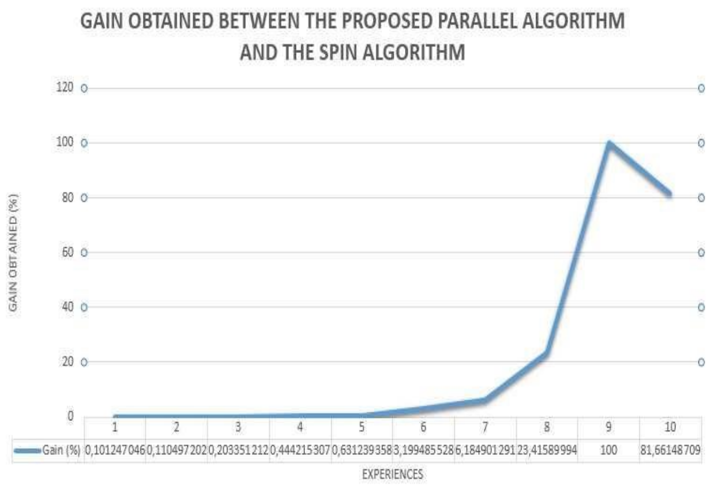


Fig. 11. Gain obtained between both approaches (parallel and SPIN)

6.3. Sequential approach vs parallel approach vs SPIN

We carried out a comparison between both proposed approaches and parallel algorithm presented in [9]. The comparison was made on the running time. We used 6 counters incremented from 0 up to 20. We divided the result on two figures (Figs 12 and 13). Fig. 12 shows that the proposed parallel approach is better than the other ones compared to the result shown in Fig. 10 where the sequential algorithm was better when the number of configurations was small, therefore, the number of counters influences on the execution time. With regard to Experiment 2 shown in Fig. 13, the proposed parallel approach gives better results even by increasing the parameter *V*. Both parallel approaches show significant results when the number of configurations is high with a gain provided by the proposed parallel algorithm.

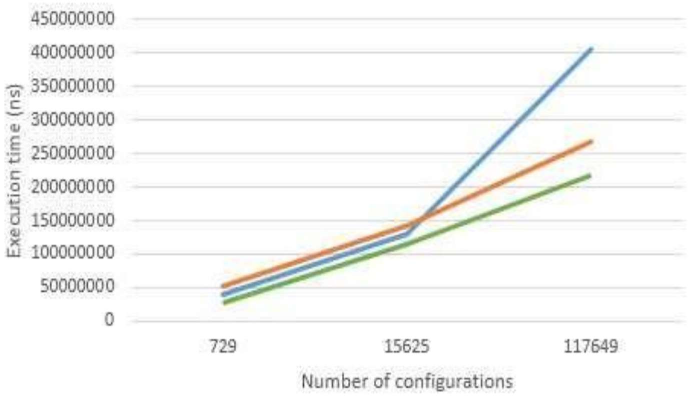


Fig. 12. Experience 1

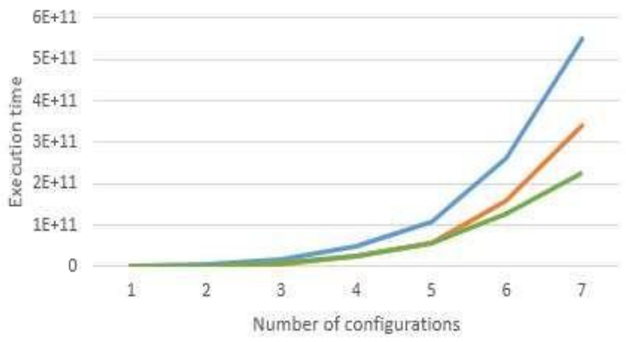


Fig. 13. Experience 2

7. Positioning our parallel approach and discussion

To position our parallel algorithm with the integrated parallel algorithm in SPIN, which is often used in the explorations of states using model checking techniques, we studied two aspects: linear regression [13] and complexity [11].

7.1. Study of regression

We realized that experience to predict execution time in the future with information from the past and therefore have knowledge about the future behavior of our algorithm compared to the algorithm proposed in [9]. For this, we created a regression line for the generated point (the scatter plot) by the simulation results. The line is obtained using the results of Fig. 10. The equation of the regression line is in the form $Y = aX + b$. In this example, for the proposed parallel approach, it is given by $Y_{\text{Par}}=2896.9X-1.9189\times10^{9}$. For SPIN, it is represented by $Y_{\text{SPIN}} = 3409.2X-2.3\times10^{9}$. To perform these calculations, we used the median-median method [18]. To represent the estimated errors in execution time, we determined the confidence interval of the regression slope that can define an error bounds between approximate and real results.

The confidence interval is calculated by

$$I_{\text{c}} = (\bar{X} - t_{\alpha}\frac{s}{\sqrt{n}};\ \bar{X} + t_{\alpha}\frac{s}{\sqrt{n}}),$$

where $\bar{X}$ is the sample mean, s is the standard deviation and n is the number of configurations. For Y_{Par}, the confidence interval I_{c} = [1935.02; 5196.82]. For Y_{SPIN}, the confidence interval I_{c} = [2229.93; 5448.87]. After estimation obtained by both straight lines of the linear regression, we can predict the execution time for a number of states greater than 85 million. For example, for 200 million states, using the regression line Y_{Par}, we can estimate a response time (taking into account an error bounds determined by I_{c}) by 5.775×10^{11} ms. Using Y_{SPIN}, execution time is estimated by 6.795×10^{11} ms. Based on these results, we can conclude that our approach gives better performance in time and allows scalability. SPIN technique becomes impractical when the number of states is very important like in critical applications.

7.2. 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: Our proposed 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* or *M*
p: number of processes
w: waiting time per process

- **Complexity of the proposed parallel algorithm.** Complexity of the proposed algorithm C_{App} is estimated by:

(1) $C_{\text{App}} = 3.\text{ae} + q(\text{ce} + \text{ae} + s.\text{ae} + s\,(\text{ce} + 2.\text{ae})) = O(q).$

- **Complexity of the parallel algorithm proposed in [9].** Complexity C_{SPIN} is estimated by:

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

According to these complexities obtained by equations (1) and (2) we can notice that our algorithm has order of $O(q)$ time complexity, the complexity of the algorithm proposed in [9] is around the square estimated to $O(q^2)$. 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. Conclusion

Model checking is a technique based on three concepts: a model system to check, a specification in the form of a system property and algorithms to check whether the model meets its specification. This technique suffers from the state explosion problem where systems become too large. We have proposed two approaches, sequential and parallel to the state space exploration. For our first experiment, we measured the performance of both algorithms then we compared the results. We showed that the sequential approach gives better results when the number of configurations is reduced and that beyond a certain number of states, the parallel algorithm gives better performance then. For our second experiment, we measured the execution time obtained by the proposed parallel algorithm and parallel algorithm proposed in [9], we calculated the gain provided by the experience and noticed that our approach gives better results. Currently, we are studying exploration on real models to perform comparisons between these algorithms. We plan in the near future to implement our algorithms in a distributed architecture composed of a set of nodes.

R e f e r e n c e s

1. A b e d, N., S. T r i p a k i s, J.-M. V i n c e n t. Resource-Aware Verification Using Randomized Exploration of Large State Spaces. − In: Proc. of 15th International SPIN Workshop, Model Checking Software, Los Angeles, CA, USA, August 10-12 2008, pp. 214-231.
2. B o u k a l a, M. C., L. P e t r u c c i. Towards Distributed Verification of Petri Nets Properties. – In: Proc. of First International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS'07, May 2007, pp. 13-24.
3. C h r i s t e n s e n, S., L. M. K r i s t e n s e n, T. M a i l u n d. A Sweep-Line Method for State Space Exploration. – In: Proc. of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'2001, April 2001, pp. 450-464.
4. C l a r k e, E. M., W. K l i e b e r, M. N o v á £ e k, P. Z u l i a n i. Model Checking and the State Explosion Problem. – In: Proc. of 8th Laser Summer School on Software Engineering, Vol. **7682**, September 2011, pp. 1-30.
5. D a u g h e r t y, J. Java Concurrency Framework. CSCI 5448, Spring 2011, May 2011.
6. D h a u s s y, P., J. C. R o g e r, F. B o n i o l. Reducing State Explosion with Context Modeling for Model-Checking. – In: Proc. of 13th International Symposium on High Assurance Systems Engineering, HASE, November 2011, pp. 130-137.
7. E v e r**,** E., O. G e m i k o n a k l i, A. K o ç y i g i t, E . G e m i k o n a k l i. A Hybrid Approach to Minimize State Space Explosion Problem for the Solution of Two Stage Tandem Queues. − J. Network and Computer Applications, Vol. **36,** 2013, No 2, pp. 908-926.
8. G u a n, N., Z. G u, W. Y i, G. Y u. Improving Scalability of Model-Checking for Minimizing Buffer Requirements of Synchronous Dataflow Graphs. – In: Proc. of 14th Asia and South Pacific Design Automation Conference, ASP-DAC'09, January 2009, pp. 715-720.
9. H o l z m a n n, G. J. The Model Checker SPIN. − IEEE Transactions on Software Engineering, Vol. **23**, May 1997, No 5, pp. 279-295.
10. H o l z m a n n, G. J. Parallelizing the Spin Model Checker. – In: Proc. of 19th International Conference on Model Checking Software, SPIN'12, Berlin, Heidelberg, 2012. Springer-Verlag, pp. 155-171.
11. I n g g s, C. P., H. B a r r i n g e r. Effective State Exploration for Model Checking on a Shared Memory Architecture. − Electr. Notes Theor. Comput. Sci., Vol. **68**, 2002, No 4, pp. 605-620.
12. K a m k i n, A. S. Projecting Transition Systems: Overcoming State Explosion in Concurrent System Verification. − Program. Comput. Softw., Vol. **41**, November 2015, No 6, pp. 311-324.
13. K o s t e r, A. M.C.A., M. T i e v e s. Network Design with Compression: Complexity and Algorithms. – In: Proc. of INFORMS Computing Society Conference (INFORMS ICS), 2015.
14. K w i a t k o w s k a, M., G. N o r m a n, D. P a r k e r. Prism: Probabilistic Model Checking for Performance and Reliability Analysis. – ACM SIGMETRICS Performance Evaluation Review, Vol. **36**, 2009, No 4, pp. 40-45.
15. L e g g e t t e r, C. J., P. C. W o o d l a n d. Maximum Likelihood Linear Regression for Speaker Adaptation of Continuous Density Hidden Markov Models. – Computer Speech and Language, Vol. **9**, 1995, No 1, pp. 171-185.
16. L e r d a, F., R. S i s t o. Distributed-Memory Model-Checking with Spin. – In: Proc. of 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, Vol. **1680**, July 1999, pp. 22-39.
17. M u k h e r j e e, A., Z. T a r i, P. B e r t o k. Memory Efficient State-Space Analysis in Software Model-Checking. – In: Proc. of Thirty-Third Australasian Conference on Computer Science ACSC'10, Vol. **102**, January 2010, pp. 23-32.
18. P e l á n e k, R. Fighting State Space Explosion: Review and Evaluation. – In: Proc. of 13th Conference on Formal Methods for Industrial Critical Systems FMICS'08, Vol. **5596**, September 2008, pp. 37-52.
19. S a a d, R. T., S. D. Z i l i o, B. B e r t h o m i e u. A General Lock Free Algorithm for Parallel State Space Construction. – In: Proc. of 9th International Workshop on Parallel and Distributed Methods in Verification, PDMC-HIBI'10, October 2010, pp. 8-16.
20. W a l t e r s, E. J., C. H. M o r r e l l, R. E. A u e r. An Investigation of the Median-Median Method of Linear Regression. – Journal of Statistics Education, Vol. **14**, 2006, No 2.