Using parallel and distributed reachability in model checking

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

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

Abstract. In the life cycle of any software system, a crucial phase of formalization and validation by means of verification and/or testing, leads to the identification of probable errors infiltrated during its design. Model-checking is one of the formal verification techniques. This technique is very powerful, but limited by the state explosion problem, that occurs when the model to be checked is too large, and can not be verified for lack of memory space. In this article, we cite two solutions, parallel and distributed, which aim to reduce the state space explosion. A comparative study between these approaches is carried out on a counters model.
Keywords: *Model checking, state explosion problem, parallel exploration, distributed exploration, formal verification*

1 Introduction

Electronic circuits, communication protocols and software are now part of everyday life. They have become so indispensable and ubiquitous, that the problem of the safety of their functioning arises. Considering, for example, a program assumed to perform a given task, it is desired that it performs this task correctly on all its possible inputs. We would also like to be able to verify other types of properties: for example, the fact that the program never wrongly assigns variables, or that it never enters in a blocking state. Today, thanks to various verification techniques, grouped under the term of "formal methods", known to check precisely that an electronic circuit complies with its specifications. Within the formal methods, there are three main families of verification techniques: simulation and testing, automatic theorem proving, and model checking.
In this article we are interested in the solving of the systems by model checking and more precisely by the state explosion problem [4, 3, 6, 11], that occurs, when the model to be checked is too large and needs high performances in time and memory space. We study in this paper two methods, to deal with this problem, the first one is a parallel solution presented in [2] and the second one is distributed.
Section 2 presents some proposed solutions to fight the state explosion problem. Sections 3 and 4 detail both parallel and distributed solutions. A serie of experiments is presented in section 5. In Section 6, we review some previous work, which has a related relationship with our contribution and by discussing and showing their differences. Finally, we conclude our article with a synthesis and highlighting some future work.

2 Model checking language

Formal methods are languages, techniques and tools that rely on mathematical logic. The main objective behind using these methods in software development is to prove that programs are correct. In order to achieve the objective of formal methods and to guarantee the accuracy of a system, we must first specify its model using a given formalism and then define the set of properties that the system must satisfy. These properties can be given as a logical formula. There are two types of formal methods to prove that a model verifies a set of properties: theorem proving and exploration of state space. In this paper, we are interested in checking systems by model checking. This method is based on the modeling of the set of system states (state space) that describe all accessible states of the system and the links between them. One of the main limitations of model checking is the state explosion problem : the model representing a complex system is often gigantic. Several methods have been proposed to combat this problem. In the following section, we present both parallel and distributed solutions.

3 Synchronized parallel algorithm (SPA)

In this section, we present the synchronized algorithm proposed in [2]. Authors used a parallel solution to fight the state explosion problem in execution time. When we have to check a model with a high number of parallel components, due to a lack of memory space, exploration step cannot be done correctly, which leads to a state space explosion. A solution to this problem is to divide the graph into several clusters, and to assign each set of configurations to a machine. Dividing the graph at each explosion in memory space, will lead on an explosion in execution time. The solution consists in fixing the number of processes carrying out the exploration step during the experiments performed. States are stored in a concurrent queue Q[i] with i varying from 1 to the number of processes (N processes). This number is fixed during the experiments. The queue Q[i] is a fifo queue containing states that need to be processed. Whenever a state is generated, a process id is generated randomly to process that state. The state is stored in Q[id] as well as in a set K containing the list of explored states. Exploration is done in parallel. The end of the exploration is triggered when the queue Q[i] (i varying from 1 to N) is empty. The instructions of the SPA algorithm are as follows:

The queue Q[i] is synchronized between the different processes (the queue is a critical resource, only one process can add or remove state from the queue) and allows the storage of unexplored states. . The Process with the identifier 0 starts the exploration, by generating the initial state and its successors and they will be stored by a process chosen randomly. Subsequently, all processes perform the same process:

- Take a state s from a queue Q[i] ;
- Generate its successors ;
- For each successor, generate a process id j randomly, to explore it. This state is stored in Q[j] ;
- The end of exploration is trigged when Q[i] is empty (for i varying from 1 to N).

4 Distributed proposed

The concept of distributed architecture is opposed to the centralized architecture. A distributed system is a set of an independent computers, connected by network and

Algorithm 1 Synchronized Parallel Algorithm (SPA)

```
1:  $exploration\_done \leftarrow false$
2:  For each (w: 1..N)
3:  DO
4:  for (each s in Q[w]) 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
```

communicate via this network. This set appears from the point of view of the user, as a single entity. The proposed approach consists in distributing the exploration step into 2 machines, the purpose of the approach, is to have more memory space to store more states.
Communication between both machines is done by sockets [13] on java [5], using an ip address and a port number. Sockets are used to manage incoming and outgoing flows to ensure reliable communication between these machines. Both machines are linked by the same TCP port number, so that the network layer can identify the sharing data. Each machine will explore its states, and store them in the set of explored states known as *known*. The distributed program running the reachability analysis, contains 3 classes.

The explorator class, responsible of performing state exploration. Each state of the queue Q[i] is visited by generating its successor states. The visited state is put in the set *known* containing the explored states. Its successor states are stored in the set *toSee*, containing the states that have not been visited yet. The exploration algorithm is as follows:

Algorithm 2 Exploration Algorithm

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

The class of states reception is used to receive messages between 2 nodes. The states reception program runs continuously waiting for incoming states. We used a bufferedReader and a printStream for sending and receiving data. The code is as follows:

Algorithm 3 Algorithm of states reception

```
1: BufferedReader reader=new BufferedReader();
2: **while** (true) **do**
3:     State = reader.readLine();
4:     toSee.add(State);
5:     Known.add(State);
6: **end while**
```

The sending class is used to send messages between both nodes, the code is as follows:

Algorithm 4 Sending Algorithm

```
1: PrintStream print = new PrintStream();
2: print.println(state);
```

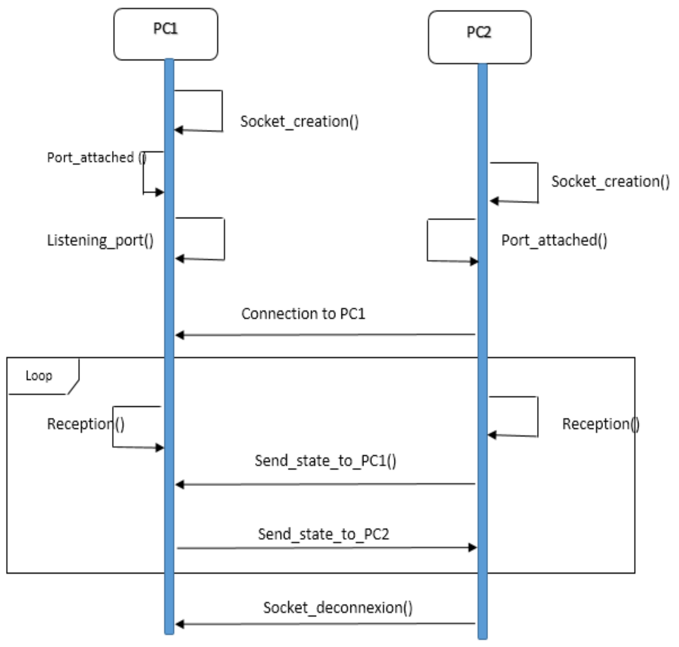

Fig. 1: Distributed exploration using 2 nodes

Fig. 1 illustrates the execution steps of the distributed algorithm. The environment consists of 2 nodes connected across the network. Communication is done via sockets. One of the machines (PC1), will play the role of the server that will initiate the conversation. Initiation involves creating the socket. The server runs on a well defined machine and is bound to a specific port number. The server listens to the client (PC2), which requests a connection. After that, an exchange of messages is carried out between both entities. Exchanged messages contains the configurations to be treated. states are received continuously until there are no more states to explore. Each node will explore states stored in Q, generate their successors and store them in the set *known*.

5 Experiments

In order to compare between parallel and distributed approaches, we performed two types of experiments using a counter model. An example of a part of reachability analysis of 3 counters, that are incremented up to a value Max (defined on experiments), and decremented down to 0, is shown in *Fig. 2*. The parallel experiments were performed on an i7 machine with 8 cores, it operates at a frequency of 2 MHz, with 16 GB of physical memory. For distributed experiments, we used the same machine and another node with 4 GB of physical memory. We have implemented both algorithms (parallel and distributed).

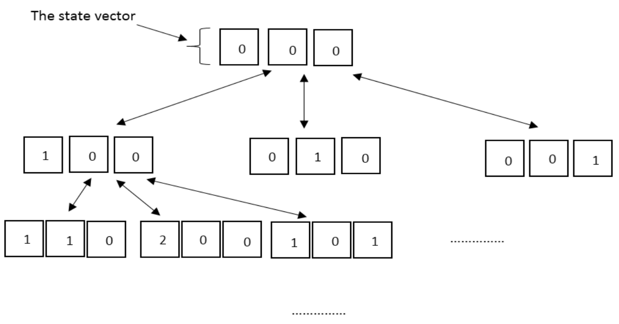

Fig. 2: A part of Reachability graph of 3 counters incremented up to Max and decremented down to 0

5.1 Experiments using 5 counters

In a first step, 5 counters were used, the value Max varies from 17 to 25, which corresponds to the input parameters of the program. Each counter is incremented to a Max value and decremented down to 0. The number of configurations (states) varies from 1889568 to 11881376. The number of configurations to be explored is calculated as follows: Nb_config $= ((the\_max\_value + 1)^{the\_number\_of\_counters}$. The number 1 corresponds to the minimum value 0. The experiments were carried out under java. From *Fig. 3*, the parallel solution provides a faster exploration of state space compared to the distributed approach. This is due to the fact that in a parallel architecture, access to memory is done directly. In the distributed solution, we used two machines, the communication between them is via sockets, a lot of messages are exchanged, which causes a saturation in the network. The average gain of the 9 experiments (in seconds) provided by the parallel approach is estimated to 39 seconds.

– **Limits of the parallel approach :** Concerning the experiment of *Fig. 3*, the maximum value on which the counters can be incremented is 25, equivalent to 11881376 configurations. Beyond this number, the simulation stops by displaying the error "out of memory". For the distributed approach, even if the execution time is higher, the simulation ends by increasing the number of configurations. We performed the exploration with a Max value equals to 26 and 27 which corresponds to 14348907 and 17210368 states respectively (see *Table 1*). In these two executions, the exploration ended successfully, so the distributed approach is more advantageous by increasing the number of configurations to be explored.

5.2 Impact of model on the reachability analysis

By taking 5 counters with a Max value equal to 26, the parallel execution did not return results. But by taking a model of counters composed of 14 million states, with 4 counters and a Max value equivalent to 61, the exploration is carried out successfully. This is explained by the fact that the exploration is not done in the same way by

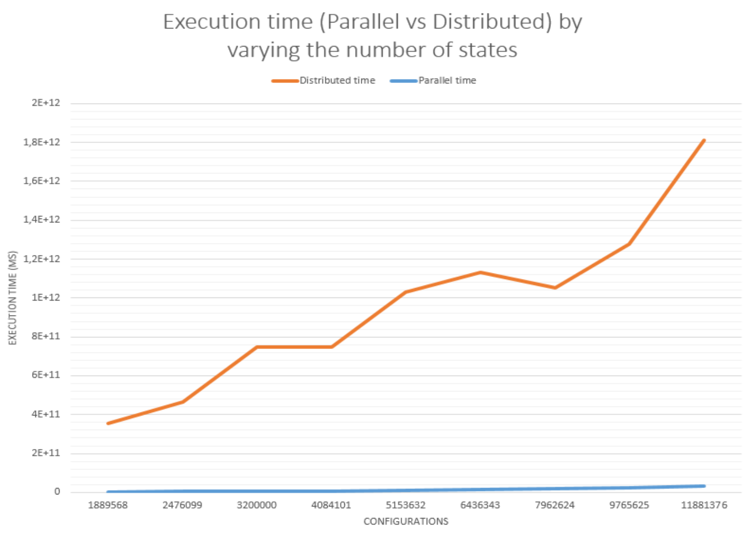

Fig. 3: Execution time (parallel vs distributed approaches) by varying the number of configurations

Table 1: Execution time of largest counters model

Configurations	Execution time
14348907	39 minutes
17210368	76 minutes

modifying the number of counters since the number of successor states as well as the state vector of each state will increase which needs more memory space.

5.3 Used memory space

The amount of memory used for the SPA parallel algorithm is described in [2]. The authors have focused on the analysis of complexity in execution time, they have defined an execution time for each type of instruction (affectation, comparison, storage, ...)
In terms of the memory space used to run the distributed algorithm, it is necessary to identify all data structures used. Each variable is stored on 32 bits.
Memory space used is as follows :
nbr_msg_received : represents the number of messages received.
succ_states : represents states successors.
explored_config: represents the list of explored states stored in the set *known*.

$Data\_used = nbr\_msg\_received * 32 + 2 * 32 + succ\_states * 32 + explored\_config * (32 + 32)$

5.4 Energy consumption study

Computers are becoming more powerful and performing calculations faster and faster. However, this computing power has a cost. Indeed, the increase in the computing power of the machines is accompanied by an increased consumption of electrical energy to deliver this power. For mobile devices, users feel it directly because of the impact on the battery. But this is not the only area where the problem of energy has become crucial. Indeed, supercomputers are composed of several thousands of computing nodes (or processors) and consume so much electrical energy, that it will soon not be possible to feed them correctly. A metric is commonly used to qualify the energy performance of programs is energy. It corresponds roughly to the amount of energy to be supplied to the computer so that it can perform the calculation required. The amount of energy required to perform calculation is determined by the complexity of the work required. In the parallel experiment, the energy is not used at 100%, which means that the power of the processor is not fully used, it depends on the number of configurations. When this number is high, the CPU usage reached 95%. Concerning the distributed experiment, the computing power is used at 100 %. This is due to the communication between both machines (sending and receiving states) and to exploration blocking that is waiting for states when the queue of unexplored states is empty and there are still states to be received. In the parallel approach, configurations to be explored are retrieved progressively, the states are in the machine so that the energy is less important compared to the distributed approach.

6 Related work

Several works have been carried out to treat the state explosion problem. The solutions proposed were executed on different architectures (parallel, distributed, sequential). In this section we cite some parallel and distributed solutions that address this problem.

In [9], the solution is based and extends the solution presented in [10]. Authors consider a distribution of states space exploration. They represented their system with a colored Petri net CPN [8]. The idea is to distribute states among workers. This distribution is based on the introduction of a coordinating process and a number of worker processes. The solution is based on two sets of states : unexplored states and visited states. The first one is the set of states whose successors have not been visited yet. The second one represents the states that have already been explored. The distribution of states can be static or dynamic. The architecture used is based on a distributed system consisting on several machines and a coordinator node whose role is to initiate treatment, to distribute states and to detect the end of exploration. This distribution is achieved through a hash function, that will return the identity of processes that will explore the generated state.

In [7], authors proposed a parallel algorithm to the space explosion problem. The proposed solution consists to fix the number of processes, performing the exploration step, and to use a three-dimensional queue Q[i][j][k] to store unexplored states, i can take 2 values 0 or 1. The variable j, represents the identifier of the process, it varies from 1 to N (N process). k varies from 1 to N. The parameter i allows to pass from the treated states to the future states (states to be explored). At each step, explored states are stored in the queue Q[i][j][k] and their successors are stored in the queue Q[1-i][j][k] with k a randomly selected process identifier to explore successors. A lockless

hashtable is used to avoid waits between processes. Exploration ends when the queue of each process Q[1-i][j][k] is empty, for j and k varying from 1 to N.
In the approach presented in [12], authors presented a parallel solution to the problem posed. The execution steps of the proposed algorithm are as follows:

- take an unvisited state
- Calculate its successors and check each time if they have been visited yet.

This work is recursive. Using a shared memory architecture, the state space is shared by processors and can be accessed through Locks. The algorithm presented is based on a bloom filter (probabilistic data structure) to indicate whether a state has been visited or not, as well as on local data structures. The Bloom filter has been used because it is fast in execution time and compact in memory. The algorithm is divided into three phases, exploration, collision and termination. During the first phase (exploration), the bloom filter is used. This phase allows all states to be processed. During this step, states processed by a processor are stored in both trees AVL structures (Adelson-Velskii and Landis) [1] which are binary trees. The first one is used to store new states and the second one, to store states that have already been visited. This information is given by the bloom filter. Explored states are stored by each processor in an AVL tree because a bloom filter can generate a false result. A collision occurs when a state is assumed to be processed (information given by the bloom filter) while it has not been yet.

7 Conclusion

Model checking is a technique that explores all possible states of a model. It checks if a system respects its specification. In this article, we presented a comparative study between two exploration approaches, the first one is parallel and the second one is distributed. We performed experiments to compare between both algorithms. According to the results in the experiments section, the parallel approach brings better performance in terms of execution and the distributed approach makes it possible to explore larger models which brings a gain in memory space. We are conducting experiments on a cloud computing environment, to evaluate performance on time, and memory space, and to compare the results with the proposed distributed approach.

References

1. G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 45, pages 1259–1263, 1962.
2. L. Allal, G. Belalem, P. Dhaussy, and C. Teodorov. A parallel algorithm for the state space exploration. *Scalable Computing: Practice and Experience*, 17(2):129–142, 2016.
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, UK, 2001. Springer-Verlag.
4. E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *Proceedings of the 8th Laser Summer School on Software Engineering*, volume 7682, pages 1–30, September 2011.

5. J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 2014.
6. 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, pages 715–720, January 2009.
7. G. J. Holzmann. Parallelizing the spin model checker. In *Proceedings of the 19th International Conference on Model Checking Software*, SPIN'12, pages 155–171, Berlin, Heidelberg, 2012. Springer-Verlag.
8. K. Jensen. Coloured petri nets-basic concepts analysis methods and practical use. *Monographs in Theoretical Computer Science. An EATCS Series*, 3, 1994.
9. L. M. Kristensen and L. Petrucci. An approach to distributed space exploration for coloured petri nets*. In *Proceedings of the 25th International Conference on Applications and Theory of Petri Nets*, volume 3099 of *ICATPN '04*, pages 474–483, June 2004.
10. 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*, volume 1680, pages 22–39, July 1999.
11. R. Pelánek. Fighting state space explosion: Review and evaluation. In *Proceedings of the 13th on Formal Methods for Industrial Critical Systems*, volume 5596 of *FMICS '08*, pages 37–52, September 2008.
12. R. T. Saad, S. D. Zilio, and B. Berthomieu. A general lock free algorithm for parallel state space construction. In *Proceedings of the 9th International Workshop on Parallel and Distributed Methods in Verification*, PDMC-HIBI '10, pages 8–16, October 2010.
13. H. Subramoni, F. Petrini, V. Agarwal, and D. Pasetto. Intra-socket and inter-socket communication in multi-core systems. *Computer Architecture Letters*, 9(1):13–16, 2010.