Enumerative Parallel and Distributed State Space Construction

Rodrigo T. Saad, Silvano Dal Zilio, Bernard Berthomieu, François Vernadat
CNRS; LAAS;
7, avenue du Colonel Roche, F-31077 Toulouse – France
Université de Toulouse;
UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
{rsaad, dalzilio, bernard, francois}@laas.fr

Abstract

Model Checking requires high end computers to verify complex systems. Consequently, it is interesting to use a multi-processors architectures in order to have more computational resources available to deal with bigger models. This work presents a survey of parallel and distributed state space construction for Model Checking purpose.

I. Introduction

Formal verification is one of the main approaches to help engineers on the development of concurrent systems. A formal verification technique is based on mathematical formal methods able to provide formal proofs that a system has the desired properties, in agreement with the supplied models. Among these techniques, this work presents an overview of parallel and distributed enumerative state space construction for Model Checking purpose. Model Checking is an automatic technique for verifying finite state systems that suffers from the state space explosion, which is when the number of reachable states grows exponentially with the number of the system's components.

Although the state space construction is considered as a problem hard to perform in parallel, it is interesting, from the point of view of time saving and memory expansion, to use a multi-processors environment in order to have more computational resources available to deal with bigger models. We consider two different kind of such architecture based on the memory model: distributed and shared memory. On the one hand, distributed memory machines may offers more memory space allowing the construction of bigger models. Solutions based on this kind of architecture are more common in the literature maybe because of two reasons; first they are easy to scale and second due its lower price. On the other hand, shared memory machines may result in a faster execution time but there are not too many solutions in literature up to now, maybe because of the price and the complexity associated with this architecture, i.e. data race, false sharing, etc.

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

The rest of this paper is organized as follows. Section II presents an introduction about parallel distributed architecture. Then, in Section III, we briefly introduce model checking and the problematic surrounding parallel and distributed state space construction. Section IV presents solutions proposed up to know for both architectures. Finally, Section V summarizes this paper.

II. Parallel and Distributed Architecture

A parallel processor is defined as a computer system consisting of multiple processing units connected via some interconnection network plus the software to make the processing units work together [4]. The interconnection network is mainly of two types:

- shared memory: coordination among processes is accomplished through global memory shared among all processes. If all processes access the memory in the same way, its considered a UMA(Uniform Memory Access) architecture. In the case where each processor has part of the memory attached and the access time depends on the distance to the processor owner, it is classified as NUMA(Non-Uniform Memory Access). Finally, the COMA(Cache-Only Memory Architecture) in the case where the shared memory consists of cache memory;
- message passing: there is no shared memory space among the processors and communication is provided through exchange of messages instead of memory access. Depending on the provided intercommunication bus, they can be classified as static or dynamic.

III. Model Checking and State Space Construction

Model Checking [3] is an autonomous formal technique used to inquire if a given model meets its specifications. This technique is based on mathematical structures to formally describe the system model and logic formulas to represent the desired properties, which are obtained from the specifications. Moreover, a finite state space representation is extracted from the mathematical structure by exploring until saturation all possible states. Finally, the finite state space is checked weather it satisfies the given logic formulas. .

For the finite state space representation, two graphs are considered: symbolic and enumerative. Enumerative graph is when the states are represented in extension, by enumerating all reachable states. By contrast, symbolic graph is the one where the state space is represented in a compressed way, by using several encoding techniques. In our case, we consider only enumerative graphs because they are more suitable to parallelize than symbolic graph, since symbolic graphs would also require parallel state encoding techniques.

A. Irregular Problem

Finite state space construction can be classified as an irregular problem in the parallel algorithms community because of the irregularity of its structure, in other words, the cost to operate this kind of structure is not exactly know or is unknown by advance. As a consequence, the parallel execution of such problems may result in a bad load balance among the processors [7, 5].

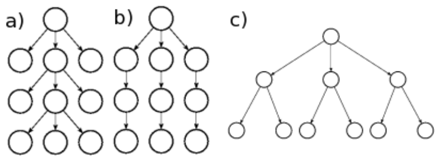

Figure 1. Three pathological models [5].

In [5], the authors explain that the characteristics of the model under consideration has a key influence on the performance of a parallel algorithm because it may result in extra overhead during the exploration task. Figure 1 shows three pathological models. Figure a) shows a model where the parallel exploration will perform like a sequential one, incapable of speedups. Figure b) illustrates a model that imposes high scheduling overheads, due to the small size of the work units. Figure c) is the ideal model where (almost) every node has more than one successor, minimizing the scheduling overhead.

B. Sequential State Space Construction

The Sequential State Space Construction is the simple and well-know reachability graph algorithm. It starts from the initial state ($S := Initial$;) by exploring until saturation (*for each a* $\in$ *Enabled(s) do*) all possible successor states ($s_{new} := NewState(s, a)$). Every new state found (*if* $s_{new} \notin S$) is stored in the state graph ($S := S\cup \{n_{new}\}$) with their input arcs ($s \rightarrow^{a} s_{new}$). The sequential algorithm is shown in Figure 2.

```
$S := \{Initial\}$; $S_{new} := \{Initial\}$; $A := \emptyset$;
**while** $S_{new} \neq \emptyset$ **do**
  $S_{new} := S_{new} \setminus s$;
  **for each** $a \in Enabled(s)$ **do**
    $s_{new} := NewState(s, e)$;
    **if** $s_{new} \notin S$ **then**
      $S_{new} := S_{new} \cup \{s_{new}\}$;
      $S := S \cup \{s_{new}\}$;
    **end if**;
    $A := A \cup \{s \rightarrow^{a} s_{new}\}$;
  **end for**
**end while**;
```

Figure 2. Sequential State Space Construction algorithm.

IV. Parallel and Distributed State Space Construction

Parallel and Distributed state space construction have been studied in various context and different solutions have been proposed. A majority of these solutions adopt a common approach: they can be considered as an “homogeneous” parallelism and they follow an SPMD (Single Program Multiple Data) programming style. The SPMD approach is commonly used to accomplish coarse-grained parallelism and it requires an explicit data and work assignation by the programmer to each processor. Another characteristic about SPMD programs is that the work executed by the threads is typically “homogeneous” in the sense that all threads performs concurrently the same steps, which is in this case a similar algorithm to the one presented at Figure 2.

A common approach to assign work and data is to partition the state space into several chunks, one for each processor available, through a slicing function. This solution is more common on distributed memory environments; they all follow almost the same architecture but differ by the nature of the slicing function, i. e. static or dynamic. In contrasts, solutions based on shared memory architectures are more concerned with synchronizations among processors, memory consistency and overheads caused by the extended used of locking systems (mutual exclusion for memory access).

A survey of solutions proposed up to now are presented below. The solutions presented on the following survey were selected because of the theoretical results they propose, hence, papers describing implementation of model checking tools is out of the scope of this overview.

A. Distributed Memory

A great deal of solutions proposed up to now are based on a partition function, i.e. $Proc : S \Rightarrow \{0, ..., N-1\}$ where $Proc(s)$ is the owner of state s among N processors. They differ basically by the nature of this function in order to provide both locality and balance. Locality is important to reduce the communication overhead caused when successors are assigned to a different processor. Balance can be measured as spatial or temporal balance. Spatial balance means that each processor will receive an equal amount of states. Temporal balance means that each processor will be busy most of the time.

Some partition functions use heuristic analysis based on the model structure in order to achieve both locality and balance [2, 12]. A partition function is proposed in [2] based on the structure of the modeling formalism SPN (Stochastic Petri Nets). They propose a hashing function taking into account a small set of places called control set (*P*). The idea is that any transition firing that does not involve a place in *P* corresponds to a state transition between two markings assigned to the same processor, otherwise this transition will correspond to a cross-arc and the new marking will be assigned to another processor. The disadvantage of this approach is that there is no automatic way to suggest this control set, therefore the technique relies on the user intuition to select the places that are part of the control set. Later, [12] suggested a heuristic to select the control set based on the notion of conflicting transitions. Thus, the data holders associated to the conflicting transitions are chosen as the control set. This approach expects to achieve both locality and balance because the successors states computed from conflicting transition can be exported to different partitions. Another work that extends [2] is [1] by introducing the dynamic load balancing strategy. This strategy consists in monitoring the average memory utilization of all processors and if one of them differs more than a fixed percentage, the dynamic loading balance phase starts. The loading balance phase estimates how much memory each processor has to give/receive to its neighbors. In the end, a new partition function is created and broadcasted to all the processors. The main disadvantage of this approach is that all processes are blocked during the dynamic loading balance phases.

In contrast, other solutions rely on previous analyzes for extracting relevant information to slice the state graph. In [10], the authors try to minimize the cross-arcs by computing a small approximation of the state space by an abstract interpretation of the system specification. From this prediction, it is possible to extract the shape of the original system and the relations among the states.

Finally, there is a group of solutions that do not take into account any structural information from the model and are based only in a mathematical function to partition the graph. The solution presented in [6] divide the graph trough a hash function taking into account only the balance requirement. As a consequence, their solution does not handle cross-arcs and suffers from communication overhead. The results reported in this work present distributions with a standard deviation smaller than 1% of the mean value. However, these results are highly influenced by the employed static function and the set of chosen examples. The same approach is followed in [11] to construct the state graph with a slight difference for the partition function. It assumes that several states have more than one successor and allocates the excess to a different processor. The authors show that the addition of this technique results in a better balance work load. However, not only this technique cannot handle cross-arcs (as [6]) but it also duplicates states.

The anatomy of a basic distributed algorithm is presented below[8]:

- all processors start their exploration program;
- processor i, for which $i := Z(initial)$, starts to explore successor states;
- upon generation a state $s\prime$ from s:
  - the allocation for $s\prime$ is computed: $j := Z(s\prime)$
  - if $j = i$ then state $s\prime$ is handled locally;
  - if $j \neq i$ then $s\prime$ and $(s \rightarrow^{a\prime})$ are sent to proc. j;
  - all processors process the states received from others, as well as those generated locally;
- the algorithms terminates when all process has no more states to be explored.

The sequential algorithm (Figure 2) can be extended from the considerations above to handle the state space partition promoted by the slicing function.

B. Shared Memory

The main advantage shared memory architecture offers over distributed memory is that it provide a shareable memory space for concurrent manipulation, thus obviating the need of passing messages among the processors. As a consequence, there is no more need for a slicing function to partition the state space because the storage structure is shared among the processors. However, it imposes other difficulties related to data consistency and synchronization operations to manipulate the shared data. Data consistency is mandatory to assure that a certain processor is accessing the most recent update of the global data. Consequently, synchronization techniques to guarantee mutual exclusive access must be implemented. Nonetheless, to achieve high degree of parallelism, the share of global data that is locked for mutual exclusion must be kept small because synchronization operations are usually time-consuming.

The work presented in [1] was one of the first to implement a parallel state space construction on a shared memory machine. They solve the consistency problem by using locking variables to protect the data structures. In this work, the authors make use of a Balanced-tree as search structure with a method called splitting-in-advance to reduce the number of data locking, allowing a better concurrent access. The major problem working with structures like Balanced-tree is when an insertion happens into a full node, which forces the node to split into two parts. One of the keys is sent to the parent node, which may also be split. This propagation may occur back to the root. In conclusion, a common Balanced-tree would force the use of several locking variables.

Accordingly, the splitting-in-advance method consists in splitting immediately each full node while crossing the Balanced-tree on the way down, regardless of whether an insertion will take place or not. Since non-full nodes serve as barrier, back propagation does not occur because parent nodes can never be full. The result is that each processor holds at most one lock at time. In addition, locking variables are also used to protect the shared stack responsible of load balance, following a *work sharing* scheduling paradigm where new work is distributed to underutilized processors. The examples in [1] report speedups close to linear.

Later, [9] proposed a parallel algorithm for state exploration based on the *work stealing* scheduling paradigm to maintain a dynamic load balance without a blocking phase. The base concept behind this paradigm is that underutilized processors attempt to “steal” work from other processors. In this work, the work-stealing paradigm was implemented by using a two-queue structure per processor and a hash table to store visited states. The two-queue structure consists in a private and a shared queue that are used to store unexpanded states. Every time a process has no more unexpanded states in its private queue, it has to acquire the mutual exclusion lock to check over its own shared queue for a state. If no state is found, the processor starts searching through all other shared queues until it finds a nonempty queue or finds that all shared queues are empty. With reference to the storage data, unlike [1], this work implements a hash table without any mutual exclusions locks to synchronize access. The authors emphasize that the duplication caused by the lack of a locking strategy is not relevant compared to the parallel computation power available. Finally, the results reported show that efficiency of the work stealing load balance strategy depends on the division of the state graph (number of successors) and the size of the shared queue. From the experiments presented, optimum results are achieved when the shared queue size of each processor are equal to the branching degree of the graph or one more.

V. Conclusion

Parallel state space construction is a problem hard to parallelize due to its irregular structure. Another important issue regarding its parallel implementation is the choice of parallel architecture. For distributed memory computers, the solutions are more concerned with partitioning the space state among processors. Regarding the solutions proposed for shared memory computers, they are more oriented towards finding the least restrictive way to share the state space. Moreover, as can be noticed there are more solutions reported for distributed than shared memory machines. This disparity can be explained in part by the higher cost (in the past) of high end shared computers. To conclude, we strongly believe that new solutions for shared computers will be proposed because this kind of machine is becoming more affordable each year.

References

[1] S.C. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of gspns on a shared-memory multiprocessor. *Petri Nets and Performance Models, 1997., Proceedings of the Seventh International Workshop on*, pages 112–121, Jun 1997.

[2] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS J. on Computing*, 10(1):82–93, 1998.

[3] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. Springer, 1999.

[4] H. El-Rewini and M. Abd-El-Barr. *Advanced computer architecture and parallel processing*. Wiley-Interscience, 2005.

[5] Jonathan Ezekiel and Gerald Lüttgen. Measuring and evaluating parallel state-space exploration algorithms. *Electronic Notes in Theoretical Computer Science*, 198:47 – 61, 2008. Proceedings of the 6th International Workshop on Parallel and Distributed Methods in verifiCation.

[6] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. Volume 2057/2001:217–234, Jan 2001.

[7] T. Gautier, J. L. Roch, and G. Villard. Regular versus irregular problems and algorithms. In *In Proc. of IRREGULAR'95*, pages 1–25. Springer, 1995.

[8] B. Haverkort, A. Bell, and H. Bohnenkamp. On the efficient sequential and distributed generation of very large markov chains from stochastic petri nets. *Petri Nets and Performance Models*, 0:12, 1999.

[9] Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. *Electronic Notes in Theoretical Computer Science*, 68(4):605 – 620, 2002. Parallel and Distributed Model Checking.

[10] S. Orzan, J. Van de Pol, and M. V. Espada. A state space distribution policy based on abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 128(3):35 – 45, 2005. Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification.

[11] D. Petcu. Parallel explicit state reachability analysis and state space construction. *Parallel and Distributed Computing, 2003. Proceedings. Second International Symposium on*, pages 207–214, Oct. 2003.

[12] C.L. Rodrigues, P.E.S. Barbosa, J.M. Cabral, J.C.A. de Figueiredo, and D.D.S. Guerrero. A bag-of-tasks approach for state space exploration using computational grids. *Software Engineering and Formal Methods, 2006.*, pages 226–235, Sept. 2006.