Parallel and Distributed Computing in Model Checking

Diana Dubu* Dana Petcu*,**

* Department of Computer Science, Western University of Timişoara,
** Institute e-Austria Timişoara
{ddubu,petcu}@info.uvt.ro

Abstract

The growing importance of model-checking in industry is driving a growing interest in those aspects of model-checking which have a direct impact on its applicability to real world problems. One of the main technical challenges is in devising tools that allow to handle large state spaces: when model-checking realistic systems, one usually must deal with the facts that the size of the state space of the system being modeled is prohibitive, and the time required to check even simple logical properties is very large. Recently, an increasing interest is in parallelizing and distributing of model-checking techniques. In this report we investigate several known approaches to parallelize state space construction allowing to reduce both the amount of memory needed and the overall execution time.

As formal verification becomes increasingly used in the industry as a part of the design process, there is a constant need for efficient tool support to deal with real-size applications. Model-checking [12] is a successful verification method based on reachability analysis (state space exploration) and allows an automatic detection of early design errors in finite-state systems. Model-checking works by constructing a model (state space) of the system under design, on which the desired correctness properties are verified. There are essentially two approaches to model checking: symbolic verification [8] represents the state space in comprehension, by using various encoding techniques (e.g., BDDs), and enumerative verification [13] represents the state space in extension, by enumerating all reachable states. Enumerative model-checking techniques can be further divided in global techniques, which require to entirely construct the state space before performing the verification, and local (or on-the-fly) techniques, which allow to construct the state space simultaneously with the verification. There are four problems that are currently preventing model-checking technology from being successfully applied to software: the state explosion problem (the exponential increase in the size of a finite-state model as the number of system components grows), the model construction problem, the requirement specification problem, and the output interpretation problem.

More memory and speed-up in model-checking The main aim in exploiting a parallel or distributed environment for model-checking is to extend the applicability of model-checking algorithms to larger and more complex systems. A parallel super-computer, grid or a network of computers can provide extra resources needed to fight more realistic verification problems. Parallel processing can help speed-up model-checking by providing many CPUs that can work on the problem, providing higher aggregate memory capacity, as well as memory bandwidth. While the problem remains exponential, the overall performance can go up by significant factors.

Often, developers have access to large parallel computers, but cannot make full use of them because most model-checkers are designed for single processor systems. Successful implementation on large parallel architectures, however, offers access to significantly more fast, local memory and potential for speedup – a three or fourfold speedup may not appear too significant, however, waiting a day is far preferable than waiting for four to discover a bug.

A natural idea is that to exploit the fact that each studied property can be checked in a separate and independent verification run. It is easy to see that large gains in performance can be made if these verification jobs are parallelized. Moreover for the verification of individual properties, all levels of iteration can be run in parallel. The verification process on all nodes is then stopped as soon as at least one of the nodes successfully generates a counterexample to the given property. Using this type of parallel, iterative, search process, near interactive performance can be achieved on verifications runs, returning results to the programmer within minutes after a revision is completed [14].

The important feature of algorithms running in a distributed environment is to solve the given task by distributing the data among the participating workstations with as small amount of coordination as possible. One of the main issues in distributing model-checking algorithms is how to partition the state space (data) among the individual computers called here network nodes. The main idea of many distributed algorithms is similar: the state graph is partitioned among the network nodes, i.e., each network node owns a subset of the state space. The differences are in the way the state space is partitioned (partition or slicing function). The model-checking algorithm running on each network node has thus access only to a part of the entire system. Depending on the type of the algorithm it communicates with other nodes to achieve the required (global) result. There are two aspects that significantly influence the overall effectiveness of model-checking in the distributed environment: locality and (spatial) balance of the state space partition. Locality means that most of the state's descendants are assigned to the same node as the parent state, thus reducing communication and cooperation overhead. Balance means that each network node is assigned approximately the same number of states, thus achieving good speed-up.

A such distributed state space construction has been studied in various contexts. All these approaches differ on a number of design principles and implementation choices such as: the choice between a shared memory architecture and a message-passing one, the use of hash tables or B-trees to store states on each machine, the way of partitioning the state space using either static hash functions or dynamic ones that allow dynamic load balancing, etc.

The main problem is the load balance between different processors involved in the model-checking procedure. There are several attempts to overcome this problem: different slicing techniques have been employed [10], [18] [23], but in most cases the slicing algorithm is static, i.e., the slicing depends on the state and not on the distribution of the workload.

The Murϕ model-checker was parallelized [23] by partitioning the visited states across the processes using randomized load balancing; i.e., the owner process of a state is calculated with a hash function (a function which, given a state, returns a numerical value between two predetermined boundaries). The algorithm achieves near linear speedups. Paper [18] reports an implementation of a distributed version of SPIN for checking safety properties: the random hash function is replaced with a slicing function that tries to minimize cross-transitions (transitions between states belonging to different processes) by using the structure of the states; the algorithm only showed speedups for state graphs that were too big to be explored on a single processor and then started swapping. In [3] the hash function technique of [23] is used in the distributed implementation of the symbolic model-checker UPPAAL: a heuristic is added for ordering the states at each processor so that the distributed search is as close as possible to a breadth-first search, which in some cases resulted in a super linear speedup. Paper [10] reports an implementation of a distributed state space construction algorithm for the CADP verification tool set: the slicing algorithm partitions states independent of their structural properties and therefore independent of the specification language used.

One model-checking algorithm that implements a dynamic slicing algorithm is the distributed symbolic algorithm from [11]. The memory balance is maintained by repartitioning the state space whenever the memory becomes unbalanced. The drawback of this approach is that while reslicing for particular processors is executed those processors can not continue with state exploration.

In [17] an approach to model-checking of software is developed which uses modularity. A module is a subset of a state space that originates from splitting the whole system in an additive way. It is defined by following the syntactical structure of the program. Besides this partition, it is also defined the notion of an assumption function that represents partial knowledge about truth of formulas provided by the rest of the system (by other parts). In [7], the problem of performing distributed computational tree logic model-checking is treated by splitting the given state space into several "partial state spaces". Each computer involved in the distributed computation owns a partial state space and performs a model-checking algorithm on this incomplete structure. To be able to proceed, the "border" states are augmented by assumptions about the truth of formulas and the computers exchange assumptions about relevant states as they compute more precise information. States that represent "border" states are those states that in fact belong to some other network node. Whenever the model-checking algorithm reaches a border state it uses information provided by other network nodes about the truth of formulas in that state – assumptions. As the assumptions can change, a re-computation is necessary in general. Paper [4] presents a scalable method for parallel symbolic on-the-fly model-checking on a distributed-memory environment of workstations; the method combines a parallel version of an on-the-fly model-checker for safety properties with a scalable scheme for reachability analysis and is implemented on a standard, loosely-connected environment of workstations, using a high-performance SMV-based model-checker.

Model checking on parallel and distributed codes Modern computing applications increasingly require concurrent/distributed software systems that are extremely reliable. Formal methods allow specifications of parallel and distributed programs to be precisely stated and the conformance of an implementation to be verified using mathematical techniques. These methods are especially important in parallel and distributed programming where the inherent non-determinism makes testing ineffective.Unfortunately, current software validation techniques, such as inspections and testing, are failing to provide high levels of assurance of correctness for these systems due to system size and complexity as well as the fundamental difficulties of reasoning about state/event sequences in concurrent behavior.

Among the scores of methods used to reduce state-explosion, a prominent method - partial order reduction - exploits the fact that whenever a set of independent actions arise during exploration, they need not be explored in more than one linear order. Given the widely recognized importance of partial order reduction, efficient methods to realize it in a parallel context can be explored. The earliest identifiable connection between parallel processing and the central idea behind partial order reduction appears in Liptons work of 1975 on optimizing P/V programs [19] for the purpose of efficient reasoning: actions are identified that can be postponed without affecting the correctness of reasoning. Additionally, it has been observed [22] that by identifying computations that "commute", one can schedule these computations for parallel evaluation without interference. Despite the above connections, if a partial order reduction algorithm involves a sequentializing step, it can significantly reduce available parallelism. Such a sequentializing step is present in current partial order reduction algorithms such as the one used in SPIN [13] (the most widely distributed software package that supports the formal verification of distributed systems). Paper [21] shows how to combine partial order reduction and parallel distributed model checking: the partial order reduction algorithm "Twophase" implemented in SPIN-like model-checker PV, extends naturally to a parallel context.

A parallel state-space exploration algorithm from a non model-checking context was developed in [1] for stochastic modelling on a shared memory architecture: the implementation uses a shared stack for dynamic load balancing and a modified B-tree for storing visited states. In [15] a parallel state exploration algorithm using a two-queue structure for load balancing is proposed. The algorithm is implemented as a parallel composition of several state-explorer processes where each process has both a private and shared queue. Load balancing is maintained by work stealing and together with the two-queue structure for storing unexpanded states it allows for implicit and dynamic load balancing with minimal synchronisation. Visited states are maintained in a global hash table, not distributed over the processes - contention is minimised by removal of all software synchronisation locks at a small cost of possible redundant work. Paper [9] describes a reduction technique which is very useful against the state explosion problem which occurs when model checking many distributed systems.

Paper [20] proposed an algorithm for livelock analysis where cycles are detected by pruning the graph: nodes with no outgoing arcs are deleted until no such nodes remain; the algorithm has not been implemented and has the drawback that the entire graph has to be constructed before pruning starts. The algorithm from [18] is extended in [2] to allow liveness checking: the algorithm performs a distributed depth-first search (DFS), but whenever an accepting state is reached, it is stored in a special data structure so that a nested DFS can be performed from that state. All states waiting for their nested DFS are sent to a manager process where a global queue of accepting states is kept.

The manager then ensures that only one nested DFS is executed at any time. This algorithm allows model checking for larger state spaces than the sequential version. In [6] the nested DFS algorithm is replaced with a negative cycle detection algorithm developed to solve the single-source shortest path problem; [5] tried to overcome the costliness of parallel cycle detection by developing an algorithm without cycle detection.

Future work We intend to create our own software for distributed state space construction using the computational power of a PC Myrinet cluster. As a first step, we will implement the parallel algorithm proposed in [10], using, instead sockets, the message passing interface MPI. The advantage of using MPI is the possibility to extend (as the second step) the code in a more general environment, on Grids, using Globus technologies which allow MPI programs to be remotely executed. A similar technique to that proposed [7] will be used to verify various properties of the tested software. We intend to verify at the beginning small Java programs for which we can manually generate a model.

References

[1] S.C. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. *LNCS* 1253, Springer, 1997.

[2] J. Barnat, L. Brim. Distributed LTL model-checking in SPIN. Proc. 8th Int.SPIN Workshop on Model Checking of Software, *LNCS* 2057, Springer, 2001.

[3] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model-checking - how the search order matters. In Proc.12th Int.Conf.Computer Aided Verification, *LNCS* 1855, 216-231. Springer, 2000.

[4] S.Ben-David, T.Heyman, O.Grumberg, A.Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. Proc. FMCAD00.

[5] B. Bollig, M. Leucker, and M. Weber. Local parallel model-checking for the alternation-free mu-calculus. Proc. 9th Int. SPIN Workshop on Model Checking of Software, *LNCS*. Springer, 2002.

[6] L. Brim, I. Cerna, P. Krcal, and R. Pelanek. Distributed LTL model-checking based on negative cycle detection. *LNCS* 2245, Springer, 2001.

[7] L.Brim, J.Crhova, K.Yorav. Using Assumptions to Distribute CTL Model Checking. Parallel and Distributed Model Checking (PDMC'2002), *Electronic Notes in Theoretical Computer Science* 68(4), 2002

[8] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a New Symbolic Model Checker. *Int.J. Software Tools for Technology Transfer* (STTT), 2(4): 410-425, 2000.

[9] F.Derepar, P.Gastin, D.Plainfosse. Avoiding State Explosion for Distributed Systems with Timestaps. *LNCS* 2021. Proc. 10th Symp.Formal Methods for Increasing Software Productivity (FME'01), 2001.

[10] H.Garavel,R.Mateescu,I.Smarandache.Parallel state space construction for model-check.*LNCS* 2057,2001.

[11] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. Proc. 12th Int. Conf. on Computer Aided Verification, *LNCS* 1855. Springer, 2000.

[12] G.J. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.

[13] G. Holzmann. The Model Checker SPIN. *IEEE Trans.on Software Engineering*, 23(5):279-295, 1997.

[14] G.J. Holzmann and M.H. Smith. Software model-checking: extracting verification models from source code. *Softw. Test. Verif. Reliab.* 11:6579, 2001.

[15] C.P. Inggs and H.Barringer, Effective State Exploration for Model Checking on a Shared Memory Architecture Electronic, PDMC'2002, Brno, *Electronic Notes in Theoretical Computer Science* 68(4), 2002

[16] International Workshop on Formal Methods for Parallel Programming: Theory and Applications, FMPPTA'2001: April 27, 2001, San Francisco; FMPPTA'2002: April 19, 2002, Fort Lauderdale.

[17] K.Laster,O.Grumberg. Modular model-checking of software. Proc. TACAS'98, *LNCS* 1384, 20-35, 1998.

[18] F.Lerda,R.Sisto. Distributed-memory model-checking with SPIN. Proc.5th Int. SPIN Workshop, 1999.

[19] R.J. Lipton. Reduction: A method of proving properties of parallel programs. *Comm.ACM*, 18(12), 1975.

[20] J. M. R. Martin and Y. Huddart. Parallel algorithms for deadlock and livelock analysis of concurrent systems. *Communicating Process Architectures*, 1-14. IOS Press, 2000.

[21] R.Palmer and G.Gopalakrishnan. Partial Order Reduction Assisted Parallel Model-Checking. PDMC'2002, Brno, *Electronic Notes in Theoretical Computer Science* 68(4), 2002.

[22] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 19(6):147, 1997.

[23] U. Stern and D. Dill. Parallelizing the Murϕ verifier. Proc. 9th Int.Conf. on Computer Aided Verification. *LNCS* 1254. Springer, 1997.