Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 107–123
www.elsevier.com/locate/entcs

CTL* Model Checking on a Shared-Memory Architecture

Cornelia P. Inggs[1,2] and Howard Barringer[3]

Department of Computer Science
University of Manchester
UK

Abstract

In this paper we present a parallel algorithm for CTL* model checking on a virtual shared-memory high-performance parallel machine architecture. The algorithm is automata-driven and follows a games approach where one player wins if the automaton is empty while the other player wins if the automaton is nonempty. We show how this game can be played in parallel using a dynamic load balancing technique to divide the work across the processors. The practicality and effective speedup of the algorithm is illustrated by performance graphs.

Keywords: Model Checking, Shared-Memory, Parallelisation, Automata, Game Theory

1 Introduction

Model checking is an established technology for automated verification of designs, now adopted by industry to check correctness properties of many critical systems. The tremendous advances that have been made over the past decade in developing specialised state encodings and algorithms to reduce the burden of the state explosion problem, inherent with this style of verification, have been paramount to this industrial take-up. Even so, the size and complexity of

[1] This work was fully supported under a Universities UK ORS award, a University of Manchester Department of Computer Science Scholarship, and a South African Harry Crossley Bursary.
[2] inggscp@telkom.co.za
[3] howard@cs.man.ac.uk

doi:10.1016/j.entcs.2004.10.022

systems that can be verified is still heavily constrained by time and available memory, and the development of techniques to alleviate the state explosion problem remains an active area of research. One technique that has gained significant interest recently is the parallelisation of model checking.

There were a few isolated publications on the parallelisation of model checkers in the 1980s and 1990s and then Stern and Dill's seminal paper for parallel reachability analysis appeared in 1997 [17]. In the past three to four years parallel model checking has gained considerable interest.

Much of the extant research has focused on implementations over distributed networks and the development of static partioning functions. Static partitioning functions depend on the state and not on the distribution of the workload. To the best of our knowledge the only algorithms that use a dynamic partitioning function are the symbolic algorithm of Heyman et al. [12] and the two algorithms based on Heyman et al.'s article [3,11]. In these algorithms the memory balance is maintained by repartitioning the state space whenever the memory becomes unbalanced. Initially only safety checking was parallelised, but in the last few years the developement of parallel algorithms for liveness checking increased and algorithms for checking LTL [2,6,16,15], CTL [7], and the μ-calculus [5] have been developed. See [13] for a full bibliography. The development of efficient parallel algorithms for liveness checking have been less successful than for safety checking and very few parallel algorithms for checking both liveness and safety properties achieve speedups.

We explored the parallelisation of model checking for shared-memory multiprocessor computers to evaluate its feasibility and identify any inherent difficulties or pitfalls when parallelising model checking for shared memory architectures. In particular, the parallelisation of explicit-state on-the-fly model checking was investigated for both safety and liveness properties and led to the development of a parallel model checker for CTL*. This research has shown the practicality and effective speedup of model checking using a shared-memory architecture. The performance of the parallel algorithm was evaluated via theoretical analysis and also via experimental analysis using a number of prototypical models, including correctness properties of the parallel model checker itself.

In our earlier paper, [14], we proposed a parallel algorithm for reachability analaysis on a shared-memory architecture. In this paper we present a parallel model checking algorithm for CTL* that uses the dynamic load balancing technique of the parallel reachability analysis algorithm described in [14]. An overview of the parallel reachability analysis algorithm is given in the next section. This is followed by a description of the serial automata-driven game-theoretic algorithm for CTL* and its parallelisation in Sections 3 and 4.

The performance of the algorithm is analysed and discussed in Section 5 and conclusions are presented in Section 6.

2 Parallel Reachability Analysis

Our parallel reachability analysis algorithm is executed on N processes where each process (thread of control) runs on one physical processor. All N processes share one *store* for storing visited states and each process has its own unbounded *private stack* and bounded *shared stack* for storing unexpanded states. A process can add states only to its own private and shared stacks, but when its own private and shared stacks are empty it can steal a state, i.e., remove a state from the shared stack of another process.

```
1  Procedure ParallelReach(initState)
2     start parallel
3        if (procid = 0) then next := initState;
4        else next := EMPTY; endif;
5        do
6           if (next = EMPTY) then next := PopFromStack(); endif;
7           if (next ≠ EMPTY) then
8              ComputeSuccessors(next);
9           else
10             CheckForTermination(procid);
11          endif;
12       while (¬terminate);
13    end parallel;
14 endprocedure
```

Fig. 1. Pseudocode for a parallel reachability analysis algorithm

The pseudocode for the parallel reachability analysis is given in Fig. 1. Each participating process executes a copy of the code between the start parallel and end parallel directives. Termination is detected by the low overhead token-based algorithm of Dijkstra et al. and the pseudocode for **ComputeSuccessors**, **PopFromStack** and **PushOnStack** is given in the Appendix. Each shared stack has a lock to synchronise read and write access to it, but the store, a hash table, has no mutual exclusion locks to synchronise access. This can result in duplicate work when more than one process creates the same state, but with significantly more parallel computation available than naturally parallel tasks, performing redundant work is not a significant overhead. It was found, however, that very little work is duplicated. For example, when the model of the parallel model checking algorithm described in this paper

was checked for freedom of deadlock on eight processors, just over 7.8 million states were visited and only six states were duplicated.

3 AltMC: A Serial Model Checker based on Alternating Automata and Game Theory

The model checking problem can be stated as follows: given a Kripke structure K and temporal logic formula ϕ determine if $K \models \phi$. Formally, a Kripke structure is a four-tuple $K = (S, R, s_0, L)$ where S is a finite set of states, $R \subseteq S \times S$ is a transition relation that must be total (for every $s_i \in S$ there exists at least one s_j such that $(s_i, s_j) \in R$), s_0 is an initial state, and $L : S \mapsto 2^P$ maps each state to a set of atomic propositions true in that state. Several approaches have been developed to solve the model checking problem. The algorithm described in this article follows the automata-driven approach, which is based on the principles that a formula ϕ can be translated to an automaton A_ϕ that accepts the set of models for ϕ, and that the Kripke structure K can be seen as an automaton. The model checker then constructs the product automaton $A_{K,\phi} = K \times A_\phi$ and if the language accepted by $A_{K,\phi}$ is nonempty, ϕ holds for K, otherwise not [18,4]. In our context we use Hesitant Alternating Automata [4] to represent formulae specified in the branching time temporal logic CTL*, and formulate the construction of the product automaton A_ϕ and its nonemptiness check in terms of a game, called the *nonemptiness game* [19].

As a brief introduction, automata over infinite trees (tree automata) run over leafless Σ-labelled trees, where Σ is a finite alphabet. A run r of an alternating automaton A on a tree T is a tree where the root is labelled by (s_0, q_0) and every other node is labelled by an element of $(\mathbb{N}^* \times S)$ [4]. Each node of r corresponds to a node of T. A node (x, q) in r corresponds to the automaton in state q reading node x in T. Note that many nodes in r can correspond to the same node in T. The labels of a node and its successors have to satisfy the transition function.

A run r is accepting if all its infinite paths satisfy the acceptance condition. Note that we can get finite branches in the tree representing the run when either true or false is read in the transition function. In an accepting run only true can be found at the end of a finite branch. Different types of alternating automata have different acceptance conditions. In Hesitant Alternating Automata (HAAs) the acceptance condition is a pair of states (G, B) of which the satisfaction depends on the following restriction on the transition

[4] A tree, here, is modelled as a subset of $\mathbb{N}^*$, where each sequence of $\mathbb{N}$ uniquely identifies a node of the tree by its pathname.

Brandy	Port
Play reaches a false	Play reaches a true
After a move by Port that revisits a position in the current play and $\inf(\text{play}) \cap G = \emptyset$	After a move by Port that revisits a position in the current play and $\inf(\text{play}) \cap G \neq \emptyset$
After a move by Brandy that revisits a position in the current play and $\inf(\text{play}) \cap B \neq \emptyset$	After a move by Brandy that revisits a position in the current play and $\inf(\text{play}) \cap B = \emptyset$

Table 1
Winning conditions for a play in the non emptiness game

structure of an HAA: the sets of an HAA can be partitioned into disjoint sets S_i and there exists a partial order $\leq$ between the sets. The sets can further be classified as transient, existential, or universal, such that for each S_i, and for all $s \in S_i, a \in \Sigma$, and $k \in \mathbb{N}$ the following holds: (1) if S_i is transient, then $\delta(s, a, k)$ (δ is the transition function of the HAA) contains no elements from S_i; (2) if S_i is existential, then $\delta(s, a, k)$ contains only disjunctively related elements of S_i; and (3) if S_i is universal, then $\delta(s, a, k)$ contains only conjunctively related elements of S_i. From this restricted structure of HAA it follows that every infinite path, π, will get trapped either in an existential or universal set, S_i. The path then satisfies (G, B) if and only if either S_i is existential and $\inf(\pi) \cap G \neq \emptyset$ or S_i is universal and $\inf(\pi) \cap B = \emptyset$, ($\inf(\pi)$ is the set of states infinitely repeated on path π).

Checking the nonemptiness of the product of the HAA ϕ and Kripke structure K can be defined as a two-player game in which player 1 (Brandy) tries to show that the alternating automaton is empty whilst player 2 (Port) tries to establish that it is nonempty [19]. A play of the game is a possibly infinite sequence of positions $(s_0, q_0), (s_1, q_1), \ldots$, where each position is a node in the product (called an and–or tree in the sequel) of the Kripke structure and the alternating automaton for the formula. The structure of the and–or tree determines which player makes the next move. The winner of a play can be established when either a node that is labelled true (Port wins) or false (Brandy wins) is found in the play, or when a position in the current play is revisited, i.e., an infinite path is found. When an infinite path is found, the acceptance condition is then considered in order to determine the winner of the play. The cases are summarised in Table 1.

The serial implementation uses a depth first search (DFS) algorithm with a stack to store the current path. An infinite path is then given by all the

elements on the stack between the depth where a position is revisited and the current depth of the stack. For efficiency a store keeps track of results for states from which all moves have been made, so that when they are revisited the results can be reused, but then it may happen that an incorrect result is stored, since play is truncated whenever a position is revisited. To ensure that a result is correct when stored, a new game is played for a state once all moves from that state have been played. This new game uses a new results store and stack, so that any infinite play that might have been truncated during the previous play is now played to completion. New games can be played recursively, so to ensure termination, new games are not played from states for which new games are already being played. Once a new game for a state has been completed the new game's stack and results store are deleted and the result is stored in the original results store.

4 PMC: A Parallel Model Checker based on Alternating Automata and Game Theory

In the serial algorithm the nonemptiness game is played following the DFS path created by a serial reachability analysis algorithm. In the parallel algorithm the nonemptiness game is played using the parallel reachability analysis algorithm described in Section 2. This latter choice has two consequences. First, in a parallel analysis the and–or tree is no longer explored in a depth-first manner as in the serial case. Instead, an unexpanded position is removed from a stack and can therefore be a random position from anywhere in the and–or tree. Consequently, at any point, a set of paths each at a possibly different stage of generation is explored concurrently. Second, since there is no single stack, positions are added to the store as and when they are generated, so that other processes can detect visited positions. This means that when a process revisits a position it should still determine whether revisiting the position closes a cycle or not. A further implication is that a process can reach an old position while another process is still busy computing the result for it. The algorithm can handle this scenario in two ways. When an old state s_{old} is reached, the process can continue computing the result of the old state and thereby duplicate the work of the first process that reached s_{old}, or s_{old} can keep track of all the positions waiting for its result and forward the result to them once the winner has been established. We implemented the latter.

The other issue we need to consider is the influence of reusing results on the final winner of the game. Recall that the winner of a play is determined when either a true or false terminal state is reached or when a position on the current path is revisited (an infinite play is found). When a process reaches

a terminal true or false state it can immediately store the result and forward the result to the position's predecessors. When a process revisits a position, and the result of the position is not known, the algorithm must first determine whether it is on an infinite play, and if so, whether Brandy or Port wins on the play. Both goals are achieved by playing a new game from the old state. To avoid duplicating work, a new game is only played from an old state if no other process is currently playing a new game from that state. When a process reaches an old position s_{old}, while expanding a position s_{pre}, and another process is currently playing a new game from s_{old}, s_{pre} is added to s_{old}'s predecessors list. Then, once the new game is finished the result will be forwarded to s_{pre}. New games are played locally on one processor and therefore the serial nonemptiness game described in Section 3 can be played. Note that only results from completed local new games or a terminal true or false state are stored and reused. Since these results are correct, no further new games are needed.

In the parallel model checking algorithm, the game logic is embedded in the parallel reachability analysis algorithm that was described in Section 3. The parallel reachability analysis algorithm follows three steps: (1) idle – obtain a state, (2) idle – check termination, and (3) busy – compute successors. In the parallel nonemptiness game these same steps are executed with two exceptions. The stacks now store work items and not states, and the action performed during step (3) now depends on the work item. There are four possible work items: Expand, Play, Result, and Play–Local–Game. The pseudocode for the main program logic of the parallel nonemptiness game is, except for line 8, identical to the parallel reachability analysis algorithm in Fig. 1. In the parallel nonemptiness game, **Procedure ComputeSuccessors** on line 8 is replaced by **Procedure ProcessWorkItem**. The pseudocode for these functions is given in the Appendix. The actions ProcessWorkItem performs for the different work items are: **Expand** Expand the expression in the alternating automaton table that corresponds to the current alternating automaton state and the truth values of the propositions at the current Kripke state; **Play** Make a move in the product automaton by combining the new alternating automaton state with all the successors of the current Kripke state; **Result** Store the result of the current product state and then forward the result to all the states in the predecessor list waiting for the result of this state; **Play–Local–Game** Play a local new game from this state. After the new game has been played, a Result work item is created.

4.1 Correctness

The correctness of the parallel algorithm has been proved by first establishing that the serial and parallel algorithms will construct isomorphic and–or graphs. Then it was established that the serial nonemptiness game and the parallel nonemptiness game will determine the same winner from the initial position of an HAA for a given Kripke structure and CTL* formula by proving that the serial nonemptiness game and the parallel nonemptiness game will determine the same winner for a specific path, that the serial nonemptiness game and parallel nonemptiness game will determine the same winner for a specific position in the and–or tree, that the stored results can be reused, and that the forwarding of results in the parallel algorithm is correct [13]. Other properties about the parallel algorithm of PMC, such as freedom from deadlock have been verified by using both SPIN and PMC itself [13].

5 Performance

Ideally, an algorithm that runs in time T_S on one processor will run in time $T_P = T_S/P$ on P processors. Unfortunately T_P is almost always greater than T_S/P, because of extra overhead involved in executing the code on more than one processor. More accurately $T_P = T_S/P + \theta_P$, where θ_P, the overhead term, is the difference between actual and ideal execution time [1,8].

There are several parallelisation costs that account for overhead. The first, θ_{IP}, is overhead because of insufficient parallelism. A parallel algorithm typically has a sequential component that cannot run in parallel, for example the combined startup and tidyup time of the algorithm. This overhead is often captured as Amdahl's law [10], which states that if the sequential component of an algorithm accounts for $1/s$ of the program's execution time, then s is the maximum possible speedup that can be achieved on a parallel computer. The second cost, θ_{Sched}, is scheduling overhead; this is the time that each thread needs to execute scheduling code at the beginning of a parallel section. The third cost, θ_{LI}, is a result of load imbalance. The fourth cost, θ_{Sync}, is the synchronisation overhead; it accounts, for example, for the time needed to acquire or release a lock. The final cost, $\theta_{Unknown}$, is for extra overhead that has not been accounted for in the other terms, for example memory access time. The time that an algorithm runs on P processors can therefore be given as: $T_P = T_S/P + \theta_{IP} + \theta_{Sched} + \theta_{LI} + \theta_{Sync} + \theta_{Unknown}$

The values for the different overheads can be measured by executing a reachability analysis on 1 process and using high resolution timers to measure the time it takes to execute different chunks of code. The value of T_P can then be calculated for $P = 1$ to 16 to predict the performance of the algo-

rithm. This can then be compared against the measured values for T_P during experiments. The performance of a parallel algorithm can be visualised by plotting $1/T_P$ against the number of processes P. These performance graphs have the advantage that they provide a visual representation of the parallel algorithm's scalability and at the same time the exact execution times can still be computed from the values presented in the graph.

5.1 Theoretical vs Experimental Performance

To model the theoretical performance of the parallel reachability analysis algorithm the overhead values for the parallel algorithm were measured and two different models were considered: Model A, which captures the best possible behaviour of the algorithm where there is no idle time and Model B, which captures a worst case scenario, where the stealing of states has to be synchronised with other accesses to the same shared stack and each time a process wants to steal a state it has to wait for all the other processes to gain and release the lock before it can access the shared stack. The performance graphs of Model A and Model B are shown in Fig. 2*a*.

The third performance graph in Fig. 2*a* shows the real performance of the algorithm; the performance graph was obtained from results of experiments that were run to time the reachability analysis on a 100 million state graph with a branching degree of 3. For each value of P, the reachability analysis was executed ten times and the performance graph labelled *real*, in Fig. 2*a*, shows the average over each set of ten runs. The vertical lines at regular intervals show the fastest and slowest result obtained during the ten runs on the corresponding process. The average over a number of runs is taken because there are many variants which influence the running time, e.g., the unpredictability of cache line usage inferred by cache contention, and the order in which processes gain access to the shared stack. Analysis of the results showed that the average has stabilised within its ten runs.

The performance of the parallel algorithm is very good and scales well with an increase in the number of processors. The implementation takes approximately 4 minutes to complete the reachability analysis of a 100 million state graph on one process and approximately twenty seconds on sixteen processes. Model B shows similar behaviour to the experimental results of the implemented algorithm if its idle overhead is changed to model a scenario where a process that wants to steal a state has to wait for only one other process to complete a single access to the shared stack before it gains access to it.

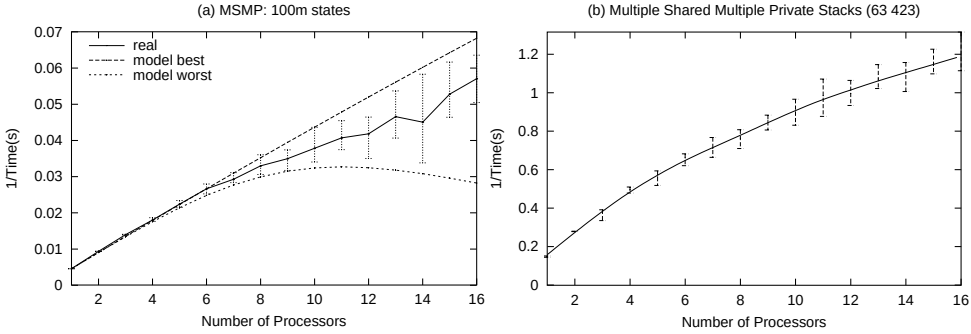

Fig. 2. (*a*) Theoretical vs real performance of exploring a 100 million state graph (*b*) Checking deadlock for an ESML model of the parallel reachability analysis algorithm

5.2 Experimental performance on ESML Models

To evaluate the performance of the model checker on real models, PMC was integrated with a state generator for ESML, a Promela-like modelling language [9]. Fig. 2*b* shows the performance of checking deadlock for an ESML model of the parallel reachability analysis algorithm described in Section 2 and Fig. 3*a* shows the performance graph of checking the liveness property EFAGp for a communications model. More graphs are given in Fig. 7 and Fig. 8.

For checking safety properties with the parallel model checker, effectively the same speedup resulted as for parallel reachability analysis. However, the properties are typically satisfied or violated before the entire state graph has been searched and can also be satisfied at different depths on different paths. The parallel algorithm also tends to find shorter paths than a model checker with a sequential DFS algorithm. For checking liveness properties the results varied, but when there is a speedup liveness checking also scaled well with an increase in the number of processors. Liveness checking requires more work (longer independent jobs) than safety checking, but requires extra synchronisation to store the information needed for checking cycles. However, the effect of longer independent jobs outweighs the effect of the extra synchronisation and generally better speedup is obtained than with safety checking.

It was further found that the variation in performance of liveness checking is usually higher than for safety checking, because the order in which states are visited also influences when and from which states local games are played. In some cases this variation is particularly high. As an example the performance of the model checker when checking a liveness property AGEFp for a Producer–Consumer model is depicted in Fig. 3*b*. On six processors, for example, the property was checked in 36 seconds after visiting 4000 states (excluding the states visited during local games) during the slowest run and in one second

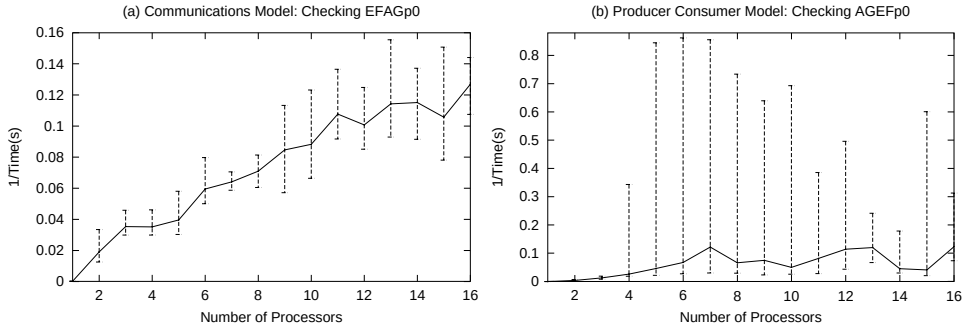

Fig. 3. Checking (a) EFAGp for a Communications Model and (b) AGEFp for a Producer–Consumer Model

after visiting 250 states during the fastest run.

The number of local games that is played also has an influence on the performance of the model checker and, for a particular liveness property, depends on the model being checked. Often a large number of local games need to be played compared to the total number of states of the finite-state machine generated for the model. This has the effect that some processes are busy playing local games while others run out of work and then stay idle for long periods of time while local games are played; see Fig. 8d.

6 Conclusions

Our first objective was to evaluate the feasibility of parallel model checking on shared memory architectures and it is clear from the experimental results that model checking can be efficiently implemented on these architectures. The speedups are good and the dynamic load balancing algorithm works effectively; there is practically no idle time due to unbalanced load, except when the number of local games is high compared to the number of global states. The second objective was to identify any pitfalls when parallelising model checking for shared memory architectures. The main pitfalls that were identified are synchronisation overhead and false sharing (see Fig. 9), which occurs when two or more processors attempt to write to different words in the same cache line. It was found that, for efficiency, mutual exclusion locks should be used with care and memory should be allocated so that false sharing is avoided.

For future work we plan to implement an improved algorithm that re-uses all intermediate results of local games to avoid redundancy and provide work faster. More generally we also identified a need for benchmark models and a full investigation into the integration of serial state space reduction techniques into parallel algorithms.

Acknowledgement

In addition to our sponsors, we also wish to thank in particular the staff of the CNC (Centre for Novel Computing) within the Department of Computer Science for helpful discussion on the parallelisation approach and implementation and for dedicated access to their SGI Origin 3400 machine.

References

[1] M. K. Bane and G. D. Riley. Extended overhead analysis for OpenMP. *Lecture Notes in Computer Science*, 2400:162–166, August 2002.

[2] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL model-checking in SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*. Springer–Verlag, May 2001.

[3] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, pages 390–404, Austin, Texas, November 2000.

[4] O. Bernholtz. *Model Checking for Branching Time Temporal Logics*. PhD thesis, The Technion, Haifa, Israel, June 1995.

[5] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software (SPIN 2002)*, Lecture Notes in Computer Science, Grenoble, France, April 2002. Springer–Verlag.

[6] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model-checking based on negative cycle detection. *Lecture Notes in Computer Science*, 2245, 2001.

[7] L. Brim, J. Crhová, and K. Yorav. Using assumptions to distribute CTL model checking. In *Proceedings of the 1st International Workshop on Parallel and Distributed Model Checking (PDMC 2002)*, pages 80–95, Brno, Czech Republic, 2002.

[8] J. M. Bull. A hierarchical classification of overheads in parallel programs. In I. Jelly, I. Gorton, and P. Croll, editors, *Proceedings of the 1st IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219. Chapman Hall, 1996.

[9] P. J. A. de Villiers and W. Visser. ESML—a validation language for concurrent systems. In *Proceedings of the 7th Southern African Computer Symposium*, pages 59–64, July 1992.

[10] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[11] O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 350–363, Paris, France, July 2001. Springer–Verlag.

[12] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In A. P. Sistla E. A. Emerson, editor, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, Illinois, July 2000. Springer–Verlag.

[13] C. P. Inggs. *Parallel Model Checking on Shared-Memory Multiprocessors*. PhD thesis, University of Manchester, Manchester, January 2004.

[14] C. P. Inggs and H. Barringer. Effective state exploration for model checking on a shared memory architecture. In *Workshop on Parallel and Distributed Model Checking (PDMC'02)*, volume 68 of *Electronic notes in Theoretical Computer Science (ENTCS)*, Brno, Czech Republic, 2002.

[15] P. Krčál. Distributed explicit bounded LTL model checking. In L. Brim and O. Grumberg, editors, *Proceedings of the 2nd International Workshop on Parallel and Distributed Model Checking (PDMC 2003)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, Boulder, Colorado, July 2003. Elsevier.

[16] A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. report 00176, Institut für Informatik, Universität Freiburg, July 2002.

[17] U. Stern and D. Dill. Parallelizing the Murϕ verifier. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–278, Haifa, Israel, June 1997. Springer–Verlag.

[18] M. Y. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *Proceedings of the 1st Symposium on Logic in Computer Science*, pages 322–331, 1986.

[19] W. Visser and H. Barringer. Practical CTL model checking: Should SPIN be extended? *Software Tools for Technology Transfer*, 2(4):350–365, March 2000.

7 Appendix

```
1  Procedure ParallelReach(initState)
2     start parallel
3        if (procid = 0) then next := initState;
4        else next := EMPTY; endif;
5        do
6           if (next = EMPTY) then next := PopFromStack(); endif;
7           if (next ≠ EMPTY) then
8              ComputeSuccessors(next);
9           else
10             CheckForTermination(procid);
11          endif;
12       while (¬terminate);
13    end parallel;
14 endprocedure
```

```
1  Procedure ComputeSuccessors(s)
2     forall successors t of s do
3        if (t is new) then
4           AddToStore(t);
5           if (t is first successor) then s := t;
6           else PushOnStack(t); endif;
7        endif;
8     endforall;
9  endprocedure
```

Fig. 4. Procedure ParallelReach and a subfunction for ParallelReach

```
1  Procedure PopFromStack(s, id, privstack)
2     if (notempty(privstack)) then
3        s := Pop(privstack);
4     endif;
5     if (s = EMPTY && notempty(sharedstacks[id])) then
6        ompLock(stacklock[id]);
7        s := Pop(sharedstacks[id]);
8        ompUnlock(stacklock[id]);
9     endif;
10    while (s = EMPTY && more shared stacks to check) do
11       id := next processor;
12       if notempty(sharedstacks[id])) then
13          ompLock(stacklock[id]);
14          s := Pop(sharedstacks[id]);
15          ompUnlock(stacklock[id]);
16       endif;
17    endwhile;
18 endprocedure
```

```
1  Procedure PushOnStack(s, id, privstack)
2     if (notfull(sharedstack)) then
3        ompLock(stacklock[id]);
4        Push(sharedstack, s);
5        ompUnlock(stacklock[id]);
6     else
7        Push(privstack, s);
8     endif;
9  endprocedure
```

```
1  Procedure ParallelGame(initState)
2     start parallel
3        if (procid = 0) then next := initState;
4        else next := EMPTY; endif;
5        do
6           if (next = EMPTY) then next := PopFromStack(); endif;
7           if (next ≠ EMPTY) then
8              ProcessWorkItem(next);
9           else
10             CheckForTermination(procid);
11          endif;
12       while (¬terminate);
13    end parallel;
14 endprocedure
```

```
1  Procedure ProcessWorkItem(next)
2     switch (next.type)
3     case PLAY:
4        cur := next;
5        deadlock := Move(cur, next);
6     case EXPAND:
7        next := ExpandFormula(next);
8     case RESULT:
9        cur := next;
10       next := EMPTY;
11       ProcessResult(cur, next);
12    case PLAY–LOCAL–GAME:
13       PlayLocalGame(next);
14    endswitch;
15 endprocedure
```

Fig. 5. Procedure ParallelGame and subfunctions for both ParallelReach and ParallelGame

```
1  Function SplitAndOr(parent, parentId, altState)
2     next.altState := FormulaLeft(altState);
3     next.type := FormulaLeftQuant(altState);
4     next.result := ResultsRemoveFreeSlot();
5     next.idCopy := next.result.id;
6     AddPredecessor(next, parent, parentId);
7     next.result.pos := ANDORPOS;
8
9     cur.altState := FormulaRight(altState);
10    cur.type := FormulaLeftQuant(altState);
11    cur.idCopy := cur.result.id;
12    cur.result := ResultsRemoveFreeSlot();
13    AddPredecessor(cur, parent, parentId);
14    cur.result.pos := ANDORPOS;
15
16    PushOnStack(cur);
17    return next;
18 endfunction
19
20 Function Move(cur, next)
21    switch (next.type)
22       case AND:
23          cur.result.resToGo := 2;
24          cur.result.andOr := AND;
25          next := SplitAndOr(cur.result, cur.idCopy, cur.altState);
26          deadlock := FALSE;
27       case OR:
28          /* same as AND case, but AND replaced with OR */
29       case AND−SUCC:
30          next.result.andOr := AND
31          next.type := EMPTY;
32          deadlock := MoveInKripke(cur, next);
33       case OR−SUCC:
34          /* same as AND−SUCC case, but AND replaced with OR */
35       endswitch;
36    endif;
37    return deadlock;
38 endfunction
```

```
1  Procedure ProcessResult(cur, next)
2     tmp := cur.result;
3     ompSetLock(tmp.lock);
4     if (tmp.id = curState.idCopy) then
5        if ((tmp.andOr = OR && Result(cur) = TRUE)
6           || (tmp.andOr = AND && Result(cur) = FALSE)) then
7           tmp.result.resToGo = 0;
8        else
9           tmp.result.resToGo−−;
10       endif;
11       if (tmp.result.resToGo ≤ 0) then
12          if (tmp.pos ≠ ANDORPOS) then
13             StoreSetResult(tmp.pos, Result(cur));
14          endif;
15          forward := TRUE;
16       endif;
17    endif;
18
19    if (foward) then
20       if (cur = initial state) then
21          mcResult := Result(cur); terminate := TRUE;
22       else
23          while NotEmpty(tmp.predecessors) do
24             cur.result := RemovePredecessor(tmp);
25             cur.idCopy := GetStoredIdCopy(tmp);
26             if (Empty(next)) then next := cur;
27             else PushOnStack(cur); endif;
28          endwhile;
29          tmp.id++;
30       endif;
31    endif;
32    ompUnsetLock(tmp.lock);
33    if (forward) then AddSlotToFreeList(tmp) endif;
34 endprocedure
```

Fig. 6. Functions for the parallel nonemptiness game

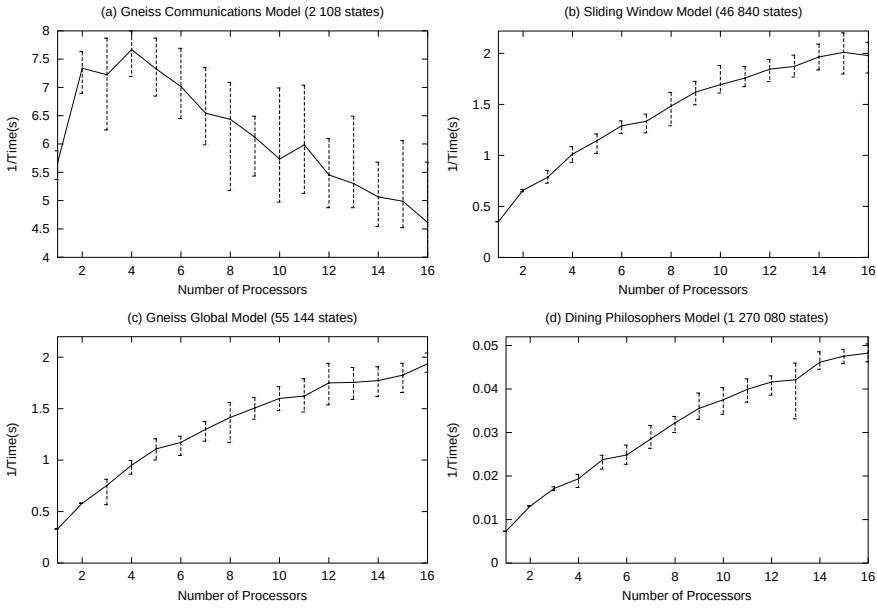

Fig. 7. The performance of an exhaustive reachability analyses on each of four ESML models; see the caption for the size of each model.

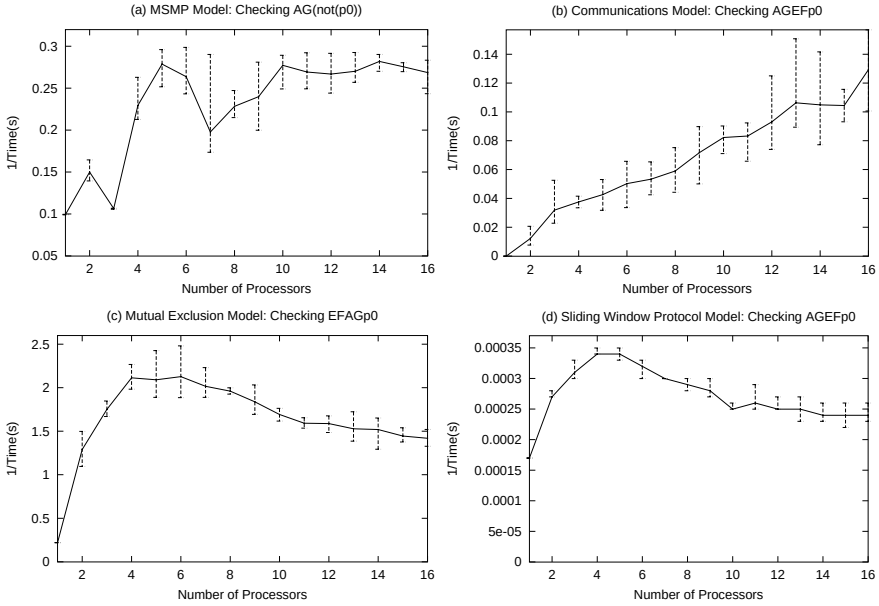

Fig. 8. The performance of the model checking algorithm when checking temporal logic properties for different models. (a) Checking AG$\neg p$ for a model of the parallel algorithm described in this paper; more or less 63400 states are visited per run. (b) Checking AGEFp for a Communications Model; between 1400 and 3000 states are visited per run. (c) Checking EFAGp for a Mutual Exclusion Model; between 230 and 260 states are visited per run. (d) Checking AGEFp for a Sliding Window Model; between 57000 and 57900 states are visited per run.

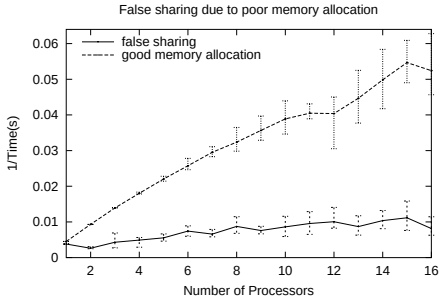

Fig. 9. The Effect of False Sharing on Performance. In the implementation each process has three private variables to store the current state, the current successor, and the next state to be expanded. The graph labelled *false sharing* shows the performance of the algorithm when the three private variables of all the processes are allocated memory in the same cache line. The graph labelled *good memory allocation* shows the performance of the algorithm when the three private variables of each process are allocated memory in such a way that they do not end up in the same cache line as the three private variables of any of the other processes. As the performance graphs show, the effects of false sharing can be quite severe.