Electronic Notes in Theoretical Computer Science 68 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume68.html 16 pages

Effective State Exploration for Model Checking on a Shared Memory Architecture

Cornelia P. Inggs [1,2] and Howard Barringer [3,4]

Department of Computer Science
University of Manchester
UK

Abstract

In this paper we present results from experimental studies investigating implementation strategies for explicit-state temporal-logic model checking on a virtual shared-memory high-performance parallel machine architecture. In particular, a parallel state exploration algorithm using a two-queue structure for load balancing is proposed and its performance analysed at the hand of experimental studies. We then discuss implementation issues for parallel automata-theoretic model checking using this parallel state exploration algorithm.

1 Introduction

Model checking is an established technology for automated verification of designs that has been taken up by industry to check correctness properties of many critical systems [8]. This is mainly due to the tremendous advances that have been made over the past decade in developing specialised codings and algorithms to reduce the burden of the state explosion problem [4,10,15]. However, state explosion is still a well-studied research problem and one technique that has gained more interest recently is the parallelisation of model checking.

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 (such as on one of our

[1] 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] cinggs@cs.man.ac.uk
[3] Partially supported under EPSRC grant GR/M05744.
[4] howard@cs.man.ac.uk

Manchester Computing machines, an SGI Origin 3000 with 512 processors and 512 GB of 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!

In this paper we present results from experimental studies investigating implementation strategies for explicit-state temporal-logic model checking on a virtual shared-memory high-performance parallel machine architecture. An efficient parallel state exploration algorithm is proposed and the implementation of a parallel automata-theoretic model checker based on this algorithm is discussed.

Related Work:

Much of the extant research of parallel model checking has focused on implementations over distributed networks, largely through the ease of access to these computing farms [13,20]. A common approach for parallel model checking on both distributed and shared memory architectures is to partition the state space across the processors using a slicing function. Different slicing techniques have been employed, 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 [20,18,12].

Stern and Dill [20] parallelised the Murϕ model checker by partitioning the visited states across the processes using randomised 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 and a number of other authors based their implementation on this algorithm. Among the first was Lerda and Sisto [18] who implemented a distributed version of SPIN [14] for checking safety properties. They replaced the random hash function with a slicing function that tries to minimise cross-transitions (transitions between states belonging to different processes) by using the structure of the states. Their algorithm only showed speedups for state graphs that were too big to be explored on a single processor and then started swapping. Behrmann et al. [6] used the hash function technique of [20] in their distributed implementation of the symbolic model checker UPPAAL, but since their model checker performs better using a breadth-first search, they added a heuristic 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. Garavel et al. [12] implemented a distributed state space construction algorithm for the CADP verification tool set [17]. Their slicing algorithm partitions states independent of their structural properties and therefore independent of the specification language used.

Recent work on parallel model checking algorithms using similar static slicing techniques focused on checking liveness properties. Martin and Huddart [19] 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. This algorithm has not been implemented and has the drawback that the entire graph has to be constructed before pruning starts. Barnat et al. [2] extended the algorithm of Lerda and Sisto [18] to allow liveness checking. The algorithm performs a distributed depth-first search (DFS) as in [18], 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 of LTL properties for larger state spaces than the sequential version. Brim et al. [3] replaced the nested DFS algorithm with a negative cycle detection algorithm developed to solve the single-source shortest path problem. Their implementation allows reasonable distribution. Bollig et al. [7] tried to overcome the costliness of parallel cycle detection by developing an algorithm without cycle detection. They studied a game-theoretic model checking algorithm for the alternation-free mu-calculus and identified a characteristic of the alternation-free fragment of mu-calculus that allowed them to define a sequential algorithm that uses a colouring mechanism instead of cycle detection and this allowed an efficient parallel implementation that scales well.

As far as we know, the only model checking algorithm that implements a dynamic slicing algorithm is the distributed symbolic algorithm of Heyman et al. [13]. 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. Finally we mention a parallel state-space exploration algorithm from a non model checking context. The algorithm was developed by Allmaier and Horton [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. Access to the B-tree is synchronised by mutual exclusion locks, essentially one per node. They achieved speedups – but maintaining the B-tree and keeping locks to synchronise access to each node are extra overheads and hence reduce performance.

This paper:

We propose a parallel algorithm for state exploration on a virtual shared memory parallel machine. The algorithm is implemented as a parallel composition of N 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, however, are maintained in a global hash table, not distributed over the processes – contention is minimised by removal of all software syn-

chronisation locks at a small cost of possible redundant work.

Our technique is described in the next section and its performance analysed at the hand of experimental results in Section 2.2. In Section 3 we discuss the implementation issues for an alternating automata model checker using the proposed parallel state-exploration algorithm. Our conclusions are outlined in Section 4.

2 Parallel State Exploration

Before we proceed to describe the parallelised state exploration algorithm, it will be helpful to clarify some nomenclature. A processor refers to the hardware and the number of processors available is therefore fixed for a specific machine, whereas processes refer to threads of control and can vary. In our case a single process runs on a single processor, so five processes will execute on exactly five processors. It is worth noting that on the distributed shared memory architecture concerned, the system views memory as a global shared entity, but the memory is in fact physically distributed across the processors.

Consider a search, which happens to be a DFS, on a single processor where a stack is used to store unexpanded states (states for which no successors have been computed), and all computed states are added to a store (hash table). The algorithm starts with the initial state on the stack and the store empty. It repeatedly pops a state from the stack, computes its successors and pushes them on the stack. The algorithm terminates when the stack is empty and all the states reachable from the initial state are in the store.

In a naïve parallelisation, where the search is divided among multiple processes, the single stack and store can be shared between all the processes by adding two mutual exclusion locks for synchronising access to the two data structures respectively. But experiments with implementations of this and similar techniques showed that the use of locks for synchronisation on a shared architecture can almost serialise an otherwise parallel implementation [16]. Experiments also revealed that Java, our initial choice of language for implementation, had major drawbacks on the SGI Origin machine. Firstly memory allocation and garbage collection on the global heap involved very costly synchronisation, and secondly no tools were available to identify the bottlenecks.

Currently all our implementations are in C. We implemented several prototypes to experiment with different load balancing techniques and analysed the synchronisation costs of the different techniques. Our most efficient solution, with respect to speedups during parallel exploration, uses a mixture of shared and private data structures for load balancing and storing results, and detects termination using the low overhead token based algorithm described in [11]. A more detailed description of the parallel setup is given in the next section.

2.1 Implementation Details

Each of the processes participating in the state exploration has a private copy of the state generator. The state generators are consistent, i.e., they all produce exactly the same successors for any given state. The data structures, as depicted in Fig. 1, include a single shared store for storing visited states and two queues per process for storing unexpanded states. The first queue is an unbounded private queue and the second a bounded shared queue. A process can only add states to its own private and shared queues, not on the queues of any other process, but it can remove states from the shared queues of other processes (which is called work stealing). Each shared queue therefore has a lock to synchronise read and write access to it. The store, a hash table, has no mutual exclusion locks to synchronise access. This can however 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.

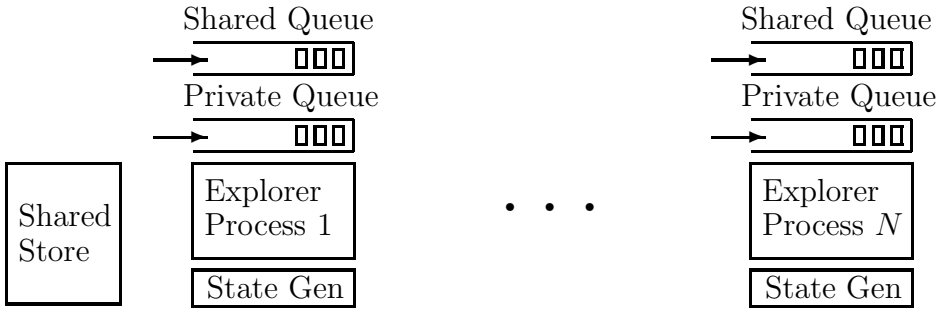

Fig. 1. Parallel Graph Search Architecture

All the explorer processes execute exactly the same algorithm. However, one explorer process is chosen to execute two extra tasks: the first is to start the termination detection algorithm and the second to set the termination flag once termination has been detected. Each explorer process executes the following set of tasks until all the reachable states have been computed and termination has been detected:

(i) *Remove a state.* When a process is idle it tries to remove a state from the queues in the following order. It first checks its private queue for a state. If no state was found, the process acquires the mutual exclusion lock to remove a state from its own shared queue. Failing again to remove a state, it then searches through all the other shared queues until it finds a nonempty queue or finds that all the shared queues are empty.

(ii) *Check termination.* When an idle process can not remove a state from any of the queues it has access to, the termination detection algorithm is executed: the idle process checks if it has the termination token and if so, passes the token to the next process. No lock is needed to check for termination.

(iii) *Compute successors.* If a process removed a state, all the successors of the state are generated and the new successors are added to the store. The first new successor computed is kept to be expanded by the current process and the others are added to one of its queues. If the shared queue is not full the state is added to that queue, otherwise the state is added to the private queue.

With sufficient work available at each process, the processes use there own private queues most of the time and this leaves the shared queue for load balancing purposes. Then when load balancing is needed, because processes have become idle, the two-queue structure ensures that idle processes stealing work do not interrupt any of the busy processes. To minimise contention on the shared queues an idle process p_i always starts its search for a nonempty shared queue at process $p_{(i+1) \bmod N}$, where N is the number of processes and processes are denoted by $p_0, p_1, \ldots, p_{N-1}$ respectively.

The efficiency of load balancing and division of the state graph between the processes depends on a combination of the structure of the state graph and the sizes of shared queues. For example, a purely linear graph, where each state has one successor, will be generated on only one processor, which is faster than sharing this task between several processes. The size of the shared queue does not influence the exploration of such a graph at all. But consider a graph with a branching factor of four. After process p_i generated the initial state's successors there is already enough work for four processes. However for a shared queue that holds one state at a time, only one other process, say p_{i-1} can also start work, because the two other extra states will be in process p_i's private queue, waiting for p_i to become idle. Small shared queues can therefore hamper load balancing. However with very large shared queues, a processor will add states that could have been added to its private queue to its shared queue, controlled by an expensive mutual exclusion lock.

The relationship between shared queue sizes and branching factors was investigated by exploring several artificially generated graphs on a 16-processor SGI Origin 3400 with 4GB of memory; the results obtained are summarised in the next section.

2.2 Experimental Results

The experiments have been run on a number of artificially generated graphs with different structures and of different sizes. However, the performance results reported in this section were obtained from a set of graphs with the following structure: each state in the graph has x unique successors and one transition to an old state (a state that would have been visited by the time this transition is computed).

For each case, the number of successors (x) and the size of the shared queues were fixed and then the graph was explored 5 times. The average over the 5 runs was used for the performance graphs in Fig. 2 and 3. 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, or where states are physically stored. Analysis of the results, run in single user mode, showed that the average has stabilised within its 5 runs.

The performance graphs on the left-hand side of Fig. 2 show the exploration times of 25-million-state graphs with a fixed branching degree, $x = 3$, while the shared queue size vary from 1 to 4 and the performance graphs on the right-hand side of Fig. 2 show the exploration times of 25-million-state graphs with a fixed shared queue size of 4 and varying branching degrees, $x = 1, 2, 3, 4$.

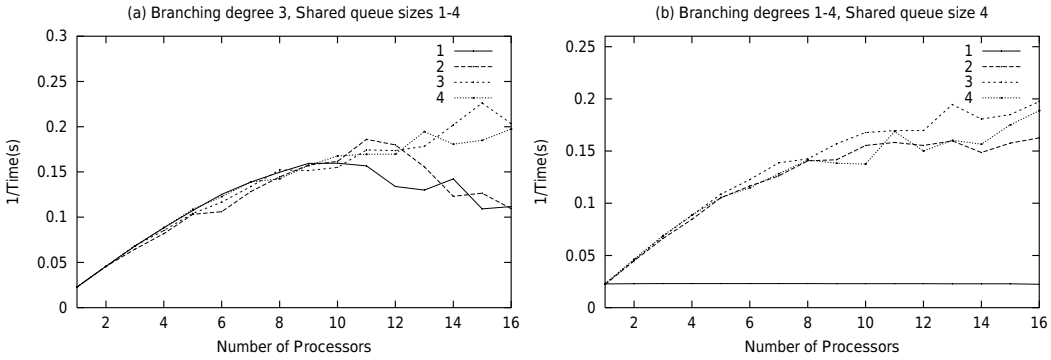

Fig. 2. Exploration performance of graphs with varying branching degrees and shared queue sizes.

From these figures it is clear that the optimum shared queue sizes are equal to the branching degree or one more than the branching degree. Other performance graphs not shown here confirmed this relationship. The worst case scenario is where a very small shared queue size is used when exploring a graph with a high branching factor. The smallest shared queue size of 1 is not that expensive for a graph with a branching factor of two, but for higher branching factors performance starts dropping when the number of processors is greater than a certain threshold as can be seen in the graph on the left-hand side of Fig. 2.

The performance graphs on the right-hand side of Fig. 2 shows that a fixed shared queue size of 4 was sufficient for effective load balancing of graphs with branching degrees less than or equal to 4. Overall it was found that a slightly larger-than-optimum shared queue size is better than a slightly smaller-than-optimum shared queue size. For a larger-than-optimum shared queue size the variation in performance for graphs with different branching factors is not orders of magnitude apart. Therefore to explore a set of graphs with different branching degrees or a single graph with varying branching degrees (as is common for the graphs explored during model checking), a branching degree closer to the optimum of the higher branching degrees should be chosen if a static queue size over the runs is preferred.

For graphs with a significant difference between the smallest and largest branching degree, one can also consider changing the shared queue size dy-

namically, or only adding successors to the shared queue when the state being expanded has more than a certain number of successors. Another strategy that will be cheaper than changing the queue size dynamically, is to start with a shared queue size of say x and then use the average number of successors generated per state to determine the size of the shared queue for the rest of the search. If necessary this calculation can be repeated every k states.

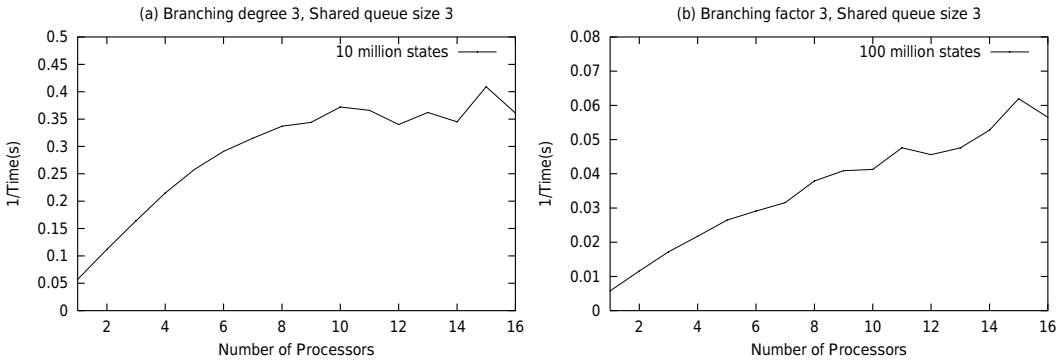

Fig. 3. Exploration performance of small and large graphs.

The performance graphs of exploring a 10 million state graph and a 100 million state graph respectively are shown in Fig. 3. The graph on the left-hand side of Fig. 3 shows the general performance of state graphs with around 10 million states or fewer. The speedups are almost linear up to a certain threshold where performance reaches a plateau. This happens because the work of exploring a state graph of only 10 million states is insufficient to make full use of all the processors; even on a single processor a full exploration completed in under 18 seconds. For larger graphs, the speedups do not reach a plateau within 16 processors as can be seen in the graph on the right-hand sided of Fig. 3. It was also found that although the rate of speedup did not vary considerably for graphs with many more than 10 million states, the rate of speedup increases slightly as the graph sizes increase. No experiments have been executed on the SGI Origin 3000 with 512 processors yet, only on the 16-processor SGI Origin 3400, but it is expected that the speedups will continue to increase until the computing power of the processors is more than what is needed for the graph being searched and the synchronisation overhead for the processors becomes dominant.

State exploration, i.e., computing the reachable states of a system, forms an integral part of model checking. In fact checking safety properties alone is a reachability analysis problem, and liveness checking follows with extra structures to identify infinite cycles. In the next section we discuss the implementation of a parallel model checker for both safety and liveness properties based on the parallel state exploration algorithm presented.

3 Parallel Model Checking using Alternating Automata and Game Theory

A state graph can be represented as a Kripke structure: a four-tuple $K = (S, T, s_0, L)$ where S is a finite set of states, $T \subseteq S \times S$ is a transition relation that must be total (for every $s_i \in S$ there exists at least one s_j such that $(s_i, s_j) \in T$), s_0 is an initial state, and $L : S \mapsto 2^{Prop}$ maps each state to a set of atomic propositions true in that state. Then for a Kripke structure K and temporal logic formula ϕ, a model checker determines if $K \models \phi$. In the automata theoretic model checking context we consider, the temporal logic formula is first translated to an automaton A_ϕ that accepts exactly all the computations that satisfy the formula. 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 [23,21,9,5].

In our context we use Hesitant Alternating Automata [5] to represent branching time formulae (CTL*) and parallelise the game oriented implementation developed by Visser [22]. As a brief introduction, automata over infinite trees (tree automata) run over leafless Σ-labelled trees. A run r of an alternating automaton A on a tree T is a tree where the root is labelled by s_0 and every other node is labelled by an element of $(\mathbb{N}^* \times S)$. Each node of r corresponds to a node of T. A node (x, s) in r corresponds to the automaton in state s 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. The run 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 (HAA's) the acceptance condition is a pair of states (G, B). 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 either, 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:

- if S_i is transient, then $\delta(s, a, k)$ contains no elements from S_i
- if S_i is existential, then $\delta(s, a, k)$ contains only disjunctively related elements of S_i
- 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 either get trapped 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).

Given a branching time formula ϕ and a Kripke structure K, model checking can be solved using HAA by following the basic steps outlined at the start of the section, but of course with the product and emptiness construction defined for HAA's.

In Visser [22] the non emptiness checking of the product automaton is defined as a two-player game, in which player 1 tries to show that the alternating automaton is empty whilst player 2 tries to establish that it is nonempty. A play of the game is a possibly infinite sequence of states $(q_0, s_0), (q_1, s_1), \ldots$, where each state is a node in the product of the Kripke structure and the alternating automaton for the formula. The structure of the product automaton determines which player makes the next move. The winner of a play can be established when either a node that is labelled true (2 wins) or false (1 wins) is found in the play or when a position in the current play is revisited, i.e., an infinite path in the product automaton is found. When an infinite path is found, the acceptance condition is considered to determine the winner of the play. When player 2 gets trapped in an existential S_i the play will be accepting if $\inf(\text{play}_\pi) \cap G \neq \emptyset$, note that player 1 can't get trapped in an existential set, because it only moves when there is an $\wedge$-choice. When player 1 gets trapped in an universal S_i the play will be accepting if $\inf(\text{play}_\pi) \cap B = \emptyset$. The cases are summarised in Table 1.

Player 1 wins	Player 2 wins
Play reaches a false	Play reaches a true
After a move by player 2, that revisits a position in the current play and $\inf(\text{play}_\pi) \cap G = \emptyset$	After a move by player 2, that revisits a position in the current play and $\inf(\text{play}_\pi) \cap G \neq \emptyset$
After a move by player 1, that revisits a position in the current play and $\inf(\text{play}_\pi) \cap B \neq \emptyset$	After a move by player 1, that revisits a position in the current play and $\inf(\text{play}_\pi) \cap B = \emptyset$

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

The implementation uses a 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 is used to keep 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 will now be played to completion. New games can be played recursively, so to ensure that new games are not played infinitely, 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 stack and results store for the new game are deleted and the result is stored in the original results store. A further optimisation discussed in [22] was to postpone new games, but is not considered in the subsequent discussion of the parallel implementation.

Parallel algorithm:

The basic setup and data structures of the parallel model checking algorithm are the same as the one developed for parallel graph search described in Section 2 and shown in Fig. 1. With such a setup, the graph is no longer explored in a depth-first manner as in the serial case above. Although a process will continue down a DFS path from a particular state until it reaches an old state, there is no backtracking mechanism. Instead an unexpanded state is removed from a queue and can therefore be a random state from anywhere in the graph. The result is a set of paths each at a possibly different stage of generation divided among the processes; there is therefore no stack storing the current path, and a new method for cycle detection is needed. A second result is that a process can reach an old state for which a result is not yet known, because another process is still busy computing it. So a mechanism is needed to keep track of states waiting for results. We'll consider these two cases separately.

Parallel Cycle Detection:

One strategy is to play new games locally instead of globally. Whenever a state is revisited a local cycle search is executed on the processor using a depth-first stack as in the single processor case, but this may not make effective use of the processors available.

Another strategy, and the one currently implemented, is to number path sections [5] such that each state in the state graph is associated with a unique section number. States are therefore mapped to a section number and its depth on that section. When a state is expanded, its first unvisited successor is assigned the same section as its parent and is at a depth one more than the depth of its parent. Each of the other unvisited successors is given a new section number. The section number is assigned by the processor that created the state. Each process i has a unique sequence of numbers $i, i+N, i+2N, \ldots$ (where N is equal to the number of processors participating in the search and processors are numbered from 0 to $N-1$) that it can assign to new path sections. Visited successors will already have a section number assigned to them. When there is a transition from state on path section π_i to a state on path section π_j, section π_j is called a branch of section π_i.

[5] Any subsequence of a path may be called a section of that path.

Consider the state graph shown in Figure 4. It was generated by 4 processes ($N = 4$). Six new path sections were encountered and numbered by threads 1 (path sections 1, 5, 9), 2 (path section 2) and 3 (path sections 3, 7) respectively. Path section $\pi_1 = s_0, s_1, s_3, s_6, s_{11}, \ldots$ has two branches, namely sections $\pi_5 = s_2, s_4, s_{10}, s_{12}, \ldots$ and $\pi_9 = s_7, s_{15}, \ldots$.

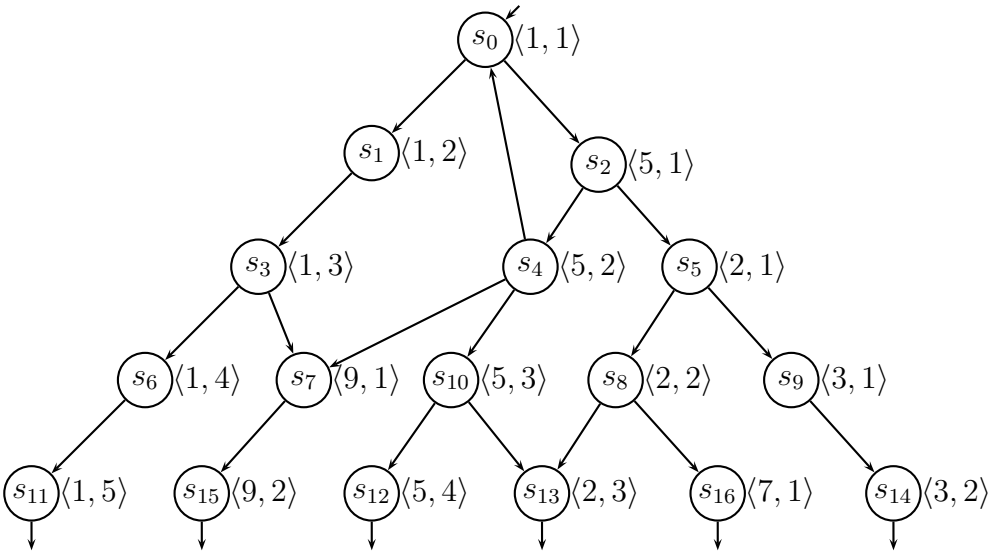

Fig. 4. A state labelled $\langle i, j\rangle$ is at depth j on path i.

Section numbers and their branches are stored in a path table, where each entry in a path table has the following fields:

(i) Section number x.

(ii) Branches. A list of entries $\langle i, y, j\rangle$ where $\langle i, y, j\rangle$ indicates that there is a transition from the state at depth i on path x to the state at depth j on path y; path y is therefore a branch of path x. As an example, the path table for Figure 4 is given in Table 2.

The path table is physically distributed across the processes. Each process i can only add new path sections to its part of the path table and since each process has a unique sequence of numbers that it can assign, no lock is needed to assign new section numbers and add new path sections to the path table. Multiple processors can however add branches to the same path section, because multiple processors can revisit a state on the same section, and therefore a mutual exclusion lock is needed to synchronise the insertion of branches.

With this implementation the following algorithm is executed to detect cycles whenever a visited state is reached. Given the path sections of the parent state and revisited state, the algorithm searches for a sequence of path sections where each section starts at a depth greater than or equal to the depth where the transition from the previous section reaches it, and the last

Section number	Branches (depth)
1	$\langle 1, 5, 1\rangle, \langle 3, 9, 1\rangle$
5	$\langle 1, 2, 1\rangle, \langle 2, 1, 1\rangle, \langle 2, 9, 1\rangle, \langle 3, 2, 3\rangle$
9	$\emptyset$
2	$\langle 1, 3, 1\rangle, \langle 2, 7, 1\rangle$
3	$\emptyset$
7	$\emptyset$

Table 2
Path sections for the graph in Figure 4.

section in the sequence reaches the starting section at a depth less than or equal to the depth where the search started.

Forwarding Results:

When a processor revisits a state for which the result is unknown, it simply inserts a pointer at the old state. Then, once the result of the old state is known, the list of pointers is traversed and the result forwarded to all the states waiting for the result. In the meantime the current processor can continue with other work.

Implementation Issues:

In the parallel model checker implementation the parallel state exploration algorithm executed at each process is replaced with the model checking algorithm and the queues no longer store states but work items. A work item then stores either a state in the product automaton or a result and each work item keeps the game counter of the game to which the work item belongs.

If the work item stores a state in the product automaton, the new successor states are computed and section numbers are assigned as described above. If the work item stores a result, several cases need to be considered. If it is the result of a state for which a new game has not been played or is not being played in this or any other game, a new game is started from the state. If the result is for a state for which a new game is already being played the process gives up and continues with other work, because the correct result will be forwarded once the new game for that state is finished; care must be taken to ensure that two new games do not wait for results from each other. Results from completed new games are stored in the original store and forwarded to the pending work items (work items waiting for the current result). And results for work items that need the result to compute a conjunction or disjunction of several results will only store and forward the result once the result of the combination is known.

Generating counter examples:

Counter examples (trails) can be generated from the path sections and their branches. For liveness properties the path sections close a cycle. Each branch points to a particular state on another path section. An outline counter example can be generated fast as a sequence of these states. A more detailed counter example can be generated by loading the state at the start of a cycle into the state generator and recursively generate only the successor on the cycle path until the starting state is reached again.

For safety properties the path sections on the path from the initial state to the final state is computed by recursively traversing the parent from which a path section branches until the initial path section is calculated. The same method as in the previous case is used to generate an outline or detailed counter example.

4 Conclusions

We have presented novel strategies for parallelising state graph exploration and model checking for a virtual shared memory parallel machine architecture. First, we presented a parallel state exploration algorithm that uses a combined shared and private queue with minimal synchronisation for efficient and dynamic load balancing. Then we discussed the implementation of a parallel on-the-fly model checking algorithm using the two-queue structure of the parallel exploration algorithm. Preliminary experiments of the parallel model checking algorithm showed that the current implementation of the path table is not memory efficient and needs further refinement. Together with refinement of the cycle detection strategy, planned future work includes experimental analysis of the parallel model checking algorithm using real graphs and investigating other parallelisation strategies for comparison.

In experiments with the parallel state exploration algorithm near linear speedup was obtained over a range of graph types and sizes up to the 16-processor limit. Although experimentation on actual graphs is still essential, owing to the nature of the artificial graphs, we expect the results on state graphs from real problems to be similar. The parallel state exploration algorithm presented in this paper has not only potential for efficient parallel model checking, but can also be effectively used in other applications based on state exploration, e.g. deadlock checking, checking state invariants, and scheduling analysis.

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] S.C. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. *Lecture Notes in Computer Science*, 1253, 1997.

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

[4] D. Bošnački, D. Dams, and L. Holenderski. Symmetric SPIN. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 1–19. Springer–Verlag, 2000.

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

[6] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model checking — how the search order matters. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 216–231. Springer–Verlag, 2000.

[7] 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*, Lecture Notes in Computer Science. Springer–Verlag, April 2002.

[8] E. Brinksma and A. Mader. Verification and optimization of a PLC control schedule. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 73–92. Springer–Verlag, 2000.

[9] O. Bernholtz, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, England, 1999.

[11] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, June 1983.

[12] H. Garavel, R. Mateescu, and I. Smarandache. Parallel state space construction for model-checking. *Lecture Notes in Computer Science*, 2057, 2001.

[13] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Proceedings of the*

12th International Conference on Computer Aided Verification, volume 1855 of *Lecture Notes in Computer Science*. Springer–Verlag, 2000.

[14] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[15] G. J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of FORTE'94*, Bern, Switzerland, October 1994.

[16] C. P. Inggs. Parallelising different model checking paradigms. Continuation report, University of Manchester, December 2000. `http://www.cs.man.ac.uk/~inggsc`.

[17] J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: a protocol validation and verification toolbox. In *Proceedings of the 8th International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440, 1996.

[18] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th International SPIN Workshop*, July 1999.

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

[20] U. Stern and D. Dill. Parallelizing the Murϕ verifier. In *9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*. Springer–Verlag, 1997.

[21] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics*, volume B, pages 135–191. Elsevier Science, 1990.

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

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