Faculty of Informatics, Masaryk University

Partial Order Reduction in Parallel Model Checking

Master's Thesis

Petr Ročkai

Brno, autumn 2009

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference.

Advisor: RNDr. Jiří Barnat, PhD.

Abstract

The main focus of this thesis is a partial order reduction technique for parallel LTL model checking. The novelty of our technique lies in its parallel implementation, and in providing truly dynamic heuristic with serial time complexity in $\mathcal{O}(n)$. In contrast, the traditional DFS-based techniques cannot be used with parallel algorithms. Moreover, the existing heuristics amenable to parallel execution either require super-linear time or otherwise are too strict compared to sequential implementations.

Additionally, we frame the mentioned technique in a broad overview of technology and algorithms for high-performance parallel enumerative LTL model checking. We give both theoretical background and technical details, and provide source code of the system described (comprising a complete model checker suitable for general use).

To augment the partial order reduction technique, an improved on-the-fly algorithm for parallel cycle detection is given, again furthering the state of the art. Importantly, the improved algorithm fits extremely well with the proposed reduction technique.

Keywords

Model Checking, Parallel Algorithms, Implementation, Multi-Threading, Multi-Core, Shared Memory, Symmetric Multiprocessing, Distributed Memory, Clusters, State Space Reductions, Partial Order Reduction

Acknowledgements

Many people have contributed to making this thesis possible. I would like to thank the whole Parallel and Distributed Systems Laboratory – this has been a great place to learn and work. Namely, I would like to thank to RNDr. Jiří Barnat, PhD. (who has also served as an advisor on this thesis) and prof. RNDr. Luboš Brim, CSc. who jointly provided excellent guidance throughout my first encounters with research and also co-authored papers which led to this thesis.

I would also like to thank prof. RNDr. Ivana Černá for both her research work (especially her joint paper with Mgr. Radek Pelánek, PhD. on OWCTY, which this thesis builds on) and her great lectures (especially the complexity course) which had been both a source of knowledge and motivation for me.

Further thanks go to Red Hat, Inc. and namely Tom Coughlan, for making it possible to allocate part of my working hours for my research activities, which directly contributed to this thesis.

I would also like to thank my family, my friends and all the good people around me who helped me in keeping my sanity and focus, and made the time generally enjoyable. Finally, I would like to thank Lucy – I know it has been hard at times, for both of us.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Model Checking . . . 2
- 1.3 Automata-based Approach . . . 2
- 1.4 Parallelisation . . . 3
- 1.5 DiVinE . . . 4
- 1.6 Experiments . . . 5

2 Algorithms **7**
- 2.1 OWCTY . . . 8
- 2.2 MAP . . . 9
- 2.3 On The Fly Execution . . . 11
- 2.4 OWCTY On The Fly . . . 13
- 2.5 Experiments . . . 14

3 Parallel Architectures **17**
- 3.1 Shared Memory Platform . . . 18
- 3.2 MPI . . . 19
- 3.3 Implementing Algorithms . . . 19
- 3.4 Communication . . . 20
  - 3.4.1 Distributed Memory . . . 22
- 3.5 Memory Allocation . . . 22
- 3.6 Termination Detection . . . 24
- 3.7 Experiments . . . 25

4 Partial Order Reduction **29**
- 4.1 Background . . . 29
- 4.2 Related Work . . . 31
  - 4.2.1 Static Partial Order Reduction . . . 32
  - 4.2.2 Dynamic Partial Order Reduction . . . 32
  - 4.2.3 Parallel reductions . . . 33
- 4.3 Cycle Detection . . . 33
- 4.4 Correctness . . . 35
- 4.5 Time complexity . . . 36

4.6 Using with OWCTY 38
4.7 Using with MAP 38
4.8 Experiments 39

5 Conclusion **43**
5.1 Future Work 43

Bibliography **45**

Chapter 1

Introduction

The focus of this thesis is parallel, explicit-state model checking of LTL properties. We will first introduce the concept of model checking, LTL and the surrounding theory. Then, we will discuss why and how is parallelisation important for success of model-checking and what can and cannot be done on parallel machines.

Model checking is an important branch of a bigger family of formal methods. Model checking itself then has a number of finer branches. The most general notion of model checking is that of a model and a property. These two form input to an automated tool, a model checker, which then decides whether the model satisfies the property given. The form of the property and the representation of the model are the basic differences among various model checking approaches. The main concern of this thesis is model checking of Linear Temporal Logic properties on models with explicitly represented states.

We will define these notions, and their alternatives, later in this chapter.

1.1 Motivation

Let us look briefly at general usefulness of model checking, and even the whole branch of formal methods. The systems we design are increasing in complexity, and it is beyond human capacity to verify their correctness. Formal methods exist to augment the human verifier with automated tools that can help them ensure correctness of a system that is far beyond their capabilities to check manually. Of course, various systems need to be "correct" to a different degree. Often, it is acceptable for a machine to fail unexpectedly, and it is enough to do some amount of testing to ensure it works "well enough". However, there are systems where a chance of failure must be minimised at all costs: most notably systems, where failure would put lives at risk.

However, formally speaking, it is not clear what "correct" actually means. In model-checking, this correctness is expressed as a set of properties the system must satisfy. These usually originate in a less formal set of requirements about safety and behaviour of the device or system in question. In other branches of formal methods,

a similar formal notion of "correct" exists. It is (a very important) part of the work of the human verifier to ensure that the informal notion of correctness corresponds to the formal properties used in automated verification.

Nevertheless, formal methods are vastly more expensive to implement than testing. The existing tool support is still mostly academic, and adoption of formal methods in industry is lagging behind other approaches. Despite these issues, formal methods are gaining acceptance and are increasingly relied on for real-world systems. For a recent example, let us just mention [30], a report about replacement of testing with symbolic model checking in the Intel Core i7 processor design.

1.2 Model Checking

In full generality, the term model checking describes simply an automated process of verifying (or falsifying) the fact that a given structure (a model) satisfies a given logical formula. The formula only needs to be specified in a suitable logic: this could be as simple as propositonal logic, although most often, some kind of temporal logic [21] is used in conjunction with model checking. Temporal logics allow to describe behaviour of a system in time – which allows useful statements to be made about hardware or software systems. Indeed, proving (or disproving) properties of software or hardware systems is currently the most common application of model checking.

Of course, there are certain limitations that need to be imposed on the models for the model checking to be practically useful. It is needed that the model is finite, as to be fully constructible by a computer: this may be especially problematic in case of software. Nevertheless, there are important classes of software systems that are finite, and therefore where fully automatic model checking is applicable. Moreover, there are even approaches for model-checking infinite-state systems, but these are out of scope of this thesis – we will only discuss finite models in the following.

Even more specifically, we will concern ourselves with model checking of properties given as Linear Temporal Logic (or LTL, for short) formulae. This is one of the temporal logics that are in current widespread use in model checking: the other being CTL – Computation Tree Logic and CTL* and various subsets of either (in fact, LTL is itself a subset of the latter).

1.3 Automata-based Approach

The contemporary explicit-state LTL model checkers are largely based on a scheme proposed in [47]. The basic idea is to translate negation of the LTL property into a Büchi automaton (which is called a negative claim automaton in this context). This automaton then accepts a language that corresponds to the runs violating the original LTL property. In itself, this automaton will accept many words – however, when a synchronous product is done with the modelled system, we obtain an automaton accepting an intersection of two languages: the one containing all the runs in the original modelled system, and those that violate the desired LTL

property. The model checking problem then reduces to verifying that the product automaton accepts exactly the empty language; moreover, if the language accepted is non-empty, the (infinite) accepted words represent the undesirable behaviours (according to the LTL property used), i.e. a set of counterexamples. Finally, the problem of language emptiness for Büchi automata is a relatively easy problem: the language is non-empty iff there is a reachable accepting cycle in the graph of the automaton.

Moreover, the model specification is usually not given as an explicit state graph of the whole system: this would be rather impractical. The preferred form is a set of generally small extended automata with communication – the full state space is then constructed on-the-fly from this compact description.

Now we can identify the two most resource-demanding portions of the actual LTL model checking: the construction of the full state space from the model, and the search for reachable accepting cycles. In practice, these two processes are often interleaved – the construction of the state space is driven by the demands of the accepting cycle detection algorithm: only the parts that the cycle detection explores are computed, and only when they are needed.

1.4 Parallelisation

Demand for parallel model checkers has been growing steadily ever since it became apparent that parallel computing is the path to further advances in hardware performance. Individual core clock speed has been stagnating and the only segment where significant performance growth is currently happening is by increasing number of compute cores per system.

The state space generation itself is easy to run in parallel: what is usually done is that a set of immediate successors of a given state (a vertex of the state graph) is computed – and it is straightforward to construct any number of these sets at any given time in parallel. However, to exploit this parallelism, we need a accepting cycle detection algorithm that can actually make use of these successor sets in parallel.

Unfortunately, the staple LTL model checking algorithm, Nested DFS [16, 25] is hard, or even impossible, to paralellise efficiently. The algorithm relies on depth-first postorder, which is a $\mathcal{P}$-complete problem [41]. In itself, this problem is being addressed by recent implementations of alternative algorithms [3]. In Chapter 2 we discuss parallel LTL checking algorithms, as well as describe the state of the art algorithm [5] as employed by DiVinE 2 [4, 6].

In distributed memory, there has been a single traditional way of implementing parallel algorithms for explicit-state system model checking. The vertices of the state-space graph, i.e. the individual system states, are partitioned among the available compute nodes statically [12, 14]– often using a hash function on the content of the state vector. This approach yields fair work distribution and relatively predictable overhead, with little variance with shape of the state graph. This is also the basic approach used in shared-memory parallel setting, although more alternatives exist in this design space. The paper [7] examines the effects of using a shared hash table for storing the already-explored part of the state graph, while

employing a dynamic work assignment scheme. The results have been, however, largely disappointing, with static partitioning coming out as a more scalable and more flexible approach.

In Chapter 4, we give a partial order reduction [38, 39, 40, 46] heuristic, suitable for use with these parallel algorithms. The p. o. r. is often used in conjunction with Nested DFS to reduce the amount of time and memory required.

For algorithms that use static partitioning and a large number of partitions (threads, or workstations), existing approaches that treat cross-transitions specially are generally impractical – all transitions are "cross" in these cases. Absence of successful widespread use of this technique so far is in part due to an inherent difficulty in adapting partial order reduction to parallel algorithms: the heuristic used for approximating the partial order reduction is tied to depth-first search order.

In this thesis, we present a new heuristic, allowing p. o. r. to be used with existing parallel LTL model checking algorithms, that is fully independent of the partitioning employed.

1.5 DiVinE

DiVinE is a parallel model checking tool created in the Parallel and Distributed Systems laboratory, Faculty of Informatics in Brno. The original DiVinE [1] was conceived as a tool for clusters of workstations (i.e. a distributed memory system) and was implemented using the Message Passing Interface, a library for distributed memory computations. In 2007, a shared memory branch of DiVinE, under the name DiVinE Multi-Core was created and implemented basically from scratch, using the same modelling language and based on the same model-checking algorithms. This new branch has been more geared towards high performance and speedup on parallel hardware – the primary motivation of original DiVinE has been memory aggregation across a number of low-cost workstations.

In late 2008, it has been decided that a future version of DiVinE (dubbed DiVinE 2) would combine both shared and distributed capabilities, and work has started on a hybrid shared/distributed memory model checker. The project built upon the relatively fresh codebase of DiVinE Multi-Core, although a number of simplifications has been done in the groundwork, since the performance impact of various optimisations was better understood by then. Since the shared memory computation model DiVinE Multi-Core used was based on message passing (with a custom implementation of message queues), it was relatively straightforward to add a thin MPI layer for distributed capabilities.

Besides being a tool intended for production environments, DiVinE 2 has served as a research vehicle for work on efficient implementation of parallel algorithms and various shared-memory-specific techniques.

1.6 Experiments

In Chapters 2, 3 and 4, we include sections with experimental evaluation of the described techniques. In these experiments, we have used a selection of models from `BEEM` [37], and a selection of example models as shipped with the `DiVinE Cluster` distribution. All these models are implemented in the `DVE` [43] language, as used in DiVinE.

Chapter 2

Algorithms

An efficient parallel solution of many problems often requires approaches radically different from those used to solve the same problems sequentially. Among classical examples are list rankings, connected components, and depth-first search in planar graphs.

In the area of LTL model checking the best known enumerative *sequential* algorithms based on accepting cycle detection are the *Nested DFS* algorithm [16, 25] (implemented, e.g., in the model checker `SPIN` [23]) and *SCC-based algorithms* originating in Tarjan's algorithm for the decomposition of the graph into strongly connected components (SCCs) [45]. However, both algorithm types rely on inherently sequential depth-first search postorder. This property of the algorithms makes them difficult to adapt to parallel architectures. The `SPIN` dual-core algorithm is a special case, where the nested (second) search is independent of the outer (first) search. Nevertheless, each of the searches is, in itself, executed serially – therefore, the algorithm cannot be generalised to more than 2 cores. Consequently, different techniques and algorithms are needed.

However, unlike LTL model checking, reachability analysis is a verification problem for which an efficient parallel solution is available. The reason is that the exploration of the state space is independent of the search order. This makes the algorithm easy to implement on parallel architectures with relatively good efficiency out of the box (assuming that efficient parallel primitives for given architecture are correctly employed – we will discuss these in more detail in Chapter 3).

We will discuss the two most successful parallel LTL algorithms available: MAP and OWCTY. First however, we need to introduce more basic notions and algorithms. First, let us define a concise way to describe the input and output of an accepting cycle detection algorithm.

DEFINITION 2.1. *The accepting cycle problem instance M is a tuple (V, E, A, I) where V is a set of vertices (states), $E \subseteq V \times V$ is a set of edges (transitions), $A \subseteq V$ is a set of accepting states and $I \subseteq V$ is a set of initial states.*

ALGORITHM 2.2. Reachability.
Input: $M = (V, E, A, I)$ a problem instance from Definition 2.1.
Output: $V' \subseteq V$ set of states reachable from I

1. $open \leftarrow I$
2. $V' \leftarrow I$
3. While $open \neq \emptyset$ do 4–5
4. $\quad open \leftarrow \{o \mid \forall o \in V.\ x \in open \wedge (x, o) \in E \wedge o \notin V'\}$
5. $\quad V' \leftarrow V' \cup open$
6. Return V'

We will discuss efficient parallel implementation of this simple basic algorithm in later chapters.

2.1 OWCTY

The full algorithm name is **One-Way Catch Them Young** – we write OWCTY for brevity. This algorithm has been introduced for explicit-state model checking in [13]. The algorithm is executed in passes, each of them consisting of a number of steps. The main algorithm loop is as is as follows:

ALGORITHM 2.3. OWCTY.
Input: $M = (V, E, A, I)$ a problem instance from Definition 2.1.
Output: True if no accepting cycles were detected.

1. $S \leftarrow$ Algorithm 2.2 (reachability from I)
2. Repeat 3–4
3. $\quad R \leftarrow$ Algorithm 2.2 (reachability from S)
4. $\quad S \leftarrow$ Algorithm 2.4 (elimination on R)
5. While $R \neq S \wedge S \neq \emptyset$
6. Return $S = \emptyset$

The above scheme basically describes how to convert any simple cycle detection algorithm into an accepting cycle detection one. The usual simple cycle detection algorithm employed here is based on topological sort. The elimination step uses this algorithm to remove all nodes that do not lie on cycles. Of course, the cycle detection algorithm does not discriminate accepting and non-accepting cycles, which is why we need to also exclude states that are not reachable from an accepting state: these clearly cannot lie on an accepting cycle.

The elimination algorithm (implemented using topological sort, as mentioned above) is as follows:

ALGORITHM 2.4. OWCTY Elimination.
Input: S a set of states and T a set of transitions.
Output: A set S' of states that are guaranteed to not lie on cycles in S.

1. $S' \leftarrow S$
2. Repeat 3–4
3. $\quad tail \leftarrow \{t \mid \forall t \in S'.\neg(\exists s \in S'.(s,t) \in T)\}$
4. $\quad S' \leftarrow S' - tail$
5. While $tail \neq \emptyset$

It can be seen that there is a reasonable amount of available parallel work: each state in *tail* can be processed independently of all the others in each iteration. Unfortunately, there is only limited parallelism available across the iteration boundary – we have to wait till all predecessors of a state are processed before we can process the given state.

Looking at serial complexity of the algorithm, we should discuss two cases: a weak graph, and an arbitrary graph. For the weak case, we ought to use a different main loop. In a weak graph, there are no cycles that would contain both accepting and non-accepting states – therefore, non-accepting states can be automatically discarded from cycle detection: only a single pass of reachability needs to be done through non-accepting components (to detect the possible neighbouring accepting components). For the accepting components, we first do a single reachability pass to discover all the vertices belonging to the given component and when this is done, execute a single elimination pass on that component. The component contains an accepting cycle iff the elimination pass returns a set of vertices that is a proper subset of the component's vertex set.

Algorithm 2.5. OWCTY for Weak Graphs.
Input: $M = (V, E, A, I)$ a problem instance from Definition 2.1.
Output: True if no accepting cycles were detected.

1. SCC-decompose the negative claim automaton
2. $R \leftarrow$ Algorithm 2.2 (reachability from I)
3. For each $C \subseteq V$ such that C is an accepting SCC do 4–5
4. $\quad S' \leftarrow$ Algorithm 2.4 (elimination) for C
5. $\quad$ If $S' \neq \emptyset$ then Return False
6. Return True

2.2 MAP

The name of this algorithm is an acronym for **Maximal Accepting Predecessors**. It has been initially designed for distributed memory systems, in [9, 10]. The algorithm is based on the fact that every accepting vertex lying on a cycle is its own predecessor (and this cycle, containing an accepting vertex, is an accepting cycle). An algorithm that is directly derived from this idea would require expensive computation as well as space to store all proper accepting predecessors of all (accepting) vertices. An improvement over that, the MAP algorithm stores only a single representative of all proper accepting predecessor for every vertex, chosen to be *maximal* accordingly to a presupposed linear ordering $\prec$ of vertices (given, for

example, by their memory representation). Clearly, if an accepting vertex is its own maximal accepting predecessor, it lies on an accepting cycle. On the other hand, it can, unfortunately, happen, that all the maximal accepting predecessors lie outside accepting cycles. In that case, the algorithm removes all accepting vertices that were the maximal accepting predecessors of any vertices in the previous pass and recomputes the maximal accepting predecessors. This is repeated until an accepting cycle is found or there are no more accepting vertices in the graph.

ALGORITHM 2.6. Single MAP pass.
Input: $M = (V, E, A, I)$ a problem instance from Definition 2.1.
Output: True an accepting cycles has been detected, $shrink \subseteq A$.

1. For each $v \in V$ do $map(v) \leftarrow 0$
2. push($waiting, I$)
3. While $waiting \neq \emptyset$ do 4–18
4. $u \leftarrow$ pop($waiting$)
5. If $u \in A$ then
6. If $map(u) < u$ then
7. $propagate \leftarrow u$
8. $shrink \leftarrow shrink \cup \{u\}$
9. Else
10. $propagate \leftarrow map(u)$
11. $shrink \leftarrow shrink - \{u\}$
12. Else
13. $propagate \leftarrow map(u)$
14. For each $(u, v) \in E$ do 15–18
15. If $propagate = v$ then Return True
16. If $propagate > map(v)$ then
17. $map(v) \leftarrow propagate$
18. push($waiting, v$)
19. Return False

A single pass may, as outlined above, fail to find an accepting cycle, due to all maximal accepting predecessors lying outside of accepting cycles. To this end, the *shrink* set is maintained throughout the computation. After a given pass, the set contains all maximal accepting predecessors that were found in that pass, and the main algorithm can remove these from the accepting set and start a new pass.

ALGORITHM 2.7. MAP.
Input: $M = (V, E, A, I)$ a problem instance from Definition 2.1.
Output: True if no accepting cycles were detected.

1. $A' \leftarrow A$
2. While $A' \neq \emptyset$ do 3–4
3. If Algorithm 2.6 on $M' = (V, E, A', I)$ then Return False
4. $A' \leftarrow A' - shrink$
5. Return True

The overall time complexity of the algorithm is in $\mathcal{O}(a^2 \cdot m)$, where a is the number of accepting vertices and m is the number of edges. The m factor comes from the relaxation along edges, while one of the a factors comes from the inner pass and the second a comes from the number of outer iterations of the algorithm.

One of the key aspects influencing the overall performance of the algorithm is the underlying ordering of vertices used by the algorithm. Computing the optimal ordering is however difficult to parallelise, hence heuristics for computing a suitable vertex ordering are used.

2.3 On The Fly Execution

In automated verification, parallel techniques both for symbolic and explicit state approaches have been considered. While the symbolic set representations, which often employ canonical normal forms for propositional logic (BDDs, for example), have been a breakthrough in the last decade (with the capacity to handle spaces of the size 10^{20} and beyond), they often turned out to not scale well with the problem sizes. Moreover, the success of their application to a given verification problem cannot be estimated in advance, since no known metrics for the system size have proved to be useful for such estimates. Moreover, the use of BDDs is often sensitive to the used variable ordering, which is sometimes difficult to determine.

For this reason, SAT-based model checking, in particular in the forms of bounded model checking and equivalence checking have recently become very popular. They still benefit from the use of symbolic methods, but tend to be more scalable as they no longer rely on canonical normal forms.

An alternative is the use of explicit state set representations. Clearly, for most real world systems, the state spaces are far too big for a simple explicit representation.

Apart from partial order reduction, another important method for coping with the state explosion problem in explicit state model checking, is the so called *on-the-fly* verification. The idea of the on-the-fly verification builds upon an observation that in many cases, especially when a system does not satisfy its specification, only a subset of the system states need to be analysed in order to determine whether the system satisfies a given property or not. On-the-fly approaches to model checking (also referred to as local algorithmic approaches) attempt to take advantage of this observation and construct new parts of the state space only if these parts are needed to answer the model checking question.

As mentioned in Section 1.3, explicit-state automata-theoretic LTL model checking relies on three procedures: the construction of an automaton that represents the negation of the LTL property (negative claim automaton), the construction of the state space, i.e. the product automaton of system and negative claim automata, and the check for the non-emptiness of the language recognised by the product automaton.

An interesting observation is that only those behaviours of the examined system are present in the product automaton graph that are possible in the negative-claim automaton. In other words, by constructing the product automaton graph the

system behaviours that are not relevant to the validity of the verified LTL formula are pruned. As a result, any LTL model checking algorithm that builds upon exploration of the product automaton graph may be considered on-the-fly. We will denote such an algorithm as Level 0 on-the-fly algorithm in the classification below.

When the product automaton graph is constructed, an accepting cycle detection algorithm is employed for detection of accepting cycles in the product automaton graph. However, it is not necessary for the algorithm to have the product automaton constructed before it is executed. On the contrary, the execution of the algorithm and the construction of the underlying product automaton graph may interleave in such a way that new states of the product automaton are constructed *on-the-fly*, i.e. when they are needed by the algorithm. If this is the case, the algorithm may terminate due to the detection of an accepting cycle before the product automaton graph is fully constructed and all of its states are visited.

Those LTL model checking algorithms that may terminate before the state space is fully constructed are generally considered on-the-fly. If there is an error in the state space (an accepting cycle), an on-the-fly algorithm may terminate in two possible phases: either an error is found before the interleaved generation of the product automaton graph is complete (i.e. before the algorithm detects that there are no new states to be explored), or an error is found after all states of the product automaton have been generated and the algorithm is aware of it. The first type of the termination is henceforward referred to as *early termination* (ET). Note that the awareness of completion of the product automaton construction procedure is important. If the algorithm detects the error by exploring the last state of the product automaton graph before it detects that it was actually the last unexplored state of the graph, we consider this to be an early termination.

We classify "on-the-flyness" of accepting cycle detection algorithms according to the capability of early termination as follows. An algorithm is

- *level 0 on-the-fly algorithm*, if there is a product automaton graph containing an error for which the algorithm will never early terminate.
- *level 1 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm may terminate early, but it is not guaranteed to do so.
- *level 2 on-the-fly algorithm*, if for all product automaton graphs containing an error the algorithm is guaranteed to early terminate.

Note that level 0 algorithms are sometimes considered on-the-fly and sometimes not, depending on research community. Since a level 0 algorithm explores full state space of the product automaton graph it may be viewed as if it does not work on-the-fly. However, as explained above, just the fact that the algorithm employs product automaton construction is a good reason for considering the whole procedure of LTL model checking with a level 0 algorithm as an on-the-fly verification process.

To give examples of algorithms with appropriate classification we consider algorithms OWCTY, MAP, and Nested DFS. OWCTY algorithm is level 0 algorithm, MAP algorithm is level 1 algorithm and Nested DFS is level 2 algorithm.

As for the state space exploration algorithms, the efficiency of the on-the-flyness of the algorithm may also be improved by other techniques. It might be the case

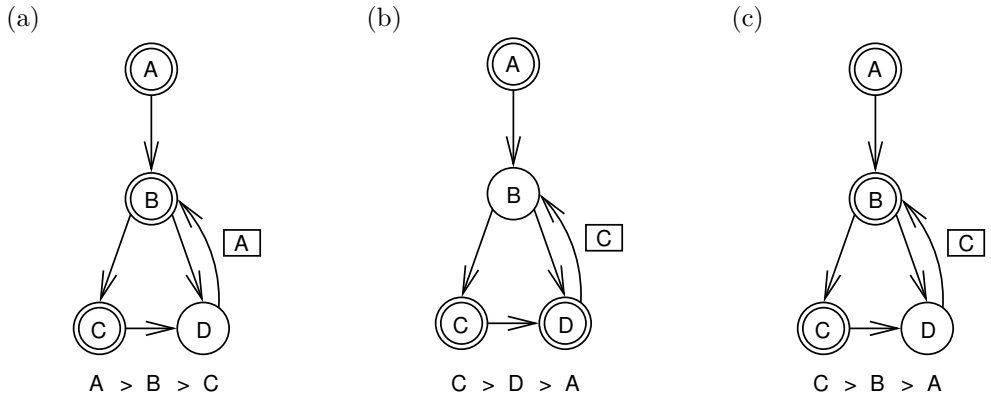

Figure 2.1: Three scenarios where no accepting cycle will be discovered using accepting state propagation. a) Maximal accepting predecessor is out of the cycle. b) There is no fresh path back to the maximal accepting state. c) Wrong order of propagation, $C \rightarrow D$ is explored before $B \rightarrow D$, hence, C is propagated from D.

that even the level 2 on-the-fly algorithm fails to discover an error, if the examined state space is large enough to exhaust system memory before an error is found. This issue has been addressed by methods of directed model checking [18, 19, 20], which combines model-checking with heuristic search. The heuristic guides the search process to quickly find a property violation so that the number of explored states is small. It is worthy to note that our approach can be extended with directed search as well.

2.4 OWCTY On The Fly

The idea of propagating one accepting predecessor along all newly discovered edges is at heart of a heuristic extension of OWCTY [5]. If the propagated accepting state is propagated into itself, an accepting cycle is discovered and the computation is terminated. Like with the MAP algorithm, an accepting state to be propagated is selected as a maximal accepting state among all accepting states visited by the traversal algorithm on a path from the initial state of the graph to the currently expanded state. Since the INITIALISE phase of OWCTY needs to explore full state space, we can employ it to perform limited accepting cycle detection using maximal accepting state propagation. Unlike the MAP algorithm, we however avoid any re-propagation to keep the INITIALISE phase complexity linear in the size of the graph. This means that some accepting cycles that would otherwise be discovered (i.e. with relaxation, or re-propagation, enabled) may be now missed. In particular, there are three general reasons for not discovering an accepting cycle with the proposed heuristic. First, the maximum accepting predecessor of the cycle may not lie on the cycle itself, see Figure 2.1(a). Second, the maximum accepting predecessor

value does not reach the originating state due to the absence of a fresh path (path made of yet unvisited states), see Figure 2.1(b). And third, the maximum accepting predecessor value does not reach the originating state due to a wrong propagation order, see Figure 2.1(c).

When the algorithm encounters an accepting state that is being propagated, it terminates early, producing a counter-example. On the other hand, if the INITIALISE phase (i.e. the first reachability) of OWCTY fails to notice an accepting cycle, the rest of the original OWCTY algorithm is executed. Either the algorithm finds an accepting cycle (and again, produce a counter-example) or, it proves that there are no accepting cycles in the graph.

2.5 Experiments

We have picked 90 models with invalid properties to assess the success rate of the on-the-fly heuristic. We only present a simple overview of the results, for full evaluation, please refer to [5]. We present the results in this chapter, since these are more pertinent to the on-the-fly heuristic itself. For experimental evaluation of parallel implementations of the algorithms themselves, please refer to Section 3.7. We have measured the number of visited states, amount of memory used and time required for verification. The ET ratio represents the number of models where early termination has happened (out of all models).

As with parallelism for the case of valid properties, we can see that the on-the-fly heuristic can save significant amount of time when the properties do not hold. Unfortunately, it is still not on par with the sequential Nested DFS algorithm and it is not clear whether the situation can be improved further.

Algorithm	Visited states	Memory	Time	ET ratio
BFS, full	52 047 342	6 712 MB	760 s	0/90
BFS, on-the-fly	23 157 474	4 858 MB	295 s	66/90
DFS, full	52 047 342	6 716 MB	760 s	0/90
DFS, on-the-fly	19 849 655	4 583 MB	272 s	56/90
Nested DFS	622 984	1 736 MB	7 s	90/90

Figure 2.2: Single core experiments.

Algorithm	Visited states	Memory	Time	ET ratio
BFS	6 820 499	2 829 MB	40 s	66/66
DFS	3 930 520	2 257 MB	23 s	56/56
Nested DFS	622 984	1 736 MB	7 s	90/90

Figure 2.3: Single core experiments restricted to runs with early termination.

Algorithm	Visited states	Memory	Time	ET ratio
BFS, 1 thread	23 157 474	4 858 MB	295 s	66/90
BFS, 2 threads	17 203 306	5 748 MB	130 s	74/90
BFS, 3 threads	20 244 429	6 955 MB	122 s	74/90
BFS, 4 threads	18 632 114	7 576 MB	102 s	72/90
DFS, 1 thread	19 849 655	4 583 MB	272 s	56/90
DFS, 2 threads	18 996 947	5 890 MB	136 s	77/90
DFS, 3 threads	22 826 318	7 037 MB	138 s	73/90
DFS, 4 threads	18 833 201	7 685 MB	100 s	72/90

Figure 2.4: Experiments involving various configurations of the algorithm and various number of CPU cores.

Chapter 3

Parallel Architectures

In this chapter, we will describe the hardware systems which our parallel algorithms and techniques target. Parallel hardware, as is nowadays widely known, brings a completely new set of problems to tackle. Many of the existing algorithms are unsuitable for parallel execution, and the traditional implementation techniques are often inappropriate and inadequate. Even though steady advances in understanding and efficiently leveraging parallelism are being made, there is still a large set problems that are notoriously difficult to implement efficiently. These problems are characterised by high number of dependencies in their data flows: which is, e.g. the case of exploring arbitrary graphs.

To further complicate the situation, the optimal solution to the LTL model checking problem relies on depth-first search [25], for which the data flow dependencies form a linear chain, leaving no room for parallelisation at all. We have discussed alternative algorithms for this problem in the previous chapter: nevertheless, none of them is fully optimal and, obviously, all of them have at least as many data-flow dependencies as has plain state space exploration.

On the other hand, even though the nature of the data flows in the algorithms employed, there is enough independence to carry out a high number of parallel tasks. Unfortunately, the dependencies – whenever they cross a boundary of a single thread of execution – translate into communication, which comes at an expense in parallel performance, due to synchronisation delays and also due to inter-thread or inter-process communication simply being more laborious than intra-thread data flow.

In the following text, we will focus on two kinds of parallel architectures. One of those are shared memory systems, either the currently ubiquitous multi-cores or the more traditional multi-CPU machines in an SMP configuration. The other are distributed-memory systems, i.e. clusters of conventional uniprocessor or SMP machines, or even more complicated NUMA systems – as long as they support MPI [32], a standard for message passing.

On the level of symmetric multiprocessor systems, we employ a threading model – one where threads share all memory, possessing separate stacks in the shared address space and a special thread-local storage to store thread-private data. Specifi-

cally, we use the threading model as specified by the POSIX Threads standard [28].

Contrast this with `SPIN` 5.1, which is based on multi-processing and inter-process shared memory (and which has no distributed memory capabilities).

3.1 Shared Memory Platform

Critical Sections, Locking and Lock Contention. In a shared memory setting, access to memory, that may be used for writing by more than a single thread, has to be controlled through use of mutual exclusion, otherwise, race conditions will occur. This is generally achieved using a "mutual exclusion device", so-called mutex. A thread wishing to enter a critical section has to lock the associated mutex, which may block the calling thread if the mutex is locked already by some other thread. An effect called resource or lock contention is associated with this behaviour. This occurs, when two or more threads happen to need to enter the same critical section (and therefore lock the same mutex), at the same time. If critical sections are long or they are entered very often, contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

Processor Cache: Locality and Coherence. There are currently two main architectures in use for Level 2 cache. One is that each processing unit has its completely private Level 2 cache (for the Symmetric Multiprocessing case) or there is a shared Level 2 cache for a package of 2 cores. In bigger shared memory computer systems, it is usual to encounter split cache, since they often contain on the order of 8-64 cores attached to a single memory block. In recent hardware, the basic building units are dual-core CPUs with shared cache, but among the different units, the caches are still separate.

Due to coherence requirements, read from thread B subsequent to write from thread A may (and usually does) incur a significant penalty. Since the cache often works with smallest units of 128 or more bytes long (called cache lines), various pieces of data may share a single cache line, if they are adjacent in memory. If different threads access a single cache line often (even though the variables they use may be disjoint – and in this case, we speak of false sharing), the performance of the computation may suffer dramatically.

Of course in multiprocessors, it is the case – like in uniprocessors – that access to data in L1 or L2 cache is many times faster than access to the main RAM. Therefore, it is important that the programs are written in such a way that the data they access is kept closely together in the main memory, so that memory prefetch can work efficiently to hide the very high RAM latencies. In multiprocessors, this is further complicated by the above-mentioned coherence issues: while data for any given thread should be as localised as possible, the data in use by different threads should be far apart in the address space.

A more detailed study of these phenomena in the context of shared-memory multiprocessors may be found in e.g. [44].

3.2 MPI

We employ MPI-based message passing for communicating in distributed memory environment. The MPI standard defines a number of modes of communication for groups of processes distributed over a cluster. Unfortunately, the architecture of MPI favours distributed applications with limited task interdependence and therefore with limited communication. The problems that have known divide-and-conquer solutions are therefore well-suited for MPI – which is however not the case of model checking, nor accepting cycle detection.

For parallel state space exploration and accepting cycle detection, the most important primitive is point-to-point messages. Even though MPI provides a number of collective communication primitives, the ones that we needed are either out of scope for MPI (distributed termination detection) or are trivially implemented in terms of point-to-point messaging (broadcast). Therefore, only a very small subset of MPI is used in our tool: initialisation and point-to-point messaging (send, receive and probe – the latter being a nonblocking check for incoming messages).

The main problem with naïve use of MPI is that sending individual messages is prohibitively expensive in MPI: nevertheless, the parallel state space exploration techniques usually rely (and this is also the case of our tool) on sending a vast number of messages: each transition among vertices belonging to a different workers (a so-called cross transition) produces a message: usually with higher numbers of workstations, all of the transitions in the system are cross. This amounts to millions of messages for models of even moderate size. Fortunately, these messages are all asynchronous, which allows certain optimisations that make it possible to use MPI more efficiently. It is important to combine many consecutive messages into a single MPI message is in obtaining good performance for this kind of workload.

3.3 Implementing Algorithms

The considerations laid out in previous paragraphs bring us to the actual algorithm implementation and the associated techniques we came up with. The presented shared-memory technology is designed to reduce communication overhead, exploiting traits of shared memory systems that are not available in distributed memory environments. Consequently, the main goal is to improve scalability of the implementation, which is inversely proportional to communication overhead and its growth with increasing number of threads. However, it was one of our goals to keep the overall architecture of the shared memory subsystem similar to the distributed memory model – which turned out to be instrumental in a successful extension of the system to a cluster of multiprocessors (i.e. a hybrid shared/distributed memory architecture).

Were we to venture into a strictly shared memory implementation, one may pose a question, whether a different approach of using a standard serial algorithm modified to allow parallelisation at lower levels of abstraction would give a scalable, efficient program for multi-CPU and/or multi-core systems. Our efforts at extracting such a micro-parallelism in our codebase have been largely fruitless, due

to high synchronisation cost relative to the amount of work we were able to perform in parallel. An example of such micro-parallel approach would be to implement a parallel successor generator, where there is certain independence of the sub-tasks per a single processed state involved, but these sub-tasks are very small and on current hardware, it is faster to perform them in sequence than it is in parallel.

As mentioned in 1.4, an approach with dynamic state space partitioning, which would be available to a purely shared memory implementation, does not bring convincing advantages over the static partitioning model – a topic more thoroughly covered in [7].

In the following sections, we explore the possibilities to build on the existing distributed memory approaches, in the vein of statically partitioned graphs, for the cases where memory locality can be leveraged.

3.4 Communication

Generally, in a distributed computation, all communication is accomplished by passing messages – e.g. using a library like MPI for cluster message passing. However, in communication-intensive programs, or those sensitive to communication delay (latency), using general-purpose message passing naïvely may be quite inefficient.

In shared memory, most of the communication overhead can be eliminated by using more appropriate communication primitives. We have opted for a high-performance, contention- and lock- free FIFOs (First In, First Out queues). We have adopted a variant of the two-lock algorithm[1] – a compromise between performance on one hand and simplicity and portability on the other – presented in [34]. Our modifications involve improved cache-efficiency and only using a single write-lock, instead of a pair of locks, one for reading and one for writing, since there is ever only one thread reading, while there may be several trying to write.

Definition 3.1. *A FIFO of T is an aggregate data type with following items:*

- *buffer: array of T*
- *next: pointer to Node*
- *read*, *write: integer*
- *nodeSize: integer (size of buffer)*
- *head*, *tail: pointer to Node*
- *writeLock: mutex*

Algorithm 3.2.
Input: f is a FIFO of T instance, x of type T is an element to enqueue
Output: Modified FIFO f, which contains x as its last element

1. lock(*f.writeLock*)

[1] The `C++` implementation of Algorithm 3.2 and Algorithm 3.3 can be found in the file `divine/fifo.h` in a DiVinE 2 source distribution.

2. If $f.tail.write = f.nodeSize$ then
3. $\quad t \leftarrow$ newly allocated Node, all fields 0
4. Else
5. $\quad t \leftarrow f.tail$
6. $t.buffer[t.write] \leftarrow x$
7. $t.write \leftarrow t.write + 1$
8. If $f.tail \neq t$ then
9. $\quad f.tail.next = t$
10. $\quad f.tail = t$
11. unlock($f.writeLock$)

Algorithm 3.3.
Input: f, a non-empty FIFO instance
Output: Modified FIFO f, with its first element dequeued and the formerly first element of f.

1. If $f.head.read = f.nodeSize$ then
2. $\quad f.head \leftarrow f.head.next$
3. $f.head.read \leftarrow f.head.read + 1$
4. Return $f.head.buffer[f.head.read - 1]$

Originally, every thread involved in the computation owned a single instance of the FIFO and all messages for this thread are pushed onto this single queue (there comes the need for the write lock). However, in communication-intensive workloads (like our parallel model checking algorithms), the write lock has been observed to be a point of contention, creating a bottleneck even when only 4 CPU cores were involved.

Although there is also a completely lock-free design described in [34]), we have opted for a matrix-style communication primitive, with a private FIFO for each pair of communicating threads. This is partially motivated by the use of atomic compare and swap instructions in the lock-free queue design, which are relatively expensive compared to regular memory access. Moreover, even when lock contention is removed, the lower level contention for a single memory location on the CPU level is likely to be a reason of concern. Moreover, for our use-case, all of the mentioned issues can be addressed by using a larger number of queues without incurring any significant penalties.

The correctness, linearizability (atomicity) and liveness proofs as given in [34] are straightforwardly adapted to our implementation and thus left out.

Alternatives to our implementation, which may be more appropriate in different settings, include a ring-buffer FIFO implementation (if there is a bound on the amount of in-flight data known beforehand, the ring-buffer implementation may be more efficient) and possibly an algorithm based on swapping incoming and outgoing queues (which could be easily implemented as a pointer swap). The latter gives results comparable to the described FIFO method, although the code and locking behaviour is much more complex and error-prone, which made us opt for the simpler FIFO implementation.

3.4.1 Distributed Memory

In distributed memory, as previously mentioned, MPI is employed for message passing. To simplify the system, the communication interface only exposes shared-memory FIFO queues even for pairs of threads executing across different MPI nodes, in both directions. A dedicated MPI thread then bridges those queues, by moving messages from outgoing queues on a node to incoming queues on remote nodes, using MPI messages. As outlined, to make the usage of MPI more efficient, queued messages are combined before being passed to MPI. When a given queue is about to be transferred, a linear buffer is allocated and all messages available in the queue at this time are copied to this linear buffer and dispatched as a single MPI message. The MPI control thread on the receiving side is then responsible for dismantling this large message into its individual components before filling in the respective incoming queue on that node.

Even though this technique in no way reduces the amount of data transferred, it reduces the overhead incurred per MPI message (i.e. the number of MPI messages sent by the system) by as much as a factor of 1000 (due to limited throughput of MPI-related processing and due to round-robin handling of the individual queues, it is common that thousands of messages accumulate in a queue before MPI control passes over it a second time). The runtime costs, as opposed to savings, seem to be very modest, although these could be possibly further reduced by using scatter-gather IO[2] instead of a linear buffer and an explicit copy. This is unfortunately not supported by MPI as of this writing.

3.5 Memory Allocation

In a distributed computation, every process has simply its own memory which it fully manages. In a shared memory, however, we prefer to manage the memory as a single shared area, since an equal partitioning of available memory and separate management may fall short of efficient resource usage. Even more importantly, managing large amount of interprocess shared memory comes with technical limitations and provides only very minor advantages. However, thread-based approach also poses some challenges, especially in allocation-intensive environment like ours.

Efficient allocation and deallocation routines. Since the workload we are facing facilitates large amounts of fixed-size allocations and deallocations, it appears natural, to implement tailored allocation and deallocation routines.

A very simple $\mathcal{O}(1)$ memory pool has been devised[3], optimised for many allocations of a limited set of sizes. Of course, from time to time it needs to obtain memory from the system and this operation is not constant-time, however it is at most linear in the block size and therefore amortises over the individual allocations as well (given a fixed allocation size).

[2] As defined by POSIX.1-2001 [29], functions `writev` and `readv`.

[3] The implementation of the memory pool structure and allocation and deallocation primitives, together with the memory stealing routine, may be found in the file `divine/pool.h` of the DiVinE 2 source distribution.

Each thread has its own private pool and therefore the implementation is lock-free. This is possible since most operations are thread-local: the remaining case of cross-thread deallocation is discussed below.

Concurrent allocation and deallocation. First, a naïve approach of protecting the allocation routines with a simple critical section is highly prone to resource contention. Fortunately, modern general-purpose allocator implementations refrain from this idea and have a generally non-contending behaviour on allocation. However, releasing memory back for reuse is more complex to achieve without introducing contention, in a setting where it is often the case that thread other than the one allocating the chunk needs to release it.

There are known general-purpose solutions to the problem of concurrent deallocation, e.g. [33], however they are currently not in widespread use in general-purpose allocators. Therefore, whenever relying on the system allocator, we have to refrain from the above-mentioned pattern of releasing memory from different than allocating thread, in order to avoid contention and the accompanying slowdown.

The message-passing implementation we employ is pointer-based – in other words, the message sent is only a pointer and the payload (actual interesting message content) is allocated on the shared heap, and it may be either reused or released by the receiving thread. Observe however, that releasing the associated memory in the receiving thread will lead to the above-mentioned undesirable deallocation pattern.

Therefore, the allocation routines handle these cases differently. Instead of manipulating the memory pool of a different thread, the memory (allocated in a different thread) is appended to the freelist of the current thread. Thanks to the workload distribution scheme employed, this approach is quite feasible and does not introduce significant memory overhead. Moreover, it is correct, since the memory handed over to the new thread is never again examined by the original thread. We have dubbed the technique "memory stealing", since the releasing thread "steals" (without communicating this fact to the original thread in any fashion) the memory from its previous owner.

General purpose memory allocation. Since apart from the state allocation and deallocation, there are several important memory-intensive routines (one example being the FIFOs, another is the model parser and interpreter) in the model checker, which do not exhibit the behaviour described above, we also need a high-performance, general-purpose memory allocator. Moreover, the pool allocator described needs to obtain memory blocks somewhere as well.

We have opted for Emery Berger's excellent `HOARD` multi-threaded memory allocator [8]. Apart from having very good performance and scalability properties, `HOARD` strives to avoid heap layouts leading to false sharing, further improving performance.

3.6 Termination Detection

Compared to our earlier work [3, 42], a new termination detection algorithm[4] has been designed and implemented for DiVinE 2. The algorithm is based on a similar concept as the previous design, using mutual exclusion provided by the low-level threading library as the basic implementation block. The importance of using this mechanism lies in the resource usage: an idle thread under this scheme, thanks to the implementation of mutual exclusion, does not consume computational resources of the system. This makes over-committing on the number of threads (versus physical processors) possible. Additionally, this makes resource sharing with other tasks on a given multiprocessor system more practical.

The idleness requirement is even more important in DiVinE 2 than it was previously, since MPI communication is handled by a separate thread, which does not require a dedicated CPU core, but should instead take advantage of any idle cores in the system, or otherwise time-slice with the remaining (worker) threads.

Algorithm 3.4. Idleness Check.

Input: The thread X that became idle, S set of all threads.
Output: True iff the system has terminated.

1. lock global mutex
2. $R \leftarrow \emptyset$
3. *done* $\leftarrow$ true
4. For each thread $T \in S$ do 5–8
5. If $T \neq X \wedge$ non-blockingly acquired per-thread mutex of T then
6. $R \leftarrow R \cup \{T\}$
7. Else
8. *done* $\leftarrow$ false
9. if any thread in R has work waiting, set *done* to false
10. unlock all per-thread mutexes for S
11. If *done* then
12. *alldone* $\leftarrow$ true
13. wake up all threads
14. Else
15. wake up all threads with work waiting
16. go to sleep, releasing both per-thread mutex for X and the global mutex
17. upon wakeup, reclaim the per-thread mutex atomically
18. reclaim the global mutex
19. release global mutex
20. Return *alldone*

On the level of distributed computation, i.e. MPI, another layer of termination detection has to happen. Since a separate thread exists for handling MPI, this thread is added to the termination detection group. The MPI thread then uses

[4] The `C++` implementation of the termination detection algorithm can be found in the file `divine/barrier.h` in DiVinE 2 source distribution.

a modified version of the Algorithm 3.4 to check that it is the only active thread on a given MPI node. When this is true, the MPI thread executes a distributed termination detection using a straightforward modification of Safra's algorithm [17]. Whenever an MPI node is active, the termination fails. When all nodes are passive (their local termination detection has finished), the MPI queues are all empty and the message counts in the system are stable, the system is terminated after a second round of the distributed algorithm is done successfully.

Algorithm 3.5. Distributed Idleness Check.
Input: The MPI node that has become idle (all local threads).
Output: True iff the system has terminated.

1. $one \leftarrow$ accumulate message counts in the system, setting $(0, 1)$ in case any MPI node is found to be busy
2. Return false If $\mathrm{fst}(one) \neq \mathrm{snd}(one)$
3. $two \leftarrow$ accumulate message counts in the system
4. If $\mathrm{fst}(one) = \mathrm{fst}(two) \wedge \mathrm{fst}(two) = \mathrm{snd}(two)$ then
5. notify all nodes of termination
6. Return result of Algorithm 3.4
7. Return false

3.7 Experiments

We have performed a number of experiments, aimed at assessing scalability and performance of our model checker. Moreover, we have evaluated the impact of some of the individual techniques presented above.

In first set of experiments, we have used a number of blades, with 4 Intel Xeon 5130 cores, each running at 2 GHz, with 16 GB of RAM per blade, interconnected using switched gigabit ethernet. The models come from `BEEM`. The results are available in Tables 3.1 and 3.2 and Figures 3.1 and 3.2. From comparing the two models, it can be clearly seen that scalability depends on the model chosen.

On the leader model, it can be seen that in shared memory, the scalability is superior when compared with MPI: for this model, in shared-memory setting, we get fair speedup of almost 3 or 36 % – using 4 CPU cores. However, using 1 core per machine on 4 machines, we only get only speedup of about 1.26, i.e. 79 % – with 3 machines, which seems to be the local optimum, we get 1.45 or 68 % – feeble, compared to shared memory.

Nevertheless, with the anderson model, the difference is far from this dramatic: again for the 4 core case, we get speedup of 2.9, or 35 % but with MPI, this time (1 core per machine) we achieve quite reasonable 2.5 or 40 %.

Another set of experiments was done on a modern shared-memory machine, sporting 16 Intel Xeon E5520 cores, each running at 2.27 GHz, with 24 GB of RAM. The results are displayed in Table 3.3. We can see that there's again certain variation depending on the model. We can also see that scalability is better in shared memory even with larger number of cores than 4 – despite the individual core being

faster, we get better speedup with 16 cores fully sharing memory than we get with 21 cores distributed among 7 machines.

In part, the disparity between distributed and shared memory performance is certainly due to more mature and better optimised implementation of the queues used for shared-memory communication – the MPI-based communication implementation is much newer and therefore has seen much less fine-tuning. Nevertheless, the overhead is clearly higher in the MPI case (and there are, of course, good reasons for this). We believe that this further validates the appropriateness of treating shared memory specially and not simply running a number of MPI processes per node.

Interestingly enough, running with MPI processes instead of shared memory queues and using the anderson model, there were only minor differences, and sometimes the fully-MPI version came out better – however, it is hard to interpret these results, since the fully-MPI-based version, when using 2 worker threads per machine loads all the 4 cores of the machine fully, due to the MPI handler running in a separate thread, which means it actually fully employs 4 cores per machine even though only 2 of the cores are executing the state space exploration algorithm.

Again, only a general survey of the overall performance is given in this thesis. For a more complete evaluation, including measurements of contributions of individual mentioned shared-memory techniques may be found in [2]. A detailed study of performance impact of various implementation approaches for MPI-based distributed model checking has not been published so far, to our knowledge.

N	1 thread		2 threads		3 threads		4 threads	
1	141.6 s	100 %	85.2 s	60 %	61.3 s	43 %	49.2 s	**35 %**
2	94.7 s	67 %	54.3 s	38 %	46.0 s	**32 %**	58.3 s	41 %
3	70.0 s	49 %	40.4 s	**29 %**	41.8 s	30 %	46.3 s	33 %
4	56.4 s	40 %	37.8 s	**27 %**	38.4 s	**27 %**	42.2 s	30 %
5	44.9 s	32 %	35.2 s	**25 %**	34.8 s	**25 %**	35.0 s	**25 %**
6	37.2 s	26 %	32.1 s	23 %	30.7 s	**22 %**	32.8 s	23 %
7	31.9 s	23 %	27.6 s	**19 %**	30.1 s	21 %	32.0 s	23 %

Table 3.1: Reachability, `anderson.6`, 32-bit, cluster.

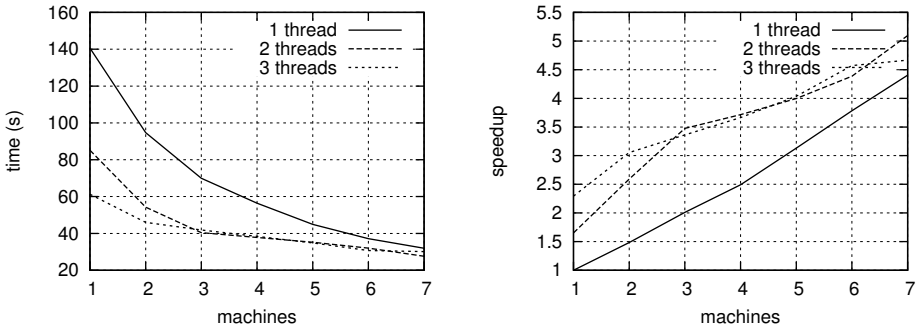

Figure 3.1: Reachability, `anderson.6`, 32-bit, cluster.

N	1 thread		2 threads		3 threads		4 threads	
1	91.9 s	100.0 %	57.8 s	62.9 %	43.7 s	47.6 %	33.8 s	**36.8 %**
2	66.3 s	72.1 %	52.6 s	57.2 %	52.0 s	56.6 %	64.9 s	70.6 %
3	62.5 s	68.0 %	64.6 s	70.3 %	68.8 s	74.9 %	79.1 s	86.1 %
4	72.2 s	78.6 %	63.0 s	68.6 %	61.2 s	66.6 %	72.9 s	79.3 %

Table 3.2: Reachability, `leader_election.5`, 32-bit, cluster.

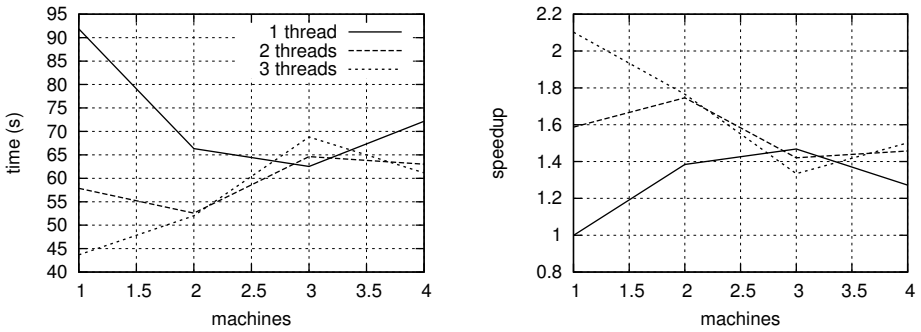

Figure 3.2: Reachability, `leader_election.5`, 32-bit, cluster.

N	peterson		leader		anderson	
1	702.5 s	100 %	66.5 s	100 %	91.6 s	100 %
2	438.9 s	62 %	42.8 s	64 %	56.9 s	62 %
4	260.0 s	37 %	31.6 s	37 %	34.2 s	37 %
8	148.7 s	21 %	14.5 s	22 %	18.5 s	20 %
16	116.7 s	17 %	13.0 s	20 %	14.1 s	15 %

Table 3.3: Reachability, various models, 64-bit, shared memory.

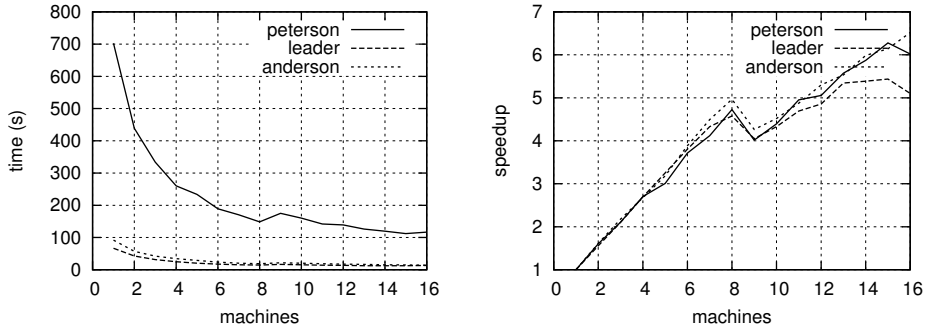

Figure 3.3: Reachability, various models, 64-bit, shared memory.

N	lamport		leader		anderson	
1	868.0 s	100 %	221.0 s	100 %	431.0 s	100 %
2	534.9 s	62 %	137.1 s	62 %	268.5 s	62 %
4	321.1 s	37 %	78.0 s	35 %	159.3 s	38 %
8	180.8 s	21 %	61.2 s	28 %	88.9 s	21 %
16	138.3 s	16 %	45.2 s	20 %	69.3 s	16 %

Table 3.4: OWCTY, various models, 64-bit, shared memory.

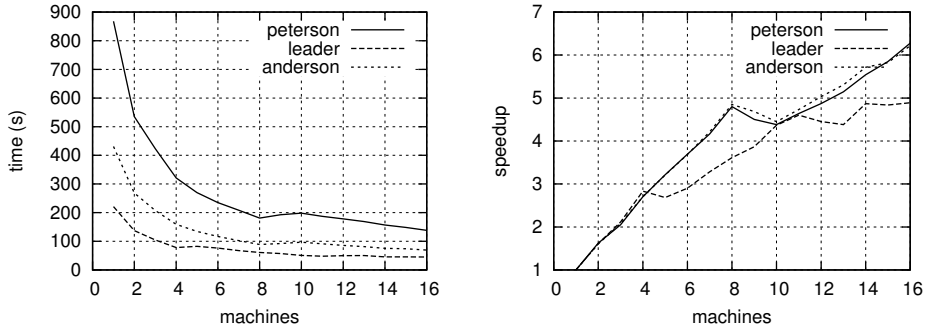

Figure 3.4: OWCTY, various models, 64-bit, shared memory.

Chapter 4

Partial Order Reduction

4.1 Background

The traditional dynamic partial order reduction is based on an efficient heuristic approximation of the full (theoretical) reduction. In this section, we shortly present the theoretical concept and the usual heuristics.

DEFINITION 4.1. *A Kripke structure is a tuple* (S, T, S_0, L) *where*

- S *is a set of states,*
- T *is a set of transitions* ($\forall \alpha \in T : \alpha \subseteq SxS$)*,*
- $S_0 \in S$ *is an initial state,*
- $L : S \to 2^{\mathcal{AP}}$ *is a labelling function, with* $\mathcal{AP}$ *being a set of atomic propositions.*

For simplicity, we will only consider deterministic finite systems. Each $\alpha \in T$ can therefore be seen as a partial function $\alpha : S \to S$. Practically, the extension to non-deterministic systems does not affect the results of this thesis.

DEFINITION 4.2 (Enabledness). *A transition* α *is enabled in a state* s*, whenever* $\alpha(s)$ *is defined.*

The idea of partial order reduction is to disable some transitions in some of the states, obtaining a new structure K', such that for a fixed $\text{LTL}_{-\text{x}}$[1] formula φ, it holds that $K \models \varphi \iff K' \models \varphi$. The reduced system K' is defined through so-called *ample sets*. For each state $s \in K$, we define $ample(s) \subseteq enabled(s)$ to be the set of transitions enabled in the reduced system.

Apart from requiring correctness (the system defined through those ample sets satisfies φ iff the original system does), two properties are crucial for successful application of the reduction:

[1] By $\text{LTL}_{-\text{x}}$, we mean Linear Temporal Logic without the X (next) operator. Please refer to [15] for full definition of LTL and $\text{LTL}_{-\text{x}}$.

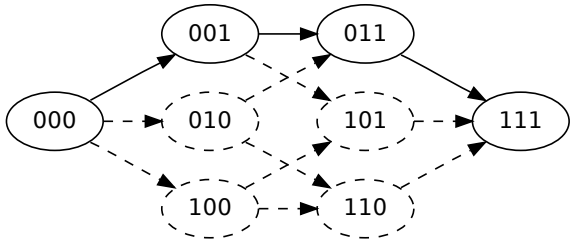

Figure 4.1: An example of p. o. r. – all the dashed vertices and edges may be left out of the explored state space, assuming that the transitions are all invisible and independent.

1. The ample sets need to be efficiently obtainable from description of the original system.
2. The reduction achieved needs to be significant, that is, the reduced system should be significantly smaller than the original.

DEFINITION 4.3 (Independence). *Let α, β transitions, we say that α is independent of β iff:*

(i) $\forall s : \alpha \in \mathit{enabled}(s) \implies \alpha \in \mathit{enabled}(\beta(s))$

(ii) $\forall s : \alpha(\beta(s)) = \beta(\alpha(s))$

DEFINITION 4.4 (Invisibility). *We say that transition α is invisible with respect to $\mathcal{AP}'$ iff it holds that $\forall s \in S : L(s) \cap \mathcal{AP}' = L(\alpha(s)) \cap \mathcal{AP}'$.*

When we refer to invisibility later in this thesis, we always refer to invisibility with respect to the alphabet of formula φ.

Traditionally, these four conditions are used to determine a suitable ample set for each state s:

C0 $\mathit{ample}(s) = \emptyset \iff \mathit{enabled}(s) = \emptyset$

C1 Along every path in the original structure K that starts in s, the following condition holds: a transition that is dependent on a transition in $\mathit{ample}(s)$ cannot be executed without a transition in $\mathit{ample}(s)$ occuring first.

C2 If s is not fully expanded, then every $\alpha \in \mathit{ample}(s)$ is *invisible*.

C3 (cycle proviso) A cycle in reduced structure is not allowed if it contains a state in which some transition is *enabled*, but is never included in *ample*(s) for *any* state s on the cycle.

LEMMA 4.5. *The conditions **C0** through **C3** are sufficient to guarantee obtaining correct ample sets.*

PROOF. For a complete proof, please refer to [15]. □

Conditions **C0** and **C2** are easily checked locally, and therefore their execution can be left intact for purposes of a parallel implementation of partial order reduction. A procedure for checking **C1** that is independent of search order is also available, such that whenever the procedure returns *true* for a given set of transitions, it is guaranteed to satisfy **C1**. This means we are in a position where only **C3** needs to be checked to obtain a working, search-order independent implementation of p. o. r.

LEMMA 4.6. *Assuming **C1** holds for all ample sets along a cycle in the reduced structure, **C3** holds for this cycle whenever at least one state s on the cycle is fully expanded (meaning ample(s) = enabled(s)).*

PROOF. Again, for a complete proof, please refer to [15]. □

Traditionally, (some variation of) Lemma 4.6 is used to implement **C3** checking in practice, using a depth-first search stack. Whenever a state is encountered that would close a cycle, it is fully expanded. However, this implementation heavily relies on depth-first search. Therefore, we need to replace this condition with a different one, that would not rely on presence of a depth-first search stack.

4.2 Related Work

The Partial Order Reduction technique has been intensively studied as a leading technique to fight the state explosion problem in explicit model checking. As a result, a number of improvements and variants of the technique has been developed and successfully integrated in verification tools. These results are mutually exclusive in many cases and their usability depends on the target domain of application. In particular, there are subclasses of properties to be verified for the system under consideration, for which the formal requirements on *ample* sets may be safely weakened, hence different reduction algorithms applied. For example, to prove a deadlock freedom, the reduced structure does not have to fulfill the **C3** property at all. Similarly, if we check the system for a safety property, such as assertion violation, it is satisfactory for the states on a cycle in the reduced structure to be able to reach at least one fully expanded (not necessarily immediate) successor state. In the following we will focus on various strategies to deal with **C3** proviso that have been introduced in the literature so far. We will particularly discuss their applicability to the distributed-memory computing.

4.2.1 Static Partial Order Reduction

Static Partial Order Reduction [31] builds upon the fact that the system under consideration is an asynchronous product of individual system components. Since every cycle in the system graph projects to cycles of the components, it is possible to a priori construct a set of states that cover every possible cycle in the system graph. Whenever a state of the reduced structure is a member of such a covering set, it is fully expanded. Static partial order reduction technique is compatible with distributed memory computing, however, it is generally considered to be less effective than dynamic approaches listed below.

4.2.2 Dynamic Partial Order Reduction

In the dynamic Partial Order Reduction approach, the decision about the full expansion of a state is done for the state when it is processed by the exploration algorithm. There are several nuances of the cycle proviso (condition **C3**) that depend on whether the reduced structure is used for verification of safety or liveness properties, or whether the exploration algorithm follows a particular search order (depth-first, breadth-first, etc.).

The classical, stack-based cycle detection proviso is connected with depth-first traversal algorithm. The depth-first search algorithm maintains a stack of states on the path from the initial state of the graph to the state currently being processed. If the currently processed state has a direct successor that is on the stack, there is a cycle in the reduced structure. In case of verification of liveness properties, such a situation requires that the currently processed state must be fully expanded. However, this is not the case when verifying safety properties, where the full expansion of the currently processed state may be safely avoided if there is at least one direct successor of the state that is outside the stack [24]. In other words, a state is fully expanded if all its successors are in the stack.

If for whatever reason the algorithm for exploration of the reduced structure does not follow a depth-first visiting strategy, it cannot maintain the search stack, hence, cannot apply the stack proviso. In general, whenever a graph traversal algorithm discovers a transition leading to an already visited state, there is a potential risk that this transition closes a cycle in the reduced structure. A conservative approach therefore is to fully expand all states that have successors lying in the visited portion of the graph.

Unfortunately, even this conservative check is not free of issues in distributed memory setting – checking whether a state is visited will cost two messages and, what is worse, the successor generation will need to wait for the answer, introducing extra synchronisation (and therefore delays) into the system. In shared-memory setting, the problem is easier to resolve (the visited check can be implemented more efficiently).

4.2.3 Parallel reductions

A comprehensive survey of existing techniques for parallel p. o. r. and an exhaustive experimental evaluation is available in [35]. The work also introduces a number of novel techniques for distributed-memory reductions, although they are often rather difficult to implement and often rely on successor locality to a workstation. The latter requirement, that is, a special treatment of so-called "cross" transitions, proves to be increasingly problematic with higher numbers of parallel workers – quickly, all transitions become cross and the often more complicated and suboptimal treatment is required for all transitions of the system. In contrast, our proposed **C3** check is independent of state space distribution and therefore a highly scattered state distribution (which is common in hash-based state distribution) does not pose a problem for the check. In a similar fashion, the check proposed in [11] relies on the ability to do a local depth-first search, which in turn relies on availability of local (i.e. non-cross) transitions. Additional heuristic is proposed, that improves handling of cross transitions at the cost of visiting any given state multiple times (at most once for each worker involved in the computation). This unfortunately still translates to a high penalty for cross transitions.

A different approach to p. o. r. has been proposed in [36] – an algorithm that does not rely on **C3** at all, and instead relies on following singleton ample sets while this is possible and fully expanding otherwise. The algorithm shows promise, although it introduces complications into generation of successors and for distributed computation. It is also less general than the usual p. o. r. approach which is not restricted to singleton ample sets, even though authors claim it often outperforms the traditional approach in practice. The ramifications of a parallel, distributed implementation of this algorithm are currently not known and are a subject of future work.

For the case of reachability (i.e. restricted to safety properties) on shared-memory systems, a heuristic has been proposed [27] that is usable with a multi-core extension of the SPIN model checker. The check assumes usage of a so-called stack-slicing algorithm (this is the reason the heuristic requires a shared memory environment), proposed in [26]. The heuristic itself, similar to the above-mentioned distributed algorithms, treats cross transitions (as represented by boundary states, in SPIN terminology) specially – in this case, however, this is less problematic, since the partitioning of the state space using the stack-slicing algorithm is not static and therefore the proportion of border states in the state space is easier to control.

4.3 Cycle Detection

In this section, we will present an algorithm[2] that guarantees that along every cycle in the reduced structure, there is at least one fully expanded state. The algorithm is based on a variation of *topological sort* that can be efficiently implemented in parallel – unlike the traditional check based on DFS, the so-called "in-stack" check.

[2]The `C++` implementation of the cycle proviso algorithm can be found in the file `divine/porcp.h` in DiVinE 2 source distribution.

In addition to checking **C3**, we require the **C3** check to avoid interfering with a desirable algorithm property called on-the-fly execution (cf. Chapter 2). This means that if the algorithm is able to produce a counterexample without exploring the full state space, checking **C3** should not prevent such an algorithm from doing so. We will discuss this property later on.

DEFINITION 4.7. *A transition graph $G = (V, E)$ induced by a Kripke structure $K = (S, T, S_0, L)$ is a graph (V, E) such that $V = S$ and $(s, t) \in E \iff \exists \alpha \in T : \alpha(s) = t$.*

We assume that the model checking algorithm is not concerned with the transitions of the Kripke structure and instead explores its induced transition graph.

We also assume that the model checking algorithm is based on accepting cycle detection and is invariant under exploration order. This means that the algorithm is correct independently of the order in which it explores new transitions, as long as it eventually explores each transition reachable in the reduced state space. Moreover, if the algorithm requires revisiting states, we assume that it is possible to defer these revisiting operations arbitrarily long in the execution of the algorithm. This is not crucial for correctness, but it is important for the algorithm to keep its asymptotic complexity under the proposed reduction algorithm.

ALGORITHM 4.8. Check C3.

Input: D denotes the set of states and T the set of edges of a transition graph already explored by the model checking algorithm

Output: The result of the model checking algorithm.

1. Execute the model checking algorithm on a reduced system K'', where for each state s, $ample(s)$ – satisfying C0 through C2 – is used as the set of outgoing transitions for s. Defer any revisiting operations.
2. Repeat 3–4
3. $new \leftarrow$ Algorithm 4.9 (re-expansion).
4. Resume the model checking algorithm, adding new to its set of unexplored edges of the transition graph.
5. Until A fixed point is reached – that is, Algorithm 4.9 produces an empty set.
6. Continue with all deferred revisiting operations of the model checking algorithm.

ALGORITHM 4.9. Re-expand.

Input: D denotes the set of states and T the set of transitions explored by the model checking algorithm. D' is initially empty, but is retained over executions of the algorithm.

Output: The set of edges that still needs to be explored.

1. $R \leftarrow$ Algorithm 4.10 with $S = (D - D')$, $E = (D - D')^2 \cap T$
2. $D' \leftarrow D$
3. Return $enabled(R) - T$

ALGORITHM 4.10. Cover cycles.
Input: S, a set of states and $E \subseteq S \times S$, a set of transitions
Output: R a set of states covering all cycles induced by S

1. $X \leftarrow S$
2. Repeat 3–9
3. While there is a state s in X such that $\forall (t, s) \in E : t \notin X$ do 4–5
4. $X \leftarrow X - \{s\}$
5. $Y \leftarrow Y \cup \{t \mid (s, t) \in E\} \cap S$
6. If $Y = \emptyset$ then
7. add an arbitrary state from X to Y
8. $R \leftarrow R \cup (Y \cap X)$
9. $X \leftarrow X - Y$
10. Until $X = \emptyset$
11. Return R

LEMMA 4.11. *Algorithm 4.10, given a set S of states and a set E of edges, returns a set R of states such that for every cycle $c \subseteq S$, it holds that $c \cap R \neq \emptyset$.*

PROOF. The main loop invariant is that a cycle $c \subseteq S$ is either fully embedded in X, or there is a state $s \in c \cap R$. This is clearly true before entering the loop for the first time, as $X = S$. When the algorithm terminates, $X = \emptyset$, therefore, for each cycle $c \subseteq S$, it must hold that $c \cap R \neq \emptyset$.

We now only need to show that this is indeed the loop's invariant. First, a state s is never removed from X in the inner loop if it is a part of a cycle fully embedded in X. This however means that such a state might only be removed in the $X \leftarrow X - Y$ assignment – but every such state is also added to the set R, meaning that if s has been a part of a cycle fully embedded in X, that $s \in c \cap R$.

Termination: The algorithm clearly terminates, as in each iteration, at least one state is removed from X. □

LEMMA 4.12. *The time complexity of Algorithm 4.10 is in $\mathcal{O}(|S| + |E \cap (S \times S)|)$.*

PROOF. Each state in S is examined exactly once, when it is being removed from X. When a state is being removed, each of its outgoing edges pointing back into S is examined exactly once. □

4.4 Correctness

THEOREM 4.13. *Algorithm 4.8 ensures that on every cycle in the reduced state space, there is at least one fully expanded state.*

PROOF. We show correctness of Algorithm 4.8 by induction.

As a base, let us consider that E' in Algorithm 4.9 (re-expand) is empty. It follows from Lemma 4.11, that result of the re-expansion is exactly the set of edges such that (by Lemma 4.11) at least one state on every cycle is fully expanded.

We can now assume that before every invocation of re-expansion, E' already has, on each cycle, at least one fully expanded state. We need to prove that after executing Algorithm 4.9, E' will keep this property.

Again, from Lemma 4.11, we can deduce that every cycle fully embedded in $E - E'$ will have at least one fully expanded state (that is, there will be at least one state on every cycle, such that all its enabled edges will be explored).

This covers all cycles that do not cross the E/E' boundary. However, when there is an edge (s, t) such that $s \in E'$ and $t \in E$ is present in $E \cup E'$, we know that s has been fully expanded. Clearly, any cycle crossing the E/E' boundary will contain at least one such edge.

When Algorithm 4.8 terminates, E' contains all of the reduced state space. □

LEMMA 4.14. *Algorithm 4.8 ensures that **C3** holds.*

PROOF. Follows immediately from Theorem 4.13, using Lemma 4.6. □

THEOREM 4.15. *The reduced system satisfies a given LTL$_{-\mathrm{x}}$ formula φ iff the original system satisfies φ.*

PROOF. Follows from Lemma 4.5 and Lemma 4.14. □

4.5 Time complexity

Clearly, it is important that a prospective **C3** check can be performed in *linear* serial time – an algorithm with super-linear complexity would clearly impede the performance of the process of state space exploration, and in turn of model-checking.

LEMMA 4.16. *Time complexity of Algorithm 4.8 is linear in size of the reduced state space.*

PROOF. Every invocation of Algorithm 4.10 is linear in its parameter S (Lemma 4.12). We show that any given state is in S in at most one invocation of Algorithm 4.10.

When a state occurs in S, it will be immediately added to E'. When a state is already in E', it will never again occur in S. Therefore, we conclude that every state occurs in S in at most one iteration. □

Even though the **C3** check itself is linear, many of the parallel accepting cycle detection algorithms are not linear in all cases. There are two requirements for the combined algorithm for accepting cycle detection in the reduced state space: firstly, the combined algorithm should have time complexity no worse than the

original accepting cycle detection algorithm employed and secondly, it should not be required for the **C3** check to perform a full reachability pass over the state space before starting the cycle detection itself.

THEOREM 4.17. *Using Algorithm 4.8 does not affect time complexity of the underlying model checking algorithm.*

PROOF. Any accepting cycle detection lies in $\Omega(|V| + |E|)$. Since we require the model checking algorithm to allow deferring any revisiting operations, we can assume that the algorithm can be reordered to run in two passes: first, it explores the state space in $\Theta(|V| + |E|)$ and then it possibly carries out additional computation with complexity t. Clearly, Algorithm 4.8 only interferes with first of these two passes, and since it runs in $\mathcal{O}(|V| + |E|)$, the overall complexity of the first pass is $\Theta(|V| + |E|)$, making the overall complexity $\Theta(|V| + |E|) + t$, which is the same as that of the original model checking algorithm. □

Further, we shall consider the proposed heuristic in terms of on-the-flyness of the underlying accepting cycle detection algorithm. It can be seen easily, that for a level 2 on-the-fly algorithm with the required properties, the algorithm would clearly stay a level 2 on-the-fly algorithm when combined with the heuristic. Unfortunately, no such algorithm is currently known.

As for level 1 algorithms, these are not required to terminate early for every input, even in the cases there is an counterexample to be found in the state space. Moreover, since the algorithm is invariant under exploration order, the ability to find a counterexample is largely dependent upon the order in which the state space is explored.

Since level 1 on-the-flyness is of a relatively heuristic nature, the following is not really a theorem – strictly speaking, to prove this property, it would be enough to find an input where the algorithm finds such a counterexample, and where $\mathit{ample}(s) = \mathit{enabled}(s)$ for each state s. However, this hardly tells us anything about practical behaviour of the reduction.

CONJECTURE 4.18. *A model checking algorithm that is level 1 on-the-fly will also be level 1 on-the-fly if combined with Algorithm 4.8, while maintaining this property to an useful degree.*

CLAIM 4.19. *If the original algorithm would find a counterexample in a small fraction of the state space, a relatively small change in exploration order induced by the reduction algorithm is unlikely to affect size of this fraction significantly.*

Moreover, if the probability of discovering a counterexample in a given percentage of state space is independent of the exploration order, then Algorithm 4.8 will not alter this probability at all.

4.6 Using with OWCTY

To successfully exploit partial order reduction algorithm presented, it needs to be combined with a suitable model checking algorithm. In this section, we show that OWCTY [13, 22] fulfils all the restrictions we have placed on the model checking algorithm. Moreover, it can be adapted to run on-the-fly and it is generally suitable for practical model checking [4].

The algorithm starts out with a single full reachability (and a heuristic may enable it to uncover a counterexample during this phase, making it on-the-fly). This reachability pass is exploration-order independent. We incorporate the **C3** check proposed in previous chapters into this pass. This also means, that no re-visits are done before the **C3** check is complete.

ALGORITHM 4.20. Owcty + C3.
Input: $M = (V, E, A, I)$ a problem instance from Definition 2.1.
Output: True if no accepting cycles were detected.

1. $S \leftarrow$ Algorithm 2.2 (reachability) from I, on **C0**-**C2**-reduced M.
2. Repeat 3–4
3. $new \leftarrow$ Algorithm 4.9 (re-expansion).
4. Resume reachability, exploring edges in new
   Add any newly explored vertices to S
5. Until $new = \emptyset$
6. Repeat 7–8
7. $R \leftarrow$ Algorithm 2.2 (reachability from S)
8. $S \leftarrow$ Algorithm 2.4 (elimination on R)
9. While $R \neq S \wedge S \neq \emptyset$
10. Return $S = \emptyset$

For details about the reachability and elimination procedures, please refer to Section 2.1. The property important here is that they re-explore the state space already stored in S. Moreover, it is not necessary to store edges explicitly – we only need a single bit for each state, remembering if $enabled(s)$ or $ample(s)$ has been used for this state. The successors are generated from the description of the input and description of the state being expanded.

4.7 Using with MAP

Another algorithm that is suitable for parallel model checking and provides different trade-offs than OWCTY is MAP [9, 10]. Again, it can be combined with our proposed **C3**-checking algorithm quite naturally.

The algorithm issues repeated propagation of maximum accepting predecessors of all accepting vertices. When it discovers a vertex that is its own accepting predecessor, it concludes it has found an accepting cycle. In MAP, there are two kinds of revisits: as part of the propagation, and due to additional passes of the

algorithm. It is easily seen that the latter kind can be easily deferred, like it was the case with OWCTY. The first kind is however somewhat more tricky.

There are two options, either ignore this problem, or try reordering the algorithm. The first will, in the worst case, add a factor of n to complexity of the first pass, where n is the number of re-expansions issued by the **C3** check. Needless to say, the actual increase on realistic inputs is likely to be much lower, on the order of a small constant increase.

However, it is still possible to defer the re-expansions, at the cost of possibly increasing memory use, for storing the queue until all the **C3**-induced re-expansions are done. The practical consequences would have to be measured in actual implementation, as it is hard to assess theoretically.

4.8 Experiments

To evaluate the usefulness of the proposed p. o. r. technique, we have experimentally evaluated the new reduction against the traditional approach based on DFS. In each case, we have performed state-space exploration of the model (with and without properties), either in full, or using one or the other partial order reduction. The results can be seen in a number of tables. The Tables 4.1, 4.2 and 4.3 present the results on the parametrised models available in `mdve` format in the `DiVinE Cluster` distribution. Table 4.4 summarises results from a number of models from the `BEEM` database. The first column always has the size of the full state space, whereas the other two show the number of states under each of the evaluated reductions, and the percentage of the full state space that has been explored.

Due to the limitations of the **C0** – **C2** implementation available for `DVE` models, and the unoptimised nature of our **C3** implementation, we omit timings of these experiments, as they cannot be meaningfully compared. It should be also noted that our implementation of **C3** is not deterministic in parallel setting – we have executed the experiments with a single thread to retain determinism: in parallel executions, the variations in exploration order can influence the choice of vertices for expansion and therefore the overall result.

N	**Full**	DFS-POR		**Par-POR**	
No property.					
2	63	55	87.3 %	52	82.5 %
3	2376	1823	76.7 %	1775	74.7 %
4	131301	104756	79.7 %	102284	77.9 %
$G((\neg p_0 cs) \rightarrow F(p_0 cs))$					
2	125	107	85.6 %	100	80.0 %
3	4751	3624	76.2 %	3527	74.2 %
4	262601	209343	79.7 %	204225	77.7 %
$GF(someoneincs)$					
2	124	104	83.8 %	96	77.4 %
3	4749	3551	74.7 %	3448	72.6 %
4	262598	207080	78.8 %	201904	76.8 %

Table 4.1: Reduction in peterson, N is the number of processes.

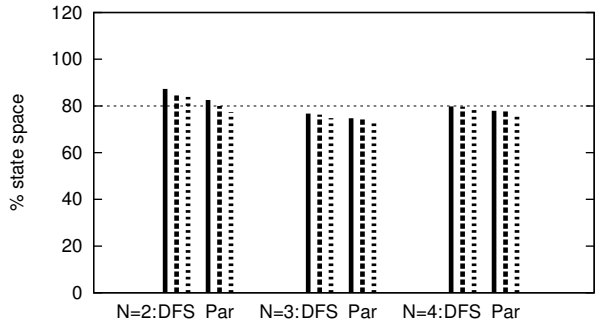

Figure 4.2: Reduction in peterson, N is the number of processes.

N	**Full**	DFS-POR		**Par-POR**	
No property.					
2	12	7	58.3 %	10	83.3 %
3	36	11	30.5 %	28	77.7 %
4	96	15	15.6 %	64	66.6 %
5	240	19	7.9 %	157	65.4 %
6	576	23	3.9 %	367	63.7 %
7	1344	27	2.0 %	835	62.1 %

Table 4.2: Reduction in token ring mutual exclusion, N is the number of processes.

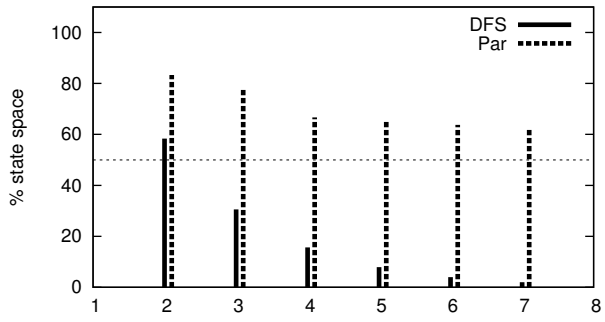

Figure 4.3: Reduction in token ring mutual exclusion, N is the number of processes.

N	**Full**	DFS-POR		**Par-POR**	
No property.					
2	141	125	88.6 %	125	88.6 %
3	2152	1528	71.0 %	1528	71.0 %
4	55361	30433	54.9 %	30433	54.9 %
$F(elected)$					
2	131	115	87.7 %	115	87.7 %
3	2122	1498	70.5 %	1498	70.5 %
4	55177	30249	54.8 %	30249	54.8 %
$FG(oneleader)$					
2	271	239	88.1 %	239	88.1 %
3	4273	3025	70.7 %	3025	70.7 %
4	110537	60681	54.8 %	60681	54.8 %
$noleader\ U\ oneleader$					
2	131	115	87.7 %	115	87.7 %
3	2122	1498	70.5 %	1498	70.5 %
4	55177	30249	54.8 %	30249	54.8 %

Table 4.3: Leader election, N is the number of processes.

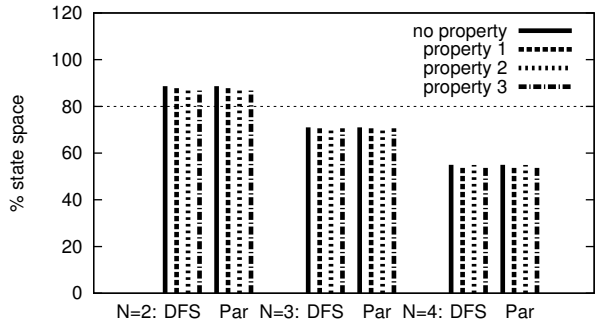

Figure 4.4: Reduction in leader election, N is the number of processes.

Model	**Full**	DFS-POR		**Par-POR**	
`peterson.1.dve`	12498	7999	64.0 %	7780	**62.2 %**
`peterson.2.dve`	124704	106949	85.7 %	102779	**82.4 %**
`peterson.3.dve`	170156	129147	75.8 %	122704	**72.1 %**
`peterson.1.prop2.dve`	22816	17481	76.6 %	17098	**74.9 %**
`peterson.2.prop2.dve`	234376	214441	91.4 %	210287	**89.7 %**
`peterson.1.prop3.dve`	24985	15907	63.6 %	15479	**61.9 %**
`peterson.2.prop3.dve`	249368	212181	85.0 %	202829	**81.3 %**
`mcs.1.dve`	7963	7312	**91.8 %**	7778	97.6 %
`mcs.2.dve`	1408	937	**66.5 %**	1332	94.6 %
`mcs.1.prop2.dve`	12206	11545	**94.5 %**	12132	99.3 %
`mcs.2.prop2.dve`	2462	1849	**75.1 %**	2370	96.2 %
`mcs.1.prop3.dve`	15815	14687	**92.8 %**	15610	98.7 %
`mcs.2.prop3.dve`	2811	1941	**69.0 %**	2672	95.0 %
`synapse.1.dve`	46756	43290	92.5 %	43108	**92.1 %**
`synapse.2.dve`	61048	61048	100.0 %	61048	100.0 %
`synapse.1.prop2.dve`	7226	6758	**93.5 %**	6780	93.8 %
`synapse.2.prop2.dve`	15713	15713	100.0 %	15713	100.0 %
`leader_filters.1.dve`	4966	4810	96.8 %	4810	96.8 %
`leader_filters.2.dve`	29284	22423	76.5 %	22423	76.5 %
`leader_filters.3.dve`	91093	87809	96.3 %	87809	96.3 %
`leader_filters.1.prop2.dve`	4966	4966	100.0 %	4966	100.0 %
`leader_filters.2.prop2.dve`	28804	23239	80.6 %	23239	80.6 %
`leader_filters.3.prop2.dve`	91093	91093	100.0 %	91093	100.0 %

Table 4.4: A selection of `BEEM` models.

Chapter 5

Conclusion

We have presented a partial order reduction technique for parallel LTL model checking. It is based on a novel C3 proviso that uses topological sort instead of the more traditional DFS-based method. Moreover, the proposed proviso preserves time complexity of the algorithm used for accepting cycle detection.

Moreover, we have given a broad overview of the current state of the art in parallel LTL model checking. The partial order reduction technique has been shown to fit nicely in the existing framework of parallel model checking technology. An algorithm has been presented that fits well with the reduction heuristic and behaves quite well in practice: it works on-the-fly and achieves significant speedup on parallel hardware.

Additionally, we have evaluated the reduction heuristic experimentally. The results are quite promising, with the heuristic matching or exceeding Nested DFS in some cases. Unfortunately, there is also an example where Nested DFS provides exponential reduction while the proposed heuristic fails to do so (even though it is super-linear, it is not exponential... see Table 4.2 for details). Nevertheless, the proposed heuristic appears to be quite usable in practice and is scheduled to be included in a future release of DiVinE 2.

5.1 Future Work

It is currently not known, whether the missed exponential reduction opportunity in the above example (Table 4.2) is inherent in the heuristic, or is only a coincidence. The technique is quite new and there is definitely space for optimisation and tuning, which could improve results in many cases.

As mentioned, the heuristic is scheduled for inclusion in DiVinE 2, which means that a practical, robust and fast implementation of the proposed algorithms is required: only a proof-of-concept has been implemented for the purpose of experimental evaluation, which is unfortunately not ready for general use. Moreover, a new modelling language is being worked on, and a production-ready implementation of **C0**-**C2** is wanted: the implementation available for the current language (DVE) is

again more-or-less a proof of concept and has not been extensively optimised.

Apart from traditional **C3** checking, an alternative algorithm (twophase, see Section 4.2.3) exists that leverages singleton ample sets. An implementation of this algorithm, possibly in a combination with the proposed **C3** check may lead to an improved reduction in some cases.

Bibliography

[1] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *The 18th International Conference on Computer Aided Verification*, volume 4144 of *LNCS*, pages 278–281. Springer-Verlag, 2006.

[2] J. Barnat, L. Brim, and P. Rockai. Scalable Shared Memory LTL Model Checking. *International Journal on Software Tools for Technology Transfer*. To appear.

[3] J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *Model Checking Software, the 14th international SPIN Workshop*, volume 4595 of *LNCS*, pages 187–203. Springer-Verlag, 2007.

[4] J. Barnat, L. Brim, and P. Ročkai. DiVinE Multi-Core – A Parallel LTL Model-Checker. In *Automated Technology for Verification and Analysis*, volume 5311 of *LNCS*, pages 234–239. Springer-Verlag, 2008.

[5] J. Barnat, L. Brim, and P. Ročkai. An Optimal On-the-fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *International Conference on Formal Engineering Methods*, volume 5885 of *LNCS*, pages 407–425. Springer-Verlag, 2009.

[6] J. Barnat, L. Brim, and P. Ročkai. DiVinE 2.0: High-Performance Model Checking. In *High Performance Computational Systems Biology*. IEEE Conference Publishing Services, 2009. To appear.

[7] J. Barnat and P. Ročkai. Shared Hash Tables in Parallel Model Checking. In *Participant proceedings of the Sixth International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007)*, pages 81–95. CTIT, University of Twente, 2007.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, November 2000.

[9] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer-Verlag, 2004.

[10] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *The 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2005)*, pages 1–12, 2005.

[11] Lubos Brim, Ivana Cerna, Pavel Moravec, and Jiri Simsa. Distributed partial order reduction of state spaces. *Electr. Notes Theor. Comput. Sci.*, 128(3):63–74, 2005.

[12] Stefano Caselli, Gianni Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In *Application and Theory of Petri Nets*, pages 181–200, 1995.

[13] I. Černá and R. Pelánek. Distributed Explicit Fair Cycle Detection (Set Based Approach). In T. Ball and S.K. Rajamani, editors, *Model Checking Software, the 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 49 – 73. Springer-Verlag, 2003.

[14] Gianfranco Ciardo, Joshua Gluckman, and David Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS J. Comp*, 1995.

[15] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[16] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1:275–288, 1992.

[17] Edsger W. Dijkstra. Shmuel Safra's version of termination detection. EWD Manuscript, January 1987.

[18] S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *SPIN'06*, pages 1–18. Springer, 2006.

[19] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2-3):247–267, 2004.

[20] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed Explicit Model Checking with HSF-SPIN. In *SPIN'01*, pages 57–79. Springer, 2001.

[21] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier, 1990.

[22] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 420–434. Springer-Verlag, 2001.

[23] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[24] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *12th Int. Conf on Protocol Specification Testing nad Verification (IFIP 1992)*, pages 349–363, 1992.

[25] G. J. Holzmann, D. Peled., and M. Yannakakis. On Nested Depth First Search. In *The SPIN Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the 2nd SPIN Workshop.

[26] Gerard J. Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. In *Participant proceedings of the Sixth International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007)*, pages 1–15. CTIT, University of Twente, 2007.

[27] Gerard J. Holzmann and Dragan Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *Software Engineering, IEEE Transactions on*, 33:659–674, 2007.

[28] IEEE. *IEEE 1003.1c-1995 Standard for Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1995.

[29] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, New York, NY, USA, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[30] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, Ch. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In *CAV'09*, volume 5643 of *LNCS*, pages 414–429. Springer-Verlag, 2009.

[31] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static Partial Order Reduction. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*, pages 345–357. Springer, 1998.

[32] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 2.1, 2008.

[33] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Notices*, 39(6):35–46, 2004.

[34] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996.

[35] P. Moravec. *Distributed State Space Reductions*. PhD thesis, Faculty of Informatics, Masaryk University Brno, January 2008.

[36] R. Palmer and G. Gopalakrishnan. Partial order reduction assisted parallel model checking. In *Proc. Parallel and Distributed Model Checking (PDMC) Workshop*, 2002.

[37] R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software, the 14th International SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer-Verlag, 2007.

[38] D. Peled. All from One, One for All: on Model Checking Using Representatives. In *The 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag.

[39] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of CAV'94*, pages 377–390. Springer Verlag, LNCS 818, 1994.

[40] Doron Peled. Ten years of partial order reduction. In *The 10th International Conference on Computer Aided Verification*, pages 17–28. Springer-Verlag, 1998.

[41] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, June 1985.

[42] P. Ročkai. Multi-Threaded Nested DFS. Bachelor's thesis, Faculty of Informatics, Masaryk University Brno, 2007.

[43] P. Šimeček. DiVinE – Distributed Verification Environment. Master's thesis, Faculty of Informatics, Masaryk University Brno, 2006.

[44] S. A. M. Talbot. Performance tuning of programs for shared-memory multiprocessors. Master's thesis, Imperial College, London, U.K., 1995.

[45] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, pages 146–160, January 1972.

[46] Antti Valmari. Stubborn set methods for process algebras. In *Proceedings of the DIMACS workshop on Partial order methods in verification*, pages 213–231. AMS Press, Inc., 1997.

[47] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symposium on Logic in Computer Science*, pages 322–331. Computer Society Press, 1986.