Parallelising Symbolic State-Space Generation Algorithms on Shared-Memory Architectures

Jonathan Arie Ezekiel

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF YORK
DEPARTMENT OF COMPUTER SCIENCE

October 2007
Abstract

Automated verification of discrete-state systems, such as temporal-logic model checking, relies on efficient algorithms for computing state-spaces of complex system models. To avoid the well known state-explosion problem, symbolic algorithms working on decision diagrams, have proved successful in practice. Even so, such algorithms are often too time-consuming, and improvements in time-efficiency are highly desirable. Several efforts have been made to implement these algorithms on parallel computer platforms, but the resultant algorithms show limited speedups.

This thesis explores the parallelisation of Saturation, a symbolic state-space generation algorithm with unique features. For asynchronous system models with interleaving semantics, it exploits the local effect of firing events on state vectors. We use this feature to create parallel tasks in Saturation that are scheduled and load balanced using a thread pool. To further examine the algorithms scheduling and load balancing efficiency, Cilk, a parallel irregular language is used to parallelise Saturation. We use a rigorous approach to measuring and evaluating state-space exploration algorithms for assessing the quality of our parallelisations, which are limited to shared-memory architectures in order to avoid communication overhead.

The research shows that a heavily optimised symbolic state-space generation algorithm such as Saturation is difficult to parallelise. Even though super-linear speedups were obtained for the parallel Cilk algorithm, in general speedups are highly model-dependent, and impossible to predict. The research testifies that time-efficiency improvements of sequential symbolic state-space generators are more productive than parallelisation.
## Contents

1 Introduction  
1.1 Contribution of this thesis  
  1.1.1 Contribution 1: an approach to measuring and evaluating parallel state-space exploration algorithms  
  1.1.2 Contribution 2: a hand crafted parallel variant of Saturation  
  1.1.3 Contribution 3: a parallel variant of Saturation using Cilk  
1.2 Structure of this thesis  

2 Reachability analysis  
2.1 Temporal logic model checking  
  2.1.1 The state-explosion problem  
  2.1.2 Reachability analysis  
2.2 Explicit reachability analysis  
2.3 Symbolic reachability analysis  
  2.3.1 Binary decision diagrams  
  2.3.2 Constructing BDDs  
  2.3.3 Symbolic representation of $S$ and $\mathcal{N}$  
  2.3.4 Symbolic state-space generation  
  2.3.5 Iteration strategy
2.4 Parallel state-space construction .................................. 35
  2.4.1 Parallel explicit approaches .................................... 36
  2.4.2 Parallel symbolic approaches .................................... 38
2.5 Evaluations of parallel state-space exploration algorithms .... 39
2.6 Chapter summary ..................................................... 41

3 Saturation ............................................................... 42
  3.1 Introduction ......................................................... 42
  3.2 Structured systems ................................................ 43
    3.2.1 Symbolic encoding of $S$ ....................................... 44
    3.2.2 Symbolic encoding of $N$ ....................................... 45
  3.3 Saturation-based iteration strategy ............................... 49
    3.3.1 Saturation example ............................................. 53
  3.4 SMART ............................................................... 57
  3.5 Issues with parallel C++ ........................................... 59
  3.6 Chapter summary ................................................... 61

4 Measurement and evaluation ......................................... 62
  4.1 Introduction ......................................................... 63
  4.2 Background on parallelising state-space exploration algorithms 65
    4.2.1 From sequential to parallel .................................... 66
    4.2.2 What are the overheads? ....................................... 67
    4.2.3 Implementing a parallel state-space exploration algo-
        rithm ............................................................... 69
    4.2.4 Pthreads ......................................................... 71
    4.2.5 Addressing parallel overheads .................................. 79
  4.3 Evaluating parallel state-space exploration algorithms ....... 80
    4.3.1 Measuring parallel overheads .................................. 81
    4.3.2 The influences of a model on parallel overheads .......... 83
4.3.3 Evaluating a parallel algorithm .............................. 85
4.4 Our benchmark ..................................................... 86
  4.4.1 Benchmark justification ...................................... 92
  4.4.2 Experimental setup .......................................... 93
4.5 Summary ............................................................ 94

5 Parallelising Saturation ........................................... 95
  5.1 Parallel BFS .................................................. 95
    5.1.1 Chained BFS ............................................ 97
    5.1.2 Run-time and memory results ......................... 98
  5.2 Parallel differences in the order of event firings .......... 101
    5.2.1 Approaches to optimising the event order .......... 102
    5.2.2 Run-time and memory results ....................... 104
  5.3 Parallel Saturation .......................................... 107
    5.3.1 Thread pool justification ............................... 110
    5.3.2 The thread pool algorithm ......................... 113
    5.3.3 Example .................................................. 117
    5.3.4 Correctness of the algorithm ....................... 121
    5.3.5 Implementation concerns ............................... 121
    5.3.6 Run-time and memory results ....................... 123
  5.4 Four core vs four processor ................................. 125
    5.4.1 Optimising the queue ordering ................. 126
  5.5 Optimising the ordering of events .......................... 128
    5.5.1 Parallel SCCs ........................................... 129
    5.5.2 Parallelism within an SCC ......................... 130
    5.5.3 Run-time and memory results ....................... 131
  5.6 Optimising the locking strategy ............................ 133
  5.7 Chapter summary ............................................. 134
## List of Figures

2.1 An example of a model with a safe state. 26
2.2 Explicit algorithm for state-space construction. 27
2.3 A binary decision diagram from a binary decision tree. 29
2.4 Pseudo-code for the apply algorithm. 31
2.5 An illustration of the apply algorithm. 32
2.6 A two bit binary counter. 33
2.7 BFS algorithm for symbolic state-space construction. 34
2.8 Chained BFS algorithm for symbolic state-space construction. 35

3.1 An example MDD [29]. 45
3.2 A partitioned Petri net example. 47
3.3 Pseudo-code for the Saturation and RecFire functions. 50
3.4 Pseudo-code for the Saturation supporting functions. 52
3.5 Saturation example: MDD construction. 54
3.6 Code for the C++ object parallelisation problem. 59

4.1 Sequential and parallel reachability algorithm. 65
4.2 Parallel state-space enumeration and processor allocation of functions. 67
4.3 Illustration of the thread pool processor mapping and synchronisation on the shared queue. 73
4.4 POSIX thread pool example (1). 73
5.19 Pthreads regional locking example. ........................... 133
5.20 Regional locking, leader election run-time results. ....... 133

6.1 Illustration of the Cilk processor mapping and workstealing
  on the deque. .................................................. 140
6.2 Cilk based Saturation using inlets. .......................... 141
6.3 Calling order for spawns. ................................. 142
6.4 Cilk based Saturation without exploiting call/return semantics. 143
6.5 Cilk inlet variant, selected run-time and memory results. . 145
6.6 Cilk spawn variant, selected run-time results. ............... 146
6.7 Cilk spawn variant, selected memory results. .............. 147
6.8 Cilk spawn variant with chaining heuristic, selected run-time
  results. ......................................................... 149
6.9 Cilk spawn variant with chaining heuristic, selected memory
  results. ............................................... 150

7.1 Pseudo-code for the anticipated firing algorithm. ......... 154
7.2 Run-time results for the anticipated firing algorithm on large
  models (1). ............................................... 157
7.3 Run-time results for the anticipated firing algorithm on large
  models (2). ............................................... 158
# List of Tables

3.1 Kronecker representation for the Petri net example. . . . . . . 47
3.2 Saturation example: next-state function. . . . . . . . . . . . . 53
3.3 C Saturation run-time improvement against C++ Saturation. 60

4.1 State-space sizes of small, medium and large models . . . . . 89

5.1 Run-time and memory results for the BFS algorithms. . . . . 99
5.2 Next-state function for the Parallel Saturation example. . . . 117

A.1 Run-time and memory results for the slotted ring model. . . 180
A.2 Run-time and memory results for the round robin model. . . 181
A.3 Run-time and memory results for the Kanban model. . . . . . 182
A.4 Run-time and memory results for the FMS model. . . . . . . 183
A.5 Run-time and memory results for the queens model. . . . . . 184
A.6 Run-time and memory results for the leader election model. . 185
A.7 Run-time and memory results for the bounded queue model. 186
A.8 Run-time and memory results for the dining philosophers model. 187
A.9 Run-time and memory results for the RSM model. . . . . . . 188
A.10 Run-time and memory results for the aloha model. . . . . . 189
Acknowledgements

Firstly I would like to thank my supervisor, Gerald Lütten. I am extremely grateful for the vast amount of high quality advice and help with my research over the last number of years, and particularly for his patience and enthusiasm. On top of this I am also grateful for the many collaboration and research opportunities he arranged for me, including arranging funding for visits to NIA and UCR.

I would like to thank my internal supervisor Alan Burns for providing feedback on the proposed thesis, encouraging my work, and for conducting the examination of the final thesis. Similarly to Howard Barringer from the University of Manchester for acting as external examiner, and asking insightful questions of my work that only someone with significant experience in this area or research could have thought of.

I would like to thank Radu Siminiceanu for inviting me to NIA and arranging funding for the visit, as well as his advice on my research. I would also like to thank Gianfranco Ciardo for inviting me to UCR and for his advice on my research. I learned a great deal from both collaborations and am highly appreciative of the valuable advice and insights from Radu and Gianfranco. I would also like to thank Ming-Ying Chung for graciously allowing me to stay with him and his family while I was at UCR.

I would like to thank my MSc supervisor, Richard Paige, who first brought my attention to the PhD studentship, and has continued to encour-
age my progression in this field. I would like to thank Jan Tobias Muehlberg from my research group, who has helped me with numerous technical matters relating to linux and provided feedback on my publications.

I would like to thank the members of the PLASMA group at the University of York who have provided a valuable platform for practising talks and presenting ideas. I would also like to say a general thankyou to the numerous staff in the computer science department, including the support staff who have always provided excellent support, and to the white rose grid staff, that helped me with matters relating to parallel architectures.

Last but not least I would like to express my deepest gratitude to my family, particularly my parents, Anne and Daniel for all their support and encouragement over the last few years.
Author’s declaration

Chapter 4 is based on the paper “Measuring and evaluating parallel state-space exploration algorithms”, published at PDMC 2007 [44]. The publication is co-authored with Dr. Gerald Lüttingen at the University of York. The key technical ideas in the paper are mine, as is the main body of the writeup. I received advice on research methods, how to proceed with the research, and revisions to the paper were made by Dr. Lüttingen.

Chapter 5 is based on the paper “Can Saturation be Parallelised? On the Parallelisation of a Symbolic State-Space Generator”, published at PDMC 2006 [46], and the corresponding National Institute of Aerospace (NIA) technical report “A Parallel Saturation Algorithm on Shared Memory Architectures” [47]. The publications are co-authored with Dr. Gerald Lüttingen at the University of York, and Dr. Radu Siminiceanu at the National Institute of Aerospace. The key technical ideas in the papers are mine, as are the main body of the writeups. I received advice on research methods, how to proceed with the research, and revisions to the papers were made by Dr. Lüttingen and Dr. Siminiceanu.

Chapter 6 is based on the paper “Parallelising symbolic state-space generators”, published at CAV 2007 [45]. The publication is co-authored with Dr. Gerald Lüttingen at the University of York, and Prof. Gianfranco Ciardo at the University of California, Riverside. The key technical ideas in the paper are mine, as is the main body of the writeup. I received advice on
research methods, how to proceed with the research, and revisions to the paper were made by Dr. Lüttgen and Prof. Ciardo.

Except for the above cases and where stated, all of the work in this thesis represents the original work of the author.
Chapter 1

Introduction

Over the past few decades, computer systems have developed rapidly to become intertwined into our everyday lives. They are heavily relied upon in the modern world, simplifying many aspects of our lives, from communication using mobile telephone networks, to embedded devices in consumer goods. Unfortunately, the complexity of today’s systems also introduces a greater potential for system failure, due to the increased difficulty in uncovering errors. The number of computational paths within these systems make them impractical to exhaustively test. This is problematic, since undetected errors at design time can be extremely costly, given the large scale on which many systems are developed.

There are numerous examples in modern history where system errors have had a huge impact on financial and time resources. The Pentium I processor division bug [41] that occurred in 1994, was estimated to have cost half a billion dollars, while the explosion of the Ariane 5 rocket in 1996 due to a software error [40], cost an estimated seven billion dollars, and ten years of development time. These incidents are regrettable, however, errors in systems that are critical to our safety are unacceptable.

Testing and simulation are the earliest approaches to tackling errors. The
former operates on the physical system, while the latter runs on a theoretical model of the system. Both methods involve examining as many execution paths in the system as possible in an attempt to uncover erroneous instances. Typically, some input is given to the system and the output is validated for correctness. These approaches have become increasingly strained by the huge number of paths in concurrent systems, where only a subset of the execution paths can be practically examined. This lack of coverage can leave subtle errors in the system undetected, that need to be corrected at a later date. The other drawback to this technique is the late stage of applying it, often after errors have already been introduced into component parts of the system.

In contrast to testing and simulation approaches, automated verification techniques such as model checking [32, 88], offer exhaustive coverage of the system under consideration. Model checking is particularly useful for control-intensive systems. The automatic nature of the process is a clear advantage, and, since it is exhaustive, is effective in uncovering subtle errors in the design. Applying the process at an early stage reduces costs, since errors do not need to be corrected at a later date. The positive aspects of this type of technology have led to a recent increase in the popularity of model checking techniques in industry [42].

Given a system model, and a property of that system, the model checking technique decides whether the property is satisfied or violated. Underlying the decision process is the construction of the state-space of the model, which is exhaustively searched. Due to the interleaving semantics of asynchronous systems, the number of global states in the system can increase exponentially through concurrency. This problem is known as the state-explosion problem, which can cause the state-space to become large enough to exhaust memory, or where enough memory is available, can render a model checker practically
unable, due to the length of time required to explore the state-space. In practice, the challenge of model checking is to develop algorithms that are time- and memory-efficient by addressing the state-explosion problem.

Several significant advances have been made in addressing the state-explosion problem, that have made model checkers practically usable. In particular, symbolic model checking [20, 79] based on decision diagrams [3, 18, 52, 93], has facilitated compact representations of complex system models that were previously unable to fit into memory. Symbolic algorithms for generating the state-space, usually operating on binary decision diagrams (BDDs)[16], have proved successful in practice [31, 78]. Several efforts have been made to implement these algorithms on parallel computer platforms, most notably on networks of workstations and on PC clusters [58, 59, 62, 81, 95], where the primary goal is to utilise extra memory resources to represent the state-space. However, exploiting extra processors for the purpose of improving time-efficiency is less well researched for symbolic algorithms.

Saturation [29], as implemented in the verification tool SMART [26], is a symbolic state-space generation algorithm with unique features. It is intended for asynchronous system models that are based on an interleaving semantics, and exploits the local effect of firing events on state vectors by locally manipulating multi-valued decision diagrams (MDDs) [70], which are a generalised version of BDDs. Saturation has proved orders of magnitude more time- and memory-efficient than other symbolic algorithms [29], including the one implemented in the popular NuSMV model checker [31]. Hence, the question arises as to whether the locality of events can also be utilised for parallelising Saturation in order to achieve further speed-ups. Previous approaches to parallelising Saturation have focused on data parallelism [22, 23, 24], but not on parallelising the algorithm itself.

This thesis is designed to test the hypothesis that symbolic model check-
ers can be parallelised to improve the time-efficiency of the underlying state-space generation algorithm. To test the hypothesis, the Saturation algorithm is parallelised, which provides both a challenge and opportunity to determine the potential of parallelising a heavily optimised symbolic algorithm. To avoid the communication overhead inherent in PC clusters, the parallelisation is explored for shared-memory architectures, in particular multi-core machines [76]. This research is especially relevant due to the recent advent of affordable multi-core machines, and the potential increase of cores in the near future. Indeed, Intel have already built an 80 core processor, and have predicted 100 core processors in the future [http://www.intel.com/pressroom/].

1.1 Contribution of this thesis

Developing time-efficient model checking algorithms for automated verification is a difficult challenge, but also a crucial one if financial, time and perhaps even lives are to be saved. Exploiting the growing number of cores on multi-core machines could offer a way to address the lack of time-efficiency that often renders model checking impractical for asynchronous systems with large numbers of processes and transitions [85].

The contribution of this thesis is important, since it investigates the time-efficiency aspects of parallel symbolic model checking, which up until now has been less well researched than increasing available memory. The thorough investigation of the causes of slowdowns, and how to address them provides a basis for which others considering time-efficiency improvements can save valuable time by using the approach and ideas in this thesis. Further to this, this thesis provide a number of answers to important questions regarding comparisons between parallelised and highly optimised symbolic model checking algorithms, which can be used to decide whether the paral-
parallelisation process for time-efficiency gains is worth considering at all.

The results of the research conducted during the course of this thesis are three particular contributions which are valuable in answering many of the questions that arise when considering parallelisations of symbolic model checkers.

1.1.1 Contribution 1: an approach to measuring and evaluating parallel state-space exploration algorithms

The first contribution of the thesis is an approach to measuring and evaluating parallel state-space exploration algorithms that allows us to select a benchmark for our experiments. This poses several challenges that so far have been rarely addressed in the literature: what exactly are the parallelisation overheads, how can and cannot they be measured, and how should system models be selected in order to expose the causes of parallelisation (in)efficiencies? The approach allows us to fully assess the performance and the quality of parallelisations of the Saturation algorithm in this thesis. This contribution should also hopefully spare newcomers to the area from having to learn these lessons the hard way, as we did over a painful period during this course of this thesis.

1.1.2 Contribution 2: a hand crafted parallel variant of Saturation

The second contribution is a hand crafted parallel variant of Saturation. This contribution offers a potential solution for scheduling and load balancing requirements, in particular for symbolic algorithms that are highly optimised and mutually recursive. The algorithm provides a detailed understanding of what is needed to efficiently parallelise the firing of events within Saturation. Key to the algorithm is how to manage the dependency
of tasks without forcing computation threads to frequently idle. To this end, we propose a task queue for storing tasks that need to be processed, from which available compute nodes can pick jobs. However, letting the operating system manage tasks is very costly, due to the overheads involved when creating threads. Consequently, we implement our own thread pool using Pthreads [21], that minimises these overheads. Another challenge is how to group firings of events such that our tasks, while still being lightweight, become sizable. Our solution here is to consider firing several events for a given MDD node within the same task.

1.1.3 Contribution 3: a parallel variant of Saturation using Cilk

The third contribution is a parallel variant of Saturation using Cilk [10, 49], a parallel language for addressing irregular problems. This contribution takes a step towards defining a generic re-usable solution for parallelising symbolic state-space generators. Investigating the use of Cilk provides an opportunity to assess the use of languages that address irregular problems to parallelise Saturation. Contrasting the Cilk implementation to the thread pool algorithm allows the language to be fully assessed as to its suitability for the task.

1.2 Structure of this thesis

Chapter 2

We provide an introduction to reachability analysis, covering the main advancements in the field that address the state-explosion problem. The existing literature on parallel state-space construction is also covered.
Chapter 3

The Saturation algorithm is presented, covering structured systems, symbolic representation of the next-state function and the Saturation iteration strategy. We also describe how Saturation is implemented in the SMART environment, and show the problems encountered when parallelising Saturation using C++.

Chapter 4

The issues of how to measure and evaluate parallel state-space construction algorithms are addressed, and an approach to measurement is suggested. Using this approach we select and justify a benchmark to evaluate our parallel algorithms.

Chapter 5

We parallelise the Saturation algorithm, as well a breadth first search for comparison. We employ a thread pool for the parallelisation, and investigate optimisation issues for efficient parallelisation. Run-time and memory results on a dual-processor dual-core architecture are presented for our thread pool implementation.

Chapter 6

The parallel language Cilk is applied to parallelise Saturation. We investigate issues relating to memory efficiency, according to the underlying parallelisation model of the language. We compare run-time and memory results for the Cilk algorithm to our thread pool implementation, to put the performance of the algorithms into context.
Chapter 7

We investigate a parallel approach known as *anticipated firing* [22, 24], which has previously been employed for a parallel variant of Saturation on a NOW, to utilise idle workstations for performing work that may be used in the future by the state-space generation task. We use this approach to parallelise the Saturation algorithm on a shared-memory architecture. We compare the run-time and memory results of our anticipated firing algorithm to our Cilk and thread pool implementations. Doing so allow us to assess the idle core utilisation of anticipated firing, compared to underlying algorithm parallelisations using our thread pool and Cilk.

Chapter 8

We summarise the work presented in the thesis and draw conclusions from the results obtained from the thread pool and Cilk algorithms, and also suggest possible future work in this area.
Chapter 2

Reachability analysis

Reachability analysis is the primary task of the model checking decision procedure. The process begins from an initial state and explores all of the transitions that lead to new states, to construct a state-space representing all of the reachable states. However, the reachable state-space must be constructed efficiently enough to fit into memory, due to the state-explosion problem. Techniques developed to address the state-explosion problem have resulted in two general ways to construct a state-space, either explicitly [32, 88] or symbolically [20, 79]. In this chapter we look at these methods for constructing reachable state-spaces, including techniques for addressing the state-explosion problem.

2.1 Temporal logic model checking

We begin the background to this thesis by providing an introduction to temporal logic model checking [32, 88] an automatic verification method that proves whether properties hold for a model of a system. It can be expressed as

\[ P \models \Phi \]
where $P$ is a finite state machine representing the system model and $\Phi$ is a temporal logic formula [86] expressing the properties that need to be checked. The logical satisfaction relation $\models$ is a decision procedure known as a model checker. The first model checkers were developed independently in the early 1980’s by Quielle and Sifakis [88] and Clarke and Emerson [32].

The starting point for using a model checker is to provide a description of $P$. Typically the model consists of a start state $s_0$ and a next-state function that describes the transitions between states $\mathcal{N}(s) = \{s'|s \rightarrow s'\}$. However, specifying individual transitions is impractical and thus a finite state machine representing the system model is often compiled from a high level system description such as a Petri net. For this reason, model checkers often provide their own language that can be conveniently used to describe the system [37, 78, 80]. As an alternative to manually describing systems, system descriptions can sometimes be automatically generated from an implementation of a system, using the programming language, or compiled program [5, 35, 38, 61, 64].

Given the description of a model, one can specify properties along computational paths or trees, and request the model checker to decide whether they are satisfied within the model. The automated nature of this process is a driver behind the adoption of model checking by industry [31, 78]. Another key advantage is the ability of the model checker to show counterexamples for properties that do not hold, i.e., the steps that were taken to violate the property. This information can be used to determine the cause of subtle errors. Typically, two types of properties are distinguishable, safety and liveness [74].

- Safety Properties: relate to the safety of the model with regards to whether something won’t occur.
• **Liveness Properties**: refer to whether something will occur.

For each of these properties the problem can be reduced to one of reachability analysis [8], which is the primary task of the model checking decision procedure [99].

### 2.1.1 The state-explosion problem

During reachability analysis, a state-space $S$ containing the reachable states of the model is constructed. The size of $S$ is a factor that affects the feasibility of the task. If $S$ is too large to fit into memory the problem of constructing a state-space becomes intractable in main memory. If $S$ is sufficiently small to fit into the memory of the system then the size of $S$ can still radically influence the run-time efficiency of the process, in some cases, detrimental to the point where it becomes impractical to use a model checker.

Models of asynchronous systems are a particular problem when the size of the state-space is a consideration. The widely employed interleaving semantics of an asynchronous system allows for an arbitrary ordering of concurrent events. This means that all of the events can be interleaved in all possible ways. If there are $n$ concurrent events then there are $n!$ potential orderings and up to $2^n$ states. The number of processes in an asynchronous system can therefore cause the number of states to grow exponentially. This phenomenon is known as the *state-explosion problem*. As a result model checking is often impractical for asynchronous systems with large numbers of processes and transitions [85].

### 2.1.2 Reachability analysis

The process of reachability analysis begins from an initial state and explores all of the transitions that lead to new states, to construct a state-space
representing the reachable states of the system. To illustrate that the model checking decision procedure can be reduced to one of reachability analysis, we use an illustrative example of a system model, shown in figure 2.1(a). The model has six states and nine transitions between the states. A safe state $s_3$ is included in the model. We ask whether the system in figure 2.1(a) can be brought into a safe state at any point during its operation.

The first step in answering this question is to compute the set $S$ of states that can reach the safe state $s_3$. These states are shown in 2.1(b) where $S=\{s_0,s_2,s_3,s_5\}$. The next step is to compute the set $R$ of states that are reachable from the start state $s_0$. These states are shown in 2.1(c) where $R=\{s_0,s_1,s_2,s_3,s_4\}$. The answer to the question is determined by whether $R \subseteq S$. In this example, the system can be demonstrated to be unsafe by computing $S$ and $R$ through reachability analysis.

### 2.2 Explicit reachability analysis

Using the start state $s_0$, and the next-state function $N$, an explicit reachability analysis algorithm constructs a state graph that explicitly represents all of the reachable states. We illustrate the explicit algorithm in figure 2.2. If $S$ is a set of enumerated states and $U$ is a set of unexplored states, then states are discovered by repeatedly applying $N$ to a state removed from $U$. 
ExplicitGen\( (s_0: \text{state}, N: \text{stateset} \rightarrow 2^{\text{stateset}}) : \text{stateset} \)

\[
\begin{align*}
\text{declare } & S, U, X : \text{stateset}; \\
1. & S \leftarrow \{s_0\}; \\
2. & U \leftarrow \{s_0\}; \\
3. & \text{while } U \neq \emptyset \text{ do} \\
4. & \quad \text{pick and remove a state } i \text{ from } U; \\
5. & \quad L = N(i); \\
6. & \quad \text{while } L \neq \emptyset \text{ do} \\
7. & \quad \quad \text{pick and remove a state } j \text{ from } L; \\
8. & \quad \quad \text{if } j \notin S \text{ then} \\
9. & \quad \quad \quad S \leftarrow S \cup \{j\}; \\
10. & \quad \quad \quad U \leftarrow U \cup \{j\}; \\
11. & \text{return } S;
\end{align*}
\]

Figure 2.2: Explicit algorithm for state-space construction.

If a state is discovered, it is checked against \( S \), and if it is new, is added to \( S \) and \( U \). The process continues until \( U = \emptyset \), at which point \( S \) contains all the reachable states.

The order in which states are removed from \( U \) defines the type of search strategy used to construct the state-space. If a stack is used to represent \( U \) then it is a depth first search (DFS), whereas if a queue is used to represent \( U \) then it is a breadth first search (BFS). Explicit reachability analysis, is typically performed using a DFS, since it is able to detect graph patterns such as strongly connected components (SCCs) which can be used for liveness checking. For this reason, Tarjan’s DFS for detecting SCCs [96] is the classic algorithm for explicit state-space construction.

During explicit reachability analysis, the state graph can become too large to fit into memory, due to the state-explosion problem. Explicit approaches that address the state-explosion problem, attempt to reduce the number of states that are required to be enumerated during the construction of the reachable state-space. Explicit model checkers often contain a number of state-space reduction techniques to make their usage practical [39, 63, 65].

- A partial order reduction [98, 65] attempts to reduce the number of
transitions that need to be executed during state-space construction when an asynchronous system is considered, by exploiting the independence of concurrently executed transitions.

• *Symmetry reduction* [43] exploits the symmetry in the design and implementation of computer systems consisting of multiple identical components to create a reduced system model, where parts of the system are equivalent.

• *Bisimulation minimisation* [83], creates a minimised model, whose truth and falsehood of the temporal logic properties are preserved if it *bisimulates* the original model.

• *Abstraction* [33] is a technique that aims to reduce the size of the model by removing redundant information that is not required for the model checking process.

• *Compositional reasoning* [34] addresses the state-explosion problem by breaking a system down into sub-components and model checking those components individually.

• *On-the-fly techniques* [36] attempt to avoid generating an entire state-space by checking properties of the temporal logic formula during state-space construction.

### 2.3 Symbolic reachability analysis

Explicit state-spaces are reliant upon the reduction of the state-space size before the state-space is built. In contrast, *symbolic techniques* [20, 79] replace the explicit representation with a data structure that facilitates a more compact encoding of the states and transitions, while still allowing for
The cornerstone of the development of symbolic model checkers lies in the thesis of McMillan [79], where binary decision diagrams (BDDs) [16] were first used to represent state-spaces. The use of BDDs led to a significant increase in the number of states that could be represented, from approximately $10^7$ states to $10^{20}$ states [20]. This increase has allowed symbolic model checkers [14, 31, 79] to become successful in practice, in particular for hardware verification [42].

### 2.3.1 Binary decision diagrams

BDDs can be used to compactly represent boolean formulas. To illustrate this with an example, consider the boolean formula $x_1 \lor x_2 \land x_1 \lor x_3$. The truth values of the boolean formula can be represented by a binary decision tree shown in figure 2.3(a). The binary decision tree is the equivalent size of a truth table representing the formula. Removing redundant information from the tree in the following way results in a more compact representation.
• A non-terminal node with two equal successors is redundant.

• Two nodes with the same children are equivalent, thus one node is redundant.

The redundant nodes can be removed during a three step process. The first step is to merge the non-unique terminals as shown in 2.3(b). The second step is to merge the nodes that duplicate each other as shown in 2.3(c). The final step is to remove non-terminal nodes with two equal successors as shown in 2.3(d), which results in a reduced ordered BDD (ROBDD).

Formally, a BDD is a rooted directed acyclic graph (DAG) with terminal and nonterminal nodes, representing a boolean function of \( n \) binary variables, \( f : \{ 0, 1 \}^n \rightarrow \{ \text{false, true} \} \). The terminal nodes correspond to the boolean values false and true. Each nonterminal node is labelled with a variable \( x_i \), \( 1 \leq i \leq n \), and has two successors corresponding to the boolean values 0 and 1. A BDD defines a boolean formula whose truth values can be determined by following the path from the root node to a terminal node.

It is important that a BDD takes a canonical form, to allow for operations between BDDs. BDDs representing the same boolean formula must be isomorphic to facilitate a simple equivalence test between BDDs. To facilitate a canonical representation, a total ordering over the variable identifiers in the boolean formula is imposed on the BDD, and duplicate nodes are not allowed to exist in the BDD. Removing redundant nodes results in a ROBDD. However, it is often desirable to leave redundant nodes in the BDD resulting in a quasi-reduced ordered BDD (QOBDD) for simpler algorithmic manipulation [29].

The choice of variable ordering can significantly influence the size of a BDDs. Unfortunately, checking that a variable ordering is optimal is
an NP-complete problem [17]. The only way in which a suitable variable ordering can be chosen is by using a heuristic to guide the choice [50, 51]. When no suitable heuristic can be found, dynamic variable reordering can be used, which rearranges the variables every so often to reduce the number of nodes [90].

### 2.3.2 Constructing BDDs

There are a number of operations that can be performed on BDDs, including simple algorithmic ways to reduce the BDD, and restrict it to a canonical form [16]. In this section, we describe the `apply` algorithm which performs operations between BDDs, and can be used to construct a BDD for any binary operation $f \ast g$ [16].

We show the apply algorithm in figure 2.4, which is based on the Shannon expansion: $f = \bar{x} \cdot (f|_{x=0}) + x \cdot (f|_{x=1})$. The algorithm accepts two BDDs, and constructs a new BDD by recursively expanding nodes $f$ and $g$, from the root node to the terminal nodes. We illustrate this order of construction for the apply algorithm in figure 2.5b, for an operation between the two BDDs.
in figure 2.5a. The Shannon expansion is used in lines 5 and 6 of the apply algorithm to expand a node into its descendant nodes. The nodes passed to the recursive apply calls are made according to tests on the variables in lines 8 to 12 of the algorithm. The test statement \( x = y \) holds for BDDs of equal height, and defined over the same ordered set of variables.

To complete the construction of the BDD in figure 2.5b, it would be further reduced and restricted. One can see from this simple example that the intermediate BDD during BDD construction can be larger than the final BDD. Another issue is the recursive nature of the apply algorithm, which defines its time complexity as exponential. By using a cache to store the previous results of apply, the time complexity can be reduced to polynomial.

### 2.3.3 Symbolic representation of \( S \) and \( N \)

Up until now we have described BDDs for representing boolean functions. We can make the transition between storing functions and sets by using the characteristic function of a set. Given this, a set of states \( S \) can be encoded using binary variables. For example, the two bit binary counter in figure 2.6 can be encoded as a BDD using the binary variables \( x_1 \) and \( x_2 \). This form of encoding is a particular reason as to why symbolic BDD based model checking is popular for hardware verification such as synchronous circuits.
Figure 2.6: A two bit binary counter.

\[ N \text{ can be encoded using the source and target variables. The next-state function in for our example in figure 2.6 would be encoded as } (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \lor (\neg x_1 \wedge x_2 \wedge x_1' \wedge \neg x_2') \lor (x_1 \wedge \neg x_2 \wedge x_1' \wedge x_2') \lor (x_1 \wedge x_2 \wedge \neg x_1' \wedge \neg x_2'). \]

Thus, by using the apply algorithm for construction, the next-state function can also be encoded as a BDD.

Representing \( N \) in this manner can lead to a large monolithic next-state function. However, \( N \) can be \textit{conjunctively} or \textit{disjunctively partitioned} [19] into a number of smaller constituent parts. In particular, for asynchronous systems, it is often beneficial to disjunctively partition \( N \), to reduce the size of intermediate BDDs during state-space generation [19].

### 2.3.4 Symbolic state-space generation

The classic BFS algorithm for symbolic state-space construction is shown in figure 2.7. Beginning with the start state \( s_0 \), a set of reachable states \( S \) and a set of unexplored states \( U \), the next-state function \( N \), is applied to \( U \), to generate a set of potential new states \( X \). The set-difference operator is then used on \( X \) and \( S \) to determine the new (unexplored) states. The new states are then added to the set of reachable states: \( S \cup U \). The process is repeated until there are no more states to explore, i.e., \( U = \emptyset \). The application of the next-state function during the BFS is known as the \textit{image computation}, which is based on logical operations between the BDD representing \( N \), and the BDD representing \( U \). Since the classic BFS algorithm for symbolic state-
space construction is based on logical operations between BDDs representing sets of states, the apply algorithm is used to construct the BDDs.

The repetitive nature of applying the next-state function, and the use of a cache for storing previous results of BDD construction operations implies that the data structures used to store BDDs must facilitate efficient insertion and retrieval. This is achieved by using hash tables [31], which applies to tables storing constructed BDDs, referred to as a unique table [31], and caches for storing previous results of BDD construction. Packages such as CUDD [93], provide generic implementations of BDDs, including data structures such as hash tables for BDD storage and caching.

2.3.5 Iteration strategy

The BDD construction algorithm spends most of its time performing the image computation, typically 80-90% of state-space generation time [89]. It is often the case that the BDD size is greater during state-space construction than the size of the completed BDD, due to the large intermediate BDD sizes resulting from image computation.

An approach to dealing with this involves the use of chaining [84]. This method can be applied when the next-state function \( \mathcal{N} \) is disjunctively partitioned, i.e., if \( e \) represents an event (or transition) between states, then

\[
\text{BFSGen}(s_0: \text{state}, \mathcal{N}: \text{stateset} \rightarrow 2^{\text{stateset}}): \text{stateset}
\]

\[
\begin{align*}
\text{declare } & S, U, X: \text{stateset}; \\
1. & S \leftarrow \{s_0\}; \\
2. & U \leftarrow \{s_0\}; \\
3. & \text{while } U \neq \emptyset \text{ do} \\
4. & X \leftarrow \mathcal{N}(U); \\
5. & U \leftarrow X \setminus S; \\
6. & S \leftarrow S \cup U; \\
7. & \text{return } S;
\end{align*}
\]

Figure 2.7: BFS algorithm for symbolic state-space construction.
ChainedBFSGen\((s_0; \text{state}, \mathcal{N}_e \in \mathcal{E}; \text{stateset} \rightarrow 2^{\text{stateset}})\): stateset

declare \mathcal{S}, \mathcal{U}, \mathcal{X}: \text{stateset};
1. \mathcal{S} \leftarrow \{s_0\};
2. \mathcal{U} \leftarrow \{s_0\};
3. \text{while } \mathcal{U} \neq \emptyset \text{ do}
4. \text{ for each } e \in \mathcal{E} \text{ do}
5. \quad \mathcal{U} \leftarrow \mathcal{U} \cup \mathcal{N}_e(\mathcal{U});
6. \quad \mathcal{U} \leftarrow \mathcal{X} \setminus \mathcal{S};
7. \quad \mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{U};
8. \text{return } \mathcal{S};

Figure 2.8: Chained BFS algorithm for symbolic state-space construction.

\(\mathcal{N}_e \in \mathcal{E}\) is disjunctively partitioned by event. The chaining approach modifies the BFS to apply the disjunctively partitioned next-state function incrementally, which can reduce the size of the intermediate iterations. A chained BFS algorithm is shown in figure 2.8. The algorithm is modified from the original BFS algorithm, so that the set of unexplored states \(\mathcal{U}\) is modified by applying \(\mathcal{N}\) iteratively, which allows any new states to be explored as soon as they have been discovered. This can result in smaller intermediate BDDs, when compared to applying a single monolithic transition to the entire decision diagram to produce the new state set.

### 2.4 Parallel state-space construction

Parallel state-space construction algorithms incur a number of parallel overheads, which is due to their irregular nature in terms of unpredictable sizes of work [77, 87, 100] and random access to data structures such as hash tables [101, 102]. These overheads are load imbalance from idle processors when work is not distributed evenly, scheduling costs for creating parallel tasks, and synchronisation on the data structures.

Most of the existing approaches to parallel state-space construction have focused on networks of workstations (NOW) for the purpose of increasing
the available memory for the representation of the state-space. Static partitioning methods are frequently applied to slice the data structure, distributing it as evenly as possible across the nodes of the network. This is often simpler for explicit approaches [7, 13, 53, 72, 75], since the states only have to be distributed individually. In symbolic state-space construction [23, 58, 59, 62, 81, 95], slicing the data structure is a more complex task. However, key to the effectiveness of both is achieving an even distribution of states to avoid a load imbalance.

Dynamic approaches have also been developed that either redistribute the data structure during state-space construction [59], or focus on a dynamic parallelisation of the algorithm [66]. These techniques are more sophisticated than static approaches, since they must keep an even balance of work or state distribution during construction, rather than taking a “partition once” approach. Novel approaches such as attempting to remove the requirement for communication [13, 59], or tailoring the algorithm to the architecture [66] have also been employed to improve parallelisability.

### 2.4.1 Parallel explicit approaches

Amongst the notable work on static slicing on a NOW, Stern and Dill [94] carried out the earliest work on parallel state-space construction, by parallelising the Murϕ verifier. During the construction task, random load balancing is employed to distribute states across a NOW. A hash function decides to which node a state gets sent for processing. The parallelisation was effective, showing linear speed-ups, however when this technique was used for other state-space exploration algorithms mixed results were reported [7, 75], depending on the effectiveness of the calculation of the next-state function compared to the cost of state distribution. If the next-state function is quite costly to calculate then the distribution costs are negligible.
Lerda and Sisto’s [75] implementation of this technique in SPIN subsequently reported slow-downs due to the efficiency of SPIN’s next-state calculation. Behrmann et al.’s [7] algorithm demonstrated a superlinear speed-up using the technique, however, since the algorithm was heavily modified to incorporate a different search strategy, it is difficult to determine whether the speed-up arises from the search strategy or the parallelisation.

One of the main costs for parallel state-space construction on a NOW is the cost of communication across workstations, due to the frequent checking of duplicate states between nodes. This is a significant barrier to run-time efficiency, as a workstation must synchronise when it discovers a new state before continuing its work. A novel idea for addressing this problem was introduced by Bollig et al. [13], by developing an algorithm that eliminates cycles within the algorithm. This effectively allows the state-space to develop with the distributed algorithm running its component parts independently. The removal of cycles resulted in effective linear speed-ups due to the elimination of related synchronisation costs.

The only work on dynamic parallel state-space construction has been carried out by Inggs and Barringer [66, 67, 68] on a shared-memory architecture. The approach employs workstealing [2, 11, 12] techniques in order to load balance. Many of the parallelisation overheads, such as synchronisation, are addressed by tailoring the parallelisation specifically to the selected architecture. In particular, employing a private queue per thread for unexplored state storage reduces synchronisation overhead and scheduling cost, which is greater than the extra cost of exploring duplicate states that occurs as a result of employing private queues. Other parallelisation overheads such as false sharing [73] which unnecessarily updates processor caches as a result of data being shared between processor caches, are eliminated by carefully designing the data structures. The optimisation of the algorithm for
the architecture allows the parallel algorithm to effectively reduce parallel overheads, resulting in good linear speed-ups for several models.

2.4.2 Parallel symbolic approaches

We can classify the symbolic parallelisation approaches into either memory efficiency based approaches [23, 59, 62, 81, 95], memory/time efficiency based approaches [58], or idle processor usage [22, 24]. In contrast, this thesis focuses on a purely time efficiency based approach.

Data parallelisation (memory):

For parallelisation of the underlying symbolic data structure, the efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [81, 95] to sophisticated approaches relying on workstealing [59]. These approaches target the increased memory available on a NOW by slicing the data structure and distributing it across processors of the NOW. The structure of decision diagrams has previously been sliced horizontally [23] and vertically [62, 81, 95]. Horizontal slicing scales well but prevents the state-space generation task from being speeded up, since each slice has to complete its work before the next slice begins its work. Finding a good vertical slicing is a non-trivial issue often leading to poor scalability. In order to facilitate scalability, load balancing techniques need to be employed. The most advanced work in this area uses workstealing techniques to distribute work dynamically [59].

Data and algorithm parallelisation (memory/time):

Researchers have parallelised symbolic state-space generation algorithms to gain speed-ups from developing vertical slices on different processors of a NOW [58]. If the algorithm developing the slices has to frequently synchro-
nise on the application of the next-state function, each round of computation is only as fast as the slowest time it takes for a slice to develop on a processor. For the algorithm to achieve speed-ups the research tackles the difficult task of removing the synchronous nature of the algorithm. The parallel algorithm allows slices to develop asynchronously while the next-state function is repeatedly applied to create more work. The work is load balanced using the workstealing techniques developed in [59]. For very large circuits this technique has proved to lead to a very efficient parallelisation showing up to an order of magnitude improvement in time efficiency.

**Utilising idle processors (memory/time):**

Recent work has also considered ways to utilise idle processors during state-space construction [22, 24]. The idle processors are used to perform and cache work that may be performed in future, while a main processor develops the state-space. If work that the main processor requires has already been performed by another processor, the main processor retrieves it from the cache. This reduces the peak size of the data structure during state-space construction and improves time efficiency if the amount of utilised work performed by the idle processors is sufficient to overcome the overhead of allocating work to the processors and synchronising on the cache.

### 2.5 Evaluations of parallel state-space exploration algorithms

The evaluation of parallel state-space exploration algorithms usually involves benchmarking in terms of run-time and some estimation of the work distribution and communication overhead. Since a run-time breakdown of parallel overheads are generally not included, it is difficult to determine how the
algorithms perform under different conditions imposed upon them by the benchmark’s models. To the best of our knowledge, only one paper gives a breakdown of the actual parallel overheads at run-time [62].

Publications of explicit parallel state-space exploration algorithms employing static partitioning on a NOW typically use one to five models to benchmark an algorithm according to its run-time [7, 13, 53, 72, 75, 94]. For example, the evaluation of the parallel Murϕ verifier in [94] estimates the performance of the algorithm according to the communication overhead and the partitioning method, but demonstrates the accuracy of the estimation for only three models. A more thorough theoretical analysis of a parallel state-space exploration algorithms’ predicted performance was carried out in [72], but is only demonstrated for a single model, that is passed a parameter which results in different model sizes, and resultant state-space sizes. Artificial models were constructed to evaluate a parallel negative cycle detection algorithm in [15]; however, only the run-time and number of messages passed between workstations is given as an indicator of performance.

Publications of symbolic parallel state-space exploration algorithms employing static partitioning on a NOW generally take a similar approach to evaluation, by using two to four models to produce run-times and memory consumption as a performance measure [23, 81]. To evaluate the algorithm for the parallel construction of BDDs on a shared-memory architecture [71], a small number of models is used, and in terms of run-time only. The evaluation of the parallel symbolic algorithm on a NOW in [95] provides more information, but these are only estimates of overheads, such as the number of messages passed between workstations to quantify communication overhead.

Dynamic approaches to symbolic parallelisation on a NOW appear to contain a more complete evaluation of performance. Heyman et al. [62] pro-
vide a thorough breakdown of the time spent in component parts of their algorithm, such as communication, sequential execution and memory balancing, using seven models. This gives a clear picture of how the algorithm performs. However, how the timings were measured and whether they were instrumented in the code, or obtained using a profiling tool, is not described in the paper. Grumberg et al.’s parallel symbolic state-space exploration algorithm on a NOW using dynamic load balancing [58], is evaluated using an estimate of how the load is balanced according to the dynamic partitioning method used. The benchmark employs nine models for evaluation, and the authors select models that exhibit a varying range of run-time speed-ups, between little over one to an order of magnitude. However, a breakdown of the individual overheads is omitted from the evaluation. Inggs and Barringer’s work using workstealing for explicit state-space exploration on a shared-memory architecture [66, 67, 68] considers the overheads associated with the parallelisation. They mention that direct measurement of some of the overheads is difficult and thus are unable to provide direct measurements when reporting results [67]. However, a theoretical model of performance is instead proposed to analyse the efficiency of the algorithm [67].

2.6 Chapter summary

In this chapter we introduced the methods for constructing reachable state-spaces, and the techniques for dealing with the state-explosion problem. In particular, we illustrated symbolic techniques based on BDDs. We also reviewed the current literature on parallel reachability analysis, including how the proposed parallel algorithms have been evaluated.
Chapter 3

Saturation

Traditional symbolic approaches treat the system model as *unstructured* by ignoring information about the system when constructing the state-space. However, the information contained within a *structured* system can be used to compute state-spaces more efficiently, when considering structure-rich classes of systems such as communication protocols and distributed systems. The unique, efficient search strategy of *Saturation* [29] is rooted in the exploitation of the structure-rich class of event-based asynchronous systems. Experimental results reported in [25, 29, 30] consistently show that Saturation outperforms breadth-first symbolic state-space generation by orders of magnitude in both memory and time, making it arguably the most efficient state-space generation algorithm for globally-asynchronous locally-synchronous discrete event systems. In this chapter we describe structured systems and illustrate the iteration strategy of the Saturation algorithm.

3.1 Introduction

The Saturation algorithm [29] that is implemented in the verification tool SMART [26], is radically different from its symbolic predecessors. It consists of a series of small, nested fixed point operations that are guided by the
current shape of the decision diagram, with the goal of systematically saturating decision diagram nodes in a bottom-up fashion. The building block of this strategy is the firing of an individual event in an individual node, which encodes a subset of states, in contrast to computing the entire next-state function on the entire current set of states. This finer-grain decomposition of symbolic operations is more flexible, by allowing more efficient firing orders, while exploiting the event locality property, that is inherently present in concurrent, asynchronous systems. In our setting, the system is structured if it consists of a collection of subsystems, such that the global system state can be written as a vector of local states and the effect of an event on the system state can be expressed as the composition of the local effects of the event on each subsystem. For structured systems, an encoding of sets of states with multi-way decision diagrams (MDDs) [70] is more natural, because the one-to-one correspondence between a state variable and the level in the MDD is always apparent.

### 3.2 Structured systems

We begin with a description of structured systems using our definitions from [45].

**Definition** A discrete-state model is a triple \(( \hat{S}, s^0, N )\), where

- \( \hat{S} \) is the set of potential states of the model.
- \( s^0 \in \hat{S} \) is the initial state.
- \( N : \hat{S} \rightarrow 2^{\hat{S}} \) is the next-state function specifying the states reachable from each state in one step.

For structured systems, high level models can be naturally partitioned into submodels. For many asynchronous systems, this decomposition is
already the case. Assuming that a model contains $K$ submodels, a (global) state $i$ is a $K$-tuple $(i_K, ..., i_1)$, where $i_k$ is the local state of submodel $k$, for $K \geq k \geq 1$, and $\hat{S} = S_K \times \cdots \times S_1$ is the cross-product of $K$ local state-spaces. This allows symbolic techniques based on decision diagrams to be used for storing sets of states. The least fixed point computation can be used to generate the reachable state-space $S \subseteq \hat{S}$, which is the smallest set containing $s^0$ and closed with respect to $N$: $S = \{s^0\} \cup N(s^0) \cup N(N(s^0)) \cup \cdots = N^*(s^0)$, where “*” denotes reflexive and transitive closure and $N(X) = \bigcup_{i \in X} N(i)$.

3.2.1 Symbolic encoding of $S$

For the encoding of $S$, the assumption is made that each $S_k$ is finite and known a priori. In practice, the local state-spaces $S_k$ can actually be generated on-the-fly by interleaving symbolic global state-space generation with explicit local state-space generation [30]. Without loss of generality, the assumption is made that $S_k = \{0, 1, \ldots, n_k-1\}$, with $n_k = |S_k|$. Any set $X \subseteq \hat{S}$ can then be encoded in a (quasi-reduced ordered) MDD over $\hat{S}$. Formally, an MDD is a directed acyclic edge-labelled multi-graph where:

- Each node $p$ belongs to a level $k \in \{K, \ldots, 1, 0\}$, denoted $p.lvl$.
- There is a single root node $r$ at level $K$.
- Level 0 can only contain the two terminal nodes Zero and One.
- A node $p$ at level $k > 0$ has $n_k$ outgoing edges, labelled from 0 to $n_k - 1$. The edge labelled by $i_k$ points to a node $q$ at level $k-1$; we write $p[i_k] = q$.
- Given nodes $p$ and $q$ at level $k$, if $p[i_k] = q[i_k]$ for all $i_k \in S_k$, then $p = q$, i.e., there are no duplicates.
MDDs are able to take values over finite sets without encoding them as boolean functions, making them a more natural form than BDDs for representing the state-space of a structured system. The set encoded by an MDD node $p$ at level $k > 0$ is $B(p) = \bigcup_{i_k \in S_k} \{i_k\} \times B(p[i_k])$, letting $X \times B(0) = \emptyset$ and $X \times B(1) = X$ for any set $X$.

**Example MDD**

To illustrate MDDs with an example, figure 3.1 shows an MDD corresponding to 4 submodels, $S_1 \times S_2 \times S_3 \times S_4$. The states encoded by the MDD, $S$, are explicitly shown to the right of the diagram. The inclusion of a state in the set can be determined by tracing from the root node of the MDD to the terminal nodes. Any state that can be traced from the root node to the terminal node 1, is included in the set. The indices 0 and 1 are reserved for each level $k$, encoding the sets $\emptyset$ and $S^k \times \cdots \times S^1$ respectively. This allows for only the highlighted nodes in the example to be stored.

### 3.2.2 Symbolic encoding of $N$

For asynchronous systems, it is often beneficial to disjunctively partition $N$ [19]. Traditionally, where disjunctive partitioning is used, $N$ is partitioned by event. For Saturation, a representation inspired by work on Markov chains is adopted, resulting in a partitioning of $N$ by event and
CHAPTER 3. SATURATION

Formally, $\mathcal{N}$ is decomposed into a disjunction of next-state functions, so that

- $\mathcal{N}(i) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(i)$, where $\mathcal{E}$ is a finite set of events and $\mathcal{N}_e$ is the next-state function for event $e$.

- Each $\mathcal{N}_e$ is conjunctively decomposed into $K$ local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, satisfying $\mathcal{N}_e(i_K, \ldots, i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$, in any global state $(i_K, \ldots, i_1) \in \hat{S}$.

- The decomposition uses $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0,1\}^{n_k \times n_k}$ with $\mathbf{N}_{k,e}[i_k, j_k] = 1 \iff j_k \in \mathcal{N}_{k,e}(i_k)$.

- $\mathcal{N}_e$ is encoded as a boolean Kronecker product:

  \[ j \in \mathcal{N}_e(i) \iff \bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,e}[i_k, j_k] = 1, \]

  where $\otimes$ indicates the Kronecker product of matrices.

Initially the decomposition required the model to be Kronecker consistent [29], a restriction that can often be automatically satisfied by concurrency models such as Petri nets. However, this restriction was lifted in [25].

The next-state function can be stored very efficiently using a Kronecker matrix, when compared to storing the next-state function as a decision diagram. The $\mathbf{N}_{k,e}$ matrices are extremely sparse; when encoding a Petri net, for example, each row contains at most one nonzero entry.
Example model

To illustrate the symbolic encoding of $\mathcal{N}$, a partitioning of a Petri net into 3 subsystems (submodels) is shown in figure 3.2. The subsystems communicate via shared events. Each of the possible states of the subsystems are highlighted: subsystem 1 has 2 states, subsystems 2 and 3 have 4 states.

Table 3.1: Kronecker representation for the Petri net example.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$I$</td>
<td>0:1,2:3</td>
<td>0:2,1:2</td>
<td>3:0</td>
</tr>
<tr>
<td>2</td>
<td>0:1</td>
<td>1:2,3:0</td>
<td>1:3,2:0</td>
<td>$I$</td>
</tr>
<tr>
<td>1</td>
<td>0:1</td>
<td>$I$</td>
<td>$I$</td>
<td>1:0</td>
</tr>
</tbody>
</table>

The corresponding Kronecker representation for this example is shown in table 3.1, containing the transitions between states via each event within each subsystem. Every row representing the transitions for a subsystem can...
be mapped directly to a level of an MDD. Event $e$ is *independent* of level $k$ if $N_{k,e} = I$, the identity matrix, i.e., if event $e$ does not affect the local state of that subsystem. This allows the efficiency of the representation of the next-state function to be improved, since identity matrices do not need to be stored.

**Event locality**

In the Kronecker representation, event $e$ is independent of level $k$ if $N_{k,e} = I$. Thus, the Kronecker representation allows *event locality* [29] to be recognised. This property can be used to avoid unnecessary work [28] when constructing the MDD. Let $\text{Top}(e)$ denote the highest level for which $N_{k,e} \neq I$. Then the firing of an event $e$ does not need to begin at the root of the MDD. If $\text{Top}(e)=k$, then $e$ can be fired only on MDD levels including and below $k$.

To illustrate this, we first introduce the notion of *enabledness*. An event $e$ for which $\text{Top}(e) = k$, is *enabled* by the local state $i$ of node $p$ at level $k$ if $p[i] \neq 0$, and $e$ contains a transition between local states $i \rightarrow j$ at level $k$. If $\text{Top}(e) = k$, $k < K$, then $e$ can be fired on the node $p$ at level $k$ rather than the root node at level $K$. This firing requires an *in place update* on node $p$: if $e$ was enabled by the local state $i$ of node $p$, and the firing of $e$ resulted in new node $q$ at level $k − 1$ then, if $e$ contains a transition between local states $i \rightarrow j$ at level $k$, we update local state $j$ *in place*, $p[j] = q \cup p[j]$.

In this sense, the update to the node can be made locally. This facilitates a saving by eliminating the extra nodes above $\text{Top}(e)$, which are traditionally required when applying the next-state function that constructs a new BDD from the root node downwards by using the apply algorithm. Similarly, if $\text{Last}(e)$ denotes the lowest level for which $N_{k,e} \neq I$ then $e$ can be applied only to the levels above $\text{Last}(e)$. 
3.3 Saturation-based iteration strategy

Given the encoding of the next-state function and our description of event locality, we can formally define what is required to compute a node.

**Definition** The MDD node $p$ at level $k$ is a fixed point with respect to all $N_e$ such that $\text{Top}(e) \leq k$, i.e., $S_K \times \cdots \times S_{k+1} \times B(p) = N_{\leq k}(S_K \times \cdots \times S_{k+1} \times B(p))$, where $N_{\leq k} = \bigcup_{e: \text{Top}(e) \leq k} N_e$. A node that has reached a fixed point is said to be saturated.

Computing the node to bring it to a fixed point is known as saturating a node. To saturate MDD node $p$ once all its descendants are saturated, it can be updated in place so that it encodes also any state in $N_{k,e} \times \cdots \times N_{1,e}(B(p))$, for all events $e$ such that $\text{Top}(e) = k$.

Saturation consists of many “lightweight” nested fixed-point iterations and is completely different from the traditional breadth-first approach that employs a single “heavyweight” global fixed-point iteration. The algorithm uses operations caches to further improve computational speed [26].

- A unique table $UT$ is used to detect duplicate nodes, $UT[k]$, for $K \geq k \geq 1$, for a node $p$, at level $k$, using a given key $p[0], \ldots, p[n^k-1]$.

- A firing cache, $FC[k]$, for $K \geq k \geq 1$, is used to retrieve a saturated node $s$ at level $k$ as a result of an event $e$ previously fired on a node $p$ at level $k$, where $\text{Top}(e) > k$ and $B(s) = \mathcal{N}_{\leq k}^e(\mathcal{N}_e(B(p)))$.

- A union cache, $UC[k]$, for $K \geq k \geq 1$, is used to retrieve a saturated node $s$ at level $k$ as a result of union previously performed on nodes $p$ and $q$ at level $k$, where $B(s) = B(p) \cup B(q)$.

The algorithm contains two main mutually recursive functions: Saturate calls RecFire to recursively perform the event firings while saturating nodes,
CHAPTER 3. SATURATION

50

Saturate\(^{(\text{in } k;l;e,p;\text{idz})}\)

Updating \(p\), a node at level \(k\) not in \(UT[k]\), in-place, to encode \(N_{\leq k}^{*}(B(p))\).

- declare \(e\): event;
- declare \(L\): set of lcl;
- declare \(f, u, s\): idx;
- declare \(i, j\): lcl;
- declare \(pCng\): bool;

1. repeat
2. \(pCng \leftarrow false\);
3. foreach \(e \in E^k\) do
4. \(L \leftarrow \text{Locals}(e, k, p)\);
5. while \(L \neq \emptyset\) do
6. \(i \leftarrow \text{Pick}(L)\);
7. \(f \leftarrow \text{RecFire}(e, k-1, p[i])\);
8. if \(f \neq 0\) then
9. foreach \(j \in N_k^l(i)\) do
10. \(u \leftarrow \text{Union}(k-1, f, p[j])\);
11. if \(u \neq p[j]\) then
12. \(p[j] \leftarrow u; pCng \leftarrow true\);
13. if \(N_k^l(j) \neq \emptyset\) then
14. \(L \leftarrow L \cup \{j\}\);
15. until \(pCng = false\);

RecFire\(^{(\text{in } e;\text{event},l;e,p;\text{idz})}\): idx

Building an MDD rooted at \(s\), a node at level \(l\), in \(UT[l]\), encoding \(N_{\leq l}^*(N_e(B(q)))\).

Return \(s\).

- declare \(L\): set of lcl;
- declare \(f, u, s\): idx;
- declare \(i, j\): lcl;
- declare \(sCng\): bool;

1. if \(l < \text{Last}(e)\) then return \(q\);
2. if \(\text{Find}(FC^l, \{q, e\}, s)\) then return \(s\);
3. \(s \leftarrow \text{NewNode}(l)\);
4. \(sCng \leftarrow false\);
5. \(l \leftarrow \text{Locals}(e, l, q)\);
6. while \(L \neq \emptyset\) do
7. \(i \leftarrow \text{Pick}(L)\);
8. \(f \leftarrow \text{RecFire}(e, l-1, q[i])\);
9. if \(f \neq 0\) then
10. foreach \(j \in N_k^l(i)\) do
11. \(u \leftarrow \text{Union}(l-1, f, s[j])\);
12. if \(u \neq s[j]\) then
13. \(s[j] \leftarrow u; sCng \leftarrow true\);
14. if \(sCng\) then \(\text{Saturate}(l, s)\);
15. \(\text{Check}(l, s); \text{Insert}(FC^l, \{q, e\}, s)\);
16. return \(s\);

Figure 3.3: Pseudo-code for the Saturation and RecFire functions.
while \textit{RecFire} calls \textit{Saturate} to saturate nodes that are created as a result of event firings. The pseudo-code for the main functions is shown in figure 3.3.

The \textit{Saturate} function contains a loop which repeats until no updates on a node \( p \) at level \( k \) are made by performing by firing events on \( p \), i.e., until the node is saturated. Within this loop, for each event \( e \) where \( e \in \mathcal{E}^k \), it finds the local states of \( p \) enabling \( e \) at line 4 of the function. For each local state \( i \) enabling \( e \), it calls \textit{RecFire} at line 7 of the function, passing \( e \) and \( p[i] \) as arguments to the function, to build an MDD encoding \( \mathcal{N}^*_{\leq k-1}(\mathcal{N}_e(\mathcal{B}(p[i]))) \).

If a node \( f \), the root node of the encoded MDD, is returned from \textit{RecFire}, then the \textit{Saturate} function makes a local update for each \( i \rightarrow j \) transition of the event, \( j \in \mathcal{N}_e^k(i) \) so that \( p[j] = f \cup p[j] \), between lines 10 and 12 of the function.

The \textit{RecFire} function begins by checking that the event \( e \) which is to be fired on node \( l \) at level \( k \) is not independent of level \( l \), i.e., \( l < \text{Last}(e) \). If \( e \) is independent of level \( l \) then node \( q \) is returned, since \( q \) is already a saturated node encoding \( \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(q))) \), because \( q \) is not affected by firing \( e \) upon it. If the pair \( \{q, e\} \) is found in the firing cache at level \( l \), then the MDD encoding \( \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(q))) \) is retrieved from the cache and returned. Otherwise, the function continues by creating a new node \( s \) at level \( l \). At line 5 of the function the local states of \( q \) enabling \( e \) are found. For each locally enabling state \( i \), \textit{RecFire} is called to recursively build the MDD encoding \( \mathcal{N}^*_{\leq l-1}(\mathcal{N}_e(\mathcal{B}(q[i]))) \). If a node \( f \), the root node of the encoded MDD, is returned from \textit{RecFire}, then the function makes a local update for each \( i \rightarrow j \) transition of the event, \( j \in \mathcal{N}_e^k(i) \) so that \( s[j] = f \cup s[j] \). If node \( s \) has been updated as a result of any recursive calls to \textit{RecFire} then, \( s \) is saturated, checked into the hash table, and added to the firing cache for the pair \( \{q, e\} \). The function returns \( s \), which is the root node of the MDD encoding \( \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(q))) \).
Generate\(\) (in \(\text{array}[1..K] \) of \(\text{lcl}\)) : \(\text{idx}\)

Build an MDD rooted at \(r\), a node at level \(K\), encoding \(N^*_X(s)\) and return \(r\), in \(UT[K]\).

declare \(r,p: \text{idx}\);
declare \(k: \text{lvl}\);
1. \(p \leftarrow 1\);
2. for \(k = 1\) to \(K\) do
3. \(r \leftarrow \text{NewNode}(k); r[s[k]] \leftarrow p;\)
4. \(\text{Saturate}(k, r); \text{Check}(k, r);\)
5. \(p \leftarrow r;\)
6. return \(r;\)

Union\(\) (in \(k: \text{lvl}, p: \text{idx}, q: \text{idx}\)) : \(\text{idx}\)

Build an MDD rooted at \(s\), in \(UT[k]\), encoding the Union of the nodes \(p\) and \(q\) at level \(k\). Return \(s\).

declare \(i: \text{lcl}\);
declare \(s, u: \text{idx}\);
1. if \(p = 1\) or \(q = 1\) then return \(1\);
2. if \(q = 0\) or \(p = q\) then return \(q\);
3. if \(p = 0\) then return \(p;\)
4. if \(\text{Find}(UC[k], \{p, q\}, s)\) then return \(s;\)
5. \(s \leftarrow \text{NewNode}(k);\)
6. for \(i = 0\) to \(n^k - 1\) do
7. \(u \leftarrow \text{Union}(k-1, p[i], q[i]);\)
8. \(s[i] \leftarrow u;\)
9. \(\text{Check}(k, s);\)
10. \(\text{Insert}(UC[k], \{p, q\}, s);\)
11. return \(s;\)

Find\(\) (in \(\text{tab}, \text{key}, \text{out} v, \text{sat}: \text{bool}\)) : \(\text{bool}\)

If \((\text{key}, x, y)\) is in hash table \(\text{tab}\), set \(v\) to \(x\) and \(\text{sat}\) to \(y\) and return \(\text{true}\). Else, return \(\text{false}\).

Insert\(\) (inout \(\text{tab}, \text{in} \text{key}, \text{v}, \text{sat}: \text{bool}\))

If \(\text{key}\) is not \((0, 0)\) insert \((\text{key}, \text{v}, \text{sat})\) in hash table \(\text{tab}\), if it does not contain an entry \((\text{key}, \cdot, \text{true})\).

Locals\(\) (in \(\text{e: event, k: lvl, p: idx}\)) : set of \(\text{lcl}\)

Return all of the local states in \(p\) locally enabling \(e\). If there are no states in \(p\) locally enabling \(e\) then return \(\emptyset\).

Pick\(\) (inout \(\mathcal{L}, \text{set of lcl}) : \text{lcl}\)

Remove and return an element from \(\mathcal{L}\).

NewNode\(\) (in \(k: \text{lcl}, \text{inout p: idx}\)) : \(\text{idx}\)

Create \(p\), a node at level \(k\) with arcs set to 0. Return \(p\).

Check\(\) (in \(k: \text{lcl}, \text{inout p: idx}\))

If \(p\), a node at level \(k\) not in \(UT[k]\), duplicates \(q\), in \(UT[k]\), delete \(p\) and set \(p\) to \(q\). Else, insert \(p\) in \(UT[k]\). If \(p[0] = \cdots = p[n^k-1] = 0\) or \(1\), delete \(p\) and set \(p\) to \(0\) or \(1\).

Figure 3.4: Pseudo-code for the Saturation supporting functions.
The algorithm also uses supporting functions for creating nodes (NewNode), performing a union on two nodes (Union), and storing saturated nodes by checking them into a hash table (Check). The pseudo-code for the supporting functions is shown in figure 3.4. The function Generate starts the process of constructing the state-space by invoking calls to Saturate. Generate begins by constructing the initial MDD from a set of initial states, by invoking calls to Saturate for the initial nodes, in a bottom-up fashion, from $k=1$ until the root node at level $K$ is saturated. In this sense, Saturation is a bottom-up strategy, since all of the descendants of a node that the saturation process is invoked for are already saturated.

3.3.1 Saturation example

Table 3.2: Saturation example: next-state function.

<table>
<thead>
<tr>
<th>level</th>
<th>$e_1$</th>
<th>$e_2$</th>
<th>$e_3$</th>
<th>$e_4$</th>
<th>$e_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>1→0</td>
<td>*</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>0→2, 1→2</td>
<td>*</td>
<td>0→2</td>
<td>0→1</td>
</tr>
<tr>
<td>1</td>
<td>0→1, 1→2</td>
<td>*</td>
<td>1→0</td>
<td>2→0</td>
<td></td>
</tr>
</tbody>
</table>

To illustrate the Saturation algorithm, we present an example of state-space construction using the Saturation algorithm, which is based on the example given in [29]. Table 3.2 shows the next-state function for the example, which has five events $e_1, \cdots e_5$, and three subsystems, $K = 3$. The symbol "*" is used to denote a level that is not affected by an event. Figure 3.5 shows the construction of the MDD, where snapshots highlight the status of the MDD, from the initial state to the final state-space. The levels of an MDD node are shown to the left of the figure, while a unique index referencing a node is defined by a number to the right of each node.
Figure 3.5: Saturation example: MDD construction.
We use the terminology \textit{index/level} throughout the example to refer to a node. A node highlighted in bold indicates a saturated node, a dashed line indicates a newly created arc, and a grey state indicates an enabling state. A description of each of the snapshots is used to show how the algorithm constructs the state-space.

- **Snapshot (a):** \textit{Generate} creates the initial MDD, setting the initial state to (0,0,0), and invokes \textit{Saturate} on node 2.1. Nodes 2.2 and 2.3 are actually created in snapshots (c) and (i) respectively, but for ease of understanding, we show them in this snapshot.

- **Snapshot (b):** \textit{Saturate} starts firing events on node 2.1. The only event $e$ for which $\text{Top}(e) = 1$ is $e_1$. State 0 enables $e_1$ causing a firing from 0 to 1, followed by an in-place update from state 1 to the terminal node 1.0. Since state 1 is now enabled, $e_1$ is fired again, resulting in an in-place update on state 2.

- **Snapshot (c):** All of the events for which $\text{Top}(e) = 1$ have been exhaustively fired on node 2.1, resulting in a fixed point. The node is therefore marked as saturated. \textit{Check} is used to insert the node into the unique table, however it is discovered that the node is the shape of the reserved node 1, and node 2.1 is subsequently replaced by node 1.1. \textit{Generate} continues the process of bottom-up saturation by invoking \textit{Saturate} for node 2.2.

- **Snapshot (d):** \textit{Saturate} starts firing events on node 2.3, beginning with event $e_2$ for which $\text{Top}(e) = 2$. State 0 enables $e_2$ causing a firing from 0 to 2, followed by an in-place update from state 2 to node 1.1.

- **Snapshot (e):** \textit{Saturate} continues firing events on node 2.3, using event $e_4$, for which $\text{Top}(e) = 2$. State 0 enables $e_4$, causing \textit{RecFire} to
be called for $e_4$ and node 1.1, which creates a new node 3.1. State 1 of node 1.1 enables $e_4$ at level 1, resulting in state 0 of node 3.1 being pointed to the terminal node 1.0.

- **Snapshot (f-g):** *RecFire* invokes *Saturate* on node 3.1. The node is saturated and checked into the unique table in exactly the same manner as snapshots (a-b).

- **Snapshot (h):** Following the completion of *RecFire*, *Saturate* completes the firing of $e_4$ on node 2.3 by making an in-place update. The *Union* is invoked for the descendant nodes at states 0 and 2. Since the nodes are the same, the arc from state 2 continues to point to node 1.1.

- **Snapshot (i):** All of the events for which $Top(e) = 2$ have been exhaustively fired on node 2.2, and the node is marked as saturated. *Check* is used to insert node into the unique table. *Generate* continues the process of bottom-up saturation by invoking *Saturate* for node 2.3.

- **Snapshot (j):** *Saturate* starts firing events on node 2.3. The events for which $Top(e) = 3$ are $e_3$ and $e_5$. *Saturate* fires $e_5$, which is enabled for state 0 causing *RecFire* to be called for $e_5$ and node 2.2. Since $e_5$ is enabled for state 0 of node 2.2, *RecFire* is recursively called for $e_5$ and node 1.1, creating node 4.1. State 2 of node 1.1 enables $e_5$ at level 1, resulting in state 0 of node 4.1 being pointed to the terminal node 1.0.

- **Snapshot (k):** *RecFire* invokes *Saturate* on node 4.1. The node is saturated in exactly the same manner as in snapshot (a).

- **Snapshot (l):** Node 4.1 is marked as saturated and checked into the unique table in exactly the same manner as snapshot (b). *RecFire*
continues to fire $e_5$, at level 2, and creates node 3.2. State 0 of node 2.2 enables $e_5$, resulting in state 1 of node 3.2 being pointed to node 1.1.

- **Snapshot (m):** $RecFire$ invokes $Saturate$ on node 3.2 Event $e_2$ for which $\text{Top}(e) = 2$ is enabled by state 1, resulting in an in-place update on the node.

- **Snapshot (n):** All of the events for which $\text{Top}(e) = 2$ have been exhaustively fired on node 3.2, and the node is marked as saturated. $Saturate$ makes an in-place update on state 1 of node 2.3 to point to the new descendant node 3.2.

- **Snapshot (o):** $Saturate$ fires the newly enabled event $e_3$ on node 2.3, by calling $Union$ for nodes 2.2 and 3.2, which results in node 1.2.

- **Snapshot (p):** $Saturate$ completes the firing of $e_3$, making an in-place update on node 2.3, pointing state 0 to the new descendant node 1.2.

- **Snapshot (q):** The redundant node 2.2 is removed. All of the events for which $\text{Top}(e) = 3$ have been exhaustively fired on node 3.3, and thus the state-space construction is complete since the root node is saturated. The final state-space is shown.

### 3.4 SMART

The SMART (Stochastic Model checking Analyser for Reliability and Timing) verification tool [26], is an environment that integrates both multiple logic and stochastic modelling formalisms. It allows parametric models to be defined using Petri nets as a formalism. Explicit and symbolic state-space generation algorithms are implemented, including Saturation, as well as CTL model checking algorithms.
The SMART software is object oriented and implemented in C++. Saturation is defined as an object, which encapsulates the MDD data structures including those for storing MDD nodes, the unique table implemented as a hash table, and the operation caches. The functions of the saturation class are the state-space generation algorithm, which operate on the encapsulated data structures.

- **MDD nodes** contain an array of downward arcs. MDD nodes are stored in an extensible array, which is allocated an initial segment of memory according to a specified pagesize, and allocated more memory according to pagesize when it becomes full.

- The hash table is an extensible array of hash pointers. Each hash pointer contains a pointer to an MDD node as well as a pointer to another hash pointer in order to deal with collisions. The hash table is initialised with a specified pagesize. When an MDD node is checked into the hash table, a hash pointer is created with a pointer to the MDD node and inserted into the hash table. The position in the hash table is calculated by executing a hash function on the MDD node. The hash function thus, returns an index within the size bounds of the hash table. If the hash pointer is inserted into a position in the hash table where there is an existing hash pointer, then it is set to point to the existing hash pointer. In this sense the hash pointers are stored as a linked list within the hash table array. If the hash table grows too large, then it can be resized according to pagesize.

- The operation caches are also stored as a fixed size array of cache pointers, which is initialised with a specified pagesize. Each cache pointer contains a pointer to an event, and two MDD nodes, representing the cached item and a pointer to a cache pointer to deal with
collisions. When an item is cached, a cache pointer is created and set to point to the cached item, and inserted into the operation cache at a position determined by a executing a hash function on the cached item. The hash function thus, returns an index within the size bounds of the operation cache. If the cache pointer is inserted into a position in the operation cache where there is an existing cache pointer, then it is set to point to the existing cache pointer. Thus, the cache pointers are stored as a linked list within the operation cache.

Other SMART objects include those to parse the model input, and generate the next-state function. Thus, when the saturation object is instantiated, it is passed an object representing the next-state function and the initial MDD. When the generate function is called, it generates the state-space and returns the resultant MDD.

### 3.5 Issues with parallel C++

```cpp
class NoParallelism{
    public:
    void update_i();
}

void NoParallelism::update_i()
{
    for(int i=0; i<1000; i++);
}

NoParallelism* npObj;
Thread1()
{
    npObj = new NoParallelism();
}
Thread2()
{
    npObj->update_i();
}
```

Figure 3.6: Code for the C++ object parallelisation problem.

During our initial experiments on parallelising the Saturation algorithm, we encountered a serialisation problem using C++ objects when creating parallel tasks. Figure 3.6 shows a segment of code that reproduces our
problem. The `NoParallelism` class defines a public function `updatei()` which increments an integer, \(i\) within a `for` loop. We define 2 threads, `Thread1()` instantiates the shared `NoParallelism` object, `npObj`, `Thread2()` makes a call to `updatei()`. One would expect the `updatei()` function to be executed by `Thread2()`. However, we found that the function was serialised and executed by the thread that created `npObj`, in this case `Thread1()`. We demonstrated this by using a loop with a high value of \(i\), and allowing 4 threads to call `updatei()` in parallel. We saw no difference in run-time from calling `updatei()` in parallel compared to calling it sequentially 4 times, suggesting that the functions were serialised.

When we first began parallelising Saturation, we did so by instantiating an instance of the saturation class, and allowing different threads to call its functions. Retrospectively, due to the problem we described, it is clear where the lack of parallelism arose from. We therefore re-wrote the Saturation algorithm in C, to allow functions to be executed in parallel, by porting the encapsulated data structures from the C++ saturation object, and removing the object orientation to define C functions. We were also required to serialise the next-state function and initial MDD passed to the saturation object within SMART, so that the C algorithm could retrieve these, in order to generate the state-space.

Table 3.3: C Saturation run-time improvement against C++ Saturation.

<table>
<thead>
<tr>
<th>Model</th>
<th>Run-time increase</th>
<th>Model</th>
<th>Run-time increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slotted Ring</td>
<td>33%</td>
<td>Round Robin</td>
<td>18%</td>
</tr>
<tr>
<td>FMS</td>
<td>20%</td>
<td>Queens</td>
<td>29%</td>
</tr>
<tr>
<td>Leader</td>
<td>25%</td>
<td>RSM</td>
<td>28%</td>
</tr>
<tr>
<td>Kanban</td>
<td>22%</td>
<td>Aloha</td>
<td>25%</td>
</tr>
<tr>
<td>BQ</td>
<td>28%</td>
<td>Philosophers</td>
<td>24%</td>
</tr>
</tbody>
</table>
In moving to a C based algorithm, we also affected the run-time of the Saturation algorithm. For all parameters of our benchmark of models, shown in section 4.4, the average increase in run-time of our C Saturation algorithm against the C++ Saturation algorithm can be seen in table 3.3. Thus, we can clearly see our C algorithm typically improves on the C++ algorithm by approximately 20%-30%.

3.6 Chapter summary

In this chapter we introduced structured systems, which can be used to compute state-spaces more efficiently, when considering structure-rich classes of systems such as communication protocols and distributed systems. We described the Saturation algorithm, which exploits the structure-rich class of event-based asynchronous systems for its efficient search strategy. We described how Saturation is implemented in the SMART environment, and showed the problem encountered when parallelising Saturation using C++, as well as describing why and how we ported the C++ algorithm to C, and the improvement this made to the run-time of the algorithm.
Chapter 4

Measurement and evaluation

In practice, parallel state-space construction algorithms often show speed-ups on very few models. This is due to a combination of factors: (i) state-space exploration algorithms incur high parallel overheads due to their irregular nature in terms of unpredictable sizes of work [77, 87, 100] and random access to data structures such as hash tables [101, 102], and (ii) they are dependent upon the model for parallelisability. Taking these issues into account when parallelising such algorithms is important; however, information related to overheads and model characteristics is usually not, or only incompletely, collected when an algorithm is evaluated. This makes it difficult to assess how well parallelised an algorithm is.

In this chapter we describe our approach to measurement and evaluation. We introduce background to the problem, including parallel overheads, and implementation concerns, and then suggest an evaluation approach based upon direct measurement of the parallel overheads at run-time. We then use this approach to select and justify a benchmark used for our run-time and memory experiments on the parallel algorithms presented in this thesis.
CHAPTER 4. MEASUREMENT AND EVALUATION

4.1 Introduction

The ability to evaluate a parallel algorithm accurately is useful for determining where improvements can be made, and for considering whether the technique underlying the algorithm can be used in other algorithms. An example of a technique that works for one algorithm and not for another is the static partitioning technique of [75, 94]: it resulted in linear speed-ups when originally applied to Murϕ [94], but in slowdowns when applied to SPIN [75]. This was due to the SPIN implementation incurring significant communication overheads that were almost negligible in Murϕ. Thus, techniques can work under certain circumstances but not under others. The ability to determine the effect of parallelisation overheads when evaluating the performance of a parallel algorithm is crucial to understanding the applicability of the underlying technique.

It is well known in the parallel community that overheads of a parallel algorithm need to be studied [73]; indeed, theoretical analysis of their impact on parallel state-space generation algorithms have been carried out previously (see, e.g., [72]). A theoretical analysis of state-space exploration algorithms on shared-memory architectures is, however, a difficult process due to the unpredictability of scheduling and synchronisation. These overheads become even more unpredictable in symbolic state-space exploration due to the increased irregularity of the employed data structure. Thus, theoretical analysis cannot always be performed satisfactorily, and direct measurement of overheads at run-time is necessary. Direct measurement is a challenging task in itself, as is reported, e.g., by Inggs in [66]. One difficulty is that any accurate measuring technique must take into account its own cost to the algorithm during the measurement process.

Previous approaches to parallelising state-space exploration algorithms have shown that the severity of parallel overheads is highly model dependent.
When using a benchmark of examples to evaluate Grumberg et al.’s parallel algorithm in [58], reported speed-ups varied greatly from little over one to an order of magnitude. Thus, a parallel state-space exploration algorithm can seem well or badly parallelised depending on the characteristics of the model. Choosing suitable models and determining their effect on parallelism is therefore key to evaluating an algorithm’s performance. Understanding the model’s influence on the overheads is key when selecting models to illustrate a parallel state-space exploration algorithm’s performance.

The rest of this chapter shares our experience of addressing these challenges while parallelising our Saturation symbolic state-space generation algorithm on a multi-processor, multi-core PC. We contribute the knowledge from our investigation into irregularity and subsequent overheads impacting on our parallel algorithm. Based on this investigation we suggest an approach to evaluating parallel state-space exploration algorithms, which compliments the use of benchmarking to assess performance and the quality of parallelisation.

We first deal with the aspects to parallelising a state-space exploration algorithm by looking at irregularity, the overheads arising from irregularity, how the algorithm can be implemented, and techniques to address the overheads. We then introduce our approach to evaluating parallel state-space exploration algorithms, showing how overheads can be measured, how models can be constructed to assess the overheads, and how the results from real models can be put into context. From this we select and document the benchmark that we use in the rest of this thesis.
4.2 Background on parallelising state-space exploration algorithms

Typically, the starting point for a parallel algorithm is to take a sequential algorithm and parallelise it, using some form of decomposition of work for distribution across processors. There are two decomposition approaches to parallelisation: functional decomposition parallelised the functions of an algorithm, and data decomposition parallelises the data structures of an algorithm. We began our work on Parallel Saturation by attempting to decompose the functions of the algorithm and execute them in parallel. However, our first attempt resulted in a significant impact on the time-efficiency of the resulting parallel algorithm. We found that understanding where the inefficiencies arose from was a painstakingly difficult task since there are a number of causes of inefficiency, and the nondeterminism of a parallel algorithm make them extremely hard to determine.

![Figure 4.1: Sequential and parallel reachability algorithm.](image-url)
4.2.1 From sequential to parallel

A simple work-set algorithm for exploring a system model’s reachable state-space, with respect to a given start state \textit{startstate} and transition relation (next-state function) \textit{Successor}, is shown on the left of figure 4.1. It is a generic work-set algorithm and does not specify a particular search strategy (such as breadth-first or depth-first). The \textit{Reachability} function begins with a start state and calls \textit{Enumerate} to enumerate its successor states. \textit{Enumerate} generates a set of successor states for any state passed to it, and for each successor state, if the state has not already been explored, it adds the successor state to the store of states (often implemented as a hash table) and pushes it onto a stack. \textit{Reachability} pops a new state off the stack and repeatedly calls \textit{Enumerate} on that state until there are no more states on the stack, at which point the entire state-space has been explored.

To a novice who wishes to parallelise this algorithm, the obvious starting point is to parallelise the \textit{Enumerate} function, allowing states to be enumerated in parallel. This seems like an easy and effective parallelisation, which leaves it to the operating system to schedule the generated threads on available processors. We show the corresponding parallel algorithm on the right of figure 4.1. \textit{ParallelReachability} performs the same task as the sequential \textit{Reachability} function; however, it also keeps track of the number of threads using the counter \textit{threads} in order to detect termination. Termination occurs when there are no threads enumerating states and no states to explore on the stack. The function must also ensure that, when it pops a state from the stack, access to the stack is atomic; this is to prevent \textit{data races} from other threads inserting states into the stack at the same time. To achieve this, \textit{ParallelReachability} locks the stack when accessing it and unlocks it when the state has been popped. When there is a state to explore, \textit{ParallelEnumerate} function,
indicated by the keyword parallel, and increments the threads counter to reflect that there is a new thread running. The ParallelEnumerate function operates in the same way as the Enumerate function, only when accessing the store of states and the stack, it locks and unlocks them for atomicity. ParallelEnumerate decrements the threads counter when it finishes, which allows the ParallelReachability function to monitor the number of threads.

![Transition Table and Parallel State Space Enumeration](image)

Figure 4.2: Parallel state-space enumeration and processor allocation of functions.

### 4.2.2 What are the overheads?

If we were to run the above parallel reachability algorithm to explore a state-space, we would encounter a significantly negative effect on run-time when compared to the sequential algorithm. By just looking at the run-times and the approach to parallelism, it is not obvious why this occurs, and analysing the underlying reasons for the negative effect can be a daunting
task for someone who is new to parallel algorithms. We can explain where the negative impact comes from using figure 4.2, which shows an example of how a state-space could be constructed in parallel when employing our algorithm on a four processor shared-memory architecture. Obviously, the details of the construction depend on how exactly threads are scheduled by the operating system.

Figure 4.2(a) shows the parallel state-space construction for the model defined by the transition table and start state 0; the displayed tree should be read as a search tree through the model’s state-space. While this example deals with explicit state-space generation, the ‘states’ could equally well be read as units of work conducted within a symbolic approach. The work of the \textit{ParallelEnumerate} function is visualised using a broken box, which includes the enumeration of duplicate states. figure 4.2(b) shows how the resulting \textit{ParallelEnumerate} functions could be scheduled on the individual processors. This highlights the dependencies between the functions, i.e., which function has to be executed before another function can be scheduled, and the frequency of locks on the data structures. Processor 1 is utilised by the \textit{ParallelReachability} function which is constantly checking the stack and scheduling new \textit{ParallelEnumerate} functions when a new state has been discovered.

From the processor utilisation of figure 4.2(b) we can infer a number of things: the computations are irregular in size, the computations can be small, parallel functions depend on others before they can be executed, locking occurs frequently, and the processors are not utilised fully. The irregularity of the computations defines state-space exploration as a particular type of parallel problem called \textit{irregular problem} in the parallel algorithms community, which are known to be difficult to parallelise \cite{54, 77, 87, 100}. Secondly, as is highlighted by the dependencies between the functions, state-
space exploration is a producer/consumer problem; in particular, work must take place before other work that is dependent upon it is even created.

As a result of these characteristics of the algorithm, a number of parallel overheads arise, which impact on the run-time of the algorithm and cannot be determined from run-time measurements alone. Firstly, irregularity causes load imbalance, where work is not distributed evenly amongst processors, which means that they are not fully utilised for work. This problem is further compounded by the producer/consumer nature of the algorithm which imposes the restriction that work cannot be scheduled on processors until other work is completed. Secondly, small computations result in scheduling overhead, where the cost for scheduling work, e.g., the cost of creating a thread (approx. 90,000 ns on a modern PC running Linux) can be higher than the work performed by a small computation (approx. 1,200 ns) [46]. Thirdly, frequent locking results in high synchronisation overhead, i.e., the time taken to lock and unlock a data structure, which would translate to communication overhead on a PC cluster where processes must frequently synchronise with each other.

### 4.2.3 Implementing a parallel state-space exploration algorithm

One of the great difficulties of parallelising state-space exploration algorithms is understanding how to efficiently implement a devised parallel algorithm. The types of languages and libraries chosen for parallelisation can have a significant effect on the performance of the parallel algorithm, since their characteristics can influence overheads. The importance of these choices can be shown when inspecting the work of Inggs [66]: the use of Java led to high synchronisation overheads for memory allocation and garbage collection, and eventually Inggs used C to implement her parallel algorithm.
Language selection can therefore be costly in time and effort if it hinders parallelisation.

A particular consideration for languages is the availability of parallel tool support, for algorithm profiling, for scheduling work, and for detecting parallelisation bugs such as data races. The best tool support is available for C and C++, which would suggest that these two languages are the primary candidates for selection. However, our own experience using C++ highlighted a problem with C++ objects, where instantiated objects on shared-memory architectures will execute their functions local to the thread that created them. This is especially relevant when shared data structures are encapsulated in objects, since access to them is then essentially sequentialised. For this reason we suggest C as the language of choice when parallelising for shared-memory architectures.

Another question that arises when parallelisation decisions are made is whether to use native thread libraries, or whether to use a library with a higher level of abstraction in order to make programming easier. For example, on shared-memory architectures, OpenMP [www.openmp.org] can be used to schedule work, isolating the programmer from having to understand thread programming, as well as allowing portability between operating systems. The drawback to this approach is that often fine control over the parallelism is lost. For example, we discovered that OpenMP is unable to effectively parallelise our mutually recursive functions in Saturation, since there is no control statement to handle recursion. We therefore found the Pthreads library [21] more suitable than OpenMP for parallelising our algorithm.

Selecting an appropriate architecture is also an important decision when considering how to parallelise a state-space exploration algorithm. Availability is a key issue when choosing hardware. Multi-processor, multi-core
PCs [76] are becoming widely available for performing experimental analysis on parallel algorithms. The drawbacks of this type of machine are that they only offer a relatively small number of processors/cores and that secondary cores are approximately 30-40% less efficient than processors when employing a multi-threading model. Larger shared-memory machines can offer more processors for performance evaluation but are less readily available. PC clusters are easily available but incur communication overheads across the network. In addition, operating system choice is often tied to the machine under usage. Different operating systems schedule their threads in different ways, which can affect scheduling overhead [60]. Another operating system decision is related to tool support, where parallel tools that can aid the development of the algorithm may only be able to run on particular operating systems.

4.2.4 Pthreads

A thread is an independent stream of instructions that can be scheduled to run independently by an operating system. Within a shared-memory environment, the operating system can select any processor on which to execute these instructions. Thus, threads are a method of implementing parallelism, by specifying streams of instructions, which the operating system can schedule concurrently on a shared-memory architecture.

Historically, the method for creating and executing threads was proprietary to the architecture on which the threads were executed. This defined a lack of portability for parallel algorithms between different types of shared-memory architecture. The result was the definition of a standard API for thread programming, the IEEE POSIX 1003.1c standard developed in 1995, better known as POSIX threads, or Pthreads [21]. Initially POSIX was developed for the UNIX operating system, but has since been extended for
coverage of the wealth of Linux operating systems.

The Pthreads API has three types of functions that can be used to develop threaded algorithms:

- **Thread specification and management:** This part of the API deals with specification and management of threads, for tasks such as creating, suspending, killing, joining and detaching threads, and specifying thread attributes such as scheduling policy.

- **Synchronisation:** This part of the API deals with synchronisation for memory safety. It provides a mutual exclusion lock known as a “mutex” which guarantees atomic access to a globally shared portion of data. The API allows creation and deletion, and locking and unlocking of mutexes.

- **Condition variables:** This part of the API allows signals to be sent between threads that use a shared mutex. For instance, functions can wait on a signal to obtain a mutex lock.

**Thread specification**

The thread specification part of the API provides a mechanism for the user to create and manage threads. The main function of this part of the API is the one which creates a thread. To create a thread using Pthreads, we can use the following function:

```c
pthread_create(thread, attr, startroutine, arg)
```

There are several arguments to the `pthread_create` function: `thread` is the unique identifier of the thread returned by the function; `attr` is the attributes of the thread such as scheduling policy; `startroutine` is the C function that will be executed by the thread; `arg` is an argument that can be passed to the function specified by `startroutine`. 
We illustrate the use of the `pthread_create` function with a running example of a thread pool. For the thread pool we wish to create a number of threads which synchronise on a shared queue, for which tasks can be added to and removed from. We use a one-to-one mapping of threads to processors. The threads then pick tasks from the queue to execute them in parallel. We illustrate this mapping of threads to processors and synchronisation on the shared queue in figure 4.3.

```c
main()
{
    int i;
    int thread_id[NUMBER_PROCESSORS];
    for (i=1; i<=NUMBER_PROCESSORS; i++) // for all processors create a thread
    {
        int ret = pthread_create(&thread_id[i], NULL, handle_tasks, (void*) &i);
    }
}
handle_tasks(void* data)
{
    int thread_number = *((int* data));
    printf("Thread number:%d", thread_number); // print the thread number
}
```

Figure 4.4: POSIX thread pool example (1).

We show the creation of the threads for our thread pool in figure 4.4. According to the number of processors we have, we create a number of threads using `pthread_create`, which calls the `handle_tasks()` function. We
will later use this function to handle the tasks on the shared queue. The `pthread_create` function, passes the thread number, \(i\) to the function. The `handle_tasks()` function then prints out the thread number. If the number of processors was 4, the output of this algorithm should be 1234.

We must remember that `main()` is a thread in itself, which may terminate before the threads of our thread pool get to print their output. If we wanted the main function to wait for the thread pool threads to complete before proceeding, we could use the `pthread_join(thread)` function, specifying the `thread_id` of the threads of our thread pool as an argument to the function. The main thread would then wait on the `pthread_join` function until the specified threads had completed.

**Thread synchronisation**

Within a shared-memory environment, all threads have access to the same global memory. The operating system does not provide its own mechanism for providing atomic access to user declared data. It is the responsibility of the programmer to ensure that access to globally shared data is protected by means of synchronisation. To facilitate synchronisation, Pthreads provides a mutual exclusion lock known as a “mutex” which guarantees atomic access to a shared portion of the data.

The API provides the following functions to declare, initialise and destroy mutex locks, and lock and unlock them:

- `pthread_mutex_t mutex`
- `pthread_mutex_init(mutex, attr)`
- `pthread_mutex_destroy(mutex)`
- `pthread_mutex_lock(mutex)`
- `pthread_mutex_unlock(mutex)`

The mutex can be initialised with `pthread_mutex_init` and assigned sev-
pthread_mutex_t queue_mutex;

main()
{
  int i;
  int thread_id[NUMBER_PROCESSORS];
  pthread_mutex_init(&queue_mutex, NULL);
  for (i=1; i<NUMBER_PROCESSORS; i++) // for all processors - 1 create a thread
  {
    int ret = pthread_create(&thread_id[i], NULL, handle_tasks, (void*) &i);
  }
  add_task(1);
  handle_tasks(0); // become part of the thread pool
  pthread_mutex_destroy(&queue_mutex);
}

add_task(int task)
{
  pthread_mutex_lock(&queue_mutex);
  Push(task); // add the task to the queue
  pthread_mutex_unlock(&queue_mutex);
}

handle_tasks(void* data)
{
  bool exit = false;
  pthread_mutex_lock(&queue_mutex);
  while(!exit)
  {
    int task = Pop(); // get the task from the queue
    pthread_mutex_unlock(&queue_mutex);
    if (task<0) // termination signal
      exit = true;
    else if (task>0) // task is available
      DoSomething(task); // do something with it
    if (!exit) pthread_mutex_lock(&queue_mutex);
  }
}

Figure 4.5: POSIX thread pool example (2).

eral attributes, such as priorities, using the attr object in the initialisation function. Once a mutex has been initialised, it can be locked using pthread_mutex_lock, and unlocked using pthread_mutex_unlock. The pthread_mutex_lock function will not return until it is able to acquire a lock. Pthreads also provides a function pthread_mutex_trylock(mutex), which acquires the lock if it is available, but the function will return if the lock is unavailable, instead of waiting to acquire it.

We illustrate the use of Pthread mutexes by continuing our running thread pool example in figure 4.5. The program declares a mutex lock, queue_mutex, which is used to provide atomic access to the shared task queue in the thread pool. The main() function then initialises queue_mutex, and creates the threads in the thread pool. Note we extend the main() function
from figure 4.4 to add the first task to the queue, by calling `add_task()`. We then allow the main thread to become part of the thread pool, by calling `handle_tasks()`, and only need to create \( \text{NUMBER\_PROCESSORS} - 1 \) threads for a one-to-one mapping of threads to processors. After the `handle_tasks()` function returns, the main function destroys the `queue_mutex`.

We introduce a new function `add_task(task)` to add a task to the queue. The function locks `queue_mutex` to gain atomic access for adding a task to the queue, before using the `Push(task)` function to add a task to the queue. `queue_mutex` is then unlocked to allow tasks to be removed from the queue.

Within the `handle_tasks()` function, we wish to repeatedly remove tasks from our shared queue and do something with them until a termination signal is received. Thus, we lock `queue_mutex` before using the `Pop()` function to remove a task from the queue. We then unlock `queue_mutex` once we have the task, so that `add_task(task)` can acquire `queue_mutex`, and tasks can be inserted into the queue. We define that, if the task number is less than 0, it indicates termination of the thread, and thus we set `exit` to true. We reserve 0 to indicate that the queue was empty, and if there was a task, we call a function to do something with it. `queue_mutex` is then locked again if the termination signal has not been received, and the loop repeated to continue removing tasks from the queue and performing them.

This method of handling tasks is not ideal, because when the queue is empty, the thread ties up the processor by polling to check for a task. In turn this would frequently lock `queue_mutex` causing a synchronisation overhead for threads trying to add tasks to the queue. Therefore, what we require is a mechanism to allow our `handle_tasks()` function to wait when there is an empty queue, until it receives a signal indicating that a task has been added to the queue.
Condition variables

Pthreads condition variables provide a way for threads to synchronise based on the value of data. They are used to pass signals between threads, in conjunction with mutex locks, so that when a signal is received via a condition variable, a mutex lock is acquired. As with mutex locks, condition variables have to be declared, initialised and destroyed. The API provides the following functions for declaration, initialisation and destruction of condition variables:

`pthread_cond_t`  
`pthread_cond_init(condition, attr)`  
`pthread_cond_destroy(condition)`

The argument to `pthread_cond_init (attr)` indicates whether the condition variable can be seen by threads in other processes.

For sending and receiving signals a thread can wait on a condition variable, and send a signal using a condition variable.

`pthread_cond_wait(condition, mutex)`  
`pthread_cond_signal(condition)`

When the `pthread_cond_wait` function is called, the thread will sleep until it receives a signal for `condition`, and when it wakes up, will acquire `mutex`. This means that the `pthread_cond_wait` function must wait for the signal, and wait to acquire the mutex lock before it returns. The function `pthread_cond_wait` sends a signal, which wakes up an arbitrarily chosen thread waiting on `condition`. If all threads need to be woken up, then another function, `pthread_cond_broadcast` can be used with the condition variable passed as an argument to the function.

We illustrate the use of Pthread mutexes by continuing our running thread pool example in figure 4.6. The program declares a condition vari-
```c
pthread_mutex_t queue_mutex;
pthread_cond_t wakeup;

main()
{
 int i;
 int thread_id[NUMBER_PROCESSORS];
 pthread_cond_init (&wakeup, NULL);
 pthread_mutex_init(&queue_mutex, NULL);
 for (i=1; i<=NUMBER_PROCESSORS; i++)
 { 
  int ret = pthread_create(&thread_id[i], NULL, handle_tasks, (void*) &i);
  }
 add_task(1);
 handle_tasks(0);
 pthread_cond_destroy(&wakeup);
 pthread_mutex_destroy(&queue_mutex);
}

add_task(int task)
{
 pthread_mutex_lock(&queue_mutex);
 Push(task); // add the task to the queue
 pthread_mutex_unlock(&queue_mutex);
 pthread_cond_signal(&wakeup); // signal that a task has been added
}

handle_tasks(void* data)
{
 bool exit = false;
 pthread_mutex_lock(&queue_mutex);
 while(!exit)
 { 
  int task = Pop(); // get a task from the queue
 if (task == 0) // queue is empty
  { 
   pthread_mutex_unlock(&queue_mutex); // unlock the queue
   pthread_cond_wait(&wakeup,&queue_mutex); // sleep until wakeup
  }
 else
  { 
   pthread_mutex_unlock(&queue_mutex);
   if (task<0) // termination
    exit = true
   else
    DoSomething(task) // do something with the task
   if (!exit) pthread_mutex_lock(&queue_mutex);
  }
  }
}
```

Figure 4.6: POSIX thread pool example (3).
able, `wakeup`, which is used to wait on the shared task queue in the thread pool. The `main()` function then initialises `wakeup`, and destroys it before the program exits. We change the `add_task(task)` function to send a signal on `wakeup`, to indicate that a task has been added to the queue, so that any thread waiting on the queue will wake up.

Within the `handle_tasks()` function, when an empty queue is encountered, `queue_mutex` is unlocked to allow `add_task(task)` to access the queue. Then, `pthread_cond_wait` is called to wait on the `wakeup` condition variable, for which a signal is received when a task is added to the queue. When the thread wakes up it automatically attempts to acquire `queue_mutex` before `handle_tasks()` can proceed. The next action for `handle_tasks()` after the thread has woken up is to remove the new task from the queue, using `Pop()`, and proceed through the requests handling loop. Since `queue_mutex` has been acquired before `Pop()` is called, access to the queue remains atomic.

### 4.2.5 Addressing parallel overheads

When considering how to parallelise a state-space exploration algorithm, techniques for addressing parallel overheads must be well chosen to minimise their impact. The most common technique for addressing load imbalance caused by irregularity is dynamic load balancing, specifically work-stealing [2, 11, 12] techniques. These have been used in parallel state-space exploration algorithms to facilitate orders of magnitude improvements in time-efficiency [58, 66]. Workstealing is based on the principle that, when one processor runs out of work to do, it steals work from another processor. For instance, if processors are given a number of states to enumerate, a processor completing its work can attempt to steal states from other processors. While this can be effective in spreading work to multiple processors efficiently, the technique can introduce its own overhead from extra code
and synchronisation. Thus, in order to improve the run-time of the parallel algorithm, sufficient parallel work must exist for the technique to spread fully across the available processors.

Scheduling overheads can typically be reduced directly using some form of thread-pool, or lightweight threads rather than native operating system threads. Indirectly, scheduling overheads can be reduced by attempting to increase the amount of work performed by each thread, so that computations are scheduled less frequently. The success of this technique is highly dependent upon there being enough work that can be independently grouped together.

Synchronisation overheads can be alleviated by facilitating task independence and minimising the amount of access required to the underlying data structures (such as the stack and the state store). These approaches are key to the order of magnitude speed-ups attained from parallel state-space exploration algorithms in [58, 66]. For example, it may be better to allow for occasional duplication of states, so that synchronisation to check for duplicate states becomes less frequent. Synchronisation can also be further improved by optimising the data structures according to the architecture of the employed machine [66]. This requires an extensive study of the machine’s architecture and careful consideration of the utilisation and placement of data structures.

4.3 Evaluating parallel state-space exploration algorithms

During our study of Parallel Saturation, we spent a great deal of time investigating how to evaluate our algorithm’s performance according to the impact of parallel overheads arising from irregularity. We found this to be
a challenging task, due to the range of parallel overheads and the lack of techniques for accurately measuring them. Most approaches to evaluation reported in the literature benchmark the run-time of an algorithm on a small number of models and include incomplete estimations of the parallel overheads. In this section we show how to strengthen evaluation by accurate and thorough direct measurement of parallel overheads and careful selection of models to illustrate the overall performance of a parallel state-space exploration algorithm.

4.3.1 Measuring parallel overheads

The quality of the measurement of parallel overheads, and how it reflects on the performance of a parallel state-space exploration algorithm, is a key issue that has yet to be addressed in the literature. The overheads of the algorithm are usually measured through some form of estimation, such as the distribution of states across processors to indicate load (im)balance. When we tried to estimate the amount of parallel work arising from our next-state function, we found that the influences on the parallelisation are much more complex than the factors we initially considered [46]. Thus, estimates of overheads may not be an accurate measurement of their true impact on a parallel algorithm, which brings their contribution towards an objective evaluation into question.

We now address the problem of measuring performance for each type of overhead in turn. Load balancing is difficult to measure. If we count the number of states enumerated on a processor and discover that this number is fairly even on different processors, we could argue that the load is well balanced amongst processors. However, what has not been taken into account here is the way in which work has been scheduled: due to the dependencies between states, the processors may not have enumerated these
states at the same time, and the processors could have been frequently idle. Similar problems also exist for measuring scheduling overheads. We could count the number of times work has been scheduled and multiply it by a measurement of the time taken to schedule work, e.g., the cost of the creation of a thread, in order to estimate the overhead. However, this would be an inaccurate measure since the operating system decides how work is scheduled, and the cost of re-scheduling work to another processor would be omitted from the measurement. Synchronisation overheads could be timed by instrumenting the code with a timer for each mutex lock, i.e., starting the timer before attaining the lock and stopping it once the lock has been attained. We found however, that this method is inaccurate, since the timer increases the amount of time the lock is opened for and introduces its own cost into the algorithm. Others have found similar problems with estimating synchronisation overheads [66].

The only solution to accurately measuring these overheads is via direct measurement, but this is a challenging task. How does one measure the activity on each of the processors? Even finding a timer whose granularity is fine enough to measure the time spent acquiring a mutex lock posed a problem for us. We tried using the gprof algorithm profiler [56, 57], but it was developed to profile sequential algorithms and could not show the individual activity of threads. What we required was a parallel profiling tool that can accurately analyse each of the parallel overheads at run-time, while taking into account the cost of its own instrumentation. To the best of our knowledge, the only such tool that currently exists on shared-memory architectures is Intel’s Thread Profiler [www.intel.com/software/products/threading/]. In order to evaluate Parallel Saturation, we applied the profiler to the parallel algorithm at runtime. We chose our experimental architecture according to the constraints
on the types of processors and operating systems the profiler supports, i.e., Intel processors, certain flavours of Linux, C and C++ and particular compilers. Figure 4.7 illustrates measurements that can be made using the profiler, showing the percentage of the algorithm’s run-time taken up by scheduling, synchronisation, serial execution, execution on less than the available processors, and fully parallel execution. This is an accurate and thorough breakdown of the algorithm’s parallel overheads.

### 4.3.2 The influences of a model on parallel overheads

While parallel overheads can be identified as a general influence on an algorithm, a subtlety of state-space exploration algorithms, and one which is not discussed in related literature, is the effect of the model on the severity of the overheads. We found this out the difficult way in Saturation, since some models showed good performance using the parallel algorithm, and others showed a lack of parallelisability [46]. At first glance these results can suggest that the parallelisation is inefficient, which is a highly frustrating point to consider when a lot of work has gone into the parallel algorithm. Our experience from investigating the effects of the underlying model can hopefully be used to alleviate future frustration relating to this point, by illustrating that the parallelisation efficiency is highly dependent upon the model as well as the techniques that have been used to address overheads.

Figure 4.8 shows the way in which the work units of our parallel reach-
ability algorithm in figure 4.1 are broken down during reachability analysis, when different model characteristics are considered. We highlight the work in the same way as in figure 4.2, where the units of work are defined functionally and the dependencies between them are decided by the way in which the functions enumerate states. We use pathological examples to illustrate particular overheads that may arise during the construction of the state-space for three stereotypical types of model. Figure 4.8(a) shows a model where the work cannot be spread across processors, since the work units are essentially sequentialised due to the dependencies between them. Figure 4.8(b) illustrates a model that imposes high scheduling overheads, due to the small size of the work units. Figure 4.8(c) is an ideal model that can potentially be well parallelised, since the work can be spread across processors in parallel and since the units of work are large enough to minimise scheduling overhead.

The illustration highlights that the characteristics of the model under consideration is a key influence on the performance of a parallel algorithm. In practice, the models that we used to evaluate our Parallel Saturation algorithm show a combination of these factors, with performance varying from
super-linear speed-ups to slowdowns of over 60%. Other parallelisation of state-space exploration algorithms also show widely varying performance of the algorithm according to the models' characteristics [58]. Thus, when applying techniques to deal with parallel overheads, it is important to consider that the technique may not be successful under a set of circumstances imposed by some model under consideration, such as insufficient parallel work.

4.3.3 Evaluating a parallel algorithm

Given the overhead measurement and model dependency issues that we have highlighted, a good quality evaluation of a parallel state-space exploration algorithm should include an accurate direct measurement of the overheads, and analyse the effect of the employed models on the parallel algorithm. Most importantly, the set of models used for evaluating a parallel state-space exploration algorithm should cover the space defined by the three stereotypes of figure 4.8. This allows a way in which each particular type of overhead can be thoroughly evaluated and (in)efficiencies in the algorithm can be pinpointed.

Figure 4.9 shows the profiles of our three stereotype models: (a) has low parallelism, (b) has high scheduling, and (c) is an ideal model that can

![Figure 4.9: Profiles of three stereotypical models.](image-url)
be parallelised well with minimal overheads. Using models matching profile (a) one can ascertain whether the load balancing function of the parallel algorithm under investigation can be improved, by attempting to increase the amount of fully parallel processor utilisation. Using models matching profile (b) one can attempt to improve the scheduling technique, by reducing the scheduling overhead. Using models matching profile (c) one can try to elaborate on various parallel overhead techniques, where any increase in fully parallel processor utilisation and decrease in scheduling and synchronisation overhead is desirable. These profiles highlight inefficiencies that can be used for the optimisation of parallel overhead techniques and allow for a quantitative comparison of the performance of different techniques. They also facilitate the understanding of how the effectiveness of the techniques can be challenged by the models under consideration.

Parallel state-space exploration algorithms are often benchmarked using a few models that illustrate the speed-up obtained by the algorithm. When considering the effect that the underlying model has on a parallel algorithm, a good evaluation must include models which cause the algorithm to incur overheads, and challenge its effectiveness at gaining run-time speed-ups. Without such challenging models, the overall effectiveness of the parallel algorithm is difficult to determine. Coupling challenging models with a profile of the overheads allows the algorithm’s performance to be truly put into context.

4.4 Our benchmark

The benchmark we selected for our algorithms contains ten models that have been previously used to parallelise Saturation [25, 29, 30]. The models are parameterised, i.e. a parameter can be set for the model, typically $N$ that alters the size of the state-space.
• The classic *Dining Philosophers* ([Philosophers](#)) [28] protocol is a solution for mutual exclusion problems. It consists of a number of philosophers seated around a table, with a fork between adjacent philosophers. When a philosopher wishes to eat they have to obtain both the left and right hand forks, which the philosopher has exclusive access to until they have finished eating. \(N\) indicates the number of philosophers, and \(K = N/2\).

• The flexible manufacturing system ([FMS](#)) [82] consists of three types of machines that process three types of parts. Two machines produce different parts that can be assembled by the third machine into a single part. The third machine can also process another type of part while it is idle. The parameter \(N\) indicates the initial number of each type of parts, and \(K = 19\).

• The *slotted ring network protocol* ([Slot](#)) [84] is a well known model of a local area network. The unidirectional ring contains a number of stations where several slots are available to transmit messages. A station sends a message by writing data into a free slot. For our model, \(N\) is the number of processors in the ring, and \(K=N\).

• The *Kanban manufacturing system* ([Kanban](#)) [97] contains sixteen stations that process parts. If there is a ticket available, a part can enter a station and be processed. If the part is processed correctly, it can move on to the next station. If it is processed incorrectly, it is sent back to the station to be fixed. The parameter \(N\) sets the initial number of parts within a station. \(K\) is the number of stations.

• The *randomised leader election protocol* ([Leader](#)) [69] solves the problem of designating a processor as leader within a unidirectional ring. The processors of the ring are required to send messages around the
ring for the determination to occur. The parameter $N$ of our model defines the number of processors in the ring, and $K = 11N$.

- The *round robin mutex exclusion protocol* (Robin) [55] is used to control access for a ring of $N$ processors that access a shared resource. Access is granted by moving a token around the ring, i.e., the processor that owns the token is granted access to the resource before passing the token to its neighbour. Each processor is mapped to a level of the MDD.

- The classic *queens problem* (Queens) models a game that finds a way to position $N$ queens on an $N \times N$ chessboard without the queens attacking each other. A solution to the problem requires that no two queens share the same row, column, or diagonal on the chessboard.

- The *runway safety monitor* (RSM) [92] was developed by NASA and Lockheed Martin, as a protocol to detect runway safety incidents. The RSM protocol defines *targets* $T$ on the ground such as ground vehicles on the runway, which have speeds $S$, and represents the takeoff and landing zone for aircraft as a 3D grid $X \times Y \times Z$, where $X$ is the width of the runway, $Y$ is the length and $Z$ is its vertical. For our RSM model we fix $T=1$ and $S=2$. The parameters of the model are $XYZ$.

- The *Aloha network protocol* (Aloha) [1] defines a simple mechanism for transmitting data across a network. The protocol is defined as follows: if there is any data to be sent, then send the data; if the message encoding the data collides with another transmission, then try to resend the data. For our model of the Aloha network protocol [27] the parameter $N$ represents the number of nodes in the network and $K = N+3$.
| Model Size                  | Parameter ($N$) | State-space size ($|S|$) |
|-----------------------------|-----------------|--------------------------|
| **Slotted Ring Network Protocol** |                 |                          |
| small                       | 90              | $5.9 \times 10^{94}$     |
| medium                      | 120             | $5.1 \times 10^{126}$    |
| large                       | 150             | $4.5 \times 10^{158}$    |
| **Round Robin Network Protocol** |                 |                          |
| small                       | 180             | $6.2 \times 10^{50}$     |
| medium                      | 210             | $7.8 \times 10^{65}$     |
| large                       | 240             | $9.5 \times 10^{74}$     |
| **Kanban Manufacturing System** |                 |                          |
| small                       | 25              | $7.6 \times 10^{12}$     |
| medium                      | 30              | $5.0 \times 10^{13}$     |
| large                       | 35              | $2.5 \times 10^{14}$     |
| **Flexible Manufacturing System** |                 |                          |
| small                       | 11              | $1.1 \times 10^{9}$      |
| medium                      | 13              | $5.8 \times 10^{9}$      |
| large                       | 14              | $1.3 \times 10^{10}$     |
| **Queen Problem**           |                 |                          |
| small                       | 11              | 166926                   |
| medium                      | 12              | 856189                   |
| large                       | 13              | 4674890                  |
| **Randomised Leader Election Protocol** |   |                          |
| small                       | 6               | $1.9 \times 10^{6}$      |
| medium                      | 7               | $2.4 \times 10^{7}$      |
| large                       | 8               | $3.0 \times 10^{8}$      |
| **Bounded Open Queuing Network** |               |                          |
| small                       | 30              | $2.4 \times 10^{8}$      |
| medium                      | 50              | $4.6 \times 10^{9}$      |
| large                       | 70              | $3.3 \times 10^{10}$     |
| **Dining Philosophers**     |                 |                          |
| small                       | 20              | $3.5 \times 10^{12}$     |
| medium                      | 40              | $1.2 \times 10^{25}$     |
| large                       | 80              | $1.4 \times 10^{50}$     |
| **Runway Safety Monitor**   |                 |                          |
| small                       | 332             | $1.3 \times 10^{10}$     |
| medium                      | 532             | $3.8 \times 10^{10}$     |
| large                       | 832             | $1.0 \times 10^{11}$     |
| **Aloha Network Protocol**  |                 |                          |
| small                       | 40              | $2.3 \times 10^{13}$     |
| medium                      | 70              | $4.3 \times 10^{22}$     |
| large                       | 100             | $6.5 \times 10^{31}$     |

Table 4.1: State-space sizes of small, medium and large models
CHAPTER 4. MEASUREMENT AND EVALUATION

Fully Parallel Execution
Under Utilised Execution
Serial Execution
Synchronisation
Scheduling

% of run-time

--- A) Low Parallelism --- B) High Scheduling --- C) Ideal ---

Figure 4.10: Profiles of the Parallel Saturation algorithm for large models.
Figure 4.11: Profiles of the Parallel Saturation algorithm for medium models.
• A bounded open queuing network (BQ) [48] determines a bound on the capacity of an open queuing network. A queuing network contains a number of interconnected queues, in which jobs flow from one to another. The network can either be closed or open. In an open network, jobs can arrive from an external source, and leave the network. For our model of the bounded open queuing network the parameter $N$ represents the bound, and $K = 8$.

We use three parameters for our model that defines them as small, medium and large. We show this correspondence between parameters and state-space sizes in table 4.1.

4.4.1 Benchmark justification

Our experiments on Parallel Saturation are put into context by the profiles in figures 4.10, and 4.11. These profiles were generated using the Intel thread profiler and our Parallel Saturation algorithm in section 5.3. We illustrate the profiles of the models in our selected benchmark. We include models that fit into the stereotypical categories shown in figure 4.9, as well as models that fall between these categories by exhibiting characteristics of a combination of their overheads. We show, for different sizes of parameters applied to the models in the benchmark, that the profiles remain the same shape, with small differences in the overheads, typically where parallelism slightly increases for larger models.

The profiles show the overheads specific to the models and how they challenge the parallelism of the algorithm. For example, the Robin profile roughly fits the low parallelism profile of the model in figure 4.9(a): cores are under-utilised which demonstrates a lack of lack of parallelism. The Bounded Queue (BQ) profile roughly fits the high scheduling model profile in figure 4.9(b): scheduling overhead is relatively high compared to the other
models. The FMS profile roughly fits the ideal model profile in figure 4.9(c): scheduling is low and the cores are fully utilised for a high percentage of the run-time. Some of the models show characteristics of different stereotypes, such as Leader, which has high scheduling and low parallelism, and Philosophers, which has high scheduling but also has relatively high parallelism.

These profiles are highly useful in drawing conclusions as to the performance of the algorithm. In fact without them, one would not be able to understand the circumstances under which Parallel Saturation is able to improve over sequential Saturation. This is why model selection should try and include models which fit the profile of each of our stereotypical models. For this thesis, the benchmark contains models that fit the stereotypical profiles, and thus the benchmark can be justified as offering complete coverage of the parallel overheads that will challenge the parallel algorithms. However, where stereotypical models are unavailable that fit these profiles, they can be artificially constructed, e.g., the authors of the parallel state-space exploration algorithm in [67] used artificial models that varied the amount of parallel work available.

\subsection*{4.4.2 Experimental setup}

The machine used for our experiments is a dual-processor, dual-core PC with 2GB of memory and Intel Xeon CPU 3.06GHz processors with 512KB cache sizes, running Redhat Linux AS 4, Redhat kernel 2.6.9-22.ELsmp, with glibc 2.3.4-2.13.

To obtain each result of an experiment we performed, we ran the corresponding algorithm three times. We present the average for each run. Run-time and memory variance for the results of running sequential and parallel algorithms on our benchmark was low, in the range of +/-5\%. For larger models the variance was lower, in the range of +/-4\%.
4.5 Summary

Techniques for parallelising state-space exploration algorithms need to be accurately evaluated, so as to provide objective insight into the quality of a parallelisation. Previous evaluation techniques employed in the literature relied on benchmarking, often using a small number of models along with an incomplete evaluation of the overheads. In this chapter we argued that in order for the quality of a parallelisation to be evaluated properly, direct measurement of each of the overheads is required during run-time, coupled with experiments using models that highlight the performance of the parallel algorithm under different conditions.

To show how such an evaluation can be carried out, we described the overheads that arise from state-space exploration, explained how to select models in order to highlight specific overheads, and showed how overheads can be directly measured at run-time. We then used this approach to justify the benchmark we use to evaluate the parallel algorithms in the rest of this thesis.

Although the direct measurement technique we used was devised for shared-memory architectures, we believe that our ideas can be extended to include PC clusters, by applying available profiling tools for measurement on these architectures, some of which can be found in [4].
Chapter 5

Parallelising Saturation

The Saturation algorithm [29] differs from other sequential symbolic state-space generators in that it exploits the locality of firing events in asynchronous system models. In this chapter, we explore whether event locality can be utilised to efficiently parallelise Saturation on shared-memory architectures. Conceptually, we propose to parallelise the firing of events within a decision diagram node, which is technically realised via a thread pool. We discuss the challenges involved in our parallel design, conduct studies on an implementation of the algorithm, and make several key optimisations the algorithm.

5.1 Parallel BFS

We begin our investigation into Parallel Saturation by parallelising a breadth first search (BFS) state-space construction algorithm. Doing so provides us with a stepping stone towards understanding how best to parallelise Saturation, since a BFS using disjunctive partitioning can be easily parallelised. We use the SMART [26] environment for our experiments, allowing us to use the disjunctively partitioned next-state function used in Saturation, as well as the MDD and operation cache structures, and the supporting functions.
**BFS** (in root:idx)

```plaintext
declare e: event;
declare f: array [1..E^K] of idx;
declare pCng: bool;

1. repeat
2. pCng ⇐ false;
3. foreach e ∈ E^K do
4. f[e] ⇐ BFSFire(e, K, root);
5. foreach e ∈ E^K do
6. if f[e] ≠ 0 then
7. u ⇐ Union(K, f[e], root);
8. if u ≠ root then
9. root ⇐ u; pCng ⇐ true;
10. until pCng = false;
11. return root;
```

**BFSFire** (in e: event, l: lvl, q: idx)

```plaintext
declare L: set of lcl;
declare f,u,s: idx;
declare i,j: lcl;

1. if k < K then
2. if Find(FC[l], {q, e}, s) then return s;
3. s ⇐ NewNode(l);
4. L ⇐ Locals(e, l, q);
5. while L ≠ ∅ do
6. i ⇐ Pick(L);
7. f ⇐ BFSFire(e, l−1, q[i]);
8. if f ≠ 0 then
9. foreach j ∈ N^l_e(i) do
10. u ⇐ Union(l−1, f, s[j]);
11. if u ≠ s[j] then
12. s[j] ⇐ u;
13. Check(l, s);
14. if k < K then
15. Insert(FC[l], {q, e}, s);
16. return s;
```

Figure 5.1: Pseudo-code for the BFS algorithm.
We show the pseudo-code for our BFS algorithm in figure 5.1. The BFS operates on the root node at level $K$ to bring it into a fixed point shape, by repeatedly firing events until they make no more updates the node. The firings are performed by calling $BFSFire$, to generate new MDDs representing the set of states resulting from the event firings. Once all of the firings have completed, the root node is updated by performing a union between the root node and all of the new nodes created as a result of the firings.

The $BFSFire$ function is similar to the $RecFire$ function in Saturation, except that new nodes are not brought into a fixed point shape by saturating them. Therefore, the function generates a set of states resulting from a firing, but does not continue to explore further states by performing more firings on the new state set. An operation cache for storing the results of fires continues to be employed for speed.

The algorithm can be easily parallelised using loop parallelisation, by performing each $BFSFire$ in parallel. At line 4 of $BFS$, we can create a new thread each time we call $BFSFire$, and synchronise on the completion of all of the threads before proceeding to line 5 of $BFS$. The only other change we need to make is to lock the data structures for atomic access, by employing a mutex lock for each cache and for the unique table, for each level of the MDD.

### 5.1.1 Chained BFS

At first glance, the Parallel BFS appears to be a superior algorithm than the BFS algorithm, since parallel $BFSFire$ functions should execute in less time than sequential calls to $BFSFire$. The parallelism is thus heavily reliant on the capability of the $BFSFire$ function to be easily scheduled in parallel. However, in a Chained BFS, firings are not performed in an independent
manner, but instead utilise the result of a previous firing.

The Chained BFS in figure 5.2 is therefore much more difficult to parallelise, since there is no clear loop to exploit for parallelism. Each time an event is fired using \textit{BFSFire}, the union of the resultant MDD and the root node is performed before firing the next event. The aim of this strategy is to create a larger root MDD before the next event is fired, so as to reduce the number of overall firings that need to be performed, by manipulating larger state sets at an earlier stage of the MDD construction. Therefore, the question arises as to how this simple sequential optimisation compares to Parallel BFS, particularly as the sequential optimisation eliminates the loop required for simple parallelisation.

\subsection*{5.1.2 Run-time and memory results}

We implemented three experimental algorithms, a Chained BFS, a BFS and Parallel BFS. Our algorithms were implemented using C and the POSIX Pthreads library [21]. We applied the algorithms to our benchmark (sec. 4.4) on our experimental machine (sec. 4.4.2).

```
BFSChained(in root:idx)
declare e:event;
declare f:idx;
declare pCng:boolean;
1. repeat
2. pCng := false;
3. foreach e ∈ E do
4.  f := BFSFire(e, K, root);
5.  if f ≠ 0 then
6.    u := Union(K, f, root);
7.    if u ≠ root then
8.      root := u; pCng := true;
9.  until pCng = false;
10. return root;
```
Table 5.1: Run-time and memory results for the BFS algorithms.

<table>
<thead>
<tr>
<th>N</th>
<th>BFSChained</th>
<th>BFS</th>
<th>BFS Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time(s)</td>
<td>Mem(b)</td>
<td>Time(s)</td>
</tr>
<tr>
<td>Slotted Ring Network Protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.61</td>
<td>3576400</td>
<td>7.16</td>
</tr>
<tr>
<td>20</td>
<td>18.52</td>
<td>78085280</td>
<td>1007.33</td>
</tr>
<tr>
<td>30</td>
<td>293.78</td>
<td>536934720</td>
<td>N/A</td>
</tr>
<tr>
<td>Round Robin Network Protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.23</td>
<td>1241644</td>
<td>13.55</td>
</tr>
<tr>
<td>30</td>
<td>0.64</td>
<td>3175464</td>
<td>81.01</td>
</tr>
<tr>
<td>40</td>
<td>1.44</td>
<td>6304884</td>
<td>293.73</td>
</tr>
<tr>
<td>Flex Manufacturing System</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.69</td>
<td>3084224</td>
<td>4.81</td>
</tr>
<tr>
<td>15</td>
<td>2.31</td>
<td>17128944</td>
<td>21.59</td>
</tr>
<tr>
<td>20</td>
<td>6.06</td>
<td>42982264</td>
<td>82.66</td>
</tr>
<tr>
<td>Kanban Manufacturing System</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3.64</td>
<td>58359236</td>
<td>14.14</td>
</tr>
<tr>
<td>15</td>
<td>8.61</td>
<td>135106208</td>
<td>32.28</td>
</tr>
<tr>
<td>20</td>
<td>17.35</td>
<td>288403644</td>
<td>427.57</td>
</tr>
<tr>
<td>Queen Problem</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.08</td>
<td>100512</td>
<td>0.09</td>
</tr>
<tr>
<td>9</td>
<td>0.46</td>
<td>632160</td>
<td>0.48</td>
</tr>
<tr>
<td>10</td>
<td>5.99</td>
<td>2741740</td>
<td>7.04</td>
</tr>
<tr>
<td>Randomised Leader Election Protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.55</td>
<td>3046768</td>
<td>2.48</td>
</tr>
<tr>
<td>5</td>
<td>4.61</td>
<td>18491440</td>
<td>30.88</td>
</tr>
<tr>
<td>6</td>
<td>52.55</td>
<td>91660264</td>
<td>543.16</td>
</tr>
<tr>
<td>Bounded Open Queuing Network</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.18</td>
<td>1248424</td>
<td>0.43</td>
</tr>
<tr>
<td>30</td>
<td>0.56</td>
<td>3801504</td>
<td>1.33</td>
</tr>
<tr>
<td>40</td>
<td>1.26</td>
<td>8572584</td>
<td>3.01</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1.26</td>
<td>8572584</td>
<td>3.00</td>
</tr>
<tr>
<td>10</td>
<td>1.65</td>
<td>9988660</td>
<td>4.92</td>
</tr>
<tr>
<td>15</td>
<td>2.45</td>
<td>3166424</td>
<td>8.10</td>
</tr>
<tr>
<td>Runway Safety Monitor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>2.34</td>
<td>15306720</td>
<td>6.97</td>
</tr>
<tr>
<td>322</td>
<td>2.39</td>
<td>17594712</td>
<td>7.17</td>
</tr>
<tr>
<td>422</td>
<td>5.14</td>
<td>34136960</td>
<td>14.35</td>
</tr>
<tr>
<td>Aloha Network Protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.34</td>
<td>2105960</td>
<td>0.39</td>
</tr>
<tr>
<td>30</td>
<td>1.25</td>
<td>7027840</td>
<td>2.05</td>
</tr>
<tr>
<td>40</td>
<td>3.18</td>
<td>16571720</td>
<td>7.10</td>
</tr>
</tbody>
</table>
We show the run-time and memory performance for the algorithms in table 5.1. The run-time in seconds is illustrated for all three algorithms, as is the memory for the Chained BFS in bytes, and the relative memory increase for the BFS and Parallel BFS algorithms against the Chained BFS. A model that exhausted the available memory is indicated by N/A.

Compared to the Chained BFS, the BFS algorithm is slower, and uses more memory for all models. The use of extra resources is significant in several models, such as slotted ring that is over 50 times slower in terms of run-time and uses over 9 times more memory. It would therefore be impossible for a Parallel BFS to catch up with the run-time of Chained BFS on our experimental machine for a model such as slotted ring, since it would require over 50 processors to facilitate a run-time improvement. The chaining optimisation is therefore highly effective in improving the run-time and memory of the BFS.

In contrast, the Parallel BFS only improves the run-time of the BFS for three of the algorithms. These improvements are comparatively small compared to the Chained BFS, with the largest improvement coming from the FMS model of approximately 20%. Even when the Parallel BFS is able to improve the run-time of the BFS, it is still at least 100% slower than the Chained BFS. The Parallel BFS also requires extra memory for the construction of extra MDDs in parallel, since more MDDs will be constructed at the same time than sequentially. This memory increase is in general between 20% and 100% more than the BFS.

One may wonder why the loop based BFS parallelism does not improve over the sequential version. While the idea may look feasible in theory, in practice the parallelisation incurs a number of overheads. The creation of a thread introduces a scheduling overhead, while the locking on the data structures introduces a synchronisation overhead. A load imbalance is also
imperative in this type of loop level parallelism, since $BFSFire$ will differ in the amount of time it takes according to the events it fires. Thus, the parallelism of each event loop is only as good as the time it takes to execute the longest timed $BFSFire$ function. If this time is less than the scheduling and synchronisation overhead, then the loop takes longer to complete in parallel than it does sequentially. The combination of these overheads means that the extra costs to the algorithm often outweigh any benefit from introducing parallelism. These costs are demonstrated by the run-time results for the Parallel BFS.

Our experiments with the BFS show that a simple sequential optimisation, such as chaining, can be much more effective than a simple parallelisation. Indeed, one would have to begin reducing overheads for the parallel algorithm and scale to a much larger number of processors, to compete with the Chained BFS. Even then, the algorithm requires a greater amount of memory to construct the state-space in parallel. The experiments also show the importance of using an already optimised algorithm as a starting point for parallelisation, as others in the parallel community have observed [73]. This is because without including the sequential optimisation the parallel algorithm may suffer a large enough run-time loss to negate the benefits of any parallelism. It is also important to consider that breaking a sequential optimisation when parallelising an algorithm can potentially outweigh any benefits of parallelism.

5.2 Parallel differences in the order of event firings

When considering parallelising Saturation, we potentially alter the order in which events are fired. In this section we show the differences event orderings can make, when considering parallel state-space construction, since any fair comparison between parallel and sequential algorithms must take
such differences into account.

The Saturation strategy exhaustively fires events on a node until a fixed point is reached. The order in which these events are fired is arbitrary, in the sense that all of the events are fired in an iterative manner, for each $e \in \mathcal{E}^k$, on any locally enabled state $i$ on a node $p$ where $p[i] \neq \emptyset$. Given this observation, more aggressive strategies for chaining events together can be employed. We can illustrate the difference in event orderings, by firing events only on local states that have been updated as a result of previous event firings. This approach offers two potential savings.

- Event firings will only be attempted for updated local states, rather than for any arbitrary local state. Thus, a firing is more likely to take place, instead of a cache hit.

- The state sets fired upon are potentially larger, since the local states considered are continually updated, rather than arbitrarily selected. For example if $e$ is fired upon $i$ as the result of $i$ being arbitrarily selected, then useless work will be performed when $i$ is updated at a later stage and $e$ is fired again, if $i$ could have been updated soon enough that $e$ only needed to be fired on $i$ once. Thus, useless work firing on smaller state sets can be avoided, if a local state becomes updated with a larger state set, before an event is fired on it.

We can achieve a firing order based on updated local states by rewriting the Saturate function to fire events on the initial locally enabled states of the node, and subsequently, only when a local state has been updated.

### 5.2.1 Approaches to optimising the event order

There are two ways in which we consider rewriting the Saturate function to achieve a more aggressively chained firing order. The first approach adds
updated local states to a set, following an in-place update. We can then
decide the order in which local states are removed from the set, and fired
upon, for example, by using either a stack or queue implementation. The
pseudo-code for a \textit{Saturate} function employing this approach is shown in
figure 5.3. Lines 1-3 add the initial locally enabled states of the node to the
set of local states $\mathcal{L}$, considered for firing upon. Line 5 picks a local state $i$
from $\mathcal{L}$, and lines 6-8 fire each event $e \in \mathcal{E}^k$ upon $i$. Line 14 adds an updated
local state $j$ to $\mathcal{L}$, after the in-place update has taken place. The process
repeats until $\mathcal{L}$ is empty, and there are no more local states to consider.

The set based approach picks updated local states from a set and fires
each event $e \in \mathcal{E}^k$ on the local state $i$ before picking the next local state $j$.
We can introduce another event order by firing on the updated local state
$j$ before the next event $e$ in the iteration is fired upon $i$. This approach
can be easily expressed as a recursive function, shown in the pseudo-code

\begin{verbatim}
Saturate(in k:level,p:idx)
Update p, a node at level k not in UT[k], in-place, to encode $\mathcal{N}^*_\leq_k(B(p))$.

1. foreach i \in \mathcal{S}^k do
2. if p[i] \neq 0 then
3. \quad \mathcal{L} \leftarrow \mathcal{L} \cup \{i\};
4. while \mathcal{L} \neq \emptyset do
5. \quad i \leftarrow \text{Pick}(\mathcal{L});
6. foreach e \in \mathcal{E}^k do
7. if $\mathcal{N}^k_e(i) \neq \emptyset$ then
8. \quad f \leftarrow \text{RecFire}(e,k-1,p[i]);
9. if f \neq 0 then
10. \quad foreach j \in \mathcal{N}^k_e(i) do
11. \quad \quad u \leftarrow \text{Union}(k-1, f, p[j]);
12. \quad \quad if u \neq p[j] then
13. \quad \quad \quad p[j] \leftarrow u;
14. \quad \quad \mathcal{L} \leftarrow \mathcal{L} \cup \{j\};
\end{verbatim}

Figure 5.3: Queue and stack based firing order.
CHAPTER 5. PARALLELISING SATURATION

104

Saturate\((in \ k:lel, p:idx)\)
Update \(p\), a node at level \(k\) not in \(UT[k]\), in-place, to encode \(N^*_\leq_k(B(p))\).

declare \(i:lel\);
1. foreach \(i \in S^k\) do
2. if \(p[i] \neq 0\) then
3. FireEvents\((k, p, i)\);

FireEvents\((in \ k:lel, p:idx, i:lel)\)
Fire \(e\) on \(p[i]\) when \(N^*_k(i) \neq 0\)

Figure 5.4: Recursive based firing order.

of figure 5.4. Lines 1-3 of the Saturate function calls FireEvents for each of the initial locally enabled states of the node. Lines 1-3 of FireEvents begins the iterative process of firing all events on a local state \(i\). Line 9 of FireEvents recursively calls itself on an updated state \(j\) after an in-place update, meaning that the event loop at Line 1 of FireEvents does not continue firing events on \(i\) until all events have been fired on \(j\).

5.2.2 Run-time and memory results

We implemented three algorithms using C++ in the SMART verification tool, one using a stack based firing order, one using a queue based ordering and one using the recursive approach. We applied the algorithms to 7 models of our benchmark (sec. 4.4) on our experimental machine (sec. 4.4.2). The run-time and memory results are shown in figures 5.5 and 5.6. We show the parameter of a model at the bottom of each graph. To the left of the
Figure 5.5: Run-time and memory results for the firing orders (1).
Figure 5.6: Run-time and memory results for the firing orders (2).
graph we show either the relative run-time or relative memory, where less than 1 indicates a decrease compared to the original Saturation firing order and greater than 1 shows a relative increase.

For 4 out of 7 models, the new strategies show a run-time improvement over the original firing order, while only a slight decrease occurs for the FMS model. The best performing strategy is the stack based approach, which shows up to a 50% increase in run-time improvement. The recursive approach demonstrates a fairly similar performance to the stack based approach, while the less aggressive queue approach is the slowest of the three. Memory performance is only improved for 2 models by the recursive and stack based approaches with a 50% improvement for the slotted ring model. The queue approach is unable to improve memory over the original firing order.

The results show that the ordering of events can have a significant effect on the run-time performance of Saturation, and the original firing order can be improved by a more aggressive chaining strategy. Thus, any comparisons we make between Parallel Saturation and sequential Saturation must take the effect of these orderings into account.

5.3 Parallel Saturation

As observed for the Parallel BFS, state-space construction algorithms are difficult to parallelise due to their characteristics. Tasks such as applying the next-state function are irregularly sized, are dependent upon each other and have to synchronise frequently. Irregular tasks cause load imbalance, and dependencies between tasks compound the problem. While there are a number of techniques to load balance irregular tasks, frequent synchronisation can only be avoided by making tasks as large and independent as possible. The goal of parallelising Saturation is to exploiting event locality
to create independent parallel tasks from event firings. However, when considering this we must also ensure that we preserve chaining so as to avoid the run-time and memory impact observed when parallelising the BFS.

While we can exploit event locality to create independent tasks, the firings of local events often cannot be parallelised due to their efficiency. On the shared-memory architectures we investigated, the cost of creating a parallel event to perform an in place update outweighs the cost of performing it. We approximated the cost of an in place update as 1200ns, compared to 90000ns for creating a thread and 8000ns for allocating a task to an existing thread. We measured the total run-time $t$ for performing a number $n$ of in place updates. The cost of an in place update was approximated as $t/n$. We approximated the scheduling cost by measuring the time it took for a number $n$ of sequential tasks to complete $t_{seq}$, and measuring the time it took to complete the same number of tasks sequentially scheduled to a separate thread or allocated as task to an existing thread $t_{par}$. The approximate scheduling cost is given by $(t_{par}-t_{seq})/n$.

An in place update can also occur when an event firing fully utilises previous work that has been cached. The cost of retrieving information from a cache is approximately 900ns. We must therefore group event firings together and only consider those events that do not result in a cache hit, or where the bottom level of the event is lower than the node being fired upon, otherwise the event firings will cost less than the scheduling.

To address irregularity we introduce a task queue to which parallel tasks can be added, and to load balance the tasks we utilise a thread pool where a thread is mapped 1-to-1 to a processor core. An available thread picks a task from the queue and performs the work associated with it. However, fitting the Saturation algorithm into this load balancing structure, while preserving chaining, is difficult due to Saturation’s mutually recursive nature. In
particular, in order to prevent threads from suspending we have to eliminate sequential waits on the result of a parallel event firing. We can achieve this by introducing upward arcs into the MDD structure, which directly replace recursive function calls waiting for work to complete. Instead, the function calls continue when parallel work is pending, leaving the upward arcs to represent future updates on a node.

We illustrate the introduction of upwards arcs into the MDD structure in figure 5.7, where the levels of the MDDs are shown to the left of the diagram, node indexes are shown to the top right of a node, and saturated nodes are highlighted by a thick box. We show some snapshots of the MDD construction at a point in time where node 3.1 has been created by a recursive firing call invoked by saturating node 2.2. Figure 5.7a shows an upward arc created as a result of this recursive firing, which is indicated by a dashed line pointing from the unsaturated node 3.1 to node 2.2. The upward arc is labelled with state 1, indicating that state 1 of node 2.2 will point to node 3.1 if node 3.1 becomes saturated. In figure 5.7b, node 3.1 becomes saturated, and thus the downward arc is set from state 1 of node 2.2 to point to node 3.1. This means that the recursive firing function that created node 3.1 does not have to wait for node 3.1 to become saturated to update node 2.2. Rather the recursive firing can return, leaving another thread to saturate
node 3.1, and update node 2.2 when node 3.1 becomes saturated, by using the upward arc to determine how node 2.2 should be updated.

Upward arcs thus allow a task that was created by firing an event on a node to continue the work on the node when it completes. However, the breaking of recursion using this approach implies that each node must keep track of the number of tasks operating on it in order to determine when it is saturated. We must also allow work requests to be cached before they have been carried out to avoid duplicate work in parallel.

5.3.1 Thread pool justification

The choice of thread pool as a starting point for our parallel algorithm is one that can be easily justified by the order of magnitude saving in scheduling cost compared to creating threads for each task, and in particular the control over which the tasks can be selected from the shared queue. A more complex workstealing system, for example, would require the use of multiple queues from which tasks can be stolen from. While the use of multiple private queues per thread may reduce synchronisation cost on the queues, within a symbolic environment, where memory is a concern, it is unknown as to what sort of memory impact this loss of control over the order of the construction of the state-space will have. This is justified by the effects observed from chaining within our BFS. Control over the task ordering is required if we wish to keep close to the same state-space construction order as the sequential algorithm. It is therefore better to approach the problem by using a mechanism over which finer control of the state-space construction can be exercised, and investigate the effects of task ordering, rather than developing a more complex scheduling approach that may impact on the chaining mechanism.
Figure 5.8: Pseudo-code for the Parallel Saturation algorithm.
ThreadLoop()

If there are no items in the task queue sleep until woken up. Otherwise remove the head item (k, p) from the task queue. If (k, p) is (0, 0) call Terminate() and terminate the thread, otherwise call Saturate(k, p).

Find(in: tab, key, out: v, sat:bool):bool,

If (key, x, y) is in hash table tab, set v to x and sat to y and return true. Else, return false.

Insert(in: tab, in: key, v, sat:bool)

If key is not (0, 0) insert (key, v, sat) in hash table tab, if it does not contain an entry (key, ... true).

Locals(in: event, k:thr, pidx):set of lcl

Return all of the local states in p locally enabling e. If there are no states in p locally enabling e then return Ø.

Pick(in: set of lcl):lcl

Remove and return an element from L.


Create nth threads. Build an MDD rooted at root, at level K, encoding the state space and return root, in UT[K].

Union(in: k:thr, pidx, qidx):dsz

Build an MDD rooted at s, a node at level k, in UT[k], encoding the Union of p q. Return s.

DWarc(s in: k:thr, pidx, qidx):bool

If p[i] ≠ 0 for any local state at level k return true otherwise return false.

SetUpArc(in: k:thr, pidx, qidx, ridx):k

Lock (p.ua). Add an arc (q,i) to the end of the list of upward arcs for p; AddOp(k+1, q); Unlock (p.ua);

SetUpArc(k:thr, pidx, qidx, ridx):bool

If the list of upward arcs is not empty retrieve and remove (q,i) from head of list and return true. Otherwise return false.

AddOp(in: k:thr, pidx)


RemoveOp(in: k:thr, pidx, out: op:bool)

Lock (p.ops). Decrement p.ops. If p.ops = 0 set op to false otherwise set op to true. Unlock(p.ops).

Find(in: tab, key, out: v, sat:bool):bool,

If (key, x, y) is in hash table tab, set v to x and sat to y and return true. Else, return false.

Insert(in: tab, in: key, v, sat:bool)

If key is not (0, 0) insert (key, v, sat) in hash table tab, if it does not contain an entry (key, ... true).

Locals(in: event, k:thr, pidx):set of lcl

Return all of the local states in p locally enabling e. If there are no states in p locally enabling e then return Ø.

Pick(in: set of lcl):lcl

Remove and return an element from L.


Create nth threads. Build an MDD rooted at root, at level K, encoding the state space and return root, in UT[K].

Union(in: k:thr, pidx, qidx):dsz

Build an MDD rooted at s, a node at level k, in UT[k], encoding the Union of p q. Return s.

DWarc(s in: k:thr, pidx, qidx):bool

If p[i] ≠ 0 for any local state at level k return true otherwise return false.

SetUpArc(in: k:thr, pidx, qidx, ridx):k

Lock (p.ua). Add an arc (q,i) to the end of the list of upward arcs for p; AddOp(k+1, q); Unlock (p.ua);

SetUpArc(k:thr, pidx, qidx, ridx):bool

If the list of upward arcs is not empty retrieve and remove (q,i) from head of list and return true. Otherwise return false.

AddOp(in: k:thr, pidx)


RemoveOp(in: k:thr, pidx, out: op:bool)

Lock (p.ops). Decrement p.ops. If p.ops = 0 set op to false otherwise set op to true. Unlock(p.ops).

ThreadLoop()

If there are no items in the task queue sleep until woken up. Otherwise remove the head item (k, p) from the task queue. If (k, p) is (0, 0) call Terminate() and terminate the thread, otherwise call Saturate(k, p).

Figure 5.9: Supporting functions for the Parallel Saturation algorithm.
5.3.2 The thread pool algorithm

The result of mapping our ideas into code is shown in figure 5.8, with supporting functions described in figure 5.9. The code extends the Saturation algorithm, using a recursive event ordering. Parallel code is highlighted in figure 5.8, dark-shaded code facilitates tasks and removes mutual recursion, while light-shaded code shows locks ensuring correct synchronisation. We use the notation introduced in section 3.2.1, and extend it to include information and locks on the node, which are denoted by p.information/lock.

Node Information

Each MDD node keeps track of the number of tasks that are currently working on it or that will perform work on it in the future (via upward arcs). The functions AddOp and RemoveOp allow current/pending task operations to be added and removed from a node respectively. The Saturation status of the node is indicated by p.saturating and determines if a node with no remaining tasks is saturated from firing all events, or a newly created node waiting to be saturated. Nodes created from event firings store a firing cache (FC) key to add to the firing cache.

Initialisation

Function Gen creates an initial MDD representing the initial state set of the underlying system model and the threads in the thread pool. Each thread calls ThreadLoop to synchronise on the task queue. Tasks are added to the queue for the bottom nodes of the initial MDD.

Saturate

This function first indicates that the node has begun saturating by setting saturating to true. Since the Saturation task is being performed by a thread,
it registers the thread on the node via AddOp. It begins the process of exhaustively firing events on the node by calling FireEvents for each non zero state. Once it has fired the events the task is complete, and it calls RemoveOp. The Saturate function allows the thread to check the status of the node to see whether it is saturated. It can continue work on any nodes dependent upon the node reaching a fixed point.

FireEvents

This function checks whether an event is enabled in the state being fired upon and calls RecFire to fire an enabled event. Successful firings result in the node being updated with the work carried out by the firing. Any updated nodes invoke a recursive call to FireEvents on the updated state.

RecFire

Uncompleted nodes discovered in the cache have upward arcs set from them to the calling node via SetUpArc. For new work, a node is created setting FC key in the process. The thread registers with the new node and adds it to FC as a work request. Upward arcs are set to the calling node. RecFire is recursively called to continue event firing then the thread de-registers from the node. Nodes at the bottom of the MDD generated by the event firing are either added to the task queue or removed if the event is disabled.

NodeSaturated

This function is called when a node is saturated. The node is checked into the unique table. NodeSaturated updates nodes dependent upon the saturated node via upward arcs, and allows the thread to continue working on them. The termination condition occurs when this function is called for the top level node.
Figure 5.10: Parallel Saturation algorithm example (part 1).
Figure 5.11: Parallel Saturation algorithm example (part 2).
Table 5.2: Next-state function for the Parallel Saturation example.

<table>
<thead>
<tr>
<th></th>
<th>$e_1$</th>
<th>$e_2$</th>
<th>$e_3$</th>
<th>$e_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>0 → 1</td>
<td>0 → 2</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>0 → 2</td>
<td>0 → 2, 2 → 1</td>
<td>*</td>
</tr>
<tr>
<td>1</td>
<td>0 → 2</td>
<td>*</td>
<td>0 → 1</td>
<td>*</td>
</tr>
</tbody>
</table>

5.3.3 Example

We illustrate the parallel algorithm in figures 5.10 and 5.11 with an example for a thread pool of two threads. The corresponding next-state function, including four events, is displayed in table 5.2. In figures 5.10 and 5.11, upward arcs are illustrated with a dashed line and labelled with the local state they point to. Unsaturated nodes are white, saturating nodes are light grey, and saturated nodes are dark grey. We indicate the index of a node at the top right of the node. The tasks operating on the node $op$ is shown to the bottom right of the node in brackets. The firing cache key $fc$ is shown in curly brackets next to $op$. We show the status of the queue and threads at the top right of the diagram. The levels of the MDD are listed to the left of the diagram.

- **a**) *Gen* generates the initial MDD. All nodes are marked as *not saturating*. None of the nodes have an FC key $\{fc\}$ since the nodes are not created from a *RecFire* operation. Operations ($op$) is incremented for each upward arc set on a node. *Saturate* tasks are added to the task queue for the bottom nodes of the MDD, i.e., nodes 2.1 and 3.1. The sleeping threads are about to be woken by the new tasks.

- **b**) The threads have woken and removed the tasks from the queue. They both call *Saturate* which marks each target node as *saturating*
and increments op to indicate they are currently being operated upon by the Saturate task.

- c) Saturate(1,2) makes a local update on node 2.1 by firing event \( e_1 \). Saturate(1,3) has completed since no events are enabled on node 3.1 and has decremented op to zero. Since op is zero, NodeSaturated is called which marks node 3.1 as saturated and checks it into the unique table.

- d) Saturate(1,2) has completed firing events and decremented op to zero, marking node 2.1 as saturated. NodeSaturated(1,3) removes the upward arc to node 2.2 and replaces it with a downward arc, decrementing op for node 2.2 in the process. Since op is nonzero, NodeSaturated terminates leaving Thread 2 to sleep.

- e) NodeSaturated(1,2) removes the upward arc to node 2.2 and replaces it with a downward arc and decrements op for node 2.2. Since op is now zero and the node is not saturating, a new Saturate task is added to the task queue for node 2.2. NodeSaturated completes thus allowing the thread to return to sleep.

- f) Thread 1 is woken up by the new task and removes it from the queue. It calls Saturate(2,2) which increments op and marks node 2.2 as saturating.

- g) Saturate(2,2) makes an in-place update on node 2.2 by firing \( e_2 \) to set local state 2 of node 2.2 to point to node 2.1.

- h) Saturate(2,2) completes firing on node 2.2, decrements op and calls NodeSaturated which replaces the upwards arc to node 2.3 with a downward arc and decrements op on node 2.3. Since node 2.3 is not saturating and has no op, a task to Saturate node 2.3 is added to
the queue. NodeSaturated terminates allowing Thread 1 to return to sleep.

• i) Thread 1 is woken by the addition of the task to the queue. It calls Saturate(3, 2) which marks node 2.3 as saturating and increments op. RecFire is called for event $e_3$ which creates node 3.2 that contains an FC key \{3, 2\} indicating that $e_3$ is being fired on node 2.2.

• j) RecFire increments op on node 3.2 and sets an upward arc to node 3.2 incrementing op in the process. It recursively calls RecFire which creates node 4.1

• k) RecFire increments op on node 4.1 and sets an upward arc to node 3.2 incrementing op in the process.

• l) RecFire sets a downward arc from node 4.1 to terminal node 1 and terminates decrementing op in the process. On termination, since op is zero and the node is not saturating a task to Saturate node 4.1 is added to the task queue.

• m) Thread 2 picks up the new task and calls Saturate(1, 4) which marks the node as saturating and increments op. Meanwhile, Thread 1 has continued with RecFire and has discovered node 4.1 as an unsaturated node in the firing cache, while firing $e_3$, which sets an upward arc to node 3.2 and increments op.

• n) Node 4.1 has completed saturating and decrements op to 0 and marks the node as Saturated. Meanwhile, RecFire has completed on node 3.2, decrementing op, and Saturate continues on node 2.3 by firing $e_4$ to make a local update.

• o) Saturate has completed on node 2.3 decrementing op and returning Thread 1 to sleep. Since op is greater than 0, the node is not
yet saturated. \textit{NodeSaturated} has been called on node 4.1 which has discovered the node is the same as node 3.1 while checking it into the \textit{unique table}, and has removed the upward arcs, setting the downward arcs to this node. Since \textit{op} is 0, a \textit{Saturate} task is added to the queue for node 3.2 and the thread goes to sleep.

- \textit{p}) Thread 1 takes the \textit{Saturate} task for node 3.2, setting the thread to \textit{saturating} and incrementing \textit{op}.

- \textit{q}) \textit{Saturate} completes on node 3.2, decrementing \textit{op} to zero and calling \textit{NodeSaturated} which replaces the upward arc to node 2.3 with a downward arc and decrements \textit{op}. Since \textit{op} is 0 and the node is \textit{saturating}, node 2.3 is now \textit{Saturated}. Since this is the root node, \textit{Terminate} is called which instructs the task queue to terminate the threads. The final MDD representing the state-space is shown.

Figure 5.12: The calling order of functions for the sequential and parallel algorithms.
5.3.4 Correctness of the algorithm

The algorithm in figure 5.8 can be expressed in terms of its sequential counterpart. Removing the highlighted parallel code gives us the sequential algorithm which is known to be correct [29]. Thus correctness of the parallel algorithm can be shown by demonstrating that the parallel code allows our algorithm to arrive at the same result and that locks prevent any data races. We can illustrate the calling structure of both sequential and Parallel Saturation using the example in figure 5.12a. The call graphs in figures 5.12b and 5.12c are the calling order of functions for the sequential and parallel code, respectively, where figure 5.12d and figure 5.12e further simplify the order. Function calls in the sequential version are directly replaced by the task queue and upward arcs in the parallel version. Since locks ensure that updating the node is atomic, firing events exhaustively will result in the same MDD shape for the saturated node as in the sequential version.

5.3.5 Implementation concerns

From an efficiency perspective it is important to keep the costs of introducing the extra code into the algorithm down. Upward arcs and node specific locks are of particular concern for memory efficiency, since these introduce an overhead into each node, which can increase the peak consumption of the intermediate MDD. We therefore use pointers to upward arcs and locks, and ensure that any memory allocated to the pointers during the process of saturation is freed as soon as a node has completed saturating. Thus, upward arcs and mutex locks have their own garbage collection policy, that is initiated immediately after a node is saturated.
Figure 5.13: Run-time results for large models.

Figure 5.14: Memory results for large models.
5.3.6 Run-time and memory results

We implemented our thread pool algorithm using C and the POSIX Pthreads library [21]. We applied the algorithms to our benchmark (sec. 4.4) on our experimental machine (sec. 4.4.2). The results for our algorithms using all parameters can be seen in appendix A.

We show the run-time results for large models in figure 5.13, representing the run-time compared to the sequential recursive ordered Saturation algorithm. We illustrate how the algorithm performs across the 4 cores of our machine. Greater than 1 indicates a run-time speed-up, while less than 1 indicates a run-time slowdown.

We show the memory results for large models in figure 5.14, representing the relative memory compared to the sequential recursive ordered Saturation algorithm. As with the run-time results, we show how the algorithm performs across the 4 cores of our machine. Greater than 1 indicates a memory increase. For both run-time and memory, we show the models for the largest parameter we used, since the results are representative for models using other parameters.

Our run-time results show that the algorithm exhibits a speed-up for 2 of the models, which for the FMS model is over 100% faster the sequential algorithm. For these models where parallelism can be exploited, we saw a slight improvement in run-time for larger parameterised models. Generally, however, the parallel algorithm is unable to improve over the sequential algorithm. The parallel algorithm typically uses 2-3x more memory than the sequential algorithm, apart from the slotted ring model, which is significantly affected by the ordering of events. The factors impacting on the run-time and memory are due to a number of specific reasons.
• **Overheads:** The run-time results and memory are affected by the overheads incurred by the parallel algorithm. High overheads often prevent the parallel algorithm from competing with the sequential version. We saw two types of overhead, the parallel overhead incurred by the introduction of locks and threads, and the code overhead incurred by removing mutual recursion from the algorithm. The GNU profiler [56, 57] showed us that the highest run-time overhead comes from the use of upward arcs, and upward arcs also contribute to the memory increase. The parallel overheads for these models can be seen in figure 4.10. When coupled with our run-time results, the profiles provides clear evidence that models with low parallelism, or high scheduling are more difficult to parallelise than those with high parallelism.

• **Extra Work:** The order in which events are fired affects the amount of work the algorithm has to perform. Due to the dependencies between events, parallel events can be fired on smaller state sets than the sequential version. This creates more work and larger intermediate MDDs. The extra work can outweigh the benefits of parallelism. It also introduces higher overheads. The slotted ring model for example, incurs a significant amount of extra work due to event orderings, and subsequently suffers a large run-time and memory penalty.

• **Parallelism:** The number of parallel events and how well this causes the work to branch in parallel during construction affects the level of parallelism of the algorithm. The lower the parallelism, the lower the number of parallel tasks to perform. Low parallelism means cores are under-subscribed during construction. The subscription of the cores can be seen in figure 4.10. The round robin model is a prime example of where the model suffers from low parallelism, and is unable to improve
over the sequential algorithm.

From our results and overhead profiles, we can clearly see that the worst performing models in terms of run-time are the bounded queue, Kanban and leader election models, which suffer from high scheduling overhead. Low parallelism models such as round robin and aloha show less degradation in performance, but are unable to improve their run-time through parallelism. For high parallelism models, 2 out of the 4 models were able to improve over the sequential version. Thus, to unlock the potential of the other high parallelism models, further optimisations are required to the algorithm.

Hence, decreasing the overheads is key to improving the algorithms performance, by investigating these overheads individually. However, it is first worth turning our attention to the affects the architecture has on the performance of the algorithm.

5.4 Four core vs four processor

Notably, when a run-time speed-up is encountered on our architecture, the increase is greater on the second core, and tails off on the third and fourth core. Others have also experienced this phenomenon when using dual core machines with multithreaded algorithms [6]. This would suggest that the operating system scheduling on these cores is inefficient, since dual cores are as efficient as secondary processors.

We can demonstrate that our dual core, dual-processor architecture influences the run-time of the algorithm by comparing our results against those of a four processor machine. We applied the algorithm to our benchmark on a four processor Opteron 875 machine with 32Gb of memory running Scientific Linux 4. Run-time improvements using the four processor architecture were obtained for eight out of the ten models of our benchmark,
with the other two models showing similar performance to the dual-core architecture. However, these improvements were only significant for our two high parallelism models that demonstrated a speed-up. Hence, we show the graphs for the FMS and Dining Philosophers models in figure 5.15. It can be seen from these models that the algorithm scales better on the four processor machine than our dual-core dual-processor architecture, and the four processors facilitate a more effective approximate linear speed-up.

5.4.1 Optimising the queue ordering

In our parallel algorithm, we pick Saturation tasks from the shared queue of the thread pool using a FIFO order. The order in which tasks are picked from the queue can be optimised, since it affects the order in which nodes are saturated, and thus the construction order of the state-space, which, as we showed with our Parallel BFS, can have a significant affect on the algorithms performance.

To investigate this phenomenon, we propose a LIFO queue and a priority queue, where tasks are ordered so that nodes at the bottom of the MDD are picked first, in line with the bottom-up strategy of Saturation. FIFO and LIFO queues have an insertion and deletion complexity of O(1), however, priority queues differ in their complexity for insertion and deletion. To
avoid an insertion and deletion overhead from using a priority queue, we use the efficient heap based priority queue [91], which, for the number of tasks in the queue \( n \), has a complexity for insertion and deletion of \( O(\lg n) \).

We extended our parallel algorithm to incorporate LIFO and priority queue selection mechanisms for the task queue.

We applied the LIFO and priority queue variants to our benchmark (sec. 4.4) on our experimental machine (sec. 4.4.2), the full results of which are shown in appendix A. For six of the models, the ordering in which tasks are picked from the queue makes negligible difference to the run-time and memory of the algorithm. However, for models that are sensitive to the order in which the nodes are saturated, the queue orderings can have a significant impact on both run-time and memory. The LIFO version has the most impact on these factors, improving the run-time on four models, and the memory usage on two models. The priority queue improves the run-time of three models, and the memory consumption of two models, but the improvements are less than those of the LIFO version.

The run-time of the algorithm for the RSM model is shown in figure 5.16. Both the LIFO and priority queue significantly improve the run-time of the algorithm, as well as decreasing memory consumption, which remains the same when adding extra cores. The high parallelism of the model is
unlocked, facilitating a run-time speed up over the sequential algorithm. The other models that were sensitive to the queue ordering were the aloha, leader election and slotted ring models, but despite the optimisation, are unable to show a run-time speed up over the sequential algorithm.

For models that are sensitive to the ordering of events, the results justify the use of a thread pool with a single queue as a mechanism for load balancing and scheduling, as opposed to using multiple queues with either a thread pool or workstealing system. While multiple queues can potentially decrease scheduling overhead for high scheduling models, they also reduce the opportunity to order the way in which tasks are picked, which as observed for the RSM model, can potentially have a significant effect on the run-time of the algorithm.

### 5.5 Optimising the ordering of events

The ordering in which events are fired in parallel can significantly affect Parallel Saturation’s run-time and memory requirements. Figure 5.17 shows the effect of events $e_1$, $e_2$, and $e_3$, with $\text{Top}(e_1) = \text{Top}(e_2) = \text{Top}(e_3) = k$, on the local states at level $k$ (these events may of course affect lower levels as well). When saturating a node $p$ at level $k$, we must repeatedly fire $e_1$, $e_2$, and $e_3$ in $p$, until no more new states are found, i.e., until $p$ encodes a fixed point. However, Saturation does not dictate the order in which these events should be fired. For example, firing $0 \xrightarrow{e_1} 1$ followed by $1 \xrightarrow{e_3} 0$ might...
be sub-optimal, since we might have to fire $1 \rightarrow 0 \rightarrow 0$ again once $0 \rightarrow 1$ has been fired, if this causes $p[1]$ to point to a different node encoding more states. Similarly, if we fire $1 \rightarrow 2$ before firing $0 \rightarrow 1$, all transitions in SCC#2 may have to be fired again.

To address this problem, we use the chaining heuristic of [25], which extracts the strongly connected components (SCCs) from a dynamic transition graph that is built from the static graph of figure 5.17 and the dynamic pattern of non-zero children of node $p$. We use these SCCs to enhance the order of parallel event firings. However, while this chaining heuristic tends to improve run-time and memory in a sequential implementation, it also reduces the potential parallelism and introduces time and memory overheads due to storing the SCC graphs, traversing them and managing parallel access. There are number of issues we need to deal with when adapting the sequential heuristic to our parallelisation: issues involving parallel SCCs and issues involving parallelisation within an SCC.

### 5.5.1 Parallel SCCs

For our example in figure 5.17, we could fire the events in SCC#1 and SCC#3 in parallel. This implies that we must store the number of incoming arcs to SCC#1, SCC#2 and SCC#3, to which we refer as $arcs_{scc}$, so that we can determine when the events in an SCC can be fired. In figure 5.17 SCC#2 has two incoming arcs, which would mean that initially $arcs_2 = 2$. Thus, When SCC#1 and SCC#3 complete their firings, they would both decrement $arcs_2$, so that SCC#2 can be fired when $arcs_2 = 0$.

In our original thread pool algorithm, we kept track of tasks operating on the node in order to determine when a node had become saturated. For parallel SCCs, we need to keep track of tasks operating on the node according to the SCC they are exploring, so that we can determine when all of the
firings have completed within an SCC. We must therefore employ a counter called \textit{scctasks}_{scc}, which keeps track of the number of tasks operating on the node, that have been created as a result of firings within a particular SCC. Since we need to update \textit{scctasks}_{scc} on the completion of a task, this also implies that upward arcs must keep a track of an SCC index.

### 5.5.2 Parallelism within an SCC

We can also exploit parallelism within SCCs. For our example in figure 5.17, within SCC#1, we can fire $0 \xrightarrow{e_1} 1$ and $0 \xrightarrow{e_2} 1$ in parallel. However, we do not wish to fire $1 \xrightarrow{e_3} 0$ before $0 \xrightarrow{e_1} 1$ and $0 \xrightarrow{e_2} 1$ have completed, in order to preserve the benefits of the chaining heuristic. This implies that we must keep track of the number of incoming firings on a state $i$, for which we refer as \textit{incoming}_i, so that when $0 \xrightarrow{e_1} 1$ and $0 \xrightarrow{e_2} 1$ fire, they individually increment \textit{incoming}_1, and decrement \textit{incoming}_1 on completion of a firing. Thus, $1 \xrightarrow{e_3} 0$ will only fire when \textit{incoming}_1=0.

We must also be able to determine when a state has changed for this approach to work correctly. For example, if we fire $0 \xrightarrow{e_1} 1$ and $0 \xrightarrow{e_2} 1$ in parallel, we would increment \textit{incoming}_1 twice, i.e., \textit{incoming}_1=2. Then, if $0 \xrightarrow{e_1} 1$ completes and updates the state, it decrements \textit{incoming}_1, so that \textit{incoming}_1=1. When $0 \xrightarrow{e_2} 1$ completes, it also decrements \textit{incoming}_1, so that \textit{incoming}_1=0. However, if $0 \xrightarrow{e_2} 1$ did not update state 1, then $1 \xrightarrow{e_3} 0$ will not be fired because it has no way of determining that the state has been changed. We therefore keep a flag for each state $i$, called \textit{changed}_i, to indicate whether a state has been updated. Thus, when $0 \xrightarrow{e_1} 1$ completes, it sets \textit{changed}_i = true, and $1 \xrightarrow{e_3} 0$ will fire when \textit{incoming}_1=0 and \textit{changed}_i = true.

The implementation of our chaining heuristic implies extra locking. We must lock all of our new counters, \textit{arcs}_{scc}, \textit{incoming}_i, \textit{scctasks}_{scc}, and
changed, to provide atomic access. As an implementation concern, we must store any mutex locks as pointers so that we can free them as soon as a node has completed saturating, in order to avoid any significant memory increase.

5.5.3 Run-time and memory results

We extended our parallel algorithm to employ the chaining heuristic, and applied it to our benchmark (sec. 4.4) on our experimental machine (sec. 4.4.2), the full results of which are shown in appendix A. We were unable to fit the chaining graphs into memory for the FMS and dining philosophers models, due to the large number of SCCs for these models.

In comparison to our FIFO algorithm, the chaining heuristic shows the potential to dramatically improve memory consumption and run-time. Memory improvements were more visible, with 5 out of the 8 models showing an improvement, compared to run-time improvements for 4 of the 8 models. Although for the majority of the models, the chaining heuristic does not significantly improve the algorithm’s run-time or decrease its memory consumption, it is effective for models that are sensitive to the event ordering. Run-time improvements were made to the algorithm for the RSM, bounded queue, leader election and slotted ring models. Only the RSM model was able to show a run-time speed-up over the sequential algorithm as a result of the improvement.

We show the run-time improvement of the algorithm for the RSM model, and the memory improvement for the slotted ring model in figure 5.18. The chaining heuristic allows the RSM run-time to significantly improve from a 50% slowdown to an over 100% speed-up. For the slotted ring model the memory consumption halves, with the algorithm utilising a similar amount of memory as the number of cores increase. Both improvements are due to
the sensitivity of the ordering of events within the models, which can be exploited by using the chaining heuristic. However, the chaining heuristic can also have a negative effect on run-time as shown for the round robin and Kanban models in figure 5.18.

Observing these run-time results, is it possible to assess the potential for using the chaining heuristic with a more advanced scheduling and load balancing system than our thread pool. Finding a good parallel ordering of events would reduce the necessity for imposing an order over which tasks are picked from the queue. However, the heuristic cannot be used successfully across all of the models, since it can have a negative impact on run-time, and for some models cannot be used at all, due to size of the dynamic transition graph. This suggests that a suitable ordering for all models cannot be imposed on a multiple queue load balancing system using the chaining heuristic, which is significant if we wish to optimise the underlying scheduling system.
5.6 Optimising the locking strategy

For our parallel algorithm, we use an individual lock for the operation caches and the unique table, on a per level basis. We can therefore optimise the locking mechanism by employing several locks per level, as well as per data structure. The unique table and caches employ a hash function to calculate the position of insertion. Thus, we can exploit the hash function for each of these data structures to implement more efficient locking, by assigning a lock to a range of hash values. We call this \textit{regional locking}. The range can be calculated using the modulo of the number of locks we wish to have, on the signature function as shown in figure 5.19. If the hash table needs to be resized, then the locks must be obtained before the resizing takes place.

![Graph showing relative runtime for different models with regional locking and leader election.](image)

Figure 5.20: Regional locking, leader election run-time results.

We implemented regional locking for the unique table and the firing cache, with differing numbers of locks, 4, 8 and 16, for each data structure per level. These were chosen to test how increasing numbers of locks influence the locking overhead. The results for all models using regional locking can be seen in appendix A. For seven of models, the increased number of
locks makes little difference to the run-time, however for three of the models
the locking facilitates a run-time improvement of between 5% and 30%. For
the leader election model shown in figure 5.20, we observe an improvement
of up to 30% for both the unique table and firing cache, which facilitates
improved parallelism on the third and fourth core. However, increasing
the number of locks from 4 to 16 does not further improve the algorithms
performance.

The run-time results demonstrate that locking can be optimised, but for
only a few models. The regional locking was unable to facilitate speed-ups
over the sequential algorithm for these models. Where improvements were
made using 4 locks, further improvements could not be made by increasing
the number of locks to 8 or 16. However, on an architecture with a larger
number of processor cores, increasing the number of locks may facilitate
scalability. Our profiles of the parallel overheads in figure 4.10 also testify
to low locking overheads on the data structures for a four core architecture.

5.7 Chapter summary

In this chapter we parallelised the Saturation algorithm. We began by par-
allelising a BFS using loop parallelisation, which demonstrated that simple
algorithm parallelisations may be inefficient due to the impact of parallel
overheads, and can be surpassed by sequential optimisations. The Chained
BFS showed that breaking sequential optimisations may negate the benefits
of parallelism. We then introduced a parallel implementation of the Satu-
ration algorithm by creating independent parallel tasks by exploiting event
locality, using a thread pool for scheduling and load balancing efficiency. We
implemented our thread pool algorithm, and applied it to our benchmark
of models on our experimental machine. The overheads impacting on the
algorithm were carefully investigated, and several optimisations to the al-
algorithm were made, for the task queue, the event orderings and the locking strategy.

The results from our experimental studies show that even an algorithm that employs several sequential optimisations, such as Saturation, can be parallelised efficiently. However, the model plays a crucial part in deciding whether the parallel variant is more efficient than its sequential counterpart. Optimisations can be made to the algorithm to reduce parallel overheads which, for some models, can significantly improve the run-time of the algorithm, or decrease memory consumption. However, for the vast majority of models within our selected benchmark, it is difficult to improve performance through these optimisations. Where improvements can be made, they are only able to facilitate run-time improvements over the sequential algorithm for high parallelism models. Thus, although we can show that our algorithm is efficiently parallelised, for some models, a heavily optimised system for scheduling and load balancing is required if further parallelism is to be exploited.

In our setting, it is questionable as to whether this can be achieved. For optimisation through architecture adaptations, symbolic data structures are likely to incur undesirable memory penalties for adjusting the data structure to the architecture. It is also questionable as to whether a more complex scheduling and load balancing mechanism such as a per-thread task queue, or workstealing system, would alter Saturation’s construction order significantly enough to negate the benefits of parallelism. For example, our optimisations to the thread pool queue showed that some models can be sensitive to the ordering of tasks picked from the queue. Using a chaining heuristic to optimise the order of events would appear to be the solution to this problem, but our results showed that such a heuristic can also have a negative affect on parallelism. Thus, when parallelising a symbolic state-space
generation algorithm, one must consider whether parallel optimisations to
an efficient underlying scheduling and load balancing system warrant inves-
tigation. We have demonstrated this by showing that optimisations to the
scheduling model may only be suitable for particular models, and the fact
that breaking sequential optimisations can potentially negate the effects of
parallelism across all models.
Chapter 6

Cilk based Saturation

Parallel languages such as Cilk \[10, 49\], tailored to irregular problems, have been shown to offer efficient scheduling and load balancing \[10\]. In this chapter, we continue our investigation into Parallel Saturation, by parallelising it using Cilk. Doing so allows us to examine the benefits of using a parallel model of computation that is highly optimised in its implementation details, and to put into context the efficiency of our thread pool algorithm by comparing it to a Cilk variant of Saturation.

6.1 A Cilk approach to parallelising Saturation

We observed from our thread pool algorithm that symbolic state-space generation algorithms incur significant overheads from parallelisation, making gains in time-efficiency difficult to achieve. Our Parallel Saturation algorithm encountered load imbalance from the irregular sizes of computations during state-space generation, and scheduling overheads since the state-space generation computations can be small. We also observed that addressing these issues can potentially break the effects of sequential optimisations, and require a significant amount of effort to optimise implementation details. Parallel tools that have been developed to automatically reduce these
overheads are thus desirable. To the best of our knowledge, Cilk [10, 49] is the only parallel language that offers both efficient scheduling and load balancing.

6.1.1 Cilk

The Cilk language simplifies parallel programming by allowing the use of C-based functions to express control over the parallelism of a program. The language is powerful enough to facilitate mutually recursive algorithms such as Saturation [29]. It is designed to run efficiently on symmetric processors, e.g., those found in shared-memory machines, and includes a scheduler employing randomised work-stealing, that is theoretically and practically efficient [10]. To achieve efficiency, Cilk employs its own model of multi-threaded computation.

Cilk uses call/return semantics to enable parallelism, and provides keywords that enable the programmer to easily express parallelism. A Cilk function can be specified by using the keyword \texttt{cilk} in front of a C function, and can be \textit{spawned} to run in parallel by using the keyword \texttt{spawn} when calling it. The C function semantics is preserved by allowing the return value of the spawned function to be stored by the parent. Multiple functions can be spawned within the calling function, and the calling function continues its computation while the spawned functions work in parallel. To permit controlled synchronisation of spawned threads, the \texttt{sync} keyword prevents the calling function from continuing its computation until all of its spawned functions have completed. Cilk functions contain an implicit \texttt{sync} before they are allowed to return.

The return value of the calling function can either be stored by the parent once the function completes, or can be handled by the parent in a more complex way via the use of an \texttt{inlet}. An inlet can be specified as an internal
function to a Cilk function, which handles the result of a spawned function. To preserve atomicity, only one completed Cilk function can be handled at a time by the inlet, and further computation by the parent is prevented until the inlet has returned. The spawn and sync keywords cannot be used within an inlet. This restriction arises from Cilk’s inability to support pipelining, making it difficult to express producer/consumer problems such as state-space generation.

### 6.1.2 Workstealing

The principle of workstealing is for a processor that has run out of work to steal uncompleted work from another processor [2, 11, 12]. This implies a one-to-one mapping of threads to each processor, each with their own queue of uncompleted work. Compared to the single queue approach of our thread pool, workstealing reduces synchronisation, since each thread can add and remove tasks to and from its own queue, and only needs to synchronise on another queue when it has run out of tasks to perform, rather than frequently synchronise on a shared queue.

Within Cilk, each thread has its own deque, onto which function frames can be pushed and popped, while other threads steal frames from the bottom of the deque. We illustrate this workstealing layout in figure 6.1. For the implementation component, when a Cilk function encounters the spawn keyword, it pushes the parent frame onto the deque and calls the spawned child function. When the child function returns, the parent frame is popped from the deque and work continues. However, if the frame was stolen from the deque by another thread, then the function returns a dummy value, and control returns to the Cilk run-time system.

The check for the stolen frame is subtle and highly optimised, ensuring that synchronisation on the deque does not contribute any overhead to the
algorithm. The system design implies that Cilk must extend the C language semantics, to employ its own system for managing frames, and transferring execution control to the Cilk run-time system. The workstealing policy is randomised, which can be shown to be theoretically efficient [10]. However, for the purposes of parallelising Saturation, Cilk eliminates the potential for ordering the way in which functions are stolen from the queue.

6.1.3 Parallel divide and conquer Saturation

Using Cilk we can easily interpret Saturation as a parallel algorithm in *divide and conquer* format. The algorithm in figure 6.2 shows the original Saturation algorithm [29] expressed as a parallel algorithm in Cilk. We call this our *inlet variant* of the algorithm. The algorithm is parallelised via *task parallelism* in exactly the same way as in our thread pool, where the *Fire*
Unfortunately, the divide and conquer approach presents two particular problems. Firstly, by using inlets and expressing the problem the divide and conquer format, we break the chaining effect in a similar way as we did for the Parallel BFS. Secondly, expressing the problem in this way creates a load imbalance, since all firings must be completed before performing

Figure 6.2: Cilk based Saturation using inlets.
CHAPTER 6. CILK BASED SATURATION

Figure 6.3: Calling order for spawns.

the union operation. The ordering can be shown in figure 6.3(a), where function *Saturate* must wait for the two spawned *Fire* calls to synchronise before spawning more work. It would be more efficient to perform the union operation and then immediately spawn new work using the ordering in figure 6.3(c).

6.1.4 Expressing a producer/consumer problem.

The call/return semantics of Cilk means that we cannot elegantly deal with a spawned function as soon as it has completed, since we cannot tell when an individual firing has completed outside of an inlet. It is desirable to use an inlet to spawn off more work as soon as a firing completes; however, inlets are restricted to prevent new functions being spawned from within them. We could attempt to let the calling function know when a firing has completed via an inlet through the use of a flag or a queue, but Cilk does not allow us to suspend the calling function outside of a sync statement, which means that the calling function would have to continue monitoring for completed child functions. It is undesirable to tie up the processor with a function that is polling in this manner, since it largely performs useless work. This means that the ordering of work shown in figure 6.3(c) cannot be achieved using Cilk, due to the restrictions arising from Cilk’s lack of
Figure 6.4: Cilk based Saturation without exploiting call/return semantics.

We can rewrite the algorithm to continue spawning firings when they have completed by utilising the spawn keyword, without exploiting the call/return semantics of Cilk. An example algorithm is shown in figure 6.4, which breaks the original Saturation function into a number of sub-functions. We call this our *spawn variant* of the algorithm. The event ordering is the same as the sequential recursive ordering. Once a spawned firing has been completed, it performs the union in $\text{DoUnion}$ and then immediately spawns further firings on the updated state.

Expressing our producer/consumer problem by bypassing the call/return semantics is not ideal. When the functions complete, they do not have any further work to do, yet they are left on the Cilk function deque after spawning more work, waiting for it to complete. This ordering is shown in figure 6.3(b). A large number of functions can be unnecessarily left on the deque during the state-space generation process, which potentially increases the amount of memory required for the process. The problem is compounded because of the mutually recursive calls between $\text{Saturate}$ and $\text{Fire}$. 
6.1.5 Spawn variant vs thread pool algorithm

To achieve the ideal ordering in figure 6.3(c), we must relinquish functions from the deque while spawned work executes. Since a function frame requires its own storage, a smaller amount of memory could be used by storing only the variables that are required once a spawned child is complete, instead of storing the calling function. While we are unable to achieve the ideal ordering using Cilk, our thread pool algorithm uses an upward arc for storing required variables, and relinquishes functions, leaving the child functions to complete the work of the calling function. Thus, the thread pool algorithm is able to achieve the ideal ordering via the use of upward arcs. However, we observed that the upward arcs also require extra memory for both the arcs and the locks to synchronise the arcs. A comparison between the memory penalties of upward arcs and function frames can be determined by comparing the thread pool algorithm to a spawn variant using Cilk.

In terms of scheduling efficiency, the thread pool is not as efficient as a highly optimised implementation such as Cilk, due to the time required to add and remove a task from the queue, which is larger than the time taken to execute functions on the Cilk deque. Expressing the problem in Cilk also removes the necessity to break mutual recursion, by using upward arcs, which introduce a code overhead into the algorithm. When we compare Cilk to a thread pool using the functionally lightweight Fibonacci problem in [49] on a dual-processor, dual-core machine, Cilk reports a 3× speed-up whereas the thread pool reports a 2× slowdown due to the time spent adding and removing tasks from the queue, and the code overhead from managing upward arcs. Thus, theoretically, the spawn version in Cilk should show greater run-time improvements than the thread pool algorithm.
6.2 Run-time and memory results

We implemented three algorithms in Cilk, one using inlets and two using the spawn ordering, which either fire events with our without the use of the chaining heuristic. We applied the algorithms to our benchmark (sec. 4.4) on our experimental machine (sec. 4.4.2). The full results can be seen in appendix A.

![Figure 6.5: Cilk inlet variant, selected run-time and memory results.](image)

6.2.1 Inlet variant

Compared to the thread pool algorithm the inlet version demonstrated a slowdown in run-time across all models, and an increase in memory consumption for nine of the models. This is unsurprising given the combined effect of breaking the chaining order and the load imbalance introduced by using inlets. The inlet version is unable to improve the run-time against the sequential Saturation algorithm for any model. For models that were previously able to improve their run-time using the thread pool, a run-time
slowdown, as well as a larger memory increase, is demonstrated against the sequential algorithm. We show the run-time slowdown and memory increase for these models in figure 6.5.

The results imply that by breaking chaining, the inlet version suffers from similar negative effects to performance as the Parallel BFS algorithm. The loss in run-time from breaking the chaining order negates any benefits of parallelism. Even though the inlet version was able to improve the run-time of six models on four cores against its own run-time on one core, these improvements were too small to improve over the run-time of the sequential algorithm. Thus, the Saturation algorithm cannot be efficiently parallelised using the traditional Cilk divide and conquer approach, since breaking the chaining significantly reduces the run-time, and the load imbalance from the ordering prevents parallelism from significantly improving the run-time.

Figure 6.6: Cilk spawn variant, selected run-time results.
6.2.2 Spawn variant

The spawn variant manages to increase the run-time for nine of the models in comparison to the thread pool algorithm. Four of the models, queens, FMS, round robin, and dining philosophers, showed a run-time improvement over the sequential algorithm. We show the run-time of the spawn variant compared to the FIFO queue for these models in figure 6.6. For the queens model, the improvement is significant, unlocking the parallelism of the model, and facilitating a run-time improvement over the sequential algorithm. However, this is largely due to the removal of the code overhead from upward arcs, rather than the efficient scheduling of Cilk. For the other models we show, the run-time improvements due to improved scheduling efficiency are relatively small in comparison.

![Figure 6.7: Cilk spawn variant, selected memory results.](image)

For memory consumption, the increase for the spawn variant over the thread pool algorithm is dramatic. For eight out of the ten models, the
algorithm shows a significant increase in memory, ranging from $12 \times$ to over $20 \times$ that of the sequential algorithm. For our four models for which run-time is improved over the sequential algorithm, we show the large memory increase in figure 6.7. This increase is due to the large size of the Cilk deque.

The results from our spawn variant show that Cilk is more effective in exploiting parallelism than our hand-crafted thread pool algorithm, but incurs a significant memory overhead due to its lack of support for pipelining. Even though scheduling efficiency is improved, the algorithm is still unable to outperform the sequential algorithm in terms of run-time, for high scheduling models. For models that already showed run-time gains over the sequential version using the thread pool algorithm, the improvements from using Cilk are relatively small. For the models where a significant run-time improvement was made in comparison to the thread pool algorithm, some of the run-time gains can be attributed to the elimination of upward arcs, as well as scheduling efficiency.

### 6.2.3 Spawn variant with chaining heuristic

As with our thread pool algorithm, we were unable to employ the chaining heuristic for the FMS and dining philosophers models. For the models we generated results for, the run-time of the spawn variant using the chaining heuristic improved over the thread pool for five out of the eight models. We show the run-time results for some of the models in figure 6.8.

For the RSM model, we saw a superlinear speed-up due to both the improved ordering of the chaining heuristic and the scheduling efficiency of Cilk. This combination of optimisations was also able to facilitate a run-time improvement over the sequential algorithm for the leader election model. Since leader election is a high scheduling model that is sensitive to the order of events, improving scheduling efficiency and optimising the
event ordering is highly beneficial. However, the converse effect occurs for the queens and round robin models, where the run-time improvements made from using Cilk are lost using the chaining heuristic.

For memory consumption, we only saw a small improvement for several models. We show the memory consumption for the leader election and bounded queue models in figure 6.9. The chaining heuristic is unable to significantly reduce the large memory overhead encountered from the growth of the Cilk deque, and the memory consumption of the spawn algorithm employing the chaining heuristic remains extremely high compared to the thread pool algorithm.

Our results demonstrate that the chaining heuristic can facilitate a run-time improvement for several models and, for those that are sensitive to the ordering of events, the improvement can be significant. We observed a superlinear speed-up for a high parallelism model, and a run-time improve-
Figure 6.9: Cilk spawn variant with chaining heuristic, selected memory results.

...ment over the sequential algorithm for a high scheduling model. For some models, the chaining heuristic has a detrimental effect to the run-time, removing any run-time gains from using the efficient scheduling model of Cilk. The heuristic facilitated small memory improvements for several models, but was unable to eliminate the high memory penalty arising from the size of the Cilk deque.

6.3 Chapter summary

In this chapter, we investigated the parallelisability of Saturation using Cilk, a parallel language for irregular problems, that is proven to be efficient for scheduling and load balancing. Indeed, the implementation of the Cilk work-stealing model provides negligible synchronisation overhead when scheduling tasks on processors. This ensures that the scheduling is more efficient than our thread pool algorithm. Cilk also allowed the problem to be expressed without the use of upward arcs to break mutual recursion, which removed the code overhead from employing them.

These efficiency optimisations are reflected in the results we obtained from running the spawn variant of Cilk, which was able to improve the time-efficiency of Parallel Saturation over our thread pool algorithm for
nine out of the ten of the models in our benchmark. However, even with the improved scheduling, without a chaining heuristic, Cilk was only able to improve the run-time for four of the models over the sequential algorithm. For the models that benefitted from improved scheduling, without a large gain from the elimination of upward arcs, only small improvements were made over our thread pool algorithm. This testifies to the suitability of using a single queue for parallelising Saturation.

When the chaining heuristic was employed, a superlinear speed-up was obtained for one of our models, but the heuristic also reduced the run-time of other models. As with our thread pool algorithm, imposing an order on events to optimise the algorithm is highly dependent upon the model for its success.

While the usage of Cilk led to improvements in time-efficiency, the restrictions imposed by the Cilk language implied an enormous memory overhead. Comparing the results to our thread pool algorithm showed that preventing idle functions from inhabiting the stack removes this memory overhead. Pipelining is therefore an essential feature of any language for parallelising symbolic state-space generators. To the best of our knowledge, there is currently no parallel language fitting this description. However, a possible future direction of parallel irregular languages extending the Cilk model of multithreaded computation to include pipelining is proposed in [103]. On top of this, our problem is likely to be solved if Cilk employs tail call optimisation [9], which would remove function frames from the end of the deque that have no more work to perform.
Chapter 7

Anticipated Firing

Parallelising the state-space generation algorithm can break sequential optimisations, resulting in a significantly negative effect on the efficiency of the corresponding parallel algorithm. In this chapter we investigate anticipated firing [22, 24], an approach to parallelising Distributed Saturation [23], that leaves the sequential state-space generation algorithm intact, while utilising extra processors to perform work that may be required by it in the future. Doing so allows us to assess the effectiveness of using extra cores to aid the state-space generation algorithm rather than parallelising it.

7.1 Origins

Anticipated firing [22, 24] was first introduced and used for a distributed version of Saturation on a network of workstations (NOW) [23]. In the distributed setting, a workstation may be frequently idle while it waits for another workstation to complete its work. During its idle time, a workstation can perform work on the state-space that it may need to carry out at some point in the future. From an efficiency perspective, this can only have a positive impact on the run-time of the algorithm, since it does not interfere with the state-space generation task.
For Saturation, work that may be needed in the future can be performed on an idle workstation by firing events on a node. If an event $e$ is fired on a node $p$ and results in a new node $q$, then $q$ can be checked into the unique table, and the pair $\{e, q\}$ can be put into the firing cache, so that $q$ can be retrieved if $e$ is fired on $p$ again at some point in the future. Events can be speculatively fired on a node while a workstation is idle, in order to perform and cache as much work as possible. The effectiveness of the method depends on how much useful work is carried out in advance, i.e., creating MDDs that will be used by the state-space generation task in the future. However, it is not possible to determine whether $e$, will be fired on $p$ in the future.

### 7.2 Shared-memory anticipated firing

We adopt the anticipated firing idea to parallelise Saturation on a shared-memory architecture. Doing so allows us to make comparisons between utilising idle cores for predictive work, and using cores for parallel state-space construction. We use our thread pool to allow anticipated work to be created and scheduled as tasks, while a main thread constructs the state-space and creates anticipated firing tasks. Then, all we are required to decide is when tasks should be created and how they should be dealt with, while leaving the state-space generation task to perform its work without interruption.

We present a naive algorithm for anticipated firing in figure 7.1. The function $\texttt{AnticipatedFire}$ carries out firings on a given node $p$ at level $k$ for events at level $k+1$, by calling $\texttt{RecFire}$ to perform the event firings. The creation of anticipated firing tasks is performed by $\texttt{RecFire}$, which adds a task for anticipated firing as soon as node $s$ becomes saturated at line 16, by calling $\texttt{AddTask}$. We do not wish to initiate anticipated firings on
CHAPTER 7. ANTICIPATED FIRING

AnticipatedFire\((k; l; e; p; \text{idx})\)
Fire events on \(p\), a node at level \(k\) from above \(k\), in anticipation of future work.

\[
\text{declare } e: \text{event};
\]
\[
\begin{align*}
1. & \quad \text{foreach } e \in \mathcal{E}^{k+1} \text{ do} \\
2. & \quad \text{RecFire}(e, k, p, \text{false});
\end{align*}
\]

RecFire\((e; l; e; q; \text{idx}; CT; \text{bool}); \text{idx}\)
Build an MDD rooted at \(s\), a node at level \(l\), in \(UT[l]\), encoding \(N^*_l(N_e(B(q)))\). Return \(s\).

\[
\text{declare } L: \text{set of } lcl;
\]
\[
\text{declare } f, u, s, r: \text{idx};
\]
\[
\text{declare } i, j: lcl;
\]
\[
\text{declare } \text{sCng}: \text{bool};
\]
\[
1. & \quad \text{if } l < \text{Last}(e) \text{ then return } q; \\
2. & \quad \text{if } \text{Find}(FC[l], \{q, e\}, s) \text{ then return } s; \\
3. & \quad s \leftarrow \text{NewNode}(l); \\
4. & \quad \text{sCng} \leftarrow \text{false}; \\
5. & \quad l \leftarrow \text{Locals}(e, l, q); \\
6. & \quad \text{while } L \neq \emptyset \text{ do} \\
7. & \quad \text{i} \leftarrow \text{Pick}(L); \\
8. & \quad f \leftarrow \text{RecFire}(e, l-1, q[i], CT); \\
9. & \quad \text{if } f \neq 0 \text{ then} \\
10. & \quad \text{foreach } j \in N^*_l(i) \text{ do} \\
11. & \quad \text{u} \leftarrow \text{Union}(l-1, f, s[j]); \\
12. & \quad \text{if } u \neq s[j] \text{ then} \\
13. & \quad \text{s[j]} \leftarrow u; \text{sCng} \leftarrow \text{true}; \\
14. & \quad \text{if } \text{sCng} \text{ then } \text{Saturate}(l, s); \\
15. & \quad r \leftarrow \text{Check}(l, s); \\
16. & \quad \text{if } CT \text{ and } r = s \text{ then } \text{AddTask}(l, s); \\
17. & \quad \text{Insert}(FC[l], \{q, e\}, s); \\
18. & \quad \text{return } s;
\]

ThreadLoop()
If there are no items in the task queue sleep until woken up. Otherwise remove the head item \((k, p)\) from the task queue, and call AnticipatedFire\((k, p)\).

Figure 7.1: Pseudo-code for the anticipated firing algorithm.
nodes that are created as a result of other anticipated firings for now, as this may lead to a large amount of unnecessary work. Thus, we include a boolean flag called create task (CT) which indicates whether RecFire should create an anticipated firing task after a node has become saturated, and AnticipatedFire calls RecFire with CT set to false. Although we do not show it in our pseudo-code, any function calls to RecFire from functions being executed by the main thread set CT to true. We also do not want to perform anticipated firing nodes that have been previously checked into the hash table, since they are more likely to have events fired on them.

For our thread pool, the ThreadLoop function allows a thread to pick an anticipated firing task and carry it out by calling AnticipatedFire. We still have a one-to-one thread to processor allocation, but we allocate one thread as the main thread, which does not synchronise on the task queue and is left uninterrupted to Saturate the root node. We do not show locks in our pseudo-code, but as with our thread pool implementation we lock the operation caches and the unique table on a per level basis for atomic access. This is the only part of the anticipated firing that can interfere with the state-space generation task, since we (potentially) introduce a locking overhead to the main thread.

7.2.1 Selecting events

As soon as a node has become saturated, anticipated firings are performed on that node to carry out work that may be used in the future. Key to the efficiency of the anticipated firing approach is selecting tasks and events that will carry out useful work. We cannot accurately predict whether an event will carry out useful work. However, there are a number of heuristics that we can employ in our algorithm that are potentially useful for selecting appropriate events.
• The number of events we wish to fire on a node is significant. If we fire a large number of events on a node, we potentially tie up a processor working on that node. It may be better to fire less events on that node, since the anticipated firing will complete quickly, which reduces the chance that the main thread will begin firing events on the node before the anticipated task has completed. Firing less events also frees up a thread quickly to perform newly created anticipated firing tasks.

• Instead of firing events from one level above the node selected for anticipated firing, we may wish to fire events from several levels above this node. This is potentially beneficial since there is less chance that the main thread will start firing the same events on that node as the anticipated task, before the anticipated firings have completed.

• We may wish to discard anticipated firing tasks for saturated nodes that are at lower levels of the MDD, since the closer to the level that the main thread is Saturating for, the greater the possibility for new and potentially useful work. For example, by the time the main thread reaches level fifty of the MDD, nodes below level ten are likely to have been fired upon several times, creating more cached work, and less new work than for nodes at higher levels.

We can use these heuristics for selecting anticipation tasks and events with our anticipated firing algorithm in order to try and increase the amount of useful work performed by idle processors.

7.3 Run-time and memory results

We implemented our anticipated firing algorithm using C and the POSIX Pthreads library [21]. We applied the algorithms to our benchmark (sec. 4.4)
Figure 7.2: Run-time results for the anticipated firing algorithm on large models (1).
Figure 7.3: Run-time results for the anticipated firing algorithm on large models (2).

on our experimental machine (sec. 4.4.2), and included three heuristics for selecting events.

- The events fired $EVF$, is the number of events fired on a node by an anticipated firing task.

- The moving minimum, $MM$ is the minimum of number of MDD levels between the node considered for anticipated firing, and the node that is being saturated by the main thread.

- The firing level $FL$ is the number of MDD levels above the node considered for anticipated firing, that events will be fired for.

We present results for our large models in figures 7.2 and 7.3, showing the different heuristic settings we tried. For the default settings $EVF$ is 1, $MM$ is unset and $FL$ is 1. We show the run-time differences when employing heuristics by changing $EVF$ to 2, $MM$ to 3 and $FL$ to 3.
For five of the models, a run-time improvement is made by the anticipated firing algorithm over the sequential Saturation algorithm. The improvements are between approximately 5% and 20% over the sequential algorithm. Slowdowns are relatively small, with the largest approximately 10% less than the sequential algorithm. Typically, the largest increase in speed is observed between cores 1 and 2, while cores 3 and 4 tend to make little difference to the run-time. This suggests that relatively small amounts of useful work are performed on these cores, or that not enough anticipation tasks are created, which we cannot currently determine.

On one core, the algorithm demonstrates a slowdown across all models. This is due to the instrumentation of the Pthreads library, and the code overhead from mutex locks that causes a small parallel overhead. Any further slowdowns on the other cores are caused by synchronisation, since this is the only way in which the threads used for anticipated firing can interfere with the main thread.

The use of heuristics makes a small and unpredictable difference to the run-time across all models, where the largest difference in run-time for employing a heuristic is around 5%. This would suggest that more complex factors influence the amount of useful work that can be performed.

Our results suggest that anticipated firing is useful for facilitating small run-time improvements on several models without significantly impacting on the state-space generation task for the others. This is due to the use of idle processors without interfering with the main task of saturating the root node. Greater run-time improvements are not possible by employing heuristics for creating tasks and selecting events based on the measures we defined.
7.4 Chapter summary

In this chapter we investigated anticipated firing, using idle processors to perform work without impacting on the main task of generating the state-space. Doing so gave us an opportunity to assess the performance of anticipated firing against our parallelisations of the state-space algorithm. This is particularly relevant as, in comparison to our thread pool and Cilk approaches, the use of anticipated firing will not break any sequential optimisations of the state-space generation algorithm.

We showed that, although any impact is small on the main task, run-time improvements from using anticipated firing with a few heuristics for selecting useful work, are also relatively small. In comparison to our thread pool and Cilk algorithms that can show run-time improvements of over 100%, anticipated firing is only able to show run-time improvements of approximately 20%. Scalability is also an issue since adding a third or fourth core does not significantly improve run-time.

It is key to the success of anticipated firing, for larger, more scalable run-time gains, that heuristics are developed that have a better success rate than our heuristics for predictively selectively useful work to be performed by the idle processors. We do not expect anticipated firing to incur any significant memory penalty, since we can discard anticipated work if memory is becoming exhausted.
Chapter 8

Conclusions

We began this thesis by asking whether symbolic model checkers could be parallelised for run-time efficiency. To answer this question we proposed to parallelise the Saturation symbolic state-space generation algorithm on shared-memory architectures. Doing so allowed us to thoroughly investigate whether a highly optimised state-space generation algorithm could be improved further by utilising extra processors for run-time improvements.

To answer whether symbolic model checkers could be parallelised for run-time efficiency with confidence, we first proposed an approach to measuring and evaluating parallel state-space construction algorithms in order to select a suitable benchmark. We argued that direct measurement of overheads was required to accurately determine the impact of parallel overheads that are prevalent in irregular algorithms such as state-space generation algorithms, and suggested a profiling tool that could accurately measure these overheads at run-time. This allowed us to select and categorise an appropriate benchmark of models according to the overheads they imposed upon our parallel algorithms. Throughout this thesis we used our benchmark, along with profiles of our algorithms’ performance, to carefully analyse the efficiency of our algorithms and pinpoint where (in)efficiencies in the algorithm were arising.
Before we parallelised Saturation, we parallelised a BFS algorithm, to gain an insight into how state-space generation algorithm parallelisations perform against sequential optimisations. We compared the Parallel BFS against a BFS employing chaining, an optimisation that is heavily relied upon by the Saturation algorithm for vast improvements in run-time efficiency. The results from running our BFS on a dual-core dual-processor architecture showed that breaking a sequential optimisation can lead to devastating run-time and memory penalties, which cannot be overcome by any benefits of parallelisation. This put into context the very nature of the problem that we were exploring, since Saturation outperforms breadth-first symbolic state-space generation by orders of magnitude in both memory and time, making it arguably the most efficient state-space generation algorithm for globally-asynchronous locally-synchronous discrete event systems. Thus, we had to consider that any parallelisations we made to Saturation which had the potential to compromise sequential optimisations to the algorithm could make the parallel algorithm inefficient.

The observation that any parallelisation should preserve sequential optimisations added weight to the already heavy problem of how to parallelise Saturation. Not only does the algorithm suffer from high parallel overheads from its irregular nature, it is also mutually recursive. This meant that any load balancing scheme we employed would either have to deal with the mutual recursion, or break it. It also ruled out using simple parallel scheduling libraries such as OpenMP which are unable to deal with mutual recursion, and meant that our implementation had to take place at much lower level, by using Pthreads. Further to this, we were required to re-implement the C++ Saturation algorithm in C, so that we could avoid the problem of C++ objects synchronising, and access our data structures efficiently in parallel.
Although this allowed us more profiling tools for measurement, it also further optimised an already heavily optimised algorithm.

We proceeded by attempting to exploit event locality within the Saturation algorithm for the purpose of creating independent parallel tasks. Although this gave us a method for creating task independence, we also had to overcome the small computations arising from event locality, by grouping tasks together into larger tasks. We proposed the use of a thread pool with a one-to-one thread to processor mapping for load balancing purposes, and broke the mutual recursion in the algorithm by using upward arcs in the MDD structure, which directly replaced the function call stack between the mutually recursive functions. We chose a single shared queue on which the threads synchronised, which Parallel Saturation tasks could be added to and removed from. We justified the use of our the thread pool by showing that it was efficient compared to creating threads on demand to perform tasks, and that the use of anything other than a single queue would afford us a loss in control of the chaining order.

Our results from implementing and running our Parallel Saturation algorithm on our dual-processor dual-core machine showed that we could achieve speed-ups of over 100% compared to the sequential algorithm for models exhibiting high parallelism, but for models exhibiting low parallelism or high scheduling overheads, we were unable to improve over the sequential algorithm. Any run-time gains could be improved by using a four processor architecture without a dual-core setup. We then carefully analysed our algorithm according to its overheads, by using our direct measurement approach, and made several optimisations to the algorithm to reduce these overheads. The order in which tasks were picked from the queue was adjusted, which testified to the use of our shared queue approach, as opposed to a more complex multiple queue system that could impact on run-time
through the loss of control over the ordering the tasks. We used a chaining heuristic for ordering the events and saw significant memory improvements and run-time improvements for some of our models, but the heuristic had a negative run-time impact on others, meaning that it could not be used with multiple queues for ordering efficiency. We optimised the locking strategy, which made some improvements to a few models, and testified to the overall efficiency of our locking approach.

We then turned our attention to issues of scheduling and load balancing efficiency by parallelising Saturation using Cilk, a parallel language for addressing irregularity. The lack of support for pipelining in Cilk meant that we could not parallelise Saturation using Cilk without incurring a significant memory penalty due to the large number of function calls left on the Cilk deque. This showed that our thread pool was efficient in managing memory by facilitating pipelining through the use of upward arcs. Cilk made run-time improvements for all models over our thread pool, and speed ups over the sequential algorithm for four models. The improvements over our thread pool were due to both the efficient scheduling strategy of Cilk and the elimination of the necessity for upward arcs. However, using Cilk also testified to the efficiency of our thread pool algorithm, since the workstealing strategy facilitated only small run-time gains over the thread pool approach. We also used the chaining heuristic with the Cilk algorithm, which resulted in a superlinear speed-up for one of our models but had a negative impact on other models, and was unable to significantly improve the large memory impact caused by using Cilk. This showed that even an efficient scheduling and load balancing strategy could not improve the run-time of the parallel algorithm over the sequential algorithm when the models exhibit low parallelism and high scheduling overheads, or are significantly affected by the order in which events are fired or tasks are performed.
The final part of our investigation was to compare Parallel Saturation to using idle processors to predictively perform work while leaving the Saturation algorithm unaltered and uninterrupted. The method has an inherent advantage in that it leaves sequential optimisations of the algorithm intact, which means that there is little loss of time, or large overheads for the parallel algorithm. However, in comparison to our Cilk and thread pool approaches, the speed-ups are small and do not scale well. This could potentially be improved if heuristics can be employed to generate sufficient useful work for idle processors to perform. However, our initial investigations into selecting useful work showed that it may not be possible to effectively predict.

Using the results from our Parallel BFS, thread pool and Cilk implementation we can determine that an efficient parallelisation of a symbolic state-space generation algorithm is difficult to achieve. This is because the memory efficiency of the parallel algorithm must be considered, and removing sequential optimisations that make the parallel algorithm inefficient needs to be avoided. Further to this, even when the algorithm is parallelised efficiently, run-time improvements over the sequential algorithm are not guaranteed, since several of the models in our benchmark are unable to facilitate parallelism, and there is always an example that can be constructed which can prevent parallelism.

We conclude that it is better to focus on sequential optimisations rather than parallelising a symbolic state-space generation algorithm. Currently, as we have shown by parallelising Saturation, a great deal of effort has to go into optimising the parallel algorithm. If the same amount of time is spent on a sequential optimisation of Saturation, it could well be the case that the sequential optimisation renders the parallel algorithm inefficient. For this reason we suggest the answer to the initial question we asked, can symbolic model checkers be parallelised for run-time efficiency? is no, because
at the present moment, the time required for optimising the parallel state-
space generation algorithm can be negated by the time spent optimising
the sequential strategy. However, generic solutions for parallelising irregu-
lar problems such as Cilk show potential, for their ease of implementation,
as well as scheduling and load balancing efficiency. If such languages can
develop to allow for memory efficiency in the future by employing tail call
optimisation, or by facilitating pipelining, then this will speed up the process
of efficiently parallelising symbolic state-space generation algorithms.

8.1 Future research

There are three ways in which our work can be extended: (i) scalability
of our Parallel Saturation algorithms across a larger number of cores, (ii)
memory efficiency improvements to Cilk, (iii) anticipated firing heuristics
for selecting useful work.

For the purposes of scalability, our thread pool and Cilk algorithms
can be run on a shared-memory machine with a larger number of cores.
This would determine whether run-time gains that were achieved for mod-
els which demonstrated parallelism can be improved, or whether gains tail
off when adding cores. If adding cores does not improve the run-time of our
parallel algorithms, then the causes can be investigated, which would be due
to either lack of model parallelisability or increased locking and scheduling
overhead. A comparison between our thread pool algorithm and Cilk algo-

rithm on a larger number of cores could be used to assess any scalability
benefits from employing a multiple queue workstealing system for schedul-
ing and load balancing efficiency. Our regional locking mechanism could be
used to help facilitate scalability by employing more locks on the unique
table and operation caches.

For Cilk, a further research direction is one which discovers a solution to
the memory impact arising from expressing a producer/consumer problem, such as symbolic state-space generation. This could either involve extending the Cilk model of multithreaded computation to include pipelining as proposed in [103], or by employing tail call optimisation [9], which would remove useless function frames from the end of the deque. Such a solution would allow Cilk to be used for parallel producer/consumer problems where memory efficiency is a concern, including symbolic state-space generation algorithms.

Finally, more heuristics for selecting useful anticipated firing tasks to utilise idle cores on a shared-memory architecture can be developed. Heuristics that successfully improve the amount of useful work have the potential to facilitate further run-time gains for our anticipated firing algorithm. This is particularly desirable as anticipated firing can be employed confidently without a significant negative impact to run-time or memory efficiency.
Bibliography


[58] Grumberg, O., Heyman, T., Ifergan, N., and Schuster, A. Achieving speedups in distributed symbolic reachability analysis through asym-


Appendix A

Run-time and memory results
Table A.1: Run-time and memory results for the slotted ring model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>fifo</td>
<td>0.23</td>
<td>0.20</td>
</tr>
<tr>
<td>lifo</td>
<td>0.25</td>
<td>0.28</td>
</tr>
<tr>
<td>pq</td>
<td>0.24</td>
<td>0.27</td>
</tr>
<tr>
<td>chain</td>
<td>0.22</td>
<td>0.24</td>
</tr>
<tr>
<td>ut4</td>
<td>0.23</td>
<td>0.20</td>
</tr>
<tr>
<td>ut8</td>
<td>0.23</td>
<td>0.20</td>
</tr>
<tr>
<td>ut16</td>
<td>0.23</td>
<td>0.20</td>
</tr>
<tr>
<td>fc4</td>
<td>0.23</td>
<td>0.21</td>
</tr>
<tr>
<td>fc8</td>
<td>0.23</td>
<td>0.21</td>
</tr>
<tr>
<td>fc16</td>
<td>0.23</td>
<td>0.21</td>
</tr>
<tr>
<td>4proc</td>
<td>0.22</td>
<td>0.19</td>
</tr>
<tr>
<td>spawn</td>
<td>0.36</td>
<td>0.29</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.35</td>
<td>0.39</td>
</tr>
<tr>
<td>inlet</td>
<td>0.17</td>
<td>0.19</td>
</tr>
</tbody>
</table>

N = 120  Sequential: 15.77(s) 13405440(b)

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>fifo</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>lifo</td>
<td>0.28</td>
<td>0.31</td>
</tr>
<tr>
<td>pq</td>
<td>0.28</td>
<td>0.31</td>
</tr>
<tr>
<td>chain</td>
<td>0.26</td>
<td>0.29</td>
</tr>
<tr>
<td>ut4</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>ut8</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>ut16</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>fc4</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>fc8</td>
<td>0.26</td>
<td>0.23</td>
</tr>
<tr>
<td>fc16</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>4proc</td>
<td>0.25</td>
<td>0.23</td>
</tr>
<tr>
<td>spawn</td>
<td>0.38</td>
<td>0.30</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.37</td>
<td>0.40</td>
</tr>
<tr>
<td>inlet</td>
<td>0.16</td>
<td>0.20</td>
</tr>
</tbody>
</table>

N = 150  Sequential: 30.88(s) 25441840(b)
APPENDIX A. RUN-TIME AND MEMORY RESULTS

Table A.2: Run-time and memory results for the round robin model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>N = 180 Sequential: 7.95(s) 1165764(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.77</td>
<td>0.96</td>
</tr>
<tr>
<td>lifo</td>
<td>0.78</td>
<td>0.97</td>
</tr>
<tr>
<td>pq</td>
<td>0.78</td>
<td>0.96</td>
</tr>
<tr>
<td>chain</td>
<td>0.50</td>
<td>0.51</td>
</tr>
<tr>
<td>ut4</td>
<td>0.77</td>
<td>0.95</td>
</tr>
<tr>
<td>ut8</td>
<td>0.76</td>
<td>0.96</td>
</tr>
<tr>
<td>ut16</td>
<td>0.77</td>
<td>0.96</td>
</tr>
<tr>
<td>fc4</td>
<td>0.77</td>
<td>0.95</td>
</tr>
<tr>
<td>fc8</td>
<td>0.78</td>
<td>0.96</td>
</tr>
<tr>
<td>fc16</td>
<td>0.78</td>
<td>0.98</td>
</tr>
<tr>
<td>4proc</td>
<td>0.75</td>
<td>0.97</td>
</tr>
<tr>
<td>spawn</td>
<td>0.97</td>
<td>1.14</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.68</td>
<td>0.69</td>
</tr>
<tr>
<td>inlet</td>
<td>0.33</td>
<td>0.34</td>
</tr>
<tr>
<td>N = 210 Sequential: 15.43(s) 1574304(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.79</td>
<td>0.99</td>
</tr>
<tr>
<td>lifo</td>
<td>0.81</td>
<td>1.01</td>
</tr>
<tr>
<td>pq</td>
<td>0.80</td>
<td>1.00</td>
</tr>
<tr>
<td>chain</td>
<td>0.50</td>
<td>0.51</td>
</tr>
<tr>
<td>ut4</td>
<td>0.79</td>
<td>0.99</td>
</tr>
<tr>
<td>ut8</td>
<td>0.78</td>
<td>0.98</td>
</tr>
<tr>
<td>ut16</td>
<td>0.79</td>
<td>0.99</td>
</tr>
<tr>
<td>fc4</td>
<td>0.79</td>
<td>0.98</td>
</tr>
<tr>
<td>fc8</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>fc16</td>
<td>0.79</td>
<td>0.97</td>
</tr>
<tr>
<td>4proc</td>
<td>0.79</td>
<td>0.98</td>
</tr>
<tr>
<td>spawn</td>
<td>0.98</td>
<td>1.15</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.68</td>
<td>0.69</td>
</tr>
<tr>
<td>inlet</td>
<td>0.32</td>
<td>0.33</td>
</tr>
<tr>
<td>N = 240 Sequential: 41.75(s) 2044044(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.80</td>
<td>1.01</td>
</tr>
<tr>
<td>lifo</td>
<td>0.82</td>
<td>1.03</td>
</tr>
<tr>
<td>pq</td>
<td>0.82</td>
<td>1.02</td>
</tr>
<tr>
<td>chain</td>
<td>0.51</td>
<td>0.51</td>
</tr>
<tr>
<td>ut4</td>
<td>0.80</td>
<td>1.00</td>
</tr>
<tr>
<td>ut8</td>
<td>0.80</td>
<td>1.01</td>
</tr>
<tr>
<td>ut16</td>
<td>0.79</td>
<td>1.01</td>
</tr>
<tr>
<td>fc4</td>
<td>0.80</td>
<td>1.01</td>
</tr>
<tr>
<td>fc8</td>
<td>0.80</td>
<td>1.01</td>
</tr>
<tr>
<td>fc16</td>
<td>0.81</td>
<td>1.01</td>
</tr>
<tr>
<td>4proc</td>
<td>0.81</td>
<td>1.02</td>
</tr>
<tr>
<td>spawn</td>
<td>0.98</td>
<td>1.19</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.69</td>
<td>0.71</td>
</tr>
<tr>
<td>inlet</td>
<td>0.29</td>
<td>0.30</td>
</tr>
</tbody>
</table>
Table A.3: Run-time and memory results for the Kanban model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>N = 25 Sequential: 2.80(s) 7334600(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.54</td>
<td>0.64</td>
</tr>
<tr>
<td>lifo</td>
<td>0.58</td>
<td>0.71</td>
</tr>
<tr>
<td>pq</td>
<td>0.48</td>
<td>0.62</td>
</tr>
<tr>
<td>chain</td>
<td>0.53</td>
<td>0.58</td>
</tr>
<tr>
<td>ut4</td>
<td>0.53</td>
<td>0.66</td>
</tr>
<tr>
<td>ut8</td>
<td>0.54</td>
<td>0.67</td>
</tr>
<tr>
<td>ut16</td>
<td>0.52</td>
<td>0.66</td>
</tr>
<tr>
<td>fc4</td>
<td>0.55</td>
<td>0.68</td>
</tr>
<tr>
<td>fc8</td>
<td>0.53</td>
<td>0.67</td>
</tr>
<tr>
<td>fc16</td>
<td>0.53</td>
<td>0.68</td>
</tr>
<tr>
<td>4proc</td>
<td>0.54</td>
<td>0.66</td>
</tr>
<tr>
<td>spawn</td>
<td>0.31</td>
<td>0.29</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.63</td>
<td>0.65</td>
</tr>
<tr>
<td>inlet</td>
<td>0.19</td>
<td>0.25</td>
</tr>
<tr>
<td>N = 30 Sequential: 5.06(s) 13784976(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.56</td>
<td>0.67</td>
</tr>
<tr>
<td>lifo</td>
<td>0.60</td>
<td>0.73</td>
</tr>
<tr>
<td>pq</td>
<td>0.52</td>
<td>0.68</td>
</tr>
<tr>
<td>chain</td>
<td>0.52</td>
<td>0.55</td>
</tr>
<tr>
<td>ut4</td>
<td>0.56</td>
<td>0.70</td>
</tr>
<tr>
<td>ut8</td>
<td>0.56</td>
<td>0.69</td>
</tr>
<tr>
<td>ut16</td>
<td>0.55</td>
<td>0.68</td>
</tr>
<tr>
<td>fc4</td>
<td>0.54</td>
<td>0.70</td>
</tr>
<tr>
<td>fc8</td>
<td>0.55</td>
<td>0.69</td>
</tr>
<tr>
<td>fc16</td>
<td>0.56</td>
<td>0.69</td>
</tr>
<tr>
<td>4proc</td>
<td>0.55</td>
<td>0.68</td>
</tr>
<tr>
<td>spawn</td>
<td>0.32</td>
<td>0.29</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.63</td>
<td>0.65</td>
</tr>
<tr>
<td>inlet</td>
<td>0.17</td>
<td>0.23</td>
</tr>
<tr>
<td>N = 35 Sequential: 10.22(s) 23957940(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.57</td>
<td>0.70</td>
</tr>
<tr>
<td>lifo</td>
<td>0.61</td>
<td>0.75</td>
</tr>
<tr>
<td>pq</td>
<td>0.61</td>
<td>0.75</td>
</tr>
<tr>
<td>chain</td>
<td>0.53</td>
<td>0.55</td>
</tr>
<tr>
<td>ut4</td>
<td>0.58</td>
<td>0.73</td>
</tr>
<tr>
<td>ut8</td>
<td>0.57</td>
<td>0.72</td>
</tr>
<tr>
<td>ut16</td>
<td>0.57</td>
<td>0.72</td>
</tr>
<tr>
<td>fc4</td>
<td>0.58</td>
<td>0.72</td>
</tr>
<tr>
<td>fc8</td>
<td>0.57</td>
<td>0.72</td>
</tr>
<tr>
<td>fc16</td>
<td>0.57</td>
<td>0.72</td>
</tr>
<tr>
<td>4proc</td>
<td>0.56</td>
<td>0.69</td>
</tr>
<tr>
<td>spawn</td>
<td>0.32</td>
<td>0.30</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.62</td>
<td>0.63</td>
</tr>
<tr>
<td>inlet</td>
<td>0.16</td>
<td>0.22</td>
</tr>
</tbody>
</table>
Table A.4: Run-time and memory results for the FMS model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>N = 11</td>
<td>Sequential: 12.17(s)</td>
<td>3148980(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.79</td>
<td>1.46</td>
</tr>
<tr>
<td>lifo</td>
<td>0.78</td>
<td>1.44</td>
</tr>
<tr>
<td>pq</td>
<td>0.79</td>
<td>1.32</td>
</tr>
<tr>
<td>chain</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ut4</td>
<td>0.78</td>
<td>1.43</td>
</tr>
<tr>
<td>ut8</td>
<td>0.79</td>
<td>1.45</td>
</tr>
<tr>
<td>ut16</td>
<td>0.80</td>
<td>1.41</td>
</tr>
<tr>
<td>fc4</td>
<td>0.80</td>
<td>1.40</td>
</tr>
<tr>
<td>fc8</td>
<td>0.79</td>
<td>1.44</td>
</tr>
<tr>
<td>fc16</td>
<td>0.78</td>
<td>1.41</td>
</tr>
<tr>
<td>4proc</td>
<td>0.80</td>
<td>1.45</td>
</tr>
<tr>
<td>spawn</td>
<td>0.98</td>
<td>1.60</td>
</tr>
<tr>
<td>spawnC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>inlet</td>
<td>0.70</td>
<td>0.77</td>
</tr>
<tr>
<td>N = 13</td>
<td>Sequential: 55.49(s)</td>
<td>8173844(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.78</td>
<td>1.49</td>
</tr>
<tr>
<td>lifo</td>
<td>0.81</td>
<td>1.50</td>
</tr>
<tr>
<td>pq</td>
<td>0.81</td>
<td>1.49</td>
</tr>
<tr>
<td>chain</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ut4</td>
<td>0.80</td>
<td>1.48</td>
</tr>
<tr>
<td>ut8</td>
<td>0.79</td>
<td>1.47</td>
</tr>
<tr>
<td>ut16</td>
<td>0.78</td>
<td>1.48</td>
</tr>
<tr>
<td>fc4</td>
<td>0.79</td>
<td>1.48</td>
</tr>
<tr>
<td>fc8</td>
<td>0.79</td>
<td>1.45</td>
</tr>
<tr>
<td>fc16</td>
<td>0.80</td>
<td>1.47</td>
</tr>
<tr>
<td>4proc</td>
<td>0.81</td>
<td>1.48</td>
</tr>
<tr>
<td>spawn</td>
<td>0.98</td>
<td>1.62</td>
</tr>
<tr>
<td>spawnC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>inlet</td>
<td>0.71</td>
<td>0.81</td>
</tr>
<tr>
<td>N = 14</td>
<td>Sequential: 119.98(s)</td>
<td>12591300(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.80</td>
<td>1.51</td>
</tr>
<tr>
<td>lifo</td>
<td>0.81</td>
<td>1.50</td>
</tr>
<tr>
<td>pq</td>
<td>0.79</td>
<td>1.47</td>
</tr>
<tr>
<td>chain</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ut4</td>
<td>0.80</td>
<td>1.50</td>
</tr>
<tr>
<td>ut8</td>
<td>0.80</td>
<td>1.48</td>
</tr>
<tr>
<td>ut16</td>
<td>0.80</td>
<td>1.49</td>
</tr>
<tr>
<td>fc4</td>
<td>0.80</td>
<td>1.49</td>
</tr>
<tr>
<td>fc8</td>
<td>0.80</td>
<td>1.49</td>
</tr>
<tr>
<td>fc16</td>
<td>0.80</td>
<td>1.50</td>
</tr>
<tr>
<td>4proc</td>
<td>0.81</td>
<td>1.53</td>
</tr>
<tr>
<td>spawn</td>
<td>0.97</td>
<td>1.67</td>
</tr>
<tr>
<td>spawnC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>inlet</td>
<td>0.74</td>
<td>0.83</td>
</tr>
</tbody>
</table>
Table A.5: Run-time and memory results for the queens model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>N = 11 Sequential: 1.79(s) 4248776(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.44</td>
<td>0.48</td>
</tr>
<tr>
<td>lifo</td>
<td>0.45</td>
<td>0.48</td>
</tr>
<tr>
<td>pq</td>
<td>0.45</td>
<td>0.48</td>
</tr>
<tr>
<td>chain</td>
<td>0.52</td>
<td>0.50</td>
</tr>
<tr>
<td>ut4</td>
<td>0.44</td>
<td>0.28</td>
</tr>
<tr>
<td>ut8</td>
<td>0.44</td>
<td>0.48</td>
</tr>
<tr>
<td>ut16</td>
<td>0.44</td>
<td>0.48</td>
</tr>
<tr>
<td>fc4</td>
<td>0.43</td>
<td>0.47</td>
</tr>
<tr>
<td>fc8</td>
<td>0.43</td>
<td>0.48</td>
</tr>
<tr>
<td>fc16</td>
<td>0.43</td>
<td>0.47</td>
</tr>
<tr>
<td>4proc</td>
<td>0.44</td>
<td>0.48</td>
</tr>
<tr>
<td>spawn</td>
<td>0.89</td>
<td>1.01</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.64</td>
<td>0.62</td>
</tr>
<tr>
<td>inlet</td>
<td>0.31</td>
<td>0.35</td>
</tr>
<tr>
<td>N = 12 Sequential: 19.42(s) 19920672(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.50</td>
<td>0.54</td>
</tr>
<tr>
<td>lifo</td>
<td>0.49</td>
<td>0.55</td>
</tr>
<tr>
<td>pq</td>
<td>0.50</td>
<td>0.54</td>
</tr>
<tr>
<td>chain</td>
<td>0.56</td>
<td>0.54</td>
</tr>
<tr>
<td>ut4</td>
<td>0.51</td>
<td>0.54</td>
</tr>
<tr>
<td>ut8</td>
<td>0.50</td>
<td>0.55</td>
</tr>
<tr>
<td>ut16</td>
<td>0.50</td>
<td>0.54</td>
</tr>
<tr>
<td>fc4</td>
<td>0.51</td>
<td>0.55</td>
</tr>
<tr>
<td>fc8</td>
<td>0.51</td>
<td>0.54</td>
</tr>
<tr>
<td>fc16</td>
<td>0.51</td>
<td>0.54</td>
</tr>
<tr>
<td>4proc</td>
<td>0.51</td>
<td>0.54</td>
</tr>
<tr>
<td>spawn</td>
<td>0.90</td>
<td>1.17</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.66</td>
<td>0.65</td>
</tr>
<tr>
<td>inlet</td>
<td>0.32</td>
<td>0.35</td>
</tr>
<tr>
<td>N = 13 Sequential: 439.16(s) 99807456(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.55</td>
<td>0.59</td>
</tr>
<tr>
<td>lifo</td>
<td>0.53</td>
<td>0.60</td>
</tr>
<tr>
<td>pq</td>
<td>0.54</td>
<td>0.60</td>
</tr>
<tr>
<td>chain</td>
<td>0.62</td>
<td>0.60</td>
</tr>
<tr>
<td>ut4</td>
<td>0.55</td>
<td>0.59</td>
</tr>
<tr>
<td>ut8</td>
<td>0.56</td>
<td>0.58</td>
</tr>
<tr>
<td>ut16</td>
<td>0.56</td>
<td>0.58</td>
</tr>
<tr>
<td>fc4</td>
<td>0.55</td>
<td>0.59</td>
</tr>
<tr>
<td>fc8</td>
<td>0.55</td>
<td>0.59</td>
</tr>
<tr>
<td>fc16</td>
<td>0.55</td>
<td>0.59</td>
</tr>
<tr>
<td>4proc</td>
<td>0.56</td>
<td>0.60</td>
</tr>
<tr>
<td>spawn</td>
<td>0.94</td>
<td>1.60</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.66</td>
<td>0.66</td>
</tr>
<tr>
<td>inlet</td>
<td>0.30</td>
<td>0.37</td>
</tr>
</tbody>
</table>
APPENDIX A. RUN-TIME AND MEMORY RESULTS

Table A.6: Run-time and memory results for the leader election model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>N = 6</td>
<td>Sequential: 3.78(s) 2422704(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.54</td>
<td>0.50</td>
</tr>
<tr>
<td>lifo</td>
<td>0.56</td>
<td>0.58</td>
</tr>
<tr>
<td>pq</td>
<td>0.55</td>
<td>0.57</td>
</tr>
<tr>
<td>chain</td>
<td>0.60</td>
<td>0.68</td>
</tr>
<tr>
<td>ut4</td>
<td>0.55</td>
<td>0.54</td>
</tr>
<tr>
<td>ut8</td>
<td>0.55</td>
<td>0.57</td>
</tr>
<tr>
<td>ut16</td>
<td>0.55</td>
<td>0.57</td>
</tr>
<tr>
<td>fc4</td>
<td>0.56</td>
<td>0.52</td>
</tr>
<tr>
<td>fc8</td>
<td>0.55</td>
<td>0.52</td>
</tr>
<tr>
<td>fc16</td>
<td>0.55</td>
<td>0.52</td>
</tr>
<tr>
<td>4proc</td>
<td>0.54</td>
<td>0.50</td>
</tr>
<tr>
<td>spawn</td>
<td>0.80</td>
<td>0.84</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.90</td>
<td>0.94</td>
</tr>
<tr>
<td>inlet</td>
<td>0.28</td>
<td>0.32</td>
</tr>
<tr>
<td></td>
<td>N = 7</td>
<td>Sequential: 24.11(s) 7063232(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.52</td>
<td>0.52</td>
</tr>
<tr>
<td>lifo</td>
<td>0.56</td>
<td>0.57</td>
</tr>
<tr>
<td>pq</td>
<td>0.55</td>
<td>0.57</td>
</tr>
<tr>
<td>chain</td>
<td>0.58</td>
<td>0.63</td>
</tr>
<tr>
<td>ut4</td>
<td>0.54</td>
<td>0.55</td>
</tr>
<tr>
<td>ut8</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>ut16</td>
<td>0.54</td>
<td>0.56</td>
</tr>
<tr>
<td>fc4</td>
<td>0.54</td>
<td>0.51</td>
</tr>
<tr>
<td>fc8</td>
<td>0.55</td>
<td>0.52</td>
</tr>
<tr>
<td>fc16</td>
<td>0.54</td>
<td>0.51</td>
</tr>
<tr>
<td>4proc</td>
<td>0.54</td>
<td>0.52</td>
</tr>
<tr>
<td>spawn</td>
<td>0.77</td>
<td>0.81</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.90</td>
<td>1.00</td>
</tr>
<tr>
<td>inlet</td>
<td>0.23</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>N = 8</td>
<td>Sequential: 128.11(s) 16107968(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.51</td>
<td>0.53</td>
</tr>
<tr>
<td>lifo</td>
<td>0.54</td>
<td>0.55</td>
</tr>
<tr>
<td>pq</td>
<td>0.53</td>
<td>0.54</td>
</tr>
<tr>
<td>chain</td>
<td>0.57</td>
<td>0.62</td>
</tr>
<tr>
<td>ut4</td>
<td>0.53</td>
<td>0.53</td>
</tr>
<tr>
<td>ut8</td>
<td>0.53</td>
<td>0.53</td>
</tr>
<tr>
<td>ut16</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>fc4</td>
<td>0.54</td>
<td>0.54</td>
</tr>
<tr>
<td>fc8</td>
<td>0.53</td>
<td>0.55</td>
</tr>
<tr>
<td>fc16</td>
<td>0.54</td>
<td>0.55</td>
</tr>
<tr>
<td>4proc</td>
<td>0.52</td>
<td>0.54</td>
</tr>
<tr>
<td>spawn</td>
<td>0.74</td>
<td>0.78</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.91</td>
<td>1.06</td>
</tr>
<tr>
<td>inlet</td>
<td>0.19</td>
<td>0.23</td>
</tr>
</tbody>
</table>
Table A.7: Run-time and memory results for the bounded queue model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N = 30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Sequential: 2.08(s) 2241036(b)</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.40</td>
<td>0.38</td>
</tr>
<tr>
<td>lifo</td>
<td>0.41</td>
<td>0.39</td>
</tr>
<tr>
<td>pq</td>
<td>0.41</td>
<td>0.39</td>
</tr>
<tr>
<td>chain</td>
<td>0.41</td>
<td>0.42</td>
</tr>
<tr>
<td>ut4</td>
<td>0.39</td>
<td>0.38</td>
</tr>
<tr>
<td>ut8</td>
<td>0.39</td>
<td>0.38</td>
</tr>
<tr>
<td>ut16</td>
<td>0.40</td>
<td>0.39</td>
</tr>
<tr>
<td>fc4</td>
<td>0.39</td>
<td>0.39</td>
</tr>
<tr>
<td>fc8</td>
<td>0.40</td>
<td>0.38</td>
</tr>
<tr>
<td>fc16</td>
<td>0.40</td>
<td>0.38</td>
</tr>
<tr>
<td>4proc</td>
<td>0.40</td>
<td>0.38</td>
</tr>
<tr>
<td>spawn</td>
<td>0.98</td>
<td>0.93</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.89</td>
<td>0.90</td>
</tr>
<tr>
<td>inlet</td>
<td>0.48</td>
<td>0.46</td>
</tr>
<tr>
<td>N = 50</td>
<td>Sequential: 24.48(s) 15112996(b)</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.43</td>
<td>0.40</td>
</tr>
<tr>
<td>lifo</td>
<td>0.43</td>
<td>0.41</td>
</tr>
<tr>
<td>pq</td>
<td>0.43</td>
<td>0.40</td>
</tr>
<tr>
<td>chain</td>
<td>0.42</td>
<td>0.43</td>
</tr>
<tr>
<td>ut4</td>
<td>0.43</td>
<td>0.40</td>
</tr>
<tr>
<td>ut8</td>
<td>0.43</td>
<td>0.41</td>
</tr>
<tr>
<td>ut16</td>
<td>0.43</td>
<td>0.41</td>
</tr>
<tr>
<td>fc4</td>
<td>0.43</td>
<td>0.40</td>
</tr>
<tr>
<td>fc8</td>
<td>0.43</td>
<td>0.40</td>
</tr>
<tr>
<td>fc16</td>
<td>0.43</td>
<td>0.40</td>
</tr>
<tr>
<td>4proc</td>
<td>0.40</td>
<td>0.38</td>
</tr>
<tr>
<td>spawn</td>
<td>0.97</td>
<td>0.95</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.89</td>
<td>0.91</td>
</tr>
<tr>
<td>inlet</td>
<td>0.44</td>
<td>0.41</td>
</tr>
<tr>
<td>N = 70</td>
<td>Sequential: 146.79(s) 54895356(b)</td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.48</td>
<td>0.43</td>
</tr>
<tr>
<td>lifo</td>
<td>0.48</td>
<td>0.43</td>
</tr>
<tr>
<td>pq</td>
<td>0.48</td>
<td>0.44</td>
</tr>
<tr>
<td>chain</td>
<td>0.47</td>
<td>0.46</td>
</tr>
<tr>
<td>ut4</td>
<td>0.48</td>
<td>0.43</td>
</tr>
<tr>
<td>ut8</td>
<td>0.48</td>
<td>0.44</td>
</tr>
<tr>
<td>ut16</td>
<td>0.48</td>
<td>0.44</td>
</tr>
<tr>
<td>fc4</td>
<td>0.48</td>
<td>0.43</td>
</tr>
<tr>
<td>fc8</td>
<td>0.48</td>
<td>0.43</td>
</tr>
<tr>
<td>fc16</td>
<td>0.48</td>
<td>0.43</td>
</tr>
<tr>
<td>4proc</td>
<td>0.47</td>
<td>0.44</td>
</tr>
<tr>
<td>spawn</td>
<td>0.98</td>
<td>0.94</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.90</td>
<td>0.91</td>
</tr>
<tr>
<td>inlet</td>
<td>0.38</td>
<td>0.37</td>
</tr>
</tbody>
</table>
# APPENDIX A. RUN-TIME AND MEMORY RESULTS

Table A.8: Run-time and memory results for the dining philosophers model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>N = 20</td>
<td>Sequential: 14.91(s) 569608(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>lifo</td>
<td>0.82</td>
<td>0.98</td>
</tr>
<tr>
<td>pq</td>
<td>0.82</td>
<td>0.99</td>
</tr>
<tr>
<td>chain</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ut4</td>
<td>0.81</td>
<td>0.99</td>
</tr>
<tr>
<td>ut8</td>
<td>0.80</td>
<td>0.98</td>
</tr>
<tr>
<td>ut16</td>
<td>0.81</td>
<td>0.99</td>
</tr>
<tr>
<td>fc4</td>
<td>0.79</td>
<td>0.98</td>
</tr>
<tr>
<td>fc8</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>fc16</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>4proc</td>
<td>0.80</td>
<td>0.99</td>
</tr>
<tr>
<td>spawn</td>
<td>0.96</td>
<td>1.11</td>
</tr>
<tr>
<td>spawnC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>inlet</td>
<td>0.49</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>N = 40</td>
<td>Sequential: 33.36(s) 1097560(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.81</td>
<td>1.01</td>
</tr>
<tr>
<td>lifo</td>
<td>0.83</td>
<td>1.02</td>
</tr>
<tr>
<td>pq</td>
<td>0.82</td>
<td>1.04</td>
</tr>
<tr>
<td>chain</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ut4</td>
<td>0.81</td>
<td>1.03</td>
</tr>
<tr>
<td>ut8</td>
<td>0.82</td>
<td>1.02</td>
</tr>
<tr>
<td>ut16</td>
<td>0.81</td>
<td>1.01</td>
</tr>
<tr>
<td>fc4</td>
<td>0.82</td>
<td>1.02</td>
</tr>
<tr>
<td>fc8</td>
<td>0.82</td>
<td>1.01</td>
</tr>
<tr>
<td>fc16</td>
<td>0.82</td>
<td>1.01</td>
</tr>
<tr>
<td>4proc</td>
<td>0.82</td>
<td>1.02</td>
</tr>
<tr>
<td>spawn</td>
<td>0.96</td>
<td>1.15</td>
</tr>
<tr>
<td>spawnC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>inlet</td>
<td>0.48</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>N = 80</td>
<td>Sequential: 76.93(s) 2321768(b)</td>
</tr>
<tr>
<td>fifo</td>
<td>0.84</td>
<td>1.04</td>
</tr>
<tr>
<td>lifo</td>
<td>0.86</td>
<td>1.04</td>
</tr>
<tr>
<td>pq</td>
<td>0.85</td>
<td>1.07</td>
</tr>
<tr>
<td>chain</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ut4</td>
<td>0.85</td>
<td>1.04</td>
</tr>
<tr>
<td>ut8</td>
<td>0.84</td>
<td>1.05</td>
</tr>
<tr>
<td>ut16</td>
<td>0.84</td>
<td>1.04</td>
</tr>
<tr>
<td>fc4</td>
<td>0.85</td>
<td>1.04</td>
</tr>
<tr>
<td>fc8</td>
<td>0.85</td>
<td>1.04</td>
</tr>
<tr>
<td>fc16</td>
<td>0.84</td>
<td>1.04</td>
</tr>
<tr>
<td>4proc</td>
<td>0.85</td>
<td>1.04</td>
</tr>
<tr>
<td>spawn</td>
<td>0.96</td>
<td>1.18</td>
</tr>
<tr>
<td>spawnC</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>inlet</td>
<td>0.48</td>
<td>0.47</td>
</tr>
</tbody>
</table>
Table A.9: Run-time and memory results for the RSM model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>N = 332 Sequential: 5.43(s) 0267568(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.59</td>
<td>0.54</td>
</tr>
<tr>
<td>lifo</td>
<td>0.86</td>
<td>0.76</td>
</tr>
<tr>
<td>pq</td>
<td>0.70</td>
<td>0.70</td>
</tr>
<tr>
<td>chain</td>
<td>0.71</td>
<td>0.79</td>
</tr>
<tr>
<td>ut4</td>
<td>0.60</td>
<td>0.54</td>
</tr>
<tr>
<td>ut8</td>
<td>0.59</td>
<td>0.53</td>
</tr>
<tr>
<td>ut16</td>
<td>0.59</td>
<td>0.54</td>
</tr>
<tr>
<td>fc4</td>
<td>0.60</td>
<td>0.55</td>
</tr>
<tr>
<td>fc8</td>
<td>0.60</td>
<td>0.55</td>
</tr>
<tr>
<td>fc16</td>
<td>0.60</td>
<td>0.55</td>
</tr>
<tr>
<td>4proc</td>
<td>0.60</td>
<td>0.55</td>
</tr>
<tr>
<td>spawn</td>
<td>0.51</td>
<td>0.53</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.78</td>
<td>0.89</td>
</tr>
<tr>
<td>inlet</td>
<td>0.61</td>
<td>0.60</td>
</tr>
<tr>
<td>N = 532 Sequential: 37.63(s) 23307704(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.64</td>
<td>0.57</td>
</tr>
<tr>
<td>lifo</td>
<td>1.03</td>
<td>1.01</td>
</tr>
<tr>
<td>pq</td>
<td>0.82</td>
<td>0.80</td>
</tr>
<tr>
<td>chain</td>
<td>1.10</td>
<td>1.25</td>
</tr>
<tr>
<td>ut4</td>
<td>0.66</td>
<td>0.58</td>
</tr>
<tr>
<td>ut8</td>
<td>0.65</td>
<td>0.58</td>
</tr>
<tr>
<td>ut16</td>
<td>0.65</td>
<td>0.58</td>
</tr>
<tr>
<td>fc4</td>
<td>0.66</td>
<td>0.57</td>
</tr>
<tr>
<td>fc8</td>
<td>0.66</td>
<td>0.57</td>
</tr>
<tr>
<td>fc16</td>
<td>0.66</td>
<td>0.57</td>
</tr>
<tr>
<td>4proc</td>
<td>0.65</td>
<td>0.58</td>
</tr>
<tr>
<td>spawn</td>
<td>0.53</td>
<td>0.69</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.84</td>
<td>1.32</td>
</tr>
<tr>
<td>inlet</td>
<td>0.59</td>
<td>0.58</td>
</tr>
<tr>
<td>N = 832 Sequential: 315.38(s) 74024440(b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fifo</td>
<td>0.74</td>
<td>0.62</td>
</tr>
<tr>
<td>lifo</td>
<td>1.74</td>
<td>2.08</td>
</tr>
<tr>
<td>pq</td>
<td>0.90</td>
<td>1.21</td>
</tr>
<tr>
<td>chain</td>
<td>1.80</td>
<td>2.41</td>
</tr>
<tr>
<td>ut4</td>
<td>0.76</td>
<td>0.64</td>
</tr>
<tr>
<td>ut8</td>
<td>0.76</td>
<td>0.64</td>
</tr>
<tr>
<td>ut16</td>
<td>0.75</td>
<td>0.64</td>
</tr>
<tr>
<td>fc4</td>
<td>0.76</td>
<td>0.64</td>
</tr>
<tr>
<td>fc8</td>
<td>0.76</td>
<td>0.63</td>
</tr>
<tr>
<td>fc16</td>
<td>0.76</td>
<td>0.63</td>
</tr>
<tr>
<td>4proc</td>
<td>0.75</td>
<td>0.63</td>
</tr>
<tr>
<td>spawn</td>
<td>0.53</td>
<td>0.70</td>
</tr>
<tr>
<td>spawnC</td>
<td>1.03</td>
<td>2.89</td>
</tr>
<tr>
<td>inlet</td>
<td>0.58</td>
<td>0.56</td>
</tr>
</tbody>
</table>
Table A.10: Run-time and memory results for the aloha model.

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>fifo</td>
<td>0.74</td>
<td>0.78</td>
</tr>
<tr>
<td>lifo</td>
<td>0.76</td>
<td>0.79</td>
</tr>
<tr>
<td>pq</td>
<td>0.76</td>
<td>0.78</td>
</tr>
<tr>
<td>chain</td>
<td>0.70</td>
<td>0.71</td>
</tr>
<tr>
<td>ut4</td>
<td>0.75</td>
<td>0.78</td>
</tr>
<tr>
<td>ut8</td>
<td>0.75</td>
<td>0.78</td>
</tr>
<tr>
<td>ut16</td>
<td>0.75</td>
<td>0.78</td>
</tr>
<tr>
<td>fc4</td>
<td>0.75</td>
<td>0.79</td>
</tr>
<tr>
<td>fc8</td>
<td>0.75</td>
<td>0.79</td>
</tr>
<tr>
<td>fc16</td>
<td>0.76</td>
<td>0.70</td>
</tr>
<tr>
<td>4proc</td>
<td>0.74</td>
<td>0.78</td>
</tr>
<tr>
<td>spawn</td>
<td>0.96</td>
<td>1.20</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.71</td>
<td>0.76</td>
</tr>
<tr>
<td>inlet</td>
<td>0.59</td>
<td>0.60</td>
</tr>
</tbody>
</table>

N = 40 Sequential: 2.68(s) 15879556(b)

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>fifo</td>
<td>0.80</td>
<td>0.84</td>
</tr>
<tr>
<td>lifo</td>
<td>0.82</td>
<td>0.85</td>
</tr>
<tr>
<td>pq</td>
<td>0.80</td>
<td>0.84</td>
</tr>
<tr>
<td>chain</td>
<td>0.81</td>
<td>0.82</td>
</tr>
<tr>
<td>ut4</td>
<td>0.81</td>
<td>0.85</td>
</tr>
<tr>
<td>ut8</td>
<td>0.81</td>
<td>0.84</td>
</tr>
<tr>
<td>ut16</td>
<td>0.80</td>
<td>0.85</td>
</tr>
<tr>
<td>fc4</td>
<td>0.81</td>
<td>0.85</td>
</tr>
<tr>
<td>fc8</td>
<td>0.81</td>
<td>0.84</td>
</tr>
<tr>
<td>fc16</td>
<td>0.81</td>
<td>0.84</td>
</tr>
<tr>
<td>4proc</td>
<td>0.81</td>
<td>0.85</td>
</tr>
<tr>
<td>spawn</td>
<td>0.97</td>
<td>1.35</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.72</td>
<td>0.79</td>
</tr>
<tr>
<td>inlet</td>
<td>0.53</td>
<td>0.56</td>
</tr>
</tbody>
</table>

N = 70 Sequential: 22.18(s) 82907316(b)

<table>
<thead>
<tr>
<th>Type</th>
<th>Relative Time (Cores)</th>
<th>Relative Memory (Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>fifo</td>
<td>0.82</td>
<td>0.84</td>
</tr>
<tr>
<td>lifo</td>
<td>0.83</td>
<td>0.85</td>
</tr>
<tr>
<td>pq</td>
<td>0.83</td>
<td>0.85</td>
</tr>
<tr>
<td>chain</td>
<td>0.82</td>
<td>0.84</td>
</tr>
<tr>
<td>ut4</td>
<td>0.83</td>
<td>0.84</td>
</tr>
<tr>
<td>ut8</td>
<td>0.82</td>
<td>0.85</td>
</tr>
<tr>
<td>ut16</td>
<td>0.82</td>
<td>0.84</td>
</tr>
<tr>
<td>fc4</td>
<td>0.83</td>
<td>0.85</td>
</tr>
<tr>
<td>fc8</td>
<td>0.83</td>
<td>0.86</td>
</tr>
<tr>
<td>fc16</td>
<td>0.83</td>
<td>0.85</td>
</tr>
<tr>
<td>4proc</td>
<td>0.83</td>
<td>0.86</td>
</tr>
<tr>
<td>spawn</td>
<td>0.99</td>
<td>1.41</td>
</tr>
<tr>
<td>spawnC</td>
<td>0.72</td>
<td>0.82</td>
</tr>
<tr>
<td>inlet</td>
<td>0.49</td>
<td>0.53</td>
</tr>
</tbody>
</table>

N = 100 Sequential: 66.20(s) 13017096(b)