To Parallelise or To Optimise?

Jonathan Ezekiel* Gerald Lüttgen† Radu Siminiceanu‡

Abstract

Model checking is a popular and successful technique for verifying complex digital systems. Carrying this technique — and its underlying state-space generation algorithms — beyond its current limitations presents itself with a number of alternatives. Our focus is on parallelisation which is made attractive by the current trend in hardware architectures towards multi-core, multi-processor systems. The main obstacle in this endeavour is that, in particular, *symbolic* state-space generation algorithms are notoriously hard to parallelise.

In this article, we describe the process of taking a sequential symbolic state-space generation algorithm, namely a generic, symbolic BFS algorithm, through a sequence of optimisations that leads up to the *Saturation* algorithm, and follow the impact these sequential algorithms have on their parallel counterparts. In particular, we develop a parallel version of Saturation, discuss the challenges faced in its design, and conduct extensive experimental studies of its implementation. We employ rigorous analysis tools and techniques for measuring and evaluating parallel overheads and the quality of the parallelisation.

The outcome of these studies is that the performance of a parallel symbolic state-space generation algorithm is almost impossible to predict and highly dependent on the model to which it is applied. In most situations, perceivable speed-ups are hard to achieve, but real-world applications where our technique produces significant improvements do exist. Nevertheless, it appears that time is better invested in optimising sequential *symbolic* model checking algorithms rather than parallelising them.

Keywords. Symbolic state-space generation, Saturation, parallelisation, benchmarking.

1 Introduction

As highlighted by this year's choice of Turing Award recipients, automated verification — and in particular *model checking* [19, 46] — has been recognised as one of the most important contributions to Computer Science in recent years. Testimony to its success is the increasing number of hardware and software companies that now use model checking in their design cycle. Model checking's most notable challenge, the state-space explosion problem, has most successfully been tackled by the introduction of symbolic techniques, based on *Binary Decision Diagrams* (BDD) [7] and their variants, whose compact representation of massively complex systems has made model checking practical [41].

The challenge ahead is looking for the next major breakthrough to take model checking beyond its current limitations. The alternatives are seemingly diverging. Customised optimisations of sequential algorithms, such as chaining, partial order reduction, event locality and advanced data structures have already been successful in the past [20]. The incorporation of more powerful deductive techniques, borrowed from the world of constraint satisfaction and SMT solvers, is appealing but is meant primarily to extend the scope of model checking from strictly discrete-state systems to hybrid and infinite domains. Our focus, however, is on *parallelisation*. With the current trend in computer architecture that makes multi-processor, multi-core platforms readily

*jezekiel@doc.ic.ac.uk, Department of Computing, Imperial College, London, UK.

†**[Corresponding author]** gerald.luettgen@cs.york.ac.uk, Department of Computer Science, University of York, Heslington, York YO10 5DD, UK. Phone: +44 1904 434774; fax: +44 1904 432767.

‡radu@nianet.org, National Institute of Aerospace, Hampton, Virginia, USA.

available, it is natural to attempt an efficient implementation of model checking techniques on these platforms, with the manifested goal of achieving speed-ups.

Various efforts have been made to implement model checking algorithms on parallel computer platforms, most notably on Networks of Workstations (NOWs) and on PC clusters (see, e.g., [29, 30, 32, 42, 50]), where the primary goal is to utilise extra memory resources to represent state spaces. However, exploiting extra processors for the purpose of improving time-efficiency is not well researched for decision-diagram-based algorithms.

For our purposes, we have focused on what we believe represents the most advanced sequential technique: *Saturation* [16], a symbolic state-space generation algorithm with unique features. It mainly targets event-based asynchronous system models that are governed by interleaving semantics, such as models expressed by Petri nets, and exploits the local effect of firing events on state vectors by locally manipulating *Multi-valued Decision Diagrams* (MDDs) [37]. Saturation has proved to be orders of magnitude more time- and memory-efficient than other symbolic algorithms [14], such as those implemented in the NuSMV model checker [18]. This is because its search strategy is not based on variants of the traditional *Breadth First Search* (BFS) algorithms [41], including *Chained BFS* [48], but employs a novel, local search. Hence, the question arises as to whether event locality can also be utilised for parallelising Saturation in order to achieve further speed-ups. Previous approaches to parallelising Saturation have focused on data parallelism [9, 10], but not on parallelising the algorithm itself.

This article develops a parallel version of Saturation that processes events concurrently, with the goal of improving the algorithm's time-efficiency on multi-core PCs. On the one hand, we discuss the design decisions underlying *Parallel Saturation*, and present the various trade-offs which have been necessary in order to minimise the inevitable parallel overheads. On the other hand, we carefully evaluate Parallel Saturation against both its sequential version, as well as against Chained BFS and a straightforward parallelisation of a generic symbolic BFS algorithm. After initial setbacks in addressing various inefficiencies of Parallel Saturation, we have undertaken an effort to scrupulously analyse the sources of parallel overhead. The byproduct of this analysis, and an added contribution, is an overhead measurement and benchmarking approach.

Our approach provides an opportunity to determine whether a heavily optimised symbolic algorithm, such as Saturation, can be efficiently parallelised on shared-memory architectures. While Parallel Saturation shows good speed-ups on some real-world system models, such as NASA's Runway Safety Monitor [47], it frequently performs disappointingly against the original, sequential algorithm. This is because of the limited parallelisation opportunities inherent in many system models and the fact that these opportunities are impossible to quantify statically. The main conclusion of the research described in this article is that optimising sequential state-space generation algorithms, such as optimising BFS to yield Saturation by exploiting event locality and interleaving semantics, appears to be more worthwhile and beneficial than parallelising them, at least for decision-diagram-based algorithms. We believe that the insights gained by us provide valuable advice for colleagues working in the field of parallelising automated verification techniques.

The remainder of this article is organised as follows. Sec. 2 provides an introduction to the challenges of parallelising state-space generation algorithms and also includes some background on dealing with parallel overheads. Sec. 3 recalls the sequential symbolic algorithms of interest to us, from generic BFS, to Chained BFS, to Saturation, and comments on their parallelisation. Sec. 4 focuses on the most challenging part, namely parallelising Saturation. Our evaluation methodology and results are then documented in Sec. 5 and discussed in Sec. 6. Finally, we present related work and our conclusions in Secs. 7 and 8, respectively.

2 The Challenges of Parallelisation

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* parallelises 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 Saturation 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 from where the inefficiencies were arising was a painstakingly difficult task, since there are a number of causes of inefficiency and the nondeterminism of a parallel algorithm makes them hard to determine.

```
Reachability(in initstate : state)
  declare nextstate : state;
 1. Add(set, initstate);
 2. do
 3.   nextstate ⇐ Remove(set);
 4.   Enumerate(nextstate);
 5. while set ≠ empty
```

```
Enumerate(in enumstate : state)
  declare succstate : state;
  declare L : set of states;
 1. L ⇐ Successors(enumstate);
 2. while L ≠ ∅ do
 3.   succstate ⇐ pickfromset(L);
 4.   if IsNew(statestore, succstate) then
 5.     AddState(statestore, succstate);
 6.     Add(set, succstate);
```

set : a set that a state can be added to or removed from using Add and Remove
statestore : a store of states that a state can be checked against using IsNew and added to using AddState
Successors : generates a list of successor states from a set of transitions
threads : number of threads, for simplicity we assume that access to this is atomic
Lock, UnLock : lock and unlock the data structures for atomic access

```
ParallelReachability(in initstate : state)
  declare nextstate : state;
 1. Add(set, initstate);
 2. threads ⇐ 1;
 3. do
 4.   if set ≠ empty
 5.     Lock(set);
 6.     nextstate ⇐ Remove(set);
 7.     UnLock(set);
 8.     threads ⇐ threads + 1;
 9.     parallel ParallelEnumerate(nextstate);
10. while threads ≠ 1 and set ≠ empty
```

```
ParallelEnumerate(in enumstate : state)
  declare succstate : state;
  declare L : set of states;
 1. L ⇐ Successors(enumstate);
 2. while L ≠ ∅ do
 3.   succstate ⇐ pickfromset(L);
 4.   Lock(statestore);
 5.   if IsNew(statestore, succstate) then
 6.     AddState(statestore, succstate);
 7.     Lock(set);
 8.     Add(set, succstate);
 9.     UnLock(set);
10.   UnLock(statestore);
11. threads ⇐ threads - 1;
```

Figure 1: Sequential and parallel reachability algorithm.

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 initial state *initstate* and transition relation (next-state function) *Successors*, is shown on the left of Fig. 1. It is a generic work-set algorithm and does not specify a particular search strategy (such as breadth-first or depth-first). The *Reachability* function begins with an initial state and calls *Enumerate* to enumerate its successor states. *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 adds it to a set of unexplored states (implemented as a stack for DFS, or a queue for BFS). *Reachability* takes a new unexplored state off the set and repeatedly calls *Enumerate* on that state until there are no more unexplored states, at which point the entire state space is explored.

The obvious starting point for a parallelisation is the *Enumerate* function, allowing states to be enumerated concurrently. 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 Fig. 1. *ParallelReachability* performs the same task as the sequential *Reachability* function; however, it also keeps track of the number of threads using the counter *threads* in order to detect termination. Termination occurs when there are no threads enumerating states and no states to explore in the work set. Note that the *Remove* operation on line 5 is always performed on a non-empty set: *initstate* is intially in the set and the while loop is executed with *set* $\neq$ *empty* as an invariant. The function must also ensure that, when it takes a state from the set, access to the set is atomic; this is to prevent *data races* from other threads inserting states into the set at the same time. To achieve this, *ParallelReachability* locks

the work set when accessing it and unlocks it when the state has been taken from the set. When there is a state to explore, *ParallelReachability* creates a new thread to run the *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 work set, 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.

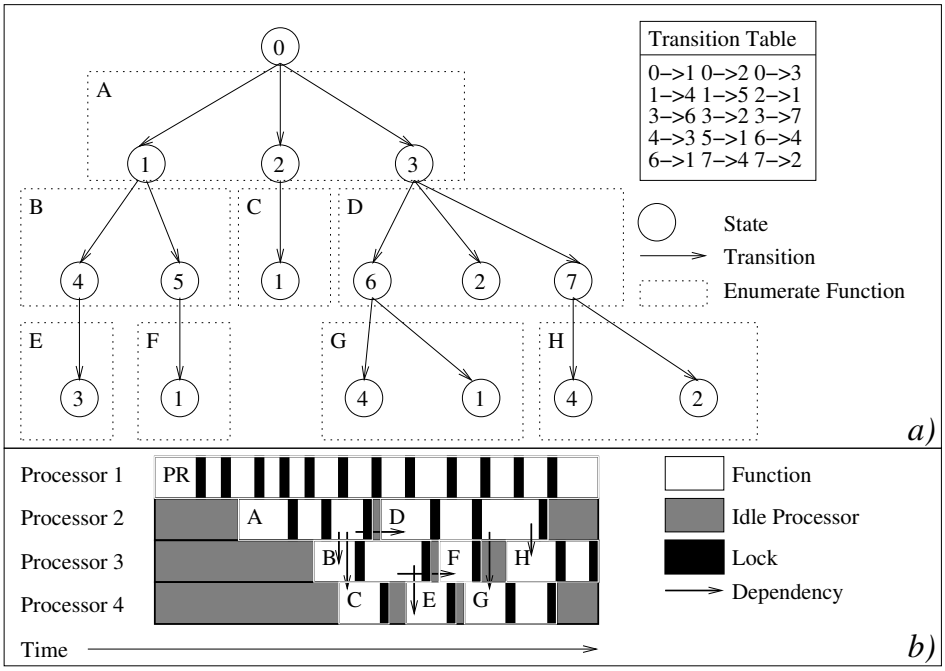

Figure 2: Parallel state-space enumeration and processor allocation of functions.

2.2 What are the Overheads?

If we were to run the above parallel reachability algorithm to explore a state space, we would notice a significantly negative effect on run-time when compared to the sequential algorithm. By just looking at the run-times, it would not be obvious why this occurs, and analysing the underlying reasons for the negative effect can be a daunting task. We can explain where the negative impact comes from using Fig. 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.

Fig. 2(a) shows the parallel state-space construction for the model defined by the transition table (next-state function) and initial state 0; the displayed tree should be read as a search tree through the model's state space and *not* as the state space itself. While this example deals with explicit state-space generation, the tree's nodes could equally well be read as units of work conducted within a symbolic approach. The work of the *ParallelEnumerate* function is visualised using a broken box, which includes the enumeration of duplicate nodes. Fig. 2(b) depicts how the resulting *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 *ParallelReachability* function which is constantly checking the work set and scheduling new *ParallelEnumerate* functions when a new state has been discovered.

¿From the processor utilisation of Fig. 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 *irregular problem* in the parallel algorithms community, which are known to be difficult to parallelise [26, 40, 45, 52]. 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, 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. 90,000ns on a modern PC running Linux) or allocating a task to an existing thread in a thread pool (approx. 8000ns) can be higher than the work performed by a small computation (approx. 1,200ns) [23]. Thirdly, frequent locking results in high *synchronisation overhead*, i.e., the time taken to lock and unlock a data structure, which translates to *communication overhead* on a PC cluster where processes must frequently synchronise with each other.

2.3 Implementing a Parallel State-space Algorithm

One of the great difficulties of parallelising state-space exploration algorithms is understanding how to efficiently implement a devised algorithm. The types of languages and libraries chosen for parallelisation can have a significant effect on the performance of the algorithm, since their characteristics can influence overheads. For example, the use of Java leads to high synchronisation overheads for memory allocation and garbage collection, when compared to C [33].

A particular consideration for languages is the availability of parallel tool support for profiling, scheduling work, and 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 recommend C as the language of choice when parallelising for shared-memory architectures.

Another question that arises is whether to use native thread libraries or 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 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 [8] more suitable than OpenMP for parallelising our algorithm.

Selecting an appropriate architecture is also an important decision. Availability is a key issue when choosing hardware. Multi-processor, multi-core PCs are becoming widely popular 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 [31]. 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.

2.4 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* [1, 5] techniques. These have been used in parallel state-space exploration algorithms to facilitate orders of magnitude improvements in time-efficiency [29, 33]. Work-stealing 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, 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. These approaches are key to the order of magnitude speed-ups attained from parallel state-space exploration algorithms in [29, 33]. 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 [33].

3 Symbolic State-space Generation

We begin our investigation into parallelising symbolic state-space generators for event-based asynchronous systems by recalling popular sequential algorithms, from the generic *Breadth First Search* (BFS) [41], to the somewhat optimised *Chained BFS* [48], to the highly optimised *Saturation* [16] algorithm. For the former two, simpler algorithms, we also discuss their parallelisation, which is based on the idea of concurrent event processing. Doing so provides us with a stepping stone towards understanding how best to parallelise the more complex Saturation algorithm, which is the topic of the next section. We defer a detailed evaluation of the algorithms, all of which are implemented in the verification tool SMART [12], to Sec. 5.

3.1 Setting and Notation

The object of our symbolic state-space generators are event-based discrete-state systems. In our setup, a discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}^0, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}^0 \in \widehat{\mathcal{S}}$ is the *initial state*, and $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in one step. Sometimes we will consider a set of initial states rather than a single initial state. Assuming that the model contains K *submodels*, a (*global*) state $\mathbf{i}$ 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 $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ is the cross-product of K *local state spaces*. This allows us to use *symbolic* techniques based on decision diagrams to store sets of states [7, 41]. We decompose $\mathcal{N}$ into a disjunction of next-state functions, so that $\mathcal{N}(\mathbf{i}) = \bigcup_{e \in \mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function for event e. We seek to build the *reachable state space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, the smallest set containing $\mathbf{s}^0$ and closed wrt. $\mathcal{N}$: $\mathcal{S} = \{\mathbf{s}^0\} \cup \mathcal{N}(\mathbf{s}^0) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^0)) \cup \cdots = \mathcal{N}^*(\mathbf{s}^0)$, where "*" denotes reflexive and transitive closure and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$.

In the sequel, we assume that each local state space $\mathcal{S}_k$ is finite and known a priori. In practise, $\mathcal{S}_k$ can actually be generated on-the-fly by interleaving symbolic global state-space generation with explicit local state-space generation [17]. Without loss of generality, we assume that $\mathcal{S}_k = \{0, 1, \ldots, n_k - 1\}$, where $n_k = |\mathcal{S}_k|$. We then encode any set $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ as a *(quasi-reduced, ordered) Multi-valued Decision Diagram* (MDD) [37] over $\widehat{\mathcal{S}}$. Formally, an MDD is a directed acyclic edge-labelled multi-graph where:

- Each node p belongs to a *level* $k \in \{K, ..., 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node r at level K.
- Level 0 can only contain the two *terminal* nodes $\mathbf{0}$ and $\mathbf{1}$.
- A node p at level $k > 0$ has n_k outgoing edges, labelled from 0 to $n_k - 1$. The edge labelled i points to a node q at level $k-1$; we write $p[i] = q$.
- Given nodes p and q at level k, if $p[i] = q[i]$ for all $i \in \mathcal{S}_k$, then $p = q$, i.e., there are no *duplicate* nodes.

The set encoded by an MDD node p at level $k > 0$ is $\mathcal{B}(p) = \bigcup_{i \in \mathcal{S}_k} \{i\} \times \mathcal{B}(p[i])$, letting $\mathcal{X} \times \mathcal{B}(\mathbf{0}) = \emptyset$ and $\mathcal{X} \times \mathcal{B}(\mathbf{1}) = \mathcal{X}$ for any set $\mathcal{X}$. MDDs are implemented using K *unique tables* (i.e., dynamically-sized hash tables).

For storing $\mathcal{N}$, we adopt a representation inspired by work on Markov chains. This requires the model to be *Kronecker consistent* [16], a restriction that can often be automatically satisfied by concurrency models such as Petri nets. 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 $(i_K, \ldots, i_1) \in \widehat{\mathcal{S}}$. Using $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0,1\}^{n_k \times n_k}$ such that $\mathbf{N}_{k,e}[i_k, j_k] = 1$ iff $j_k \in \mathcal{N}_{k,e}(i_k)$, we encode $\mathcal{N}_e$ as a boolean Kronecker product: $\mathbf{j} \in \mathcal{N}_e(\mathbf{i})$ iff $\bigotimes_{K \geq k \geq 1} \mathbf{N}_{k,e}[i_k, j_k] = 1$, where $\otimes$ indicates the Kronecker product of matrices. The $\mathbf{N}_{k,e}$ matrices are extremely sparse; when encoding a Petri net, for example, each row contains at most one non-zero entry.

3.2 Breadth First Search

We first discuss a well-known, generic BFS algorithm whose pseudo code is shown in Fig. 3. We introduce some new notation in the code, *idx* indicates a unique identifier for an MDD node, $\mathcal{E}^K$ is the set of events for level K and *lcl* indicates a local state. The algorithm 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 to 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 new nodes that have been created as a result of the firings.

The *BFSFire* function is similar to the *RecFire* function in Saturation (shown below), 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. A *firing cache*, $FC[k]$ for each level k, stores the results of firings, similar to the standard cache for union operations on MDDs. The uniqueness on MDD nodes is checked by employing a *unique table*, $UT[k]$, for each level k.

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.

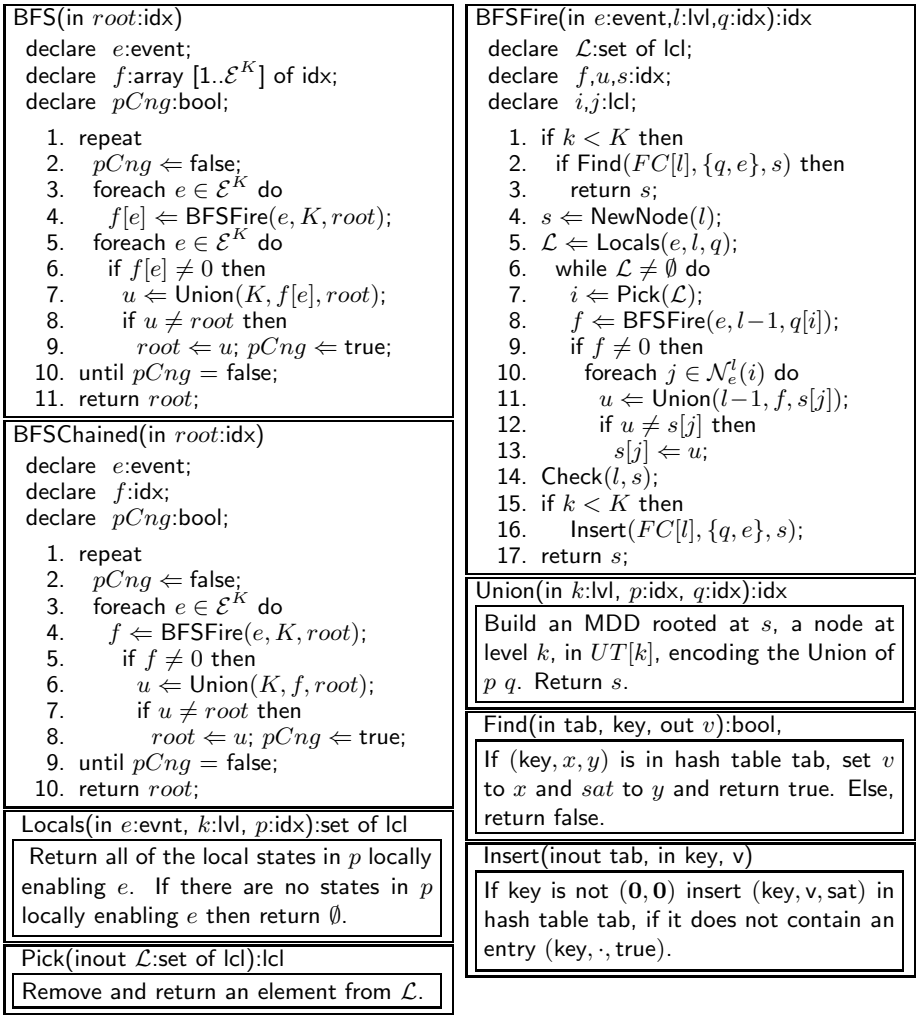

Figure 3: Pseudo code for the BFS and Chained BFS algorithms.

3.3 Chained Breadth First Search

At first glance, 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. In contrast, in a Chained BFS [48], firings are not performed in an independent manner, but instead utilise the result of a previous firing.

The Chained BFS algorithm, also depicted in Fig. 3, is therefore much more difficult to parallelise, since there is no clear loop to exploit for parallelism. Each time an event is fired using *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. We investigate this within the evaluation section (Sec. 5).

3.4 Saturation

The *Saturation* algorithm [16] is radically different from the traditional symbolic reachability algorithms, such as BFS, in that it does not perform a standard fixed-point computation. The encoding of the next-state function is not monolithic, hence a more flexible strategy can be used, consisting of many "lightweight" nested fixed-point iterations, rather than a single "heavyweight" global fixed-point iteration.

The *saturation* search has at its core the idea of exploiting *event locality*. The choice of representing the next-state function $\mathcal{N}$ as a Kronecker product allows us to recognise event locality, as captured by the notion of independence of events on subsystems. We say that event e is *independent* of level k if $\mathbf{N}_{k,e} = \mathbf{I}$, the identity matrix. Let $Top(e)$ denote the highest level[1] for which $\mathbf{N}_{k,e} \neq \mathbf{I}$. An MDD node p at level k is *saturated* if it is a fixed point wrt. all $\mathcal{N}_e$ such that $Top(e) \leq k$, i.e., $\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p) = \mathcal{N}_{\leq k}(\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:\, Top(e) \leq k} \mathcal{N}_e$. To saturate MDD node p once all its descendants are saturated, we *update it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for all events e such that $Top(e) = k$. This can create new MDD nodes at levels below k, which are saturated immediately, prior to completing the saturation of p. If we start with the MDD encoding the initial state $\mathbf{s}^0$ and saturate its nodes bottom up, the root r will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^0)$ at the end, as shown in [16].

The algorithm contains two main mutually recursive functions: *Saturate* calls *FireEvents* to recursively perform the event firings while saturating nodes, while *FireEvents* indirectly calls *Saturate* to saturate nodes that are created as a result of event firings. The greedy strategy of saturating every node immediately upon its creation (by pre-empting the undergoing event firing operation) results in a non-trivial series of recursive, preemptive firings. The intuition behind using this strategy is that *only* saturated nodes can be part of the state-space representation, while non-saturated nodes are guaranteed not to. Also, once a node is saturated, it does not need to be considered for further exploration. The bottom-up order of saturating nodes ensures that all descendants are already saturated when a node is considered for saturation. All these effects combine into a dramatic reduction of the peak MDD size. Figure 4 attmepts to illustrate a "typical" order of saturating nodes in a fictitious model with 5 MDD levels, assuming (for simplicity) that only one additional new node is generate by each event firing on the level below.

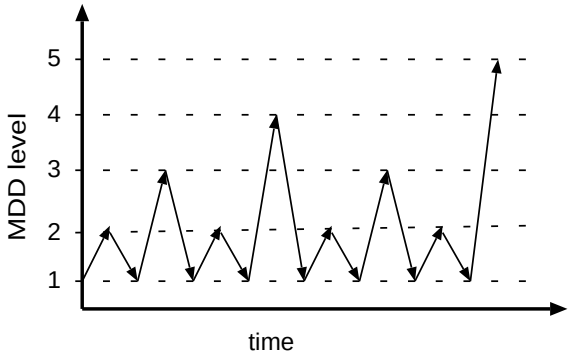

Figure 4: A "typical" order of saturating nodes, by MDD level

The pseudo code for the sequential Saturation algorithm is displayed in Fig. 6, when leaving out the shaded lines, and is explained in detail in [16].

Experimental results reported in [11, 16, 17, **?**] show that Saturation consistently 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. However, the optimal and irregular nature of the algorithm makes it difficult to parallelise [22, 23].

[1] Such level always exists for an event, except for the degenerate case when the event does not affect any level.

4 Parallel Saturation

The goal of parallelising Saturation is to exploit event locality to create independent parallel tasks for firing events. There are several challenges to overcome and important design decisions to make.

4.1 Design Decisions and Justifications

While one can exploit event locality to create independent tasks for firing events, the firings of single events often cannot be sensibly parallelised for efficiency reasons. This is because of the high costs of creating threads and allocating tasks to an existing thread (cf. Sec. 3), and the order-of-magnitude lower cost of an in-place update. These in-place updates occur relatively frequently, and also when an event firing fully utilises previous work that has been cached. 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 their scheduling.

To address the irregularity underlying any decision-diagram-based algorithm, we use a *thread pool* that runs as many threads in parallel as processors or processing cores are available on the shared-memory architecture under consideration. These threads pick work that needs to be performed from a *task queue*. The choice of thread pool is justified by the order of magnitude saving in scheduling cost compared to creating operating-system threads for each task, and in particular the *control* over which the tasks can be selected from the shared queue. A more complex work-stealing system, for example, would require multiple queues from which tasks can be stolen. While the use of multiple private queues per thread may reduce synchronisation cost on the queues, it is unknown what sort of memory impact the resulting loss of control over the order of the construction of the state space will have. This is particularly true for a symbolic environment where memory is a concern, and is justified by the effects observed from Chained 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, rather than developing a more complex scheduling approach that may impact on the chaining mechanism.

Fitting the Saturation algorithm into this load balancing structure, however, is difficult due to Saturation's mutually recursive nature. In particular, 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 an MDD node. The cost of breaking recursion this way is 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 cache work requests to avoid duplicate work in parallel.

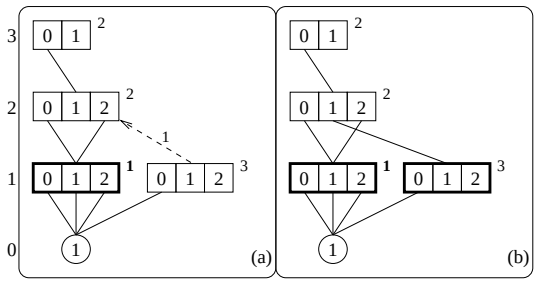

Figure 5: Using an upward arc to replace a function call.

We illustrate the introduction of upwards arcs into the MDD structure in Fig. 5, where the levels of the MDDs are displayed to the left of the diagram, node indexes are written 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. Fig. 5(a) depicts 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, meaning that state 1 of node 2.2 will point to node 3.1 if node 3.1 becomes saturated. In Fig. 5(b), 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 implies that the recursive firing function that created node 3.1 does *not* have to wait for node 3.1 to become saturated in order to update node 2.2. Instead, 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.

4.2 The Thread Pool Algorithm

The result of mapping our design decisions into code is shown in Fig. 6, with supporting functions being described in Fig. 7. The code extends the sequential Saturation algorithm, using a recursive event ordering. Parallel code is highlighted in dark-shaded code which facilitates tasks and removes mutual recursion, while light-shaded code shows locks for ensuring correct synchronisation. We use the notation introduced in Sec. 3.4, and extend it to include information and locks on MDD nodes p, which are denoted by p.information and p.lock, respectively.

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*, or is 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, or a set of initial states, of the underlying system model. It also allocates 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. The task is complete once it has fired the events, and it then calls *RemoveOp*. The *Saturate* function allows the thread to check the status of the node in order 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 node invokes 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 when 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, and 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 (i.e., root node).

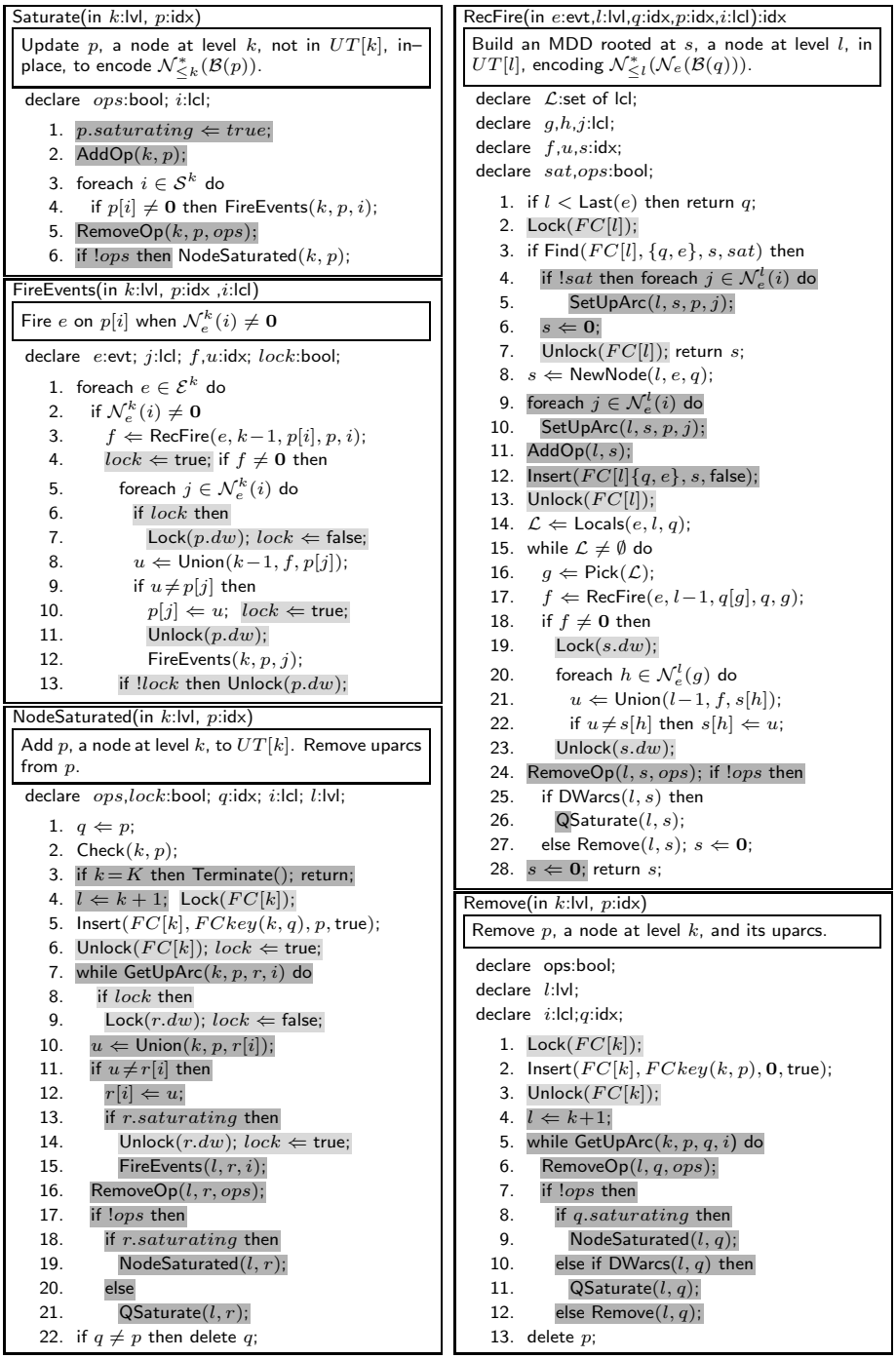

Figure 6: Pseudo code for the Parallel Saturation algorithm.

Gen(in s:array[$1..K$] of lcl, $nthr$:int):idx
Create $nthr$ threads. Build an MDD rooted at $root$, at level K, encoding the state space and return $root$, in $UT[K]$.

DWarcs(in k:lvl, p:idx):bool
If $p[i] \neq \mathbf{0}$ for any local state at level k return true otherwise return false;

SetUpArc(in k:lvl, p:idx, q:idx, i:lcl)
Lock($p.ua$). Add an arc (q,i) to the end of the list of upward arcs for p; AddOp($k+1, q$); Unlock($p.ua$);

GetUpArc(k:lvl,p:idx,out q:idx,i:lcl):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:lvl, p:idx)
Lock($p.ops$). Increment $p.ops$. Unlock($p.ops$).

RemoveOp(in k:lvl, p:idx, out op:bool)
Lock($p.ops$). Decrement $p.ops$. If $p.ops = 0$ set op to false otherwise set op to true. Unlock($p.ops$).

FCkey(in k:lvl, p:idx):key
Return the key for p, a node at level k.

NewNode(in k:lvl,e:evt,q:idx):idx
Create p at level k, with arcs set to $\mathbf{0}$, set the key (e,q) for p, return p.

Check(in k:lvl, inout p:idx)
Lock($UT[k]$) If p, not in $UT[k]$, duplicates q, in $UT[k]$, delete p and set p to q. Else, insert p in $UT[k]$. Unlock($UT[k]$) If $p[0] = \cdots = p[n^k - 1] = \mathbf{0}$ or $\mathbf{1}$, delete p and set p to $\mathbf{0}$ or $\mathbf{1}$.

QSaturate(in k:lvl, p:idx)
Lock the task queue. Add item (k, p) to the task queue. Unlock the task queue. Request any sleeping threads wake up.

Terminate()
Lock the task queue. Add item $(\mathbf{0}, \mathbf{0})$ to the task queue. Unlock the task queue. Request any sleeping threads wake up.

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

Figure 7: Supporting functions for the Parallel Saturation algorithm.

Table 1: Next-state function for the Parallel Saturation example

	e_1	e_2	e_3	e_4
3	*	*	$0 \rightarrow 1$	$0 \rightarrow 2$
2	*	$0 \rightarrow 2$	$0 \rightarrow 2$, $2 \rightarrow 1$	*
1	$0 \rightarrow 2$	*	$0 \rightarrow 1$	*

4.3 Example

As Parallel Saturation is a nontrivial algorithm, we illustrate it by means of a small example, utilising a thread pool of two threads and operating on a discrete-state model that is defined by the next-state function of Table 1. In Figs. 8 and 9, 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 *op* operating on the node are shown to the bottom right of the node in brackets. The firing cache key *fc* is annotated in curly brackets next to *op*. The status of the queue and threads is displayed at the top right of the diagram, and the levels of the MDD are listed to the left of the diagram.

We now narrate through the execution of Parallel Saturation on this model, focusing on the selected snapshots of Figs. 8 and 9. The initial state set is $\{(0,0,0), (0,1,1)\}$.

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. The operations (*op*) field 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 that these nodes are currently being operated upon by the *Saturate* task.

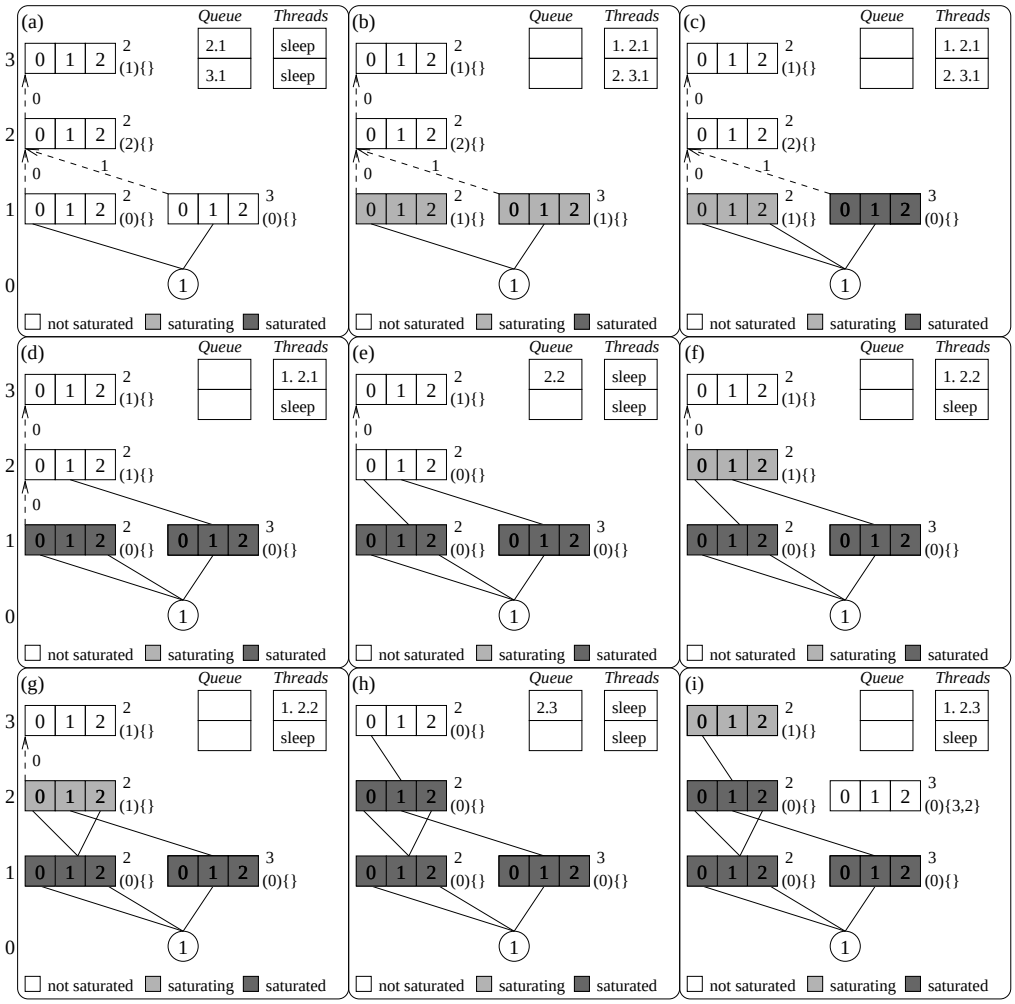

Figure 8: Parallel Saturation example.

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 non-zero, *NodeSaturated* terminates, leaving thread 2 to sleep.

e) *NodeSaturated(1,2)* removes the upward arc to node 2.2, 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 the task 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 upward 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 2.3, 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, while firing e_3, has discovered node 4.1 as an *unsaturated* node in the firing cache, which sets an upward arc to node 3.2 and increments *op*.

n) Node 4.1 has completed saturating, 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. *NodeSaturated* has been called on node 4.1 which has discovered, while checking it into the *unique table*, that the node is the same as node 3.1, and has removed the upward arcs, setting the downward arcs to this node. Since *op* is 0, a *Saturate* task is added to the queue for node 3.2, and the thread goes to sleep.

p) Thread 1 takes the task for node 3.2, setting the thread to *saturating* and incrementing *op*.

q) *Saturate* completes on node 3.2, decrementing *op* to zero and calling *NodeSaturated* which replaces the upward arc to node 2.3 with a downward arc and decrements *op*. Since *op* is 0 and the node is *saturating*, node 2.3 is now *Saturated*. Moreover, because this is the root node, *Terminate* is called which instructs the task queue to terminate the threads. The final MDD representing the state space $\{(0,0,0), (0,0,2), (0,1,1), (0,2,0), (0,2,2), (1,1,1), (1,2,1), (2,0,0), (2,0,2)\,(2,1,1), (2,2,0), (2,2,2)\}$ that is reachable from the initial state set $\{(0,0,0), (0,1,1)\}$ is shown.

4.4 Correctness of the Algorithm

The algorithm in Fig. 6 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 [16]. 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 Fig. 10(a). The call graphs in Figs. 10(b) and 10(c) are the calling order of functions for the sequential and parallel code, respectively, where Figs. 10(d) and 10(e) 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 a node is atomic, firing events exhaustively will result in the same MDD shape for a saturated node as in the sequential version.

5 Evaluation

During our study of parallelising Saturation, we have spent much time investigating how to evaluate performance according to the impact of parallel overheads that arise 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 that are reported in the

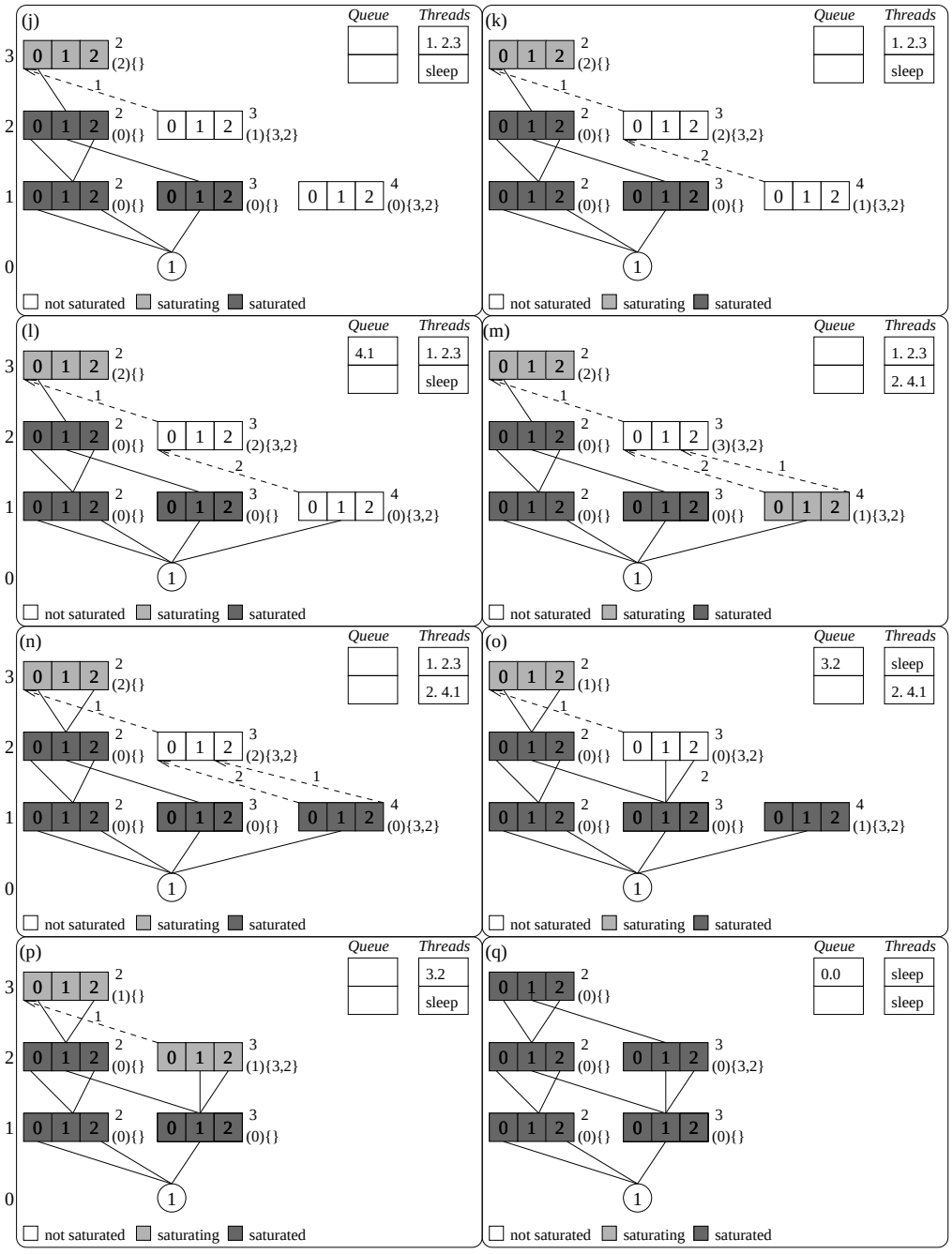

Figure 9: Parallel Saturation example (continued).

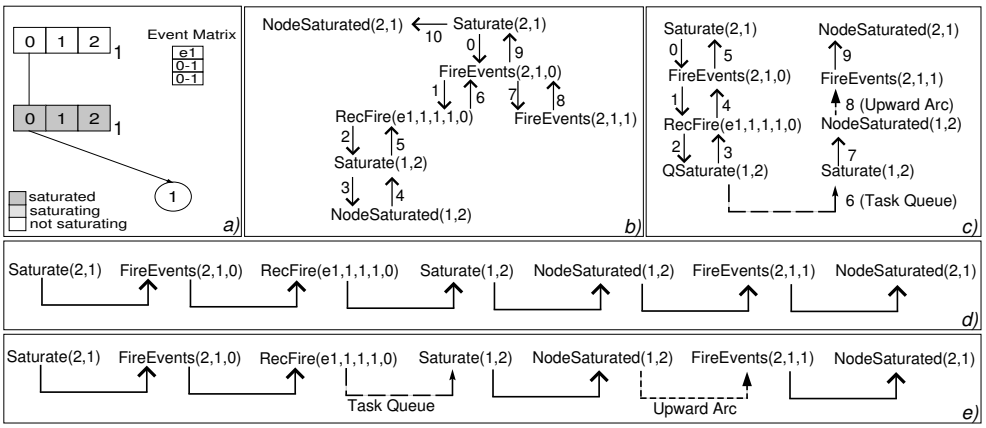

Figure 10: The calling order of functions for Sequential and Parallel Saturation.

literature, benchmark the run-time of an algorithm on a small number of models and include rather incomplete estimations of parallel overheads. In this section, we begin by showing how we have strengthened our evaluation by accurate and thorough direct measurement of parallel overheads and by carefully selecting models that illustrate the overall performance of our parallel state-space exploration algorithms. With these new techniques and benchmark in hand, we evaluate Parallel BFS and Parallel Saturation against their sequential counterparts.

5.1 Benchmarking Parallel State-space Algorithms

Any benchmark must be able to stress test a parallel algorithm against the various parallel overheads. Hence, the quality of a benchmark depends on the ability to measure parallel overheads accurately, which is a key issue that has yet to be addressed in the literature. The overheads of the algorithm are usually measured by 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 [23]. 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.

5.1.1 Measuring Parallel Overheads

We first address the problem of measuring parallel overheads for each type of overhead in turn:

- Load balancing is difficult to measure: if we would count the number of states, or nodes, enumerated on a processor and discover that this number is fairly evenly distributed among 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.

- Scheduling overheads cause similar problems for measuring. 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. However, this would be an inaccurate measure since, particularly in cases when the operating system decides how work is scheduled, 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 to the algorithm. Others have found similar problems with estimating synchronisation overheads [33].

The only solution to accurately measuring these overheads is via *direct* measurement. But how does one measure the activity on each of the processors? Even finding a timer that is fine-granular enough to measure the time spent acquiring a mutex lock poses a problem. We tried using the *gprof* profiler [28], but it was developed to profile sequential algorithms and could not show the individual activities 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.

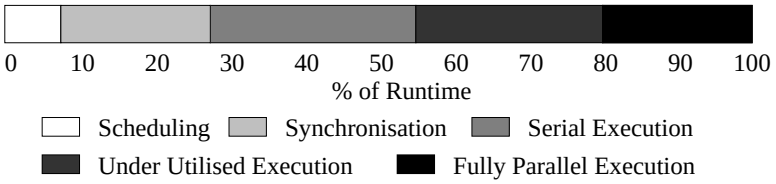

Figure 11: Parallel algorithm profile obtained from the Intel thread profiler.

Most suitable to our requirements for profiling on shared-memory architectures is Intel's *Thread Profiler* [`www.intel.com/software/products/threading/`], due to its ability to analyse applications using *Pthreads* [8] and provide a breakdown of parallel overheads. In order to evaluate our parallel algorithms, we applied the profiler at run-time. We chose our experimental architecture according to the constraints on the types of processors and operating systems that the profiler supports, i.e., Intel processors, certain flavours of Linux, C and C++, and particular compilers. Fig. 11 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 an algorithm's parallel overheads.

5.1.2 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 affect of the model on the severity of the overheads. We found this out the difficult way in Parallel Saturation, since some models showed good performance and others showed a lack of parallelisability [23]. At first glance, these results 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 related 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.

Fig. 12 shows the way in which the work units of our parallel reachability algorithm in Fig. 1 are broken down during reachability analysis, when different model characteristics are considered. We highlight the work in the same way as in Fig. 2, where the units of work are defined functionally and the dependencies between them are decided by the way in which the functions enumerate nodes of the search tree. We use pathological examples to illustrate particular overheads that may arise during the construction of the state space for three stereotypical types of model. Fig. 12(a) underlies a model where the work cannot be spread across processors, since the work units are essentially sequentialised due to the dependencies between them. Fig. 12(b) underlies a model that imposes high scheduling overheads, due to the small size of the work units. Fig. 12(c) underlies an

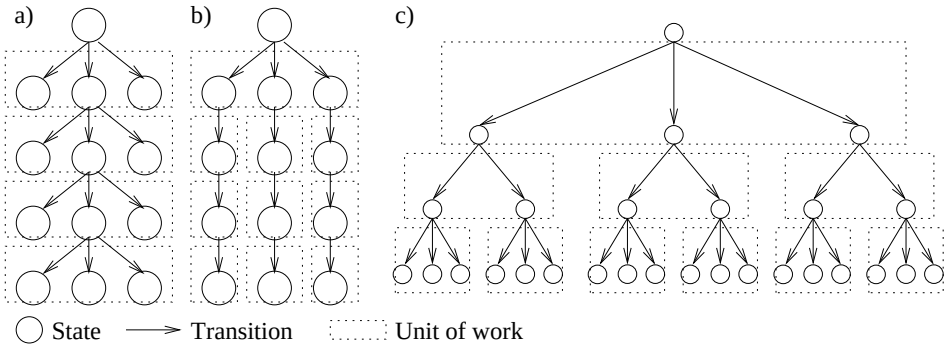

Figure 12: Search trees constructed during reachability analyses for three pathological models.

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 algorithms exhibit a combination of these factors, with performance varying from super-linear speed-ups to slow-downs of over 60%. Other parallelisationss of state-space exploration algorithms also show widely varying performance of the algorithm according to the model's characteristics [29]. 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.

5.1.3 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 Fig. 12. This allows a way in which each particular type of overhead can be thoroughly evaluated and (in)efficiencies in the algorithm can be pinpointed.

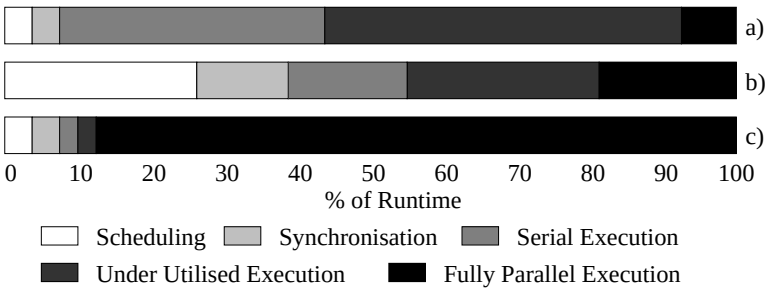

Figure 13: Profiles of our three stereotypical models.

Fig. 13 shows the profiles of our three stereotype models: (a) has low parallelism, (b) has high scheduling, and (c) is an *ideal* model that can be parallelised well with little 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 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 an algorithm's performance to be truly put into context.

5.2 Our Benchmark

The benchmark we selected for our algorithms contains ten models that have been previously used to evaluate Saturation [11, 16, 17]. These are given as Petri nets, which is the specification language adopted by the verification tool SMART [12] in which are algorithms are implemented.

5.2.1 The Benchmark Models

All ten models are *parameterised*, i.e., a parameter can be set for each model, typically N, which alters the size of its state space. Each model is naturally partitioned into K components, according to its subsystems.

Slot. The *slotted ring network protocol* [44] 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 stations in the ring, and $K = N$.

Robin. The *round robin mutual exclusion protocol* [27] 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, i.e., $K = N$.

Kanban. The *Kanban manufacturing system* [51] contains 16 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, and K is the number of stations.

FMS. The *flexible manufacturing system* [43] 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. The parameter N indicates the initial number of each type of parts, and $K = 19$.

Queens. The classic *queens problem* 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. For this model, $K = N$.

Leader. The *randomised leader election protocol* [36] 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 $K = 11N$.

BQ. The *bounded open queuing network* (**BQ**) [24] 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 $K = 8$.

Philosophers. The *dining philosophers* protocol [15] refers to the classic mutual exclusion problem. Parameter N indicates the number of philosophers, and $K = N/2$.

RSM. The *runway safety monitor* [47] was developed by NASA and Lockheed Martin, as a protocol to detect runway safety incidents. The RSM protocol defines *targets* T (either airborne or on the ground), 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 $N = X \cdot Y \cdot Z$, and $K = 4T$.

Aloha. The *Aloha network protocol* [13] 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, the parameter N represents the number of nodes in the network, and $K = N + 3$.

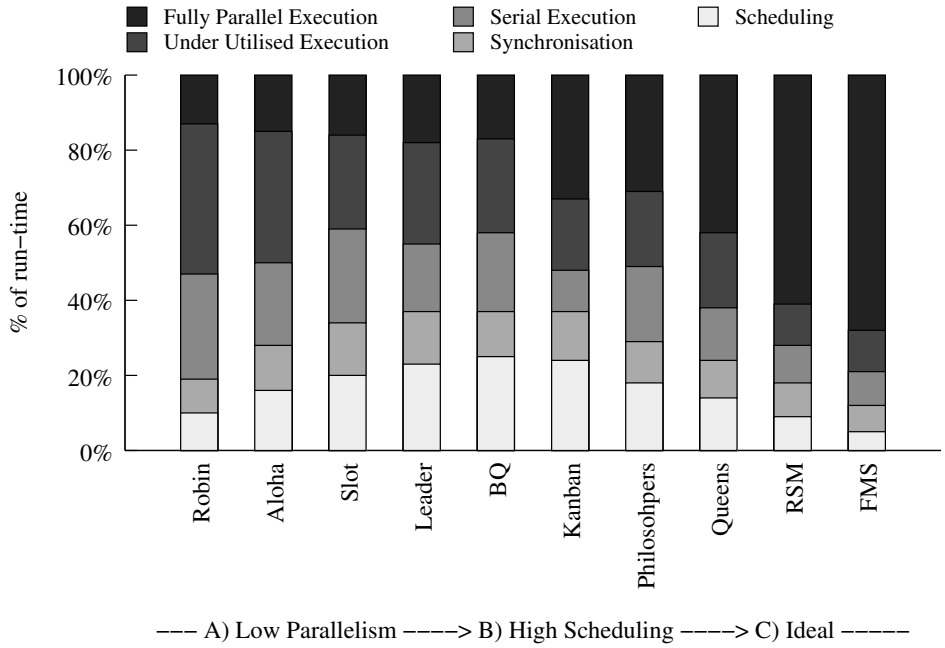

Figure 14: Profiles of the Parallel Saturation algorithm for large models.

5.2.2 Benchmark Justification

Our experiments are put into context by the profiles in Fig. 14. These profiles were generated using the Intel thread profiler and our Parallel Saturation algorithm in Sec. 4, for the largest parameter of each model to which we applied the algorithm. Fig. 14 shows that our benchmark includes a good mix of models that fit into the stereotypical categories of Fig. 13, as well as models that fall between these categories by exhibiting characteristics of a combination of overhead types. The profiles retain their shapes when different sizes of parameters are applied to the models in the benchmark; there are only small differences in the overheads: typically 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 of Fig. 13(a): cores are under-utilised which demonstrates a lack of parallelism. The BQ profile roughly fits the high-scheduling model profile of Fig. 13(b): scheduling overhead is relatively high compared to the other models. The FMS profile roughly fits the ideal model profile of Fig. 13(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 exhibits relatively high parallelism. These profiles are invaluable when drawing conclusions as to the performance of our parallel algorithms, and in particular Parallel Saturation. Without them, one would not be able to understand the circumstances under which Parallel Saturation is able to improve over Sequential Saturation.

5.3 Run-time and Memory Results

We have implemented our parallel algorithms using C and the POSIX Pthreads library [8]. 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. Each experiment has been performed three times, and the average run-time and memory performance numbers for each run are presented here. Run-time and memory variance has been consistently low, in the range of $\pm 5\%$.

5.3.1 BFS Chained vs. BFS vs. BFS Parallel

We first evaluate three BFS-based algorithms: generic BFS, Chained BFS and Parallel BFS, as discussed in Sec. 3. Each algorithm is applied to our benchmark models, and the resulting run-time and memory performances are shown in Table 2. The run-time and memory for the Chained BFS are given, as well as the relative run-time (in seconds) and memory (in bytes) increases for the BFS and Parallel BFS algorithms against Chained BFS. A relative performance factor of greater than 1 indicates a run-time speed-up or memory increase, while a factor of less than 1 indicates a run-time slow-down or memory decrease. A model that exhausted the available memory is indicated by "*N/A*."

Compared to Chained BFS, the generic BFS algorithm (BFS) is slower and uses more memory for all models. The use of extra resources is significant in several models, such as Slot which is over 50 times slower in terms of run-time and uses over 9 times more memory. It would therefore be impossible for Parallel BFS to catch up with the run-time of Chained BFS on our experimental machine for a model such as Slot, 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 BFS.

In contrast, Parallel BFS only improves the run-time of BFS for three of the models. These improvements are relatively small compared to Chained BFS, with the largest improvement coming from the FMS model of approximately 20%. Even if Parallel BFS is able to improve the run-time of BFS, it is still at least twice slower than Chained BFS. Parallel BFS also requires extra memory for the construction of extra MDD nodes in parallel, since more MDD nodes will be constructed

Table 2: Run-time and memory results for the BFS-based algorithms

	BFSChained		**BFS**		**BFS Parallel**	
N	**Time(s)**	**Mem(b)**	**Time(s)**	**RMem**	**Time(s)**	**RMem**
Slot.						
10	0.61	3576400	0.09	8.67	0.05	16.65
20	18.52	78085280	0.02	9.18	0.01	18.98
30	293.78	536934720	N/A	N/A	N/A	N/A
Robin.						
20	0.23	1241644	0.02	16.53	0.01	18.99
30	0.64	3175464	0.01	21.56	0.01	28.97
40	1.44	6304884	0.01	24.89	0.01	29.3
Kanban.						
10	0.69	5084224	0.14	4.94	0.14	5.95
15	2.31	17128944	0.11	5.22	0.1	7.05
20	6.06	42982264	0.07	5.64	0.07	7.96
FMS.						
7	3.64	58359236	0.26	3.67	0.3	4.4
8	8.61	135106208	0.27	3.83	0.32	4.56
9	17.35	288403644	0.04	3.88	0.05	4.81
Queens.						
8	0.08	190512	0.89	1.27	1	1.57
9	0.46	632160	0.96	1.3	0.96	1.64
10	5.99	2741740	0.85	1.41	0.84	1.66
Leader.						
4	0.55	3046768	0.22	2.78	0.26	2.84
5	4.61	18491440	0.15	2.95	0.17	3.02
6	52.55	91660264	0.1	3.07	0.11	3.21
BQ.						
20	0.18	1246424	0.42	1.86	0.39	2.57
30	0.56	3801504	0.42	1.87	0.4	2.59
40	1.26	8572584	0.42	1.87	0.37	2.64
Philosophers.						
5	1.26	8572584	0.42	1.88	0.42	2.01
10	1.65	9988660	0.34	1.89	0.28	2.03
15	2.45	31666424	0.3	1.92	0.3	2.07
RSM.						
222	2.34	15306720	0.34	2.21	0.31	2.4
322	2.39	17594712	0.33	2.23	0.3	2.4
422	5.14	34436960	0.36	2.25	0.34	2.41
Aloha.						
20	0.34	2105960	0.87	1.71	0.92	1.98
30	1.25	7027840	0.61	1.71	0.61	2
40	3.18	16571720	0.45	1.73	0.45	1.99

at the same time than sequentially. The memory requirements are generallly between 20% and 100% more than for 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, the parallelisation incurs a number of overheads in practice. 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 inevitable in this type of loop-level parallelism, since *BFSFire* will differ in the amount of time it takes for 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. The combination of these overheads means that the extra costs to the algorithm often outweigh any benefit from introducing parallelism. Hence, it is not worth considering Parallel BFS any further, as it is performing worse than Chained BFS which in turn, as previous studies have established [14], is outperformed by Saturation.

Our experiments with BFS-style algorithms demonstrate 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 Chained BFS. Even then, the algorithm would require 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. 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 must also be considered that breaking a sequential optimisation when parallelising an algorithm can potentially outweigh any gains.

5.3.2 Sequential Saturation vs. Parallel Saturation

We now turn our attention to evaluating Parallel Saturation. The run-time and memory results displayed in Table 3 represent the relative run-time or memory of Parallel Saturation when compared to the original, sequential Saturation algorithm. The table illustrates how Parallel Saturation performs across one to four cores of our machine. Again, a relative performance factor of greater than 1 indicates a run-time speed-up or memory increase, while a factor of less than 1 indicates a run-time slow-down or memory decrease. The state space size is indicated by $|S|$.

Our run-time results show that Parallel Saturation exhibits a speed-up for two of the benchmark's models, which for FMS is over 100% faster than the sequential algorithm. For the models where parallelism can be exploited, we see a slight improvement in run-time for larger parameterised models. Generally, however, Parallel Saturation is unable to improve over the sequential algorithm: it uses two to three times more memory than the sequential algorithm, apart from the Slot 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 and memory results are affected by the overheads incurred by the parallel algorithm. High overheads often prevent Parallel Saturation from competing with Sequential Saturation. 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 [28] 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 Fig. 14. When coupled with our run-time results, the profiles provide clear evidence that models with low parallelism or high scheduling overhead 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 could be fired on smaller state sets than in the sequential version, which creates more work and larger intermediate MDDs. This extra work can outweigh any benefit of parallelism and also introduce higher overheads. The Slotted 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 events that can be explored in parallel affects the level of parallelism of

Table 3: Run-time and memory results for the thread pool algorithm

N	Time(s)	Mem(b)	**Relative Time (Cores)**				**Relative Memory (Cores)**			
			1	**2**	**3**	**4**	**1**	**2**	**3**	**4**
Slot. $\|S\|$ 90 : 5.9×10^{94} 120 : 5.1×10^{126} 150 : 4.5×10^{158}										
90	6.82	5923040	0.23	0.20	0.22	0.25	9.22	12.40	12.14	11.85
120	15.77	13405440	0.26	0.24	0.25	0.28	9.43	12.63	12.26	12.02
150	30.85	25441840	0.30	0.28	0.30	0.31	9.70	12.80	12.43	12.12
Robin. $\|S\|$ 180 : 6.2×10^{56} 210 : 7.8×10^{65} 240 : 9.5×10^{74}										
180	7.95	1165764	0.77	0.96	0.94	0.94	1.87	1.87	1.88	1.88
210	15.45	1574304	0.79	0.99	0.97	0.96	1.91	1.91	1.91	1.91
240	41.75	2044044	0.80	1.01	1.01	1.00	1.94	1.94	1.94	1.94
Kanban. $\|S\|$ 25 : 7.6×10^{12} 30 : 5.0×10^{13} 35 : 2.5×10^{14}										
25	2.80	7334600	0.54	0.64	0.65	0.65	1.89	1.89	1.90	1.91
30	5.06	13784976	0.56	0.67	0.67	0.69	1.98	1.99	1.99	1.99
35	10.22	23957940	0.57	0.70	0.71	0.72	2.07	2.07	2.08	2.08
FMS. $\|S\|$ 11 : 1.1×10^{9} 13 : 5.8×10^{9} 14 : 1.3×10^{10}										
11	12.17	3148980	0.79	1.46	1.85	2.12	1.78	1.73	1.70	1.71
13	55.49	8173844	0.78	1.49	1.90	2.17	1.80	1.81	1.84	1.88
14	119.98	12591300	0.80	1.51	1.98	2.18	2.02	2.00	2.00	1.97
Queens. $\|S\|$ 11 : 166926 12 : 856189 13 : 4674890										
11	1.79	4248776	0.44	0.48	0.50	0.52	2.01	1.91	1.91	1.91
12	19.42	19920672	0.50	0.54	0.55	0.57	2.13	2.01	2.01	2.01
13	439.16	99807456	0.55	0.59	0.61	0.62	2.41	2.24	2.24	2.24
Leader. $\|S\|$ 6 : 1.9×10^{6} 7 : 2.4×10^{7} 8 : 3.0×10^{8}										
6	3.78	2422704	0.54	0.50	0.50	0.48	2.74	2.75	2.77	2.80
7	24.11	7063232	0.52	0.52	0.52	0.46	3.01	3.13	3.18	3.20
8	128.11	16107968	0.51	0.53	0.53	0.49	3.39	3.44	3.49	3.60
BQ. $\|S\|$ 30 : 2.4×10^{8} 50 : 4.6×10^{9} 70 : 3.3×10^{10}										
30	2.08	2241036	0.40	0.38	0.38	0.37	1.99	2.05	2.05	2.06
50	24.48	15112996	0.43	0.40	0.39	0.39	2.05	2.13	2.15	2.15
70	146.79	54895356	0.48	0.43	0.42	0.42	2.19	2.49	2.50	2.50
Philosophers. $\|S\|$ 20 : 3.5×10^{12} 40 : 1.2×10^{25} 80 : 1.4×10^{50}										
20	14.91	569608	0.80	0.99	1.08	1.12	1.66	1.74	1.79	1.83
40	33.56	1097560	0.81	1.01	1.10	1.14	1.71	1.81	1.89	1.96
80	76.93	2321768	0.84	1.04	1.12	1.19	1.90	2.09	2.21	2.28
RSM. $\|S\|$ 332 : 1.3×10^{10} 532 : 3.8×10^{10} 832 : 1.0×10^{11}										
332	5.43	6267568	0.59	0.54	0.52	0.50	1.70	1.80	2.03	2.21
532	37.63	23307704	0.64	0.57	0.56	0.56	1.75	1.87	2.14	2.36
832	315.38	74024440	0.74	0.62	0.65	0.67	1.88	2.05	2.45	2.59
Aloha. $\|S\|$ 40 : 2.3×10^{13} 70 : 4.3×10^{22} 100 : 6.5×10^{31}										
40	2.68	15879556	0.74	0.78	0.79	0.79	1.52	1.52	1.53	1.53
70	22.18	82907316	0.80	0.84	0.86	0.86	1.65	1.66	1.66	1.66
100	66.20	13917096	0.82	0.84	0.87	0.88	1.74	1.75	1.75	1.77

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 in our experiments for Parallel Saturation can be seen in Fig. 14. Robin 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 BQ, Kanban and Leader, which suffer from high scheduling overhead. Low-parallelism models such as Robin and Aloha show less degradation in performance, but are unable to improve their run-times through parallelism. For the four high-parallelism models, two models were able to improve Parallel Saturation over Sequential Saturation. Thus, to unlock the potential of the other two high-parallelism models, further optimisations to our algorithm are required. We explore several starting points for such optimisations next.

6 Optimising Parallel Saturation

This section briefly discusses various potential optimisations of Parallel Saturation, whose details can be found in [21]. Before doing so, however, we first observe a couple of subtleties regarding current multi-core hardware architectures.

6.1 Four-core vs. Four-processor Architectures

Notably, when a run-time speed-up is encountered on our dual-processor, dual-core 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 multi-threaded algorithms [2]. This would suggest that the operating system scheduling on these cores is inefficient, since dual-cores are conceptually as efficient as secondary processors.

We can demonstrate that our dual-processor, dual-core architecture influences the run-time of Parallel Saturation by comparing our results against those obtained on 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. For these models a run-time improvement of approximately 10% was observed on the fourth core. A more even distribution of run-time improvement for each core was facilitated by the four processor machine, and the increase did not tail off on the third and fourth core.

6.2 Optimising the Task Queue Ordering

In Parallel Saturation, we pick tasks from the shared queue of the thread pool using a FIFO order. This order can be optimised since it affects the order in which nodes are saturated and hence the construction order of the state space. This can have a significant effect on an algorithm's performance, as we have already showed for Parallel BFS.

To investigate this phenomenon, we propose to replace the FIFO queue with 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 a constant insertion and deletion complexity; these complexities differ for variants of priority queues. To avoid an insertion and deletion overhead from using a priority queue, we employ a so-called *heap-based priority queue* which, for the number n of tasks in the queue, has a complexity for insertion and deletion that is logarithmic in n.

We implemented and applied the LIFO and priority queue variants to our benchmark on our dual-processor, dual-core machine. For six of the ten models, the ordering in which tasks are picked from the queue makes negligible difference to the run-time and memory efficiency. However, for models that are sensitive to the order in which MDD nodes are saturated, the queue orderings can

have a significant impact on both run-time and memory. The LIFO version has the most impact, improving the run-time on four models by up to approximately 80% and decreasing the memory usage on two models by up to approximately 20%. 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, typically around 10% less run-time improvement and 15% less memory decrease.

For the RSM model, both the LIFO and priority queue significantly improve the run-time and decrease memory consumption, the latter of which remains static when adding extra cores. The high parallelism of RSM is unlocked, facilitating a run-time speed-up over Sequential Saturation of over 100%. The other models that were sensitive to the queue ordering were Slot, Leader and Aloha; however, despite the optimisation, they did not show a 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 for load balancing and scheduling, as opposed to using multiple queues with either a thread pool or work-stealing mechanism. 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 — can have a significant effect on the run-time of Parallel Saturation.

6.3 Optimising the Ordering of Events

As pointed out before, the order in which events are fired in parallel can significantly affect Parallel Saturation's run-time and memory requirements. Fig. 15 shows the effect of events e_1, e_2, and e_3, with $Top(e_1) = Top(e_2) = Top(e_3) = k$, 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 $1 \xrightarrow{e_3} 0$ again once $0 \xrightarrow{e_2} 1$ has been fired, if this causes $p[1]$ to point to a different MDD node encoding more states. Similarly, if we fire $1 \xrightarrow{e_1} 2$ before firing $0 \xrightarrow{e_2} 1$, all events in $SCC\#2$ may have to be fired again.

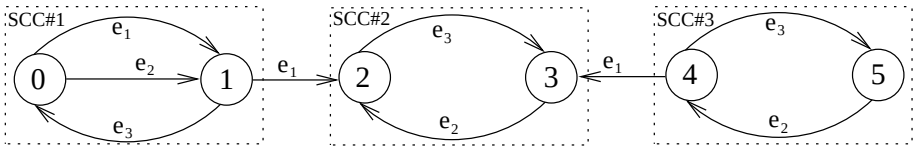

Figure 15: The effect of events with $Top = k$ on the states $\{0, 1, 2, 3, 4, 5\}$ of $\mathcal{S}_k$.

To address this problem, we use the *chaining heuristic* of [11], which extracts the strongly connected components (SCCs) from a *dynamic transition graph* that is built from the static graph of Fig. 15 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 heuristic tends to improve run-time and memory in a sequential implementation, it reduces the potential parallelism. It also introduces time and memory overheads due to storing SCC graphs, traversing them and managing parallel access. We refer the reader to [21] for implementation details.

The chaining heuristic significantly improves the run-time of Parallel Saturation, when applied to RSM, from a 50% slowdown to an over 100% speed-up. For Slot, 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 these two models, which can be exploited by using the chaining heuristic. However, the heuristic can also have a negative effect on run-time, as is the case for Robin and Kanban.

6.4 Optimising the Locking Strategy

For Parallel Saturation, we use an individual lock for the operation caches and the unique table, on a per level basis. We can therefore use a finer 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 *regional locking*. The range can be calculated using the modulo of the number of locks we wish to have, on the signature function. If a hash table needs to be resized, then the locks must be obtained before the resizing takes place.

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. For seven of our 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 between 5% and 30%. For the leader election model we observed 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 algorithm's performance.

The run-time results demonstrate that locking can be optimised, but only for a few models. The regional locking was unable to facilitate speed-ups over Sequential Saturation for these models. Where improvements were made using four 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 processors, increasing the number of locks may facilitate scalability. Our profiles of the parallel overheads in Fig. 14 also testify to low locking overheads on the data structures for a four-core architecture.

7 Related Work

This section discusses related work on parallelising state-space generation and model checking algorithms. The vast majority of existing approaches have focused on Networks of Workstations (NOW) and PC clusters rather than shared-memory architectures, for the purpose of increasing the available memory that is needed for storing a system model's state space. One major design decision when parallelising such algorithms is whether to partition a state space *statically* or *dynamically*. The other major design decision involves whether to represent a state space *explicitly* or *symbolically* (i.e., using decision diagrams).

7.1 Static vs. Dynamic Partitioning

Static partitioning methods are frequently applied to slice the data structure employed for storing state spaces, in order to distribute it as evenly as possible across the nodes of a network or cluster. This is often simpler for explicit approaches [3, 6, 25, 38, 39], since the states only have to be distributed individually. In symbolic state-space construction [10, 29, 30, 32, 42, 50], slicing the data structure is a more complex task. However, key is achieving an even distribution of states in order to avoid a load imbalance.

Dynamic partitioning methods have also been considered. They either re-distribute the data structure during state-space construction [30], or focus on a dynamic parallelisation of the underlying state-space generation algorithm [33]. 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 [6, 30], or tailoring the algorithm to a particular computer architecture [33] have also been employed to improve parallelisability.

7.2 Parallel Explicit Approaches

Stern and Dill [49] carried out early work on parallel state-space construction, by parallelising the explicit-state verifier Murφ on a NOW using a static slicing approach. During the construction task, random load balancing is employed to distribute states across a NOW. A hash function decides to which node a state is 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 [3, 39]. These demonstrate that the technique's efficiency depends on the cost of calculating the next-state function when 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 implementation of this technique in SPIN subsequently reported slow-downs due to the efficiency of SPIN's next-state calculation [39]. Behrmann et al.'s algorithm demonstrated a super-linear 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 [3].

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. A novel idea for addressing this overhead was introduced by Bollig et al. [6] who developed 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 in explicit-state approaches was carried out by Inggs and Barringer [33, 34, 35] on a shared-memory architecture. The approach employs *work-stealing* [1, 5] techniques in order to load balance. Many of the parallelisation overheads, such as the synchronisation overhead, 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 using private queues. Other parallelisation overheads such as *false sharing*, 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 allows the parallel algorithm to effectively reduce parallel overheads, resulting in good linear speed-ups for several models.

7.3 Parallel Symbolic Approaches

We can classify the symbolic parallelisation approaches into either approaches focused on data parallelism [10, 30, 32, 42, 50], on a combination of data and algorithm parallelism [29], or on utilising idle processors [9]. In contrast, the approach underlying this article is solely based on algorithm parallelisation.

Regarding data parallelism, the efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [42, 50] to sophisticated approaches relying on work-stealing [30]. These 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 [10] and vertically [32, 42, 50]. Horizontal slicing scales well but prevents the state-space generation task from being sped up, since each slice has to complete its work before the next slice can begin its work. Finding a good vertical slicing is a non-trivial issue that is 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 work-stealing techniques to distribute work dynamically [30].

Regarding combined data and algorithm parallelism, researchers have parallelised symbolic state-space generators to gain speed-ups from developing vertical slices on different processors of a NOW [29]. If the algorithm developing the slices has to synchronise frequently 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. To achieve speed-ups, the research tackles how best to

decrease the inherent synchronous nature. The resulting 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 work-stealing techniques developed in [30]. For very large circuits, this technique has proved to lead to a very efficient parallelisation that shows up to an order of magnitude improvement in time efficiency.

Recent work has also considered ways to utilise idle processors during state-space construction [9]. 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.

8 Conclusions and Future Directions

This article investigated whether *symbolic* state-space generation algorithms, and in particular the MDD-based *Saturation* algorithm for computing reachable state spaces of asynchronous system models, can be parallelised on shared-memory architectures, such as multi-processor, multi-core PCs. This turned out to be a challenging question due to the *irregularity* of this computing task. While we partially anticipated that run-time improvements similar to those attributable to sequential optimisations were not realistic expectations, the difficulties encountered in our endeavour to parallelise Saturation went far beyond that.

We demonstrated that a successful recipe for parallelising complex and irregular algorithms needs to account for all the intricate subtleties of the original algorithm and data structures, and needs to employ sophisticated techniques for benchmarking and measurement. We thus proposed a general evaluation technique for parallel state-space generation algorithms via systematic benchmarking. The classification methodolgy is based on a rigorous measuring of parallel overheads, such as load balancing, scheduling, and synchronisation. We assert that in the absense of a parallel overhead profile of the models under consideration it would be difficult to understand and improve parallel algorithms.

Regarding the question whether it is actually worth investing in parallelising rather than optimising a symbolic, sequential state-space exploration algorithm, one must lean towards optimisation, given the amount of effort necessary for parallelisation and the often small return in run-time efficiency. However, as illustrated by the Runway Safety Monitor application, there exist practical models where parallelisation is clearly beneficial.

This insight thus preempts a clear-cut verdict. We believe that the best alternative is investing on all fronts: fine-tuned, model-specific parallelisation, together with sequential optimisation and use of highly optimised, off-the-shelf packages for load balancing, such as *Cilk* [4]. For this approach to succeed, the model checking community needs to collaborate closely with parallel computing experts, and get advice on efficient implementation techniques and architecture-specific issues [22].

Funding. Engineering and Physical Sciences Research Council (GR/S86211/01); National Aeronautics and Space Administration (NCC-1-02043).

Acknowledgements. We thank Gianfranco Ciardo for many fruitful discussions on the topic.

References

[1] Agrawal, K., He, Y., and Leiserson, C. E. Adaptive work stealing with parallelism feedback. In *Principles and Practice of Parallel Programming*, pp. 112–120. ACM, 2007.

[2] Barnat, J. and Ročkai, P. Shared hash tables in parallel model checking. In *PDMC'07*, ENTCS, 2008. To appear.

[3] Behrmann, G., Hune, T., and Vaandrager, F. W. Distributing timed model checking: How the search order matters. In *CAV'00*, vol. 1855 of *LNCS*, pp. 216–231. Springer, 2000.

[4] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *Parallel and Distributed Computing*, 37(1):55–69, 1996.

[5] Blumofe, R. D. and Papadopoulos, D. The performance of work stealing in multiprogrammed environments. In *Measurement and Modeling of Computer Systems*, pp. 266–267. ACM, 1998.

[6] Bollig, B., Leucker, M., and Weber, M. Local parallel model checking for the alternation-free μ-calculus. In *SPIN'02*, vol. 2318 of *LNCS*, pp. 128–147. Springer, 2002.

[7] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, 1986.

[8] Butenhof, D. R. *Programming with POSIX threads*. Addison-Wesley, 1997.

[9] Chung, M.-Y. and Ciardo, G. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *IPDPS'03*. IEEE, 2003.

[10] Chung, M.-Y. and Ciardo, G. Saturation NOW. In *QEST'04*, pp. 272–281. IEEE, 2004.

[11] Chung, M.-Y., Ciardo, G., and Yu, A. J. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *ATVA'06*, vol. 4218 of *LNCS*, pp. 51–66. Springer, 2006.

[12] Ciardo, G., Jones, R. L., Miner, A. S., and Siminiceanu, R. Logical and stochastic modeling with SMART. *Performance Evaluation*, 63:578–608, 2006.

[13] Ciardo, G. and Lan, Y. Faster discrete-event simulation through structural caching. In *Performability Modeling of Computer and Communication Systems*, pp. 11–14. IEEE, 2003.

[14] Ciardo, G., Lüttgen, G., and Miner, A. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.

[15] Ciardo, G., Lüttgen, G., and Siminiceanu, R. Efficient symbolic state–space construction for asynchronous systems. In *ICATPN'00*, vol. 1839 of *LNCS*, pp. 103–122. Springer, 2000.

[16] Ciardo, G., Lüttgen, G., and Siminiceanu, R. Saturation: An efficient iteration strategy for symbolic state-space generation. In *TACAS'01*, vol. 2031 of *LNCS*, pp. 328–348. Springer, 2001.

[17] Ciardo, G., Marmorstein, R. M., and Siminiceanu, R. Saturation unbound. In *TACAS'03*, vol. 2619 of *LNCS*, pp. 379–393. Springer, 2003.

[18] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NuSMV: A new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[19] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent sys tems using temporal logic specifications. *Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[20] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. MIT, 1999.

[21] Ezekiel, J. *Parallelising Symbolic State-Space Generation Algorithms on Shared-Memory Architectures*. PhD thesis, University of York, UK, 2007.

[22] Ezekiel, J., Lüttgen, G., and Ciardo, G. Parallelising symbolic state-space generators. In *CAV'07*, vol. 4590 of *LNCS*, pp. 268–280. Springer, 2007.

[23] Ezekiel, J., Lüttgen, G., and Siminiceanu, R. Can Saturation be parallelised? On the parallelisation of a symbolic state-space generator. In *PDMC'06*, vol. 4346 of *LNCS*, pp. 331–346. Springer, 2007.

[24] Fernandes, P., Plateau, B., and Stewart, W. J. Efficient descriptor vector multiplications in stochastic automata networks. *ACM*, 45(3):381–414, 1998.

[25] Garavel, H., Mateescu, R., and Smarandache, I. Parallel state space construction for model-checking. In *SPIN'01*, vol. 2057 of *LNCS*, pp. 217–234. Springer, 2001.

[26] Gautier, T., Roch, J.-L., and Villard, G. Regular versus irregular problems and algorithms. In *IRREGULAR'95*, vol. 980 of *LNCS*, pp. 1–25. Springer, 1995.

[27] Graf, S., Steffen, B., and Lüttgen, G. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[28] Graham, S. L., Kessler, P. B., and McKusick, M. K. `gprof`: A call graph execution profiler. *ACM SIGPLAN Notices*, 39(4):49–57, 2004.

[29] Grumberg, O., Heyman, T., Ifergan, N., and Schuster, A. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME'05*, vol. 3725 of *LNCS*, pp. 129–145. Springer, 2005.

[30] Grumberg, O., Heyman, T., and Schuster, A. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.

[31] Gupta, A., Tucker, A., and Urushibara, S. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. *ACM SIGMETRICS Performance Evaluation Review*, 19(1):120–132, 1991.

[32] Heyman, T., Geist, D., Grumberg, O., and Schuster, A. Achieving scalability in parallel reachability analysis of very large circuits. In *CAV'00*, vol. 1855 of *LNCS*, pp. 20–35. Springer, 2000.

[33] Inggs, C. P. *Parallel Model Checking On Shared Memory Architectures*. PhD thesis, University of Manchester, UK, 2004.

[34] Inggs, C. P. and Barringer, H. Effective state exploration for model checking on a shared memory architecture. In *PDMC'02*, vol. 68(4) of *ENTCS*, 2002.

[35] Inggs, C. P. and Barringer, H. CTL* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, 2006.

[36] Itai, A. and Rodeh, M. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[37] Kam, T., Villa, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[38] Knottenbelt, W. J., Harrison, P. G., Mestern, M. A., and Kritzinger, P. S. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Performance Evaluation*, 39(1-4):127–148, 2000.

[39] Lerda, F. and Sisto, R. Distributed-memory model checking with SPIN. In *SPIN'99*, vol. 1680 of *LNCS*, pp. 22–39. Springer, 1999.

[40] Lucco, S. A dynamic scheduling method for irregular parallel programs. In *PLDI'92*, pp. 200–211. ACM, 1992.

[41] McMillan, K. *Symbolic Model Checking*. Kluwer, 1993.

[42] Milvang-Jensen, K. and Hu, A. J. BDDNOW: A parallel BDD package. In *FMCAD'98*, vol. 1522 of *LNCS*, pp. 501–507. Springer, 1998.

[43] Miner, A. S. and Ciardo, G. Efficient reachability set generation and storage using decision diagrams. In *ICATPN'99*, vol. 1639 of *LNCS*, pp. 6–25. Springer, 1999.

[44] Pastor, E., Roig, O., Cortadella, J., and Badia, R. M. Petri net analysis using boolean manipulation. In *PNPM'94*, vol. 815 of *LNCS*, pp. 416–435. Springer, 1994.

[45] Prechelt, L. and Hanssgen, S. U. Efficient parallel execution of irregular recursive programs. *IEEE Trans. on Parallel and Distributed Systems*, 13(2):167–178, 2002.

[46] Queille, J. P. and Sifakis, J. Specification and verification of concurrent systems in CESAR. In *Symp. on Programming*, vol. 137 of *LNCS*, pp. 337–351. Springer, 1982.

[47] Siminiceanu, R. I. and Ciardo, G. Formal verification of the NASA runway safety monitor. *Software Tools for Technology Transfer*, 9(1):63–76, 2007.

[48] Solé, M. and Pastor, E. Traversal techniques for concurrent systems. In *FMCAD'02*, vol. 2517 of *LNCS*, pp. 220–237. Springer, 2002.

[49] Stern, U. and Dill, D. L. Parallelizing the Murφ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.

[50] Stornetta, T. and Brewer, F. Implementation of an efficient parallel BDD package. In *DAC'96*, pp. 641–644. ACM, 1996.

[51] Tilgner, M., Takahashi, Y., and Ciardo, G. SNS 1.0: Synchronized network solver. In *ICATPN'96*, pp. 215–234, 1996.

[52] Yang, T. and Fu, C. Space/time-efficient scheduling and execution of parallel irregular computations. *ACM Trans. on Programming Languages and Systems*, 20(6), 1998.