Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 198 (2008) 47–61
www.elsevier.com/locate/entcs

Measuring and Evaluating Parallel State-Space Exploration Algorithms[1]

Jonathan Ezekiel and Gerald Lüttgen[2]

Department of Computer Science, University of York, York, YO10 5DD, U.K.

Abstract

We argue in this paper that benchmarking should be complemented by direct measurement of parallelisation overheads when evaluating parallel state-space exploration algorithms. This poses several challenges that so far have not been addressed in the literature: what exactly are those overheads, how can and cannot they be measured, and how should system models be selected in order to expose the causes of parallelisation (in)efficiencies? We discuss and answer these questions based on our experience with parallelising Saturation – a symbolic algorithm for generating state-spaces of asynchronous system models – on a shared-memory architecture. Doing so will hopefully spare newcomers to the growing PDMC community from having to learn these lessons the hard way, as we did over a painful period of almost three years.

Keywords: Model checking, parallel state-space generation, parallel overhead, shared-memory architecture.

1 Introduction

Algorithms for *automated verification*, such as those used to compute the state-spaces of system models, are often too time-consuming, given the complexity of today's systems. Approaches to parallelising such algorithms [2,15,19,20,24,26] seek to improve their time-efficiency by utilising the extra processing power available from multi-processor machines. In practice, the resulting parallel algorithms often show speedups on very few models. This is due to a combination of factors: (i) state-space exploration algorithms incur high parallel overheads due to their irregular nature in terms of unpredictable sizes of work and random access to data structures such as hash tables, and (ii) they are dependent upon the model for parallelisability. Taking these issues into account when parallelising such algorithms is important; however, information related to overheads and model characteristics is usually not or

[1] Research funding was provided by the EPSRC under grant no. GR/S86211/01.
[2] {jezekiel,luettgen}@cs.york.ac.uk

doi:10.1016/j.entcs.2007.10.020

only incompletely collected when an algorithm is evaluated. This makes it difficult to assess how well parallelised an algorithm is.

The ability to evaluate a parallel algorithm accurately is useful for determining where improvements can be made, and for considering whether the technique underlying the algorithm can be used in other algorithms. An example of a technique that works for one algorithm and not for another is the static partitioning technique of [24,26]: it resulted in linear speedups when originally applied to Murφ [26], but in slowdowns when applied to SPIN [24]. This was due to the SPIN implementation incurring significant communication overheads that were almost negligible in Murφ. Thus, techniques can work under certain circumstances but not under others. The ability to determine the effect of parallelisation overheads when evaluating the performance of a parallel algorithm is crucial to understanding the applicability of the underlying technique.

It is well known in the parallel community that overheads of a parallel algorithm need to be studied [14]; indeed, theoretical analyses of their impact on parallel state-space generation algorithms have been carried out previously (see, e.g., [23]). A theoretical analysis of state-space exploration algorithms on shared-memory architectures is, however, a difficult process due to the unpredictability of scheduling and synchronisation. These overheads become even more unpredictable in *symbolic* state-space exploration due to the increased irregularity of the employed data structure. Thus, theoretical analysis cannot always be performed satisfactorily, and *direct measurement* of overheads at run-time is necessary. Direct measurement is a challenging task in itself, as is reported, e.g., by Inggs in [19]. One difficulty is that any accurate measuring technique must take into account its own cost to the algorithm during the measurement process.

Previous approaches to parallelising state-space exploration algorithms have shown that the severity of parallel overheads is highly model dependent. When using a benchmark of examples to evaluate Grumberg et al's parallel algorithm in [15], reported speedups varied greatly from little over one to an order of magnitude. Thus, a parallel state-space exploration algorithm can seem well or badly parallelised depending on the characteristics of the model. Choosing suitable models and determining their effect on parallelism is therefore key to evaluating an algorithm's performance. Understanding the model's influence on the overheads is key when selecting models to illustrate a parallel state-space exploration algorithm's performance.

This paper shares our experience of addressing these challenges while parallelising our *Saturation* symbolic state-space generation algorithm on a multi-processor, multi-core PC [7]. We contribute the knowledge from our investigation into irregularity and subsequent overheads impacting on our parallel algorithm [8,9]. Based on this investigation we suggest an approach to evaluating parallel state-space exploration algorithms, which compliments the use of benchmarking to assess performance and the quality of parallelisation. There are two key sections to this paper: Sec. 2 deals with the aspects to parallelising a state-space exploration algorithm by looking at irregularity, the overheads arising from irregularity, how the algorithm

can be implemented, and techniques to address the overheads. Sec. 3 presents our approach to evaluating parallel state-space exploration algorithms, showing how overheads can be measured, how models can be constructed to assess the overheads, and how the results from real models can be put into context. We round off the paper with a discussion of related work (cf. Sec. 4) and our conclusions (cf. Sec. 5).

2 Background on parallelising state-space exploration algorithms

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

```
Reachability(in startstate : state)
 declare nextstate : state;
 1. Push(stack, startstate);
 2. do
 3.   nextstate ⇐ Pop(stack);
 4.   Enumerate(nextstate);
 5. while stack ≠ empty
```

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

stack : a stack that a state can be pushed onto or popped from using *Push* and *Pop*
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 access to this is atomic
Lock, *UnLock* : lock and unlock the data structures for atomic access

```
ParallelReachability(in startstate : state)
 declare nextstate : state;
 1. Push(stack, startstate);
 2. threads ⇐ 1;
 3. do
 4.   Lock(stack);
 5.   nextstate ⇐ Pop(stack);
 6.   UnLock(stack);
 7.   threads ⇐ threads + 1;
 8.   parallel ParallelEnumerate(nextstate);
 9. while threads ≠ 1 and stack ≠ empty
```

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

Fig. 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 start state *startstate* and transition relation (next-state function) *Successor*, 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 a start 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 pushes it onto a stack. *Reachability* pops a new state off the stack and repeatedly calls *Enumerate* on that state until there are no more states on the stack, at which point the entire state-space has been explored.

To a novice who wishes to parallelise this algorithm, the obvious starting point is to parallelise the *Enumerate* function, allowing states to be enumerated in parallel. This seems like an easy and effective parallelisation, which leaves it to the operating system to schedule the generated threads on available processors. We show the corresponding parallel algorithm on the right of 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 on the stack. The function must also ensure that, when it pops a state from the stack, access to the stack is atomic; this is to prevent *data races* from other threads inserting states into the stack at the same time. To achieve this, *Parallel-Reachability* locks the stack when accessing it and unlocks it when the state has been popped. 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 stack, it locks and unlocks them for atomicity. *ParallelEnumerate* decrements the *threads* counter when it finishes, which allows the *ParallelReachability* function to monitor the number of threads.

2.2 What are the overheads?

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

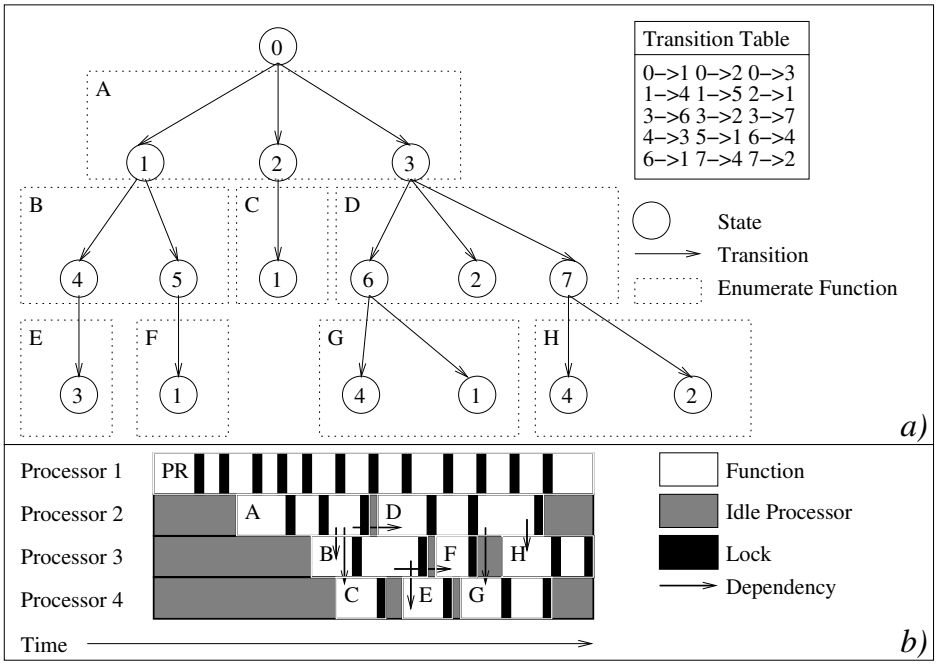

Fig. 2. Parallel state-space enumeration and processor allocation of functions.

Fig. 2(a) shows the parallel state-space construction for the model defined by the transition table and start state 0; the displayed tree should be read as a search tree through the model's state-space. While this example deals with explicit state-space generation, the 'states' could equally well be read as units of work conducted within a symbolic approach. The work of the *ParallelEnumerate* function is visualised using a broken box, which includes the enumeration of duplicate states. Fig. 2(b) shows 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 stack 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 [12]. Secondly, as is highlighted by the dependencies between the functions, state-space exploration is a *producer/consumer* problem; in particular, work must take place before other work that is dependent upon it is even created.

As a result of these characteristics of the algorithm, a number of *parallel overheads* arise, which impact on the run-time of the algorithm and cannot be deter-

mined from run-time measurements alone. Firstly, irregularity causes *load imbalance*, where work is not distributed evenly amongst processors, which means that they are not fully utilised for work. This problem is further compounded by the producer/consumer nature of the algorithm which imposes the restriction that work cannot be scheduled on processors until other work is completed. Secondly, small computations result in *scheduling overhead*, where the cost for scheduling work, e.g., the cost of creating a thread (approx. 90,000 ns on a modern PC running Linux) can be higher than the work performed by a small computation (approx. 1,200 ns) [9]. Thirdly, frequent locking results in high *synchronisation overhead*, i.e., the time taken to lock and unlock a data structure, which would translate to *communication overhead* on a PC cluster where processes must frequently synchronise with each other.

2.3 Implementing a parallel state-space exploration algorithm

One of the great difficulties of parallelising state-space exploration algorithms is understanding how to efficiently implement a devised parallel algorithm. The types of languages and libraries chosen for parallelisation can have a significant effect on the performance of the parallel algorithm, since their characteristics can influence overheads. The importance of these choices can be shown when inspecting the work of Inggs [19]: the use of Java led to high synchronisation overheads for memory allocation and garbage collection, and eventually Inggs used C to implement her parallel algorithm. Language selection can therefore be costly in time and effort if it hinders parallelisation.

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

Another question that arises when parallelisation decisions are made is whether to use native thread libraries, or whether to use a library with a higher level of abstraction in order to make programming easier. For example, on shared-memory architectures, OpenMP [`www.openmp.org`] can be used to schedule work, isolating the programmer from having to understand thread programming, as well as allowing portability between operating systems. The drawback to this approach is that often fine control over the parallelism is lost. For example, we discovered that OpenMP is unable to effectively parallelise our mutually recursive functions in Saturation, since there is no control statement to handle recursion. We therefore found the *Pthreads* library [5] more suitable than OpenMP for parallelising our algorithm.

Other libraries can sometimes offer a method for addressing scheduling and load

balancing issues. For example, Cilk [10], which is a library for shared-memory architectures, offers an in-built efficient load balancing and scheduling model. However, because of Cilk's restrictions on this model and its inability to deal with producer/consumer problems efficiently, our Cilk implementation of parallel Saturation encountered high memory overhead [8]. Thus, any library used to schedule and load balance a parallel state-space exploration algorithm must be able to express producer/consumer problems efficiently, but currently no library exists that is suitable for this type of problem.

Selecting an appropriate architecture is also an important decision when considering how to parallelise a state-space exploration algorithm. Availability is a key issue when choosing hardware. Multi-processor, multi-core PC are becoming widely available for performing experimental analysis on parallel algorithms. The drawbacks of this type of machine are that they only offer a relatively small number of processors/cores and that secondary cores are approximately 30-40% less efficient than processors. 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 [17]. 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 (see also Sec. 3).

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 *workstealing* techniques. These have been used in parallel state-space exploration algorithms to facilitate orders of magnitude improvements in time-efficiency [15,19]. We applied workstealing to parallel Saturation [8], and our results using this technique demonstrated speedups on several models including a super-linear speedup. Workstealing is based on the principle that, when one processor runs out of work to do, it *steals* work from another processor. For instance, if processors are given a number of states to enumerate, a processor completing its work can attempt to steal states from other processors. While this can be effective in spreading work to multiple processors efficiently, the technique introduces its own overhead from extra code and synchronisation. Thus, in order to improve the run-time of the parallel algorithm, sufficient parallel work must exist for the technique to spread fully across the available processors.

Scheduling overheads can typically be reduced directly using some form of *thread-pool*, or lightweight threads rather than native operating system threads. We used a thread-pool in our Saturation algorithm [9], and the cost to schedule work improved by an order of magnitude, compared to creating a native thread to perform work. Indirectly, scheduling overheads can be reduced by attempting to

increase the amount of work performed by each thread, so that computations are scheduled less frequently. The success of this technique is highly dependent upon there being enough work that can be independently grouped together.

Synchronisation overheads can be alleviated by facilitating task independence and minimising the amount of access required to the underlying data structures (such as the stack and the state store). These approaches are key to the order of magnitude speedups attained from parallel state-space exploration algorithms in [15,19]. 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 [19]. This requires an extensive study of the machine's architecture and careful consideration of the utilisation and placement of data structures.

3 Evaluating parallel state exploration algorithms

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

3.1 Measuring parallel overheads

The quality of the measurement of parallel overheads, and how it reflects on the performance of a parallel state-space exploration algorithm, is a key issue that has yet to be addressed in the literature. The overheads of the algorithm are usually measured through some form of estimation, such as the distribution of states across processors to indicate load (im)balance. When we tried to estimate the amount of parallel work arising from our next-state function, we found that the influences on the parallelisation are much more complex than the factors we initially considered [9]. Thus, estimates of overheads may not be an accurate measurement of their true impact on a parallel algorithm, which brings their contribution towards an objective evaluation into question.

We now address the problem of measuring parallel overheads for each type of overhead in turn. Load balancing is difficult to measure. If we count the number of states enumerated on a processor and discover that this number is fairly even on different processors, we could argue that the load is well balanced amongst processors. However, what has not been taken into account here is the way in which work has been scheduled: due to the dependencies between states, the processors

may not have enumerated these states at the same time, and the processors could have been frequently idle. Similar problems also exist for measuring scheduling overheads. We could count the number of times work has been scheduled and multiply it by a measurement of the time taken to schedule work, e.g., the cost of the creation of a thread, in order to estimate the overhead. However, this would be an inaccurate measure since the operating system decides how work is scheduled, and the cost of re-scheduling work to another processor would be omitted from the measurement. Synchronisation overheads could be timed by instrumenting the code with a timer for each mutex lock, i.e., starting the timer before attaining the lock and stopping it once the lock has been attained. We found however, that this method is inaccurate, since the timer increases the amount of time the lock is opened for and introduces its own cost into the algorithm. Others have found similar problems with estimating synchronisation overheads [19].

The only solution to accurately measuring these overheads is via *direct* measurement, but this is a challenging task. How does one measure the activity on each of the processors? Even finding a timer that is fine-granular enough to measure the time spent acquiring a mutex lock posed a problem for us. We tried using the *gprof* algorithm profiler [13], but it was developed to profile sequential algorithms and could not show the individual activity of threads. What we required was a parallel profiling tool that can accurately analyse each of the parallel overheads at run-time, while taking into account the cost of its own instrumentation. To the best of our knowledge, the only such tool that currently exists on shared-memory architectures is Intel's *Thread Profiler* [www.intel.com/software/products/threading/]. In order to evaluate parallel Saturation, we applied the profiler to the parallel algorithm at run-time. We chose our experimental architecture according to the constraints on the types of processors and operating systems the profiler supports, i.e., Intel processors, certain flavours of Linux, C and C++ and particular compilers. Fig. 3 illustrates measurements that can be made using the profiler, showing the percentage of the algorithm's run-time taken up by scheduling, synchronisation, serial execution, execution on less than the available processors, and fully parallel execution. This is an accurate and thorough breakdown of the algorithm's parallel overheads. Although we show the overall profile, much more detailed profiles are also available, such as profiles of the algorithm on specific processors and the cost of individual locks.

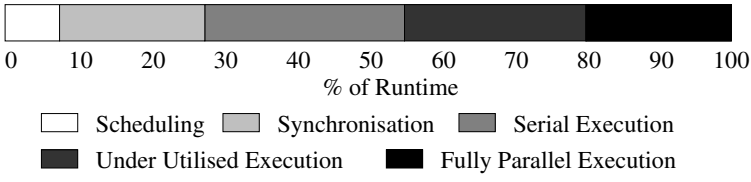

Fig. 3. Parallel algorithm profile obtained from the Intel thread profiler.

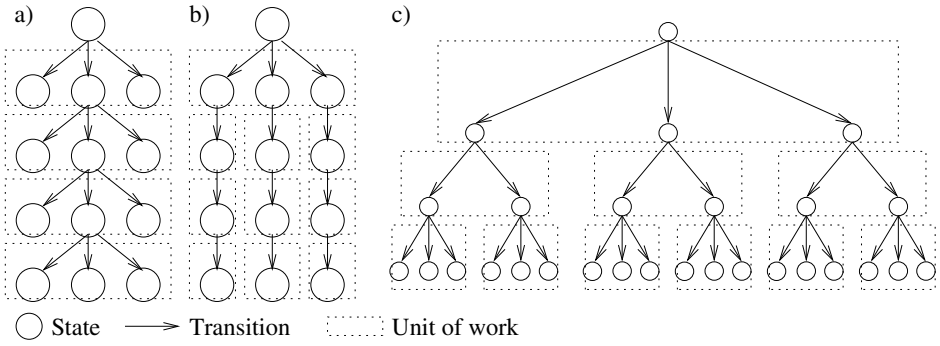

Fig. 4. Search trees constructed during reachability analysis for three pathological models.

3.2 The influences of a model on parallel overheads

While parallel overheads can be identified as a general influence on an algorithm, a subtlety of state-space exploration algorithms, and one which is not discussed in related literature, is the affect of the model on the severity of the overheads. We found this out the difficult way in Saturation, since some models showed good performance using the parallel algorithm, and others showed a lack of parallelisability [9]. At first glance these results can suggest that the parallelisation is inefficient, which is a highly frustrating point to consider when a lot of work has gone into the parallel algorithm. Our experience from investigating the effects of the underlying model can hopefully be used to alleviate future frustration relating to this point, by illustrating that the parallelisation efficiency is highly dependent upon the model as well as the techniques that have been used to address overheads.

Fig. 4 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 states. We use pathological examples to illustrate particular overheads that may arise during the construction of the state-space for three stereotypical types of model. Fig. 4(a) shows a model where the work cannot be spread across processors, since the work units are essentially sequentialised due to the dependencies between them. Fig. 4(b) illustrates a model that imposes high scheduling overheads, due to the small size of the work units. Fig. 4(c) is an ideal model that can potentially be well parallelised, since the work can be spread across processors in parallel and since the units of work are large enough to minimise scheduling overhead.

The illustration highlights that the characteristics of the model under consideration is a key influence on the performance of a parallel algorithm. In practice, the models that we used to evaluate our parallel Saturation algorithm show a combination of these factors, with performance varying from super-linear speedups to slowdowns of over 60% [8,9]. Other parallelisations of state-space exploration algorithms also show widely varying performance of the algorithm according to the models' characteristics [15]. 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.

3.3 Evaluating a parallel algorithm

Given the overhead measurement and model dependency issues that we have highlighted, a good quality evaluation of a parallel state-space exploration algorithm should include an accurate direct measurement of the overheads, and analyse the effect of the employed models on the parallel algorithm. Most importantly, the set of models used for evaluating a parallel state-space exploration algorithm should cover the space defined by the three stereotypes of Fig. 4. This allows a way in which each particular type of overhead can be thoroughly evaluated and (in)efficiencies in the algorithm can be pinpointed.

Fig. 5 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 speedup obtained by the algorithm. When considering the effect that the underlying model has on a parallel algorithm, a good evaluation must include models which cause the algorithm to incur overheads, and challenge its effectiveness at gaining run-time speedups. Without such challenging models, the overall effectiveness of the parallel algorithm is difficult to determine. Coupling challenging models with a profile of the overheads allows the algorithm's performance

Fig. 5. Profiles of three stereotypical models.

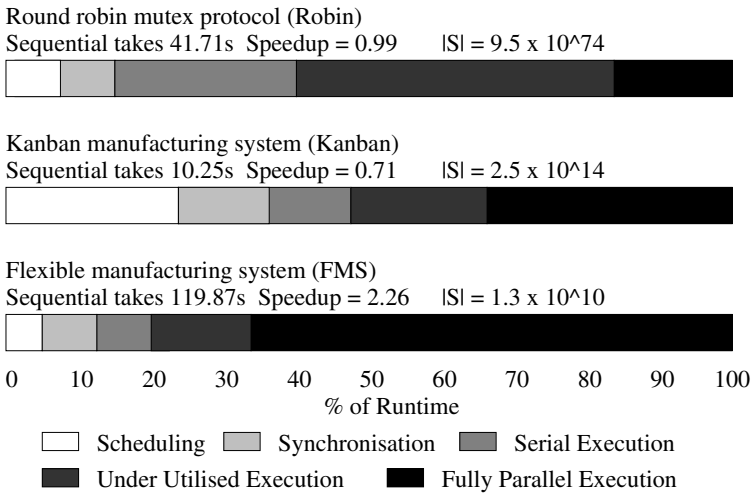

Fig. 6. Profiles of the parallel Saturation algorithm for different models.

to be truly put into context. For example, our experiments on parallel Saturation are put into context by the profiles in Fig. 6, where we illustrate three of our ten profiled real models, namely Robin, Kanban and FMS, which were included in our benchmark that was used in [8,9]. The models fit into the stereotypical categories shown in Fig. 5, and the other seven models in our benchmark fall in between these categories. The time for the sequential algorithm is stated, along with the time-efficiency of parallel Saturation relative to sequential Saturation, where a number greater than 1 indicates a speedup; |S| denotes the approximate size of the state space.

The profiles show the overheads specific to the models and how they challenge the parallelism of the algorithm. The Robin profile roughly fits the low parallelism profile of the model in Fig. 5(a): cores are under-utilised which prevents an improvement in run-time through a lack of parallelism. The Kanban profile roughly fits the high scheduling model profile in Fig. 5(b): high scheduling overheads cause a decrease in run-time. The FMS profile roughly fits the ideal model profile in Fig. 5(c): scheduling is low and the cores are fully utilised for a high percentage of the run-time, resulting in a speedup. These profiles are highly useful in drawing conclusions as to the performance of the algorithm. In fact without them, one would not be able to understand the circumstances under which parallel Saturation is able to improve over sequential Saturation. This is why model selection should try and include models which fit the profile of each of our stereotypical models. Where stereotypical models are unavailable that fit these profiles, they can be artificially constructed, e.g., the authors of the parallel state-space exploration algorithm in [20] used artificial models that varied the amount of parallel work available.

In summary, profiling the performance of parallel state-space exploration algorithms using models to illustrate specific overheads provides an opportunity to evaluate the efficiency of individual parallelisation techniques. Selecting and profiling challenging models shows the circumstances under which the algorithms perform

well, and can be used to evaluate their overall performance. For these reasons, we suggest that benchmarking should be complemented by the evaluation technique that we have described, in order to provide a clear picture of how well a state-space exploration algorithm is parallelised.

4 Related work

Most of the work on parallel state-space exploration has focused on networks of workstations (NOWs), primarily using static partitioning of state-spaces [2,3,4,6,11,24,25,26,27]. The evaluation of the proposed parallel algorithms usually involved benchmarking in terms of run-time and some estimation of the work distribution and communication overhead. To the best of our knowledge, only one paper gives a breakdown of the run-time overheads [18].

Publications of *explicit* parallel state-space exploration algorithms employing *static* partitioning on NOWs typically use one to five models to benchmark an algorithm according to its run-time [2,3,11,23,24,26]. For example, the evaluation of the parallel Murφ verifier in [26] estimates the performance of the algorithm according to the communication overhead and the partitioning method, but demonstrates the accuracy of the estimation for only three models. A more thorough theoretical analysis of a parallel state-space exploration algorithms' predicted performance was carried out in [23], but is only demonstrated for a single parameterised model. Artificial models were constructed to evaluate a parallel negative cycle detection algorithm in [4]; however, only the run-time and number of messages passed between workstations is given as an indicator of performance. Publications of *symbolic* parallel state-space exploration algorithms employing *static* partitioning on NOWs generally take a similar approach to evaluation, by using two to four models to produce run-times and memory consumption as a performance measure [6,25]. For the algorithm for the parallel construction of BDDs on a shared-memory architecture of [22], also a small number of models is used to evaluate its performance, and in terms of run-time only. The evaluation of the parallel symbolic algorithm on a NOW in [27] provides more information, but these are only estimates of overheads, such as the number of messages passed between workstations to quantify communication overhead.

Dynamic approaches to symbolic parallelisation on NOWs appear to contain a more complete evaluation of performance. Heyman et al. [18] provide a thorough breakdown of the time spent in component parts of their algorithm, such as communication, sequential execution and memory balancing, using seven models. This gives a clear picture of how the algorithm performs. However, how the timings were measured and whether they were instrumented in the code, or obtained using a profiling tool, is not described in the paper. Grumberg et al. successfully parallelised a symbolic state-space exploration algorithm on a NOW using dynamic load balancing [15], based on the workstealing techniques in [16]. Their evaluation of the algorithm includes an estimate of how the load is balanced according to the dynamic partitioning method used. The benchmark employs nine models for

evaluation, and the authors select models that exhibit a varying range of run-time speedups, between little over one to an order of magnitude. However, a breakdown of the individual overheads is omitted from the evaluation. Inggs and Barringer's work uses workstealing for explicit state-space exploration on a shared-memory architecture [19,20,21]. They consider the overheads associated with the parallelisation and minimise them, by optimising the data structures employed by their algorithm for the specifics of the architecture. They mention that direct measurement of some of the overheads is difficult and thus are unable to provide direct measurements when reporting results [20].

Our approach to evaluation is unique, firstly, with regards to the quality and thoroughness of parallel overhead measurement and, secondly, in the way we choose models to measure specific overheads. Using a profiler that provides a direct measurement of overheads, while taking into account the cost of its own instrumentation, assures the accuracy of measurement. Combining a thorough evaluation of a parallel state-space algorithm with a carefully selected benchmark of its run-time performance serves to provide a clear picture of how efficiently parallelised the algorithm is.

5 Conclusions

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

To show how such an evaluation can be carried out, we described the overheads that arise from state-space exploration, explained how to select models in order to highlight specific overheads, and showed how overheads can be directly measured at run-time. Although the direct measurement we used was devised for shared-memory architectures, we believe that our ideas can be extended to include PC clusters, by applying available profiling tools for measurement on these architectures, some of which can be found in [1].

References

[1] Baker, M. and R. Buyya, *Cluster computing: The commodity supercomputer*, SPE **29** (1999), pp. 551–576.

[2] Behrmann, G., T. Hune and F. W. Vaandrager, *Distributing timed model checking – How the search order matters*, in: *CAV*, LNCS **1855**, 2000, pp. 216–231.

[3] Bollig, B., M. Leucker and M. Weber, *Local parallel model checking for the alternation-free μ-calculus*, in: *SPIN*, LNCS **2318**, 2002, pp. 128–147.

[4] Brim, L., I. Černá, P. Krčál and R. Pelánek, *Distributed LTL model checking based on negative cycle detection*, in: *FSTTCS*, LNCS **2245**, 2001, pp. 96–107.

[5] Butenhof, D. R., "Programming with POSIX Threads," Addison-Wesley, 1997.

[6] Chung, M.-Y. and G. Ciardo, *Saturation NOW*, in: *QEST* (2004), pp. 272–281.

[7] Ciardo, G., G. Lüttgen and R. Siminiceanu, *Saturation: An efficient iteration strategy for symbolic state-space generation*, in: *TACAS*, LNCS **2031**, 2001, pp. 328–342.

[8] Ezekiel, J., G. Lüttgen and G. Ciardo, *Parallelising symbolic state-space generators*, in: *CAV*, LNCS, 2007, to appear.

[9] Ezekiel, J., G. Lüttgen and R. Siminiceanu, *Can Saturation be parallelised? On the parallelisation of a symbolic state-space generator*, in: *PDMC*, LNCS **4346**, 2006, pp. 331–346.

[10] Frigo, M., C. E. Leiserson and K. H. Randall, *The implementation of the Cilk-5 multithreaded language*, in: *SIGPLAN* (1998), pp. 212–223.

[11] Garavel, H., R. Mateescu and I. Smarandache, *Parallel state space construction for model-checking*, in: *SPIN*, LNCS **2057**, 2001, pp. 217–234.

[12] Gautier, T., J.-L. Roch and G. Villard, *Regular versus irregular problems and algorithms*, in: *IRREGULAR*, LNCS **980**, 1995, pp. 1–25.

[13] Graham, S. L., P. B. Kessler and M. K. McKusick, *Gprof: A call graph execution profiler (with retrospective)*, in: *PLDI* (1982), pp. 49–57.

[14] Grama, A., A. Gupta, G. Karypis and V.Kumar, "Introduction to Parallel Computing," Addison-Wesley, 2002.

[15] Grumberg, O., T. Heyman, N. Ifergan and A. Schuster, *Achieving speedups in distributed symbolic reachability analysis through asynchronous computation*, in: *CHARME*, LNCS **3725**, 2005, pp. 129–145.

[16] Grumberg, O., T. Heyman and A. Schuster, *A work-efficient distributed algorithm for reachability analysis*, in: *CAV*, LNCS **2725**, 2003, pp. 54–66.

[17] Gupta, A., A. Tucker and S. Urushibara, *The impact of operating system scheduling policies and synchronization methods of performance of parallel applications*, in: *SIGMETRICS* (1991), pp. 120–132.

[18] Heyman, T., D. Geist, O. Grumberg and A. Schuster, *Achieving scalability in parallel reachability analysis of very large circuits*, in: *CAV*, LNCS **1855**, 2000, pp. 20–35.

[19] Inggs, C. P., "Parallel Model Checking on Shared Memory Architectures," Ph.D. thesis, University of Manchester, UK (2004).

[20] Inggs, C. P. and H. Barringer, *Effective state exploration for model checking on a shared memory architecture*, in: *PDMC*, ENTCS **68(4)**, 2002.

[21] Inggs, C. P. and H. Barringer, *CTL* model checking on a shared-memory architecture*, FMSD **29** (2006), pp. 135–155.

[22] Kimura, S. and E. M. Clarke, *A parallel algorithm for constructing binary decision diagrams*, in: *ICCD* (1990), pp. 220–223.

[23] Knottenbelt, W. J., P. G. Harrison, M. A. Mestern and P. S. Kritzinger, *A probabilistic dynamic technique for the distributed generation of very large state spaces*, Perform. Eval. **39** (2000), pp. 127–148.

[24] Lerda, F. and R. Sisto, *Distributed-memory model checking with SPIN*, in: *SPIN*, LNCS **1680**, 1999, pp. 22–39.

[25] Milvang-Jensen, K. and A. J. Hu, *BDDNOW: A parallel BDD package*, in: *FMCAD*, LNCS **1522**, 1998, pp. 501–507.

[26] Stern, U. and D. L. Dill, *Parallelizing the Murφ verifier*, FMSD **18** (2001), pp. 117–129.

[27] Stornetta, T. and F. Brewer, *Implementation of an efficient parallel BDD package*, in: *DAC* (1996), pp. 641–644.