Communicating Process Architectures 2000
P.H. Welch and A.W.P. Bakkers (Eds.)
IOS Press, 2000

Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems

Jeremy M.R. MARTIN
Oxford Supercomputing Centre,
Wolfson Building,
Parks Road,
Oxford,
OX1 3QD, UK

Yvonne HUDDART
Oxford University Computing Laboratory,
Wolfson Building,
Parks Road,
Oxford,
OX1 3QD, UK

Abstract. Conventional model-checking techniques for analysing concurrent systems for deadlock or livelock are hampered by the problem of exponential state explosion: the overall number of global states that needs to be checked may grow exponentially with the number of component processes in the system. The state-of-the-art commercial tool FDR provides deadlock and livelock analysis but it is limited at present to analysing systems of up to one hundred million global states.

The Deadlock Checker tool is capable of analysing very much larger systems by taking certain short-cuts. But this is achieved at the cost of incompleteness – there are certain deadlock-free and livelock-free networks that may not be proven so using that tool.

Here we investigate a different approach. We present *parallelised* model-checking algorithms for deadlock and livelock analysis and describe their implementation. The techniques are found to scale well running either on a conventional supercomputer or on a PC cluster.

1 Introduction

In this paper we look at parallelising the algorithms that are used for model checking concurrent systems. The first algorithm that we consider is deadlock-checking by performing an exhaustive breadth-first-search over all global states, looking for a deadlock-state. The technique used is fairly simple: global states are distributed between processors using a hash function. However this technique turns out to work very well and we shall demonstrate that the parallel algorithm is highly scalable.

Next we consider the problem of checking for livelock. A livelock is characterised by a cycle of hidden events in the global transition system under consideration. Normally depth-first-search is used to solve this problem, but unfortunately nobody knows any way to parallelise the depth-first-search effectively. So we present an alternative graph-pruning algorithm for livelock-analysis, which has the potential to be parallelised.

2 A simple deadlock checking algorithm

We consider a transition system defined in terms of an initial state, *initial*, of abstract type *State*, and a function $successors : States \rightarrow \mathcal{P}States$, which calculates the set of states reachable by performing a single transition from a given state. Our task is to determine whether the transition system contains any state s from which no further progress can be made, i.e. $successors(s) = \{\}$.

The standard approach to solving this problem is to maintain two lists of states: *pending* and *checked*. Initially *pending* contains only the initial state and *checked* is empty. Then the algorithm proceeds by repeatedly removing a state *x* from *pending*, from which the successor states are calculated. If there are no successors then *x* is a deadlock-state. Otherwise those successors of *x* that have not already been seen, are merged into *pending*. The search terminates either when a deadlock state is found or when *pending* is empty. This is summarised by the following pseudocode (in the style of Morgan[1]).

```
**var** $pending, checked$ : **set of** $State$.
$pending := \{initial\}$;
$checked := \{\}$;
**do** $(pending \neq \{\}) \longrightarrow$
   **var** $x : State$.
   # select a pending state to check
   $x :\in pending$;
   $pending := pending - \{x\}$;
   $checked := checked \cup \{x\}$;
   **if** $successors(x) = \{\}$ **then output** $x$; **stop else skip fi** ;
   $pending := pending \cup (successors(x) - checked)$
**od**
```

Although the basic algorithm is simple, there are a number of major implementation issues. First of all the *successors* function is non-trivial. Typically each state of the global transition system will consist of a *vector* of states from each individual component of the concurrent system. For instance figure 1 shows transition systems for a deadlock-free version of the famous Dining Philosophers network. In figure 2 we see how these are combined in parallel. Working out the successor states of a particular global state vector will involve looking up the possible transitions of each component process and applying the laws of CSP to see which transitions are globally enabled. The initial global state of the system is $\langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$: each component process is in its initial state. From that global state there are five possible transitions, each corresponding to a different philosopher picking up a fork. And each possible transition results in two of the component processes changing state, while all the rest remain in the same state.

There are also important data-refinement issues involved in efficient implementation of the potentially huge sets of global states. There may well be a requirement for using disk-based storage.

In the above deadlock-checking algorithm, the order in which states are removed from set *pending* has not been specified. Note that the search can be made strictly *depth-first* by choosing the most-recently discovered state, and *breadth-first* by choosing the least-recently discovered state.

Unless a strictly depth-first search (DFS) is required, we can decouple the discovery of new states from the process of inserting them into the pending pool. This is useful as it can be more efficient to insert states in large batches than to do so individually.

Here is a modified 'batch' version of the algorithm. A new variable *bucket* has been added to represent the buffer of newly discovered states. We assume the existence of a predicate

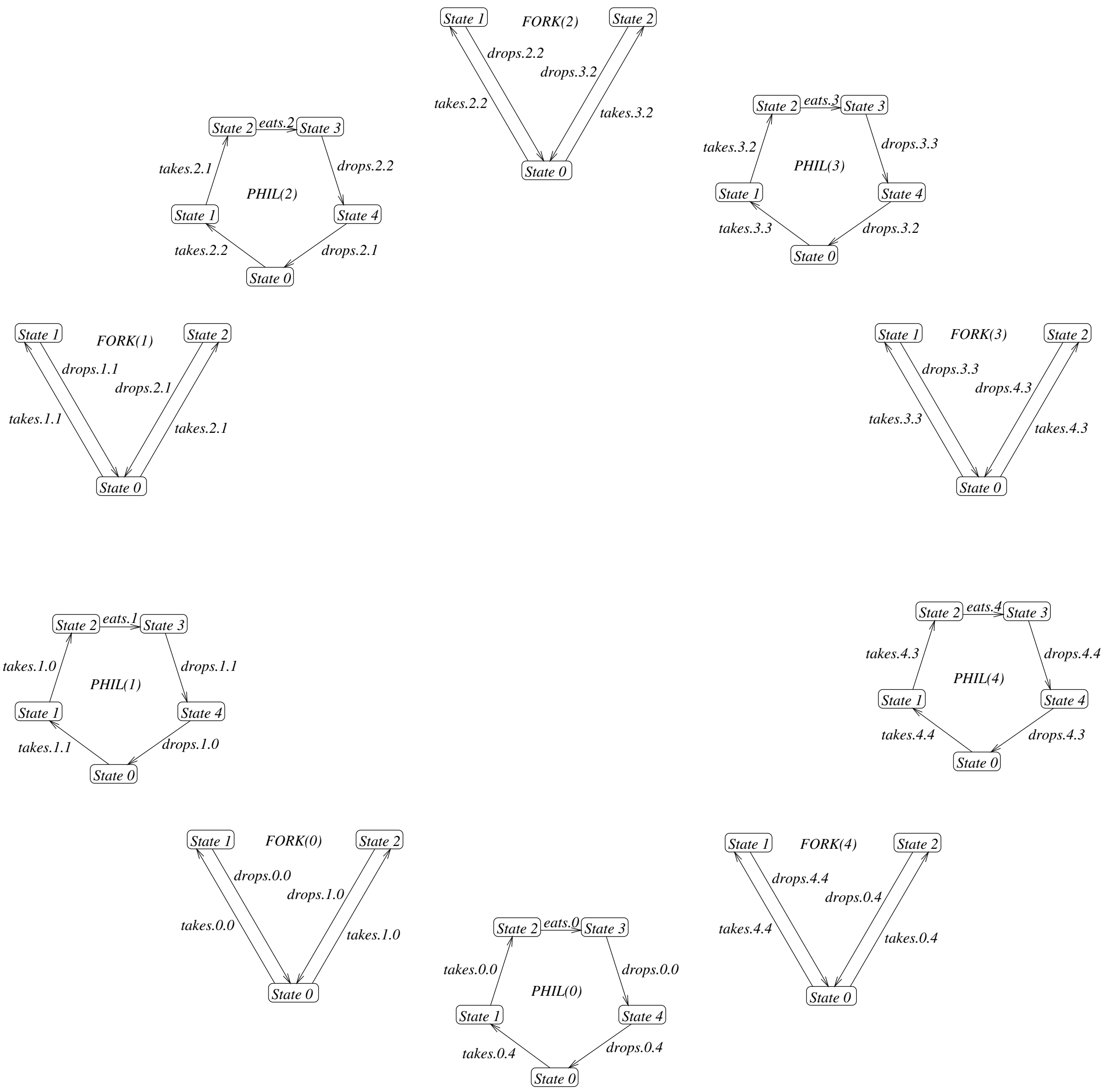


Figure 1: Individual transition systems for Dining Philophers Network

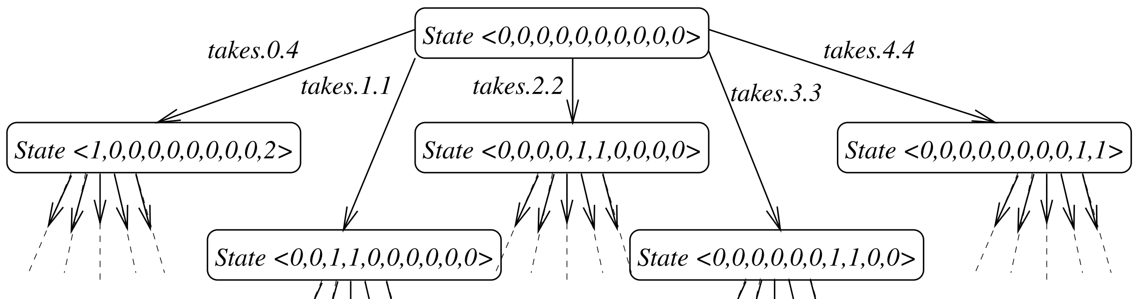

State <0,0,0,0,0,0,0,0,0,0>
takes.0.4
takes.1.1
takes.2.2
takes.3.3
takes.4.4
State <1,0,0,0,0,0,0,0,0,2>
State <0,0,1,1,0,0,0,0,0,0>
State <0,0,0,0,1,1,0,0,0,0>
State <0,0,0,0,0,0,1,1,0,0>
State <0,0,0,0,0,0,0,0,1,1>

Figure 2: Global transition system for Dining Philophers

full to indicate when this buffer is full.

```
var $pending, checked, bucket$ : set of $State$.
$pending := \{initial\}$;
$checked := \{\}$;
do $(pending \neq \{\}) \longrightarrow$
  # empty the bucket
  $bucket := \{\}$;
  do (not $full(bucket)$ and $pending \neq \{\}) \longrightarrow$
    var $x$ : $State$.
    $x :\in pending$;
    $pending := pending - \{x\}$;
    $checked := checked \cup \{x\}$;
    if $successors(x) = \{\}$ then output $x$; stop else skip fi ;
    # store successor states in the bucket
    $bucket := bucket \cup successors(x)$
  od ;
  # merge bucket into pending set
  $pending := pending \cup (bucket - checked)$
od
```

This is the algorithm that is used by the FDR tool for deadlock checking CSP programs.[2].

3 Parallelisation of the deadlock-checking algorithm

In order to parallelise the deadlock-checking algorithm we shall use the Bulk Synchronous Parallel computation model (BSP). Each processor runs the same code, and communication is by asynchronous remote-memory access. However periodic barrier-synchronisations may be applied, after which point a consistent global view of the data is guaranteed.

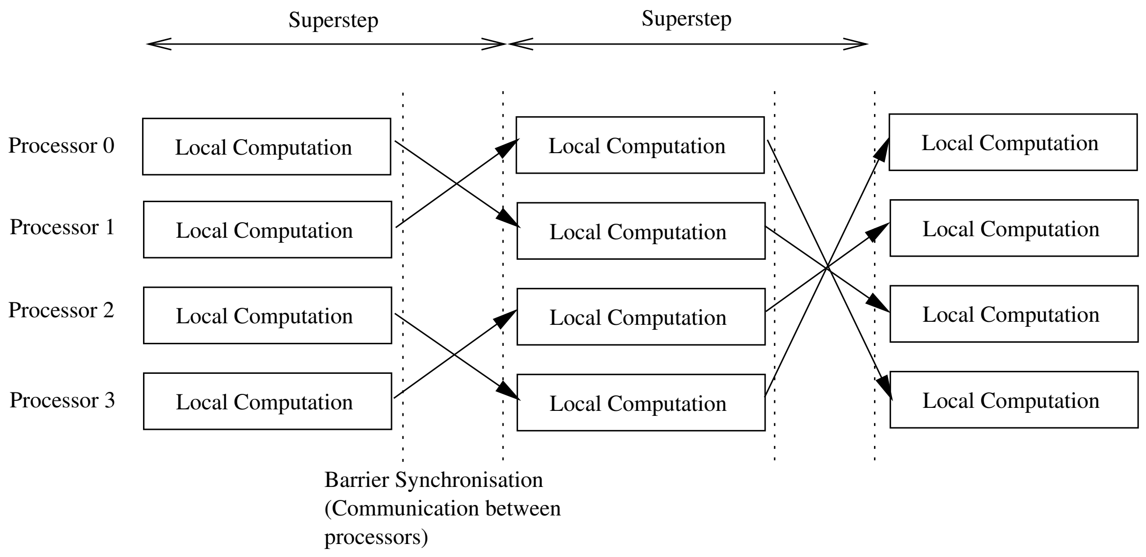


Figure 3: A BSP superstep

There are two key design decisions in our parallelisation strategy.

- The number of global states is likely to be large so we distribute them between processes,

- The number of states of individual processes is relatively small, and we need to access them quickly to calculate successor states, so we copy them to each processor.

The parallel model-checking algorithm depends on the existence of a 'hash' function *hash* which randomly and uniformly distributes states between N parallel processes. Each process i checks only those states y for which $hash(y) = i$. However, in the process of checking its own states, a process will probably discover new states that belong to other processes. So, rather than a single bucket of new states, an $N \times N$ array of buckets is maintained. States which have been discovered by process i but which actually belong to process j are stored temporarily in $bucket[i][j]$.

On each iteration every process works on its own data until either it has all been checked, or its N local buckets are collectively full. Then the buckets must be swapped around so that each process only possesses buckets containing its own states. This is achieved by a standard all-to-all communication[3], which has the effect of transposing the matrix of buckets. Finally a global 'reduction operation' is performed to discover the total number of pending states which remain in the system as a whole, which is used to decide whether the search is complete. (This is illustrated in figure 4.)

4 Implementation

Based on the above a parallel program has been developed for checking concurrent systems for deadlock-freedom. An important part of the implementation was to design efficient and compact data structures, and various bit-maps, compression techniques, and hash functions were used in order to extract good serial performance.

Each global state consists of a compressed vector of states of individual processes within the concurrent system being analysed, plus a pointer to a parent state which is used to reconstruct traces when deadlocks are discovered. The information required to calculate the successors of a particular state is stored in a number of hash-indexed lookup tables.

Parallelisation was achieved using the BSP communications library[4, 5]. The main issue was to choose a good hash function for distributing global states $\langle \sigma_1, .., \sigma_M \rangle$ between N processes, evenly and randomly. Each global state is bit-packed into several words $W_1, ..W_k$. The hash function chosen was

$$((37W_1 + 37^2W_2 + .. + 37^kW_k) modulo 31717) modulo N$$

This seems to work fine in practice.

Note that the data relating to the individual transition systems within the network (alphabet, states, and transitions) is duplicated on each parallel processor. This seems reasonable as we would expect the size of this data to be dwarfed by the eventual size of the global states of network V.

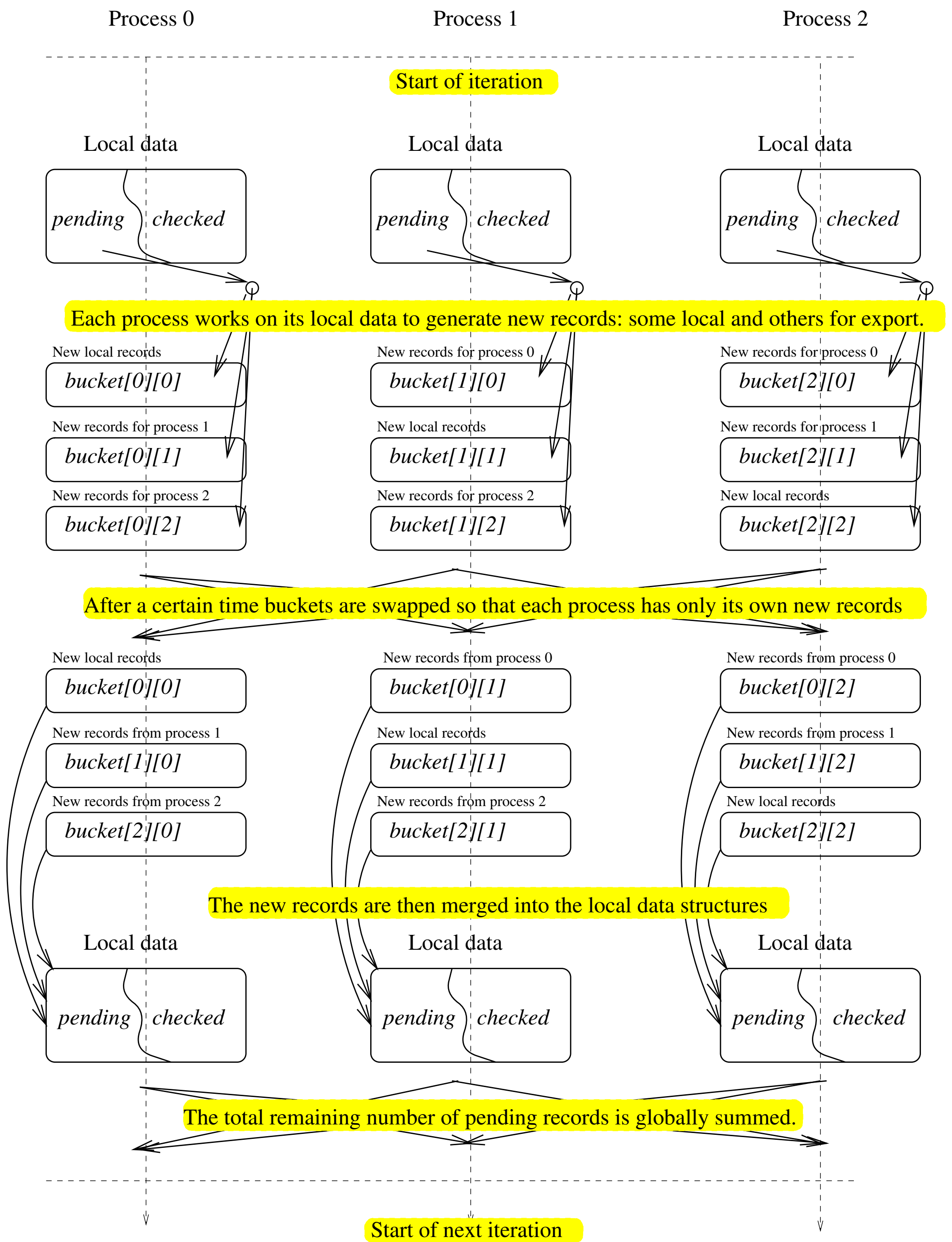

Process 0
Process 1
Process 2
Start of iteration
Local data
pending
checked
Each process works on its local data to generate new records: some local and others for export.
New local records
New records for process 0
New records for process 1
New records for process 2
bucket[0][0]
bucket[0][1]
bucket[0][2]
bucket[1][0]
bucket[1][1]
bucket[1][2]
bucket[2][0]
bucket[2][1]
bucket[2][2]
After a certain time buckets are swapped so that each process has only its own new records
New records from process 0
New records from process 1
New records from process 2
The new records are then merged into the local data structures
The total remaining number of pending records is globally summed.
Start of next iteration

Figure 4: The parallel deadlock-checking algorithm

Running the parallel program

The program that we have developed, called bspchecker, prints statistics about the distribution of global states after each superstep. Initially one BSP process holds the initial state and all the rest hold none.

```
oscar 107% ./bspchecker
 How many processors?
8
 Type data file name:
okphils13.dat
 Input data broadcasting complete
 Derived data structures complete
 Process:           0 hashing           0 global states.
 Process:           1 hashing           0 global states.
 Process:           2 hashing           0 global states.
 Process:           3 hashing           0 global states.
 Process:           4 hashing           1 global states.
 Process:           5 hashing           0 global states.
 Process:           6 hashing           0 global states.
 Process:           7 hashing           0 global states.
 Process:           0 now has           0 global states.
 Process:           1 now has           0 global states.
 Process:           2 now has           0 global states.
 Process:           3 now has           0 global states.
 Process:           4 now has           1 global states.
 Process:           5 now has           0 global states.
 Process:           6 now has           0 global states.
 Process:           7 now has           0 global states.
```

After a while a good load balance is achieved.

```
 Process:           0 hashing       40359 global states.
 Process:           1 hashing       39671 global states.
 Process:           2 hashing       40410 global states.
 Process:           3 hashing       39846 global states.
 Process:           4 hashing       39786 global states.
 Process:           5 hashing       40036 global states.
 Process:           6 hashing       39882 global states.
 Process:           7 hashing       40037 global states.
 Process:           0 now has      350025 global states.
 Process:           1 now has      350054 global states.
 Process:           2 now has      349919 global states.
 Process:           3 now has      349774 global states.
 Process:           4 now has      350130 global states.
 Process:           5 now has      350016 global states.
 Process:           6 now has      350321 global states.
 Process:           7 now has      349572 global states.
```

Finally the program prints out the any deadlock trace that has been discovered, or, if none exist, the total number of global states and transitions,

```
 Deadlock free:  5564522 global states 46200973 transitions
```

Timing results

The above Dining Philosphers network, expanded to 13 philosophers and forks, was used as input data to evaluate the efficiency of the program. Running on a single processor our program was found to outperform FDR by a factor of around 4. Timing results for parallel execution follow. We checked 13 dining philosophers on a MIPS R10000 processor SGI Origin computer. The total number of global states was 5,564,522 and the number of global transitions was 46,200,973.

Number of processors	Execution time(s)	Speedup	Parallel efficiency
1	478.4	1.0	100.0%
2	249.0	1.92	96%
4	128.7	3.72	93%
8	66.34	7.21	90%
16	40.52	11.81	74%
32	23.40	20.44	64%
48	18.40	26.0	54.2%

These figures indicate a near-linear speedup factor using up to 48 processes. But this is highly dependent on the communication characteristics of the machine being used. BSP profiling can be used to predict performance on other architectures[6, 7]. We have observed similar scalability and performance on a low-cost network of PCs, connected by switched fast ethernet.

5 A new livelock checking algorithm

The technique that we have used for parallelising deadlock-analysis can be applied to various other forms of model-checking. It has already been adapted to check refinement between two processes in the CSP failures model. However livelock-analysis is more challenging.

Livelock is a state of a concurrent system from which an unbounded sequence of internal events might occur, without interaction with its environment. In a finite transition system the presence of such a state implies that there must exist a cycle of transitions of hidden events in the graph. Such events are generally referred to as τ events. Usually we would consider events that are shared between two or more processes to be hidden τ events and those events that are performed by a single process to be visible to the environment. So, for instance, in the case of the Dining Philosophers network we would take all the *takes* and *drops* events to be τ events, and just the *eats* events to be visible.

We define the τ-graph of a transition system to be its subgraph of τ transitions. (This may well be disconnected.) Thus the task of determining livelock-freedom of a concurrent system comes down to deciding whether its τ-graph is acyclic.

The standard approach to testing acyclicity is to use a depth-first-search (DFS), and this is certainly the approach used by FDR[2] and SPIN[8] in livelock-checking. However the DFS is essentially a serial algorithm: no known effective parallelisation exists[9]. So we need to use a different algorithm.

An alternative way of checking for cycles in a digraph is to use 'graph-pruning'[10]. The approach is simply to delete nodes from the graph which have no outgoing arcs until none remain. Once this process is complete either the entire graph will have been deleted, in which case it is acyclic, or the residual graph will contain a cycle.

This technique would appear to parallelisable in the same-way as the breadth-first search, as nodes may be pruned off in layers and each layer can be processed in parallel. Figure 5 illustrates the pruning algorithm being applied to the τ-graph for three dining philosophers.

However the DFS has a major advantage: it does not require that all the τ transitions of the graph be held in memory. We have come up with three possible strategies for avoiding this problem.

Strategy 1: incremental pruning

This is the simplest algorithm as described above: nodes with no outgoing arcs are removed from the graph until none remain. This can be done in stages as the transition graph is built up, to try to cut down the number of transitions that need to be stored at any time. The hope here is that the memory needed for the transitions and states during the process will be less than the space required for the total number of states. We investigated this for networks of Dining Philosophers and buffers. Figure 6 shows the build up of the number of transitions with the number of states with and without pruning, for networks of 8 buffers and 8 dining philosophers. Figure 7 shows the space that would be needed to hold states and transitions, pruning in stages as the graph is built up. From these graphs it is clear that this algorithm is feasible for certain networks: holding the transitions and pruning in stages does not need much more space (if any) than is necessary for states only.

Strategy 2: predecessors function

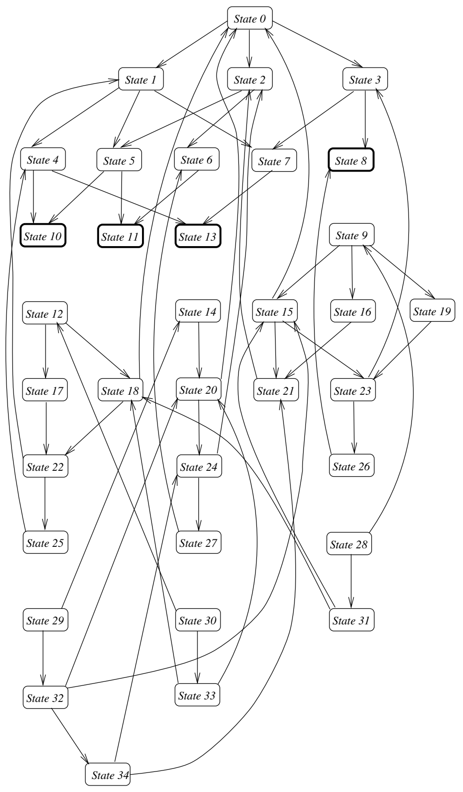
This version assumes the existence of a 'predecessors' function for calculating the immediate τ-predecessors of a state. A full breadth first search of the state space is constructed, and each state s is annotated with a τ-successor count $tcount(s)$. The pruning is then conducted by removing each state s such that $tcount(s) = 0$ from the graph, and decrementing the τ-successor count of each predecessor of s. The advantage of this version is that the transitions do not have to be stored - we only need one extra integer per state for the τ-successor count. The difficulty with this method is that the predecessor function may be complicated to calculate. Also, it is possible that the predecessor function may calculate some states that are 'imaginary' in that they cannot actually be reached from the initial state, which would affect the efficiency of the algorithm.

Strategy 3: pruning in reverse direction

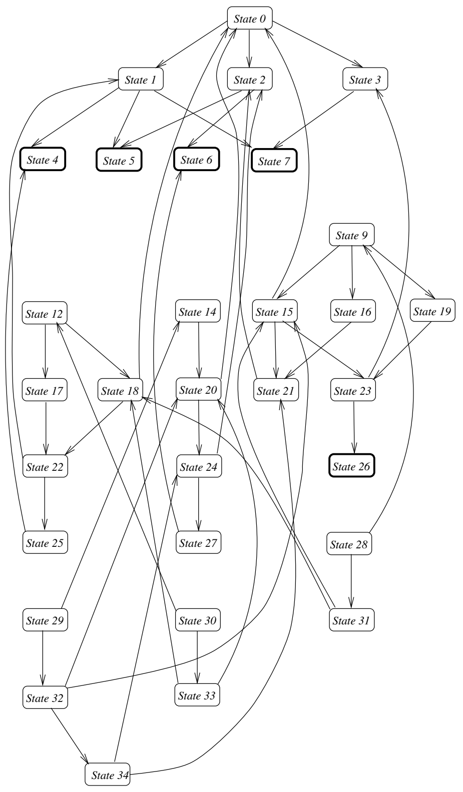
In this version there is no need to store the transitions and there is no complicated predecessor function. A complete breadth first search is conducted, annotating each state with a τ-*predecessor* count. Then for any state s with no τ-predecessors, s is removed from the graph and each of its successors has its τ-predecessor count decremented. This will leave some non-divergent states and remove some divergent states, but it will decide livelock freedom. This is the version that will is being coded as it is the simpler than strategy 2, and does not have the memory problems associated with strategy 1.

6 Conclusions and directions for further work

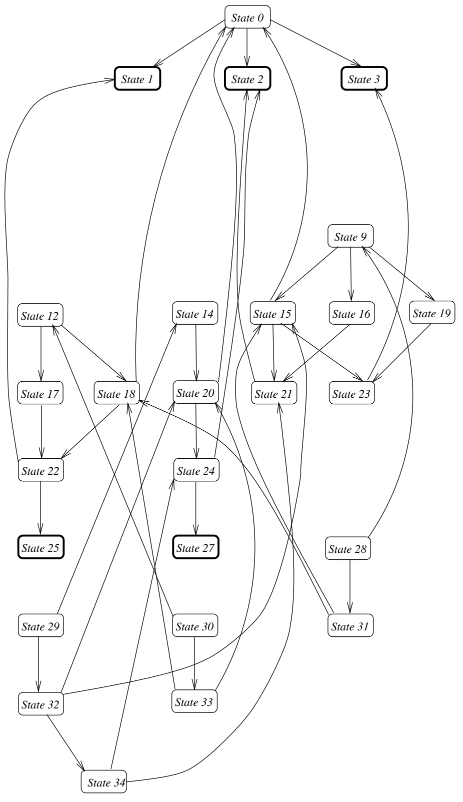
We have described how model-checking for deadlock analysis and livelock analysis may be effectively parallelised. The existing BFS algorithm used in deadlock-analysis was found to be suitable for parallelisation. However the DFS algorithm that is conventionally used for livelock analysis was not and so had to be discarded in favour of a graph-pruning algorithm.



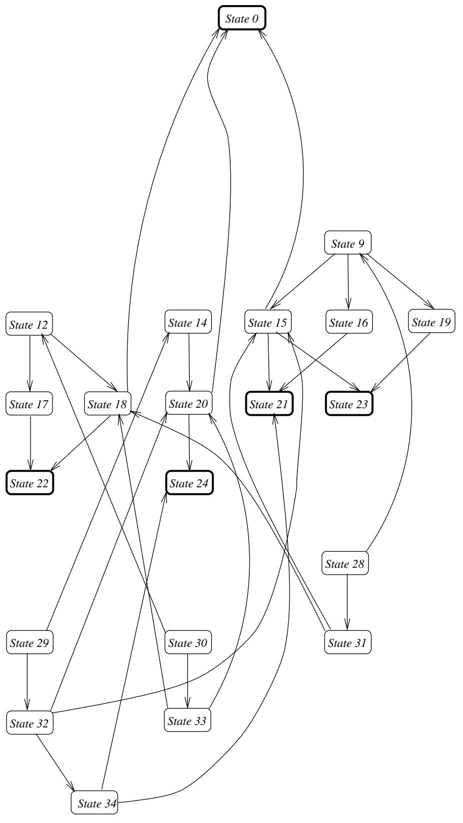
Level 1: prune states 8, 10, 11, 13



Level 2: prune states 4, 5, 6, 7, 26

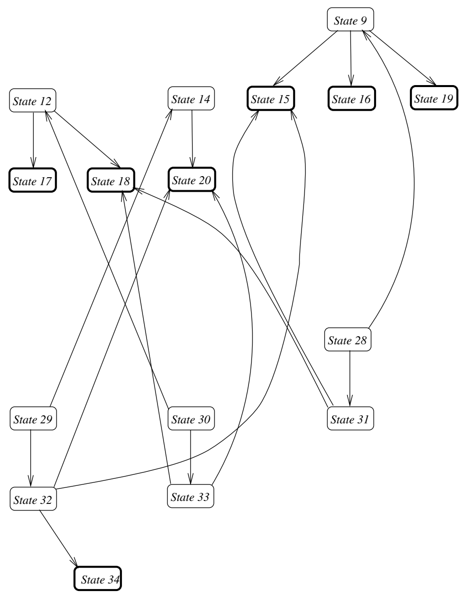


Level 3: prune states 1, 2, 3, 25, 27

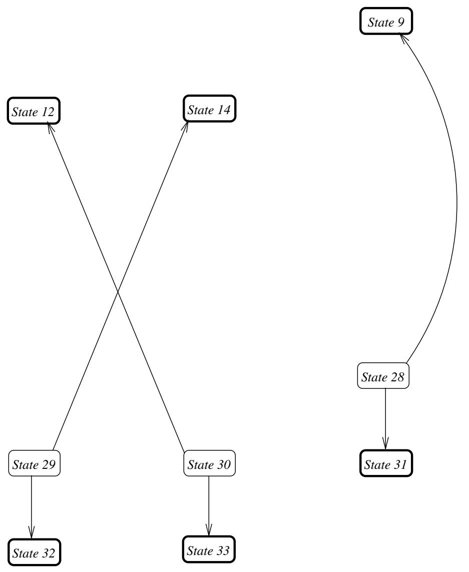


Level 4: prune states 0, 21, 22, 23, 24

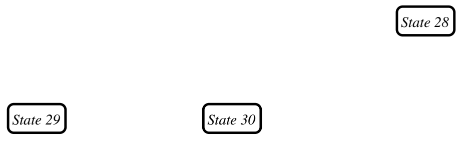
Figure 5: Pruning a τ-graph to detect divergent states



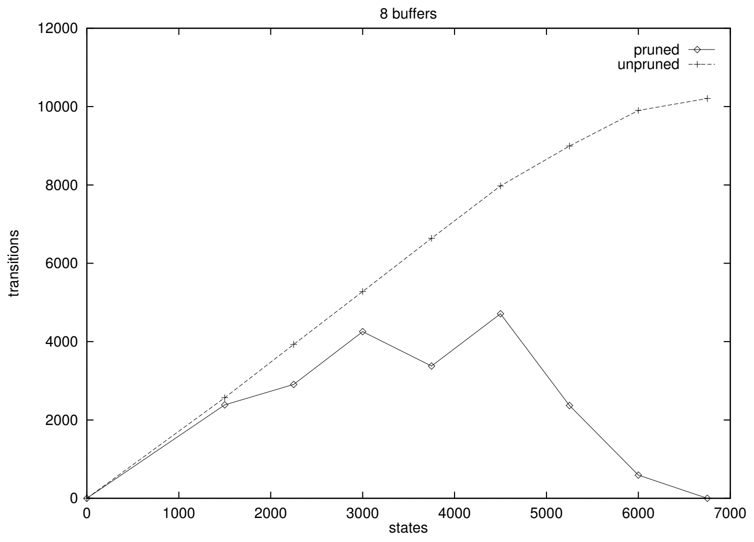
Level 5: prune states 15, 16, 17, 18, 19, 20, 34



Level 6: prune states 9, 12, 14, 31, 32, 33



Level 7: prune states 28, 29, 30



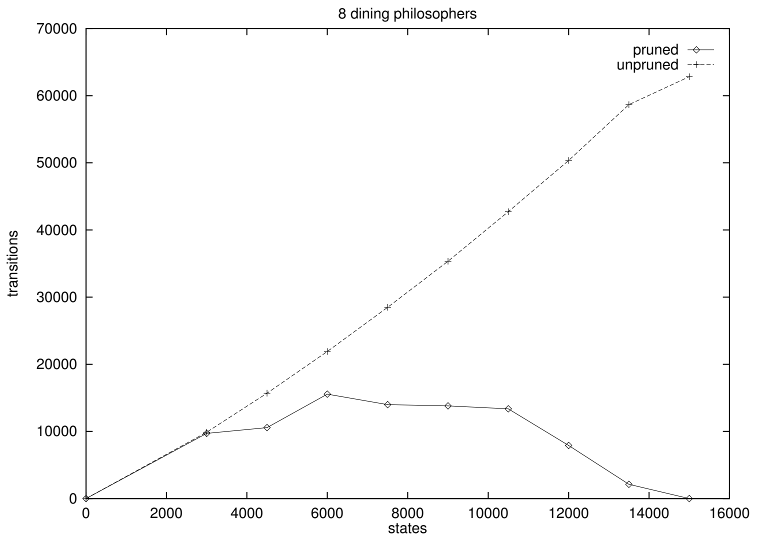
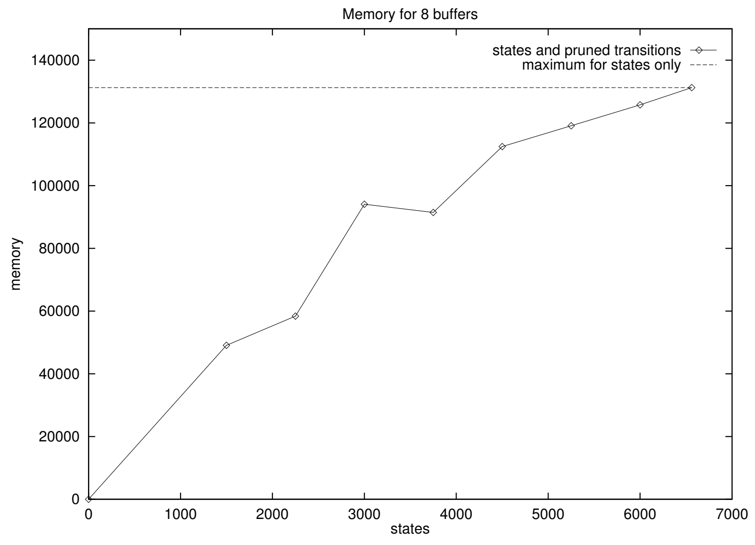


Figure 6: Size of τ-graph with or without incremental pruning



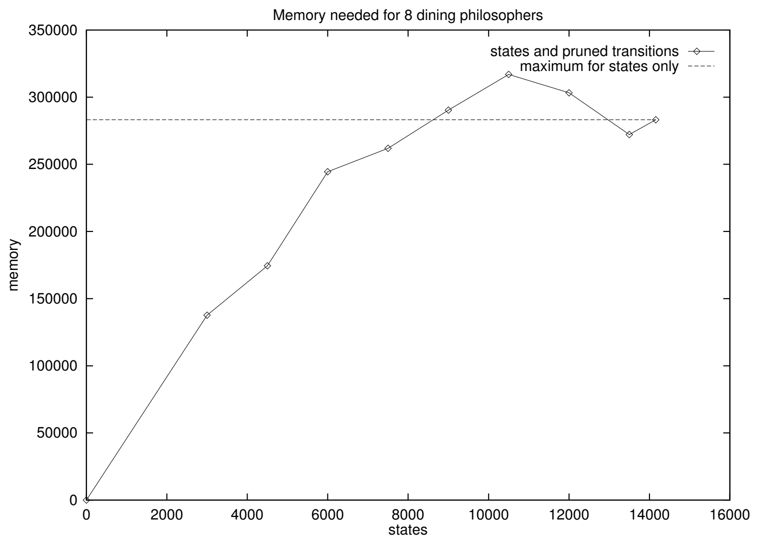


Figure 7: Memory requirements for the incremental pruning technique: the final memory value is the storage for states only

It is hoped that the techniques described in this paper will soon be directly applied to parallelise the FDR tool. A refinement check between two processes essentially requires a livelock analysis followed by a BFS of the combined state-spaces of the two processes.

This work will have major implications for the size of system that may be model checked in the future. At the time of writing the largest system to have ever been verified by FDR had around one hundred million states, and required a full week to perform on a powerful PC. Using sixty-four processors of an SGI Origin supercomputer we deadlock-checked six hundred million states in under an hour. However state-explosion will continue to be a major obstacle. Simple concurrent systems can easily have more states than there are atoms in the Universe. There will always be a need for short cuts.

References

[1] Carroll Morgan, *Programming from Specifications, Second Edition*, *Prentice-Hall*, 1994.

[2] A.W. Roscoe, *The Theory and Practice of Concurrency*, *Prentice-Hall*, 1998

[3] *MPI: A message passing interface*. Proc. Supercomputing '93. *IEEE Computer Society*, Message Passing Interface Forum series, pp878-883, 1993.

[4] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling, *BSPLib: The BSP Programming Library*, to appear in Parallel Computing.

[5] W.F. McColl, *Scalable Computing*. Computer Science Today: Recent Trends and Developments, ed: J. van Leeuwen. *Springer-Verlag*, 1995, Series LNCS, Volume 1000, pp46-61.

[6] Jonathan M. D. Hill, Paul I. Crumpton and David A. Burgess. *The theory, practice and a tool for BSP performance prediction*. EuroPar'96, LNCS *Springer-Verlag*, Volume 1124, pp697-705, August 1996.

[7] BSP Machine Parameters, see URL
`http://www.BSP-Worldwide.org/implmnts/oxtool.htm`

[8] G. J. Holzmann, *Design and Validation of Computer Protocols*, *Prentice-Hall*, 1991

[9] Lisa Fleischer, Bruce Hendrickson and Ali Pinar, *On Identifying Strongly Connected Components in Parallel*, Proc. Irregular'2000 Springer-Verlag LNCS 2000.

[10] S.Jassim and J.Martin, *Graph Techniques for Deadlock Analysis of Process Networks*, Proceedings of MFCS'98 Workshop on Communication 1998

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 1990

[12] C.A.R. Hoare, *Communicating Sequential Processes*, *Prentice-Hall*, 1984

[13] J. M. R. Martin, *The Design and Construction of Deadlock-Free Concurrent Systems*, University of Buckingham D. Phil thesis, 1996.

[14] J.M.R. Martin and S.A. Jassim, *A Tool for Proving Deadlock-Freedom*, Proceedings of the 20th World Occam and Transputer User Group Technical Meeting, IOS Press 1997

[15] A.W. Roscoe, *A Classical Mind*, *Prentice-Hall*, 1995