Concurrent Depth-First Search Algorithms

Gavin Lowe

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom
gavin.lowe@cs.ox.ac.uk

Abstract. We present concurrent algorithms, based on depth-first search, for three problems relevant to model checking: given a state graph, to find its strongly connected components, which states are in loops, and which states are in "lassos". Our algorithms typically exhibit about a four-fold speed-up over the corresponding sequential algorithms on an eight-core machine.

1 Introduction

In this paper we present concurrent versions of algorithms based on depth-first search, all variants of Tarjan's Algorithm [17]. We consider algorithms for three closely related problems:

1. To find the strongly connected components (SCCs) of a graph (i.e., the maximal subsets S of the graph's nodes such that for any pair of nodes $n, n' \in S$, there is a path from n to n');
2. To find which nodes are part of a cycle in the graph (i.e., such that there is a non-empty path from the node to itself);
3. To find which nodes are part of a "lasso" (i.e., such that there is a path from the node to a node on a cycle).

Our main interest in these algorithms is as part of the development of the FDR3 model checker [6,18] for CSP [16]. In order to carry out checks in the failures-divergences model, it is necessary to detect which nodes are divergent, i.e. can perform an unbounded number of internal τ events; this is equivalent to detecting whether the node is part of a lasso in the transition graph restricted to τ-transitions (Problem 3).

FDR's main failures-divergences refinement checking algorithm performs a concurrent breadth-first search of the product of the state graphs of the system and specification processes, testing whether each system state is compatible with the corresponding specification state. In particular, this involves testing whether the system state is divergent; hence several divergences tests need to be performed concurrently starting at different nodes.

Further, FDR can perform various compressions upon the transition graphs of processes. One of these, `tau_loop_factor`, works by identifying all nodes within an SCC in the transition graph restricted to τ-transitions (Problem 1).

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 202–216, 2014.
© Springer-Verlag Berlin Heidelberg 2014

Problem 2 has applications in other areas of model checking: the automata-theoretic approach for LTL model checking [19] involves searching for a cycle containing an accepting state in the graph formed as the product of the Büchi property automaton and the system.

We present concurrent algorithms for each of the above three problems. Our implementations typically exhibit about a four-fold speed-up over the corresponding sequential algorithms on an eight-core machine; the speed-ups are slightly better on graphs with a higher ratio of transitions to states.

These are challenging problems for the following reasons. In many graphs, threads will encounter nodes that are currently being considered by other threads; we need to ensure that the threads do not duplicate work, do not interfere with one another, but do obtain information from one another: depth-first search seems to be an area where it is difficult to achieve a high degree of independence between threads. Further, many graphs contain a *super-component* that contains a large proportion of the graph's nodes; for Problems 1 and 2, it seems impossible to avoid having the nodes of this super-component being considered sequentially.

In [14], Reif showed that computation of depth-first search post-ordering of vertices is P-complete. This is often used to claim that parallelising algorithms based on depth-first search is difficult (assuming $NC \neq P$): no algorithm can run in poly-logarithmic time with a polynomial number of processors. Nevertheless, it is possible to achieve significant speed-ups, at least for a fairly small number of processors (as is common in current computers), for the types of graphs that are typical of those encountered in model checking.

In Section 2 we review the sequential version of Tarjan's Algorithm. In Section 3 we present our concurrent algorithm. In Section 4 we describe some aspects of our prototype implementation, and highlight a few tricky aspects. In Section 5 we report on some experiments, comparing our algorithm to the sequential version. We sum up and discuss related work in Section 6.

2 Tarjan's Algorithm

In this section we review the sequential Tarjan's Algorithm [17]. We start by describing the original version, for finding SCCs; we then discuss how to adapt the algorithm to find loops or lassos.

Tarjan's Algorithm performs a depth-first search of the graph. The algorithm uses a stack, denoted tarjanStack, to store those nodes that have been encountered in the search but not yet placed into an SCC. Each node n is given two variables: index, which is a sequence counter, corresponding to the order in which nodes were encountered; and lowlink which records the smallest index of a node n' in the stack that is reachable via the descendents of n fully considered so far. The following function (presented in pseudo-Scala) to update a node's low-link will be useful.

```
def updateLowlink(update: Int) = { lowlink = min(lowlink, update) }
```

```
1  var index = 0
2  // Set node's index and lowlink, and add it to the stacks
3  def addNode(node) = {
4    node.index = index; node.lowlink = index; index += 1
5    controlStack.push(node); tarjanStack.push(node)
6  }
7  addNode(startNode)
8  while(controlStack.nonEmpty){
9    val node = controlStack.top
10   if(node has an unexplored edge to child){
11     if(child previously unseen) addNode(child)
12     else if(child is in tarjanStack) node.updateLowlink(child.index)
13     // otherwise, child is complete, nothing to do
14   }
15   else{ // backtrack from node
16     controlStack.pop
17     if(controlStack.nonEmpty) controlStack.top.updateLowlink(node.lowlink)
18     if(node.lowlink == node.index){
19       start new SCC
20       do{
21         w = tarjanStack.pop; add w to SCC; mark w as complete
22       } until(w == node)
23     }
24   }
25 }
```

Fig. 1. Sequential Tarjan's Algorithm

Also, each node has a status: either *complete* (when it has been placed in an SCC), *in-progress* (when it has been encountered but not yet been placed in an SCC), or *unseen* (when it has not yet been encountered).

Tarjan's Algorithm is normally described recursively; however, we consider here an iterative version. We prefer an iterative version for two reasons: (1) as is well known, iteration is normally more efficient than recursion; (2) when we move to a concurrent version, we will want to suspend searches; this will be easier with an iterative version. We use a second stack, denoted controlStack, that corresponds to the control stack of the recursive version, and keeps track of the nodes to backtrack to.

We present the sequential Tarjan's Algorithm for finding SCCs (Problem 1) in Figure 1. The search starts from the node startNode. When an edge is explored to a node that is already in the stack, the low-link of the edge's source is updated (line 12). Similarly, when the search backtracks, the next node's low-link is updated (line 17). On backtracking from a node, if its low-link equals its index, all the nodes above it on the Tarjan stack form an SCC, and so are removed from that stack and collected (lines 18–23).

The following observation will be useful later.

Observation 1. 1. For each node in the tarjanStack, there's a path in the graph to each subsequent node in the tarjanStack.
2. For any node n in the tarjanStack, if n is nearer the top of that stack than controlStack.top, then there is a path from n to controlStack.top (and hence the two nodes are in the same SCC).
3. If nodes n and l are such that n.lowlink = l.index, then all the nodes between n and l in the tarjanStack are in the same SCC.

If, instead, we are interested in finding cycles (Problem 2) then: (1) at line 12, if node == child then we mark the node as in a cycle; and (2) after line 22, if the SCC has more than one node, we mark all its nodes as in a cycle.

If we are interested in finding lassos (Problem 3) then: (1) at line 12, we immediately mark node and all the other nodes in the Tarjan stack as being in a lasso; and (2) if we encounter a complete node (line 13), if it is in a lasso, we mark all the nodes in the Tarjan stack as being in a lasso.

3 Concurrent Tarjan's Algorithm

We now describe our concurrent version of Tarjan's Algorithm. We again start with an algorithm for finding SCCs, presented in Figure 2; we later consider how to adapt this for the other problems.

Each search is independent, and has its own control stack and Tarjan stack. A search is started at an arbitrary node startNode that has not yet been considered by any other search (we describe this aspect of our implementation in Section 4.2). Each search proceeds much as in the standard Tarjan's Algorithm, as long as it does not encounter a node that is part of another current search. However, if the search encounters a node child that is not complete but is not in its own stack (line 13) —so it is necessarily in the stack of another search— then the search suspends (detailed below). When child is completed, the search can be resumed (line 24). This design means that each node is in the stacks of at most one search; each node has a field search identifying that search (set at line 5).

A difficulty occurs if suspending a search would create a cycle of searches, each blocked on the next. Clearly we need to take some action to ensure progress. We transfer the relevant nodes of those searches into a single search, and continue, thereby removing the blocking-cycle. We explain our procedure in more detail with an example, depicted in Figure 3; it should be clear how to generalise this example. The bottom-left of the figure depicts the graph G being searched; the top-left depicts the tarjanStacks of the searches (oriented downwards, so the "tops" of the stacks are towards the bottom of the page).

Suppose search s_1 is blocked at n_1 waiting for node c_2 of search s_2 to complete, because s_1 encountered an edge from n_1 to c_2 (corresponding to node and child, respectively, in Figure 2). Similarly, suppose search s_2 is blocked at n_2 waiting for node c_3 of search s_3 to complete; and search s_3 is blocked at n_3 waiting for node c_1 of search s_1 to complete. This creates a blocking cycle of suspended searches (see Figure 3, top-left). Note that the nodes between c_1 and n_1, between

```
1  var index = 0
2  // Set node's index, lowlink and search, and add it to the stacks
3  def addNode(node) = {
4    node.index = index; node.lowlink = index; index += 1
5    node.search = thisSearch; controlStack.push(node); tarjanStack.push(node)
6  }
7  addNode(StartNode)
8  while(controlStack.nonEmpty){
9    val node = controlStack.top
10   if(node has an unexplored edge to child){
11     if(child previously unseen) addNode(child)
12     else if(child is in tarjanStack) node.updateLowlink(child.index)
13     else if(child is not complete) // child is in-progress in a different search
14       suspend waiting for child to complete
15     // otherwise, child is complete, nothing to do
16   }
17   else{ // backtrack from node
18     controlStack.pop
19     if(controlStack.nonEmpty) controlStack.top.updateLowlink(node.lowlink)
20     if(node.lowlink == node.index){
21       start new SCC
22       do{
23         w = tarjanStack.pop; add w to SCC
24         mark w as complete and unblock any searches suspended on it
25       } until(w == node)
26     }
27   }
28 }
```

Fig. 2. Concurrent Tarjan's Algorithm (changes from Fig. 1 underlined)

c_2 and n_2, and between c_3 and n_3 are all in the same SCC, by Observation 1(1); we denote this SCC by "C".

Let t_1 be the top of the Tarjan stack of s_1: t_1 might equal n_1; or s_1 might have backtracked from t_1 to n_1. Note that all the nodes between n_1 and t_1 are in the same SCC as n_1, by Observation 1(2), and hence in the SCC C. Similarly, let t_2 and t_3 be the tops of the other Tarjan stacks; all the nodes between n_2 and t_2, and between n_3 and t_3 are likewise in C.

Let l_1 be the earliest node of s_1 known (according to the low-links of s_1) to be in the same SCC as c_1: l_1 is the earliest node reachable by following low-links from the nodes between c_1 to t_1 (inclusive), and then (perhaps) following subsequent low-links; equivalently, l_1 is the last node in s_1 that is no later than c_1 and such that all low-links of nodes between l_1 and t_1 are at least l_1 (a simple traversal of the Tarjan stack can identify l_1). Hence all the nodes from l_1 to t_1 are in the SCC C (by Observation 1(3)). Let l_2 and l_3 be similar.

Consider the graph G' formed by transforming the original graph by adding edges from n_1 to l_2, and from n_3 to l_1, as illustrated in Figure 3 (middle top

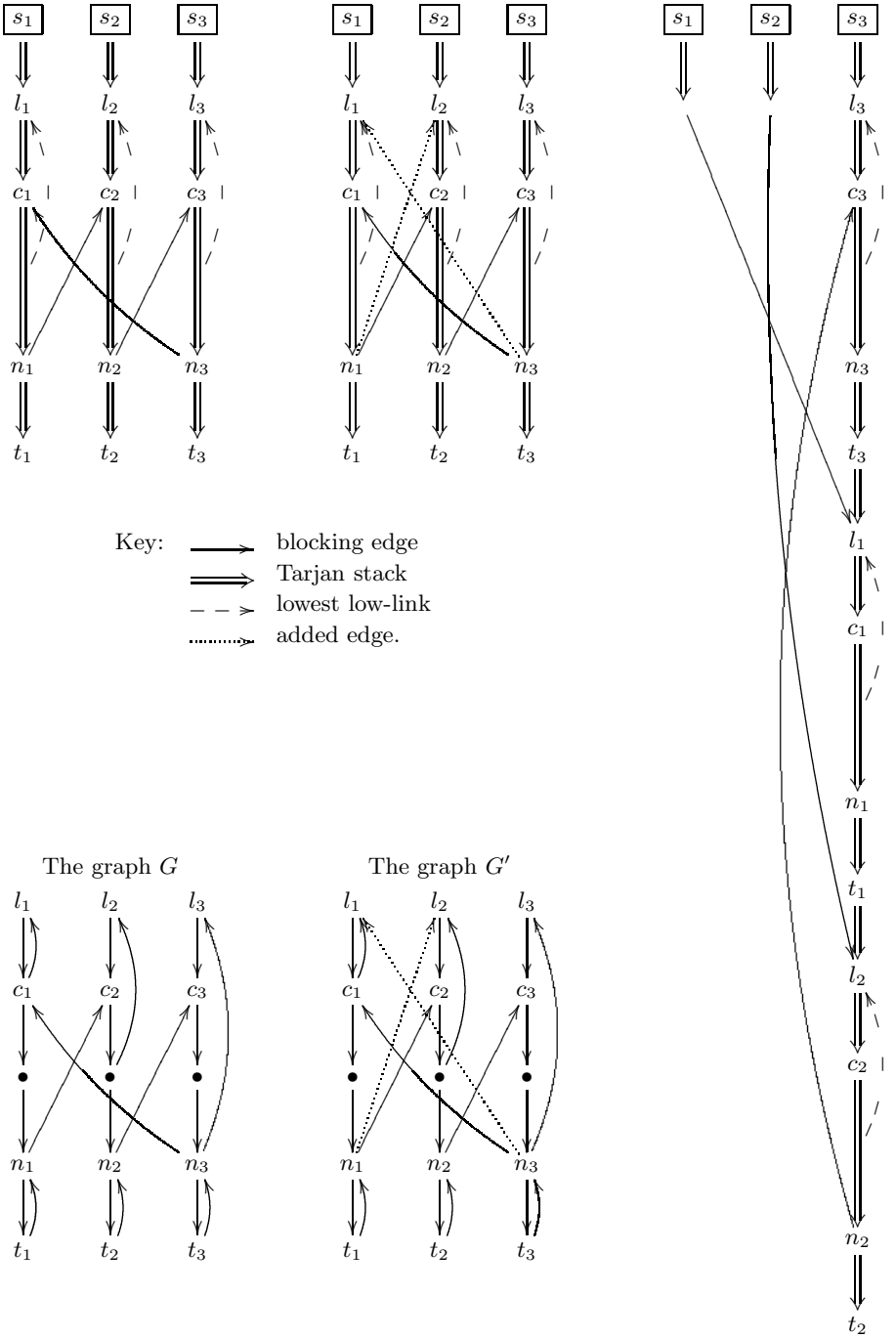


Fig. 3. Illustration of the blocking cycle reduction

and middle bottom). It is clear that the transformed graph has precisely the same SCCs as the original, since all the nodes below l_1, l_2 and l_3 in the figure are in the same SCC C. Consider the following scenario for the transformed graph: the search s_3 explores via nodes l_3, c_3, n_3 (backtracking from t_3), l_1, c_1, n_1 (backtracking from t_1), l_2, c_2, n_2 (backtracking from t_2), and then back to c_3; meanwhile, the searches s_1 and s_2 reach l_1 and l_2, respectively, and are suspended.

We transform the stacks to be compatible with this scenario, as illustrated in Figure 3 (right), by transferring the nodes from l_1 to n_1, and from l_2 to n_2 onto the stack of search s_3. Note, in particular, that the indexes and lowlinks have to be updated appropriately (letting $\delta_1 = s_3.\mathsf{index} - l_1.\mathsf{index}$, we add δ_1 onto the index and lowlink of each node transferred from s_1 and update $s_3.\mathsf{index}$ to be one larger than the greatest of the new indexes; we then repeat with s_2).

We then resume search s_3. We start by considering the edge from n_2 to c_3, and so update the lowlink of n_2. Searches s_1 and s_2 remain suspended until l_1 and l_2 are completed.

We now consider the other two problems. If we are interested in finding cycles (Problem 2) then we adapt the algorithm as for the sequential algorithm: (1) at line 12, if node == child then we mark the node as in a cycle; and (2) after line 25, if the SCC has more than one node, we mark all its nodes as in a cycle.

If we are interested in finding lassos (Problem 3) then we again adapt the algorithm as for the sequential algorithm: (1) at line 12, we immediately mark node and all the other nodes in the Tarjan stack as being in a lasso; and (2) if we encounter a complete node (line 15), if it is in a lasso, we mark all the nodes in the Tarjan stack as being in a lasso. Further, if a search encounters an in-progress node (line 13), if that node is in a lasso, then there is no need to suspend the search: instead all the nodes in the Tarjan stack can also be marked as in a lasso. Similarly, when a node is marked as being in a lasso, any search blocked on it can be unblocked; when such a search is unblocked, all the nodes in its Tarjan stack can also be marked as in a lasso. Finally, the procedure for reducing blocking cycles can be greatly simplified, using the observation that all the nodes in the Tarjan stacks are in a lasso: the search that discovered the cycle (s_3 in the example) marks all its nodes as in a lasso, and so unblocks the search blocked on it (s_2 in the example); that search similarly marks its nodes as in a lasso, and so on.

4 Implementation

In this section we give some details of our prototype implementation of the algorithm, and highlight a few areas where care is required. Our implementation[1] uses Scala [12].

[1] Available from `http://www.cs.ox.ac.uk/people/gavin.lowe/parallelDFS.html`.

4.1 Suspending and Resuming Searches

Each node n includes a field blocked : List[Search], storing the searches that have encountered this node and are blocked on it. When the node is completed, those searches can be resumed (line 24 of Figure 2). Note that testing whether n is complete (line 13 of Figure 2) and updating blocked has to be done atomically. In addition, each suspended search has a field waitingFor, storing the node it is waiting on.

We record which searches are blocked on which others in a map suspended from Search to Search, encapsulated in a Suspended object. The Suspended object has an operation suspend(s: Search, n: Node) to record that s is blocked on n.

When s suspends blocked by a node of s', we detect if this would create a blocking cycle by transitively following the suspended map to see if it includes a blocking path from s' to s. If so, nodes are transferred to s, and s is resumed as outlined in the previous section. This is delicate. Below, let s_b be one of the searches from which nodes are transferred and that remains blocked.

1. Each node's search, index and lowlink are updated, as described in the previous section.
2. Each s_b with remaining nodes has its waitingFor field updated to the appropriate node of s (the l_i nodes of Figure 3); and those nodes have their blocked fields updated.
3. The suspended map is updated: each s_b that has had *all* its nodes transferred is removed; each other s_b is now blocked by s; and any other search s' that was blocked on one of the nodes transferred to s is now also blocked on s.

The Suspended object acts as a potential bottleneck. Perhaps surprisingly, it is possible to allow several calls to suspend to proceed semi-concurrently. Considered as a graph, the suspended map forms a forest of reverse arborescences, i.e. a forest of trees, with all edges in a tree oriented towards a single sink search; further, only the sink searches are active. Hence, concurrent reductions of blocking cycles act on distinct reverse arborescences and so distinct searches.

We may not allow two concurrent attempts to detect a blocking cycle (consider the case where each of two searches is blocked on the other: the cycle will not be detected). Further, if no blocking cycle is found, the suspended map needs to be updated before another attempt to find a blocking cycle; and the suspended map must not be updated between reading the search field of the blocking node n and completing the search for a blocking cycle (to prevent n being transferred to a different search in the meantime)[2]. Finally, the suspended map itself must be thread-safe (we simply embed updates in **synchronized** blocks).

Other than as described in the previous paragraph, calls to suspend may act concurrently. In particular, suppose a call suspend(s,n) detects a blocking cycle. It updates search fields (item 1, above) *before* the suspended map (item 3). Suppose, further, a second call, suspend(s',n'), acts on the same reverse arborescence,

[2] We believe that the amount of locking here can be reduced; however, this locking does not seem to be a major bottleneck in practice.

and consider the case that n' is one of the nodes transferred to s. We argue that the resulting race is benign. The second call will not create a blocking cycle (since only the sink search of the reverse arborescence, s, can create a blocking cycle); this will be correctly detected, even in the half-updated state. Further, suspended(s') gets set correctly: if suspend(s',n') sets suspended(s') to n'.search before suspend(s,n) updates n'.search, then the latter will subsequently update suspended(s') to s (in item 3); if suspend(s,n) sets n'.search to s before suspend(s',n') reads it, then both will set suspended(s') to s.

4.2 Scheduling

Our implementation uses a number of worker threads (typically one per processor core), which execute searches. We use a Scheduler object to provide searches for workers, thereby implementing a form of task-based parallelism.

The Scheduler keeps track of searches that have been unblocked as a result of the blocking node becoming complete (line 24 of Figure 2). A dormant worker can resume one of these. (Note that when a search is unblocked, the update to the Scheduler is done *after* the updates to the search itself, so that it is not resumed in an inconsistent state.)

The algorithm can proceed in one of two different modes: *rooted*, where the search starts at a particular node, but the state space is not known in advance; and *unrooted*, where the state space is known in advance, and new searches can start at arbitrary nodes. In an unrooted search, the Scheduler keeps track of all nodes from which no search has been started. A dormant worker can start a new search at one of these (assuming it has not been reached by another search in the meantime). Similarly, in a rooted search the Scheduler keeps track of nodes encountered in the search but not yet expanded: when a search encounters a new node n, it passes n's previously unseen successors, except the one it will consider next, to the Scheduler. Again, a dormant worker can start a new search from such a node.

4.3 Enhancements

We now describe a few details of our implementation that have an effect upon efficiency.

We use a map from node identifiers (Ints) to Node objects that store information about nodes. We have experimented with many representations of this map. Our normal implementation is based on the hash table described by Laarman et al. in [7]. However, this implementation uses a fixed-size table, rather than resizing the table, thus going against the design of FDR (we have extended the hash table to allow resizing, but this makes the implementation somewhat slower). On some problems (including our experiments on random graphs in the next section), the implementation works better with a sharded hash table[3] with

[3] A sharded hash table can be thought of as a collection of M individual hash tables, each with its own lock; an entry with hash value h is stored in the table with index h mod M.

open addressing. Even with these implementation, the algorithms spend about 40% of their time within this map. (Other implementations are worse; using a Java ConcurrentHashMap increases the running time by a factor of two!)

It is clearly advantageous to avoid suspending searches, if possible. Therefore, the implementation tries to choose (at line 10 of Figure 2) a child node that is not in-progress in a different search, if one exists.

Some nodes have no successors. It is advantageous, when starting a search from such a node, to avoid creating a Search object with its associated stacks, but instead to just mark the node as complete and to create a singleton SCC containing it.

5 Experiments

In this section we report the results of timing experiments. The experiments were carried out on an eight-core machine (an Intel® Xeon® E5620) with 12GB of RAM. Each of the results is averaged over ten runs, after a warm-up round.

We have performed timing experiments on a suite of CSP files. We have extracted the graphs of τ-transitions for all implementation processes in the FDR3 test suite (including most of the CSP models from [15,16,1]) and the CSP models from [10]. The top of Figure 4 gives statistics about a selection of the graphs with between 200,000 and 5,000,000 states (we omit eleven such, in the interests of space), plus a slightly smaller file tring2.1 which we discuss below[4]. For each graph we give the number of states (i.e. nodes), the number of transitions (i.e. edges), the number of SCCs, the size of the largest SCC, the number of trivial SCCs (with a single state), the number of states on a loop, and the number of states on a lasso.

The bottom of Figure 4 gives corresponding timing results. For each of the three problems, we give times (in ms) for each of the concurrent and sequential algorithms, and the ratio between them (which represents the speed-up factor). The penultimate row gives totals for these running times, and their ratios. The final row gives data for tring2.1. Even on a single-threaded program, the JVM uses a fair amount of concurrency. The sequential algorithm typically uses about 160% of a single core (as measured by `top`). Hence the maximum speed-up one should expect is a factor of about five.

We have performed these experiments in unrooted mode, because it more-closely simulates our main intended use within FDR, namely for detecting divergences (i.e. τ-lassos) during failures-divergences checks. Such a check performs a breadth-first search of the product of the system and specification processes; for each pair of states encountered, if the specification state does not allow a divergence, then FDR checks that the system state does not have a divergence. The overall effect is normally that a lasso search is started at every reachable system state.

The concurrent algorithms normally give significant speed-ups. Further, the speed-up tends to be larger for larger graphs, particularly for graphs with more

[4] The file matmult.6 contains no τ-transitions, only visible transitions.

Graph	States	Transitions	SCCs	Largest SCC	Trivial SCCs	Loop states	Lasso states
cloudp.0	691692	1020880	691692	1	691692	0	0
cloudp.2	480984	643790	480984	1	480984	0	0
soldiers.0	714480	688110	714480	1	714480	0	0
comppuz.0	1235030	1558042	1235030	1	1235030	0	0
solitaire.0	494372	2271250	494372	1	494372	0	0
solitaire.1	4001297	5387623	4001297	1	4001297	0	0
matmul.6	2252800	0	2252800	1	2252800	0	0
virtroute.2	390625	1937500	390625	1	390625	0	0
tabp2.0	430254	310312	430254	1	430254	0	0
tabp2.1	427192	308978	427192	1	427192	0	0
tabp2.2	437908	316254	437908	1	437908	0	0
tringm.1	921403	925998	921403	1	921403	0	0
alt12.2.0	344221	1034608	344221	1	344221	0	0
alt12.2.1	344221	1114628	251927	2400	241821	102400	277229
alt12.3.0	575627	1283160	575627	1	575627	0	0
alt12.3.1	575627	1507604	447053	6560	439291	136336	440255
alt11.2.0	589149	1757856	589149	1	589149	0	0
alt11.2.1	589149	1883340	389713	1442	368629	220520	512131
alt11.3.0	990167	2227720	990167	1	990167	0	0
alt11.3.1	990167	2576732	652431	3168	628759	361408	886425
tring2.1	175363	355287	45822	129542	45821	131594	175363

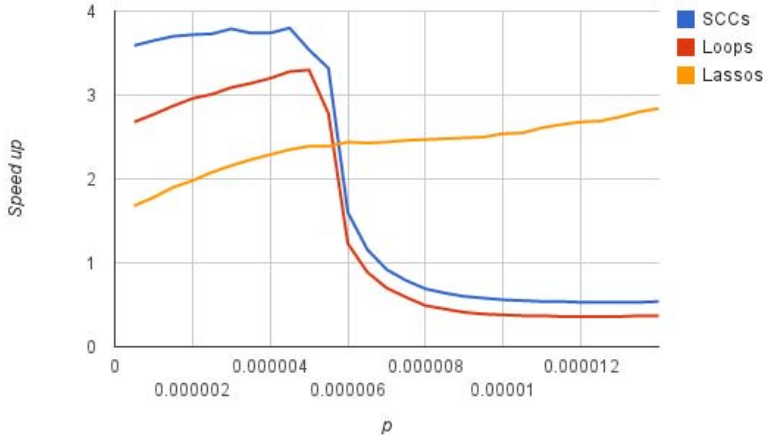
	SCCs			Loops			Lassos		
Graph	Conc	Seq	Ratio	Conc	Seq	Ratio	Conc	Seq	Ratio
cloudp.0	174	697	3.99	158	562	3.54	159	348	2.18
cloudp.2	127	414	3.24	120	317	2.63	118	167	1.42
soldiers.0	193	708	3.66	185	579	3.12	186	331	1.78
comppuz.0	334	1626	4.86	304	1399	4.59	305	990	3.24
solitaire.0	191	668	3.50	178	573	3.22	175	423	2.41
solitaire.1	1531	5058	3.30	1034	4396	4.25	1035	3220	3.11
matmul.6	543	2533	4.67	543	2161	3.97	543	1365	2.51
virtroute.2	138	457	3.31	129	381	2.95	129	261	2.03
tabp2.0	152	370	2.43	142	279	1.96	143	137	0.96
tabp2.1	153	370	2.41	141	278	1.97	144	137	0.95
tabp2.2	157	379	2.40	144	286	1.98	145	140	0.97
tringm.1	281	869	3.09	258	699	2.71	258	390	1.51
alt12.2.0	118	404	3.42	111	338	3.03	108	232	2.14
alt12.2.1	118	383	3.23	111	316	2.84	125	263	2.10
alt12.3.0	172	766	4.43	161	652	4.03	159	469	2.94
alt12.3.1	186	744	4.00	174	630	3.61	189	520	2.75
alt11.2.0	178	805	4.51	164	692	4.20	163	503	3.07
alt11.2.1	188	760	4.04	174	642	3.67	198	558	2.81
alt11.3.0	276	1226	4.44	256	1042	4.07	254	741	2.92
alt11.3.1	287	1178	4.10	274	983	3.58	315	845	2.68
Total	8505	34092	4.01	7554	28964	3.83	7816	21258	2.72
tring2.1	461	207	0.45	420	125	0.30	76	118	1.55

Fig. 4. Results for tests on CSP files: statistics about the graphs, and timing results

transitions. However, beyond a few million states, the speed-ups drop off again, I believe because of issues of memory contention.

The results for tring2.1 deserve comment. This graph has a large SCC, accounting for over 70% of the states. The first two concurrent algorithms consider the nodes of this SCC sequentially and so (because the concurrent algorithms are inevitably more complex) are slightly slower than the sequential algorithms. However, the algorithm for lassos gives more scope for considering the nodes of this SCC concurrently, and therefore gives a speed-up.

The above point is also illustrated in Figure 5. This figure considers a number of random graphs, each with $N = 200{,}000$ states. For each pair of nodes n and n', an edge is included from n to n' with probability p; this gives an expected number



p	SCCs	Largest SCCs	Loop states	Lasso states
0.0000005	200000	1	0	0
0.0000010	200000	1	0	1
0.0000015	200000	1	0	0
0.0000020	200000	1	1	1
0.0000025	200000	1	1	2
0.0000030	200000	1	1	3
0.0000035	199999	2	2	7
0.0000040	199998	3	4	20
0.0000045	199994	6	8	120
0.0000050	199908	72	96	2573
0.0000055	194020	5973	5984	34743
0.0000060	179848	20149	20155	63368
0.0000065	164038	35959	35966	84903
0.0000070	147928	52071	52075	101907

p	SCCs	Largest SCCs	Loop states	Lasso states
0.0000075	131836	68165	68167	116882
0.0000080	117613	82387	82390	128340
0.0000085	104296	95704	95706	138443
0.0000090	92685	107316	107318	146591
0.0000095	82651	117350	117351	153129
0.0000100	73020	126981	126982	159418
0.0000105	64830	135171	135171	164467
0.0000110	57715	142286	142287	168734
0.0000115	51080	148921	148922	172610
0.0000120	45565	154436	154437	175740
0.0000125	40710	159291	159292	178494
0.0000130	36322	163679	163679	180849
0.0000135	32422	167579	167580	183109
0.0000140	28929	171072	171072	184999

Fig. 5. Experiments on random graphs

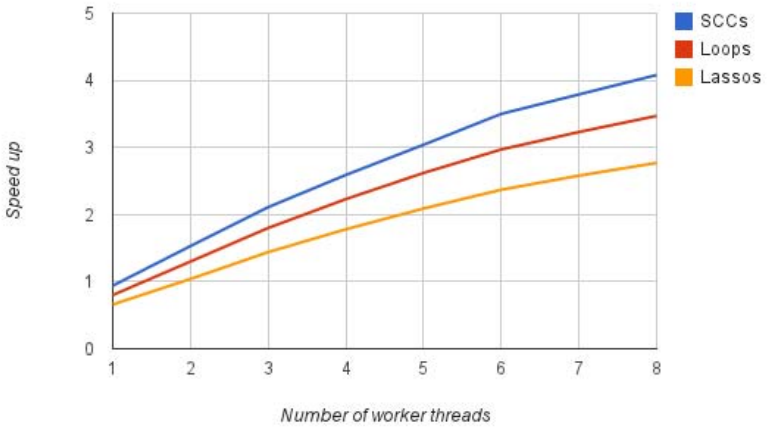


Fig. 6. Speed ups on CSP files as a function of the number of worker threads

of edges equal to N^2p. (Note that such graphs do not share many characteristics with the graphs one typically model checks!) The graph plots the speed-up for the three algorithms for various values of p; the tables give statistical information about the graphs considered (giving averages, rounded to the nearest integer in each case). For p greater than about 0.000005, the graph has a large SCC, and the algorithms for SCCs and loops become less efficient. However, the algorithm for finding lassos becomes progressively comparatively more efficient as p, and hence the number of edges, increases; indeed, for higher values of p, the speed-up plateaus at about 5.

It is worth noting that graphs corresponding to the τ-transitions of CSP processes rarely have very large SCCs. The graph tring2.1 corresponds to a CSP process designed for checking in the traces model, as opposed to the failures-divergences model, so the problems considered in this paper are not directly relevant to it.

Figure 6 considers how the speed up varies as a function of the number of worker threads. It suggests that the algorithm scales well.

6 Conclusions

In this paper we have presented three concurrent algorithms for related problems: finding SCCs, loops and lassos in a graph. The algorithms give appreciable speed-ups, typically by a factor of about four on an eight-core machine.

It is not surprising that we fall short of a speed-up equal to the number of cores. As noted above, the JVM uses a fair amount of concurrency even on single-threaded programs. Also, the concurrent algorithms are inevitably more complex than the sequential ones. Further, I believe that they are slowed down by contention for the memory bus, because the algorithms frequently need to read data from RAM.

I believe there is some scope for reducing the memory contention, in particular by reducing the size of Node objects: many of the attributes of Nodes are necessary only for *in-progress* nodes, so could be stored in the relevant Search object. Further, I intend to investigate whether it's possible to reduce the amount of locking of objects done by the prototype implementation.

We intend to incorporate the lasso and SCC algorithms into the FDR3 model checker. In particular, it will be interesting to see whether the low-level nature of C++ (in which FDR3 is implemented) permits optimisations that give better memory behaviour.

As noted earlier, a large proportion of the algorithms' time is spent within the map storing information about nodes. I would like to experiment with different implementations.

Related Work. We briefly discuss here some other concurrent algorithms addressing one or more of our three problems. We leave an experimental comparison with these algorithms for future work.

Gazit and Miller [5] describe an algorithm based upon the following idea. The basic step is to choose an arbitrary *pivot* node, and calculate its SCC as the intersection of its descendents and ancestors; these descendents and ancestors can be calculated using standard concurrent algorithms. This basic step is repeated with a new pivot whose SCC has not been identified, until all SCCs are identified. A number of improvements to this algorithm have been proposed [13,11,2].

Several papers have proposed algorithms for finding loops, in the particular context of LTL model checking [8,4,9,3]. These algorithms are based on the SWARM technique: multiple worker threads perform semi-independent searches of the graph, performing a nested depth-first search to detect a loop containing an accepting state; the workers share only information on whether a node has been fully explored, and whether it has been considered within an inner depth-first search.

Acknowledgements. I would like to thank Tom Gibson-Robinson for many interesting and useful discussions that contributed to this paper. I would also like to thank the anonymous referees for their useful comments.

References

1. Armstrong, P., Lowe, G., Ouaknine, J., Roscoe, A.W.: Model checking timed CSP. In: Proceedings of HOWARD-60 (2012)
2. Barnat, J., Chaloupka, J., van de Pol, J.: Distributed algorithms for SCC decomposition. Journal of Logic and Computation 21(1), 23–44 (2011)
3. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
4. Evangelista, S., Petrucci, L., Youcef, S.: Parallel nested depth-first searches for LTL model checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 381–396. Springer, Heidelberg (2011)

5. Fleischer, L.K., Hendrickson, B.A., Pinar, A.: On identifying strongly connected components in parallel. In: Rolim, J.D.P. (ed.) IPDPS-WS 2000. LNCS, vol. 1800, pp. 505–511. Springer, Heidelberg (2000)
6. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — A modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 180–194. Springer, Heidelberg (2014)
7. Laarman, A., van de Pol, J., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010 (2010)
8. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core nested depth-first search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
9. Laarman, A.W., van de Pol, J.C.: Variations on multi-core nested depth-first search. In: Proceedings of the 10th International Workshop on Parallel and Distributed Methods in Verification. Electronic Proceedings in Theoretical Computer Science, vol. 72, pp. 13–28 (2011)
10. Lowe, G.: Implementing generalised alt: A case study in validated design using CSP. In: Communicating Process Architectures (2011)
11. McLendon III, W., Hendrickson, B., Plimpton, S.J., Rauchwerger, L.: Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing 65(8), 901–910 (2005)
12. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima (2008)
13. Orzan, S.: On Distributed Verification and Verified Distribution. PhD thesis, Free University of Amsterdam (2004)
14. Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters 20(5), 229–234 (1985)
15. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice Hall (1998)
16. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)
17. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2), 146–160 (1972)
18. University of Oxford. Failures-Divergence Refinement—FDR 3 User Manual (2013), `http://www.cs.ox.ac.uk/projects/fdr/manual/index.html`
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of Logic in Computer Science (1986)