Parallel Nested Depth-First Searches for LTL Model Checking

Sami Evangelista, Laure Petrucci, and Samir Youcef

LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
firstname.lastname@lipn.univ-paris13.fr

Abstract. Even though the well-known nested-depth first search algorithm for LTL model checking provides good performance, it cannot benefit from the recent advent of multi-core computers. This paper proposes a new version of this algorithm, adapted to multi-core architectures with a shared memory. It can exhibit good speed-ups as supported by a series of experiments.

1 Introduction

The model checking problem aims at verifying whether a given hardware or software system meets its specification. For the analysis of properties expressed in the Linear-time Temporal Logic (LTL) this problem is often reduced to checking the emptiness of a Büchi automaton defined as the product of the system and an automaton negating the formula to check [26]. Thus, model checking boils down to find a cycle in a directed graph, and more precisely, to verify the existence of an *accepting cycle*. The latter is defined as a cycle (in the sense of graph theory) containing at least one accepting state.

This problem has been intensively explored because of its importance, using diverse techniques. In the context of *explicit-state* model checking, algorithms usually rely on depth-first-search (DFS) strategies allowing to check for Büchi emptiness in linear time. They are split in two main families: *Nested* DFS (ndfs), originally proposed by Courcoubetis et al [11], consist of two procedures where the first one allows to find and sort the accepting states while the second one, interleaved with the first one, searches for cycles containing these states ; SCC (*strongly-connected components*) based algorithms [12, 17] exploit the fact that a counter-example exists if and only if a strongly connected component containing an accepting state is reachable from the initial state.

Despite the existence of algorithms with linear complexity for this emptiness check, combinatorial aspects remain due to the state space size of real systems, their exact analysis often being intractable. However, recent hardware developments, such as 64-bits technologies, contribute to harnessing formal verification memory limitations. Hence, the problem we can now often face is a "time explosion" rather than a lack of memory. For instance, using aggressive memory reduction techniques [21] one can hope to analyse state space graphs with e.g. 10^{10}–10^{11} states. Even with the fastest tools available, such as SPIN [19], a full exploration of such a graph would require weeks.

T. Bultan and P.-A. Hsiung (Eds.): ATVA 2011, LNCS 6996, pp. 381–396, 2011.

The use of parallel search algorithms can naturally leverage this time explosion. Most algorithms of this category were initially designed for distributed-memory architectures [1,5,7,8,9], fostered by easy access to networks of workstations. The availability of multi-core chips on desktop computers now offers opportunities to speed up tasks execution and also for the development of new approaches to model checking [3, 22].

Our contribution is a parallel algorithm designed for shared memory and multi-core architectures: Multi-Core ndfs (mc-ndfs). It solves the Büchi emptiness problem by launching multiple instances of ndfs. The use of both randomisation and synchronisations allows, to some extent, to force processes to visit different parts of the graph and to avoid, as much as possible, multiple revisits of a same state. Thus, even if our algorithm is theoretically not scalable it provides significant speed-ups for many case studies as attested by a wide range of experiments.

The paper is organised as follows. Section 2 presents related works: the well-known ndfs algorithm is recalled and existing parallel algorithms for LTL model checking summarised after outlining the accepting cycle detection problem. Section 3 details the proposed algorithm and gives its formal proof. Section 4 presents experimental results. Our work is concluded by Section 5 that also gives some perspectives for future work.

2 Background

In order to facilitate the understanding of our algorithm and its comparison with algorithms from the literature, we begin with a brief state of the art: LTL model checking, some algorithms based on ndfs, and parallel algorithms for LTL model checking.

2.1 The LTL Model Checking Problem

This paper addresses LTL model checking of finite-state systems where both the systems and their properties are modelled as automata. Then, verification is often reduced to checking the emptiness of a Büchi automaton defined as the product of the system and the negated formula [26]. This problem can be stated in its basic form as follows:

Definition 1 (Synchronised graph). *A* synchronised graph *is a tuple* $\mathcal{G} = (\mathcal{S}, \mathcal{T}, \mathcal{A}, s_0)$*, where* $\mathcal{S}$ *is a finite set of states;* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *is a set of transitions;* $\mathcal{A} \subseteq \mathcal{S}$ *is the set of accepting states, and* $s_0 \in \mathcal{S}$ *is an initial state.*

The set of *successors* of $s \in \mathcal{S}$ is denoted by $succ(s) = \{s' | (s,s') \in \mathcal{S}\}$. A *path* is a sequence of states $s_1 \dots s_k$ with $(s_i, s_{i+1}) \in \mathcal{T}$ for all $i \in \{1, \dots, k-1\}$ denoted by $s_1 \rightsquigarrow s_k$. A *cycle* is a path with $s_1 = s_k$. An *accepting cycle* is a cycle that contains at least one state $a \in \mathcal{A}$. An *accepting run* is a path from s_0 to s_l through s_k where $s_k \dots s_l$ form an accepting cycle. The *accepting cycle detection problem* aims at determining if a given graph $\mathcal{G}$ contains an accepting cycle. The major algorithms addressing this problem are based either on nested DFS (ndfs) or on SCCs (originating from Tarjan's algorithm for decomposing the graph into strongly connected components). Since the algorithm proposed in this paper is essentially based on ndfs, we shall focus on this one only. Details on SCC-based algorithms can be found elsewhere [12, 17].

2.2 Algorithms Based on Nested Depth-First Search

The well-known nested-depth first search algorithm for LTL model checking, was initially introduced in [11]. All algorithms belonging to this category still follow the same scheme. The ndfs algorithm (see Algorithm 1.1) is defined by two procedures called *dfsBlue* and *dfsRed*. The first one, which is the main loop, allows for marking each newly visited state as blue. The second one tries to find a loop back to a given accepting state s, and marks all encountered states as red. If a cycle is detected then a counterexample is reported, otherwise the first DFS continues and the red markings remain. Note that each DFS visits each state at most once and requires one bit per state. Procedure *dfsBlue* performs a depth-first search and sets the blue bits of all visited states. Procedure *dfsRed* is invoked when the search from an accepting state s finishes. Finally, if *dfsRed* finds that some accepting state s can be reached from itself, an accepting cycle is returned, otherwise the graph does not contain any cycle.

Since its introduction, several improvements have been proposed. Some aim at reporting accepting runs faster [13, 14, 15, 18] while others [16] focus on the length of counter-examples. Nested DFS is now implemented by a large range of explicit state model checkers among which SPIN [19] was historically the first.

2.3 Parallel Algorithms for LTL Model Checking

The best known enumerative *sequential* algorithms in the area of LTL model checking are Nested DFS and SCC-based algorithms. Adapting them to take advantage of parallel architectures is difficult since they rely on inherently sequential depth-first search postorder. Hence, it is necessary to propose new techniques and algorithms. Before getting into the details of the proposed algorithm, seven existing parallel algorithms are outlined: *Maximal Accepting Predecessor* (map), *One Way Catch Them Young* (owcty), *One Way Catch Them Young On-The-Fly* (owcty-otf), *Negative Cycle* (negc), *Back-Level Edges* (bledge), *Back-Level Edges On-The-Fly* (bledge-otf), and SPIN's double DFS (2-ndfs). All except the last one have been initially designed for distributed memory architectures, but it is well known that they can easily be transformed into shared memory algorithms. To compare the different complexities of these algorithms the following notations are used: $n = |\mathcal{S}|$, $m = |\mathcal{T}|$, $a = |\mathcal{A}|$, $p =$ number of working processes and h (height) is the smallest integer s.t. s_0 can reach all states using at most h transitions.

Algorithm map [8] uses an order relation on states to compute the maximal accepting predecessor function *map* mapping each state s to the identity of the greatest accepting state that is backward reachable from s.

Algorithm 1.1. The ndfs algorithm adapted from [11]

```
1  procedure ndfs(s) is
2      initialise all flags to false
3      dfsBlue(s_0)
4      if ¬ cycle reported then
5          report no-cycle
6  procedure dfsBlue(s) is
7      s.blue := true
8      for s' ∈ succ(s) do
9          if ¬s'.blue then
10             dfsBlue(s')
11     if s ∈ A then
12         seed := s
13         dfsRed(s)
14 procedure dfsRed(s) is
15     s.red := true
16     for s' ∈ succ(s) do
17         if s' = seed then
18             report cycle
19         else if ¬s'.red then
20             dfsRed(s')
```

The key idea behind algorithm owcty is to repeatedly remove from the graph states that cannot lead to an accepting cycle [9], according to two rules: a state s can be removed if it has no successor in the graph and/or it cannot lead to an accepting state. An extension of owcty algorithm is presented in [2]. The owcty-otf algorithm employs back-level edges as computed by the breadth-first search.

An extension of the owcty algorithm is presented in [4]. The owcty-otf algorithm combines the basic owcty algorithm with a limited propagation of selected accepting states as performed within the map algorithm.

Algorithm negc [7] reduces the LTL model checking problem to a negative cycle detection problem. To do so, the initial graph is transformed: every edge exiting an accepting state is labeled with -1 while every edge exiting a non-accepting state is labeled with 0 (a counter-example exists iff the transformed graph contains a negative cycle).

Every accepting cycle contains at least one accepting state and one *back-level edge* (s,s') such that $d(s) \geq d(s')$, where $d(x)$ is the length of the shortest path from s_0 to x. Algorithm bledge [1] stems from this observation. It detects all back-level edges using a distributed BFS and then checks in parallel whether at least one back-level edge belongs to a cycle by using DFS. In [2], an extension of the bledge algorithm has been proposed (bledge-otf) that allows on-the-fly accepting cycle detection.

An extension of ndfs for a dual-core machine, called double-DFS (2-ndfs) hereafter, is presented in [22] and implemented in SPIN [19]. It is based on the observation that the blue and the red DFS can be performed independently. The linear complexity of ndfs is kept although the algorithm can only be applied to dual-core systems.

After preparing this final version, we noticed that another approach on parallelising Nested Depth First Search appears in this same volume [23]. Both approaches appear to be complementary, since the colours shared are not the same, thus affecting different parts of the program execution. Moreover, in the other approach, a synchronisation mechanism is required whereas we use randomised executions with a repair procedure.

Table 1 summarises explicit states algorithms designed for LTL model checking. It provides, for each algorithm, the reference introducing it, its time complexity, the number of core(s) it can be run on, the acceleration (experimentally observed) that can be provided and finally its "on-the-flyness" as defined in [4]:

level 0. The algorithm has to explore the whole graph before checking emptiness.

level 1. The algorithm can find an accepting run before building the whole synchronised graph but is not guaranteed to do so.

level 2. The algorithm works on-the-fly. There is always an exploration order of transitions guaranteeing an early termination in the presence of an accepting run.

Note that, with our new algorithm mc-ndfs, the aggregate work performed by all processes increases as more processes get involved in the verification. Hence mc-ndfs does not scale in theory and, in the worst case, does not offer any improvement with respect to a sequential ndfs. Our algorithm is therefore a heuristic algorithm: we can hope to reduce the exploration time through the mechanism it implements, but for some problems it may be equivalent to spawning multiple instances of ndfs. Nevertheless, even in this pathological situation, the use of randomisation can help to report counter-examples faster. This is one of the founding principles of the Swarm tool [20].

Table 1. Explicit state algorithms for Büchi emptiness check

Algorithm	Source	Time Complexity	Scalability	Acceleration	On-the-flyness
ndfs	[11]	$O(n+m)$	1 core	-	2
couv-tarjan	[12]	$O(n+m)$	1 core	-	2
GV-tarjan	[17]	$O(n+m)$	1 core	-	2
2-ndfs	[22]	$O(n+m)$	1–2 core(s)	average	2
map	[8]	$O(a^2 \cdot m)$	1–N core(s)	excellent	1
owcty	[9]	$O(h \cdot m)$	1–N core(s)	excellent	0
owcty-otf	[4]	$O((h \cdot (m+n)))$	1–N core(s)	excelent	1
negc	[7]	$O(n \cdot m)$	1–N core(s)	excellent	0
bledge	[1]	$O(m \cdot (n+m))$	1–N core(s)	excellent	0
bledge-otf	[2]	$O(m \cdot (n+m))$	1–N core(s)	excellent	2
mc-ndfs	this paper	$O(p \cdot (n+m))$	1–N core(s)	average-good	2

3 mc-ndfs, a Multi-core Algorithm for LTL Model Checking

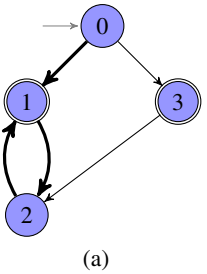
This section introduces mc-ndfs, a new algorithm for LTL model checking, designed for multi-core, shared memory architectures. It first emphasises the difficulty of parallelising ndfs. The principle of mc-ndfs is then explained, the algorithm detailed and formally proven. Finally, its complexity is discussed and a possible extension introduced.

Throughout this section, we denote by $\mathcal{G} = (\mathcal{S}, \mathcal{T}, \mathcal{A}, s_0)$ a synchronised graph and by $\mathcal{P} = \{1, \ldots, P\}$ a pool of running processes.

3.1 Difficulty of Parallelising ndfs

Fig. 1(a) describes a synchronised graph used as a running example throughout this section. Accepting states are drawn, as usual, using double circles. This graph contains a single accepting run 0→1→2→1 highlighted using thick arcs.

Let us consider a naive multi-core version of the ndfs algorithm: processes execute procedure ndfs and share all data (i.e. blue and red flags). Running this algorithm with two processes p_1 and p_2 on the graph of Fig. 1(a) will not necessarily report the accepting cycle, as shown by the execution in Fig. 1(b).



(a)

Process p_1	Process p_2	Blue states	Red States	Seed
dfsBlue(0)	*dfsBlue*(0)	0	-	-
dfsBlue(1)		0, 1	-	-
dfsBlue(2)		0, 1, 2	-	1
dfsRed(1)		0, 1, 2	1	1
	dfsBlue(3)	0, 1, 2, 3	1	3
	dfsRed(3)	0, 1, 2, 3	1, 3	3
	dfsRed(2)	0, 1, 2, 3	1, 2, 3	3

(b)

Fig. 1. A synchronised graph (1(a)) and a possible faulty execution with a naive parallel version of ndfs (1(b))

The blue DFS launched by p_1 starts exploring the left part of the graph and colours the states it meets in blue (i.e. 0, 1 and 2). When backtracking from 2 and then 1, process p_1 initiates a red DFS on state 1. Suppose that meanwhile the blue DFS launched by process p_2 visits the right part of the graph. It colours state 3 in blue and then reaches state 2 previously marked blue by p_1. Since all successors of state 3 are blue, p_2 can start a red DFS on this same state. If p_2 progresses faster than p_1, it will colour states 3 and 2 in red before terminating. Process p_1, when evaluating the successor of state 1, will only find state 2 (already red) and terminate without noticing the accepting cycle.

This small example highlights the key idea behind the correctness of ndfs: the red DFS being nested in the blue DFS guarantees that the invocation sequence of *dfsRed* respects a DFS post-ordering of states. Hence, if two accepting states a_1 and a_2 are such that $a_1 \rightsquigarrow a_2 \wedge \neg(a_2 \rightsquigarrow a_1)$ (noted $a_1 > a_2$ in the sequel) then for all executions the red DFS on a_1 cannot start unless the red DFS on a_2 did terminate. Otherwise the red DFS started on a_1 would colour in red the states of the accepting cycle including a_2 (if any), which would then not be detected by the red DFS initiated on a_2. In all other cases, the invocation order is irrelevant: either the two red DFS cannot interfere ($\neg(a_1 \rightsquigarrow a_2) \wedge \neg(a_2 \rightsquigarrow a_1)$), or a_1 and a_2 belong to the same accepting cycle ($a_1 \rightsquigarrow a_2 \wedge a_2 \rightsquigarrow a_1$) and this cycle will be detected anyway. This naive parallel version of ndfs exhibits the first situation since the DFS post-order is not respected anymore (with $a_2 = 1$ and $a_1 = 3$ in our example).

Solving this kind of conflict constitutes the core difficulty when designing a multi-core version of ndfs.

3.2 Principle of the Algorithm

The previous problem has first been detailed in [5] that proposes an algorithm designed for distributed memory algorithms. Its underlying principle is to maintain a dependency graph that avoids these conflicts and ensures that the red DFS is initiated in the appropriate order: $a_1 > a_2 \Rightarrow \mathit{dfsRed}(a_2)$ terminates before $\mathit{dfsRed}(a_1)$ starts. The principle of mc-ndfs is instead to detect, on-the-fly, configurations in which the invocation order of the red DFS is broken. It is optimistic in the sense that processes evolve without preventing conflicts, and operations to fix problems are performed a posteriori. Thus, all synchronisations required in [5] are avoided, but states may be revisited if a conflict is detected. More precisely, a process notifies its peers by marking state a_2 as *dangerous*. It must then be treated differently as explained below. In our previous example, process p_2 would detect that the red DFS it initiated on state 3 interferes with the one on state 1 still handled by p_1. This conflict is detected by p_2 and reported to p_1 by marking state 1 as *dangerous*. In this situation, p_1 restarts a nested DFS using Algorithm 1.1 but exploits local data only. Hence, the red flag set to *true* by p_2 (i.e. 2.*red*) during the red DFS it performed on state 3 is ignored by p_1, which reports the cycle 1→2→1.

Marking an accepting state a as *dangerous* is thus a means for a process to warn its peers that a red DFS it performed has potentially corrupted the outcome of a red DFS on a. The easiest way to proceed is then, after the red DFS on a has terminated, to reinitiate a nested depth-first search on a since an accepting state could have been missed. Hence, mc-ndfs can be viewed as a two levels algorithm: a *multi-core level* with inter-processes

synchronisations to distribute work among processes; and an *emergency level* without any synchronisation and triggered in case of failure of the first level.

3.3 Details of the Algorithm

Algorithm 1.2 shows the pseudo-code of our new algorithm. States have several attributes. Some are local to a process p (attributes $s.blue_p$ and $s.red_p$ for $p \in \mathcal{P}$) while others are global and shared by all processes ($s.blue, s.dangerous, s.red$).

The main procedure (ll. 1–6) first initialises all boolean attributes of states to *false* and spawns P working processes that will start a blue DFS on the initial state. If they terminate without reporting any accepting cycle, the algorithm reports that none exists.

Roughly speaking, two modifications have been brought to the sequential algorithm. First, to ensure, as much as possible, that processes will engage in different parts of the graph, successor states are visited in a random order thanks to the *shuffle* function (l. 11 and l. 20). Second, inter-process synchronisations have been integrated to both DFSs — through the global attributes $s.blue$ and $s.red$ — in order to limit the visits of a same state by different processes (see l. 16 and l. 21).[1]

Modifications to the blue DFS. First, states visited by the red DFS are not directly marked as red but instead put in set $\mathcal{R}_p$ to be later marked by the blue DFS once the red search has terminated (ll. 28–30). Note that a dangerous state may not be marked as red, unless it is the state currently visited. Second, a state s, marked as dangerous by another process, is revisited with $ndfs_p$ (ll. 31–32). Red and blue attributes associated with each state s by $ndfs_p$ — the same as in Algorithm 1.1 except for the few minor changes listed below — are distinct from those used by mc-ndfs and local to each process so that data computed by another process may not corrupt the result that will come out from a call to procedure $ndfs_p$. Moreover, the computation result of an invocation of $ndfs_p$ can be used during subsequent calls to this same procedure. Therefore, a state cannot be visited more than once by a process p with procedure $ndfs_p$. Consequently, the initialisation step (l. 2 of Algorithm 1.1) is not performed during an invocation of $ndfs_p$ and local flags used in this procedure can be initialised at l. 1 of Algorithm 1.2.

Modifications to the red DFS. First, a successor state s' of s is marked as dangerous (ll. 14–15) when it is accepting but not red. In this situation, the red DFS on s' has not terminated (since $\neg s'.red$) although it may have started. The red flags of states reachable from s' that the current process p will set to *true* (at l. 30) must thus be ignored by any process $q \neq p$ that will later launch $ndfs_q(s')$ if $dfsRed_q(s')$ does not report a cycle. This situation corresponds to the kind of conflict exhibited by our previous example.

[1] Attribute $s.blue$ is set to *true* as soon as s is backtracked from a process whereas it could instead be set before the exploration loop of ll. 20–22. This second alternative would have severely limited the degree of parallelism: as soon as a process p would push a state s on its blue DFS stack, it would prevent all other processes from visiting s and all its successors. For instance, if the initial state had a single successor mc-ndfs would then most likely degenerate into a sequential ndfs. However, by doing so, we leave the possibility to have different processes visiting the same state with the blue DFS: this is thus a tradeoff between the degree of parallelism and the amount of work performed.

Algorithm 1.2. The mc-ndfs algorithm for P working processes

```
1  initialise all flags to false
2  execute dfsBlue_1(s_0) || ... || dfsBlue_P(s_0)
3  wait for termination of
4        dfsBlue_1, ..., dfsBlue_P
5  if ¬ cycle reported then
6     report no-cycle
7
8  procedure dfsRed_p(s) is
9     s.red_p := true
10    R_p := R_p ∪ {s}
11    for s' ∈ shuffle(succ(s)) do
12       if s' = seed_p then
13          report cycle
14       if s' ∈ A ∧ ¬s'.red then
15          s'.dangerous := true
16       if ¬s'.red ∧ ¬s'.red_p then
17          dfsRed_p(s')
18 procedure dfsBlue_p(s) is
19    s.blue_p := true
20    for s' ∈ shuffle(succ(s)) do
21       if ¬s'.blue ∧ ¬s'.blue_p then
22          dfsBlue_p(s')
23    s.blue := true
24    if s ∈ A then
25       seed_p := s
26       R_p := ∅
27       dfsRed_p(s)
28       for r ∈ R_p do
29          if ¬r.dangerous ∨ s = r then
30             r.red := true
31       if s.dangerous then
32          ndfs_p(s)
```

Another major change with respect to ndfs is that mc-ndfs marks states as red when a red DFS terminates (ll. 28–30) by storing in $\mathcal{R}_p$ all states visited by the DFS. Indeed, as a red DFS terminates, all states it visited are guaranteed not to belong to an accepting cycle unless a state marked as red led to a non-red and accepting state (hence marked dangerous, as explained above). This information can also be used by other processes. The proof of the algorithm will clarify the motivation for marking states as red only when the red DFS terminates and not earlier.

3.4 Proof of the Algorithm

Intuitively, the correctness of our algorithm stems from the way states are marked red and dangerous. When a red DFS on an accepting state a_1 is triggered by process p before the red DFS has terminated on a state a_2 with $a_1 > a_2$, some states $s \in \mathcal{R}_p$ around a_2 will be marked red and a_2 dangerous. However, since a_2 is marked as dangerous before states of $\mathcal{R}_p$ become red (states are marked as dangerous during the red DFS while states become red once the red DFS has terminated), if an accepting cycle on state a_2 is not discovered, then it is due to the fact that the red DFS on a_2 reached a red state which in turn implies that a_2 has been marked as dangerous. Hence, $ndfs_p(a_2)$ will necessarily be triggered after the red DFS, and the cycle will be reported.

The proof proceeds in five steps. First, it is straightforward that all states will be visited by a blue DFS and thus all accepting states will be visited by a red DFS.

Proposition 1. *After the termination of algorithm* mc-ndfs, either *an accepting cycle is reported* or $\forall s \in \mathcal{S}, s.blue \wedge s \in \mathcal{A} \Rightarrow s.red$.

Second, it is an invariant property that an accepting state a can only be marked red after the termination of $dfsRed_p(a)$ (for a process $p \in \mathcal{P}$).

Proposition 2. *Let $a \in \mathcal{A}$. There exists $p \in \mathcal{P}$ such that $dfsRed_p(a)$ is initiated by* mc-ndfs *with $a.red = false$.*

Proof. Initially, $s.red = false, \forall s \in \mathcal{S}$. From the conditions at l. 29 and l. 14, it holds that $dfsBlue_p(s)$ changes $r.red$ from *false* to *true* (at l. 30) if and only if $r \notin \mathcal{A} \vee r = s$. Hence, if $a \in \mathcal{A}$, $a.red$ can be set to *true* by $p \in \mathcal{P}$ after the termination of $dfsRed_p(a)$. Since, from Prop. 1, $\forall a \in \mathcal{A}, a.red = true$ when mc-ndfs terminates, our claim is proven. □

Third, all accepting states reachable from a red state are either red or dangerous.

Proposition 3. *For any $(s,s') \in \mathcal{S} \times \mathcal{A}$: $s.red \wedge s \rightsquigarrow s' \Rightarrow s'.red \vee s'.dangerous$.*

Proof. The proof proceeds by induction on set $\mathcal{S}$. Initially, $s.red = false, \forall s \in \mathcal{S}$ and the proposition holds. Let $s \in \mathcal{S}$ be a state marked as red at l. 30 by $dfsBlue_p$. Now assume that the proposition does not hold for s: $\exists a \in \mathcal{A}$ with $s \rightsquigarrow a \wedge \neg a.red \wedge \neg a.dangerous$.

Necessarily, $dfsRed_p(s)$ has been initiated and terminated (since s has been put in $\mathcal{R}_p$). Let us consider a path $s = s_0 \rightarrow \ldots \rightarrow s_n \rightarrow a$. After the initiation of $dfsRed_p(s)$ we necessarily reached a configuration where $dfsRed_p(s_i)$ is initiated; and $s_j = s_{i+1}$ is not visited by the red DFS: $s_j.red \vee s_j.red_p$. Otherwise it would hold, from ll. 14–15, that $a.dangerous$. Now two possibilities arise:

$s_j.red$ — Using our induction hypothesis, $s_j.red \Rightarrow a.dangerous$ (since $s_j \rightsquigarrow a$) which leads to a contradiction.

$\neg s_j.red \wedge s_j.red_p$ — $s_j.red_p$ implies that $dfsRed_p(s_j)$ has been initiated. By recursively applying the same reasoning with path $s_j \rightarrow \ldots \rightarrow s_n \rightarrow a$ we will necessarily find $s_k \in \{s_j, \ldots s_n\}$ with $s_k.red$ which, again, leads to a contradiction.

Hence, if the proposition holds before the assignment at l. 30 then so does it after its execution. Using the induction hypothesis, the proposition holds. □

The fourth point is the key to ensure the correctness of our algorithm: for any accepting cycle going through accepting states $a_1 \ldots a_n$, at least one process p will, by executing $dfsRed_p(a_i)$ for some a_i, report the cycle or revisit a_i through the execution of $ndfs_p(a_i)$ because $a_i.dangerous$ has been set to *true* by another process before $dfsRed_p(a_i)$ terminates.

Proposition 4. *Let $a_1 \in \mathcal{A}, \ldots, a_n \in \mathcal{A}$ belong to the same accepting cycle. Then there exists $p \in \mathcal{P}$, $a_i \in \{a_1, \ldots, a_n\}$ such that* either *$dfsRed_p(a_i)$ reports the accepting cycle* or *$a_i.dangerous = true$ once $dfsRed_p(a_i)$ has terminated.*

Proof. Let us consider an accepting cycle $s_1 \ldots s_n$ with $s_1 = s_n \in \mathcal{A}$. In this proof we assume that $dfsRed_p(s_1)$ starts for some $p \in \mathcal{P}$ and that $s_1.red = false$. This will necessarily happen thanks to Prop. 2. If $dfsRed_p(s_1)$ does not report this accepting cycle we necessarily reach the following configuration:

1. States $s_1, \ldots, s_i$ (with $i < n$) are (in this order) on the red DFS stack of process p.
2. When visiting the successor(s) of s_i, $dfsRed_p(s_i)$ ignores state $s_j = s_{i+1}$ and does not launch $dfsRed_p(s_j)$.

This situation occurs since otherwise the cycle would be discovered by process p. From the condition at l. 16, there are two possibilities:

$s_j.red$ — From Prop. 3, $s_1.dangerous = true$ since $s_j \rightsquigarrow s_1 \wedge s_j.red = true \wedge s_1.red = false$. Hence, since $s_1.dangerous = true$ when $dfsRed_p(s_1)$ terminates, our proposition holds.

$\neg s_j.red \wedge s_j.red_p$ — $s_j.red_p$ has necessarily been set to *true* during a previous invocation of $dfsRed_p$. Hence, s_j was previously added to $\mathcal{R}_p$ and it holds that either $s_j.red = true$ (which leads to a contradiction), or $s_j.dangerous$ and, again, our proposition holds since we had $s_j.dangerous$ when $dfsRed_p(s_j)$ terminated. □

At last we can prove that the nested DFS initiated, at l. 32, on a dangerous state s will report any accepting cycle containing a or a' reachable from a.

Proposition 5. *For any $a \in \mathcal{A}$, $p \in \mathcal{P}$, $ndfs_p(a)$ reports an accepting cycle if and only if there is an accepting cycle around state $a' \in \mathcal{A}$ with $a \rightsquigarrow a'$.*

Proof. The correctness of Prop. 5 is a direct consequence of the correctness of algorithm ndfs (see [11]). If $ndfs_p(a)$ is initiated and if a cycle containing $a' \in \mathcal{A}$ (with $a \rightsquigarrow a'$) is not reported then $ndfs_p(a)$ necessarily reaches a state s belonging to the cycle and already visited by a previous invocation of $ndfs_p(a'')$. This is however impossible, since the cycle would have been visited during this previous search. □

Theorem 1 establishes the correctness of mc-ndfs as a consequence of Prop. 2, 4 and 5.

Theorem 1. *Algorithm mc-ndfs reports an accepting cycle if and only if there is an accepting cycle in $\mathcal{G}$.*

Proof. Let us consider an accepting cycle containing $a \in \mathcal{A}$. From Prop. 2, there exists $p \in \mathcal{P}$ s.t. $dfsRed_p(a)$ will be invoked with $a.red = false$. From Prop. 4, it will report the accepting cycle, or $a.dangerous = true$ will hold after the termination of $dfsRed_p(a)$. In the latter case, $ndfs_p(a)$ will be initiated and the accepting cycle reported (from Prop. 5). □

3.5 Complexity of the Algorithm

It is straightforward to see that a state will be visited at most four times by each process: by the blue and red DFS of mc-ndfs and by the blue and red DFS of ndfs. Hence, following the notations of Section 2, the time complexity of mc-ndfs is $O(p \cdot (m+n))$.

To encode flags associated with a state $3 + 4 \cdot p$ bits are required: 3 bits for global attributes (*dangerous*, *blue*, and *red*); and 4 bits for local process attributes ($blue_p$, red_p for mc-ndfs and ndfs). This is negligible if we perform an exact exploration and store full state vectors, but a trade-off has to be made if we use e.g. bitstate hashing [21] that encodes the graph as a large bit vector where each bit represents a single state. For instance, with 8 cores and 16 GB, we can visit graphs with up to $3.8 \cdot 10^9$ states and may divide the execution time by 8. With the same amount of RAM and 16 cores, the execution time can drop by the same factor, but the graph size is limited to $2 \cdot 10^9$ states.

3.6 Using Tarjan's Algorithm in Nested Searches

Algorithm mc-ndfs waits for a red DFS to be completed before reporting new red states. However, one could proceed more efficiently. Indeed, the important property to be verified is that dangerous states are discovered and reported as such before states leading to them become red. Hence, we could easily replace the existing $dfsRed_p$ procedure by Tarjan's algorithm for SCC decomposition and register red states as the search progresses. When Tarjan's algorithm pops states belonging to a same strongly connected component *scc*, we are sure that all states reachable from $s \in scc$ (and hence, all states potentially dangerous) have already been visited. Therefore all states of a same component can become red as the component is backtracked from. Although this extension is expected to improve the time performance of our algorithm, it also requires the use of extra memory (2 integers per state, see [25] for details on Tarjan's algorithm), which, once again, can be problematic if we combine mc-ndfs with bitstate hashing.

4 Experimental Results

We implemented a prototype of the mc-ndfs algorithm on top of the pthread library and experimented with it on a 16-core machine. Instead of selecting the execution time as a *performance* criterion, we consider the maximal number of visited states over all CPU cores. Several reasons motivated this choice. First the input graphs analysed were given implicitly as a disk file. Therefore, all time-consuming operations (e.g. successor computation, state comparison, insertion in hash table) were already performed and synchronisations dominate the whole execution times. This observation is not only valid for mc-ndfs but also with the map algorithm, that we also implemented in our prototype. Therefore using time as a performance criterion did give a good insight of their performances. Moreover this measure is more reliable than the execution time as it gives a very accurate idea on the "theoretical" scalability of an algorithm: it is independent of the implementation; and it focuses on the search algorithm by putting aside all other time consuming operations like, e.g. synchronisations or data structure initialisations. All measurements reported in this section are expressed this way. These results and the accompanying comments must therefore be taken with care: they do not show the exact acceleration of mc-ndfs but what can be achieved in the ideal situation. As explained in Section 5 our next goal is to provide a real implementation of algorithm mc-ndfs in a verification platform to evaluate its concrete performance.

Input models. All models are issued from the BEEM database [24] that includes more than 50 models of different categories, e.g. mutual exclusion algorithms, communication protocols. We deliberately removed instances of families *Puzzles* (9 models) and *Planning* (5 models) that contain mostly toy examples and only experimented with graphs containing more than 10^6 states. This finally represented a total of 163 input graphs out of which 44 do not have an accepting run while the other 119 do. The results shown below deal only with the former family. Indeed, in most cases, ndfs could easily report an accepting cycle by visiting only a few hundreds of states. Therefore, it did not make much sense to experiment with mc-ndfs on these instances. We found only very few graphs (6 out of 119) for which the use of mc-ndfs could significantly speed up the

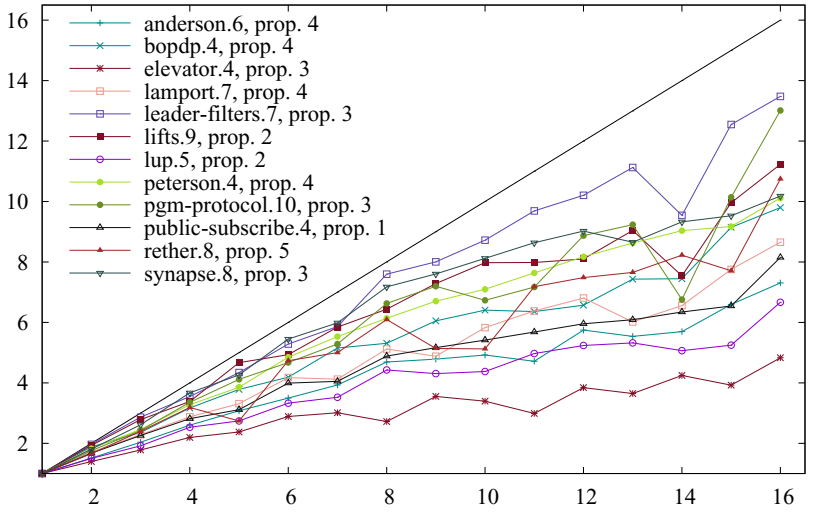


Fig. 2. Acceleration of mc-ndfs on some selected instances

reporting of an accepting run w.r.t. ndfs. Due to space constraints we have selected a few representative instances from our experiments to be presented in this section. The description of all models used can be found on the BEEM webpage [24].

Accelerations. We first analysed the acceleration of mc-ndfs, defined, for N cores, as the ratio of the performance (as defined above) with 1 core over the performance with N cores (using the same algorithm on the same model instance). Figure 2 shows the acceleration as a function of the number of processing cores used for some selected instances. We tried to select a representative set of instances according to different criteria: characteristics of the state graph (width, height, SCC graph structure, ...), type of system modeled (mutual exclusion algorithm, communication protocol, ...), complexity of the model (from simple models to industrial protocols).

The results observed are more or less in line with expectations. The performance of our algorithm is largely impacted by the graph structure. Indeed, for graphs composed of a single or few large SCCs (e.g. lup, public-subscribe) processes often visit the same part of the graph and the use of additional cores does not always bring significant improvements. In contrast, when the graph is clustered into unconnected parts (e.g. pgm-protocol) or acyclic (e.g. leader-filters) processes engage in different parts of the graph, thanks to the use of randomisation, and the acceleration observed is much better. An important parameter also seems to be the length of the longest elementary cycle (i.e. a cycle that does not contain two occurrences of the same state). Since, our algorithm proceeds in a depth-first manner, it is obvious that at least one of the blue DFSs performed concurrently will have, at some point, all the states of this cycle in its stack, and the acceleration will stay low. We also applied mc-ndfs on some graphs randomly generated with long such cycles and the acceleration observed was negligible. Fortunately, real-life systems usually do not exhibit this characteristic.

Table 2. Process workload of mc-ndfs for 16 cores on instances of Figure 2

Instance	Prop.	States	Min.	Max.	Avg.	Std. Dev.
anderson.6	4	36,119,671	5,894,164	7,396,706	6,617,656	12,956
bopdp.4	4	15,923,138	1,291,852	1,625,304	1,396,039	10,203
elevator.4	3	1,006,453	187,061	232,980	209,744	14,128
lamport.7	4	74,413,141	9,938,566	12,723,438	10,958,991	5,308
leader-filters.7	2	26,302,351	2,983,182	3,902,860	3,383,017	7,068
lifts.9	2	7,831,426	1,016,685	1,161,696	1,093,915	12,757
lup.5	2	34,425,340	4,797,633	6,107,470	5,453,463	13,906
peterson.4	4	2,239,039	247,738	332,069	279,644	8,841
pgm-protocol.10	3	7,233,361	458,128	618,476	509,445	5,808
public-subscribe.4	1	1,977,587	248,933	258,743	253,194	2,410
rether.8	5	25,405,545	3,252,470	3,541,148	3,397,022	11,524
synapse.8	3	19,045,831	1,079,676	1,871,015	1,362,764	15,684

Workload. Table 2 provides for instances of Figure 2: the number of states visited by the least and most loaded processes (columns Min. and Max.), the arithmetic average workload (column Avg.) and the standard deviation in the workload (column Std. Dev.) for 16 cores only. It appears the work is usually well balanced among processes although there can be some important variations between the most and least loaded processes. This is clear from the low standard deviation, even in these cases.

Comparison with the map algorithm. Algorithm map uses a modified parallel breadth-first search to compute maximal accepting predecessors. It is as such a very good candidate for parallelisation and, indeed, we observed that mc-ndfs can not compete with it if we compare them w.r.t. acceleration: map always provides a quasi-optimal acceleration

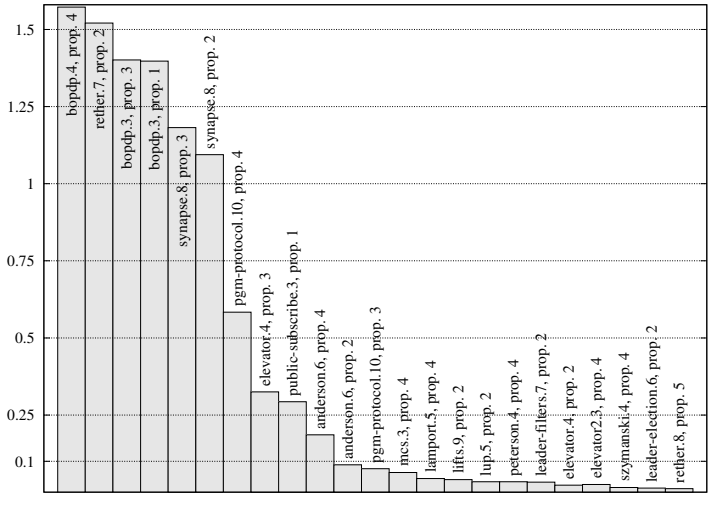


Fig. 3. Absolute performances of mc-ndfs and map on 23 instances for 16 cores

regardless of the model considered. Nevertheless, since map has a polynomial complexity, mc-ndfs often outperforms it when considering their absolute performances. For some selected model instances the ratio of the performance of mc-ndfs over the performance of map for 16 cores only is plotted on Figure 3. Hence, above 1 (resp. below 1), map behaves better (resp. worse) than mc-ndfs. Algorithm map provides better results for a few instances, but in most cases, mc-ndfs is faster, and sometimes significantly. Especially for graphs having a large proportion of accepting states (e.g. lifts.9 with property 2), mc-ndfs often outperforms map. In contrast, map is to be preferred for problems having few or no accepting states (e.g. bopdp.4 with property 4), in which case map reduces to a parallel BFS.

5 Conclusion and Perspectives

We have proposed in this paper a new parallel algorithm for the accepting cycle detection problem. It is a variation of the well-known nested depth-first search algorithm dedicated to multi-core and shared memory architectures. Although, it does not theoretically scale, our experiments revealed that it could provide good accelerations on a variety of interesting instances through the mechanisms it implements. Moreover, similar to the sequential algorithm it is built on, mc-ndfs can detect accepting cycles on-the-fly which few parallel algorithms designed so far are able to do.

We focus on several perspectives for this work. Our experiment only revealed the optimal acceleration that can possibly be achieved using mc-ndfs but the experimentation context can not lead to any conclusion concerning the effective speed-up of our algorithm. A first short term goal is thus to integrate our algorithm into a verification platform such as Divine [6] that also implements many other algorithms (e.g. map, owcty) and will allow a direct comparison of these. Second, we would like to study the combination of our algorithm with existing reduction techniques. Indeed, although mc-ndfs is intended to reduce search times its use can still face the state explosion problem that can only be tackled using dedicated techniques. If mc-ndfs can clearly be combined with some of these techniques, such as bitstate hashing [21] that is a state representation techniques independent of the search algorithm. This observation is not that trivial for some other algorithms such as *partial order reduction* [10]. An implementation of this technique is typically made of two components: a selection mechanism (independent of the search algorithm and, hence, compatible with mc-ndfs) that filters executable transitions of a given state and an *ignoring problem* solver ensuring that a transition will not always be forgotten by the selection function. This solver usually relies on the model checking algorithm. We therefore have to investigate if existing *provisos* used to prevent the ignoring problem can be safely used in conjunction with mc-ndfs and, if not, to devise another solution to this problem, tailored for this algorithm.

References

1. Barnat, J., Brim, L., Chaloupka, J.: Parallel Breadth-First Search LTL Model-Checking. In: ASE 2003, pp. 106–115. IEEE Computer Society, Los Alamitos (2003)
2. Barnat, J., Brim, L., Chaloupka, J.: From distributed memory cycle detection to parallel LTL model checking. ENTCS 133, 21–39 (2005)

3. Barnat, J., Brim, L., Ročkai, P.: Scalable Multi-core LTL Model-Checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 187–203. Springer, Heidelberg (2007)
4. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 407–425. Springer, Heidelberg (2009)
5. Barnat, J., Brim, L., Stríbrná, J.: Distributed LTL Model-Checking in SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
6. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
7. Brim, L., Černá, I., Krčál, P., Pelánek, R.: Distributed LTL Model Checking Based on Negative Cycle Detection. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 96–107. Springer, Heidelberg (2001)
8. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
9. Černá, I., Pelánek, R.: Distributed Explicit Fair Cycle Detection (Set Based Approach). In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 49–73. Springer, Heidelberg (2003)
10. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: State Space Reduction Using Partial Order Techniques. In: STTT, pp. 279–287 (1999)
11. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
12. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
13. Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly Emptiness Checks for Generalized Büchi Automata. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 169–184. Springer, Heidelberg (2005)
14. Esparza, J., Schwoon, S.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
15. Gaiser, A., Schwoon, S.: Comparison of Algorithms for Checking Emptiness on Büchi Automata. In: MEMICS 2009 (2009)
16. Gastin, P., Moro, P., Zeitoun, M.: Minimization of Counterexamples in SPIN. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 92–108. Springer, Heidelberg (2004)
17. Geldenhuys, J., Valmari, A.: Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
18. Godefroid, P., Holzmann, G.J.: On the Verification of Temporal Properties. In: PSTV 1993, pp. 109–124. North-Holland Publishing Co., Amsterdam (1993)
19. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
20. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification Techniques. IEEE Transactions on Software Engineering (2010)
21. Holzmann, G.J.: An Analysis of Bistate Hashing. In: PSTV 1995, pp. 301–314 (1995)
22. Holzmann, G.J., Bosnacki, D.: The Design of a Multi-Core Extension of the Spin Model Checker. IEEE Trans. on Software Engineering 33(10), 659–674 (2007)

23. 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, pp. 321–335. Springer, Heidelberg (2011)
24. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007), http://anna.fi.muni.cz/models/
25. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
26. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, pp. 332–344. IEEE Computer Society, Los Alamitos (1986)