Multi-Core Nested Depth-First Search

Alfons Laarman, Rom Langerak, Jaco van de Pol, Michael Weber, Anton Wijs

{a.w.laarman,langerak,vdpol,michaelw}@cs.utwente.nl
Formal Methods and Tools, University of Twente, The Netherlands

a.j.wijs@tue.nl
Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands

Abstract. The LTL Model Checking problem is reducible to finding accepting cycles in a graph. The Nested Depth-First Search (NDFS) algorithm detects accepting cycles efficiently: on-the-fly, with linear-time complexity and negligible memory overhead. The only downside of the algorithm is that it relies on an inherently-sequential, depth-first search. It has not been parallelized beyond running the independent nested search in a separate thread (dual core).
In this paper, we introduce, for the first time, a multi-core NDFS algorithm that can scale beyond two threads, while maintaining exactly the same worst-case time complexity. We prove this algorithm correct, and present experimental results obtained with an implementation in the LTSmin tool set on the entire BEEM benchmark database. We measured considerable speedups compared to the current state of the art in parallel cycle detection algorithms.

1 Introduction

Moore's Law [18] states that the number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years. Since several years, though, the law no longer relates to the processing speed, while it still relates to the memory capacity of computer hardware. In order to mitigate the declining increase of processing speed, hardware developers have opted for so-called *multi-core* architectures, where multiple cores exist on a processing unit. However, for many algorithms where the main bottleneck was traditionally memory related, a shift to speed related issues can be observed, since these algorithms do not automatically run faster on a multi-core machine. Instead, the introduction of multi-core machines demands a redesign of those algorithms.

This also holds for Model Checking (MC) algorithms; typically, in order to fully verify whether a system specification adheres to a given temporal property, an MC algorithm needs to store the entire so-called *state space* in memory. A state space is a directed graph which explicitly describes all potential behavior of the system specification. Recent observations [2] support that research should be focused on achieving faster MC; currently, memory capacity of the latest hardware allows the analysis of very large state spaces, but the required time to do so is often impractically long.

One advanced MC task is the verification of full Linear Temporal Logic (LTL) properties [1]. LTL can be subdivided into two classes of properties: safety properties, e.g. "nothing bad ever happens", and liveness properties, e.g. "eventually something good happens". While safety properties can be handled with so-called *reachability*, which entails visiting all states in the state space reachable from the initial state, liveness properties require a more complicated analysis.

An algorithm introduced by Courcoubetis et al. [5], often referred to as *Nested Depth-First Search* (NDFS), is particularly useful for checking liveness properties. It has a linear time-complexity and runs *on-the-fly*, i.e. without the need to generate the whole state space, and requires only two bits per state [21].

While reachability has been parallelized efficiently [16], a linear-time multi-core LTL MC algorithm was still unknown. NDFS cannot trivially be adapted to a multi-core setting, since it relies on depth-first search (DFS), which is inherently sequential [20]. And even though many other parallel LTL MC algorithms have been introduced over the course of years, none of them exhibits a worst-case linear-time complexity (or even $\mathcal{O}(n \times \log(n))$, with n the number of states) and the complete on-the-fly property [2–4].

Recent developments, which we group here under the term *Swarm Verification* (SV) [13, 14], have introduced new DFS-based techniques [6, 22] to perform MC tasks in parallel. Although mainly targeted at distributed-memory settings, in which multiple machines are employed, SV can trivially be used on a multi-core, i.e., shared-memory, machine as well. However, when doing so, the fact that the memory is shared is obviously not exploited.

In this paper, we first propose SV-based multi-core NDFS with shared state storage. While this speeds up cycle detection significantly, in the absence of accepting cycles each core still has to traverse the complete state space. Next, we introduce a fine-grained and basic sharing mechanism between threads. Even though parallel search may endanger the correctness of a multi-core NDFS by breaking the post-order, we prove that our algorithm is in fact correct. We subsequently add several known NDFS optimizations [21] to the new parallel setting. Finally, we demonstrate its usefulness in practice by comparing many experimental results obtained with an implementation of our algorithm with results obtained with existing parallel LTL MC algorithms.

Contributions. We present the first multi-core on-the-fly LTL model checking algorithm which is linear-time in the size of the input graph, and has a potential speedup greater than two. We provide a rigorous proof of its correctness and many benchmarks. Though the new algorithm does not scale perfectly for all inputs yet, we still believe to have come one step closer to solving the open question, put forth by Holzmann et al. and Barnat et al. [4, 12], of finding a time-optimal, scalable, parallel algorithm for accepting cycle detection.

Next, in Section 2, the preliminaries behind LTL MC are explained. Related work is discussed in Section 3. We propose a multi-core NDFS algorithm, prove its correctness and provide optimizations in Section 4. Section 5 contains a discussion on the experiments we conducted. Finally, in Section 6, considerations are addressed, conclusions are drawn and possibilities for future work are given.

2 Background (LTL Model Checking)

LTL MC entails checking that a system under verification $\mathcal{P}$ satisfies an LTL property ϕ, which may be a liveness property that reasons over infinite traces of the system ("eventually something good happens"). In order to reason about this, we first introduce the notion of a Büchi automaton:

Definition 1. *A* Büchi automaton *(BA) is a quadruple* $\mathcal{B} = (\mathcal{S}, s_I, \text{post}, \mathcal{A})$*, with* $\mathcal{S}$ *a finite set of states,* s_I *the initial state,* $\text{post} : \mathcal{S} \to 2^{\mathcal{S}}$ *the successor function, and* $\mathcal{A} \subseteq \mathcal{S}$ *a set of accepting states.*

If for $s, t \in \mathcal{S}$, we have $t \in \text{post}(s)$, then we can also write $s \to t$. The reflexive transitive closure of $\to$ is denoted by $\to^*$, and the transitive closure by $\to^+$. We call $s \to^* t$ and $s \to^+ t$ *paths* through $\mathcal{B}$, i.e. sequences of states connected by the successor function. Sometimes we interpret a path π as a set of states, and write $s \in \pi$, meaning that $s \in \mathcal{S}$ is included in the sequence of states of π. A *run* through $\mathcal{B}$ is an infinite path starting at s_I. Finally, we call a run π *accepting* if and only if for infinitely many $s \in \pi$, we have $s \in \mathcal{A}$. Checking the existence of such a run is called the *emptiness problem*.

To check an LTL property ϕ on $\mathcal{P}$, it suffices to solve the emptiness problem for the product of the state graph $\mathcal{G}_\mathcal{P}$ and the Büchi automaton $\mathcal{B}_{\neg\phi}$ (e.g. [23]). Here, $\mathcal{G}_\mathcal{P}$ is an explicit representation of all possible behavior of $\mathcal{P}$ in the form of a graph, and $\mathcal{B}_{\neg\phi}$ is the Büchi automaton accepting all infinite paths described by the negation of ϕ. A counterexample for ϕ in $\mathcal{B} = \mathcal{G}_\mathcal{P} \times \mathcal{B}_{\neg\phi}$ exists iff there exists some $a \in \mathcal{A}$ such that $s_I \to^* a$ and $a \to^+ a$ (i.e. there is an accepting run), where the latter is called an "accepting cycle". Hence, solving the emptiness problem corresponds with determining the reachability of an accepting cycle. The use of a successor function instead of a *transition relation* more closely corresponds with the setting for on-the-fly MC, where the graph structure is unknown in advance.

The first linear-time algorithm to detect accepting runs was proposed by Courcoubetis et al. [5] and, today, is often referred to as NDFS. Over the years, extensions to NDFS have been proposed in, e.g., [9, 15, 21]. In this paper, we propose a *multi-core* NDFS (MC-NDFS), which is based on NNDFS from [21]. Alg. 1 most closely resembles NNDFS from [21] with one minor modification: it does not include *early cycle detection* in dfs_blue, for this extension does not contribute to the understanding of MC-NDFS.

As in all NDFS algorithms, nndfs(s_I) initiates a DFS from state s_I, here called the *blue* DFS, since explored states are colored blue (note that initially, all states are white). As is usual, dfs_blue is performed with a stack, and a state is colored *cyan* if it is on the stack of dfs_blue. Hence, a newly visited state is first colored cyan, and after exploration, it is colored blue. At l.16, if the blue DFS backtracks over a state $s \in \mathcal{A}$, then dfs_red(s) is called, which is a secondary DFS to determine whether there exists a cycle containing s. As described in [21], on l.6, if a successor of s is colored cyan, then an accepting cycle is found, and the NNDFS exits. Otherwise, for each blue successor, dfs_red is called on l.10. Note that an accepting state s is colored red only after its red DFS is finished (l.18). During its red DFS it is cyan, hence it can be detected at l.6.

```
1  proc nndfs(s_I)
2    dfs_blue(s_I)
3    report no cycle
4  proc dfs_red(s)
5    for all t in post(s) do
6      if t.color=cyan
7        report cycle & exit
8      else if t.color=blue
9        t.color := red
10       dfs_red(t)
11 proc dfs_blue(s)
12   s.color := cyan
13   for all t in post(s) do
14     if t.color=white
15       dfs_blue(t)
16   if s ∈ A
17     dfs_red(s)
18     s.color := red
19   else
20     s.color := blue
```

Alg. 1. An adapted New NDFS algorithm

NNDFS runs in linear time, since each reachable state is at most visited twice, once in the blue DFS and once in a red DFS. The algorithm is correct due to the fact that the red DFSs are initiated according to the post-order of the accepting states imposed by the blue DFS (i.e. the last visited accepting state is considered first, the last but one next, etc.), hence an already red state does not need to be re-explored later in another red DFS. This intuition is demonstrated with an abstract proof in [5]. In [9], a standalone correctness proof is given for NNDFS with early cycle detection and an extension called *allred* (both are explained in Section 3). In Section 4.4, we show how these extensions can be introduced in MC-NDFS in an elegant and correct way.

3 Related Work

Two prominent classes of linear-time algorithms to detect accepting runs are formed by the NDFS-*based* and the *Strongly Connected Component* (SCC)-*based* algorithms. The performance of both classes of algorithms is known to be similar, up to some exceptions: Algorithms in the NDFS class use less memory, while algorithms in the SCC class tend to find counter-examples faster [9,10,21]. Since we propose an NDFS-based algorithm, the emphasis here is on related work in the NDFS class. Finally, we also discuss breadth-first search (BFS)-*based* algorithms.

NDFS. As mentioned in Section 2, NDFS was introduced in [5]. There, a correctness proof is given based on the fact that red DFSs are initiated for accepting states based on the post-order enforced by the blue DFS. Holzmann et al. [15] observe that it suffices in a red DFS to check the reachability of a state currently on the stack of the blue DFS, i.e. a state colored cyan in NNDFS, since such a state can reach the accepting state which initiated the current red DFS, closing an accepting cycle.

Schwoon and Esparza [21] combine all of the above extensions and observe that some combinations of colors can never occur. This allows them to introduce a *two-bit color encoding*, also encoding a cyan color for states on the stack of

the blue Dfs. Finally, Gaiser and Schwoon [9] introduce the *allred* extension and give a standalone proof for their Nndfs. The *allred* extension incorporates an additional check in the blue Dfs: if all successors of a state s are red, then s can be colored red as well. This avoids some calls of dfs_red. We will show later that for our Mc-ndfs, this extension is very useful.

Parallel Ndfs. Holzmann and Bošnački [11] proposed a dual-core Ndfs based on the observation that a transition initiating a red Dfs is an "irreversible state transition", i.e. it splits the state graph. A new thread is launched to handle the red Dfs. Since both Dfss are still inherently sequential, the number of threads cannot exceed two, and both potentially have to search the entire state graph. Courcoubetis et al. already mentioned that the two Dfss could be interleaved.

Prominent model checking approaches primarily aimed at settings with distributed memory, e.g., when using a cluster or grid, are swarm verification (SV) [13,14] and *Parallel Randomized DFS* [6,22] (Prdfs). These are so-called *embarrassingly parallel* [8] techniques, since the individual workers operate fully independently, i.e. without communication with the other workers. From here on, when mentioning SV, we refer to existing SV and Prdfs techniques. Note that the search direction of a Dfs is determined by the order in which states are selected for exploration from $\mathsf{post}(s)$ (for any $s \in \mathcal{S}$), e.g. on l.13 of Alg. 1. In SV, basically each worker performs a Dfs with a unique ordering of the successor states. In this way, workers explore different parts of the reachable state graph first. This method has proven to be very successful for bug-hunting. In the absence of bugs, though, the graph will be explored N times, with N the number of workers, since the workers are unaware of each other's results. Although not explicitly mentioned before, SV can be performed in a multi-core setting as well with each worker performing the Ndfs algorithm.

Bfs-*based methods.* Several other LTL MC methods exists which are not Dfs-based. Instead these algorithms rely on Bfs techniques and are therefore easier to parallelize, even in a distributed setting. On the down side, the linear-time complexity and on-the-fly property is often lost. Tab. 1 gives a brief overview of those parallel LTL MC algorithms that have been found suitable for implementation in a multi-core setting [2,3].

Table 1. Multi-core Bfs-based LTL MC algorithms and their worst-case time complexity and on-the-fly property. ($\mathcal{T}$ the set of reachable transitions, and h the height of the Scc quotient graph).

Algorithm	Time complexity	On-the-fly
Map [2]	$\mathcal{O}(\lvert\mathcal{A}\rvert^2 \cdot \lvert\mathcal{T}\rvert)$	Heur.
Owcty [4]	$\mathcal{O}(h \cdot \lvert\mathcal{T}\rvert)$	No
Otf_Owcty [4]	$\mathcal{O}(h \cdot (\lvert\mathcal{T}\rvert + \lvert\mathcal{S}\rvert))$	Heur.

Map preserves the on-the-fly property to the extent that it is heuristic: cycles can be detected early, but this is not guaranteed. By combining Map with *One-Way-Catch-Them-Young* (Owcty), the same property is transferred to the new on-the-fly Owcty (Otf_Owcty) algorithm. For the important class of weak LTL, the algorithm has been shown to be time-optimal [4], therefore it is the current state of the art in multi-core LTL MC.

4 Multi-Core NDFS

4.1 A Basic Multi-Core Swarmed NDFS

As already mentioned, SV is compatible with a shared-memory setting. However, the independence of workers in SV may result in duplicated states on the different machines, hence, when mapped naively to a multi-core machine, the shared memory is not exploited. Therefore, we store all states in a shared lockless hash table that has been shown to scale well for this purpose [16].

A basic SV NDFS algorithm executes an instance of Alg. 1 for each worker i with thread-local color variables. The two bits needed per state per worker are small compared to the state itself and for a dozen or so workers, memory usage is still lower than for SCC-based algorithms [21]. Local permutations of the post function direct workers to different regions of the state graph, resulting in fast bug-finding typical for SV. With post_i^b (post_i^r) we denote the permutation of successors used in the blue (red) DFS by worker i. For inputs without accepting cycles this solution does not scale. In the next section, we attack this problem.

4.2 Multi-Core NDFS with Global Coloring

A naive sharing of colors between multi-core workers is prone to influence the independent post-orders on which the correctness of the NDFS algorithm relies [5]. In the current section, we present a color-sharing approach which preserves correctness. The next section provides a correctness proof of this MC-NDFS algorithm.

The basic idea behind MC-NDFS in Alg. 2 is to share information in the backtrack of the red DFSs (dfs_red). A new (local) color *pink* is introduced to signify states on the stack of a red DFS, analogous to cyan for a blue DFS. When a red DFS backtracks, the states are globally colored red. These red states are now ignored by both *all* blue and red DFSs, thus pruning the search spaces for all workers i.

```
1  proc mc–ndfs(s, N)
2    dfs_blue(s,1)‖..‖dfs_blue(s,N)
3    report no cycle
4  proc dfs_blue(s, i)
5    s.color[i] := cyan
6    for all t in post_i^b(s) do
7      if t.color[i]=white∧¬t.red
8        dfs_blue(t, i)
9    if s ∈ 𝒜
10     s.count := s.count + 1
11     dfs_red(s, i)
12   s.color[i] := blue
13 proc dfs_red(s, i)
14   s.pink[i] := true
15   for all t in post_i^r(s) do
16     if t.color[i]=cyan
17       report cycle & exit all
18     if ¬t.pink[i] ∧ ¬t.red
19       dfs_red(t, i)
20   if s ∈ 𝒜
21     s.count := s.count − 1
22     await s.count=0
23   s.red := true
24   s.pink[i] := false
```

Alg. 2. A Multi-core NDFS algorithm, coloring globally red in the backtrack

Additionally, we count the number of workers that initiate dfs_red in *s.count* (l.10) and wait with backtracking until this counter is 0 (l.21,22). This enforces that if multiple workers call dfs_red from the same accepting state, they will finish simultaneously. Fig. 3 illustrates the necessity of this synchronization by a simple counter example that could occur in absence of this synchronization.

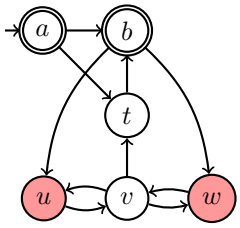

Fig. 3. Counter example to correctness of MC-NDFS without **await** statement.

A worker 1 could explore a, b, u, v, w, backtrack from w, explore t and backtrack all the way to the accepting state b where it will call a dfs_red at l.11. Then this dfs_red$(b, 1)$ could explore u, v, w and halt for a while. Now, a worker 2 could start dfs_red$(b, 2)$ in a similar fashion. Next, it could explore w, v, u, backtrack, mark u red and halt for a while. Then worker 1 continues to mark w red.

Note that the two accepting cycles contain red states, but both workers can still detect a cycle by continuing to explore v and t (b is cyan in the local coloring of both workers). However, a third worker can endanger this potential, while the first two workers halt for a while. After worker 3 searches a and subsequently t and b in a blue DFS, it will start a dfs_red at b, but because its successors are now red, worker 3 will backtrack and mark b red. Note that exactly this step is prevented by adding the **await** statement. Continuing with dfs_red$(a, 3)$, states t and a will also become red, obstructing workers 1 and 2 from finding a cycle.

No worker finds a cycle in this way, which thus constitutes a counter example for correctness. However, because worker 3 is forced to wait for the completion of the red DFSs of workers 1 and 2 before it can backtrack from state b in dfs_red$(b, 3)$, this counter example is invalid for MC-NDFS.

Finally, we note that MC-NDFS in Alg. 2 is presented in a form that eases analysis of correctness: without superfluous details. For example, the *pink* variable of states is separate from the *color* variable, which stores only the colors white, blue and cyan. The two-bit color encoding of [21] is thus dropped for a while. In the following section, we prove correctness of MC-NDFS, after which we amend the algorithm in Section 4.4 with the extensions discussed in Section 3. The *allred* extension is shown to improve sharing between workers significantly.

4.3 Correctness Proof

In this section, we provide a correctness proof for MC-NDFS. We assume that each line of the code above is executed atomically. The global state of the algorithm is the coloring of the input graph $\mathcal{B}$ and the program counter of each worker.

We use the following notations: The sets $White_i$, $Cyan_i$, $Blue_i$ and $Pink_i$ contain all the states colored white, cyan, blue, and pink by worker i, and Red contains all the red states. E.g., if $s.color[i] = blue$, we write $s \in Blue_i$. It follows from the assignments of the respective colors to the *color* variable that $White_i$, $Cyan_i$ and $Blue_i$ are disjoint. Also, we denote the state of one worker as dfs_red$(s, i)@X$, meaning that worker i is executing l.X in dfs_red for a state s. Finally, we use the modal operator $s \in \Box X$ to express that $\forall t \in \mathsf{post}(s) : t \in X$.

Correctness of MC-NDFS hinges on the fact that it will never miss all reachable accepting cycles, i.e. it will always find one if one exists. Recall from Section 2 that NDFS ensures that all reachable states are visited only once by both dfs_blue and dfs_red. MC-NDFS ensures that each reachable state is visited *at least* once by both some dfs_blue and dfs_red, therefore for a reachable $a \in \mathcal{A}$, there is at least one $\mathsf{dfs\_red}(a, i)@11$ for some i, that initiates the recursion of the dfs_red.

This recursion continues at l.19, where it tries to find a $t \in \mathit{Cyan}_i$ at l.16 that would close the cycle. Now, if the cycle $a \rightarrow^+ a$ exists, worker i will either find a $t \in \mathit{Cyan}_i$, or is obstructed because it encounters a $t \in \mathit{Red}$ at l.18. Fig. 4 illustrates that workers can obstruct each other from finding cycles. For example, it is possible that a worker 1 initiates a dfs_red for a_1, marking r red. Then, a worker 2, with a different post_i^b, could start a dfs_red for a_2 and be obstructed from finding cycle $\{a_2, r, t, s\}$.

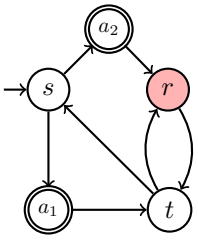

Fig. 4. An obstructed accepting cycle.

We first state invariants that express basic relations between the colors in MC-NDFS. Then, after Lemma 1, we prove the crucial insight (Thm. 1), termination (Thm. 2) and our main correctness result (Thm. 3).

L1. $\forall i: \ \mathit{Blue}_i \cup \mathit{Pink}_i \subseteq \Box(\mathit{Blue}_i \cup \mathit{Cyan}_i \cup \mathit{Red})$
L2. $\mathit{Red} \subseteq \Box(\mathit{Red} \cup \bigcup_i (\mathit{Pink}_i \setminus \mathit{Cyan}_i))$
L3. $\forall i, a \in \mathcal{A}: \ a \in \mathit{Blue}_i \implies a \in \mathit{Red}$
L4. $\forall i, a \in \mathcal{A}: \ a \in \mathit{Pink}_i \implies a \in \mathit{Cyan}_i$
L5. $\forall i: \ \mathit{Pink}_i \subseteq (\mathit{Blue}_i \cup \mathit{Cyan}_i)$

Lemma 1. *The following invariant holds for* MC-NDFS*:* $\forall s \in \mathit{Red}, a \in \mathcal{A} \setminus \mathit{Red}:$
$s \rightarrow^* a \implies (\exists i, p \in \mathit{Pink}_i, c \in \mathit{Cyan}_i: \ s \rightarrow^+ p \xrightarrow{\neg \mathit{Red}}{}^+ c \rightarrow^* a)$

Proof. We show that the property follows from the previous invariants L1-4. Assume $s \rightarrow^* a$ for some $s \in \mathit{Red}$ and $a \in \mathit{Acc}$ with $a \notin \mathit{Red}$. Let $s' \in \mathit{Red}$ be the *last* red state on the path $s \rightarrow^* a$. Then, since $s' \neq a$, it has a successor $t \notin \mathit{Red}$ in this path. By L2 we obtain $t \in \mathit{Pink}_i$ for some worker i, so let $p := t$.

Note that $t \neq a$, otherwise by L4 $t \in \mathit{Cyan}_i$ and by L2 $t \notin \mathit{Cyan}_i$. So we find another successor t' such that $s \rightarrow^* s' \rightarrow t \rightarrow t' \rightarrow^* a$. Assume towards a contradiction that no state on the path $t' \rightarrow^* a$ is in Cyan_i; recall that $t' \rightarrow^* a$ contains no Red states either. Then by L1, all states on $t' \rightarrow^* a$ are in Blue_i. But then also $a \in \mathit{Blue}_i$ and by L3, $a \in \mathit{Red}$, contradiction. So there exists a $c \in \mathit{Cyan}_i$ with $s \rightarrow^* p \rightarrow^+ c \rightarrow^* a$.

Theorem 1. MC-NDFS *cannot miss all accepting cycles.*

Proof. Assume an MC-NDFS run would miss all accepting cycles. Since there are only finitely many cycles, we can investigate the *last* "obstructed cycle" in this run, i.e., the last time that a dfs_red (which originated from some accepting state a on a cycle) encounters Red. That is, we are in $\mathsf{dfs\_red}(s, i)@18$ but we see $t \in \mathit{Red}$, although $s \rightarrow t \rightarrow^* a$.

Note that $a \notin \mathit{Red}$: Just before dfs_red(a,i)@11, $a.\mathit{count}$ was increased by l.10. Therefore, no other worker can make a red, because they are all forced to wait at l.22.[1]

Hence we can apply Lemma 1, to obtain a path $p \xrightarrow{\neg \mathit{Red}+} c$ for some $p \in \mathit{Pink}_j$ and $c \in \mathit{Cyan}_j$. It follows that there is an $a' \in \mathcal{A}$ with $c \to^* a' \to^* p$ (property of DFS stacks). Fig. 5 provides an overview of the shape of the subgraph that we just discussed with the deduced colorings.

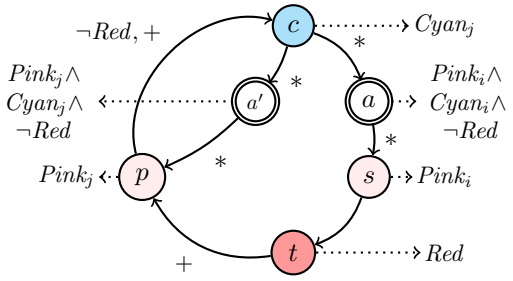

Fig. 5. Snapshot of the cycle in the *last* "obstructed cycle search". Edges with $*, +$ indicate paths of length ≥ 0 and > 0. Dotted arrows denote node colors and $\neg \mathit{Red}, +$ a path without red.

But now we have constructed a cycle for worker j which has not yet been obstructed. This contradicts the fact that we were considering the *last* obstructed cycle. We conclude that there is no last obstructed cycle, hence there exists no run that misses all cycles. □

This proves partial correctness of MC-NDFS. In order to prove that an accepting cycle will eventually be reported, the algorithm is required to terminate.

Theorem 2. MC-NDFS *always terminates with some report at l.3 or l.17.*

Proof. Assuming dfs_red terminates, we can conclude termination of dfs_blue from the fact that for each worker i the set $\mathit{Blue}_i \cup \mathit{Cyan}_i$ grows monotonically (blue is never removed). Eventually, all the states are in the set and the blue search ends. Termination of the **await** statement at l.22 state follows from the basic observation that every worker i can have at most one counter increment on some accepting state, which is decremented at l.21 before waiting. Hence, when worker i is waiting, there can be no other worker waiting for i. Finally, all red DFSs terminate because also the set $\mathit{Red} \cup \mathit{Pink}_i$ grows monotonically. □

Theorem 3. MC-NDFS *reports* **cycle** *if there exists a reachable accepting cycle in the input graph* $\mathcal{B}$ *and it reports* **no cycle** *otherwise.*

Proof. By Theorem 2, the algorithm terminates with some report. If a cycle is reported at l.17 by worker i, we find an $s \in \mathit{Pink}_i$ and $t \in \mathit{Cyan}_i$ with $s \to t$. In that case there is a state $a \in \mathit{Acc}$ on the stack such that $t \to^* a \to^* s \to t$, so there is indeed an accepting cycle.

Otherwise, if no cycle is reported at l.3, all workers have terminated without reporting a cycle. By Theorem 1 there is no accepting cycle in the graph. □

[1] A race condition can occur here, because worker i could increase $a.\mathit{count}$ right after some worker j passed the check at l.22 in dfs_red(a,j). Next, worker i would start its dfs_red(a,i), and find that $a \in \Box(\mathit{Red})$. So i will also make a red and return from dfs_red. It does not matter whether i or j makes a red first. Therefore, we can safely ignore such race conditions.

4.4 Extensions

We can improve MC-NDFS further. Alg. 3 presents MC-NNDFS, which is MC-NDFS with the extensions discussed in Section 3. First, we opted to extend MC-NDFS with *allred* [9] (l.16 and l.24–27). Since the parallel workload of the MC-NDFS algorithm depends entirely on the proportion of the state graph that can be marked red (see Section 5.2), *allred* can improve the scalability. Second, early cycle detection in dfs_blue (l.19–21) is needed to compete with SCC-based algorithms. Finally, the introduction of the two-bit color-encoding from [21] for each worker will eliminate the extra bit per worker used for the pink color.

Sketch of Correctness. The *allred* extension in dfs_blue introduces a new red coloring of a state s at l.27, affecting the proof of Lemma 1. But, since $s \in \Box(Red)$, the induction hypothesis can be applied for the successor t of s.

Due to the early cycle detection at l.19–21, some accepting cycles can be detected already in the blue search. The stack configuration of the blue search thus guarantees us that indeed a cycle with an accepting state exists that is reachable from s_I: $s_I \rightarrow^* t \rightarrow^* s \rightarrow t$ with $t \in \mathcal{A} \vee s \in \mathcal{A}$ (l.20).

The two-bit color encoding overwrites the value of the $s.color[i]$ at l.5. However, L5 shows that only $Cyan_i$ and $Blue_i$ are affected (not $White_i$). The removal of s from $Blue_i$ does not affect dfs_red, since it is insensitive to $Blue_i$. The removal of s from $Cyan_i$ seems more problematic, since cycle detection on l.7 depends on it. However, we also know that the only case where s is removed from $Cyan_i$, is in the initial dfs_red call from l.11 (recursive dfs_red calls are never made on $Cyan_i$ states, since a cycle would be detected at l.16 and l.19 would not have been reached). Hence, $s \in \mathcal{A}$. It turns out that if there exists a

```
1  proc mc–ndfs(s, N)
2    dfs_blue(s,1)‖..‖dfs_blue(s,N)
3    report no cycle

4  proc dfs_red(s, i)
5    s.color[i] := pink
6    for all t in post_i^τ(s) do
7      if t.color[i]=cyan
8        report cycle & exit all
9      if t.color[i]≠pink∧¬t.red
10       dfs_red(t, i)
11   if s ∈ A
12     s.count := s.count − 1
13     await s.count=0
14   s.red := true

15 proc dfs_blue(s, i)
16   allred := true
17   s.color[i] := cyan
18   for all t in post_i^b(s) do
19     if t.color[i]=cyan ∧
20         (s ∈ A ∨ t ∈ A)
21       report cycle & exit all
22     if t.color[i]=white∧¬t.red
23       dfs_blue(t, i)
24     if ¬t.red
25       allred := false
26   if allred
27     s.red := true
28   else if s ∈ A
29     s.count := s.count + 1
30     dfs_red(s, i)
31   s.color[i] := blue
```

Alg. 3. MC-NDFS with extensions (MC-NNDFS)

path $\pi \equiv s \rightarrow^* s$ with $(\pi \setminus s) \cap \mathit{Cyan}_i = \emptyset$, this accepting cycle would have been detected by early cycle detection in dfs_blue ($s_I \rightarrow^* s \rightarrow^* s' \rightarrow s$ with $s \in \mathcal{A}$). Hence, we do not need any provisions to *fix* the removal of s from Cyan_i. This fact was overlooked by Schwoon et al. [9, 21], leading them to complicate their NDFS algorithm (Alg. 1) with delayed red coloring of accepting states.

5 Experiments

We implemented NDFS, multi-core SV NDFS and MC-NNDFS in the multi-core backend of the LTSmin model checking tool suite [17]. This enabled us to use the same input models (without translation) and the same language frontend (compiler). We also implemented randomized post_i functions to direct threads to different regions of the state space, as discussed in Section 4.1.

We performed experiments on an AMD Opteron 8356 16-core (4×4 cores) server with 64 GB RAM, running a patched Linux 2.6.32 kernel. All tools were compiled using gcc 4.4.3 in 64-bit mode with high compiler optimizations (-O3). For comparison purposes, we used all 453 models with properties of the BEEM database [19]. To mitigate random effects in the benchmarks, runtimes are always averaged over 6 benchmark runs. We compared MC-NNDFS against multi-core SV NDFS to answer the question whether a more integrated multi-core approach can win against an *embarrassingly parallel* algorithm. Furthermore, we compared with the best existing parallel LTL MC algorithm OTF_OWCTY, as implemented in DiVinE 2.5.1 [3].

Due to the *on-the-fly* nature of LTL algorithms, we distinguish models containing accepting cycles from models that do not contain them. On the former set, algorithms that build the state space *on-the-fly* and terminate early when a counter example can be found, are expected to perform very well.

5.1 Models with Accepting Cycles

We demonstrate the merits of multi-core SV NDFS by comparing the runtimes with the sequential NDFS. As expected, SV speeds up the detection of accepting cycles (crosses in Fig. 4) significantly compared to sequential NDFS runs. We do not expect to see perfect speedups (16× on 16 cores) across all benchmarks, since the search is undirected and some threads traverse parts of the state space which do not contribute to finding a cycle. However, for some models, multi-core SV NDFS does exhibit perfect speedups, or even superlinear speedups. Due to randomization, multiple workers are more likely to find counter examples [6, 22].

Both multi-core SV NDFS and MC-NNDFS find accepting cycles roughly within the same time (Fig. 5), there is only a small edge for MC-NNDFS (most crosses are in the upper half of the figure), due to work sharing effects. Apparently, the global red coloring does not cause much "obstruction" (see Section 4.3).

We isolated those runs of MC-NNDFS on models with cycles, that have a runtime longer than 0.1 sec, because only those yield meaningful scalability figures.

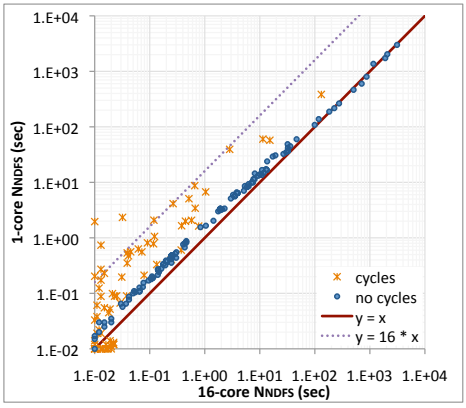

Fig. 4. Log-log scatter plot of multi-core SV NDFS / sequential NDFS runtimes.

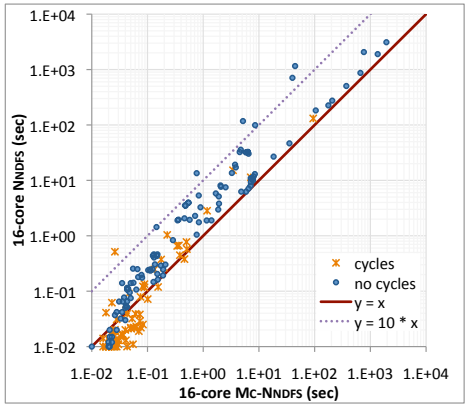

Fig. 5. Log-log scatter plot of MC-NNDFS / multi-core SV NDFS runtimes.

Fig. 7 on the next page shows that these models scale very well (the figure is cut off after a speedup of 20, but it extends well beyond speedups of 100). Out of 54 models with cycles (and runtimes $\geq$ 0.1 sec), $\approx$ 75 % exhibit at least eight-fold speedups and almost half exhibit superlinear speedups (factor > 16).

Finally, a comparison with OTF_OWCTY unsurprisingly shows that MC-NNDFS finds counter examples much faster (crosses in Fig. 6), due to its depth-first on-the-fly nature, while OTF_OWCTY is only heuristically on-the-fly.

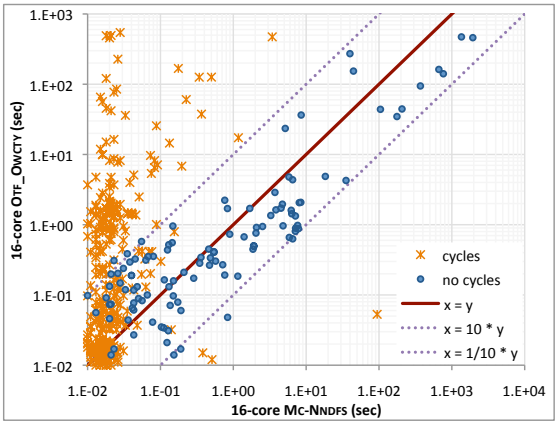

Fig. 6. Log-log scatter plot of MC-NNDFS / OTF_OWCTY runtimes.

5.2 Models without Accepting Cycles

For models without accepting cycles, on-the-fly algorithms lose their edge over other algorithms, as the state space has to be traversed fully. We demonstrate this with our multi-core SV NDFS benchmark runs, which degrade timewise to sequential NDFS (dots in Fig. 4). We note that multi-core SV NDFS causes little overhead compared to the sequential NDFS version, hence it would be safe to run multi-core SV if the presence of a counter example is uncertain.

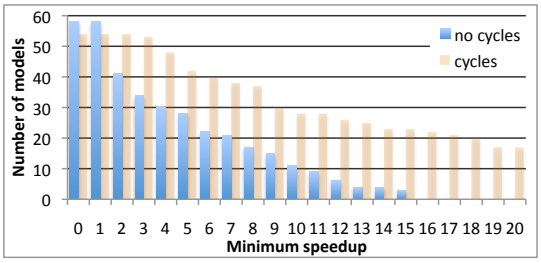

Fig. 7. Model counts of speedups with MC-NNDFS (base case: sequential NDFS)

However, when comparing multi-core SV NDFS against MC-NNDFS (Fig. 5), we observe significant speedups, in some cases more than ten-fold (dotted line) on 16 cores. Again, we isolated the runs of MC-NNDFS on models without cycles that run more than 1 sec (Fig. 7). We observed at least ten-fold speedups for 11 models out of 58 such models. In the BEEM database, we verified the nature of the 40 models that exhibit speedup greater than factor two. These include: *leader election* and other *communication protocols*, *hardware models*, *controllers*, *cache coherence protocols* and *mutual exclusion algorithms*.

Fig. 6 reveals that MC-NNDFS can mostly keep up with the performance of OTF_OWCTY. However, on some models without accepting cycles DiVinE is faster by a factor of 10 on 16 cores. Which algorithm performs best in these cases likely depends on model characteristics, which we have yet to investigate.

However, we did investigate the lack of MC-NNDFS scalability for some models without cycles in Fig. 7. All these cases lack states colored red by dfs_red. However, this does not hold the other way around: many models with few of these red states still exhibit speedups. This can be attributed to the red coloring by the *allred* extension. In fact, for all models without cycles, the proportion of states colored red by dfs_red turned out to be negligible, while *allred* accounts for the vast majority of the red colorings.

We found that the number of red colorings is strongly dependent on the exploration order (post_i). Fig. 8 illustrates that this is indeed possible. If a search advances first from s through t, then t cannot be colored red. This also holds for s, because one of its successors remains blue. However, if a is visited first, then u becomes red, hence later also t and s. It would be interesting to find a heuristic that maximizes red colorings.

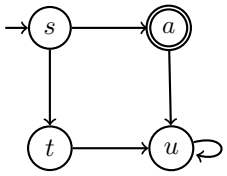

Fig. 8. Exploration order can influence r_N

We also observed that the speedup S_N is dependent on the fraction of red states r_N, as can be expected from the fact that r_N is the fraction of work that can be parallelized: $S_N \approx \frac{T_{seq}}{T_{seq}\times(1-r_N)+T_{seq}\times r_N/N} = \frac{1}{1-(1-1/N)r_N}$, where $T_{seq} \times (1 - r_N)$ is duplicated work. This shows us that the algorithm barely waits for a long time at l.22, which is also confirmed by direct measurements.

6 Conclusions

In this paper, we introduced a multi-core NDFS algorithm, starting from a multi-core SV version, and proved its correctness. Its time complexity is linear in the size of the input graph, and it acts on-the-fly, addressing an open question put

forward by Holzmann et al. and Barnat et al. [4,12]. However, in the worst case, each worker might still traverse the whole graph. We showed empirically that the algorithm scales well on many inputs. The on-the-fly property of MC-NNDFS, combined with the speedups on cycle-free models, makes MC-NNDFS highly competitive to OTF_OWCTY.

The experiments were needed because MC-NNDFS is a heuristic algorithm: in the worst case (no accepting states, hence no red states) no work is shared between workers and the performance reduces to the SV version. However, in these cases no other known linear-time parallel algorithm obtains any speedup (including dual-core NDFS [11]).

The space complexity of MC-NNDFS remains decent: per state $2 \times N$ local color bits, $\log_2(N)$ bits for the *count* variable, and one global red color bit, with N the number of workers. The *count* variable could be omitted, at the expense of inspecting the pink flags of all other workers. However, this would lead to a significant memory contention. The overhead of $\log_2(N)$ bits per state is insignificant next to the space required by the local colors.

Recent development. After preparing this final version, we noticed that another approach on parallelizing NDFS appears in this same volume [7]. Their approach seems complementary, since they share the blue color, where we share red. Instead of our synchronization, they speculatively continue parallel execution and call a sequential repair procedure in the case of dangerous situations.

Future work. We have strong indications that MC-NNDFS can be improved. The previous section showed that a heuristic for exploration order might be of great benefit for the scalability. Furthermore, we think that early cycle detection and work sharing can be improved with SCC-like techniques.

Acknowledgements. We thank Elwin Pater for providing feedback on our algorithms and proofs.

References

1. C. Baier and J.P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.
2. J. Barnat, L. Brim, and P. Ročkai. Scalable Shared Memory LTL Model Checking. *STTT*, 12(2):139–153, 2010.
3. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.
4. L. Barnat, L. Brim, and P. Ročkai. A Time-Optimal On-The-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *ICFEM 2009*, volume 5885 of *LNCS*, pages 407–425. Springer, Heidelberg, 2009.
5. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

6. M.B. Dwyer, S.G. Elbaum, S. Person, and R. Purandare. Parallel Randomized State-Space Search. In *Proc. ICSE 2007*, pages 3–12. IEEE Computer Society Press, 2007.
7. S. Evangelista, L. Petrucci, and S. Youcef. Parallel nested depth-first searches for LTL model checking. In T. Bultan and P.-A. Hsiung, editors, *ATVA 2011*, LNCS. Springer Verlag, 2011. (elsewhere in this volume).
8. I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.
9. A. Gaiser and S. Schwoon. Comparison of Algorithms for Checking Emptiness on Büchi Automata. *CoRR*, abs/0910.3766, 2009.
10. J. Geldenhuys and A. Valmari. Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer Berlin / Heidelberg, 2004.
11. G.J. Holzmann and D. Bošnački. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Trans. On Software Engineering*, 33(10):659–674, 2007.
12. G.J. Holzmann and D. Bošnački. The Design of a Multicore Extension of the SPIN Model Checker. *Software Engineering, IEEE Transactions on*, 33(10):659 –674, oct. 2007.
13. G.J. Holzmann, R. Joshi, and A. Groce. Swarm Verification. In *Proc. ASE 2008*, pages 1–6. IEEE Computer Society Press, 2008.
14. G.J. Holzmann, R. Joshi, and A. Groce. Tackling Large Verification Problems with the Swarm Tool. In *Proc. SPIN 2008*, volume 5156 of *LNCS*, pages 134–143. Springer-Verlag, 2008.
15. G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth First Search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996.
16. A.W. Laarman, J.C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In N. Sharygina and R. Bloem, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss*, USA, October 2010. IEEE Computer Society.
17. A.W. Laarman, J.C. van de Pol, and M. Weber. Multi-core LTSmin: Marrying modularity and scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the Third International Symposium on NASA Formal Methods, NFM 2011, Pasadena, CA, USA*, volume 6617 of *LNCS*, pages 506–511, Berlin, July 2011. Springer Verlag.
18. G.E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(10):114–117, 1965.
19. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *Proc. of SPIN Workshop*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
20. J.H. Reif. Depth-first Search is Inherently Sequential. *Information Processing Letters*, 20(5):229–234, 1985.
21. S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer Berlin / Heidelberg, 2005.
22. H. Sivaraj and G. Gopalakrishnan. Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking. *Electronic Notes in Theoretical Computer Science*, 89(1):51–67, 2003.
23. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.