Multi-Core Nested Depth-First Search

ECE 5510 Course Project, Fall 2019

Annette Feng, Yuqing Liu

December 14, 2019

Abstract

Model Checking is a method for analyzing the finite-state model of a system for correctness against a given specification. A certain class of algorithms for exploring the state space in Model Checking employ Nested Depth-First Search. Depth-First Search algorithms are inherently sequential. The first *multi-core* version of Nested Depth-First Search was introduced in [LLP+11] and has since been followed up by various improvements. In this paper, we present our exploration of Multi-Core Nested Depth-First Search (MC_NDFS) algorithms for our ECE 5510 Course Project. We compare five different implementations of MC_NDFS, including our own variation based on our observations of the other four.

1 Introduction

In Model Checking (MC), verification of the Linear Temporal Logic (LTL) property of liveness of a system specification is a non-trivial task, unlike that of safety which can be handled as a matter of reachability. The LTL MC problem is reducible to finding accepting cycles in a graph. This is accomplished through techniques employing Depth-First Search (DFS). The Nested Depth-First Search (NDFS) algorithm, as introduced in [CVWY92], solves this problem in linear time using a 2-phase recursive search process, with each phase visiting any state at most once. In the first phase, *dfs_blue*, the search process colors blue all states encountered as it looks for all the accepting states in its path. The second phase, *dfs_red*, is initiated from *dfs_blue* on all accepting states found, but in reverse order. The *dfs_red* search colors each state encountered red as it looks for any cycles around the accepting states. If the node s that initiated the *dfs_red* search is encountered during this phase, s is reachable from itself and *dfs_red* reports that a cycle has been detected. Because DFS is an inherently sequential operation, solutions involving these techniques must be carefully adapted to run correctly in Multi-Core (MC) environments.

This paper is organized as follows. In Section 2 we discuss the different MC variations of NDFS that we tested. In Section 3 we discuss the details of our implementation and testing procedure. In Section 4 we present our experiments and results. Finally, in Section 5 we present our conclusions.

2 Background

Swarm Verification (SV), mainly used in distributed memory environments, can be used to run NDFS algorithms in multi-core environments by adapting NDFS to use shared-state storage. This naive MC implementation has each thread traversing the complete state space, which can be stored in shared memory. Each thread independently performs a DFS, but using a unique order of successor states. This ensures that each thread explores a different part of the reachable state space first. However, this adaptation also results in the graph being explored N times.

Variations of NDFS, based on observations of the feasible color combinations with which the states can be colored by the path-coloring scheme, allows for the early detection of cycles and pruning of the search space. A first variation of MC_NDFS utilizes global coloring to share state among the threads [SE05]. The basic idea is that when *dfs_red* backtracks, the states are globally marked red so that these states can be ignored by both red and blue searches, thus pruning the search space of other threads.

A second variation makes further improvements as suggested in [GS09]. Here, the *dfs_blue* search adds an additional check: if all successors of a state s are red, then s can be colored red as well to avoid some calls to *dfs_red*.

In the next section we discuss our implementation of these three MC_NDFS algorithms, in addition to two others, one that makes an improvement in the second variation, and the last variation being our own adaptation.

3 Implementation

To run from the project directory, execute: `./bin/runtest` with no arguments to print usage information. The expected arguments are:

```
<graph_size>
<thread_count>
<iterations>
<num_graphs>
```

We implemented the algorithms outlined in the preceding section using Java 8. The following subsections describe the details of each implemented algorithm.

3.1 Graph Generation

Because we didn't have access to the graph generation tools described in [LLP+11], we implemented our own graph generator, loosely described as follows:

1. Generate the initial graph as a tree structure with the root as the initial state. We recursively generate a variation of trees with a different ratio of nodes at each level by using a random number to decide the level-to-level ratio of nodes.

2. Add internal connections to produce cycles in the tree. To do that in a straightforward way, we randomly select two node and connect them. The tree has $n - 1$ edges for n nodes, so, we randomly select a percentage p, and add $p \times n$ edges to the graph. The result indicates that a random p between certain range can introduce from 0 to several cycles.

3. Randomly mark a percentage of nodes as accepting. Although we can let the algorithm detect any cycle, we want to bring the benchmark close to what the algorithm expects. We select a small percentage of nodes to better randomize the chances of encountering cycles.

We used 40% to 167% downward level-to-level ratio for graph generation, $p = 10\%$ to 60% to generate additional edges, and have 0.5% to 3.5% accepting nodes. As a result, we can generate about 50% graphs with cycles.

3.2 Naive_MC_NDFS

The naive implementation is found in file `src/main/java/ndfs/RB2_Thread.java`. Each thread is initialized with a `ThreadLocal` HashMap to store the visited states along with their set color. For this, as well as all other variations that we describe, the initial coloring of each state is `WHITE`. Because the threads in the naive implementation operate independently, we expect this version to show the worst performance, as the entire state space is explored N times, where N = *thread_count*.

3.3 MC_GC_NDFS: Global Color Variation

The first variation is found in file `src/main/java/ndfs/MC_NDFS_Thread.java`. In this version, a thread globally colors states red when `dfs_red` backtracks. This signals both red and blue searches that these states can be ignored, allowing other threads to prune them from their searches. We attach this variable, as well as a global `count` variable to each state, or node. The `count` variable is used to count the number of threads that have initiated a `dfs_red` search on state s.

3.4 MC_AR_NDFS: AllRed Variation

The second variation is found in file `src/main/java/ndfs/MC_NDFS2_Thread.java`. This version keeps the improvements of the first variation but adds a check in the blue search to ignore successor states that are colored red in order to prune the search.

3.5 MC_RS_NDFS: RedSet Variation

The third variation is found in file `src/main/java/ndfs/MC_CNDFS_Thread.java`. In this version, when a red search is initiated on seed node s, the node collects all reachable states and, if no cycle is found, waits for any and all non-red nodes in the set to become red, as they eventually will because no cycles had been detected during the red search.

3.6 MC_NEW_NDFS: Our Variation

The final variation is our own variation which can be found in file:

`src/main/java/ndfs/NEW_Thread.java`

We saw that the AllRed variation has the best performance when there is no cycle, so we want to try to improve it in some other way.

Our observation is that AllRed can detect if the connecting nodes of a node are red or not faster than the RedSet variation. Marking a node red means that there is no cycle that contains the node and an accepting node. In the AllRed variation, non-accepting nodes are marked red before `dfs_red` returns, and in the RedSet variation only marks a node red after an entire red search from the `dfs_blue` is finished. As the path branches in each node, RedSet marks a node red relatively late, and we observed 2x speedup for AllRed comparing to any other variation.

In RedSet, an optimization is that when a `dfs_blue` call finishes its calls on its successors, it prevents other threads from calling `dfs_blue` on the node. We believe this is helpful, since at this stage the only thing left for the `dfs_blue` call to do is initiate the red search. This seems expensive for the RedSet variation since it involves the usage of a set.

We can use a similar mechanism to improve the performance of other variations, so we chose the AllRed implementation and attached the mechanism on it. The correctness still holds since we still have at least one thread calling `dfs_blue` on the connections, and then the `dfs_red` on the node itself.

4 Results

We ran 900+ graphs on the algorithms such that each graph runs on each algorithm for 5 iterations, and we take the average time consumption. Graph generation is configured as stated in the graph generation section.

In Figure 1, we can see some lines forming up as time consumption increases. We conclude that it is due to property of random generated graph. The average speedup is 1.379, which meets our expectation that we can encounter some cycle sooner with more threads.

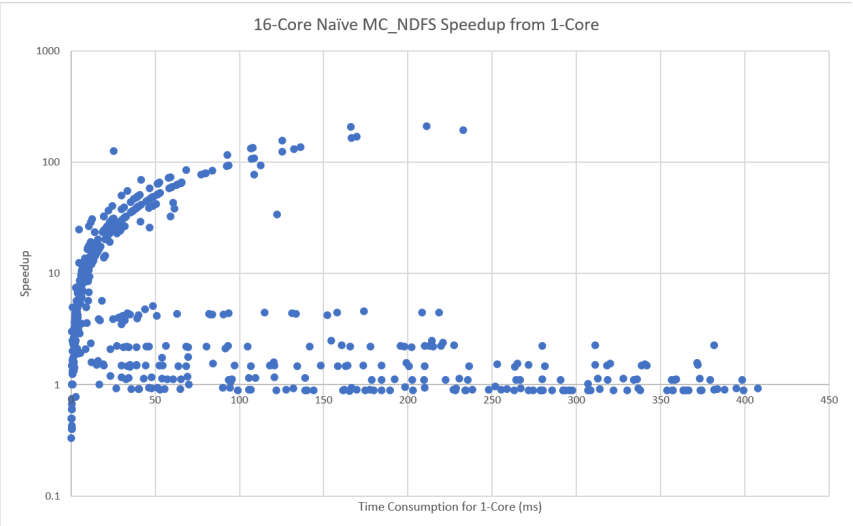

Figure 1: 16-core Naive_MC_NDFS vs sequential with cycles.

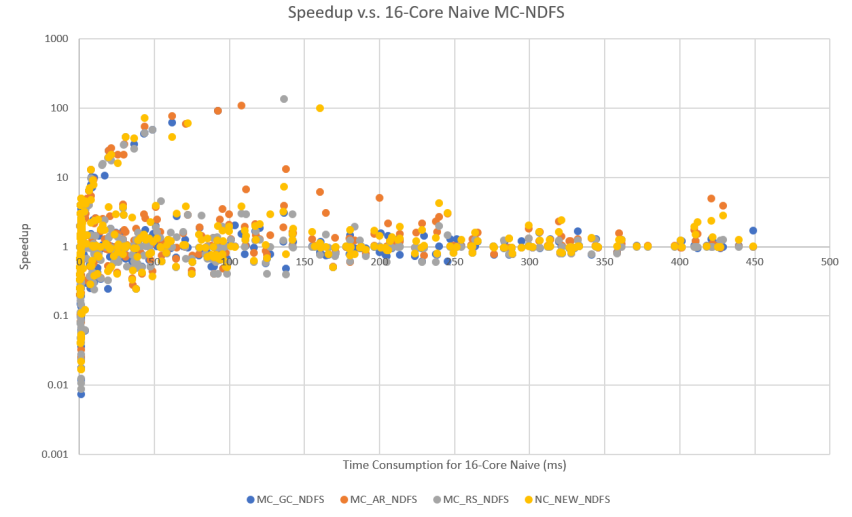

Figure 2: 16-core tests over graphs with cycles.

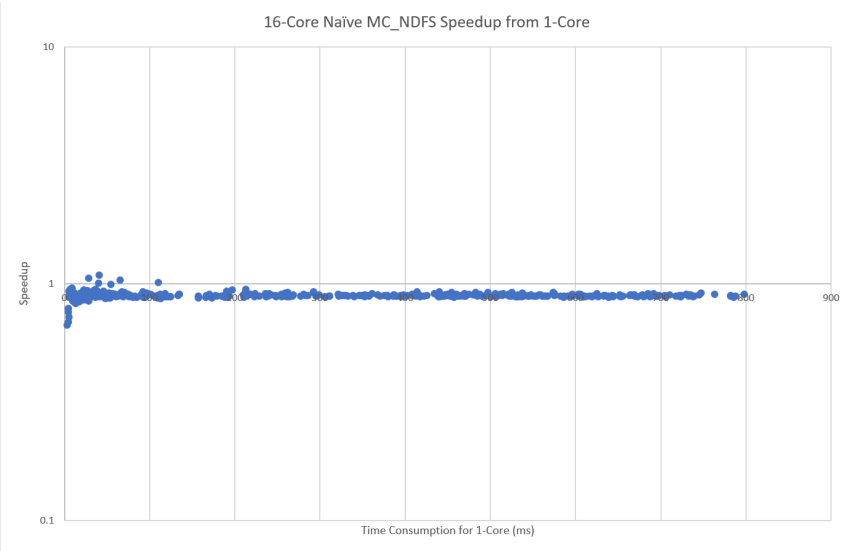

Figure 3: 16-core Naive_MC_NDFS vs sequential without cycles.

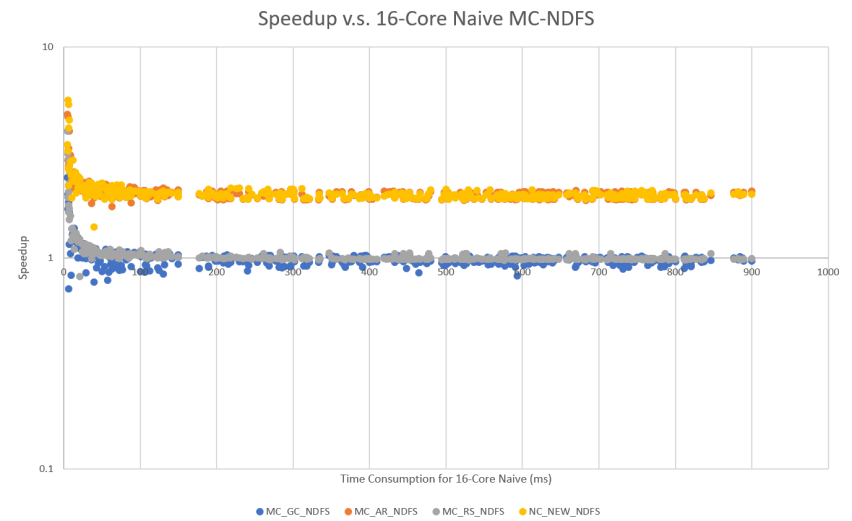

Figure 4: 16-core tests over graphs without cycles.

In Figure 2, we can see the improvement is relatively small from the 16-core naive implementation, which is a speedup of 1.3794. Since threads can encounter a cycle at a random time, we can see that there are chances for the improved variants to perform worse than the 16-Core Naive implementation.

The average speed up for MC_GC_NDFS is 0.9572, for MC_AR_NDFS is 1.1175, for MC_RC_NDFS is 0.9377, and for MC_NEW_NDFS is 1.0887. Two of the variants perform worse than the 16-Core baseline, which we believe is due to more collision between threads and the overhead introduced by accessing concurrent objects. The MC_AR_NDFS performs little better, which we think is due to preventing a thread from running `dfs_red`. Our new implementation is also slightly better than the naive 16-Core baseline, but not as good as MC_AR_NDFS. We think this is because the optimization idea we obtained from MC_RC_NDFS is not actually improving the performance, which is indicated by MC_RC_NDFS result.

Similarly in single core setting, the overhead of accessing multiple parts of the graph damages the 16-Core naive implementation's performance, resulting in a speedup of 0.8903, showing in Figure 3. However, the MC_AR_NDFS overcomes this by the same mark-red algorithm, which shows speedup of 2. The average speed up for MC_GC_NDFS is 0.9717, for MC_AR_NDFS is 1.9723, for MC_RC_NDFS is 0.9944, and for MC_NEW_NDFS is 1.9733, which is shown in Figure 4. Our implementation has received the same benefit since it is based on the MC_RC_NDFS, but it does not improve from there since MC_RC_NDFS doesn't shown improvement in speed.

5 Conclusion

We have presented several implementations of MC_NDFS that show improvement over the sequential implementation of NDFS. We tried to improve the performance of the best-performer, MC_RC_NDFS, with some insight gained from our observations of our test results and from our understanding of the various algorithms. However, in the end, our implementation yielded no observable improvement over the MC_RC_NDFS algorithm. We have also seen the linear-like results in the cycle speedup plot, which indicates cycles can sometimes be found very quickly. We think this has something to do with the characteristics of the graph, and the randomization may happen to run into some cycles right away for all of our 5 iterations per graph per algorithm.

Our code is available on GitHub at:

`https://github.com/NAP64/Multicore_Nested_Depth-First_Search.git`

References

[CVWY92] Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 10 1992.

[GS09] Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on buechi automata. *CoRR*, abs/0910.3766, 2009.

[LLP+11] Alfons Laarman, Rom Langerak, Jaco Pol, Michael Weber, and Anton Wijs. Multi-core nested depth-first search. volume 6996, pages 321–335, 10 2011.

[SE05] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 174–190, Berlin, Heidelberg, 2005. Springer-Verlag.