Design Report concurrency multithreading

Maximilian Danelli (mdi780), David Baumgartner (dbr318)
Vrije Universiteit Amsterdam
Department of Computer Science

23 September 2019

1 Introduction

This paper describes a naive solution for a multi-threaded program which takes a graph structure as input and detects whether any accepting cycles can be found in said graph. This kind of task is needed for *LTL-model-checking*, where we can create a *state graph* by converting the memory snapshot of any arbitrary program and feed it into our code. If we can detect a cycle within this graph which contains an *accepting state*, we can safely claim that the program corresponding to the memory snapshot will terminate. Our multi-threaded software achieves this by implementing an algorithm described in a paper by laarman et al. [1]. We will present a **naive approach** and an **improved version**, as well as showing better performance of the second version.

2 Previous work

At the start of the course, we were provided with a base program written by *Pieter Hijma*. It contained examples on how to implement multiple threads in a java program and a single core Implementation of laarman's algorithm [1].
We also implemented the algorithms described in this paper in our program. The single core implementation is the following:

```
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
```

Figure 1: single core algorithm; Source: [1]

As you can see, there are two procedures: *dfsBlue* and *dfsRed*. DfsBlue traverses the graph and looks for accepting states; if such a state is found, it starts the DfsRed, which checks for cycles by seeing if a node was already colored cyan, meaning that the blue procedure has been here recently. If such a node is found, it exits and terminates, else it marks the node red so it gets ignored in the future. After the blue procedure is done, it also marks the field which was cyan as blue, which makes it possible to mark it red.
Our task was to implement a multi-threaded version of this algorithm, by expanding on the already given program which we just explained. The procedure for the multi-threaded program works like this:

```
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
```

Figure 2: multi core algorithm; Source: [1]

The main changes in this algorithm compared to the last is that each blue procedure is started for each thread, and that each thread works on its own copy of the graph. The only shared data is the information of which states are red, and a counter variable which expresses how many threads are active on one accepting node. This is necessary because we have to wait until all threads have moved on from a node before coloring it red. Red nodes still mean that they can be ignored, which is way they are shared between threads: the area to search gets incrementally smaller. Furthermore, a new local color *pink* is introduced: pink nodes already pink nodes won't be explored by dfsRed again (since they are already on red's stack).

3 Design and implementation

3.1 Naive approach

Since the base program already provided us with a class for the different node colors. It consists of a map with the state as key and an enum for the colors as value, and methods to access the map. The *colors* class is created in each thread locally and acts as a copy of the state graph. We decided to implement the color pink within this class. However, we define a new, separate map for the pink values: the reason for this is that nodes in the algorithm by Laarman could theoretically be both cyan and pink at the same time: A different boolean field is used for the color pink, instead of assigning the value *pink* to the variable *color*. Furthermore, we created two additional maps which get passed to the single threads: a map assigning a boolean value to states called *red states* and a map containing the *counter variables* for said states. The critical section of our program is comprised of these two maps, which is why we used Java's *ConcurrentHashmap* for these two data structures. Furthermore, the counter map contains *AtomicInteger* values for additional security. It should be noted that this duplicate use of thread-safe data structures has been done for additional security. It isn't fast, but this is a naive approach, which should be improved further.

3.2 Improved version

4 Performance comparison

4.1 naive approach

The naive approach with two threads takes a total time of 2ms on the DAS4-cluster for *accept-cycle.prom* For comparison, the sequential solution takes 1 ms to detect a cycle for the same input. If we decide to run the naive approach with 10 threads, it takes 11ms. It becomes apparent that this solution comes with significant performance overhead.

4.2 improved version

5 Conclusion

References

[1] Alfons Laarman, Rom Langerak, Jaco van de Pol, Michael Weber, and Anton Wijs. Multi-core nested depth-first search. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, pages 321–335, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.