Design Report concurrency multithreading

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

October 14, 2019

1 Introduction

This paper describes a naive solution for a multi-threaded program which takes a graph structure and accepted states 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 recursively 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 having the color 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 imply that they can be ignored, which is why they are shared between threads: the area to search gets incrementally smaller. Furthermore, a new local color *pink* is introduced: 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

First, we make sure that each thread explores different parts of the graph by shuffling the lists of states returned from calling *graph.post()* on the current state s. We use *collections.shuffle()* from java.util.collections to achieve this. This is obligatory because otherwise the program wouldn't really be multi threaded.
We abort our search in **Worker.java** by writing to static volatile boolean for all workers called *terminate*. The boolean must be volatile because without volatile, threads will read the variables out of the cache and there the values for the variable could be not up to date. To avoid this problem we use volatile, forcing threads to directly write the variable into the memory. Each thread is checking *terminate* at the beginning of dfsRed() and dfsBlue() and throws an InterruptedException() in case it was set to true. We set the variable to true shortly before throwing a CycleFoundException() in dfsRed(), indicating a cycle was found.
The framework already provided us with a class for the different colors for the algorithm, which colors the nodes. It consists of a map with the state of the node as key and an enumeration for the colors (e.g. RED) as value, and methods to access and modify 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 creating a new enumerate value PINK and assigning it to the variable *color*. Furthermore, we created two additional maps which get passed to the single threads: a map assigning boolean values 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. It doesn't only allow concurrent access to the data structure and is thread safe without synchronizing, but also is faster than synchronized *HashMap*. Furthermore, the counter map contains *AtomicInteger* values for additional security and to

guarantee all operations on the counter are atomic. This is extremely important in concurrent environments. Different threads have to wait until the counter counts down to 0, we achieve this by busy waiting with a while-loop in our naive version. Our main improvement will be to use a **wait() - notify()** construct instead.

3.2 Improved version

This version differentiates itself from the naive approach in two improvements:

1. improved memory efficiency for the maps
2. removal of busy waiting

3.2.1 Improved memory efficiency

We improved the map containing all the red states, the map with the counters, and the pink map. The red state map used to be filled with the entire graph on initialization, it's value fields being set to false. The counter map was initialized with 0 values in the same manner. Our pink map would never remove values after they had been added.
We changed the red map check in a way that a null value is interpreted as false, therefore, states are only added to the map when they are colored red. Then it is no longer necessary to initialize the entire graph.
Our improvement for the counter map consists of dynamically initializing the the AtomicInteger. The will only get initialized when an accepting state is reached, since counters are only needed in these states (compare with multithreaded algorithm by laarman). We do this in the worker threads, at the beginning of **dfsRed** and **dfsBlue** functions.
Finally, we changed the pink map so that values are removed when set to false. Also null values are interpreted as false. This change is analogue to the change to the red map.
These modifications should decrease initialization times and make the program more secure for bigger graphs, since we might not be able to represent the entire graph in memory.

3.2.2 Removal of busy waiting

Our naive implementation contained the following code snippet in **Worker.java**:

```
if (s.isAccepting())
{
    thread_count.get(s).decrementAndGet();
    while (thread_count.get(s).get() > 0 ){} // wait until value is 0 again

}
```

This is obviously not very efficient, since the thread ends up checking on the value repeatedly and other threads cannot access this value in the meantime. We replaced it with the following code:

```
if (s.isAccepting())
{
    thread_count.get(s).decrementAndGet();
    synchronized (thread_count.get(s))
    {
        if (thread_count.get(s).get() <= 0)
        {
            try
            {
                thread_count.get(s).notifyAll();
            }
            catch(IllegalMonitorStateException e)
            {
            System.out.println("monitorexception in thread: " + this.getId());
            }

        }
        else
        {
            thread_count.get(s).wait();
        }
}
```

This version should be faster, since waiting threads will no longer continuously check the counter variable. With the new code, all waiting threads must be woken up by another thread. This will happen, when one thread sees, that the counter reached the value 0. After seeing the value to be 0, he will use the *notifyAll()* method to wake all other threads up.

4 Performance comparison

We tested our alogrithms with the promela files *bintree-cycle-min.prom* and *bintree-cycle-max.prom* provided. For the performance testing we decided to use the following number of threads for testing: 1, 2, 4, 8, 16, 32 and 64. Each algorithm was executed five times for each thread number to avoid outliers better. We also took the average of all five measures. The later presented data consists of the average waiting times for each thread number, algorithm and input file. For example, the waiting time for one thread with the naive algorithm for input file bintree-cycle-max.prom is 5978 ms. This value is the average of five calls with these parameters. Note, that the x-axis are the time in ms and the y-axis the number of threads.

These are our findings:

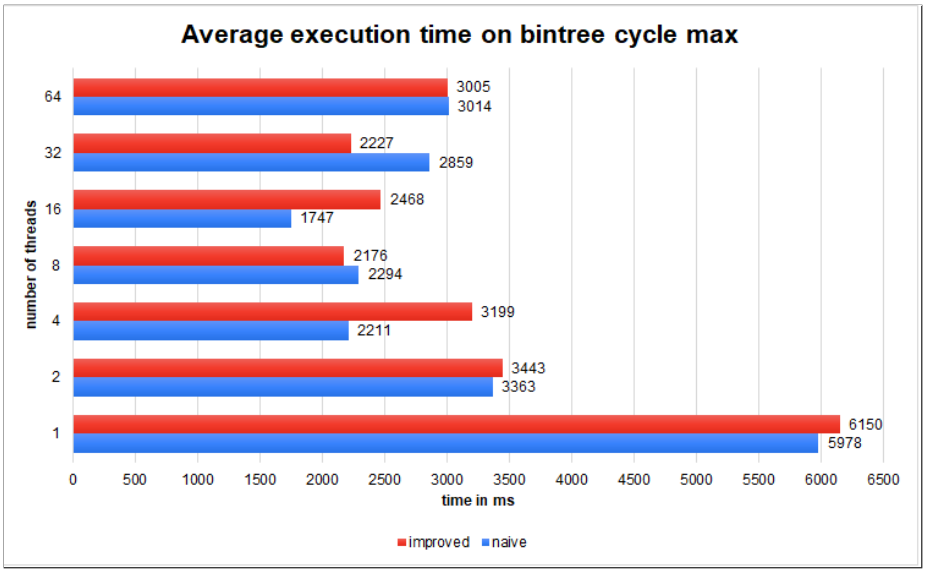

Figure 3: average times comparison on the cycle-max tree

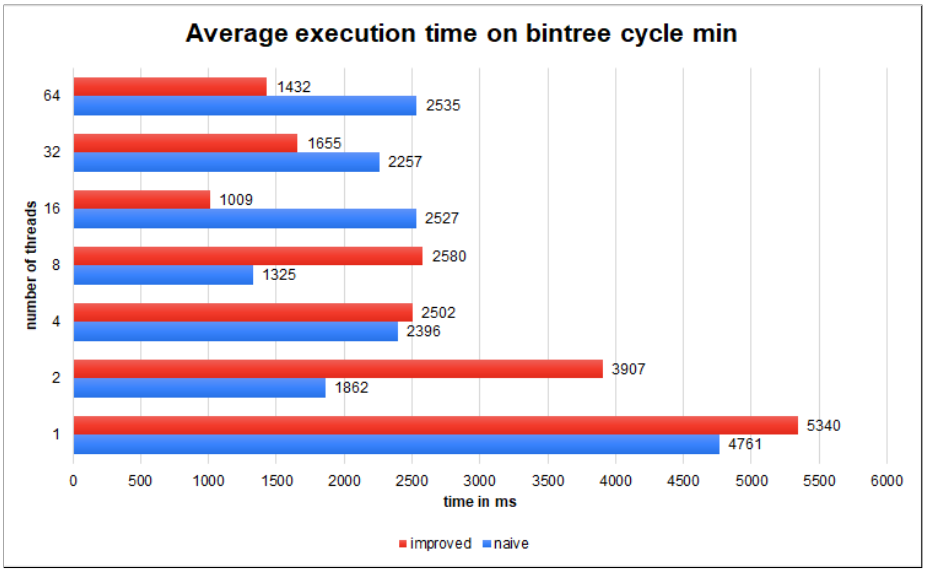

Figure 4: average times comparison on the cycle-min tree

We can see that for lower numbers of threads, the naive implementation can be even faster than the improved version. This is likely due to the fact

that notify()-wait scheme comes with a certain overhead, since java switches the state of a thread when calling wait() to a blocking state. Lower thread counts have a higher likelihood of not looking at a single node at the same time, therefore we cannot take advantage of the improvement and the overhead makes itself noticeable, slowing the improved version down. At higher thread counts, especially at around 32 threads, this is reversed, since the program will look at the same state's counter variable with different threads more often. Both versions are also generally faster when using bintree-cycle-min.prom as input:

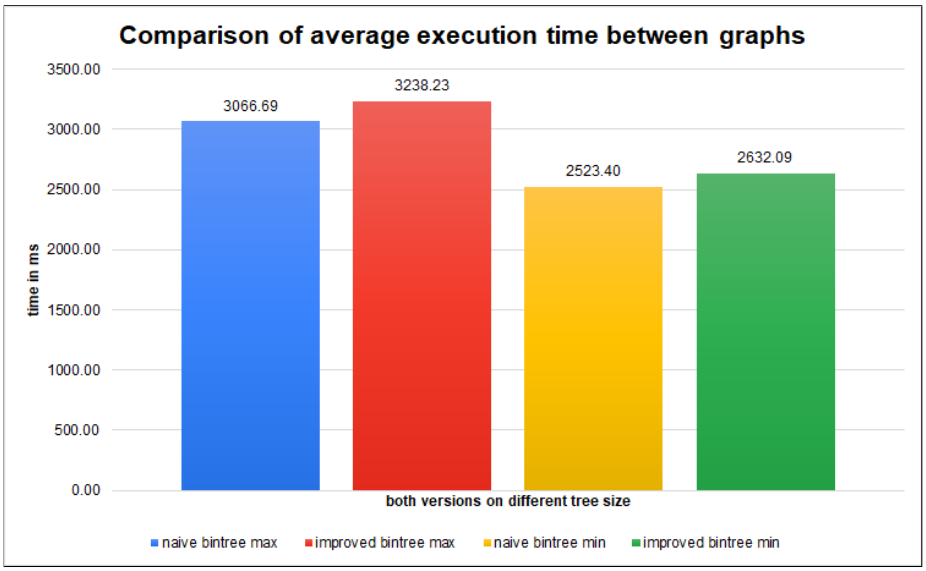

Figure 5: average time total for each input and version

It may appear that the naive version generally performs better when looking at Figure 5. However, we need to take into consideration that results for few thread counts might influence this statistic. More experiments with higher thread counts and more general executions of the programs could deliver different results.
We were not able to observe any performance changes due to our memory efficiency improvements, this could be because our input graphs aren't large enough for a noticeable difference. Program scalability should still be better, at least in theory. Further analysis is needed.

5 Conclusion

We implemented two versions of the algorithm of laarman, a naive version and an improved implementation. The improved program seems to perform better at executions with greater thread counts, while the naive variation gives better

results for a small number of workers. We could possibly observe better results when modifying the thread count range by leaving out results for one thread etc. and looking at bigger thread pools. This experiment could also deliver more accurate results with more input data at each thread number, to generate better average time measurements.

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.