The java implementation of a multi-core nested depth-first search. Concurrency & multithreading

Sam Ferwerda, 10446982, University of Amsterdam
Daan van Ingen, 10345078, University of Amsterdam

October 14, 2019

In this assignment we will program a multi-core nested depth-first search in java. To make this possible we will change the given sequential program such that it runs on different threads and will then be testing it on the DAS-4 supercomputer of the Vrije Universiteit. The major point of this assignment is to use Java locks for the first time. At first, these locks will not be in favor of the computation time as will be shown with data from different runs. However, the last part will be dedicated to adjusting one thing about the algorithm which will improve performance.

Shared variables

In the paper of Laarman, the algorithm for a multi-core program has already been given and shown in figure 1.

```
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 1: *Alg. 2 of the Laarman paper.*

This pseudo-code is helpful in determining what variables need to be shared between threads. Every variable that needs the threads ID *i* is a local variable and the others are global variables. Therefore we need to make sure that the **s.count** and **s.red** are shared. To make sure that this actually happens, we have made the colors *pink, cyan, blue and white* part of the colors class, while *red and count* are in another class called **SharedData**. The colors class will therefore not be initialised in the main thread, but in every single worker thread such that they all have their own colors instance.

The SharedData however is initialised in the main thread and then passed on to the worker threads, such that they all have the same instance of SharedData. To make sure all the threads do not make local adjustments to these variables **red** and **count** we have made these variables *volatile*. Red and count are both *Hashmaps* because it makes it easy to link a state to a Boolean or a State to an integer. However, when multithreading, we need to be careful when adjusting values in these hashmaps, but we will study this problem in another section.

Implementation in Java

In the main NNDFS function we create the SharedData class and the n different workers. Each of these workers need a promelaFile, defining the structure of the graph, and the SharedData instance just created. After that, the workers are put to work on different threads. Every worker activates their algorithm with the graphs initial state.

To make sure that the threads do not go down exactly the same path, we have included **collection.shuffle** to the original post function. This way we can try optimising without changing the initial state of the graph. The algorithm of every worker functions just as the sequential algorithm, however every time the worker needs to check or set a red (or check or set a count), it has to call a method in SharedData. The algorithm of the worker is therefore pretty similar to the algorithm of figure 1, only **t.red** is now actually **SharedData.getRed(t)**. If we want to do **s.red = true**, then we have to call **SharedData.setRed(s)**. This makes sure that the values are synchronized.

If a thread wants to make a state red it will not be able to do so until the counter reaches zero. Until that time it will be waiting in a while loop (or a wait statement in the improved version), such that the thread does not prematurely make a state red.

Main thread

While the worker threads are trying to find a solution, the main thread is constantly checking for results, see figure 2.

```
  while(!this.checkWorkers());

  pool.shutdownNow();
  for (int i = 0; i < this.numThreads; i++) {
    if (this.workers[i].result) return true;
  }
  return false;
}

public boolean checkWorkers () {
  boolean allDone = true;
  for (int i = 0; i < this.numThreads; i++) {
    if (this.workers[i].result) return true;
    if (allDone && !this.workers[i].done) allDone = false;
  }
  return allDone;
```

Figure 2: *The main thread is constantly checking for results and finished threads.*

The moment a worker thread finds a cycle, the thread will set his result to true. If the main thread then asks for this result and notices it being true he will command all threads to shut down and returns the value true or false depending on an actually found cycle. To make sure that the threads stop, we have placed if-statements containing **Thread.interrupted()** to check whether or not the thread should stop searching.

Note that the main thread is constantly checking for updates, this is a very busy way of doing so and it should be possible to let the main thread wait with a wait-statement until some thread has found a solution or is finished. Unfortunately, for now we had little time to do so.

Locking in the naive version

As we have already said in the shared variable section, we have to be careful when using a hashmap over different threads. We do not want the hashmap to get corrupted by two threads writing to the same state at the same time. To make sure this does not happen, we have implemented two locks: a *RedLock* and a *CounterLock*. Before any thread can get or set a value in **red**, it will have to acquire the RedLock first. This is the same for the CounterLock and the **count** variable.

Even though this method of locking is save, it hurts the performance. Every time a thread wants to change a value it needs the global lock, which blocks all the other threads from working. These threads might not even be working on the same state as the thread who acquired the lock and therefore slow everything down unnecessarily.

Locking in the improved version

We would like to adjust values for two or more different states simultaneously and a global lock will not suffice in this case. In particular a Hashmap is not very thread-safe and therefore java released the **ConcurrentHashmap** which is similar to a normal Hashmap. The advantages of a ConcurrentHashmap is that no two threads can write to the same *key* at the same time, but it allows writing to two different keys at the same time and reading from one key at the same time. Note that reading at the same time is fine, since it does not change the value. So the only thing we have to do is change the **red** and **count** hashmaps from the naive version and make them **ConcurrentHashmaps**.

We do not need to acquire a global lock, or use synchronized blocks since it already part of the ConcurrentHashmap. Therefore it does not only improve computation time, but also significantly reduces the amount of code needed to do the same thing. This is the one adjustment we made in our code and the difference in speed can be observed in the performance section.

Due to the feedback stating that our workers were waiting in a busy way, we have also adjusted the way our workers are waiting for the **s.count** to reach zero. Before the adjustment they were constantly calling the function **SharedData.getCount(s)** and checked whether or not this is zero. In the improved version this still happens, but inside this while loop there is a wait statement, which will put the thread to sleep until there is an update in the count variable or he has been waiting for 500ms.

Performance

In this section we will talk about speed and scalability for the naive version and the improved version. If you do speed checks, you could set a random seed before comparing. However we found this not really presentable since one could then just take the seed corresponding to the biggest change. Therefore all speeds are measured using the rounded mean of 15 runs.

In this report we will use the bench-deep.sumo file since this one does not have accepting states and the sequential algorithm can not seem to compute it due to stack-overflow errors. We will also use the bintree-cycle-min.prom since it takes a while to run on the sequential algorithm and contains a cycle and finally bintree-converge.prom since it does have accepting states but no cycles. We will present them as **bench, cycle and converge** in further research. Before we can show meaningful results of the multi-core algorithms, we should know the computation time of the sequential algorithm and therefore we have added table 1.

Bench	NaN
Cycle	4350
Converge	5500

Table 1: *Sequential computation time in* ***ms***

Performance of the naive version

Let us first discuss scalability since this is pretty obviously not the case. We will not achieve any better computation time just by adding more threads since we are using a global lock. The more threads we use the more threads will be blocked and progress will slow down. This hypotheses is partly supported by the following table.

no. threads	2	4	8	16
Bench	2100	5000	12300	33500
Cycle	3800	7600	8800	7400
Converge	12200	14300	13700	13000

Table 2: *Scalability of the naive algorithm, time is in* ***ms***

In the bench graph it is clear that the increase of threads is increasing the computing time. This makes sense since there are no cycles in this graph, the whole graph needs to be searched. Furthermore, there are no accepting states in this graph and therefore no state is set to red and therefore a multi-threaded algorithm is worthless here. In the cycle graph we can see that the run with 2 threads performed best, even better than the sequential algorithm. However, as we added threads the performance became worse with a strange result for 16 threads. The best educated guess we can make is that the cycle graph does have multiple cycles. Then 16 threads would be beneficial, even with the blocking of other threads, as long as at least one thread goes to a cycle quickly it could perform better than a 8-thread run.

Performance of the improved version

Now that multiple threads can access different states at the same time, the scalability should be a lot better. The only thing synchronized is the initialisation of the counter per state, since that seemed to be causing deadlocks. Nevertheless, the results are astonishing. See table 3.

no. threads	2	4	8	16
Bench	2100	3700	7900	18900
Cycle	3800	3600	3100	4200
Converge	7800	6300	6500	5300

Table 3: *Scalability of the improved algorithm, time is in* ***ms***

Every single run is better than the naive version and for the converge and cycle graphs we even see a speed up. Unfortunately, the sequential algorithm runs the converge file in 5500ms, so our improved version does only little better than the sequential algorithm when using this graph. However, it is 25% faster than the sequential algorithm when looking at the cycle graph!

Evaluation

In our **naive version** we used global locks and that did not make the algorithm very efficient. Threads constantly need to compete for the lock even though they are not working on the same state. This made running multiple threads inefficient. This algorithm would perform decent with low amounts of threads and when a cycle is easy to be found. This way, the blocking could be counterbalanced by the increased chance of finding a cycle and that could potentially give a minor speed up.

In our **improved version** we used the ConcurrentHashmap to replace the lock methods we implemented in the naive version. This hashmap allowed different threads to work on different states simultaneously and therefore increased in speed significantly. We have observed speed ups for the cycle and converge graph in table 3. The improved version is useful in a lot more cases than the naive version. Even though this version is faster, it should be possible to make it even better and more efficient. The busy waiting in the main thread should be changed to a wait-statement and we believe that there would be an advantage in spending more time in splitting the search space. Currently this is done by shuffling the post states, but with bad luck this can still result in a bad performance.

Furthermore, threads do no block each other as often as in the naive version and therefore increasing the number of threads can lead to a speed up. However, the graph should be big enough for these threads to actually not go down the same path. We believe this is the reason why we can see an increase in computation time in table 3 at the cycle version: 8 would be ideal since they would not get in each others way (if we could only choose between 4, 8 and 16).

Overall, the improved version is a pretty efficient multi-core algorithm with good performance. With some results better than the sequential algorithm and that is what multi-core programming is all about.