Amin Soleimani (2640338)

What data needs to be shared, and what doesn't?

- Shared:
  - Red color for each state is shared between threads(in the backtrack of dfs_red).
  - Counter that keeps track of threads initiating dfs_red on the same state. Threads should wait until count is zero and all finish at the same time, otherwise accepting cycle might not be detected.
- Local:
  - Other colors (cyan,white,blue,pink) are thread local

How are you going to prevent threads from searching the same area of the graph?

The performance of NDFS depends on the randomization of successor node from post(state). If states are not randomized correctly, threads will search the same area of the graph.
The list of states returned by post(s) is shuffled by calling Collections.shuffle with ThreadLocalRandom.current().nextLong() as seed.

Seeds that did not work well:
thread id did not provide enough randomization.
System.nanoTime introduced bottleneck.

How will you terminate the search and your program?

After submitting tasks, take() methos of ExecutorCompletionService is called, which blocks until the future is returned (). Futures return in the order of completion.
When a task is returned it checks if the result is true (by calling get() method of the future to get the worker, and then calling getResult method of the worker), which indicates a cycle has been detected.
When a worker detects a cycle, it throws CycleFoundException, which is caught in the call method, which sets the result and returns. Next shutdownNow() is called which attempts to stop all actively executing tasks by interrupting them.

Initially I was checking Thread().currentThread().isInterrupted() inside dfs-blue and dfs-red, which worked fine. However, it would still take a few seconds for the threads to terminate.

I use volatile Boolean variable "done" in my implementation for signalling to other threads that a cycle is found.

1_naive:

Which data structures are you going to use? What are the advantages and disadvantages of these?
For this implementation I use hashmap for both counter and global red.
Advantage: fast and easy to use.
Disadvantage: no synchronization provided

Initially I used Integer values for the counter hashmap. This was not a good choice since Integer is immutable and every update creates a new object. I later changed it to atomicInteger.

When and where do threads need to synchronize? The paper states that lines of pseudo-code are executed atomically. How does this influence your implementation?
Since this version uses hashmaps, all reads and writes to the shared datastructures need to acquire the lock first.
In this implementation I used ReentrantReadWriteLock for both hashmaps.
I also used a condition for signalling when counter has reached zero.

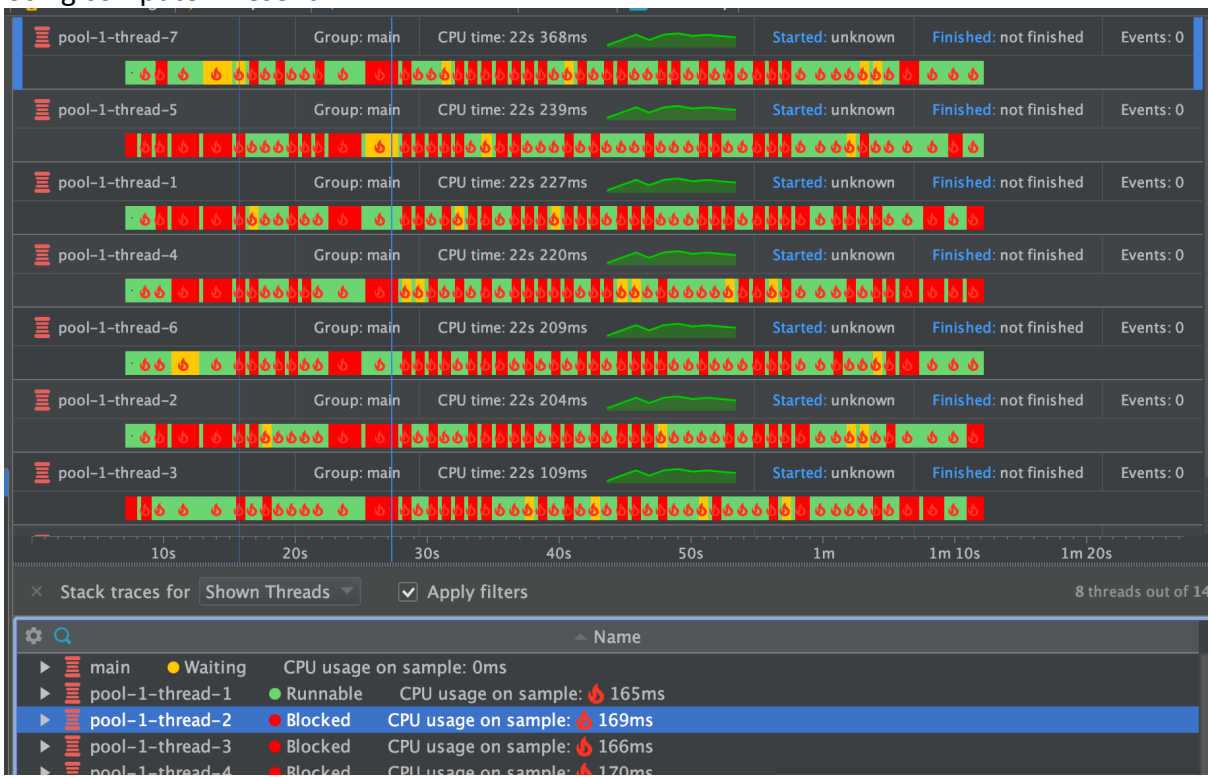
Based on the algorithm, how well do you expect your implementation to perform? What kind of graphs would the algorithm perform well on? On which would it perform less so? What about your naive version? And your improved version?
I expect this version perform worse than the sequential version due to the overhead of locking and the sequential bottleneck caused by locking the entire map.
If the number of threads is small (lower contention), there might be some improvements in performance of trees with deeper depth and accepting cycle, due to the randomization of post function.

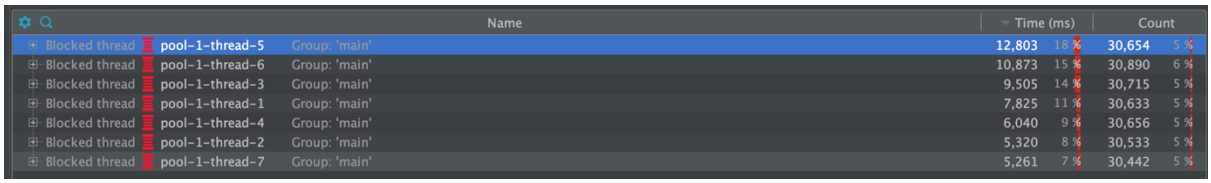
2_naive:

In this version I used ConcurrentHashMap for both shared data structures.
Advantage: provides some synchronization. The reading and setting the Boolean in the global red state map is done without any difficulties.

Disadvantage:
Updating the counter should be atomic. ConcurrentHashMap provides methods that are guaranteed to get performed atomically.

Unfortunately, some of these methods hold map's monitor while iterating the map and cause other threads to block.
Using computeIfPresent:





Using compute ("Attempts to compute a mapping for the specified key and its current mapped value"):

Expectation:

- Good speedup in reading/writing of red states' map.
- Improved performance for the counter compared to the 1-naïve

I don't expect the overall speedup to be noticeable since threads have to synchronize on the map in order to wait for the counter to become zero.

3_improved:

avoids blocking other threads by synchronizing on the map key instead.
In this implementation I used a custom mutableInteger class for the counter.
My idea was to use a condition and reentrant lock in each value object, and use getWaitThread() method to determine of it is safe to remove the mapping when counter reaches 0. This did not work since java doc states this method should not be used for synchronization control.
In this version mappings are not removed after counter reaches 0, since there is no way to know if all waiting threads have returned and no signal is lost.

I expect this version perform better when trees have higher depth(and there is a cycle in the lower levels).

4_improved: added all red for detection in dfs-blue