1 Performance of Models

I ran my tests on the SEASnet GNU / Linux with Java version 1.8.0_51. I tried gathering memory information and cpu information by running /proc/meminfo and /proc/cpuinfo respectively but I did not have those permissions on the servers.

1.1 Synchronized

java UnsafeMemory Synchronized 4 1000000 6 5 6 3 0 3 - **1599.20 ns / transition**
java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3 - **2930.15 ns / transition**
java UnsafeMemory Synchronized 16 1000000 6 5 6 3 0 3 - **4930.45 ns / transition**
java UnsafeMemory Synchronized 8 1000000 12 5 6 3 0 3 - **2976.33 ns / transition**
java UnsafeMemory Synchronized 8 1000 6 5 6 3 0 3 - **33684.4 ns / transition**

The provided SynchronizedState class is a 100% reliable and DRF method because it utilizes Java's synchronized keyword which prevents thread interference and allows for atomicity when writing to variables. However, by ensuring this, it is also a very slow method.

1.2 Null

java UnsafeMemory Null 4 1000000 6 5 6 3 0 3 - **432.974 ns / transition**
java UnsafeMemory Null 8 1000000 6 5 6 3 0 3 - **1934.32 ns / transition**
java UnsafeMemory Null 16 1000000 6 5 6 3 0 3 - **4237.49 ns / transition**
java UnsafeMemory Null 8 1000000 12 5 6 3 0 3 - **2060.19 ns / transition**
java UnsafeMemory Null 8 1000 6 5 6 3 0 3 - **29488.6 ns / transition**

The NullState class is reliable in the sense that it passes all the cases. As expected, it outperforms Synchronized in every test case since it just returns dummy values.

1.3 Unsynchronized

java UnsafeMemory Unsynchronized 1 1000000 6 5 6 0 3 - **51.0469 ns / transition**
java UnsafeMemory Unsynchronized 2 1000000 6 5 6 0 3 - **freezes**
java UnsafeMemory Unsynchronized 2 100000 6 5 6 0 3 - **1013.94 ns / transition**
java UnsafeMemory Unsynchronized 4 1000 6 5 6 0 3 - **14679.0 ns / transition**
java UnsafeMemory Unsynchronized 8 1000 6 5 6 0 3 - **32599.8 ns / transition**

The UnsynchronizedState class is not DRF, since it has the same code as SynchronizedState but does not use the synchronized keyword. It also is not reliable and runs into frequent sum mismatches, especially as you parallelize and increase the amount of threads, like I did in the last two test cases.

1.4 GetNSet

java UnsafeMemory GetNSet 2 1000 6 5 6 3 0 3 **9208.09 ns / transition**
java UnsafeMemory GetNSet 4 1000 6 5 6 3 0 3 - **19755.2 ns / transition**
java UnsafeMemory GetNSet 8 1000 6 5 6 3 0 3 - **39110.5 ns / transition**
java UnsafeMemory GetNSet 8 10000 12 5 6 3 0 3- **10305.3 ns / transition**

I implemented this class using Java's AtomicIntegerArray. Although the get and set methods of the class are volatile / atomic by themselves, they are used together in the code and therefore can cause race conditions. As a result, it runs into frequent sum mismatches as well. Comparing the 3rd test case to the last tests for Unsynchronized and Synchronized, it seems that although GetNSet is between Unsynchronized / Synchronized in implementation, it is slower than both.

1.5 BetterSafe

java UnsafeMemory BetterSafe 4 1000000 6 5 6 3 0 3 - **570.405 ns / transition**
java UnsafeMemory BetterSafe 8 1000000 6 5 6 3 0 3 - **1208.86 ns / transition**
java UnsafeMemory BetterSafe 16 1000000 6 5 6 3 0 3 - **2462.92 ns / transition**
java UnsafeMemory BetterSafe 2 1000000 8 5 6 3 0 3 - **740.888 ns / transition**
java UnsafeMemory BetterSafe 8 1000 6 5 6 3 0 3 - **41923.0 ns / transition**
java UnsafeMemory BetterSafe 8 100 12 5 6 3 0 3 - **295414 ns / transition**

This method is DRF and also reliable. Instead of using the synchronized keyword, I used Java's ReentrantLock to ensure thread safety. After some reason, it seems that ReentrantLock often performs better than using synchronized because it has extended capabilities, like lock polling, interruptible lock waits and also a configurable fairness policy. This allows more multiple condition variables per monitor while synchronized only have one wait / notify queue.

1.6 BetterSorry

java UnsafeMemory BetterSorry 2 1000000 8 5 6 3 0 3 - **413.319 ns / transition**
java UnsafeMemory BetterSorry 8 1000 12 5 6 3 0 3 - **27289.3 ns / transition**
java UnsafeMemory BetterSorry 8 100 12 5 6 3 0 3 - **210935 ns / transition**

This method uses an array of AtomicIntegers instead of ReentrantLocks so as expected, it performs better than BetterSafe since locks are slower to use. It is also more reliable than Unsynchronized because it uses AtomicInteger's atomic operations to increment and decrement.

The first test case would hang forever on Unsynchronized so this avoids that race condition. However this method still has issues, e.g. while you're getting the value at an

index, it can be changed in the period between the get and the increment / decrement. I recreated this by making my initial values all 0's like so: java UnsafeMemory BetterSorry 4 1000000 8 0 0 0 0 0.

2 Conclusion

I would suggest using either BetterSafe or BetterSorry for GDI's applications. The safer but slower route would be use BetterSafe but the spec mentions that GDI does not mind "a small amount of breakage" so BetterSorry would also work, and plus, it's faster.