Alvin Lim
804 011 675
CS 131 Eggert

Homework 3 Report

System Information: *Java Version:* 1.8.0_45
CPU: Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz
Memory: 32 GB

Overview:

For this assignment, we were given a simulation program which measures the reliability and performance of different implementation models running through Java's shared memory model. We had to implement 4 new models, "Unsynchronized", "GetNSet", "BetterSafe", and "BetterSorry" in addition two the two given models, "Synchronized" and "Null". Each of these models had a different way of performing state transitions, in which a transition meant (1) adding 1 to a random element in an array, not to exceed a max value, (2) subtracting 1 from another random element in an array, if the element isn't negative; when done ideally, the sum of all the values in the array should always remain constant after a swap. When taking memory-synchronization issues into account, however, the transactions may be done incorrectly, and the sum may not be constant.

Several tests were done under each model under varying conditions for number of threads, number of transitions, size of state array, and sum of values in the state array. These results can be used to characterize the performance of the different models. I found that generally, as long as the size of the array is large enough that there is less chance of collisions between the different threads, the size of the array stops becoming an effecting factor on the performance and reliability after a point. One just needs to increase the array size as thread count goes up. Similarly, I found that for the state array sum, it didn't seem to affect results as much, except for cases when members of the array were close to 0 or maxval, since at those points, we have to do a lot of rerandomizing for the transitions. As a result, for this report, I stuck to a set array of 30 values that I randomly generated values for, to show general trends with changes in thread count and transition count. I used the following randomly generated 30-element array: [100 30 45 75 11 98 94 100 108 1 8 117 14 90 32 117 94 68 12 81 23 49 15 65 64 100 35 21 22 13] on the given "Null" and "Synchronized" models on various thread and transition counts, in order to see what effects these had with the JMM.

Model	nThread	nTransition	ns/transition	Model	nThread	nTransition	ns/transition
Null	1	1000000	38.6567	Synchronized	1	1000000	89.9203
Null	2	1000000	134.041	Synchronized	2	1000000	465.721
Null	4	1000000	329.323	Synchronized	4	1000000	1272.07
Null	8	1000000	1566.8	Synchronized	8	1000000	2935.05
Null	16	1000000	3577.99	Synchronized	16	1000000	4861.05
Null	32	1000000	7621.19	Synchronized	32	1000000	8669.04
Null	8	100	30749	Synchronized	8	100	322028
Null	8	1000	41331.3	Synchronized	8	1000	43661.1
Null	8	10000	7766.58	Synchronized	8	10000	10069.2
Null	8	100000	2053.46	Synchronized	8	100000	4849.63
Null	8	1000000	1282.41	Synchronized	8	1000000	2694.43
Null	8	10000000	130.346	Synchronized	8	10000000	2658.65

We can see that as the number of threads increases, there are more threads for the JMM to account for, causing more overhead in between transition. With just one thread, the JMM doesn't have to worry about conflicts with other threads, and performs the best for this simple transition. On the other hand, in looking at how number of transitions affects the performance time, we have to consider that there are other operations that take place other than the transition, such as the work needed to set up and use the for loops, etc. This time is included and then distributed evenly over all of the transitions; with less transitions, more of that time is considered for each transition.

Model Analysis:

To analyze and compare the different models, I ran them each (except Null) through a set test with 10 threads, 1000000 transitions, and the same array as the previous test: [100 30 45 75 11 98 94 100 108 1 8 117 14 90 32 117 94 68 12 81 23 49 15 65 64 100 35 21 22 13] and averaged their results over 10 runs each.

Model	Synchronized	Unsynchronized	GetNSet	BetterSafe	BetterSorry
Average Time (ns/transition)	3689.491	2090.9345	2641.166	1835.129	1753.721

Synchronized Model
The given Synchonized state model is reliable and DRF because it used the "synchronized" keyword in the swap method, which guarantees that only a single thread can modify the array elements at a time. This means that no two conficting accesses can happen, so it is DRF. The synchronized execution, however, causes this model to run the slowest.

Unsynchronized Model
This model is the same as the Synchronized model with the "synchronized" keyword removed from the swap function. As a result,it is unreliable because there is no synchronization and atomic operations. This made the Unsychronized model not DRF and a victim to race conditions. In my test cases, the Unsynchronized model had an average error percentage of 31.34%. It is faster than the Synchronized model, however, since it doesn't have to deal with the overhead of synchronizing the different threads.

GetNSet Model
GetNSet was implemented using AtomicIntegerArray and its functions get() and set() to handle synchronization. Its speeds are in between that of the Synchronized model and the Unsynchronized model. It is not DRF though - since get() and set() are two separate functions, there is still the chance for data races, because the threads atomically get() each of the elements to be swapped, but it doesn't do set() concurrently, so threads could be interweaved in a way so that two or more threads may alter the same memory location. My test case yielded an average error percentage of 36.37%.

BetterSafe
For this model, I used a lock instead to implement synchronization. The lock would block out other threads from using the swap method at the same time – similar to how synchronization keyword would work. Since its implementation is much more simple though - by completely saying "no other threads may use this" rather than "how can we synchronize this with how everyone else is working so that there aren't any conflicts" – the overhead is much lower so the performance speed was much better, and since one thread executes swap() at a time, the it is DRF and reliable.

BetterSorry
In the BetterSorry model, I implemented it similarly to GetNSet, but it uses an array of AtomicInteger rather than an AtomicIntegerArray. In addition, because of the problems with data races using get() and set(), I used getAndDecrement() and getAndIncrement() instead. Using AtomicInteger instead of AtomicIntegerArray, instead of threads having exclusive access of the whole array whenever using it, this allows threads to have exclusive access to *members* of the array, allowing for concurrent use of the array (providing different elements) and therefore better performance. This model is still not supposed to be 100% reliable, but I failed to find a program to create any data races. Theoretically, the values of elements could change in the window between the time they are being checked against the range limits (0 and maxval) and when the decrement and increment occurs. I guess this is extremely rare, however, because I've run the test on many different circumstances and have not encountered it yet.