Homework 3 Report

I) Discussion of Implementation Strategies, Optimizations, and Challenges

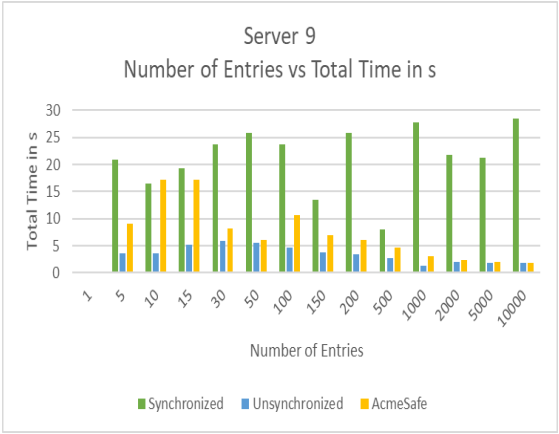
The basic idea of the final version of my AcmeSafe implementation involves declaring an Atomic Long Array which is private to the AcmeSafeState class, and utilizing the built-in atomic functions offered by the AtomicLongArray class to safely ensure no data races are occurring, while maintaining efficiency. In essence, I got rid of long[] value as a private variable to the class; this array was no longer needed, because we used an object of type AtomicLongArray to store the long values instead. In the AcmeSafeState constructor, we initialize an AtomicLongArray object with the given length. In the getter method size(), we simply return our array's length, by calling one of the AtomicLongArray's methods, length(). Modifications were necessary within the method current(), as the return type needs to be an array with elements of type long, whereas our private variable is an object of type AtomicLongArray. So, within current(), we initialize an array of long elements that has the same length as our AtomicLongArray, and then we utilize a loop to iterate over the elements in our AtomicLongArray and copy them into our initialized long array. This implementation might at first seem like an inefficiency, as we must physically create a new array and transfer all of our AtomicLongArray's elements into it each time we call current(). However, this turns out to be quicker and more memory-efficient than if we were to also declare a private member long array and update it as we update our AtomicLongArray, in essence maintaining two arrays. Because current() now has high overhead, programs in which the number of times current() is called is greater than or equal to the number of times swap() is called will have lower efficiency than the two private members strategy. However, since we are dealing with programs that are calling swap() millions or billions of times, in comparison to only a few calls to current(), the trade-off is much more efficient for my implementation, where only one private member is maintained. For the swap() function, we can call the built-in class functions decrementAndGet() as well as incrementAndGet(), which safely and atomically decrement/increment our values, preventing a race. Some optimizations I implemented included getting rid of the long array private member and just maintaining the AtomicLongArray. In addition, in the current() class method, I set a variable equal to length() outside of the loop, which completely eliminated the overhead of calling length() on the AtomicLongArray during every iteration of the loop. This caused a significant increase in performance. Next we discuss some challenges I encountered while implementing AcmeSafe. The first version of my function maintained only one private member variable, an array of long elements. The only change I made was within the swap() function, where I declared an AtomicLongArray with elements copied from our long array. Then, I used the AtomicLongArray decrementAndGet() and incrementAndGet() methods to increment/decrement, and finally I set the values at these indexes in the long array equal to the new values from these function calls. However, this caused a data race / output sum mismatch. Further, I introduced a private final ReentrantLock member, which I initialized upon construction. This, I locked before calling decrementAndGet() and incrementAndGet() and unlocked afterwards. However, this also resulted in an output sum mismatch. I moved on to the idea of instead introducing a private member AtomicLongArray object in tandem with the long array, and maintaining both, but using the object's atomic operations in swap() to avoid data races. This, however, still caused data races. Therefore, I moved on to the final implementation, in which only a private AtomicLongArray is maintained, and to satisfy the return type of current(), a long array is created only within that method.

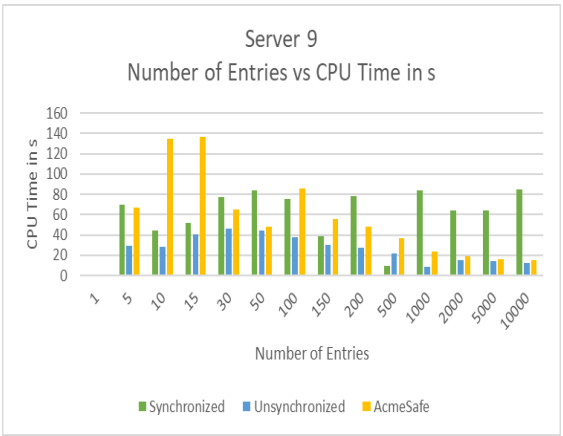
II) DRF Characterization and Discussion in Terms of "Using JDK 9 Memory Order Nodes"

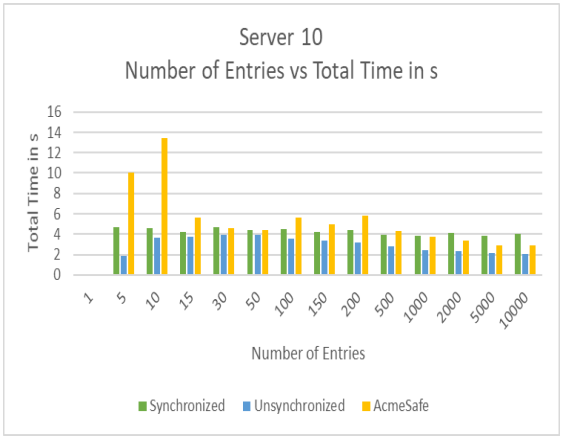
To be DRF, or data-race-free, it has to be true that the parallelized program does not have any conflicting access events, in Lea's terminology, without synchronization in between. Since we know that a conflict is essentially when two access events occur at the same place and from differing threads, the access location is not

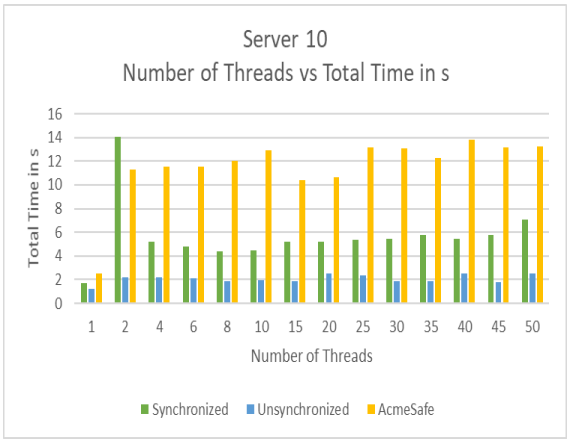
volatile, and the access events are not both Reads (there is a Write), then since my program utilizes an AtomicLongArray object and this class's methods to implement atomic operations, we must look at the class to ensure that these conditions are met. The AtomicLongArray class uses atomic operations, obviously, where atomic operations are taken to be quick operations in which a process both reads and writes at a location simultaneously in a way so that no other process can conduct reading or writing until after the first atomic operation has been returned, which should ensure DRF conditions. This is in opposition to the synchronization used by SynchronizedState, which is much less efficient. Going more in-depth on atomic operations in reference to creating a data-race-free environment, we can characterize atomic operations as using CAS (compare and set) operations which are at the machine level. These CAS operations work without causing the sequential blockage that locks cause, by looking at a memory address and, if the value there is what it was before (no conflicting access and write has occurred in the meantime) then it updates the value in the memory location with whatever operation was specified. If the value there has been changed, the value of the variable at that memory location will not be updated, but will still be returned. So, when we have a parallel, multi-threaded program, multiple threads can use a CAS operation on a memory location at the same time. None of the threads will be locked out and stopped from continuing to work. Instead, the first thread that gets to the value will update that value, and the rest will not, and will continue on. If necessary, the threads that failed can run the CAS operation again. So, these atomic operations not only deal with the problem of synchronization, but are much more efficient, because no lock-like sequential blockages are caused, and with one machine operation we can do three different things: looking at the value, seeing if it is safe to be updated, and updating. This, therefore, is the key to how the AtomicLongArray in my implementation created DRF conditions, by using atomic operations in the incrementAndGet() as well as decrementAndGet() calls. This follows from some of Lea's thoughts, especially those which touch upon low-level processing parallelization in opposition to multithreading, where these atomic operations are embedded in machine instructions which integrate out-of-order execution as well, beyond the threads that we are using at a higher level. In addition, while Java has types that are primitively bitwise atomic, such as int and char, long is not one of them, which is why our explicit usage of AtomicLongArray was necessary. The article also suggests that the java.util.concurrent classes, of which AtomicLongArray is one, operate with Release/Acquire mode, though this does not eliminate races when multiple threads attempt to write to a variable simultaneously, which is why our atomic operations were necessary. And, according to Lea, the CAS operations are what are known as Universal Consensus operations, which extends the Consensus-2 idea of RMW (read-modify-write) to checking with the expected value, as was explained above.

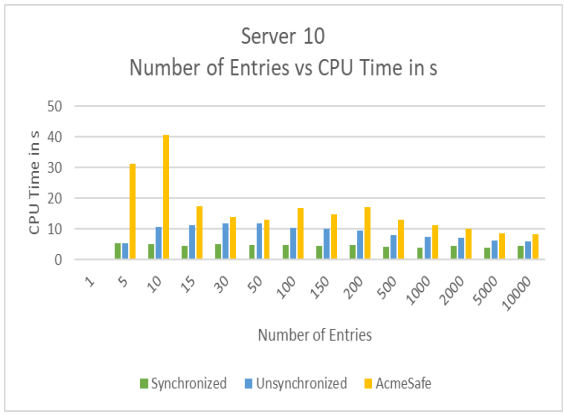
III) Comparison Measurements and Graphs/Tables for AcmeSafe, Synchronized, & Unsynchronized Performances

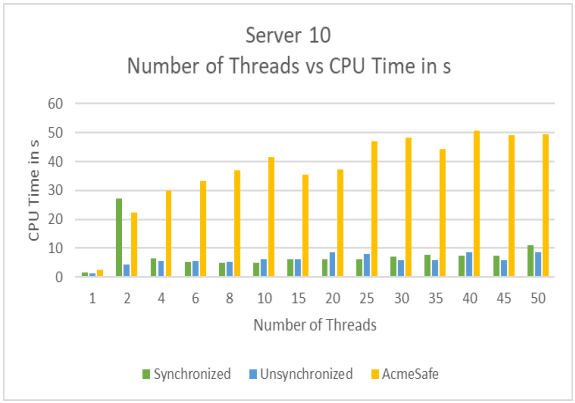


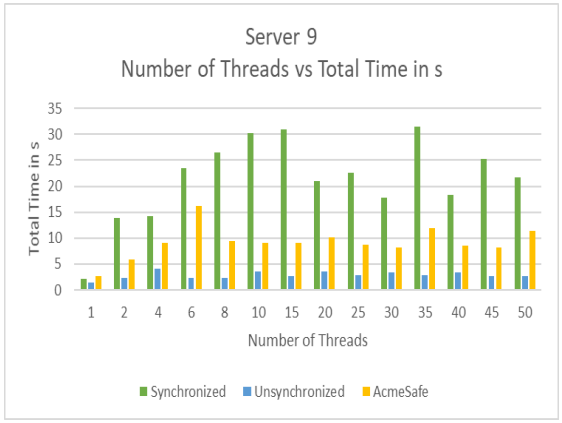


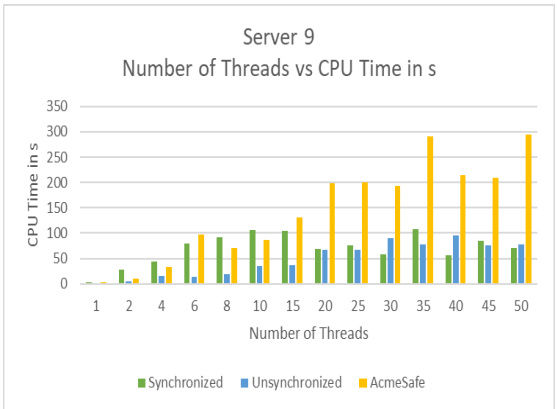












IV) Discussion/Analysis of Comparison Measurements

We can split our analysis into two sections: varying of the number of threads and varying of the number of entries. For varying of the threads, we seek out generalizations based on the data to characterize the performance of the three implementations by comparing them. As we can see in the above graphs, for both total time and CPU time, the Unsynchronized version has the fastest speeds for every single number of threads the program was run with. This would make sense. The more critical comparison is between AcmeSafe and Synchronized, where we see that one or the other performs better in different scenarios. On the whole, AcmeSafe performs quicker than Synchronized for most thread values. There is only one scenario (where there is only one thread), in which Synchronized performs faster than AcmeSafe, with, for example, 2.76 seconds of total time on server 9, versus 2.21 seconds of total time on server 9, relatively. This generalization holds true for CPU

time, the average/CPU swap times, and both servers, as well. This performance is likely due to the fact that for one thread, Synchronized will not be halting execution on any other threads and will be quick, whereas AcmeSafe has an inefficient current() method that adds overhead despite quick atomic operations. However, this is the only scenario in which Synchronized performs better than AcmeSafe (for server 9) – for all other thread values, from 2 to 50, AcmeSafe realizes quicker speeds than Synchronized (except for outliers, which we discuss later). Because total time and CPU time are not strictly increasing nor nondecreasing functions based on the number of threads (possibly because of server load differences during consecutive program runs), we cannot generalize by how much the times increased based on an increase in threads. However, we calculate that (based on total times) AcmeSafe takes anywhere from 29.9% to 68.79% of the time Synchronized takes to run (utilizing the thread values with the lowest and highest disparities in performance, see thread numbers 6 and 10, relatively). However, when compared to Unsynchronized, AcmeSafe takes anywhere from 185% to 705% of the time Synchronized takes (utilizing thread values 1 and 6). This disparity in performance is much greater than that between AcmeSafe and Synchronized; despite AcmeSafe's optimizations, it still performs much more poorly than the incorrect version without DRF operations. We can also check for these generalizations with variance in the number of entries. The same generalizations hold true. Unsynchronized has quicker speeds than either AcmeSafe or Synchronized, while AcmeSafe has quicker speeds than Synchronized. There is only one outlier for which Synchronized performs better than AcmeSafe, at 10 entries, for which AcmeSafe has a total time of 17.2 seconds versus Synchronized's 16.4 seconds. Because there is no reason for 10 entries specifically causing this discrepancy, we might chalk it up to a light server load and therefore quicker time during the Synchronized run, seeing as it is lower than both the times for 5 and 20 entries, in opposition to the slightly increasing time for increasing entries. Conducting a similar analysis to that for the number of threads, we see, based on total time, that AcmeSafe takes anywhere from 6.6% to 89% of the time Synchronized takes (utilizing entry values 15 and 10,000). This is an extreme difference in efficiency, and the generalization to note here is that whereas the time Synchronized took to ran increased as the entries increased, the time AcmeSafe took to run actually began to decrease beyond 150 entries. AcmeSafe, therefore, is especially efficient for exceptionally large numbers of entries. In comparison with Unsynchronized, AcmeSafe took anywhere from 105% to 472% of the time (see thread values 10,000 and 10). For high entry values such as 10,000, there is an exceptionally low difference in efficiency between AcmeSafe and Unsynchronized. However, this is in stark contrast to performance on server 10, in which AcmeSafe ended up performing worse than Synchronized on a majority of the test cases involving variation of both entries and threads. See the graphs for details. Here, Synchronized gives very stable results, but there is no underlying trend in performance difference between Synchronized and AcmeSafe; for some thread/entry values Synchronized performs quicker, and for others AcmeSafe does. Therefore, performance is highly dependent not only on the program details, but the hardware of the testing platforms as well. For servers 7-9, these patterns suggest that the efficiency of the three versions of this program do not deviate from: Unsynchronized > AcmeSafe > Synchronized. However, seeing as AcmeSafe produces correct results where Unsynchronized does not, this tradeoff suggests that AcmeSafe is the most effective and efficient implementation. This does not hold true for other hardware models, such as server 10.

Replication Notes:

The following is the java version used:

java version "13.0.2" 2020-01-14

Java(TM) SE Runtime Environment (build 13.0.2+8)

Java HotSpot(TM) 64-Bit Server VM (build 13.0.2+8, mixed mode, sharing)

The following are the server model names for servers 9 and 10, respectively:

Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz

Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz

Index of Measurements

Entries	Server 9 Synchronized Total Time in s	Server 9 Unsynchronized Total Time in s	Server 9 AcmeSafe Total Time in s	Server 9 Synchronized CPU Time in s	Server 9 Unsynchronized CPU Time in s	Server 9 AcmeSafe CPU Time in s	Server 10 Synchronized Total Time in s	Server 10 Unsynchronized Total Time in s	Server 10 AcmeSafe Total Time in s	Server 10 Synchronized CPU Time in s	Server 10 Unsynchronized CPU Time in s	Server 10 AcmeSafe CPU Time in s
1	0.00248048	0.00202048	0.00179174	0	0	0	0.00675706	0.00940076	0.00707577	0	0	0
5	20.8299	3.65387	8.98524	69.3799	28.9058	66.5898	4.72725	1.86878	10.0472	5.43358	5.39628	31.2151
10	16.4107	3.64314	17.2047	44.3334	28.6018	134.331	4.62642	3.64832	13.4701	5.08348	10.5822	40.8392
15	19.2424	5.24746	17.2414	52.2554	40.5987	136.157	4.20572	3.71	5.63692	4.52995	11.2001	17.3372
30	23.7366	5.87855	8.25366	77.0134	46.3156	65.3507	4.69585	3.89421	4.62342	5.14833	11.7923	13.9998
50	25.8699	5.53471	6.07653	83.7075	43.8336	48.0271	4.39781	3.96093	4.40473	4.7372	11.9856	13.1591
100	23.7248	4.7267	10.7181	74.9767	37.2254	85.2022	4.53991	3.53934	5.6699	4.89141	10.4012	16.8158
150	13.4862	3.82242	6.97054	38.8093	30.132	55.2178	4.22976	3.32239	4.98515	4.44963	10.1211	14.9237
200	25.7877	3.44529	6.00971	78.5564	27.4504	47.8812	4.40299	3.15611	5.774	4.86053	9.57717	17.2026
500	8.02194	2.7198	4.69666	9.79065	21.6436	37.1536	3.95112	2.77013	4.27405	4.14396	8.1166	13.1742
1000	27.8306	1.35152	3.03995	84.0609	8.38615	23.9767	3.82649	2.47312	3.69758	3.98595	7.49596	11.3419
2000	21.7314	1.92796	2.40761	64.034	14.8128	19.0286	4.13759	2.35603	3.34318	4.65189	7.04553	10.017
5000	21.1536	1.90697	2.06074	63.996	14.5483	16.2737	3.84075	2.12767	2.90872	4.03459	6.434	8.7964
10000	28.5385	1.7914	1.87999	84.4302	12.312	14.8561	4.0478	2.00701	2.88507	4.41155	5.95202	8.27753

Threads	Server 9 Synchronized Total Time in s	Server 9 Unsynchronized Total Time in s	Server 9 AcmeSafe Total Time in s	Server 9 Synchronized CPU Time in s	Server 9 Unsynchronized CPU Time in s	Server 9 AcmeSafe CPU Time in s	Server 10 Synchronized Total Time in s	Server 10 Unsynchronized Total Time in s	Server 10 AcmeSafe Total Time in s	Server 10 Synchronized CPU Time in s	Server 10 Unsynchronized CPU Time in s	Server 10 AcmeSafe CPU Time in s
1	2.21312	1.49331	2.76133	2.21175	1.49208	2.76001	1.66905	1.21559	2.53824	1.66073	1.21391	2.53678
2	13.8799	2.28391	5.89125	27.442	4.34168	10.9227	14.0491	2.23107	11.2773	27.1121	4.3708	22.411
4	14.2843	4.10404	9.1272	44.6772	16.2396	33.1956	5.21316	2.17114	11.5533	6.34562	5.63733	30.085
6	23.4489	2.28949	16.138	79.531	13.1231	96.5015	4.80183	2.11105	11.522	5.39012	5.64023	33.308
8	26.455	2.39952	9.4652	91.4443	18.4847	70.608	4.39565	1.83384	12.0457	4.95849	5.33826	36.8214
10	30.2768	3.6132	9.05347	105.979	35.0842	87.093	4.45894	1.95164	12.9496	5.04866	6.13199	41.5598
15	30.9605	2.59917	9.12192	104.357	37.4046	131.724	5.19512	1.90181	10.3636	6.2463	6.11176	35.3898
20	20.9305	3.51491	10.1728	68.0774	66.8747	198.323	5.16133	2.50706	10.6602	6.21869	8.53626	37.2139
25	22.504	2.7969	8.77147	75.4372	66.9253	200.474	5.35362	2.38474	13.1788	6.33754	8.05408	47.076
30	17.7439	3.47543	8.2617	58.4452	90.5254	193.238	5.40477	1.87597	13.1107	7.22703	5.94547	48.0909
35	31.5348	2.92548	11.9739	108.709	76.8689	290.333	5.75598	1.87046	12.2869	7.82347	6.01354	44.4125
40	18.3789	3.39604	8.54715	55.7384	94.9024	214.069	5.44611	2.48029	13.8166	7.28089	8.49487	50.6242
45	25.1611	2.76169	8.2203	84.5668	76.7179	209.202	5.73636	1.81956	13.1613	7.50799	6.02185	49.0153
50	21.6995	2.75624	11.4464	71.1081	76.8993	293.835	7.02791	2.4896	13.2416	11.0834	8.57672	49.3267

References

Jain, Deep. “An Introduction to Atomic Variables in Java.” *Baeldung*, 12 May 2019, www.baeldung.com/java-atomic-variables.

Lea, Doug. “Using JDK 9 Memory Order Modes.” *Using JDK 9 Memory Order Modes*, 16 Nov. 2018, gee.cs.oswego.edu/dl/html/j9mm.html.

“What Is an Atomic Operation? - Definition from Techopedia.” *Techopedia.com*, 29 Aug. 2011, www.techopedia.com/definition/3466/atomic-operation