Shared Memory and Multithreaded Performance in Java

[redacted], *UCLA Computer Science Dept.*

Abstract

Parallelization of programs is a key concern for many software developers, especially in today's many-core world. Along with the gains of parallel processing, there are potential pitfalls of race conditions, which can cause problems in even the most carefully designed programs. Here we explore some methods of synchronization, including the *synchronized* keyword, atomic array access and user implemented locks. We found that better success can be achieved using small locks on individual array elements, as compared to synchronized methods. In addition, pushing our model towards the edge of reliability, we determined that sacrificing some consistency can give performance gains which may be useful in certain applications.

1. Various Synchronization Methods

The default Java method of synchronization is achieved by using the *synchronized* keyword. This method is akin to locking the object containing the synchronized method, and unlocking at the completion of the method. *Synchronized* permits a fast and easy method to create critical sections, but the performance of this solution is less than desirable for many applications, meaning other avenues need to be explored.

1.1. An Overview of concurrent, atomic, locks and VarHandle

Several other methods of synchronizing exist within the Java Memory Model, and four will be briefly explored below. The package java.util.concurrent is a broad class, entailing an interface, locking mechanisms and atomic access to memory. As such, many classes inheriting from this package are valuable for fixing race conditions. One such subclass is java.util.concurrent.locks. This interface provides several interfaces and classes used to lock particular portions of code. Another package, java.util.concurrent.atomic gives several types of atomic data types, including AtomicInteger, AtomicIntegerArray, and others. These classes have methods permitting atomic access, comparison and updates. The class AtomicBoolean was utilized in this experiment to implement a lock on a critical section. Finally, there exists the more abstract package java.lang.invoke.VarHandle. This package defines several types of memory access that are permitted in the Java Memory Model, including volatile, opaque, and plain. Using this package, fences can be created, forcing in-order execution at certain points by the Java Virtual Machine. However, even forcing in order execution and mandating volatile variables is not sufficient synchronization to remove all race conditions from the sample program. However, these could be used to increase data integrity, which may fall under a "good enough" case.

1.2. Selected Implementation

To create a function that improved on the performance of *synchronized*, but maintained reliability, a locking system was implemented using the AtomicBoolean class. Each value in the member array had a corresponding AtomicBoolean, indicating whether the array element was locked or unlocked. In order to perform operations, a thread had to acquire locks on both elements before executing the critical section. If one or more locks couldn't be acquired, the thread would sleep for 10 nanoseconds before attempting to get the locks again. By using this method, 100% reliability can be achieved, as only one thread is ever operating on a pair of elements, as guaranteed by the AtomicBooleans. In addition, this solution is more performant than the *synchronized* keyword. Since the entire array doesn't need to be locked, there can be multiple threads operating on seperate elements concurrently. As long as two threads don't attempt to access the same element, all threads can operate at the same time. This performance improvement becomes greater with a larger array, as this reduces the chance that any two threads are attempting to access the same array element.

1.3. Testing Environment and Process

All analysis was run on the UCLA lnxsrv09 server. This piece of hardware has 32 Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz processors, each containing 8 cores. In addition, the system has a total memory of 65,7558,84 kB, or approximately 64 gigabytes. Of that memory, at least 50 gigabytes were available at the time of testing. Testing was done in two phases, with both openjdk version "11.0.2" and openjdk version "9". One limitation, as described in more detail below, was that Java 9 severely limited the number of threads that could be spawned. For Java 11, each program (*synchronized*, Null, BetterSafe, Unsynchronized and GetNSet) was run with 100,000 iterations, and the results for each were averaged over up to 20 trials, fewer if some trials failed. This test was run for thread counts 8, 12, 16, 24, and 32. Each of these was then run with an array size of 4, 8, 16, 32, 64, 1024, and 2048. The maxval for every trial was 64, and each element of the array was a randomly generated value from 0 to 63. Overall, this testing suite ran the UnsafeMemory test program 700 times for each program, with results sent to a CSV file, which was used to analyze and generate results.

2. Analysis of Results

The BetterSafe class was found to perform at or above the level of *synchronized*, although worse than the two data race unsafe implementations, Unsynchronized and GetNSet.

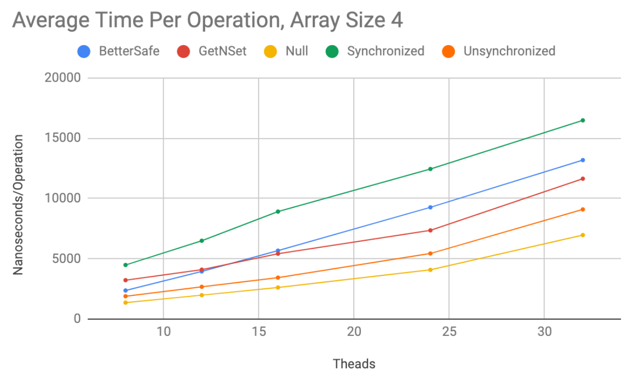

Figure 1.1 Comparison of the cost-per-operation of various methods of synchronization

As can be seen from figure 1.1, the cost-per-operation increases for all methods of synchronization as a function of the number of threads. However, these data also show that for all numbers of threads, *synchronized* performs worse than any other method, and has a steeper slope as threads increase, indicating that scaling issues will occur as more and more threads are added to the system. Additionally, it can be seen that both GetNSet and Unsynchronized perform significantly better than either BetterSafe or *synchronized* for increasing thread count. Because these two methods eschew synchronization, they sacrifice data integrity for speed.

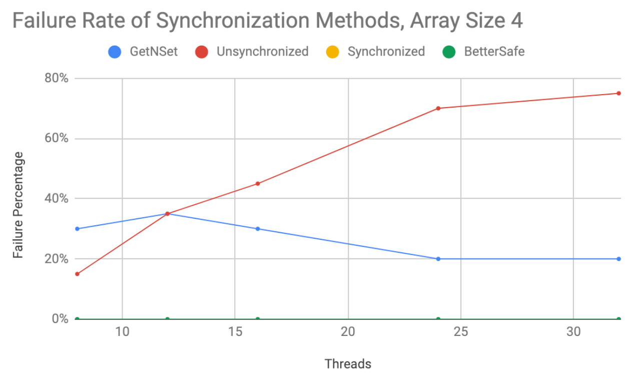

Figure 1.2 Failures with lack of synchronization

Without synchronization, an increasing percentage of test cases fail as thread count increases. We define a “failed” test case as one that either took to long to execute, generally because the array was put into an unrecoverable bad state (all values below 0 or above maxval), or if the sum was inconsistent at the end of the trial. As can be seen from figure 1.2 above, the rate of failure for the GetNSet and Unsynchronized methods are much higher than the zero rate of BetterSafe and *synchronized*. The array size of 4 was chosen to demonstrate because a smaller has more collisions between thread access, meaning that there is a higher chance of errors without synchronization. BetterSafe also exhibits better performance in relation to array size than *synchronized*.

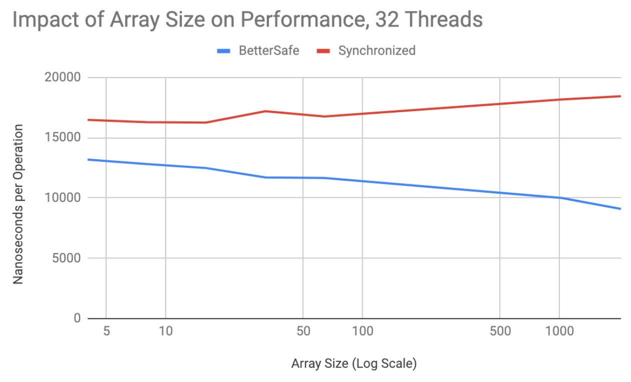

Figure 1.3 Comparison of performance vs array size

The figure above demonstrates the results for a changing the array size of BetterSafe and *synchronized* while keeping the number of threads at a constant 32. The comparison shows that BetterSafe scales much better with increasing array size, and that the cost per operation decreases as array size grows. By comparison, *synchronized* maintains a similar cost as array size increases, and the cost per operation even increases slightly. This indicates that the BetterSafe implementation provides better scalability to larger data sets, where the locking of individual elements permits more concurrent access to the array, as compared to the brute-force locking of *synchronized*.

3. Conclusions and Limitations

3.1. A defense of BetterSafe’s implementation

As seen from the data above, BetterSafe provides significantly faster operations as compared to using the *synchronized* keyword. By creating a lock around individual elements, the implementation permits multiple read and write accesses by different threads, which in turn provides more speed. In addition, unlike Unsynchronized and GetNSet, BetterSafe is Data Race Free (DRF). Because elements are locked before they can be checked or modified, there is a critical section that guarantees that no two threads can possible be accessing the same variables at the same time. This is done through the use of atomic operations on an array of booleans, which provide an ad-hoc locking mechanism. Ordered lock acquisition is also implemented, preventing a deadly embrace between two threads accessing the same memory resource.

3.2. Modifications to execute testing

In order to run automated tests, the provided program UnsafeMemory.java had to be modified slightly to accept Unsynchronized, GetNSet and BetterSafe as command line arguments, and the timed output was modified to output a floating point. One issue that occurred during testing was that Unsynchronized and GetNSet occasionally hang, if a series of bad operations has corrupted the state of the array. In order to get around this issue, the testing script ran these in the background, and then if they were still executing after a timeout period, they were killed, and the number of successful tests was not increases.

3.3. Limitations

A major limitation of this research is that it failed to completely compare the performance of Java 11 and Java 9. As mentioned above, Java 9 was unable to spawn more than about 12 threads, even for the freshly unjared program. Because of this limitation, all tests with Java 9 were only run with 2, 4, 8, and 12 threads, meaning that the results between Java 9 and 11 could not be directly compared. Several fixes were attempted, including increasing the maximum amount of memory available to Java, as well as increasing the thread stack size, to no avail. While this limitation prevented comparison between versions of Java, within a single version, the effects of various memory accesses and synchronization methods could be compared.

3.4. Recommendation to Ginormous Data Inc. (GDI)

Based on the results of extensive testing of the various implementations above, the author would recommend the use of GetNSet for Ginormous Data Inc.'s internal applications. While GetNSet was not 100% reliable, and did suffer from some data races, it was much more accurate than Unsynchronized, and had a higher performance than BetterSafe. Since GDI's applications are more reliant on speed than total accuracy, GetNSet provides a good middle ground, balancing performance against correctness. This recommendation comes with a cavat, however. Because GetNSet can place the array into a bad state, causing the program to hang, the environment wrapping this function must be robust enough to deal with and potentially kill a process that hangs.