Developing an Ideal Synchronization Strategy for Big Data Analysis: A Study of Java Shared Memory Performance

Naim Ayat, *UCLA Department of Computer Science*

Abstract

Ginormous Data Inc. (GDI) specializes in statistical trend analysis for big data. The company employs multithreaded Java programs on shared-memory representations of the state of pattern-finding simulations. A known bottleneck in the current software architecture is the Java `Synchronized` method, which GDI seeks to eliminate from its codebase entirely. Through the examination of sequential-consistency-violating performance data and the development of a corresponding reliability testing program, it is possible to determine the most efficient (*i.e.* shortest runtime below the maximum error threshold) simulation state application programming interface (API) for GDI to achieve its mission.

1. Introduction

The consultation began with the development of a new Java class, `Unsynchronized`, which shares the same implementation as the standard `Synchronized` class included in `jmm.jar`, a JAR file holding the source code of a GDI simulation. Each method contained in this archive is based on the Java memory model (JMM) to prevent data races when accessing shared memory during multithreaded programs.

A third simulation state API, `GetNSet`, was written as a combination of features from `Synchronized` and `Unsynchronized`. The class does not use synchronized code; rather, it uses volatile accesses to array elements. The implementation of `GetNSet` uses the `get` and `set` methods from the Java atomic utility class `AtomicIntegerArray`.

Finally, a class `BetterSafe` was written for the purpose of outperforming `Synchronized` while still upholding 100% reliability. Its efficiency is a result of a more sophisticated locking mechanism – while `Synchronized` simply locks entire objects, `BetterSafe` locks critical sections only.

1.1. Development & Testing Environment

This study in was conducted on UCLA SEASNet Linux Server 09, an 8-core Intel Xeon CPU E5-2640 v2 at 2.00GHz with 64GB RAM. The Java version installed at the time of development is as follows:

- Java 9.0.4
- Java SE Runtime Environment (build 9.0.4+11)
- Java HotSpot 64-Bit Server VM (build 9.0.4+11)

2. Results

Each simulation API included in `jmm.jar` and developed during the consultation was tested with 10,000,00 transitions per execution combined with either 8, 16, or 32 threads for a 650-byte array on the ASCII range (0 to 127). The API types are displayed with their average time per transition for each thread count below (Fig. 1).

	8 Threads	16 Threads	32 Threads
`Synchronized`	1.80991 ± 0.00003ns	8.05969 ± 0.00005ns	25.9948 ± 0.00006ns
`Unsynchronized`	1.41767 ± 0.00004ns	5.92344 ± 0.00005ns	19.4282 ± 0.00005ns
`GetNSet`	1.48819 ± 0.00004ns	6.44599 ± 0.00005ns	23.4412 ± 0.00004ns
`BetterSafe`	1.62914 ± 0.00004ns	7.51235 ± 0.00004ns	23.7719 ± 0.00004ns
`Null` (control)	1.35678 ± 0.00003ns	5.11349 ± 0.00004ns	23.2503 ± 0.00004ns

Figure 1: Effect of simulation state API and thread count type on average time per transition in a 650-byte array over the range of decimal ASCII character representations. Each value and its uncertainty is the best representation of 10 trials according to equations (1) and (2).

Due to the volatility of the testing environment, it was nonviable to obtain consistent data for transition time in the various test cases. Therefore, each numerical result from Fig. (1) is the best value λ_{best} generated over ten trials, as calculated by the formula:

$$\lambda_{best} = \frac{1}{N}\sum_{i=1}^{N}\lambda_i \qquad (1)$$

Where $N = 10$. Hence, the corresponding uncertainty is given by:

$$\delta\lambda_{best} = \sum_{i=1}^{N}\left(\frac{\delta\lambda_i}{N}\right)^2. \qquad (2)$$

3. Analysis

As shown in Fig. (1), the `Synchronized` method was found to consume the most CPU time for all observed thread counts. It is, however, important to note that this simulation state API is guaranteed to be data-race free (DRF) by definition, as the method locks code such that no thread can access the object in contention (apart from the one currently operating on it) until the context

switches back to the preempted thread. These transitions require significant overhead; and, out of all five methods, occur most frequently in `Synchronized`. Thus, GDI's current simulation API method is 100% reliable at the cost of speed.

Conversely, `Unsynchronized` is the fastest model (of course, not including the control method `Null`) while being the most unreliable. Its implementation is identical to that of `Synchronized`, however, it does not include locks and is therefore immune to the associated time expenses. `Unsynchronized` is clearly not DRF, as it has no procedure for protecting critical sections against race conditions. Thus, the method becomes increasingly susceptible to incorrect results as the number of threads increases. An example of a reliability test likely to fail is: `Java UnsafeMemory Unsynchronized 32 1000 127 3 111`. Out of ten trials, this test failed six times with `sum mismatch` errors.

`GetNSet` outperforms `Synchronized` in all cases due to its use of an atomic integer array, which allows it to access data concurrently without significant risk. Although it is rare that an error will occur, it is not strictly data-race free.

Proof (1): Recall that the maximum value of decimal-represented ASCII characters is 127. Assume a thread t_1 is operating on a member of the array which holds a value of 126. Immediately as t_1 prepares to increment the value, there is no longer protection against preemption. Hence, if another thread t_2 attempts to swap concurrently, it is possible that t_2 increments the value to 127 and passes it back to t_1. In turn, t_1 will increment the value to 128, resulting in an error. Because `GetNSet` is vulnerable to this race condition, it is not DRF.

Finally, `BetterSafe` employs a `ReentrantLock` from `java.util.concurrent.locks` to protect critical sections. As per Fig. (1), it is quicker than `Synchronized` since it only locks portions of code which are at risk of race conditions. However, it is not as fast as `GetNSet`, as the latter employs atomic operations in place of true locks. Note that `BetterSafe` is, in fact, DRF by definition of `ReentrantLock`; it is impossible for a race condition to occur.

3.1 `BetterSafe` Design Considerations

The package `java.util.concurrent` contains two interfaces of interest:

1. `java.util.concurrent.locks`
2. `java.util.concurrent.atomic`

The first interface allows for standard mutex operations; such as locks, unlocks, waits, and notifies. This is the basis for the `Synchronized` method, which has proven to be unsophisticated. However, a clever implementation of this interface could lock critical sections only, thus reducing the overhead associated with `Synchronized` but still maintaining 100% reliability.

Moreover, the `locks` interface is the most syntactically simple, providing an added benefit of familiarity for developers.

The `atomic` interface, as demonstrated by `GetNSet`, is useful in that its execution speed is quicker than that of its `lock`-oriented counterparts. However, it is quite difficult to design a completely thread-safe simulation state model with atomic operations. Most implementations emerge with bugs in edge cases – a result of the lack of true mutex operations.

Another package, `java.lang.invoke.VarHandle`, is also a possible framework. A `VarHandle` is a dynamically strongly typed reference to a variable; thus, this class allows for plain and direct read/write access, as well as compare-and-swap. However, an issue arises in that GDI's simulation state API must read in two values. The `VarHandle` interface makes concurrent reads and writes complex.

The final decision, therefore, was to implement `BetterSafe` with the `locks` interface to protect critical sections. In doing this, the thread safety of `BetterSafe` is guaranteed. Although there are slight performance drawbacks when compared to `Atomic` implementations, there is no risk of computational error.

4. Conclusion

To summarize, `BetterSafe` and `Synchronized` are the only simulation state APIs which are 100% reliable. `GetNSet` comes close – although it fails in `get/set` edge cases such as that specified in Proof (1). `Unsynchronized`, of course, is the least reliable because it completely lacks protection against race conditions.

In terms of CPU speed, `Unsynchronized` is quickest due to the absence of thread safety measures. It is followed by `GetNSet`, which utilizes the atomic utility class instead of a lock mechanism. `BetterSafe` is next; it locks critical sections of code whereas `Synchronized` locks entire functions. The latter, therefore, is slowest.

It is strongly recommended that GDI employ `GetNSet` as their simulation state API. The class outperforms all others tested with race condition protections. Although the method is not strictly DRF, it fails only in rare edge cases. The mission of Ginormous Data Inc. is to provide the quickest results possible while keeping error below a reasonable threshold. Hence, for the purpose of fast, virtually error-free big data analysis, `GetNSet` is the ideal solution.