Homework 3: Java Shared Memory Performance Races

Abstract

We assess the performance of Java synchronization, which is based on the Java memory model, or JMM, using four different implementations of an interface that manages an array of `long`s. The paper assesses the performance and reliability tradeoffs when utilizing different implementation of the same prototype interface. It was found that while an implementation method might improve performance when measured with total real runtime, it actually can perform worse if taken into account its CPU utilization as a metric.

1. Introduction

The Java memory model, or JMM, defines how an application can safely avoid data races when accessing shared memory. Unlike traditional intuitive semantics where the system utilizes a global clock to determine actions and priorities of threads through a scheduling system, JMM allows Java programs to be data-race free, or DRF, by classifying that a conflict occurs when two different threads are accessing the same location without synchronization in between.

According to Doug Lea's "Using JDK 9 Memory Order Nodes", concurrent programming using multi-processors systems can encounter three forms of parallelism: task parallelism, memory parallelism, and instruction parallelism. Multi-threaded programs on such systems may introduce race conditions as two threads executing basic actions may be unordered with no preceding mechanism to prioritize one over the other; memory (including caches) is managed by independent parallel agents and a variable is simply an agreement among threads about values associated with an address thus memory may not perform operations atomically; and CPUs no longer single-step instructions as they rather process them in an overlapped fashion to optimize multiple instructions occurring at the same time.

This paper studies the performance and reliability tradeoffs between different Java programs through an attempt to break sequential consistency in their implementation. Using a simple prototype that manages a data structure that represents an array of `long`s, the multi-threaded program performs a swap method, or a state transition, where a thread simultaneously decrements the value at one index in the array while incrementing the value at another index. At termination, the sum of all array entries should remain zero. A nonzero case indicates that one or more transitions were not done correctly during the process.

2. Testing Platforms

We tested the programs on two different SEASnet GNU/Linux servers `lnxsrv06` and `lnxsrv11`. The two platforms run different CPU types, models, processors, and operating systems which will be discussed more in depth in below sub-sections. The details of the testing platforms can be found in `/proc/cpuinfo`, `/etc/os-release`, and `/proc/meminfo`. Both servers run the same Java version as follows:

```
openjdk version "15.0.2" 2021-01-19

OpenJDK Runtime Environment (build 15.0.2+7-27)

OpenJDK 64-Bit Server VM (build 15.0.2+7-27, mixed mode, sharing)
```

2.1. GNU/Linux `lnxsrv06` server

The lnxsrv06 server runs a CPU with the model name "Intel(R) Xeon(R) CPU E5620 @ 2.40GHz" with 15 processors, 4 cores, and cache size of 12288 KB. The operating system is Red Hat Enterprise Linux Server version 7.8 (Maipo). The system has a total memory of 65794548 kB, with 52997080 kB free memory. The processor architecture and the version of the kernel running on the system is as follows:

```
Linux lnxsrv06.seas.ucla.edu 3.10.0-1062.9.1.el7.x86_64 #1 SMP Mon Dec 2 08:31:54 EST 2019 x86_64 GNU/Linux
```

2.2. GNU/Linux `lnxsrv11` server

The `lnxsrv11` server runs a CPU with the model name "Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz" with 3 processors, 4 cores, and cache size of 16896 KB. The operating system is Red Hat Enterprise Linux version 8.2 (Ootpa). The system has a total memory of 65649184 kB, with 1703208 kB free memory. The processor architecture and the version of the kernel running on the system is as follows:

```
Linux lnxsrv11.seas.ucla.edu 4.18.0-193.19.1.el8_2.x86_64 #1 SMP Wed Aug 26 15:29:02 EDT 2020 x86_64 GNU/Linux
```

2.3. Comparison

The machine running on `lnxsrv06` is slightly faster than that on `lnxsrv11` server with the processor base frequency of 2.40 GHz vs 2.10 GHz. Moreover, it has 15 processors compared to 3 processors on the latter machine, which can impact our measurements later as we can run more threads at once with more processors available. Despite that `lnxsrv11` has a larger cache size, it has much less free memory available compared to `lnxsrv06`. More free

memory available will allow faster access speed and may increase the total speed of our program.

3. Implementation

I implemented my AcmeSafeState using the built-in library java.util.concurrent.atomic.AtomicLongArray, which represents a long array in which elements may be updated atomically. The implementation was straightforward and simple. I replaced the original `value` variable from one of type `long[]` to `AtomicLongArray` type. Using built-in methods of the `AtomicLongArray` library, I utilized the `getAndDecrement(int i)` and `getAndIncrement(int i)` methods to atomically decrement and increment the value of the element at index i, respectively. Lastly, for the `AcmeSafeState.current()` method, since the prototype interface requires that we returns an array of type `long[]`, I had to convert the `AtomicLongArray` value to a `long[]` by iterating through my current `value` array and copying its values into a temporary variable `longArr` using the `AtomicLongArray.get()` method, which returns the current value of the element at index `i`.

3.1. `java.util.concurrent.atomic.AtomicLongArray`

The AtomicLongArray class utilizes the properties of atomic accesses in accordance with the VarHandle specification. A VarHandle is a dynamically strongly typed reference to a variable, and access to such variables is supported under specific modes, including plain read/write access, volatile read/write access, and compare-and-set. Most importantly, these access modes control atomicity and consistency properties.

4. Performance Analysis

The following charts displayed my results from executing the different Java classes implementations under varying conditions with 1, 8, 32, 40 threads and array size of 5, 50, and 100. The dataset compares the difference in total CPU time and total real time separately and includes the results from running on both machines as specified above.

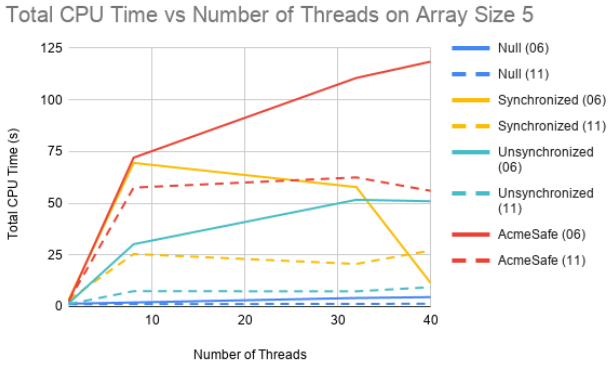

Figure 1. Total CPU time vs. Number of Threads running with Array Size of 5.

Overall, the different classes all performed much faster on the lnxsrv11 machine (the dashed lines). This can be accounted for how much less processors available on the lnxsrv11 machine. Since CPU time quantifies how busy the system is, with less processors available, the number of threads increased does not greatly affect the total CPU time as the processors is shared equally among the 3 processors, allowing for lower CPU usage overall.

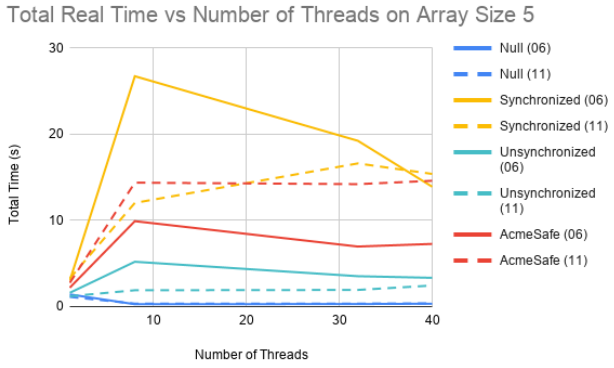

Figure 2. Total Real Time vs Number of Threads running with Array Size 5.

Without considering our CPU usage due to our CPU model specifications, the performance of the programs generally increases starkly when going from 1 thread to 8 threads. However, the time starts to level down across the board with higher number of threads.

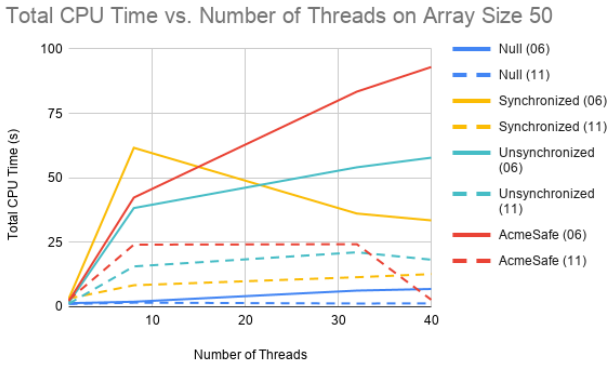

Figure 3. Total CPU Time vs Number of Threads running with Array Size 50.

As we can see distinctly from the Figure 1 and Figure 3, increasing the array size, or number of operations from 5 to 50, allows us to see the performance separation between the 2 machines, where the lnxsrv11 machine done significantly better in its CPU usage compared to the lnxsrv06 machine in all four classes. It is also important to note that the CPU

usage for the lnxsrv11 machine remains fairly constant even with more threads added to our programs while it increases on the lnxsrv06 machine.

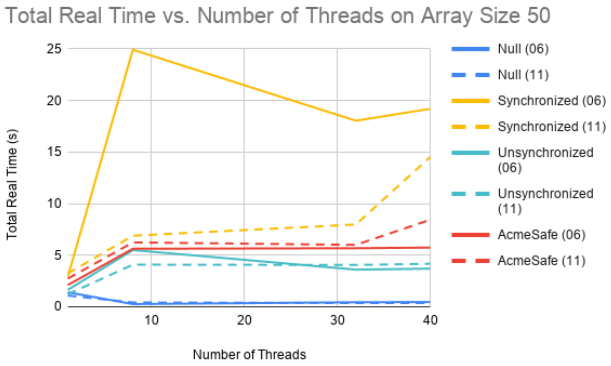

Figure 4. Total Real Time vs Number of Threads running with Array Size 50.

In this graph, the synchronized class performs much worse with almost double total runtime compared to all other classes when the number of threads increase. All the other classes perform at virtually similar performance.

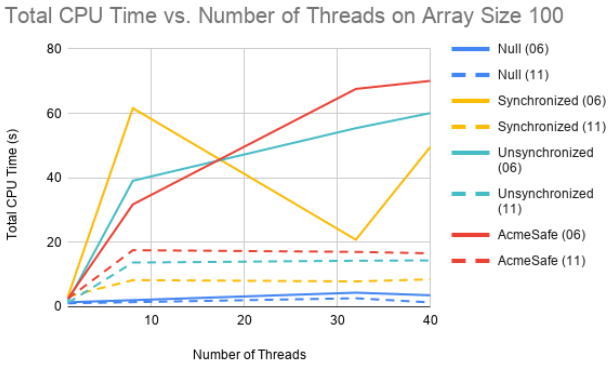

Figure 5. Total CPU Time vs Number of Threads running with Array Size 100.

When run the program with 100 operations, again, the performance on the lnxsrv11 machine stays fairly constant while the CPU time of the lnxsrv06 machine increases with added number of threads, with the exception of the Synchronized class where its performance dips when it was ran with 32 threads.

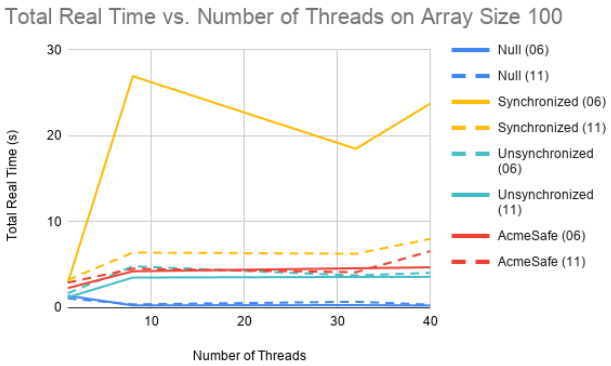

Figure 6. Total Real Time vs Number of Threads running with Array Size 50.

However, when looking at the Real time, all classes performed similarly with an exception of the Synchronized class running on the lnxsrv06 machine where its total runtime almost triples all other programs.

Another important factor to note that all programs were able to remain 100% reliable except the Unsynchronized class, which returns errors in the sum at the termination of the program when running with more than one thread. So, the Unsynchronized class might not improved performance as much compared to our AcmeSafeState, and it is also not DRF when running as a multi-threaded program. Moreover, all of these programs performs best when it is only running one thread, this might be due to the overhead introduced in multi-threading programs, where one thread must wait for another in synchronization programs to access a shared variable. This can also potentially increase CPU usage due to the waiting time that can also cause starvation of threads on multi-processors machines where there is no priorities system implemented globally among the processes.

My AcmeSafeState implementation actually has lower performance than the Synchronized class when using Total CPU time as our measurement factor, which means this AcmeSafecState class utilizes our CPU less efficiently than the Synchronized class. However, when looking at Total Real Time, the AcmeSafeState class did perform better than the Synchronized class while still able to retain 100% reliability in multi-threaded applications. This can be credited to the AtomicLongArray class, where each operation is performed atomically and cannot be interrupted, which allows our program to be Data Race Free.

5. Problems Encountered

One problem I encountered during the implementation of the program is implementing the current() method in the AcmeSafeState class. I was afraid that copying each value iteratively to a temporary array might not be the most efficient method to return our current array. However, after doing more research, it was the only method to convert an AtomicLongArray type to a type of long[].

Another problem I also encountered was accidentally running the program under a depreciated version of Java (before version 15.0), which caused errors when I executed the program. However, the problem was easy to resolve by appending the /usr/local/cs/bin: to my PATH variable to make sure it is running in the appropriate environment as the compiled Java program.

Lasly, to run the different test results and collect data more efficiently, I wrote a bash shell script that execute all of the

commands iteratively so that the program can run all the test cases more efficiently than being inputted manually.

6. Conclusion

Through the experiment, we can see that the implementation of a multithreaded, concurrent program will affect the performance and reliability of our program. The CPU model and system architecture also plays a role in affecting these measurement heuristics as memory and number of processors can affect the overhead that will incur in a multithreading system. The AcmeSafeState while was able to improve performance and retain reliability compared to the Synchronized class by utilizing atomic operations, it actually uses much more CPU resouces when compared by total CPU time. Lastly, increasing number of threads might not necessarily impact performance as the programs all appear to reach a certain threshold with a certain number of threads, and the performance plateau, or level down, even with increasing number of threads.

References

AtomicLongArray documentation

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/concurrent/atomic/AtomicLongArray.html

VarHandle documentation

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/invoke/VarHandle.html

Doug Lea's "Using JDK 9 Memory Order Modes"

http://gee.cs.oswego.edu/dl/html/j9mm.html