CS 111 Java HW 3 Synchronization

University of California, Los Angeles

Abstract

Java synchronization, based on the JMM defines how a programmer and application can avoid data races when using shared memory. However, this can lead to significant performance hindrances. To avoid this, we investigate several methods for speeding up multi-threaded computation in Java in this lab. The two main approaches we tried were

1. Not using any synchronization and measuring how significant the effect on the correctness of the program is, and

2. Using atomic operations provided by Java's AtomicLongArray class. The intention here is to see if this method of synchronization is faster than using critical sections via the synchronized keyword and still fully correct.

1 Introduction

Several methods for synchronization (and simply choosing not to synchronize anything at all) exist. As an employee of Ginormous Data Inc, we explore these options to speed up multithreaded computation. The expectation is that normal synchronization via Java's synchronized keyword has a lot of overhead, and is too expensive for the simple, quick computations we want to do.

The first approach we will use is choosing to completely forgo synchronization, as the boss said that slightly incorrect results are fine to have to increase speed. We know this will cause incorrectness issues, as the data race presented by two threads incrementing a single memory location will almost surely manifest itself often and cause incorrect behavior.

The second approach will be to use AtomicLongArray, which should be expected to have less overhead than wrapping the entire swap function in synchronized. We expect this because the extra function call overhead and lower amounts of optimization should slow down the original synchronized approach.

2 AcmeSafeState Implementation

To implement AcmeSafeState, I used the suggested AtomicLongArray class provided in the default Java provided util.concurrent.atomic package. The goal here was to minimize the amount of time spent in critical sections, and therefore allow threads to have more uptime where they are actually performing the intended computation. To make sure I kept the DRF behavior, I used functions that atomically modified shared memory. However, the rest of the function (the unseen overhead of function calls and other things like stack maintenance) is not protected, as it does not use shared memory. Implementing the other non-swap functions was simple. To add the size function, I simply made use of the safe AtomicLongArray.length() function. To get the current state, I allocated a new long array to be used as the return, and filled it in with each value from the inner AtomicLongArray. To make sure the correct values were getting read, the memory-safe AtomicLongArray.get() function is used.

The exact implementation of the central and important swap() function utilizes the AtomicLongArray.getAndDecrement() and AtomicLongArray.getAndIncrement() functions (their Increment/DecrementAndGet counterparts could be used as well). What can be seen here is that the memory access and modification are protected by the AtomicLongArray implementation, guaranteeing that they are DRF. These functions are in their implementation completely atomic, meaning other threads modifying the same memory location are unable to cause a data race. This creates a much more efficient method of synchronization, where we as programmers can treat the entire AcmeSafeState class as if it were unsynchronized, reducing the overhead of synchronizing behavior.

3 Problems

Implementing the specified classes was fairly simple and problem-free; the majority of figuring out the minor changes

that needed to be made were very small. To actually implement them, all that had to really be done was reading the Java API and documentation and finding the relevant operations in it.

Additionally, I saw inconsistencies in my testing methodology when running my tests at first, as there were many other users on the servers putting heavy load on them. To alleviate this, I later ran the tests when there were very few other users online. The two servers I chose to use were lnxsrv09 and lnsrv10, and I only ran on those two to preserve consistency.

4 Measurements and Analysis

The number of swaps for all tests was 100 million.

All timing measurements are average swap time in nanoseconds. Other times will be mentioned and noted when they are relevant.

4.1 lnxsrv09

lnxsrv09 has 32 Intel Xeon E5-2640 v2 CPUs, each with 8 cores and 16 threads running at 2 GHz with one 20 MB cache per processor.

4.1.1 Varying Array Size

The number of threads was kept constant at 8

Class/Elements	10	100	1000
Null	20.30	22.03	23.21
Synchronized	791.48	851.90	1097.90
Unsynchronized	301.54	358.08	221.92
AcmeSafe	1286.09	818.79	214.12

We can see here that as the array size increases, the Synchronized class performs slightly worse and worse. Even though there's less true contention (i.e. two threads trying to modify the same locations), the synchronized keyword simply prevents multiple SynchronizedState objects from modifying ANY of the array simultaneously. This is why no improvement is seen - we still have the same amount of contention as with any sized array. Additionally, it continues to get worse probably more as a side effect of additional overhead and, more than likely, increased numbers of cache misses since the array is larger and less of it can be stored in cache at once.

The Unsynchronized class does better, and tends to not care much about the size of the array. Since there's no contention, we see higher variance in the times

The AcmeSafe class does fairly poorly at low element count, but quickly gets better and better as the array size increases. This makes sense - the use of the AtomicLongArray means each piece of memory can have its own lock, so two threads operating on different indicies of the value array can simply do as they please without bothering each other. As the number of elements increase, the chance of two (or more) threads competing to change the same values in value decreases thereby reducing contention.

4.1.2 Varying Thread Count

The number of elements was kept constant at 100

Class/Threads	1	8	16
Null	12.88	22.84	47.91
Synchronized	19.73	2274.04	4225.92
Unsynchronized	14.48	367.08	585.22
AcmeSafe	26.03	828.01	1127.47

As the number of threads increases, so does the synchronized runtime. This is very much expected - only one thread can work at a time, so the other (n-1) threads have to simply stand around and wait, wasting valuable time. There's not much more to say about this - the synchronized keyword is really bad for this.

As thread count increases, the Unsynchronized runtime also increases. This is likely due to things like extra overhead and inefficiencies within the threading mechanisms themselves.

The AcmeSafe runtime gets worse as the number of threads increases, but its by a much smaller amount than Synchronized. As threads increase, the chance of two threads colliding and attempting to modify the same memory location increases so contention does increase as thread count increases. However, the effect is less severe than Synchronized as there still exists a fair chance that any two given threads don't collide, which avoids any contention.

4.2 lnxsrv10

lnxsrv10 has 4 Intel Xeon 4116 CPUs, each with 4 cores running at 2.1 GHz with 16.5 MB of L3 cache per processor.

4.2.1 Varying Array Size

The number of threads was kept constant at 8

Class/Elements	10	100	1000
Null	37.67	46.00	28.67
Synchronized	406.96	370.01	343.05
Unsynchronized	282.99	279.40	191.18
AcmeSafe	869.07	467.49	259.35

No significant change in the runtime for Synchronized can be seen - again, the amount of contention the synchronized keyword causes is still the same for each. However, we do notice an overall slight speedup. This is likely due to the new CPU architecture and optimizations made by Intel between Sandy Bridge (the architecture of lnxsrv09) and Skylake (lnxsrv10's architecture).

The case for Unsynchronized is the same as on lnxsrv09, and that makes sense. The same still holds - no contention means changing the array size doesn't significantly affect runtime.

We see the same trend for AcmeSafeState as we did in lnxsrv09 for the same reasons. Increasing the size of the array means contention is less common, and therefore less time is wasted waiting for resources by threads.

4.2.2 Varying Thread Count

The number of elements was kept constant at 100

Class/Threads	1	8	16
Null	21.02	32.32	63.82
Synchronized	16.76	371.67	791.44
Unsynchronized	15.31	278.91	565.86
AcmeSafe	25.39	380.88	1017.37

Overall, we again see the same trends on lnxsrv09 as we do on lnxsrv10, but with slight differences.

In synchronized especially, we notice an immense speedup relative to lnxsrv09 (although the same trend still exists, and for the same reasons). This is likely because of the significant improvements in the single threaded performance of the newer system, along with other hardware differences. If those changes improve the amount of time spent in the synchronized function call, then we know it will be quite an improvement as that contention causes most of the slowness.

Unsynchronized stays mostly the same, although one thing to think about is the insane difference in core count between the servers. If I chose to run higher than 16 threads (the max of lnxsrv10), then lnxsrv10 would likely have suffered significantly.

AcmeSafe again follows the same trend as before, although at 8 vs 16 a large gap is notable. This might be due to the system starting to run out of available cores to run on and having to split cores between threads (via Intel Hyper-threading), leading to worse per-thread performance.

5 Concluding Thoughts

The results we found were very consistent with what was expected - the synchronized method was extremely inefficient as it blocks all threads but one from doing computations at any given time, AcmeSafe's use of AtomicLongArray allowed far more efficient computation, and Unsynchronized gave the best performance at the price of extremely incorrect results.

The level of incorrectness cause by Unsynchronized is, however, a direct function of what kind of work is being done. Since the swap function was very short, and almost only consisted of data-race dependent and data-race causing code, two threads executing simultaneously are fairly likely to collide. If, however, the thread spent only, for example, %10 of its time doing data-race dependent and data-race causing behavior, collision amount would significantly go down. Each solution has its pros and cons, and the each of them apply better to different situations.