Homework 3. Java shared memory performance races

Anonymous Author, *University of California, Los Angeles*

Abstract

Race conditions are a major issue in Java programming. A standard approach is the *synchronize* keyword, but this can be much too slow for many cases. To investigate possible alternatives to synchronized array access, we performed a series of swap operations with a variety of versions of the state class, varying both the length of the array and the number of concurrent threads.

1. Program Overview

Our program to evaluate approaches to concurrent access in Java was relatively simple. We define an interface *State*, and implement it in a variety of ways. We then utilize this state to perform a series of 100 million swaps, where a swap increments one element of the array contained within *State* and decrements another. The access to this array is the race condition we would like to address – uncontrolled, threads may concurrently access the array and thus some updates in either direction may be ignored. Additionally, a processor may cache an update and thus a later update by another processor may invalidate that update. We can see this with the below unsynchronized approach.

2. Machines

Tests were performed on two different Linux servers within SEASNet. Both ran RHEL 7 as an operating system, and ran Java version 13.0.2. Both were four core machines in the Intel Xeon line. The differences in the specs will be discussed in the following subsections. It seems as though server 06 is a larger version of an older machine, whereas server 10 is a smaller version of a newer machine.

2.1. Linux Server 06

Linux server 06 ran the basic Intel Xeon CPU. The machine ran at 2.4 GHz and had a free memory size of 345732 kB. Each processor of its fifteen processors had a cache size of 12288 kB and ran at 2395 MHz.

2.2. Linux Server 10

Linux server 10 ran the Intel Xeon Silver 4116 CPU. The machine ran at 2.10 GHz and had a free memory size of 390844 kB. Each processor of its four processors had a cache size of 16896 kB and ran at 2095.079 MHz.

2.3. Comparison

Since server 10 has a larger memory and a smaller processor count yet a lower power cost, it leads me to believe that this is a more powerful computer. The major factor determining the power is the balance between heat generation and computing power, and this one being lower tells me it probably has more powerful processors which are more prone to overheating.

3. Methodology

Concurrency can be handled in a variety of ways. These have both varying degrees of success and varying degrees of efficiency. Over the next few subsections we will lay out the various methodologies used to test array swapping. Firstly, the number of threads was varied with a constant array size of 5. Then, the threads were set to 8 and the array size was varied between 5, 25, and 100 longs.

3.1. Unsynchronized Memory Access

The first and simplest form of access is completely unsynchronized array access. Threads are initialized and each perform their updates independent of the other threads. Due to changes in values between the read and write time of a given thread/variable, the thread may effectively nullify the effects of another thread's update. As such, this method is extremely susceptible to race conditions, and returns incorrect results when run with more than one thread,

3.2. Synchronized Memory Access

The next form of access is synchronized access to the array between threads. This method only allows one thread to access the array at a time, performing its entire swap operation before allowing another thread to work. This is implemented using a lock over the entire class. As the lock protects the entirety of the class but it would theoretically be safe for multiple threads to access different elements of the same array at the same time, this lock is too course to give optimal performance in terms of speed.

3.3. AcmeSafe Access

The last form of access addresses the main problem of the synchronization method; this implementation allows concurrent access to the array, but not to a given element in the array. It is therefore race condition free. We utilize an atomic update function to increment and decrement the elements, and thus we can guarantee that any given update is both before-or-after and all-or-nothing atomic. As is such, the array will never be left in an inconsistent state, and therefore the sum of the elements will be zero at the end of the run. Below the level of abstraction of our code, this is implemented using a compareAndSwap function, which corresponds to the machine code command of the same name. This command both updates the variable and gets the old value; in doing so, it can guarantee that the update has the correct value at the end. If the value is not our expected value, it will attempt the update again.

4. Results

We will first discuss the results of varying the number of cores and the length of the array with a given approach. After that, we will tie the various approaches together, comparing their overall performance in terms of both speed and correctness.

4.1. Unsynchronized Memory Access

Our unsynchronized access, as expected, resulted in an output sum mismatch on both machines with any greater than one thread. On server 6, with less than forty threads, the average real and CPU times were roughly equivalent at ~16 nanoseconds for single thread, ~217 nanoseconds for 8 thread, and ~470 nanoseconds for 16 thread. When we reached forty threads, however, the real time followed the pattern to ~1200 nanoseconds, whereas the CPU time matched that of the 16 thread case. On server 10, the real time followed roughly the same pattern, albeit slightly faster, at ~12, ~155, ~445, and ~1030, respectively. The difference was in the CPU time; this seemed to increase logarithmically, with a time of ~12 nanoseconds on the single thread, ~77 seconds on the 8 thread, ~87 nanoseconds on the 16 thread, and ~102 nanosecond on the 40 thread.

As the number of threads increased, we saw the average real swap time increase nearly linearly with the number of threads with a constant of proportionality 2. This was true on both machines. Presumably, this is because the cost of creating the threads was too high relative to the cost of performing the actual swaps, so the variation in time cost was slightly washed out in terms of real time. The variation is seen in the CPU time; on server 10, we saw a reduced time cost. This is because server 10 is anticipated to have a smaller number of faster cores. As is such, while the threads have to wait longer, increasing the real time, the actual computation is performed faster and therefore CPU time does not increase as fast.

Access to arrays of different length, however, varied in efficiency according to a curve of some sort. On server 6, the array of length 5 had an average real and CPU swap time of ~216 nanoseconds. The array of length 25 had an average real and CPU swap time of ~489 nanoseconds. The 100 element array had an average real and CPU swap time of ~381 nanoseconds. On server 10, the lengths 5, 25, and 100 arrays had average real swap times of ~155 nanoseconds, ~322 nanoseconds, and ~271 nanoseconds, respectively. The same tests had average CPU swap times of ~77 nanoseconds, ~160 nanoseconds, and ~135 nanoseconds.

In both cases, we again saw similar responses in the real time. Both machines saw an increase in time cost when jumping from 5 to 25 elements, but saw a drop when jumping from 25 to 100 elements. The differences between the machines can be explained similarly to in the discussion of the thread count, but the actual curve needs another explanation. I suspect that in the case of the jump from 5 to 25 longs, the array exceeded the length of the cache. This meant that for a processor to update an element, it had to read from memory, a costly endeavor. In the jump from 25 to 100, it is increasingly likely that a processors cache may remain valid after an update, in which case the additional read could be neglected.

4.1. Synchronized Memory Access

Our synchronized access, as expected, resulted in a consistent result where the sum of all elements in the array equals zero. On server 6, the single threaded case saw an average real and CPU swap time of ~23 nanoseconds. The 8 thread case saw a real average swap time of ~2023 nanoseconds and a CPU average swap time of ~870 nanoseconds. The 16 thread case had an average real and CPU swap time of ~3514 and ~718 nanoseconds, respectively. The forty thread case had corresponding time values of ~9200 nanoseconds and ~770 nanoseconds. On server 10, the pattern was similar but much less drastic, with corresponding real time values of ~16 nanoseconds, ~397 nanoseconds, ~792 nanoseconds, and ~1993 nanoseconds. The CPU time values were ~17 nanoseconds, ~60 nanoseconds, ~60 nanoseconds, and ~63 nanoseconds.

As the number of threads increased, we again saw an increase in real average swap time. In the single thread case, synchronization is relatively inconsequential, so there was no difference between the real and CPU time. Once we got beyond a single thread, however, we saw the average real time increase drastically. This can be attributed to the additional wait time imposed on each thread on accessing the array. In both cases, however, once beyond a single thread, we saw only minor increases in the real time. This is expected, since the operations, while running, will take roughly the same amount of time. Server 6 has many more cores, so as expected, the threads had to wait much longer and therefore the average real time grew much faster.

With synchronized access, we saw an interesting pattern emerge in terms of time cost variation between array sizes. On server 6, the 5, 25, and 100 element arrays had real time costs of ~2023 nanoseconds, ~2100 nanoseconds, and ~2087 nanoseconds, respectively. The corresponding CPU times were ~870 nanoseconds, ~867 nanoseconds, and ~870 nanoseconds. On server 10, the 5, 25, and 100 element cases had real average swap times of ~397, ~387, and ~371 nanoseconds and CPU average swap times of ~60 nanoseconds, ~59 nanoseconds, and ~51 nanoseconds.

Since access to the array is synchronized in this case and thus threads must both wait and empty the cache in all cases, we saw very little change between cases. Server 10 performed much better on all measures, since it has a less number of faster cores, which results in smaller computation and wait time.

4.3. AcmeSafe Access

The AcmeSafe example highlights well the differences between the two machines. Server 6 saw 1, 8, 16, and 40 thread average real swap time values of ~23 nanoseconds, ~943 nanoseconds, ~1451 nanoseconds, and ~3666 nanoseconds, and CPU time values to match in the first three cases, with again no change between the CPU times of the 16 and 40 thread case. Server 10 saw the real average swap times ~25 nanoseconds, ~897 nanoseconds, ~2177 nanoseconds, and ~5476 nanoseconds, and corresponding CPU average swap times of ~25 nanoseconds, ~444 nanoseconds, ~529 nanoseconds, and ~540 nanoseconds.

These time costs parallel the case of the unsynchronized access. With more threads, we see an increase in wait time and computation time, although the computation increase is less drastic on server 10 due to the faster cores. The interesting thing with this method is that server 6 is faster than server 10; this is the first time it has happened, and it can be explained due to the little waiting being outweighed by the extra computation power of having 11 extra processors.

The patterns of the two servers as related to array length were completely different. Server 6, when running the 5, 25, and 100 element tests, saw real average swap times of ~~943 nanoseconds, ~983 nanoseconds, and ~547 nanoseconds. It saw roughly equivalent average CPU swap times. Server 10, when running the same tests, saw real average swap times of ~897 nanoseconds, ~546 nanoseconds, and ~437 nanoseconds, and CPU average swap times of ~445 nanoseconds, ~232 nanoseconds, and ~88 nanoseconds.

The array length average swap times between the two cases serve to highlight the differences in the hardware. When server 6 jumped from 5 to 25 elements, it saw a slight increase in time cost, since there were enough processors that updates were still relatively inefficient. Upon jumping to 100 elements, it saw a massive drop in average time cost, however, since the number of threads had now been exceeded, and the machine could run the maximum number of concurrent threads. On server 10, the computations, as expected, are much faster. The real and CPU average swap times follow the same trend, however. An increase in elements means less waiting, since more threads can run. Jumping from 5 to 25 elements allows server 10 to run the maximum number of threads, so we see a large jump in performance, while jumping to 100 causes a less drastic jump.

5. Summary

We will now compare the different approaches. It is worth mentioning, first, that the differences between the machines reared their ugly heads here; the extra threads on server 6 allowed for more parallelism, which in the case of locking resulted in more wait time. Server 10 spend much less time waiting, since it had less threads, and individual computations were cheaper, but due to a decreased optimal performance, it was defeated by server 6 when waiting was optimized away.

In general, we saw that both the unsynchronized and the AcmeSafe approach performed much better than the synchronized approach. The AcmeSafe approach was about two to three times slower than the unsynchronized in cases with many threads. AcmeSafe was still up to four times faster than the synchronized case as we extended the length of the array however. The inconsistencies resulting in the unsynchronized case prevent us from recommending it; as we work with large data, we recommend the AcmeSafe approach, since it removes inconsistencies while much outperforming the current model.