Jacob Grunwald

# Concurrent Arrays

**Introduction:**

The goal of this project is to implement and benchmark the performance of a concurrent array data structure. Arrays are useful ways to store data in general, but can be slow to access and find the data you need. To benchmark the performance of the array, I will be using pseudo random accesses to the array in order to stimulate and facilitate multiple reads and writes. Hopefully the result will be that the concurrent accesses are faster than the serial accesses. For the purpose of consistency, the array will be 512 integers and I will benchmark how long it takes to run 1024 reads and writes each with varying numbers of threads and concurrency methods.

**Example:**

An optimized concurrent array would be useful for the payload I am working on at Space Grant. We are to collect data characterizing the deployment of a boom in a zero g environment. We have a limited amount of time where our payload is powered, and we want to collect the maximum amount of data in that timeframe. In order to do this we could utilize a concurrent array. This would allow us to store all of the data we can while processing as threads become available. Optimizing the concurrent nature of the array would allow us to optimize the amount of data we could process during the mission.

**Proposed Solution:**

To address this problem I will first benchmark how long it takes to run 1024 reads and writes serially. I will run this 10 times then take the average after excluding the highest and lowest value to try and remove outliers. I will then repeat this process with coarse grained reentrant locks and a lock free implementation using atomic integers. Based on previous uses of these implementations, I think the atomic integers will likely be the fastest but will use the most memory. Coarse locks should be the second fastest, but will also be blocking which means some threads will spin and still take processing time. I think the serial approach will be the slowest.

**Proposed Quantitative Evaluation:**

In order to evaluate which method is most efficient for my purposes I will prepare a test bench to run each implementation through and mark how long it takes to complete. Ultimately what I need from this array is for it to transfer data as quickly as possible, so the total execution time will make a fine evaluation tool.

**Implemented Solution:**

The noLock.java implementation is actually the serial implementation and extends the circBuf.java class. noLock uses an array of booleans to track whether or not the location has valid data in it. This is used to prevent remove calls from getting uninitialized data. In the add function, the program will first check if the array is full, or if the data location that is going to be written to already has data in it. If either of those are true, it returns one which would make the program generate a new location to store the data then try again. The remove function works in a

similar fashion. It will return -3 if the buffer is empty or if the location it is trying to pull data from does not have valid data in it. For the purpose of this simulation, if a -3 is received, a new location will be generated to pull data from. noLock.java is tested using a single thread.

coarseReentrant.java also extends circBuf.java and functions similarly to noLock.java, but uses reentrant locks. These locks are locked as soon as the program enters the add or remove functions. I had originally considered doing a fine grained version of this, but realized that only the valid check at the beginning of the function would be able to be moved outside of the lock. I thought this change would be negligible in the execution time so I didn't make a fine grained locking implementation.

atomicBuf.java is very different from the other two implementations. This implementation removes all locks from the array and instead uses atomic integer arrays for the data and for the valid arrays. It also uses an atomic integer for the number of items in the array. The add function checks if the buffer is empty or if the data it is trying to write to is already valid and returns one if either are true. If this isn't the case, it will set the location in the array specified with the data. The remove function will do something similar. It checks to make sure the array isn't empty and that the data is valid and if both are true it will return the data from the location. If either is false, it will return -3.

To benchmark the implementations, I have set up a test file that I can easily change the implementation type as well as the number of threads and the size of the array easily. I currently have each thread adding its ID to the array every time it runs. Then after the array has filled, I dumped its contents to the terminal. In my first iteration I realized that one thread was doing most of the adds, but if I increased the number of spaces available in the array, then I started to

see other threads contributing. This lead me to believe that in the time it was taking to start each thread, the array was being mostly filled by the initially started thread. I decided that this may be affecting the speed of the execution so I implemented a cyclic barrier to start all of the threads at roughly the same time. To do this I added a CyclicBarrier type variable named gate, and added a gat.await() to all of the threads run routines. Then, when I was prepared to start the threads I ran a final gate.await(). After this, I saw a greater variety in the threads that were writing to the array.

To benchmark timings, I am using the System.nanotime() function to record the start time and the end time of the execution. This gave me somewhat precise data in the nanosecond range. I then piped this output data into a text file that's called output.txt.

**Results:**

| Implementation | Average Execution Time (ns) |
|---|---|
| noLock Serial | 6702807.625 |
| 1 Thread Atomic Integer | 5042294.375 |
| 30 Thread Atomic Integer | 175033108.125 |
| 100 Thread Atomic Integer | 597251090.625 |
| 1 Thread Coarse Grain Locks | 9563701.125 |
| 30 Thread Coarse Grain Locks | 90526229.75 |
| 100 Thread Coarse Grain Locks | 2060401189.5 |

The results of the testing surprised me. I found that the fastest implementation was the noLock serial implementation which was about 2 to 10 times faster than either other implementation. The only exception would be the single thread implementation of atomicBuf.

This implementation was actually slightly faster than the serial implementation. The only explanation I can think of to explain this is that atomic reads and writes are faster than non atomic reads and writes. This could be true if the number of assembly instructions per atomic read or write are fewer than the number of instructions for a normal read or write. This would also make sense to me based on how atomic reads and writes work. Since the read or write needs to be atomic, it needs to occur in a single instruction. This would reduce the amount of time by a small amount because of the reduced instructions.

I found that as I increased the amount of threads in general, the total execution time increase, but not linearly. I'm not sure what kind of relation exists there, but it appears to be logarithmic for the atomic integer implementation and either quadratic or tetric for the coarse grained lock. I imagine these changes are due to the large amount of context changes in the processor as it switches between threads. I think my approach was likely slightly flawed in that a lot of my threads had local variables that would increase the amount of time needed for context switching. A better approach may have been to have the threads run some kind of scheduler that calls a thread out of an available pool of threads to run an add given a certain set of inputs. This would have created less context switching as only the threads that have an instruction to execute would be running at any given time.

**Lessons Learned:**

I learned a lot about java in general while working on this project. Throughout the course of the class my understanding has been kind of shaky, but through this project I have had to actually understand some of the key aspects of the language. I now know how to implement

multiple threads with little difficulty as well as how to debug code. I still haven't figured out how to run a debug session with java, but I have a pretty good idea of what I can do otherwise to test the code. For example, I was originally having difficulty with some of the conditions I was using to test corner cases for the data structure. For example I was struggling with an infinite loop I kept encountering while trying to add in the remove function to threads. I added in some print statements to find out exactly where each thread was getting stuck and found out that some threads would get stuck after most had executed. This let me realize that the problem wasn't in my logic for the array, but in how I was assigning locations to write and read to and from.

I also think that this project really taught me what is and isn't really parallelizable. I now know that what I was trying to do could never really be implemented more quickly in parallel because each read and write needs to secure too much of the structure before it executes. This combined with the amount of context switching involved results in an overall slower implementation when compared to the serial implementation. I think parallelization is better utilized in complex, but repetitive tasks like graphical processing or large scale data processing rather than with things on the smaller scale like what I was attempting with this project.

If I could do this project again, I would change my topic to something more clearly parallelizable so I could get results that are more exciting than "No, this doesn't work how you think it does."