Concurrency and Parallelism - Report 2

Francisco Cunha & Ricardo Esteves

November 27, 2017

Abstract

For this assignment, we were given the task of parallelizing the implementation of a Java program that uses a linked list to implement a set-of-integers. After inspecting and testing the original source code, we proceeded to develop the different proposed concurrent versions. After various tests and analysis (in the 16-core test machine), we examined and compared the behavior of the different strategies. This lead to interesting results - some of them were expected but others were a bit more surprising.

1 Introduction

In order to build an efficient, concurrent data structure, one might think that we could simply take the original sequential implementation of the `IntSetLinkedList` class, add a scalable lock field to it and ensure that each method call acquires and releases the lock. This approach - which we call coarse-grained locking - is very easy to implement and reason about. However, - and as we'll see with the multiple variations of coarse-grained locking later on - they don't offer us the best performance. By locking the whole structure, we'll be presenting sequential bottlenecks (that increase with the number of envolved threads). Whilst definitely thread-safe, this approach won't let us retain the real benefit of concurrency - speed! -, and so, we must try different algorithms that work around varying levels of granularity.

2 Approach

2.1 Starting out with Coarse-Grained Locking

Following the project's guidelines, we first implemented the `IntSetLinkedListSynchronized`. The usage of synchronized methods (and not blocks!) focuses on simplicity and general-purpose, and so, adding the synchronized keyword to our data structure's methods granted us a rather correct yet elegant solution.

We then proceeded to our second implementation, `IntSetLinkedListGlobalLock`. This approach consisted in protecting our data structure with a single, global lock. To achieve this, we used java.util.concurrent's ReentrantLock. It provides more extensive locking operations than those that can be obtained using synchronized methods and statements, but we didn't *actually* use any of these extra functionalities (lock polling, configurable fairness, ...). Thus, we simply ensured that every method call to our data structure's methods would acquire and release the global lock, by wrapping the methods' bodies in try/finally blocks.

As for our last coarse-grained approach, we carried out `IntSetLinkedListGlobalRWLock`. Similarly to the previous approach, we were still relying on a single global lock to protect our data structure. The difference is that, instead of using a Lock object (specifically, the ReentrantLock), we used a ReadWriteLock (ReentrantReadWriteLock). Essentially, this ReadWriteLock maintains a pair of locks: a read lock that may be held simultaneously by multiple reader threads (as long as there are no writers), and a write lock, that, in the other hand, is exclusive. With calls to the list's `add()` and `remove()` acquiring and releasing

the *write* lock and calls to `contains()` doing the same for the *read* lock (once again, with try/finally blocks), we ensured safe and correct execution in a multi-threaded environment.

2.2 Moving to Finer-Grained Locking

Following the order, we shifted to finer-grained strategies, and `IntSetLinkedListPerNodeLock` was the new target. Now, instead of using a single lock to synchronize every access to the data structure, we added a new Lock field to the Node class itself. This way, method calls would only interfere when trying to access the same nodes at the same time, allowing concurrent threads to traverse the list together in a pipelined fashion. To prevent undefined behavior in multi-threaded environments, we used hand-over-hand locking to traverse the list (in all operations). This meant that, whilst safe and *theoretically* more performant than the previous coarse-grained solutions, the long chains of acquire/release operations could introduce considerable overheads and, thus, lead to inefficiency.

If the operation of searching for a specific node were to succeed more often than not, we could take an optimistic approach and completely ditch the overhead-increasing hand-over-hand locking, traversing the list without acquiring any locks at all. Once found, we'd lock the node we've been looking for (call it NodeA) and its predecessor (call it NodeB). Afterwards, we'd check if NodeB points to NodeA and if it is reachable from head. If this validation failed, we'd release the locks and start over. This is how we dealt with `OptimisticPerNodeLock`.

The problem with the previous optimistic approach is that the `contains()` calls - arguably the most common ones - are blocking. To improve this, we carried out a lazy approach in `LazyLockPerNode`. We added a Boolean field - *marked* - to the Node class, indicating whether that node is in the set. List traversals would no longer need to lock the target node (they would just check its marked field), and validations would now be a `O(1)` operation - no traversals needed, as a new invariant would be maintained: every unmarked node is reachable.

As a thread traverses the list, each time it advances curr, it checks whether that node is marked. If so, it calls `compareAndSet()` to attempt to physically remove the node by setting pred's next field to curr's next field. If the call fails, the thread restarts the traversal from the head of the list; otherwise the traversal continues.

For the `add()` method, call the node we want to add A. We start out by finding the neighbour/edge nodes, PRED and CURR (i.e. PRED –> (A) –> CURR). Unless the node we're trying to add is already in the list, we set A's next to CURR, and finalize by trying to set PRED's next to A, with `compareAndSet()`. For the `remove()` method, call the node we want to remove B. We start out by finding the neighbour/edge nodes, PRED and CURR (i.e. PREV –> B –> CURR). Unless the node we're trying to remove is not in the list, we try to mark CURR as logically removed, using `attemptMark()`. In case this attempt was successful we finalize by physically removing CURR, using, once again, `compareAndSet()`. The `contains()` method is, once again, wait-free.

3 Validation

The primary way for us to validate our solution was to effectively run the benchmarking application under different conditions of concurrency and workload, and see if the results observed would match our expectations. We'll go through those expectations breafly here, but in the next section we'll go in-depth in all of these, one-by-one.

A. On average, the higher the write rate, the lower the throughput. This should stand true for all algorithms.

B. All three of our coarse-grained implementations (Synchronized, Global and Global R/W) should perform roughly as well as the original sequential version **when executing with a single thread**, independently of the other parameters used.

C. Our coarse-grained solutions (Synchronized, Global and Global R/W) should all perform better with a single thread than with multiple ones.

D. The Global R/W implementation should perform better, on average, than the GlobalLock implementation. However, this performance difference should be less and less visible as the percentage of write operation increases.

E - Fine-grained PerNodeLock should be better than all coarse-grained algorithms.

F. For small write rates, the Lazy algorithm's throughput should increase with the number of threads (up until the number of real physical threads). Although, for higher write rates, the algorithm's throughput should decrease as the number of threads increases.

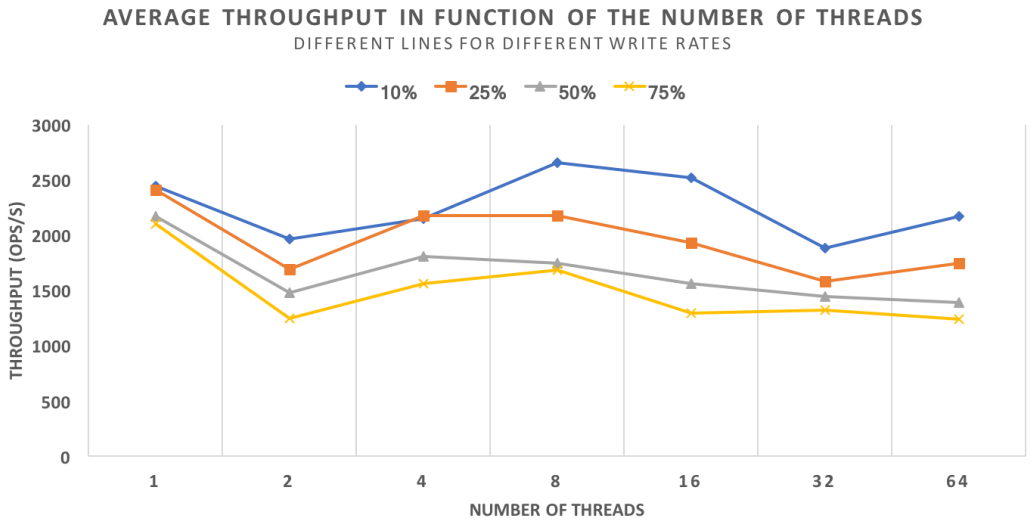
G. The LazyLockPerNode implementation should perform better overall than the OptimisticLockPerNode. This performance difference should be less and less visible as the write rate increases.

H.The Lazy implementation should perform roughly as well as the Non-Blocking one.

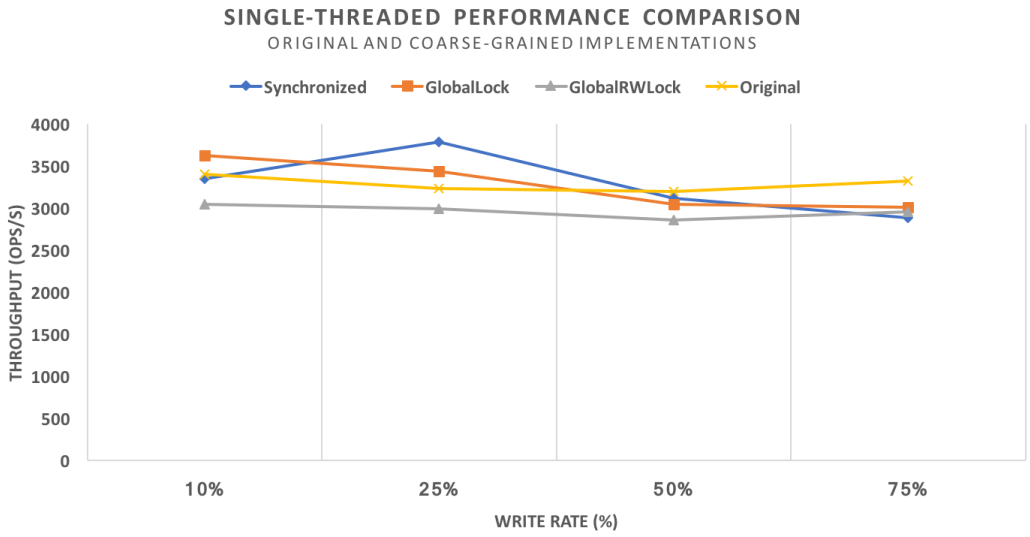
4 Evaluation

The results that we'll show were obtained from the performance tests ran on the provided multicore machine. Throughout the whole process we used a warmup value of 2000ms, and each individual test ran for 5000ms. We executed a lot of tests for each implemented algorithm - specifically, multiple times each combination of concurrency and workload (i.e. 1, 2, 4, 6, 8, 16, 32, 64 threads and 10, 25, 50 and 75 percent write rates). In order to get trustworthy results, the values we used were the average from each set of tests.

The first thing we noticed was how, on average, the throughput of (all) the algorithms would decrease as the write rate increased, independently of the number of threads. This confirmed expectation A (see above), and can be seen in the chart below, where we calculated the combined average throughput of **all** our implementations for different amounts of threads and write rates:

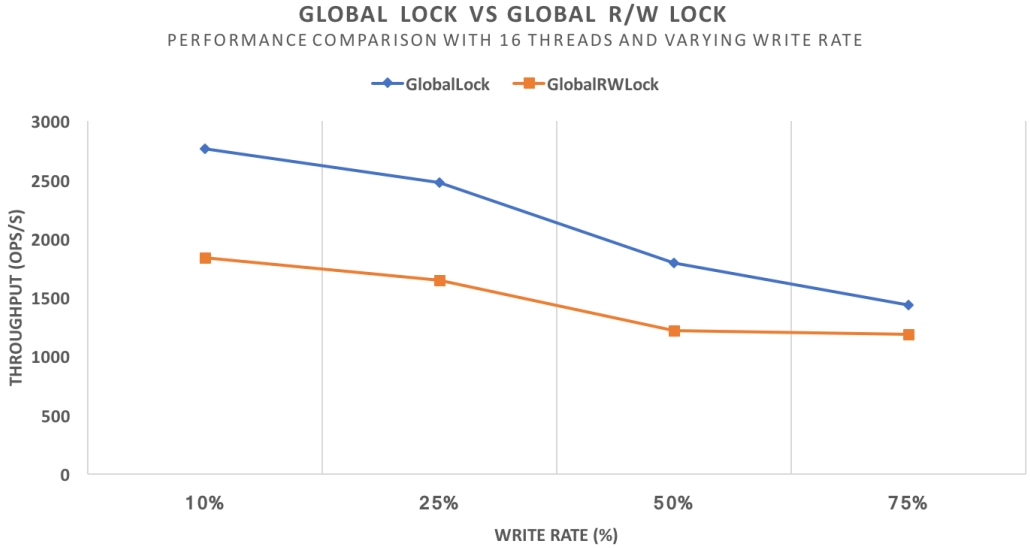


Moving on, we started analyzing our coarse-grained algorithms:



Indeed the three coarse-grained algorithms performed roughly as well as the original, sequential list implementation. Why was this expected? Because in a single-threaded environment, all those implementations would be essentially sequential, and the locking overhead wouldn't be too noticeable. We also registered that these coarse-grained solutions performed better with a single thread than with multiple ones. This was also expected, as these were purely competition concurrency scenarios, and so additional threads would only case more contention and sequential bottleneck. With this results, we knocked down expectations B and C.

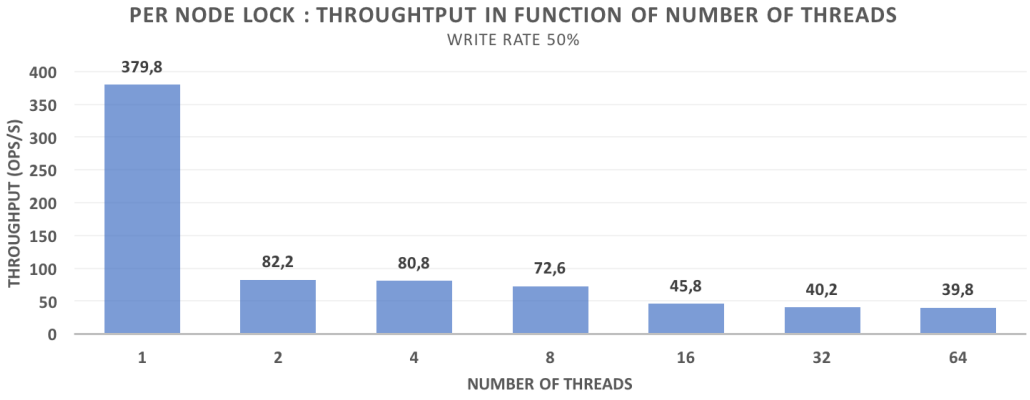
Between GlobalLock and Synchronized there weren't visible performance differences (from our research, this could be different for JDK 1.5 and below)[1]. However, we assumed that under small write rates, the Global R/W implementation should perform better, on average, than the other former two:



As you can see on the chart below, this wasn't exactly the case. From what we can see here [2], RW locks are not built-in to the JVM, unlike synchronized blocks, and use CAS which is pretty expensive in Java, so that could explain the difference there. When comparing with the Global lock implementation,

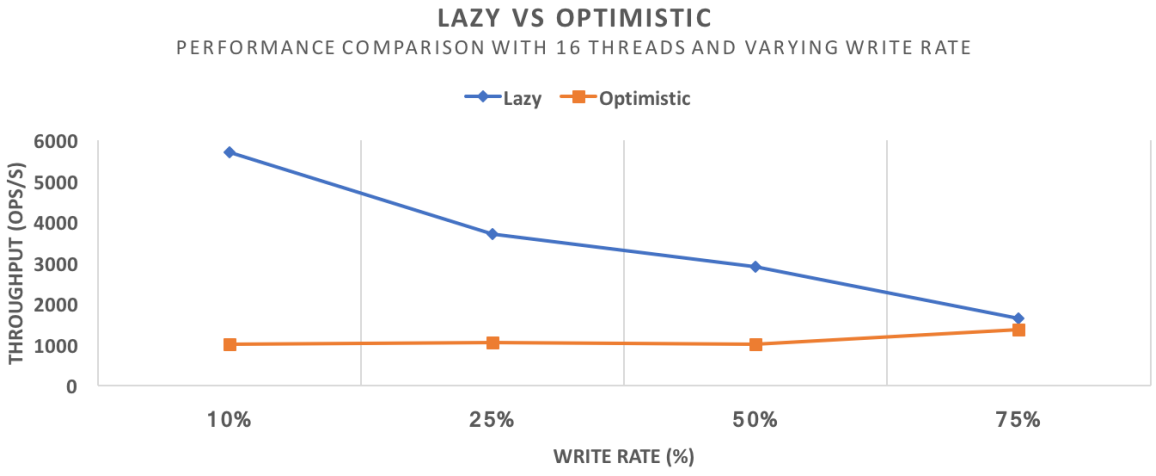
however, the only explanation we came with is that the access patterns for the shared data aren't suitable for this type of locking approach (surprisingly even under low write rate scenarios), and so the extra overhead and the costly operations deteriorated performance. With this analysis, we checked assumption D off the list and moved on to the finer-grained algorithms.

Here we encountered probably the most surprising results. We expected our PerNodeLock implementation to be straight-out better than all three of the previous coarse-grained algorithms discussed above. Yet, the observed results didn't match this at all:



It is effectively better in theory, but, in practice, it is highly dependent on the implementation itself. Lock acquires/releases are expensive operations in Java, and so the hand-over-hand traversals (which caused major chains of these two operations) severely deteriorated the algorithm's performance. The highest throughput value we registered for this algorithm was a whooping 379.8 ops/sec, which was far, far below all others.

The Optimistic and Lazy algorithms' analysis were next. Both performed well; however there were very noticeable differences between the two's performance for low write rates, as you can see below. Why exactly was this? Because the lazy `contains()` method has linear time complexity, as it will only traverse the list once, without locks, making it effectively wait-free. Being wait-free was essentially the justification of our expectation G: for small write rates, the algorithm's throughput did effectively increase with the number of threads (up until the number of real physical threads, of course). This gap became less and less noticeable as the write rate increased.



Under low write rates (read: observed for 10 and 25 percent), both the Lazy and the Non-Blocking throughputs scaled positively as the number of involved threads increased. This was expected, as, at the end of the day, they both have very similar wait-free `contains()` methods. However, the former scaled much, much faster than the latter. Based on some research, we figured that the root for this was not only the extra cost and overhead imposed by the presence of an atomically markable reference structure, but also its `compareAndSet()` performance. Unnecessary traversals forced by concurrent cleanup of removed nodes could also have some impact on the Non-Blocking algorithm's performance.

5 Conclusions

In conclusion, we can say that there's definitely a trade-off involved in the process of picking the optimal granularity to work with. If, on the one hand, we go for the more coarse-grained locking, we'll benefit from a (generally) simpler solution (engineering-wise), but our performance will start to suffer when multiple processes are running concurrently. On the other hand, if we join the finer-grain side of locking, we'll benefit more from the perks of concurrency (namely speed), at the cost of much more complex implementations. For this assignment - and based on our observations - lazy synchronization fitted us the best. Yet, It's most certainly not a straight-forward process, and making a good decision will involve experience, astuteness and a whole lot of testing.

6 Bibliography

[1] ReentrantLock vs Synchronized

`https://blogs.oracle.com/dave/javautilconcurrent-reentrantlock-vs-synchronized-which-should-you-use`

[2] ReadWriteLocks and Synchronized
`https://blog.takipi.com/java-8-stampedlocks-vs-readwritelocks-and-synchronized/`

[3] Raynal M.,(1998), *"Concurrent Programming: Algorithms, Principles, and Foundations"*,ISBN: 978-3-642-32026-2