CS5510: Multiprocessor Programming : Homework V

Sergio Sainz

November 2019

1 Part II : Programming [44 points]

1.1 Evaluating Sets (20 points)

In this implementation, you will write a microbenchmark to test each of the linked-list algorithms for sets described in Chapter 9 of the textbook; for the locking algorithms, replace usage of explicit locks with synchronized blocks that provide the same semantics as the explicit lock usage (this will not be possible in all cases). Don't forget to add volatile to those variables that need it (but keep usage to a minimum as otherwise performance may suffer). Note that there may be errors in the code that you will need to correct as well.
Use sets of integers (Set¡Integer¿) as the implementation to benchmark. A worker thread does an operation ITER times. The operation on the set could be either add, or remove or contains. For add and remove operations, the concerned value is passed as a parameter.
Experiment with different values of ITER (e.g. 5000, 10000, etc.) and choose one that best fits your machine or RLogin, so as to not overload or underload them. Use numbers in range of 0 to 100 (a random choice) for add/remove operation for each worker thread as it ensures some degree of contention. You will need to vary workload parameters (e.g. ratio of contains/add/remove operations) and find trends in behavior of all algorithms. Vary percentage of contains operation from 20% to 80% with the following spacing - 20%, 40%, 60% and 80%. Divide rest of the operations between add and remove equally as required. Benchmark from 4 threads to 40 threads with reasonable spacing and at least for 8 different thread counts (Use Rlogin).
There are two ways for approaching mixed workloads. Choose whichever you feel comfortable with.

1. Have worker threads that only perform one operation. For example, for example , for 20 threads, you can have 12 threads that always invoke contains, and 4 threads each that always invoke add/remove. Approximate as required.

2. Each thread chooses between contains, add and remove through a uniformly distributed float random number generator.

Hint: Read about java.util.concurrent.ThreadLocalRandom. Use it in the run() method of your thread.
Don't forget proper benchmarking practices (accounting for JVM warm-up- the more the better, no points of contention beyond the set object itself,etc.); you do not need to use JMH, but you may find it useful to avoidthe pit- falls discussed in http://www.oracle.com/technetwork/articles/java/ architect-benchmarking-2266277.html.
Provide the following plots:

1. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 20; all algorithms.

2. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 40; all algorithms.

3. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 60; all algorithms.

4. *throughput vs. threads*; threads varying from 4 to 40 (at least 8 different thread counts); contains % = 80; all algorithms.

5. *throughput vs. contains %*; contains = 20%, 40%, 60%, 80% for fixed thread count = 20; all algorithms.

Analyze your findings in a writeup in less than 250 words.
Answers

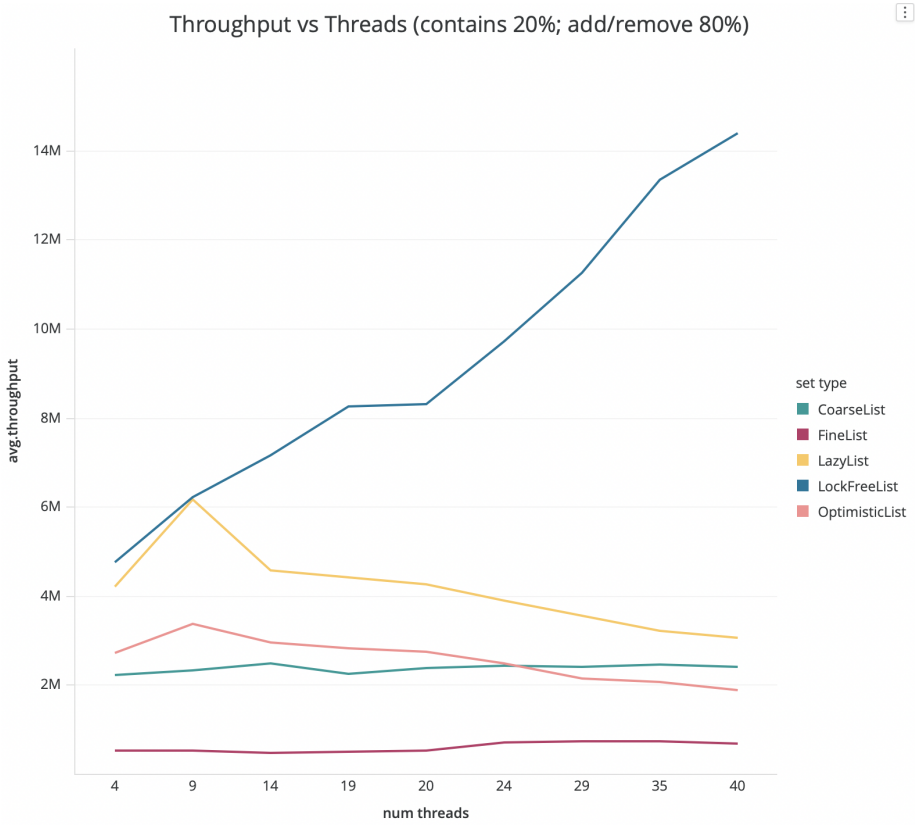

Figure 1: Throughput vs Threads - 20% of contains

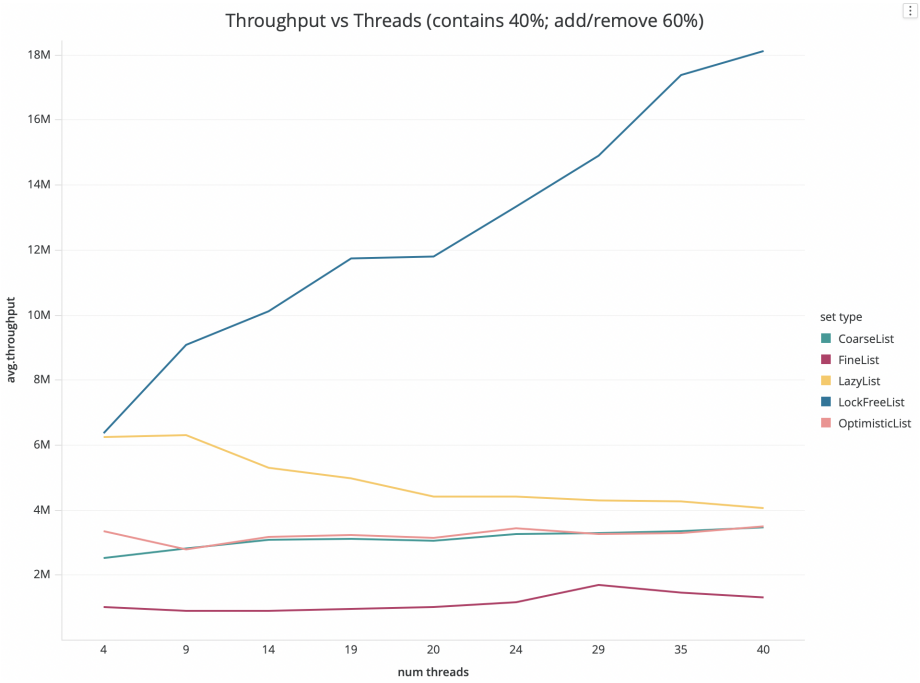

Figure 2: Throughput vs Threads - 40% of contains

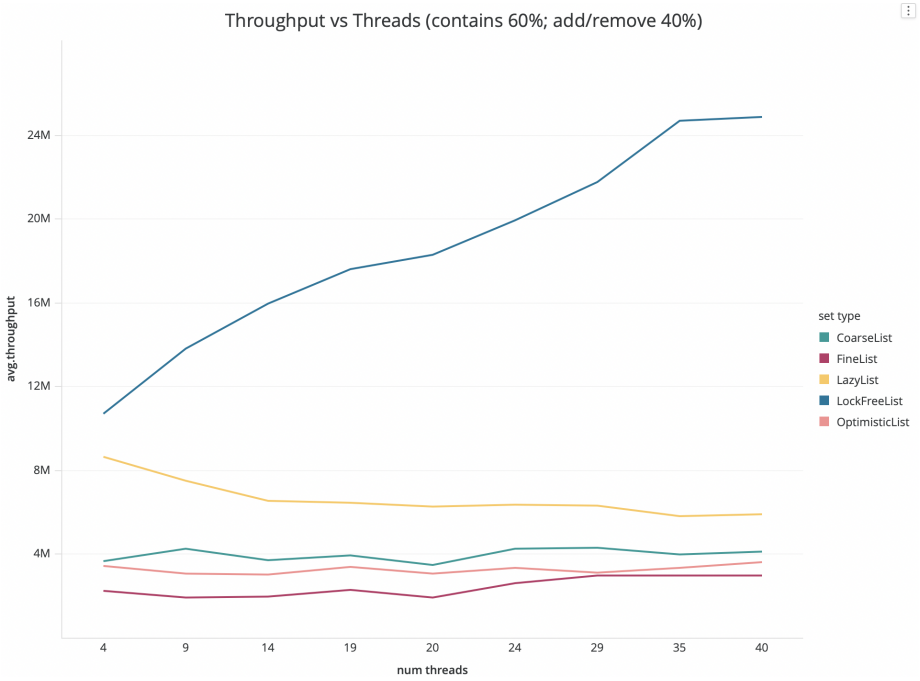

Figure 3: Throughput vs Threads - 60% of contains

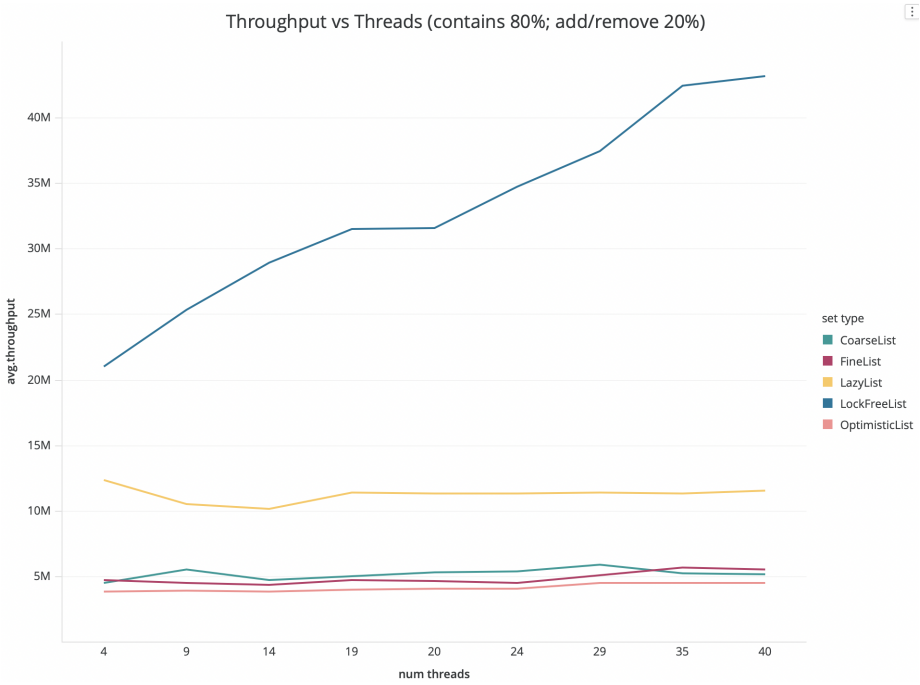

Figure 4: Throughput vs Threads - 80% of contains

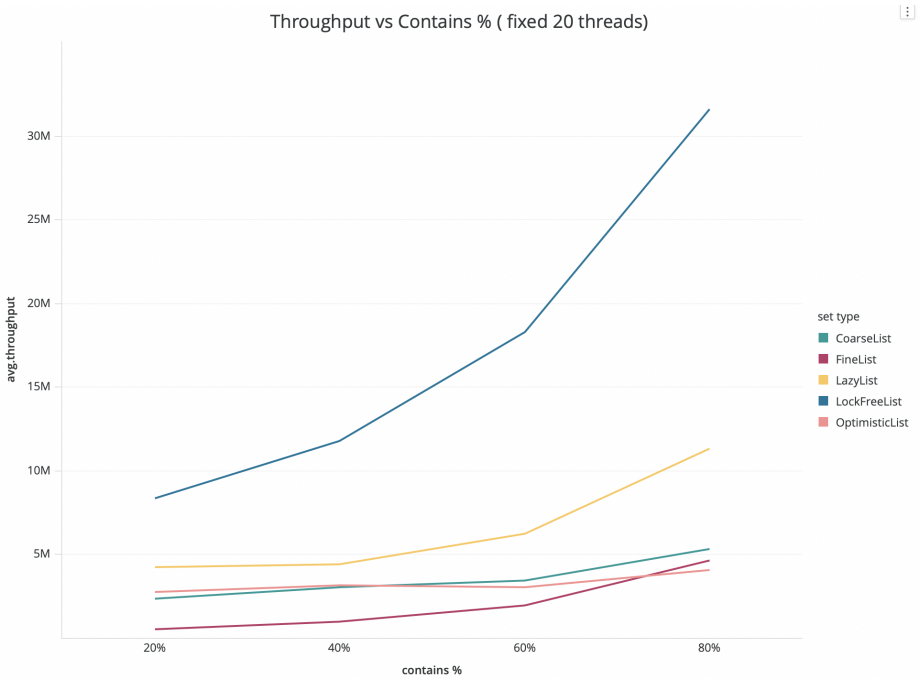

Figure 5: Throughput vs Contains percentage - 20 threads fixed

We can appreciate that the lock free list is the fastest one as expected. Then second best is the lazy list as expected. Nevertheless, the rest of the results are not as expected. For example, in the test where the number of contains is 20%, the worst performing data structure is fine grained list for wide margin. Later this margin decreases little by little until the point that in the test where number of contains is 80%, the worst performing list is the optimistic list followed by both fine-grained list and coarse grained list. My explanation for this is that I could not find easy way to remove the lock and add synchronized instead in fine grained list methods. Meanwhile for all others I could remove the locks (except the contains in Lazy and the methods in Lock free list). Maybe if we add back locks to other methods fine grained list wont perform in such low ranking relative to other data structures.
Another surprising result is that the optimistic list did not perform well as expected when the number of contains is 80%. I believe this is because the optimistic list does two synchronization one in pred and one in curr. Meanwhile the coarsed grained just does one synchronization in the list itself. Perhaps the mechanism in java to synchronize any object is more expensive than the cost of having other threads waiting for the list synchronization bloc to become available. And thus the optimistic list performs worse than coarse grained list in high number of threads.

1.2 Concurrent Lock-free Bag (24 points)

Implement a lock-free bag data structure for multiple producers and consumers, supporting the Add and the TryRemoveAny operations. The Add operation adds an item to the collection and the TryRemoveAny operation removes an item from the collection unless it is empty. The collection allows any number of occurrences of an item and keeps no information about the order in which items were inserted. A common use of this kind of collection is to communicate data items between producers and consumers in the task of parallelizing applications. Speedup is a major goal in these scenarios where both data and task parallelism are heavily exploited and the bag merely constitutes a "glue" in the application design. Because of this, it is essential that the collection implementation have as high scalability and low overhead as possible. By measuring throughput vs. threads, show that your lock-free bag imple- mentation outperforms the lock-free LinkedList implementation for this use case.

Design Ideas:

Don't confuse with FIFO queues. Remember, bags allows for duplicate items and don't care about removal order. Thus, reduce contention to zero when adding items. For removal, enable threads to remove items from different non-overlapping locations such that the contention is zero most of the time. Modify the LockFreeList to enable it to accept duplicate items. Construct your bag using the LockFreeList. To provide disjoint access parallelism (i.e. to allow different threads to add/remove without contending), more than one instance of LockFreeList may be required. For instance, one per thread: however, you should only create new LockFreeList instances as threads arrive (no prepopulation); note that remove operation must succeed if there is an item in the bag (not just on the thread-owned portion).

Answer

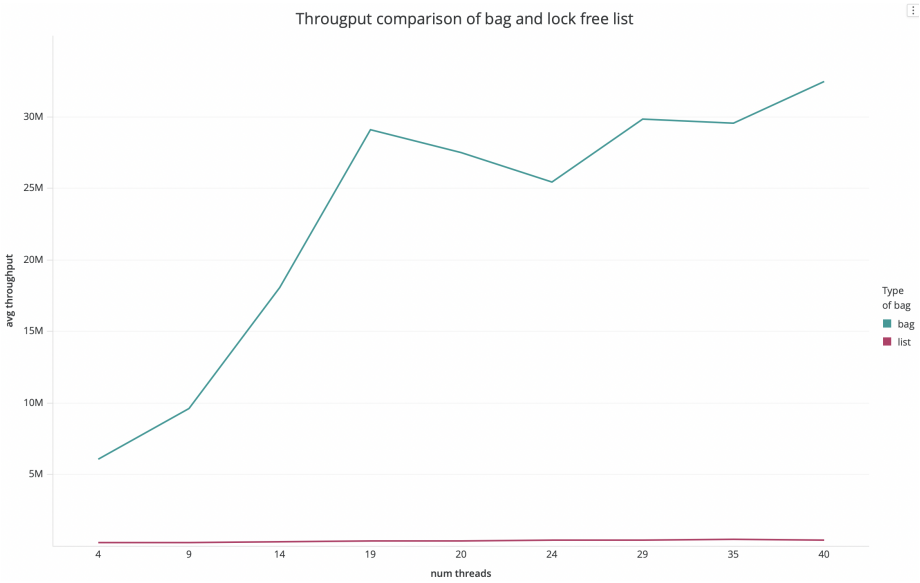

Figure 6: Throughput of bag vs lock-free list with different threads

As we can appreciate from the figure 6 the performance of the bag is much better based on operations per second compared with the lock-free list.

Implementation details are that I follow recommendations from the homework: (1) use a list of list to keep control of different entry points while still remove items from other threads' lists.