ECE/CS 5510 Multiprocessor Programming
Homework 5

Mincheol Sung

November 3, 2018

Part I

Problem 1.
Yes. It is linearizable.
In case of true, there must have been a moment during the call when the node was reachable from head. We can linearize the call at any such moment.
In case of false, there must have been a moment during the call when the node was not reachable from head. We can linearize the call at any such moment.

Problem 2.
No. It can deadlock because remove() method locks the same nodes with different order.

Problem 3.
There are three cases where the add() can be overlapped with other method.

1. Overlapped with other add()
Let there be two threads and a list $a \rightarrow b \rightarrow c$. If they are trying to add an entry at the same position, one has to wait for the other because of the pred lock.
In case that one thread is trying to add a' between a and b, and the other is concurrently trying to add b' between b and c, there is no confliction.
In addtion, if a thread is trying to add an entry after an entry that is currently being added by the other thread, it has to validate it.

2. Overlapped with remove()
In case of the curr in the add() being removed, a thread holds the pred lock and prevent the other thread from proceeding.
If the node marked as pred is the one being deleted, it is safe thanks to the lock of the pred.

3. overlapped with contains()
The contains() method does not cause any problems since it simply goes through the list and checks whether it contains an element in the list. Without locking the curr in add(), overlapping of add() and contains() is fine.

Problem 4.
Let there be a list $a \rightarrow b \rightarrow c$. During a thread is trying to add b' between b and c, other thread removes the b.
(1) expected reference: The adder thread expects b is the curr. But the remover thread advances the reference in #14.
(2) marked value: The adder thread expects the expect mark as false. But the remover thread marks it true in its #11.

Problem 5.
Let there be a list $a \rightarrow b \rightarrow c$.
(1) expected reference: During a thread is trying to remove b, other thread adds the b' between b and c. The remover thread expects c is the succ. But the adder thread adds b' in #11 that succ becomes b'
(2) marked value: During a thread is trying to remove b, other thread removes the a. The first thread expects the expect mark as false. But the second thread marks it true in its #11.

Problem 6.
When the new added object is removed, the node is marked as remove. This means that the other object in the node is marked as removed.

Problem 7.
During the node 10 is removed, the node 20 is added. The correct result is list of HEAD $\rightarrow$ 20 $\rightarrow$ 30 $\rightarrow$ TALE, but the result is list of HEAD $\rightarrow$ 30 $\rightarrow$ TALE

Part II

Problem 1.
I replaced lock(), unlock() with synchronized() in the CoarseList.java. I ran my benchmark on rlogin machine of boxelder with 40 cores and 100GB memory.
We can observe a trend that throughput decreases with increasing number of threads. With small ratio of the contains() and large number of threads, the LockFreeList outperforms others (Figure 1, Figure 2). However, the LazyList has the highest throughput in most cases (Figure 3, Figure 4). Figure 5 and Table 1 shows that the throughput increases with increasing contains ratio. This is because add() and remove() are costly than contains() in general. The FineList is the slowest in the most cases, and the OptimisticList is as slow as the FineList.

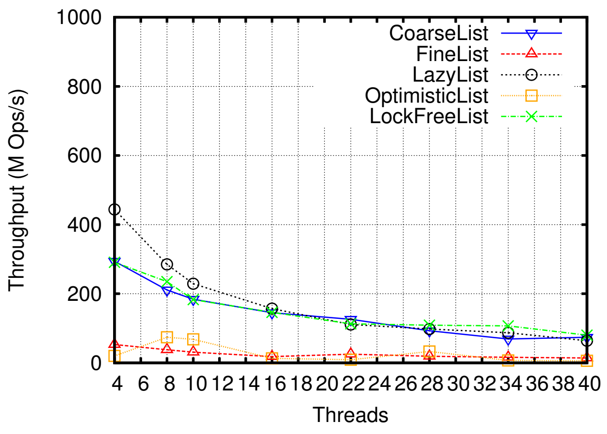

Figure 1: 20% contains

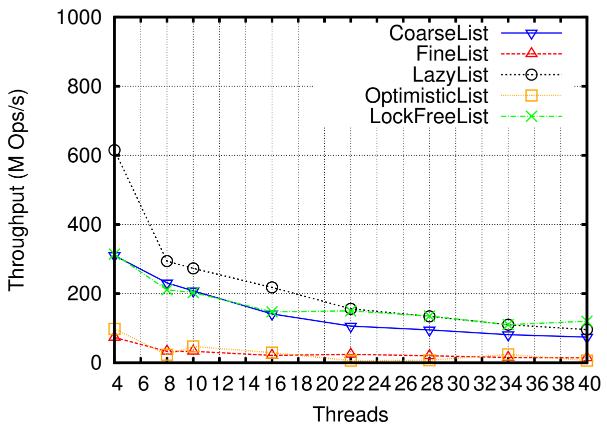

Figure 2: 40% contains

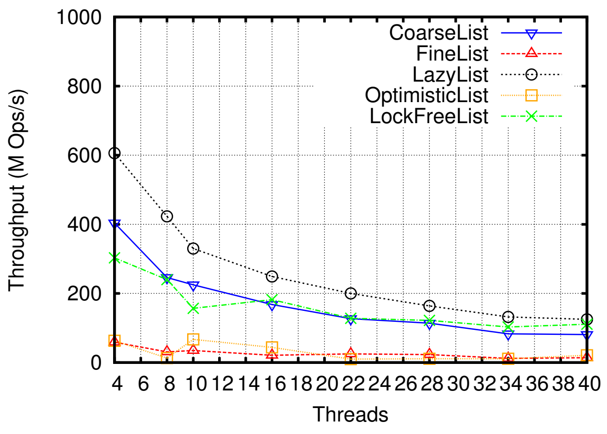

Figure 3: 60% contains

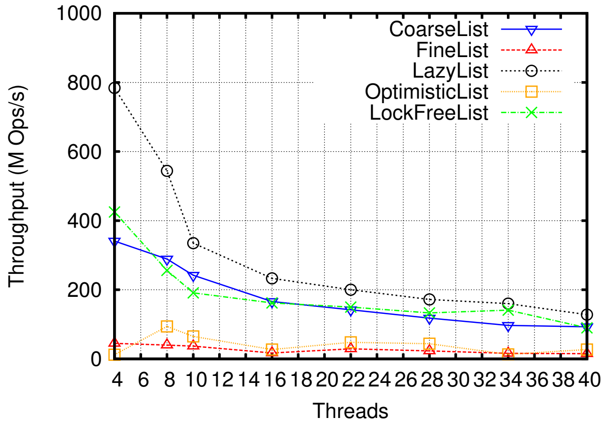

Figure 4: 80% contains

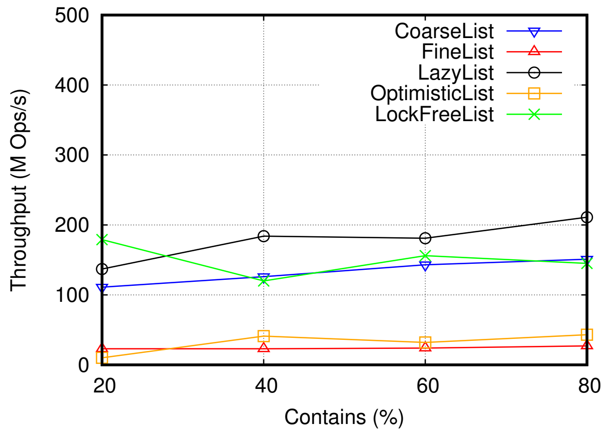

Figure 5: 20 Threads

Table 1: Thoughput (M ops/s) with 20 Threads and different contains ratio

Contains	20%	40%	60%	80%
CoarseList	111	126	143	151
FineList	23	23	24	27
LazyList	137	184	181	211
OptimisticList	10	41	32	43
LockFreeList	179	120	156	145