CSE 411: Assignment 1
Tradeoffs in Concurrent Set Implementations

*****Note: ' = 0% update rate, " = 50% update rate, "' = 100% update rate**

<u>Linked-List Based Sets: Hand-Over-Hand, Lazy, and Lock-Free</u>

Linked-List insertion executes with linear time complexity O(n), so throughput (transactions/second) diminishes as both the size of the data structure (i.e. the number of keys) and the update ratio increases. Throughput decreases as the update rate increases since you need to traverse the linked list more frequently. Updating the elements requires deleting the node and inserting the new value with O(1) amortized time complexity, but traversing the list for each update operation requires linear search time O(n). The parameters used for benchmarking these data structures includes: -i (initial size), -t (# of threads), -r (range), -u (update ratio), -A (alternate). -A is a particularly interesting parameter, indicating whether the benchmark alternates between inserting and removing the same value to maximize effective updates. Without this parameter, the effective update rate is capped to ~50%. This parameter allowed me to fine tune the range to achieve about ~99% effective update rate. And the data is normally distributed, so you should expect similar results running it a few times.

Lazy Linked lists incorporate laziness into the data structure by using lazy evaluation, which delays the evaluation of an expression until its value is needed. Hand over Hand locking (lock coupling) is a finer grain locking mechanism that associates a lock with every node rather than a single lock for the entire list. Lock-Free Linked Lists is a non-blocking scheme that uses atomic read-modify-write operations (and some other synchronization primitives) rather than implementing heavy locks.

(See Next Page for Plots)

Threads	1024	2048	4096	8192	16384	Update Rate	Data Structure
Threads (1)	855891.2	427728.6	92571.5	38740	17644.8	0%	Lazy Linked-List
Threads (2)	1622529.4	658493.5	167761.6	78547.1	34416.3		
Threads (4)	2460298.1	1301023.9	310127.7	141108.4	61166.6		
Threads (8)	2337737.9	1349455.5	310645.5	137266.2	60111.9		
Threads (1)	773766.2	379816.9	93005.5	37950.8	17598.1	50%	Lazy Linked-List
Threads (2)	1300570.5	648878.5	148218.5	68568.1	32640.3		
Threads (4)	1724415.2	1089638.3	281854.6	114929.8	53227.0		
Threads (8)	1638718.4	1020839	267926.4	111957.6	53039.5		
Threads (1)	685065.7	372976.3	88306.0	39648.6	17402.0	100%	Lazy Linked-List
Threads (2)	1219071.5	596421.1	143951.0	67543.8	32927.9		
Threads (4)	1700268.1	1091548.6	275577.9	111486.9	53868.1		
Threads (8)	1572690.7	1048515.1	275761.2	114845.6	54761.5		

Threads	1024	2048	4096	8192	16384	Update Rate	Data Structure
Threads (1)	460255.4978	239034.893	121431.5	48228.8	21112.1	0%	Lock-Free List
Threads (2)	881629.5482	449620.6276	228742.9285	81019.39806	39510.7		
Threads (4)	1424693.8	803131.1	409912.1088	154330.6669	70642.25043		
Threads (8)	1565447.3	817916.6	403274.8725	167550.7101	72247.37526		
Threads (1)	436064.1	233751.3	118612.6	47435.0	21113.9	50%	Lock-Free List
Threads (2)	859954.2	442392.4	223708.6	75612.9	34845.1		
Threads (4)	1370137.7	754509.1	405629.2	147853.1	64558.4		
Threads (8)	1350174.1	804343.2	413658.6	153564.8	64569.4		
Threads (1)	442355.2	229656.8	116194.6	47425.2	20044.9	100%	Lock-Free List
Threads (2)	822130.5	442219.4	222703.5	73892.8	33914.9		
Threads (4)	1402781.5	775710.6	396656.8	148922.4	61837.1		
Threads (8)	1394475.8	800603.1	407348.8	148791.4	62783.8		

Threads	1024	2048	4096	8192	16384	Update Rate	Data Structure
Threads (1)	33828.0172	16999.4003	8515.9	4353.4	2143.9	0%	Hand-Over-Hand List
Threads (2)	50348.9651	27335.4	13905.4	7508.549145	3880.1		
Threads (4)	73305.0305	44046.9953	24745.45091	13013.2	6715.9		
Threads (8)	44805.91941	29643.8	17025.1	9909.981996	5439.056094		
Threads (1)	34511.3	17194.3	8464.0	4288.7	2120.8	50%	Hand-Over-Hand List
Threads (2)	51960.8	26741.0	13794.5	7569.0	3962.0		
Threads (4)	70894.83155	43015.2	23648.0	12727.22728	6886.7		
Threads (8)	44860.9	29161.2	16992.2	9554.5	5383.0		
Threads (1)	33312.6	17001.6	8484.2	4193.2	2144.3	100%	Hand-Over-Hand List
Threads (2)	50784.7	25464.6	13693.1	7387.7	3940.5		
Threads (4)	63128.48715	42860.79744	23722.6	12837.2	6742.3		
Threads (8)	45111.08889	28876.9	16950.0	9740.2	5277.6		

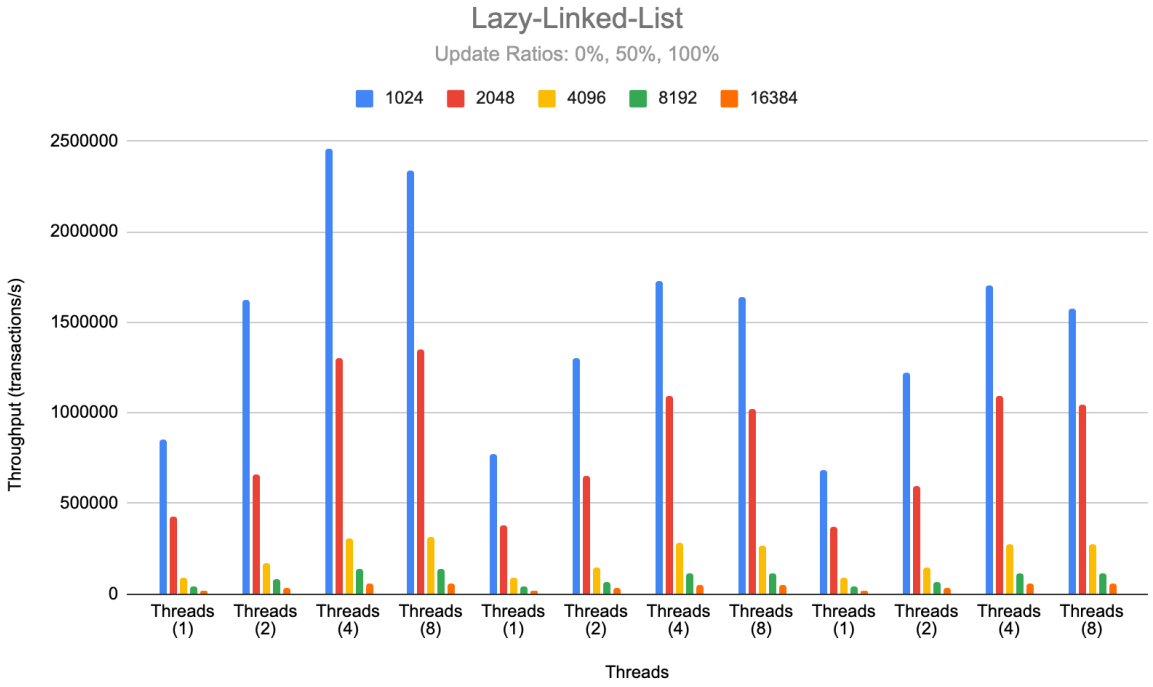

Figure 1: Lazy Linked List for Different Threads and Update Ratios

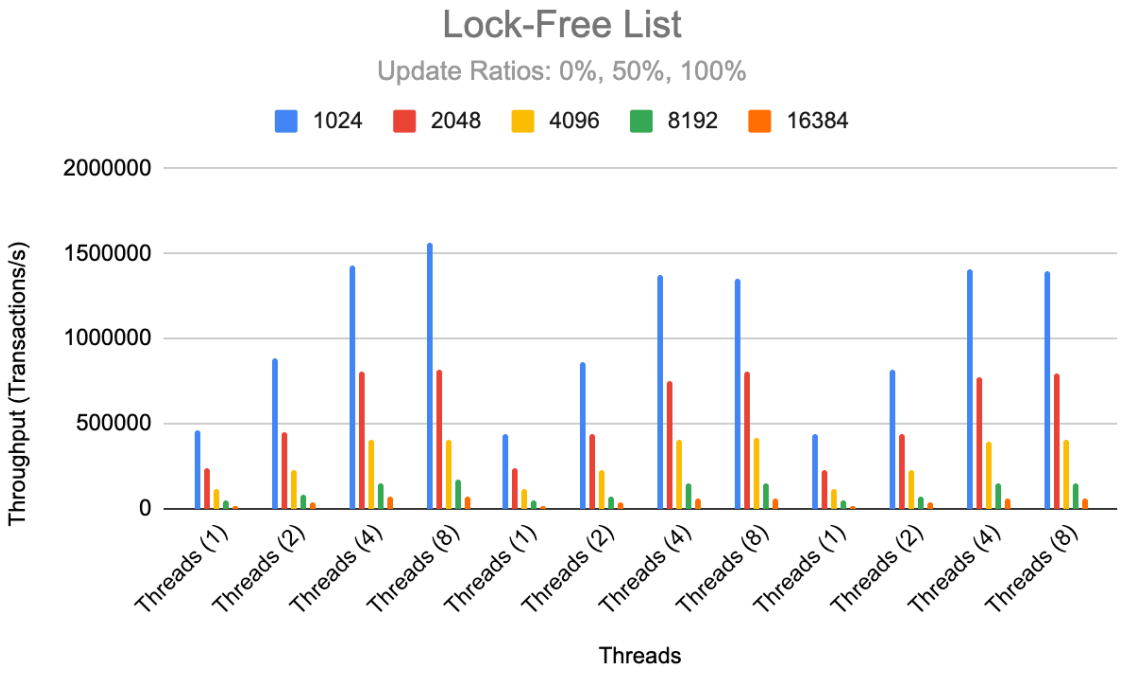

Figure 2: Lock-Free Linked List for Different Threads and Update Ratios

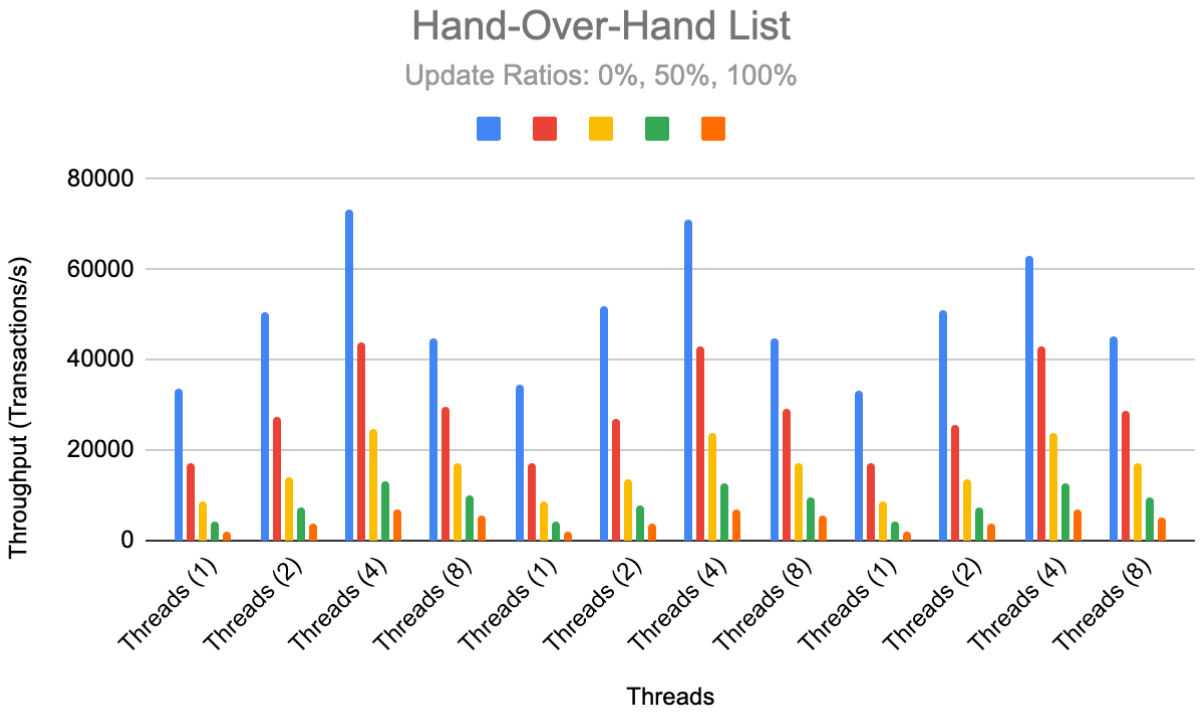

Figure 3: Hand-Over-Hand Linked List for Different Threads and Update Ratios

The graphs above are EACH divided into 3 separate quadrants by update ratios, respectively 0%, 50%, and 100%. In the graph, we see that the throughput degrades as the update ratio increases from 0% to 100%. Similarly, throughput consistently decreases as the size of the linked list increases from 1024 to 16384, whereby a 16x size increases results ~50x reduction in throughput, as the data structure is bounded by its linear time complexity. On the other hand, throughput linearly increases as the number of threads increases from 1 to 4, and then slightly falls after reaching its limits with 8 threads.

The performance statistics indicated that Lazy > Lock-Free > HOH in regards to throughput performance as a function of the number of executing threads. Lazy Linked-List performed the best, nearly 2x better than Lock-Free and 25x better than HOH. Hand-Over-Hand performed the worst, as throughput drops off after 4 threads, with the performance degradation compounded as the list's key size increases.

This is clearer when comparing the three different data structures to each other across the different threads. Again, the following labels indicate: ' = 0% update rate, '' = 50% update rate, ''' = 100% update rate. As shown below, lazy locking has the highest throughput (for all threads) since the algorithm is more efficient at evaluating expressions and performing calculations ONLY when it's value is needed. Hand-Over-Hand is the least efficient since [1] it's "don't look back" policy, where you can only traverse the list in a single direction (like a one-direciton

linked-list), and [2] since every node has a lock rather than a single lock on the entire data structure, that makes the underlying locking mechanism very expensive since there's many concurrent locks at the same time, and potential lock contention between multiple threads.

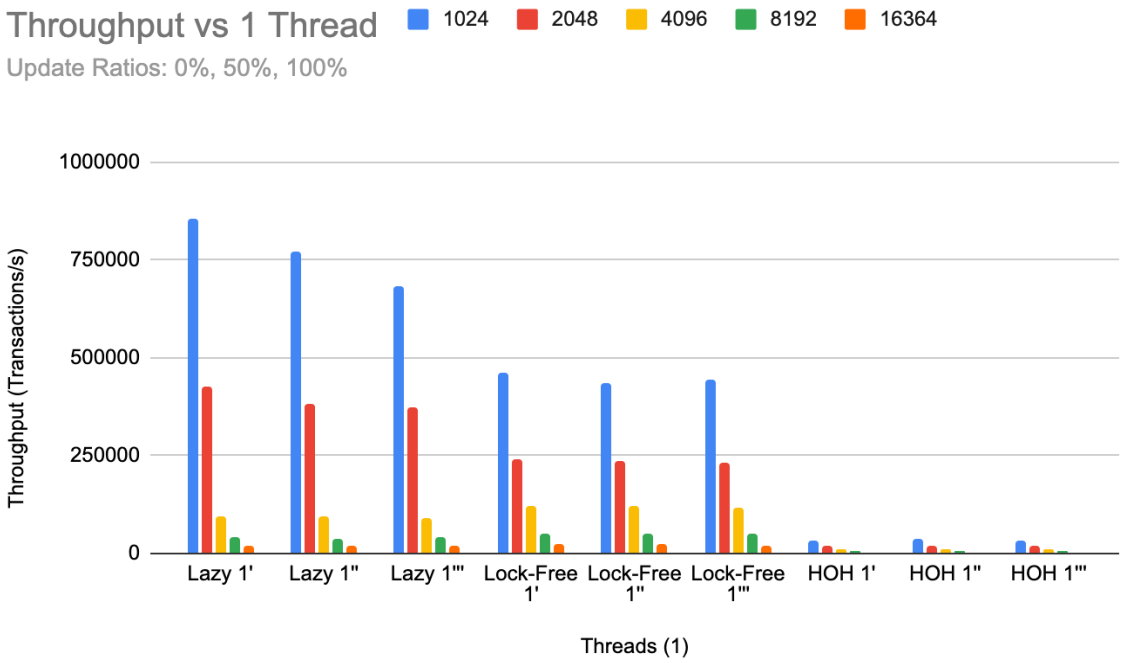

Figure 4: Lazy, Lock, HOH Lists Compared for 1 Thread of Execution

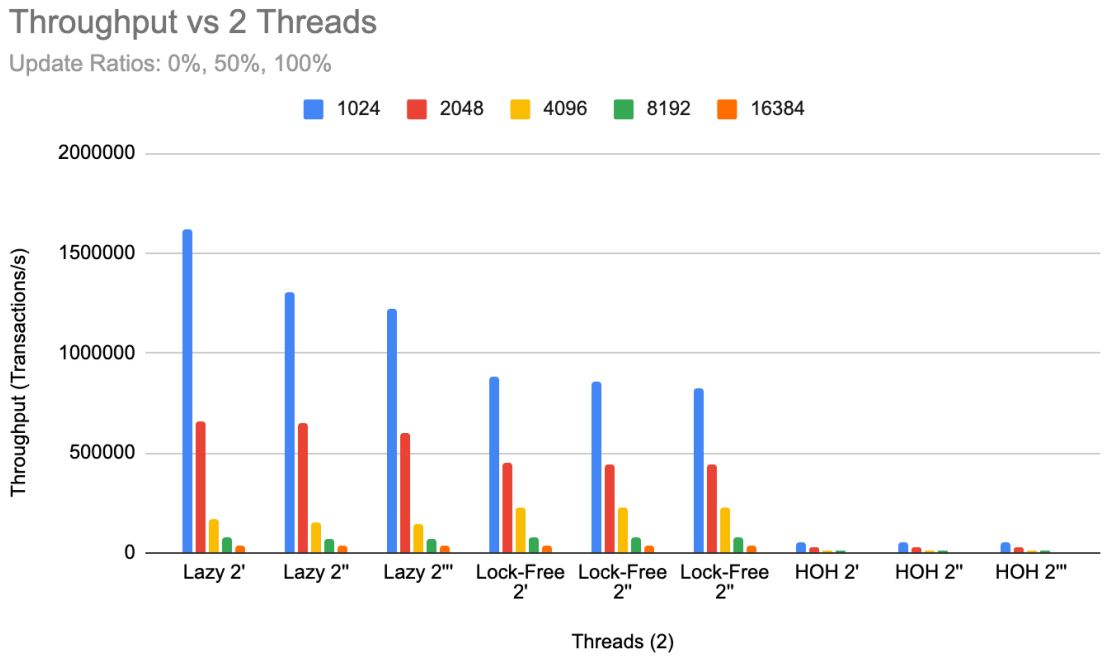

Figure 5: Lazy, Lock, HOH Lists Compared for 2 Threads of Execution

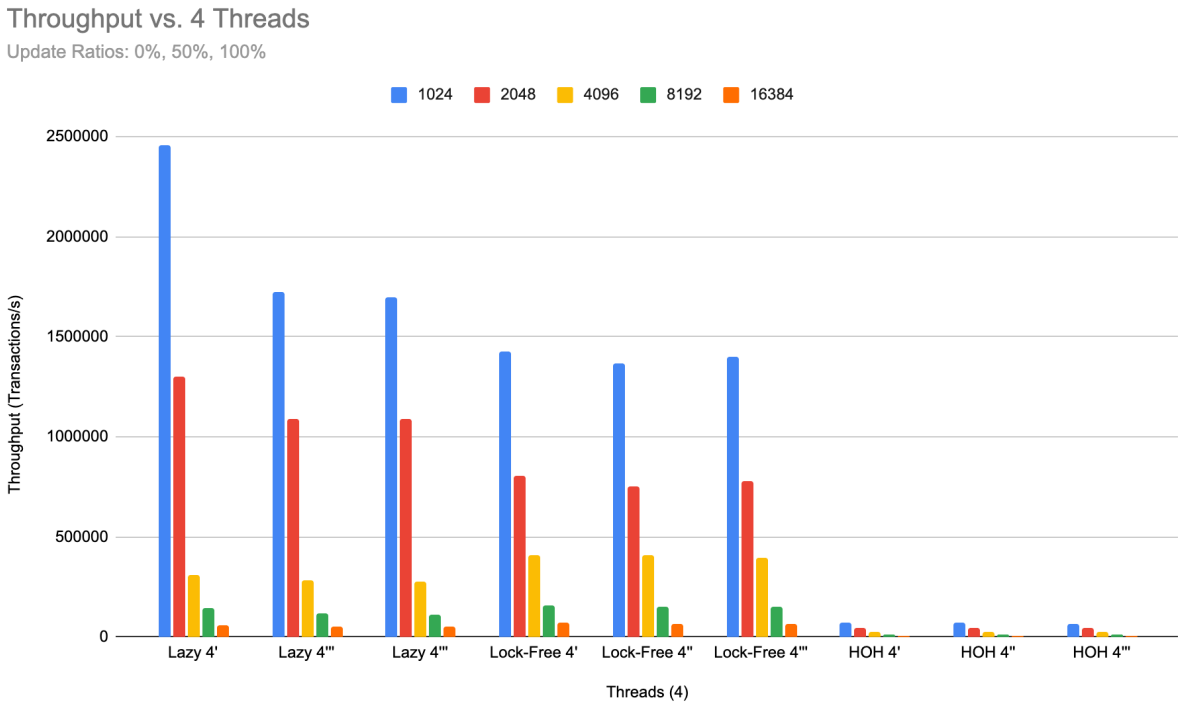

Figure 6: Lazy, Lock, HOH Lists Compared for 4 Threads of Execution

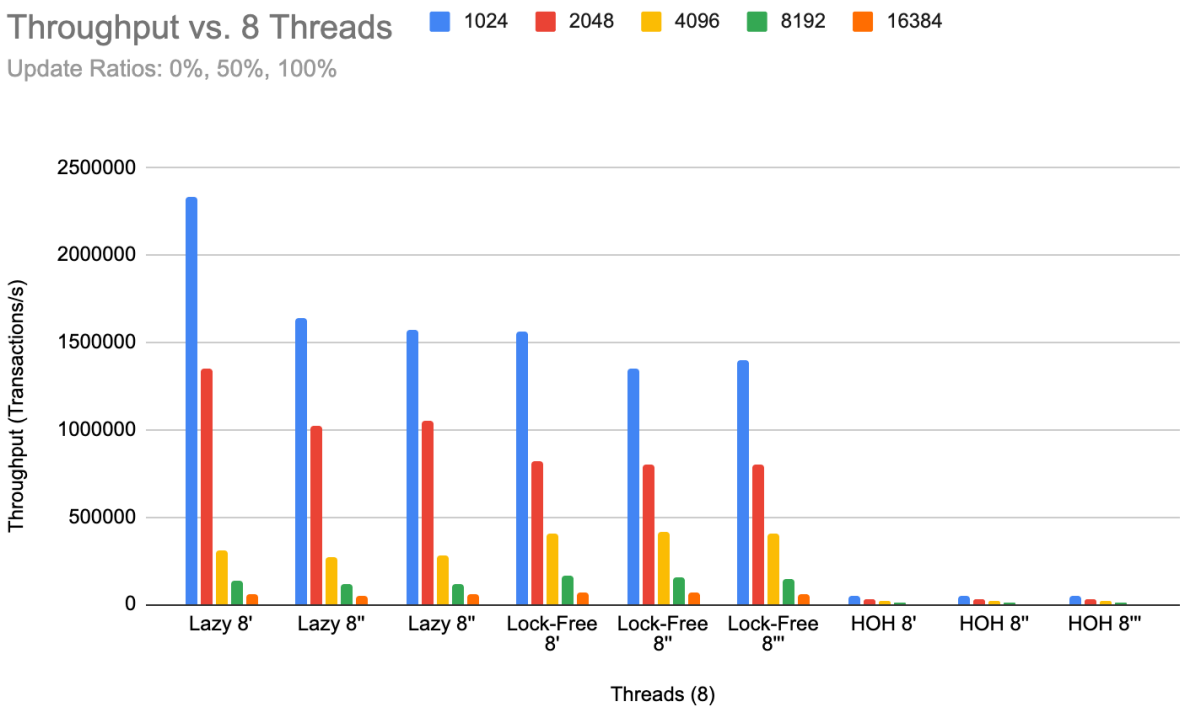

Figure 7: Lazy, Lock, HOH Lists Compared for 8 Threads of Execution

Other Set Implementations: Mutex Skiplist, ESTM Hashtable, and Mutex RCU Tree

I Compared 3 other set implementations (Mutex Skiplist, ESTM Hashtable, and Mutex RCU Tree) with logarithmic O(log n) time complexity. Throughput (transactions/second) diminished as both the size of the data structure (i.e. the number of keys) and the update ratio increases, but increased as the number of threads of execution increased. The parameters used for benchmarking these data structures includes: -i (initial size), -t (# of threads), -r (range), -u (update ratio), -A (alternate).

Skiplist uses multiple “layers” of lists built upon the original linked list, where every additional layer contains fewer elements allowing for O(log n) time complexity. ESTM Hashtables are neither wait-free or even lock-free, but rather use locks internally. RCU Trees uses read-copy-update synchronization primitives to avoid the use of locks, and act as lightweight atomic operations.

.

(See Next Page for Plots)

Threads	1K	100K	1M	Update Rate	Data Structure
Threads (1)	3964276.4	1343573.4	247844.4	0%	Mutex Skiplist
Threads (2)	7459289.0	2446787.3	658724.1		
Threads (4)	14073579.3	4417944.4	1301599.0		
Threads (8)	13709564.2	4341767.1	1313367.6		
Threads (1)	3614793.3	1341150.8	538718.1	50%	Mutex Skiplist
Threads (2)	6117145.7	2280476.1	1040387.0		
Threads (4)	11,389,633.4	4234384.9	1881436.8		
Threads (8)	11454946.1	4,288,634.86	1903842.1		
Threads (1)	3593525.1	1056159.9	493853.4	100%	Mutex Skiplist
Threads (2)	3634055.3	1929667.0	953233.6		
Threads (4)	9910734.5	2499861.7	1720055.0		
Threads (8)	12324618.88	2566193.1	1593942.5		

Threads	1K	100K	1M	Update Rate	Data Structure
Threads (1)	2199063.3	1918659.97	1606952.1	0%	ESTM Hashtable
Threads (2)	4036652.9	3662483.7	2853079.9		
Threads (4)	7574551.2	6641685.0	5533562.5		
Threads (8)	7807247.7	7052250.575	5637915.9		
Threads (1)	1641514.8	1411684.3	1198043.0	50%	ESTM Hashtable
Threads (2)	2965811.7	2724115.5	2271456.4		
Threads (4)	5,044,200.1	4720520.5	4107959.1		
Threads (8)	3410278.0	3,781,327.40	3407828.3		
Threads (1)	1344370.9	1190412.9	983391.8	100%	ESTM Hashtable
Threads (2)	2347932.4	2145946.3	1859543.8		
Threads (4)	3325733.5	3193680.3	3061394.4		
Threads (8)	2142288	2354914.0	2177093.1		

Threads	1K	100K	1M	Update Rate	Data Structure
Threads (1)	6180850.0	3836033.983	1569611.3	0%	Mutex RCU Tree
Threads (2)	10548836.0	6540851.8	2867033.0		
Threads (4)	20501567.9	11972945.3	5810248.7		
Threads (8)	22276316.4	13099877.5	5753492.9		
Threads (1)	3463929.0	2328091.0	1221707.8	50%	Mutex RCU Tree
Threads (2)	6221962.2	4347886.0	2320923.5		
Threads (4)	11,245,269.4	8116407.6	4139570.8		
Threads (8)	10130851.3	7,739,365.60	4233306.1		
Threads (1)	2620041.9	1907926.7	1038649.9	100%	Mutex RCU Tree
Threads (2)	4638217.9	3397547.2	1971201.7		
Threads (4)	7151074.8	6375922.7	3724557.1		
Threads (8)	7202273.318	5945817.5	3660833.4		

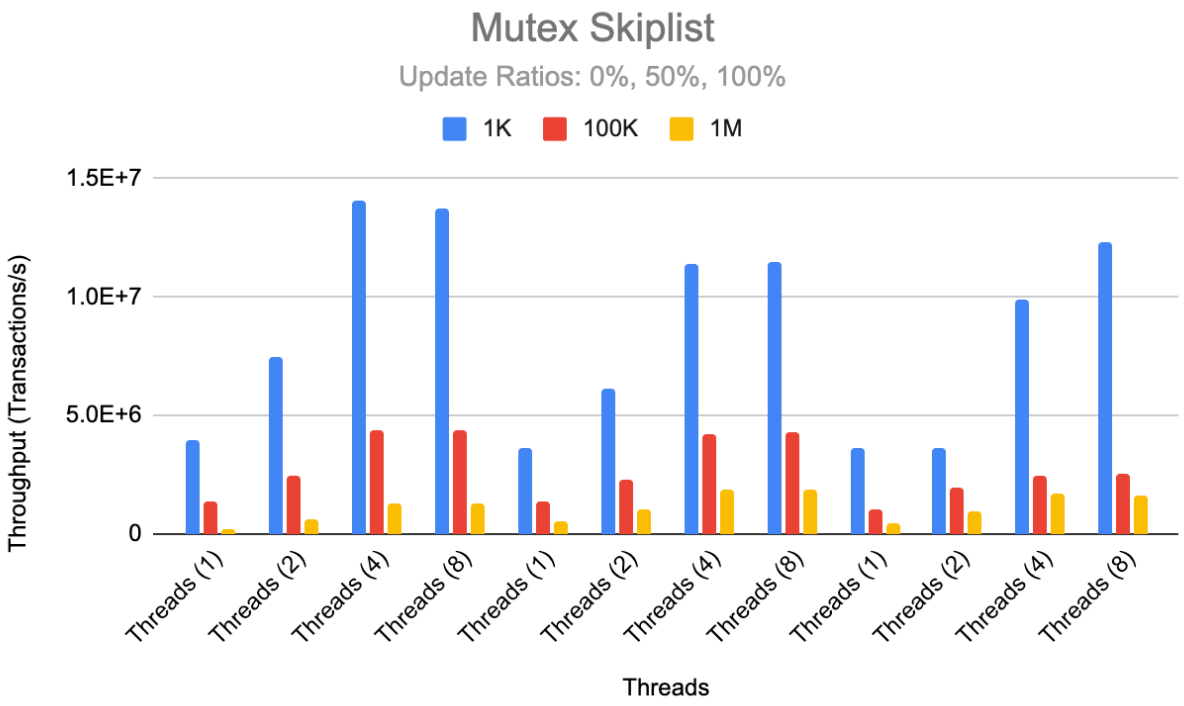

Figure 8: Mutex Skiplist for Different Threads and Update Ratios

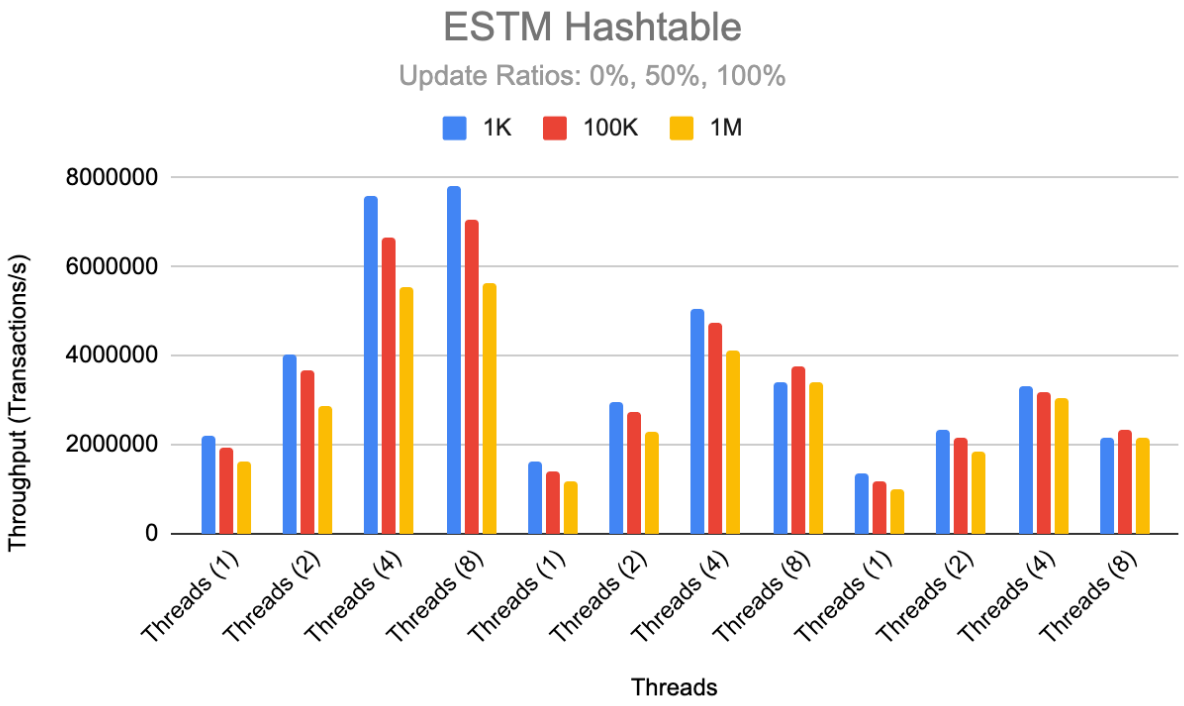

Figure 9: ESTM Hashtable for Different Threads and Update Ratios

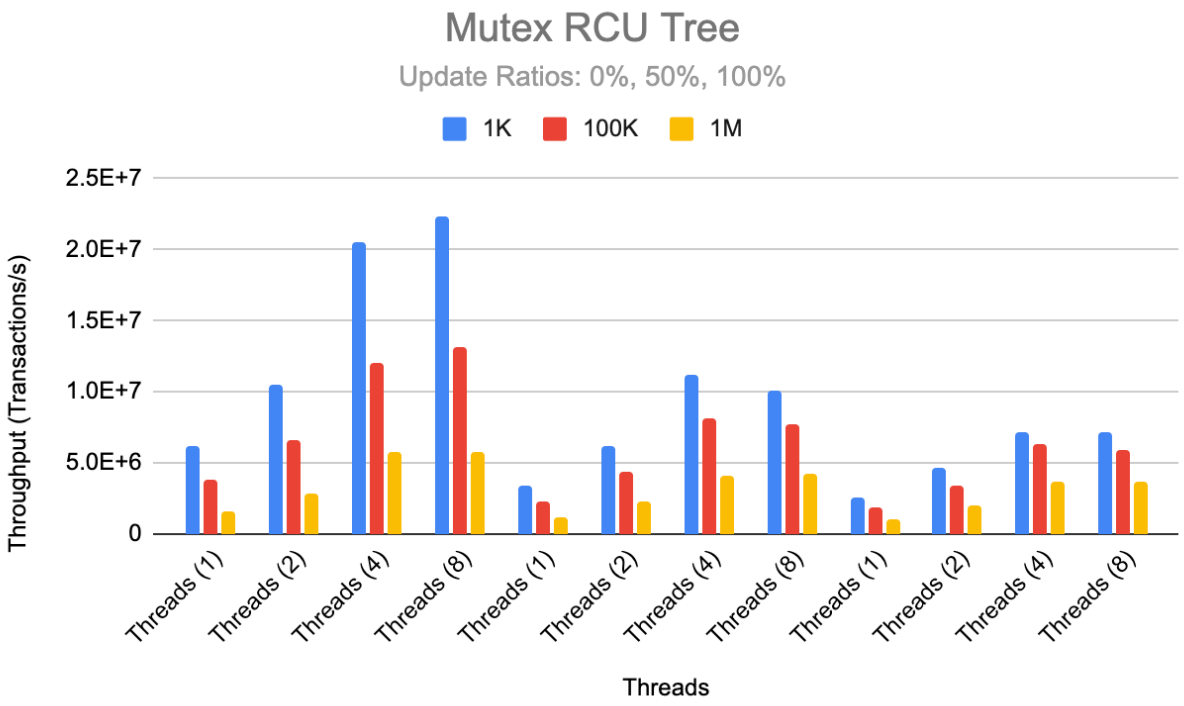

Figure 10: Mutex RCU Tree for Different Threads and Update Ratios

The ranges were changed to 100K and 1M, as a million+ elements is a typical workload for logarithmic data structures.In regards to the performance results, Mutex RCU Tree > Mutex Skiplist > ESTM Hashtable. Off the record, I compared the Mutex RCU Tree (in addition to mutex skiplist and ESTM hashtable) to a lock-free skiplist, mutex hashtable, and harriss-lock free hashtable...and the RCU Tree data structure outperformed all of those implementations. It utilizes Read-Copy-Update under the hood, which is a synchronization mechanism that avoids the overhead of locks. Initial testing also indicated that MUTEX hashtables and SPIN hashtables performed better than ESTM hashtables (which uses locks), but they would seg fault in sizes of 100K+ for some reason….so I had to switch data structures to ESTM hashtables.

When comparing the data structures to each other as a function of threads of execution, the results are clearer below.

For single threaded workloads, MUTEX RCU Trees dominate for smaller workloads 1K and 100K, but the throughput performance starts to degrade to mimic the SkipList and ESTM Hashtable around 1M elements. Interestingly, as the workloads (and number of threads increased), RCU Trees performed WORSE with respect to throughput compared to the skiplist, but still better than the ESTM hashtable (since the hash table uses heavy locks in it's implementation). The RCU Tree data structure seems to be extremely efficient for smaller workloads under 10K keys, but starts to scale poorly as they approach the 100K-1M range.

Skiplists scale much better in regards to their throughput performance as the number of threads increases.

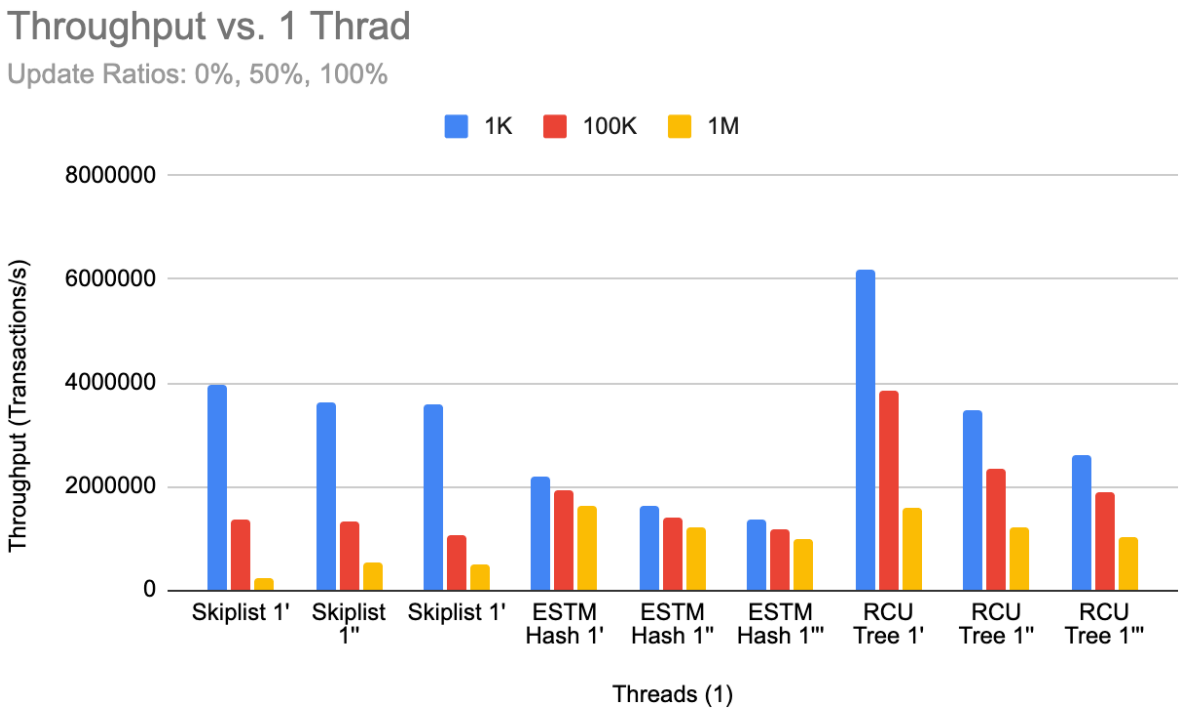

Figure 11: Mutex Skiplist, ESTM Hashtable, and Mutex RCU Tree Compared for 1 Thread of Execution

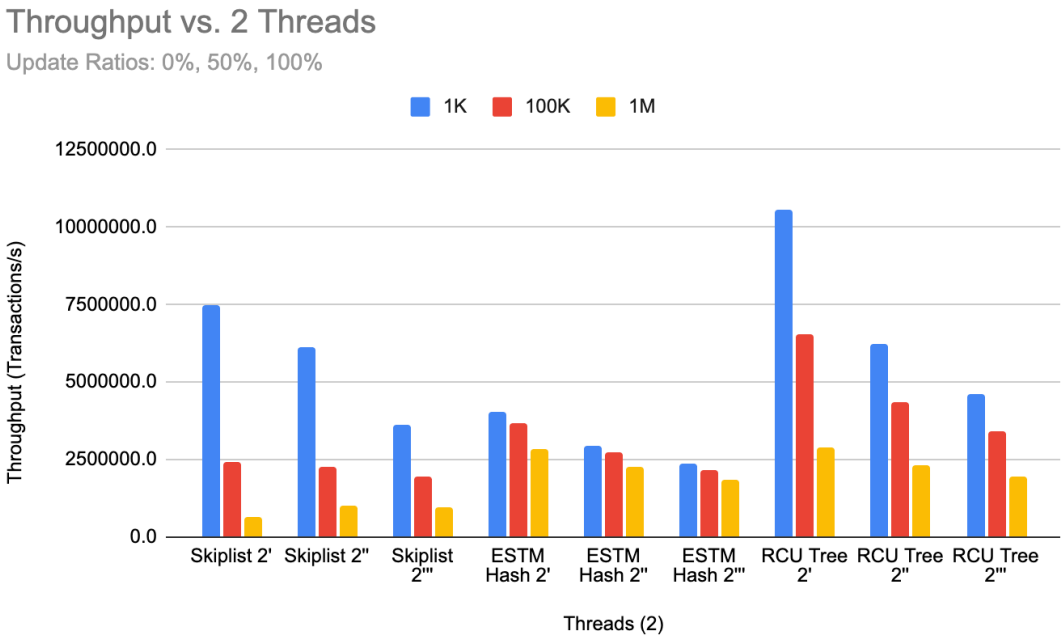

Figure 12: Mutex Skiplist, ESTM Hashtable, and Mutex RCU Tree Compared for 2 Threads of Execution

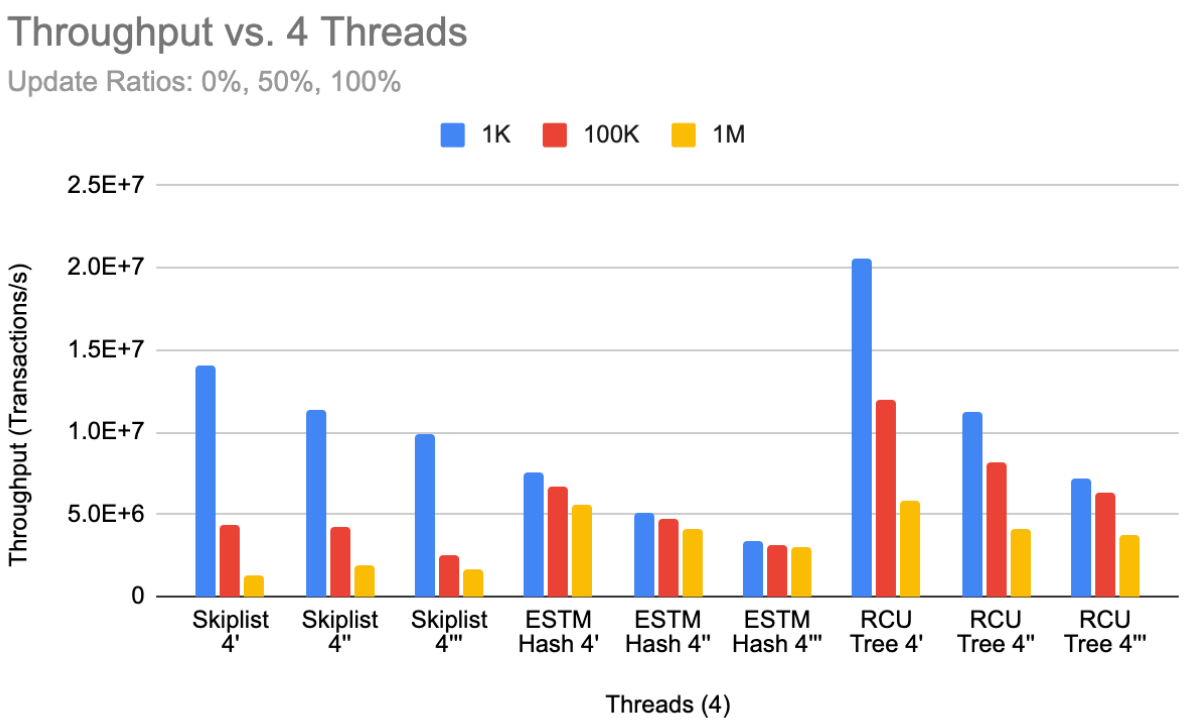

Figure 13: Mutex Skiplist, ESTM Hashtable, and Mutex RCU Tree Compared for 4 Threads of Execution

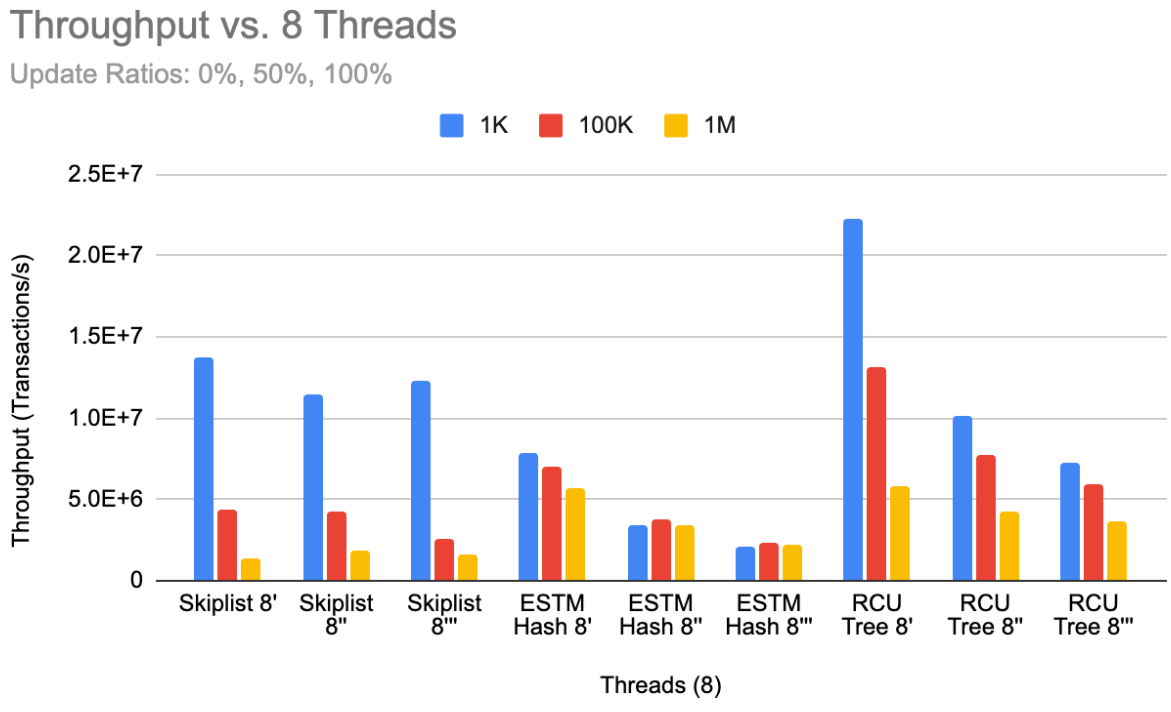

Figure 14: Mutex Skiplist, ESTM Hashtable, and Mutex RCU Tree Compared for 8 Threads of Execution