Task 2: Fine-grained data structures

Design Considerations:

Thread Synchronization & Reentrant Lock:
Each Node class in the FineGrainedTree and the FineGrainedList have an Reentrant Lock as a private variable for threads to lock it when accessed. Reentrant locks are used because it allows threads to hold multiple locks at any given time, meaning that it can lock several nodes simultaneously. This feature makes the lock coupling protocol possible, where a thread can only acquire a lock for a node when it's holding the lock for its predecessor.

The hand-over-hand procedure for acquiring locks is necessary because if threads only held one lock for the current node considered, concurrent operation calls will not work as intended and mess up the data structure. For example, concurrent remove calls on a binary tree could fail in the following scenario. Thread B is about to remove **node b**, where **node a** is its parent and **leaf node c** is its child and is supposed to replace **node b**. Simultaneously, thread C is about to remove **leaf node c**, where **node b** is its parent. Consequently, only **node b** will be removed because thread B sets c.parent to **node a** and a.left to **node c**. (Thread C will set b.right to null, but that is trivial because Thread B has put **node b** out of the picture). Similar logic applies to concurrent remove calls on linked lists [2]. To avoid this, both threads need to hold locks on both the parent and the child node, since threads try to acquire conflicting locks causing them to wait for the other to finish, as shown in figure 1. All this is made possible through Reentrant locks allowing threads to hold multiple locks.

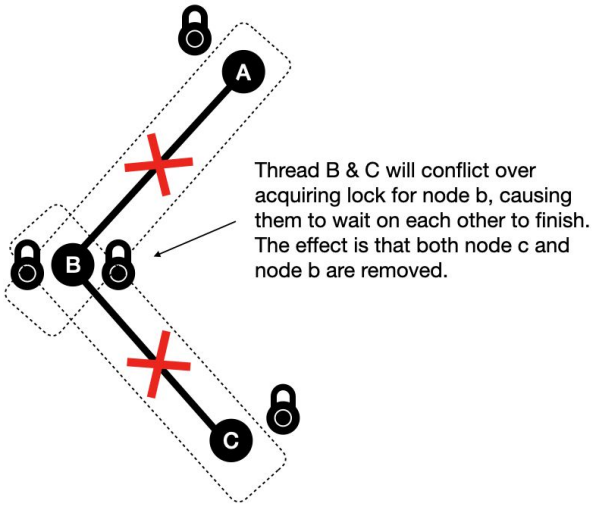


Figure 1

Linked List:
The linked list uses two sentinel nodes, as seen in figure 9.6 & 9.7 of [2]. The first dummy node is at the beginning of the list and the second dummy node is at the end, where both nodes hold null as data. The purpose of this is to avoid the edge cases, like adding or removing a node of an empty list, and thereby keep the code clean and comprehensive. To avoid comparing null to an integer or string, the Node class has a boolean variable called 'isDummy' set to true for dummy nodes and causes the while loop to exit when it reaches the end (the dummy tail).

Binary Tree:
The binary tree does not use any dummy nodes, since edge cases are less common. Instead, it has a pointer to the root of the tree that stays consistent throughout program execution. The tree adds elements as new leaves to the tree, and finds the appropriate leaf position by iterating along the height of the tree (starting from the root). Finding an element to remove is also done

iteratively. Alternatively, nodes could have also been found recursively but this approach is susceptible to deadlocks with fine-grained locking. To make sure to find the right path down the tree, removing an element causes the tree to rebalance itself by finding the node's successor and replacing it with the node to be removed.

Hypotheses:

Hypothesis 1: Binary Tree performs better than the Linked List.
Add and remove operations on a linked list take way longer than they do on the binary tree, because it takes more loop iterations to find the element to remove or place to insert an element. For example, adding the largest element to the list requires traversing the entire list, starting from the head, as elements are stored in a non-decreasing order. This is why the running time of the linked list is $\Theta(n)$ (where n is the number of elements stored in the list). In comparison, the binary tree has a running time of $\Theta(h)$ (where h is the height of the tree). This is because operations on the binary tree are only required to traverse the height of the tree. Note that the height of the binary tree varies depending on the order in which the elements are added. For example, if they are added in a non-decreasing order, the height of the tree h will equal to the number of elements n stored in the tree. Thus, operations will take n time (as bad as the linked list). However, according to *Theorem 12.4* the 'expected height of a randomly built binary tree on n nodes, is $\Theta(\lg n)$' [1].

Hypothesis 2: Increasing the number of threads operating on the fine-grained locked data structures increases performance.
Fine-grained locking only requires threads to usually lock 2 nodes, as discussed above, meaning that multiple threads can access the data structure concurrently. Thus, the chance of threads waiting for each other is low and more of the execution can be performed in parallel. According to Admahl's law, this will lead to a speedup proportional to the amount of how much of the execution is performed in parallel. The ratio between sequential execution and parallel execution is shown by the below formula:

$$Speedup = \frac{1}{(1-p) + p/N}$$

This means that increasing the number of threads operating on the data structure will cause an increase in the data structures performance, since more of the operations are done concurrently.

Hypothesis 3: Fine-grained locking performs better than coarse-grained locking with multiple threads.
The difference between coarse-grained locking and fine-grained locking is that coarse-grained locking locks the entire data structure while fine-grained locking only locks individual nodes. As a result, none of the operations on a coarse-grained locked data structure can be done concurrently. Thus, there is no speedup and the contention between the threads to hold the lock leads to a decrease in performance. In comparison, fine-grained locking results in an increase in performance when multiple threads operate on the data structure as parallel execution increases.

Hypothesis 4: Increasing the workTime parameter will give a relative speedup in performance.
The workTime parameter when running the scripts lets each thread do some additional 'work' in between remove and add calls. Increasing the parameter could lead to an increase in performance because less threads will be operating on the data structure, meaning that there will be less contention between the threads for acquiring locks. The effects of this will be seen on data structures not storing many elements, because the less nodes there are the more contention for them in fine-grained locking. Letting threads 'sit out' and not populate the data structure should therefore consequently lead to an increase in performance.

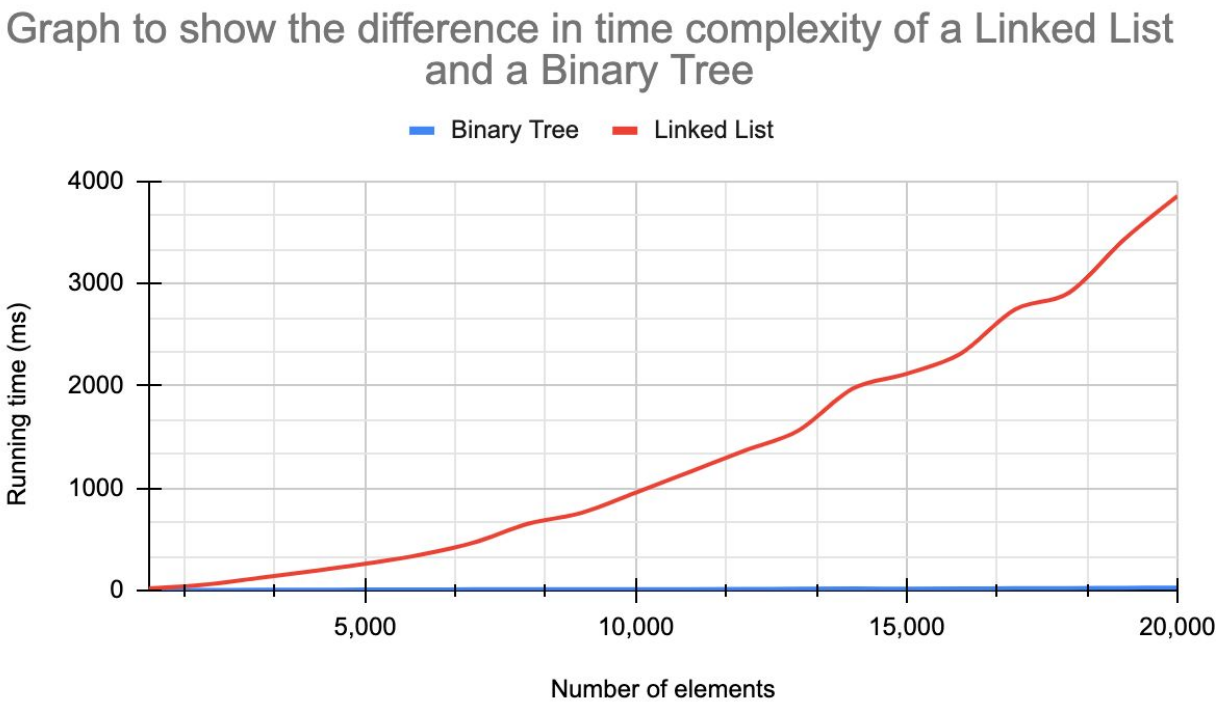
Evaluation:

Experimental setup:
To test the above hypothesis, the running time of the two fine-grained data structures is measured on a DAS5 node. This node runs programs on the Intel Haswell E5-263-v3, which has 8 cores and supports 16 threads [3]. Experiments will therefore only test the data structure with 1, 4, 8 and 16 threads. The input size n increases by intervals of 1,000 starting at 1,000 and ending at 20,000 elements. Each recorded running time is the average of 10 runs with the given input size. This should provide enough reliable data to determine the hypothesis.

Variables for testing hypothesis 1-3:
Independent variable: Number of elements & number of threads.
Dependent Variable: Time (ms).
Controlled Variables: Operations are done with integers, WorkTime is 0.

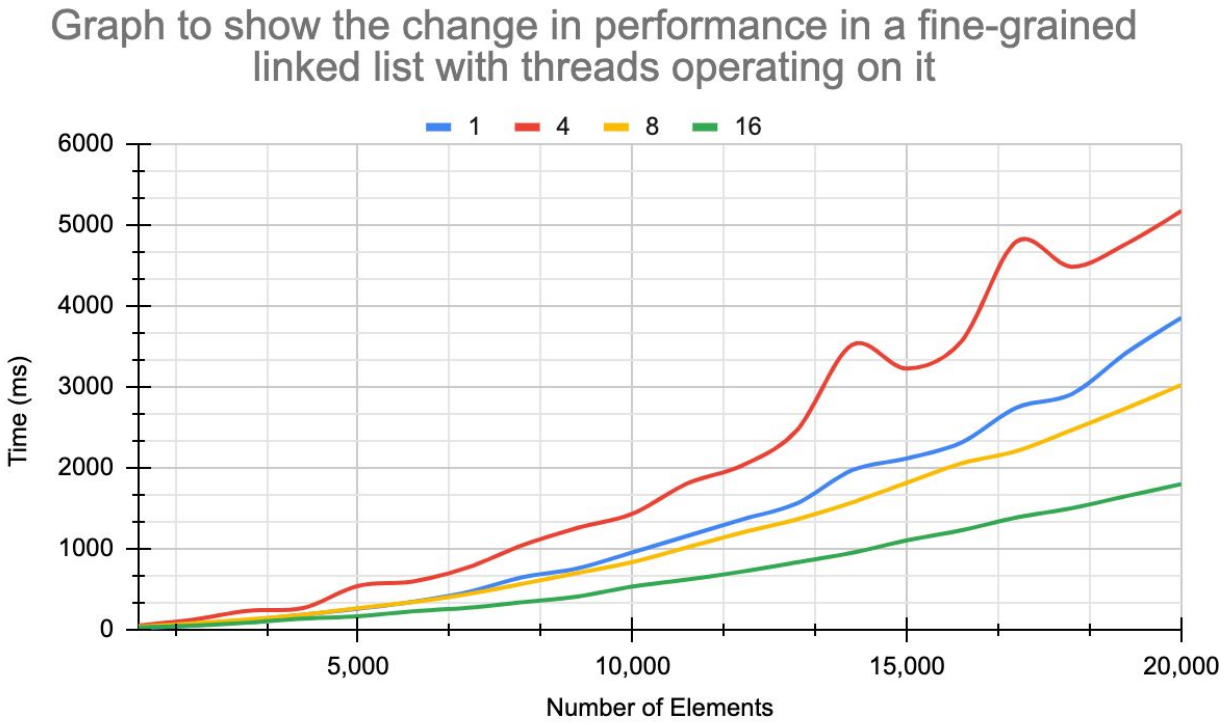
Variables for testing hypothesis 4:
Independent variable: Number of elements & workTime.
Dependent Variable: Time (ms).
Controlled Variables: Operations are done with integers, number of threads is 16.

Testing Hypothesis 1: Binary Tree performs better than the Linked List.



The above graph shows that the time complexity of the binary tree is way lower than that of the linked list, thereby confirming the given hypothesis. The running time of the linked list increased from 20.5 ms to 956.29 ms all the way up to 3,854.6 ms when the number of elements stored increased from 1,000 to 10,000 up to 20,000 elements, respectively. In comparison, the running time of the binary tree only marginally increased from 3.2 ms to 12 ms to 26.9 ms for the same respective number of elements. Note that the graph does not seem to show an increase of the running time of the binary tree because the scale on the vertical axis is so large from the linked list. This is because operations on a binary tree only take $\Theta(h)$ time (where h is the height of the tree) while operations on a linked list take $\Theta(n)$ time (where n is the number of elements in the list). Since the list only has one entry point, namely the head of the list, removing the largest element requires traversing the entire list. In comparison, removing the largest element in the binary tree only requires traversing the height of the tree because the largest element is the rightmost leaf node in the tree. To conclude, the experiment has shown that the binary tree performs far better than the linked list.

Testing Hypothesis 2: Increasing the number of threads operating on the fine-grained locked data structures increases performance.



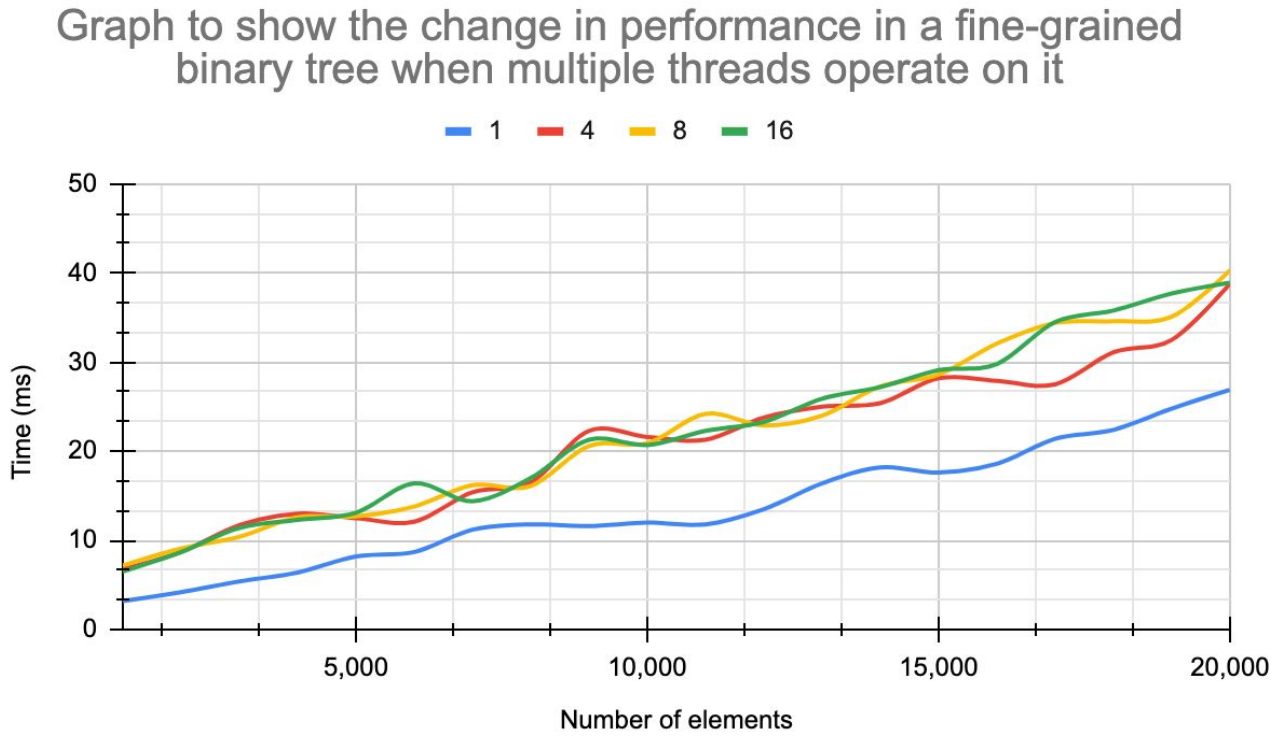
The experiment shows that running multiple threads on a fine-grained linked list causes its running time to decrease, as was hypothesised. The running time of a single thread performing add/remove operations on a linked list containing 15,000 elements was 2117.39 ms, which decreased to 1817.3 ms when adding 8 threads. With 16 threads, the running time decreased further down to 1105.19 ms. So adding threads decreases the running time of add/remove operations. This is because in sequential execution (i.e. single thread), all of the operations are done one after the other. In comparison, more of the operations can be performed in parallel when multiple threads can access the same data structure (as long as the threads do not access the same node). This is in line with Amdahl's law, which states that the speedup of multiple processors depend on how much of the execution can be performed in parallel. Thus there is greater speedup in a fine-grained locked list when the number of threads operating on the data structure increases.

Using Amdahl's law and the running time for 1, 8 and 16 threads on the list containing 15,000 elements, the amount of speedup can be calculated. The running time of a single thread is the sequential time (since operations are performed one after the other), which is 2117.39 ms according to the experiment. Then the speedup caused by 8 threads was (2117.39ms / 1817.3ms =) 1.16 and the speedup of 16 threads was (2117.39ms / 1105.19ms =) 1.91.

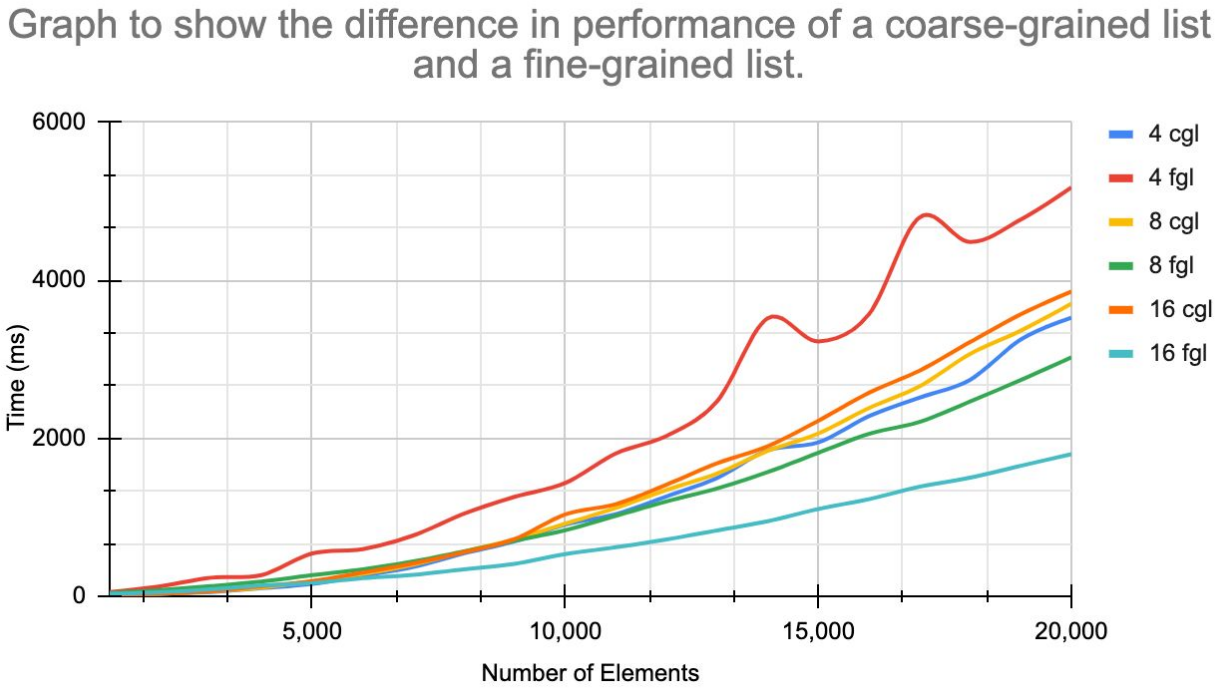
However, the graph also shows that having 4 threads performing parallel operations takes longer than having sequential execution. The running time having 1 thread operating on a linked

list containing 15,000 elements was 2117.39 ms, whereas the running time for 4 threads was 3227.39 ms. A decrease in performance of (2117.39ms / 3227.39ms =) 0.65. This seems odd since more of the operation can be performed in parallel. It may be that the contention between the different threads to acquire locks on individual nodes outways the performance benefit of having more of the execution in parallel. Arguably, there may be more contention between threads when there are more of them, however when there are more threads more of the operations can be performed in parallel which in turn boosts performance.

Similarly, concurrent execution on a binary tree performs worse than sequential execution. As is seen in the below graph, the number of threads all produce similar running times, meaning that the overhead of every thread locking every node accessed while traversing the tree causes a performance decrease. To conclude, concurrent execution made possible through fine-grained locked data structures only improves performance when the overhead of locking and unlocking nodes remains relatively small.



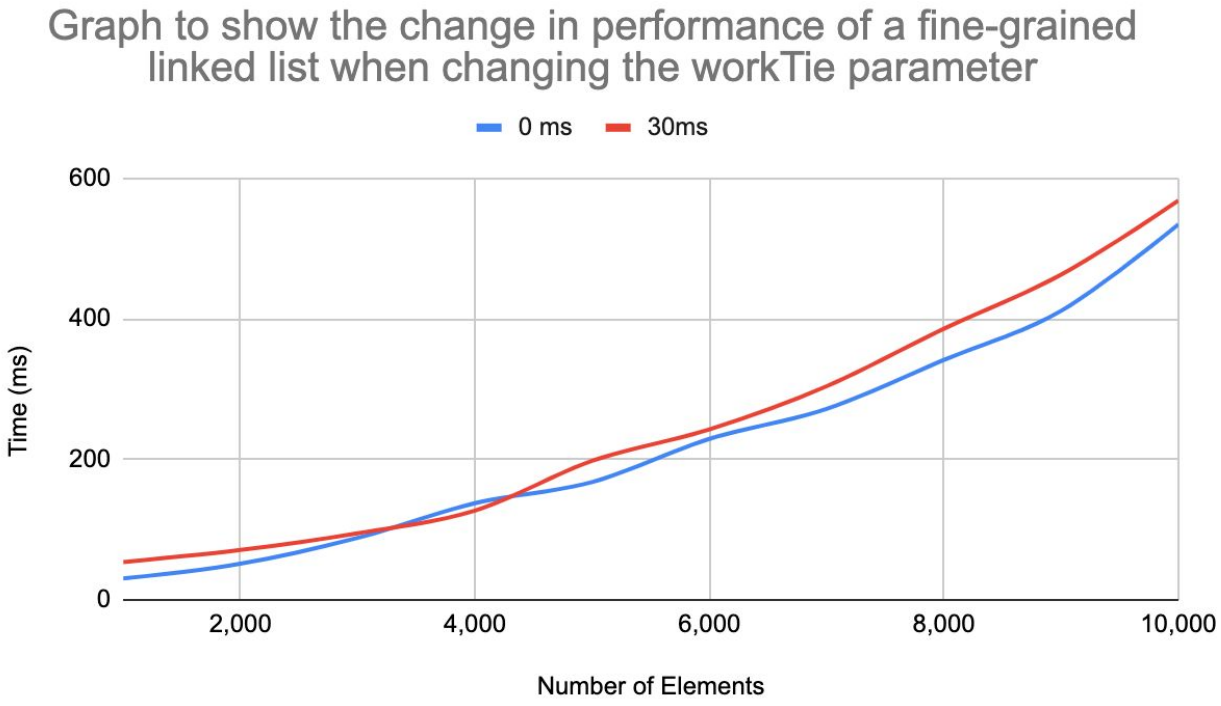
Testing Hypothesis 3: Fine-grained locking performs better than coarse-grained locking with multiple threads.



The experiment highlights that fine-grained locking performs better than coarse-grained locking on high levels of concurrency, whereas coarse-grained locking performs better on low levels of concurrency. For example, the running time of add/remove operations on a coarse-grained list containing 14,000 elements is 1,900 ms when 16 threads operate on it, whereas the running time of a fine-grained list is only 951 ms. This is because coarse-grained locking considers the entire list as the critical section, while fine-grained locks only consider individual nodes as the critical section. Thus more of the operations can be performed in parallel, resulting in a speedup in performance as discussed earlier. In comparison, threads do not operate concurrently in coarse-grained locking as each thread accesses the list sequentially.

However, the mechanism of locking the entire list rather than individual nodes works better when the levels of concurrency is low. 4 threads take 1849.4 ms to perform operations on a coarse-grained list of 14,000 elements, whereas they take 3520 ms on a fine-grained list. This is because locking each individual node when it is accessed produces an overhead that outways the performance benefit of performing operations concurrently- threads need to lock and unlock n nodes in a fine-grained locked list if it traverses to the end, while threads only lock and unlock once in coarse-grained locking. Thus coarse-grained locking is the better locking mechanism for low levels of concurrency.

Testing Hypothesis 4: Increasing the workTime parameter will give a relative speedup in performance.



The above graph shows the running time of a varying workTime parameter for the fine-grained list containing a maximum of 10,000 elements with 16 threads operating on them. The maximum number of threads was chosen for this experiment because it should lead to the most contention between the threads to hold the locks. Interestingly enough, both workTime parameters showed similar performance when the number of elements contained in the list was small. For example, when the list contains 5,000 elements, the time complexity of operations when the workTime parameter was set was 169 ms, while when it was not set, the running time was 167.6ms. This is because when the input size is small, it is more likely for the threads to access the same node and therefore wait. This is avoided by having less threads operating in the list simultaneously, which is achieved by setting the workTime parameter. However, as the number of elements grows in the list, the performance of the workTime parameter set decreases, since it is less likely for the threads to content for nodes when there are more of them. Having the workTime parameter set decreases the amount of concurrent execution happening on the list, meaning that performance decreases. Thus the performance was false.

References:

[1] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein Clifford. *Introduction to Algorithms*. Cambridge, Mass: MIT Press, 2001. Print.
[2] Herlihy, Maurice and Shavit, Nir. *The Art of Multiprocessor Programming.* Waltham, MA, 2012
[3]https://www.cs.vu.nl/das5/special.shtml#:~:text=Accelerators%20and%20special%20compute%20nodes,types%20for%20specific%20research%20purposes.

Appendix:

	Binary Tree Fine Grained			
Number of elements	Thread 1	Thread 4	Thread 8	Thread 16
1,000	3.2	6.8	7.2	6.5
2,000	4.2	8.7	9.1	8.7
3,000	5.4	11.7	10.4	11.4
4,000	6.4	13	12.6	12.3
5,000	8.2	12.5	12.7	13.1
6,000	8.7	12.1	13.8	16.4
7,000	11.2	15.4	16.2	14.4
8,000	11.8	16.5	16.1	17
9,000	11.6	22.29	20.6	21.29
10,000	12	21.6	20.9	20.7
11,000	11.8	21.29	24.2	22.29
12,000	13.5	23.79	22.9	23.29
13,000	16.4	25	24	25.9
14,000	18.2	25.4	27.29	27.2
15,000	17.6	28.2	28.6	29.1
16,000	18.6	27.9	32.09	29.79
17,000	21.4	27.5	34.4	34.5
18,000	22.4	31.1	34.59	35.79
19,000	24.79	32.5	35.09	37.7
20,000	26.9	38.79	40.29	38.9

	Linked List Fine Grained			
Number of elements	Thread 1	Thread 4	Thread 8	Thread 16
1,000	20.5	52.5	39.79	30.4
2,000	54.09	126.3	78.8	51.4
3,000	119.4	235.3	128.5	88.19
4,000	187.00	268.29	187.3	137.6
5,000	260.10	542.4	267.79	167.6
6,000	346.00	597.4	342.39	229.1
7,000	466.20	770.20	441	271.7
8,000	650.50	1,046.40	569	341.39

9,000	759.00	1,259.40	702.4	411.29
10,000	956.29	1,433.69	836.7	534.29
11,000	1,160.00	1,808.69	1,019.29	620.59
12,000	1,364.80	2,030.90	1,201.69	719.4
13,000	1,562.50	2,467.19	1,365.30	834
14,000	1,972	3,520.00	1,573	951
15,000	2,117.39	3,227.39	1,817.30	1,105.19
16,000	2,315	3,569.39	2,057.39	1,229.90
17,000	2,746.19	4,798.20	2,208.10	1,387.69
18,000	2,910.30	4,485.39	2,466.50	1,502
19,000	3,422.89	4,770.00	2,738.60	1,652
20,000	3,854.60	5,175.39	3,025.89	1,799.40

	Linked List Coarse Grained		
Number of elements	4	8	16
1,000	10.50	12.5	11.2
2,000	30.79	33	32.59
3,000	54.80	62.40	65
4,000	103.09	111.50	126.9
5,000	156.89	191.19	191.39
6,000	261.79	283.50	300.6
7,000	371.60	400.79	416.20
8,000	543.90	566.40	558.00
9,000	695.29	722.70	724.09
10,000	908.29	919	1,036.19
11,000	1,039.90	1,118.09	1,167.30
12,000	1,264.90	1,346.30	1,413.30
13,000	1,497.80	1,554.09	1,684.30
14,000	1,849.40	1,840.69	1,900
15,000	1,948.30	2,061.60	2,223
16,000	2,280.10	2,383.69	2,577.39
17,000	2,515.60	2,658.50	2,857.19
18,000	2,741	3,069.80	3,219.60

19,000	3,252.10	3,363.19	3,570.39
20,000	3,529.00	3,705.00	3,860.00

	Fine-grained linked, 16 threads		
Number of Elements	0 ms	15ms	30ms
1,000	30.4	48.299	53.9
2,000	51.4	76.4	71
3,000	88.19	105	94.8
4,000	137.6	134.3	127
5,000	167.6	169	197.89
6,000	229.1	234.19	242.6
7,000	271.7	300.1	304
8,000	341.39	384.89	385.79
9,000	411.29	438.89	462.79
10,000	534.29	534.4	568.09
11,000	620.59	621.9	643.59
12,000	719.4	746.7	766.09
13,000	834	853.29	870.2
14,000	951	973.79	1026.09
15,000	1,105.19	1117.8	1107.9
16,000	1,229.90	1229.8	1282.8
17,000	1,387.69	1366.3	1411.3
18,000	1,502	1510.9	1,567.80
19,000	1,652	1638.09	1665.5
20,000	1,799.40	1829	1861.09