Concurrency and Multithreading Assignment 1 - Report

Mehmet Cetin - 2644886 - mcn283
Ghasim Gholi - 2553232 - gmi290

1. Design Considerations

- Our linked list must be accessible by every thread from the main memory. Hence, our Node head variable is declared volatile.
- The critical sections for the linked list is the add(T t) and remove(T t) methods. Thus, we lock the method when a thread accesses the method. Also, we synchronize the add and remove methods to prevent thread interference and provide mutual exclusion among the threads.
- The lock we use in the program is a reentrant lock. We use a reentrant lock because it allows the lock to reacquire itself and this prevents a deadlock situation.

We have observed that, when the number of ms worked between the add/remove methods, increasing the number of threads doesn't speed up the execution for the CGT and CGL. Although, increasing the number of ms worked results in different speed up rates for the CGL and CGT. Hence, we came up with the hypothesis in the next section.

2. Hypothesis: Multiple threaded runs will be faster than single threaded runs when the number of seconds worked, besides calling the add() or remove() methods, is not zero.

3. Evaluation

RUNS	nrTreads, nrItems, nrMillSecondsWorked	Coarse-grained List average run time(ms)	Coarse-Grained Tree average run time(ms)
1	1, 1000, 0	8.6	2.2
2	2, 1000, 0	12.2	5.9
3	5, 1000, 0	14.9	8.3

4	1, 10000, 0	460.39	8.5
5	2, 10000, 0	815.70	22.2
6	5, 10000, 0	901.40	29.1
7	1, 10000, 100	2454.69	2015.19
8	2, 10000, 100	1302.5	1018.29
9	5, 10000, 100	1013.0	412.39

Table 1. Table depicting the runs with different nrThreads, nrItems and nrMilliSecondsWorked.

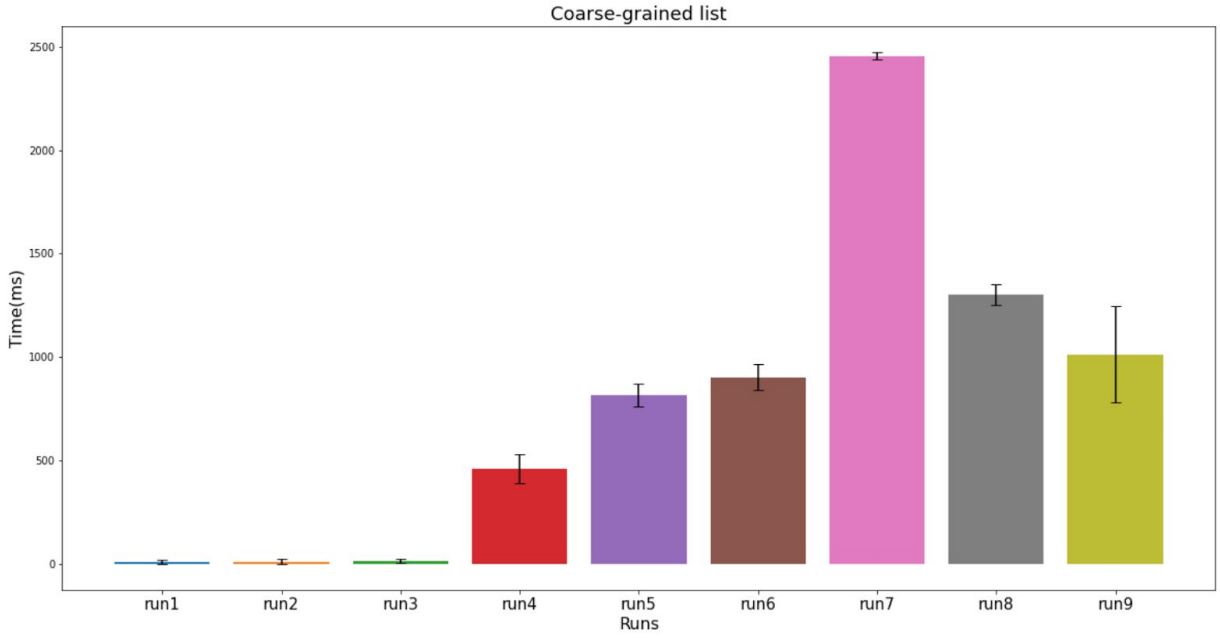

Figure 1. Bar plot with vertical error bars depicting all the runs for the coarse-grained list.

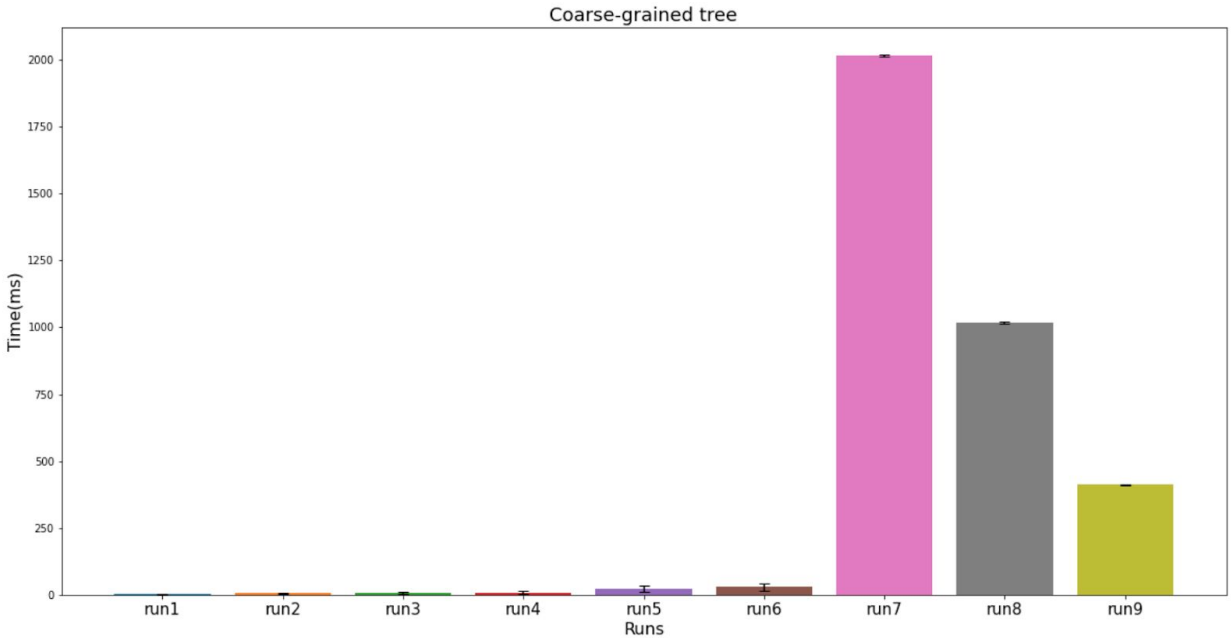

Figure 2. Bar plot with vertical error bars depicting all the runs for the coarse-grained tree.

3.1 Speedup when nrSecondsWorked is 0ms: Tree vs List

- ➢ Given the number of elements as N, the coarse-grained tree is faster in terms of executing due to the fact that the time complexity for traversing a tree is O(log(N)), while the time complexity for traversing a list is O(N).
- ➢ For instance, run 9 is 2.5 times faster in a coarse-grained tree than in list.

In Figure 1 and 2, there is no speedup in the first 6 runs. The reason is that when the number of threads increases, the program execution slows down. This is an indicator that the program is not parallelizable. The execution time increases due to the threads that attempt to acquire the lock. They reduce the performance and increase the execution time by accessing the main memory or spinning on their local cache to see if the lock is free. Hence, for coarse-grained implementation, according to Amdahl's law, the program doesn't speed up even if we increase the number of threads because the program isn't parallelizable. Given the formula for Amdahl's law below, if p is ~0, the speedup ratio is around 1, which means no speedup. That is the case for both coarse-grained list(CGL), and coarse-grained tree(CGT) in our implementation. Although, when we change the parameter for the number of seconds done work, besides adding and removing items from the list, the program speeds up when the number of threads increases.

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

For constant values in the following parameters: nrThreads and nrSecondsWorked, CGT spreads the workload more efficiently than the CGL with a higher amount of items. In contrast,

CGL has a better spread in workload efficiency than CGT in smaller amounts of items. The reason that CGT is more efficient than CGL is that for a higher amount of items, traversing a tree is much faster than linearly scanning a list. Hence, the higher the number of items is, the better the CGT performs compared to CGL.

3.2 Speedup when nrSecondsWorked is 10ms: Tree vs List

RUNS	nrTreads, nrItems, nrMilliSecondsWorked	Coarse-grained List average run time(ms)	Coarse-Grained Tree average run time(ms)
1	1, 10000, 10	684.099976	219.100006
2	2, 10000, 10	1019.200012	124.900002
3	5, 10000, 10	1359.699951	59.599998
4	1, 10000, 25	961.000000	517.400024
5	2, 10000, 25	1034.000000	271.299988
6	5, 10000, 25	1404.300049	115.199997

Table 2. Table depicting the runs with different nrThreads and nrMilliSecondsWorked.

3.3 Speedup when nrMilliSecondsWorked is 10ms and 25ms: Tree vs List

Data Structures affect the speed up ratio because of their time complexity to access a specific element. The CGT is faster in adding and removing elements than the CGL. The reason is that it takes, in average, longer to find the specified element in CGL. In table 3, we can see that increasing the number of threads doesn't speed up the execution for CGL. Our observation is that 10ms work time is too small. In CGL, while a thread is adding an item, it may take longer than 10ms or 25ms, as seen in table 3. Therefore, multiple threads won't speed up the execution of the CGL. In contrast, for CGT, multiple threads speed up the execution time because adding and removing an element takes less than 10ms, which is shorter than the CGL. This enables for multiple

threads to speed up the execution in CGT. While one thread does other work than adding/removing, another thread can call the add/remove methods.

3.4 Speedup when nrMilliSecondsWorked is 100ms: Tree vs List

When nrMilliseconds is assigned to a more significant value, like 100, for both data structures, there is a performance enhancement when the number of threads are increased. Like before, the execution speedup for CGT is superior to CGL because of the time complexity to add and remove elements. 100ms is a significant time. While one thread finishes adding/removing an item and starts working on something else for 100ms, another thread can immediately add/remove any element it wants. In this way, the effect of the 100ms bottleneck can be reduced to the minimum when more threads execute the program. Although, adding too many threads may result in reduced performance and the speedup ratio may be zero after adding a certain amount of threads.

4. Conclusion

In conclusion, when the number of threads increase, the performance and execution speed decreases for both data structures. Nevertheless, when a certain number, other than zero, is assigned to the nrMilliSecondsWorked, the execution speedup ratio differentiates for different data structures. For CGT, when a small value like 10ms is assigned to nrMilliSecondsWorked, the speed up of the execution increases when the number of threads increase. For CGL, however, that is not the case. The execution speedup increases when the number of threads increase only when a large number, for example 100ms, is assigned to the nrMilliSecondsWorked.

This doesn’t fully prove our hypothesis. The execution for CGT speeds up for all the arguments (10ms, 25ms, 100ms) assigned to the nrMilliSecondsWorked. The execution for CGL, however, doesn’t speed up unless the argument 100ms is assigned to the nrMilliSecondsWorked.