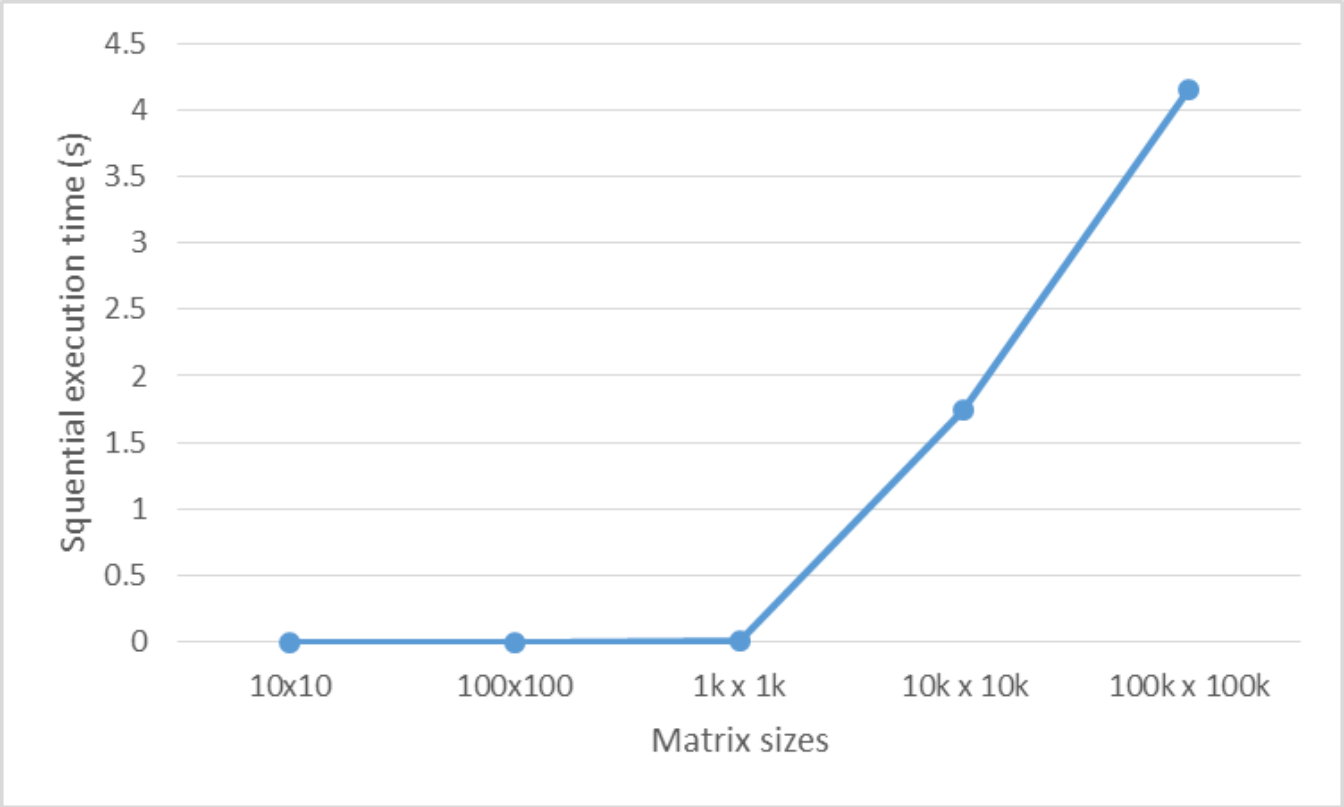
Parallel and Distributed Computing: Homework 5

Q1: A characterization of the sequential transposer execution time as the size of the matrix increases. Add plots and figures as necessary to explain your results.

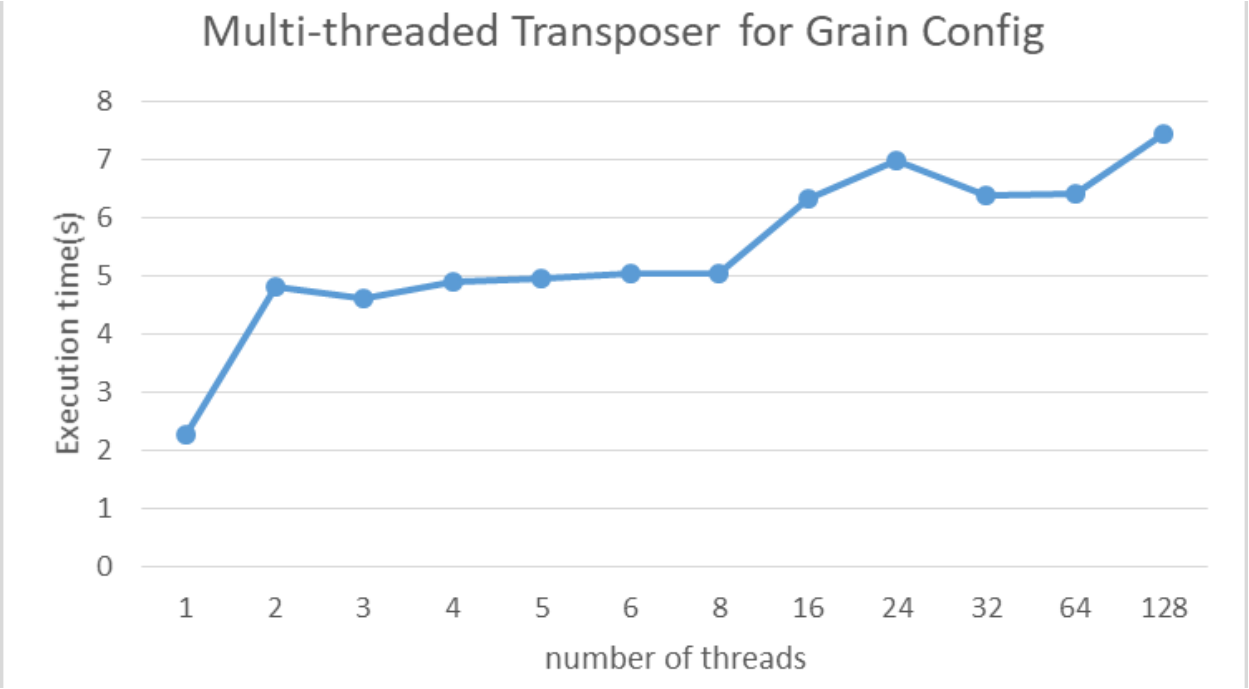


Matrix sizes	10	100	1k	10k	100k
Sequential	0.00000133	0.00018131	0.01456157	1.74303375	4.1543411

- Sub-Quadratic Growth: From matrix sizes 10x10 to 1kx1k, we see a less than quadratic growth in execution time, which is typical for matrix operations that often have quadratic complexity (O(n^2)).
- Quadratic Growth: The jump from 1kx1k to 10kx10k, and further to 100kx100k, shows a steep increase in execution time. This suggests that the execution time growth is approaching quadratic complexity as matrix size increases, which is expected since matrix transposition involves accessing each element once.
- Resource Limitation: As the matrix size grows to 100kx100k, the dramatic increase in execution time indicates that the system may be hitting resource limitations such as memory bandwidth or cache sizes, leading to less efficient memory access patterns.

Q2: A characterization of the multi threaded transposer execution time for a fine-grain configuration.

# of Threads:	1	2	3	4	5	6	8	16	24	32	64	128
Execution Time	2.266	4.812	4.625	4.911	4.952	5.033	5.033	6.321	6.985	6.384	6.406	7.442



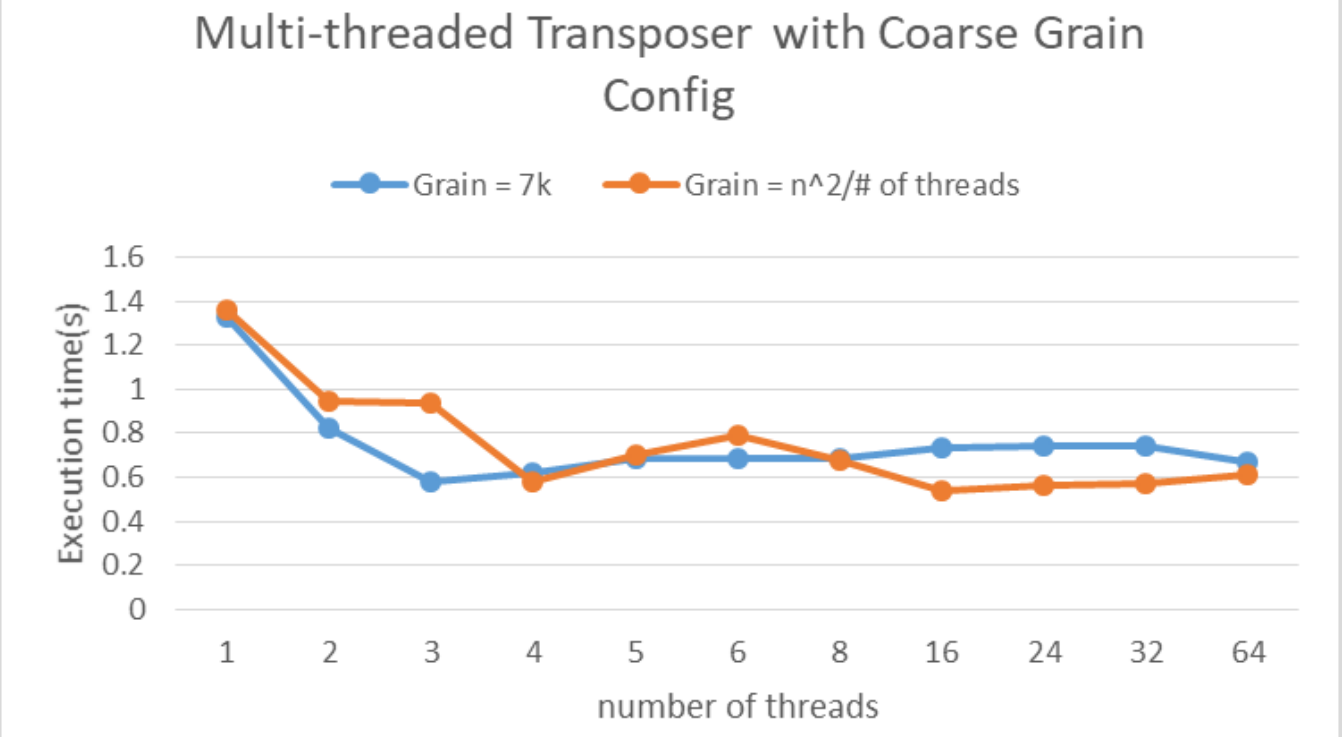
The execution times for the multi-threaded transposer indicate that the performance does not scale linearly with the number of threads:

- The execution time with a single thread is the fastest at 2.266 seconds.
- Surprisingly, the execution time more than doubles when increasing to 2 and 3 threads. This is counterintuitive as performance typically improves with more threads up to the number of physical cores.
- Beyond 3 threads, the execution time remains relatively constant with minor fluctuations, even as the number of threads increases up to 8.
- Further Increase in Execution Time: There's a general trend of increased execution time as the number of threads continues to grow beyond the number of cores, particularly noticeable at 16, 24, 32, 64, and 128 threads.

In summary, the expected performance gain from parallel execution is not observed. Instead, the execution time increases with more threads, which could be due to several factors such as overhead from managing a high number of threads, contention among threads, or inefficient use of system resources like CPU cache or memory bandwidth.

3.

# of Threads:	1	2	3	4	5	6	8	16	24	32	64	128
Grain = 7000	1.326	0.821	0.578	0.618	0.686	0.686	0.687	0.735	0.739	0.743	0.666	0.87
Grain = n^2/number of threads	1.36	0.947	0.936	0.581	0.706	0.794	0.677	0.536	0.56	0.576	0.614	0.851



Grain = 7000 (Coarse-Grained): With a single thread, the execution time is 1.326 seconds. The execution time decreases as the number of threads increases, reaching the fastest time at 3 threads with 0.578 seconds. Adding more threads up to 8 does not significantly improve performance, and with more than 8 threads, execution time increases, suggesting that the workload may not be evenly distributed or that threading overhead becomes significant.

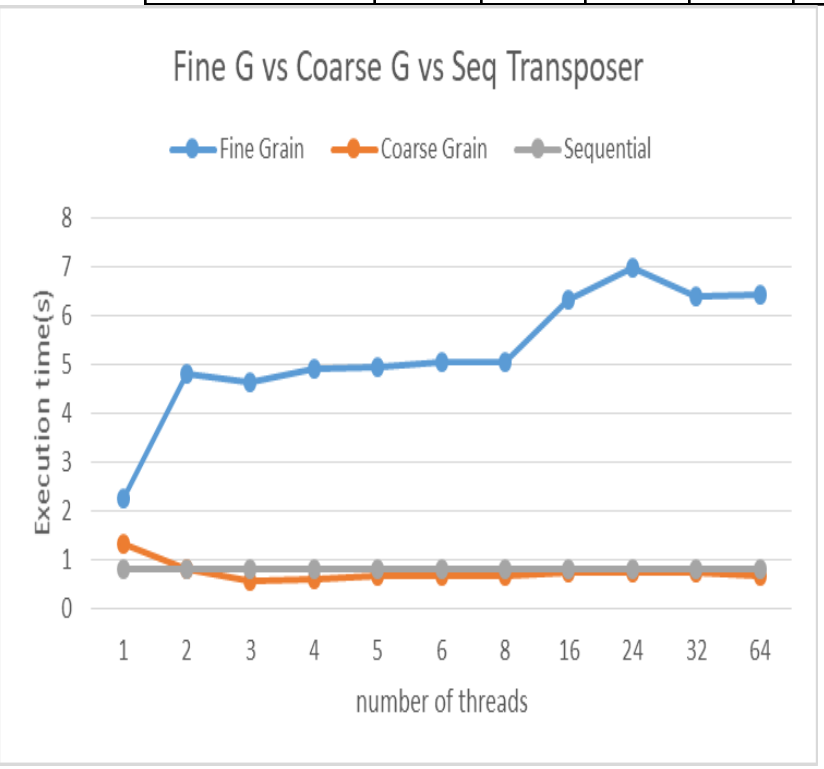
Grain = n^2/number of threads (Variable Grain Size): This grain setting starts at 1.36 seconds for a single thread and decreases as threads are added, with the most efficient execution at 24 threads (0.56 seconds). Unlike the coarse-grained configuration, this approach sees improved performance with increasing threads beyond the number of physical cores, but still experiences increased execution times at very high thread counts (64 and 128).

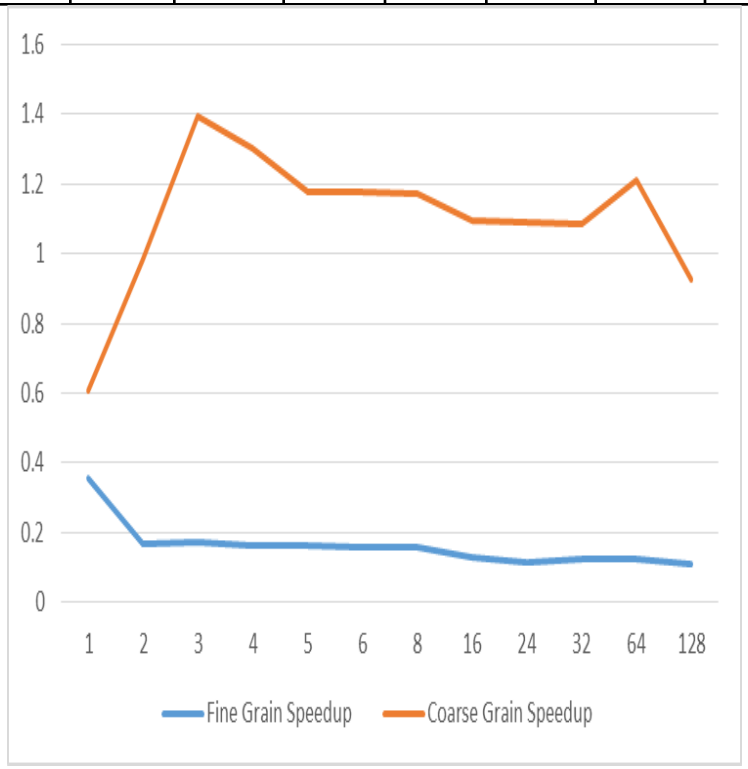
In conclusion, the optimal number of threads for coarse-grained transposition on this 8-core processor is 3, which is counterintuitive as we might expect 8 threads to be optimal. This could be due to the particulars of how the workload is distributed and managed across threads. For variable grain size, the performance improves as threads increase but eventually deteriorates, likely due to overhead. The

optimal thread count for variable grain size is 24, indicating a complex interplay between grain size, workload distribution, and threading overhead.

4.

# of Threads:	1	2	3	4	5	6	8	16	24	32	64	128
Fine Grain	2.266	4.812	4.625	4.911	4.952	5.033	5.033	6.321	6.985	6.384	6.406	7.442
Coarse Grain	1.326	0.821	0.578	0.618	0.686	0.686	0.687	0.735	0.739	0.743	0.666	0.87
Sequential	0.806	0.806	0.806	0.806	0.806	0.806	0.806	0.806	0.806	0.806	0.806	0.806

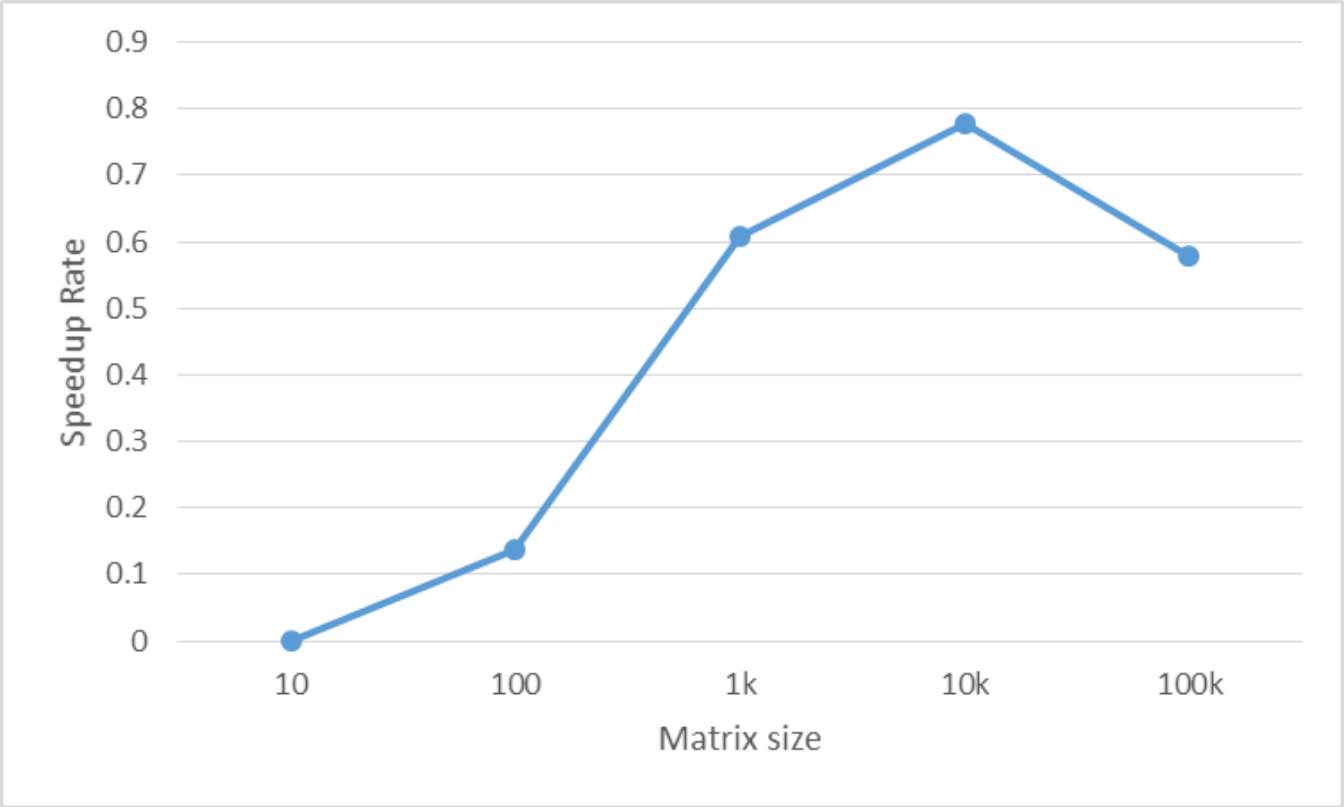




- The fine-grain configuration is consistently slower than the sequential version, as shown by speedups less than 1.
- The coarse-grain configuration starts out slower than the sequential version with one thread but quickly surpasses it with two threads, achieving speedups greater than 1, which means it's faster.
- The best speedup for coarse-grain is at 3 threads, this configuration is ~39.5% faster than the sequential version at this thread count.
- As the number of threads increases beyond 3, the speedup for coarse-grain decreases but remains above 1, indicating it's still faster than the sequential version but with diminishing returns.
- For fine-grain, increasing the number of threads increases the execution time compared to the sequential version, indicating that the overhead of managing fine-grain tasks outweighs the benefits of parallelization.

In summary, the coarse-grain configuration benefits from multi-threading up to a point, while the fine-grain configuration incurs too much overhead, resulting in slower performance than the sequential version.

5.Consider matrices of size n from 10 to 10,000. Graph the speedup obtained by the fastest coarse-grain transposer with respect to the sequential version. Explain your results.



Matrix sizes	10	100	1k	10k	100k
Sequential	0.00000133	0.00018131	0.01456157	1.74303375	4.1543411
Parallel with grain= n^2/threads	0.001427	0.00137113	0.07965172	6.92296731	14.3547932

16	
0.735	
0.536	

The speedup graph for the fastest coarse-grain transposer (i.e.grain= n^2/threads, with thread=16) compared to the sequential version across matrix sizes from 10 to 100,000 would show that the parallel version never achieves a speedup greater than 1. This means the parallel transposer is consistently slower than the sequential one for all matrix sizes in the range given.

The likely explanation for this is that the overhead of parallelism, such as thread management and synchronization, outweighs the benefits of concurrent execution for the sizes of matrices tested. Additionally, for larger matrices, memory access patterns and potential cache misses might further hinder the performance of the parallel algorithm.
Therefore, the parallel implementation with the coarse-grain approach does not provide a performance improvement over the sequential transposer for the given matrix sizes and grain configuration.