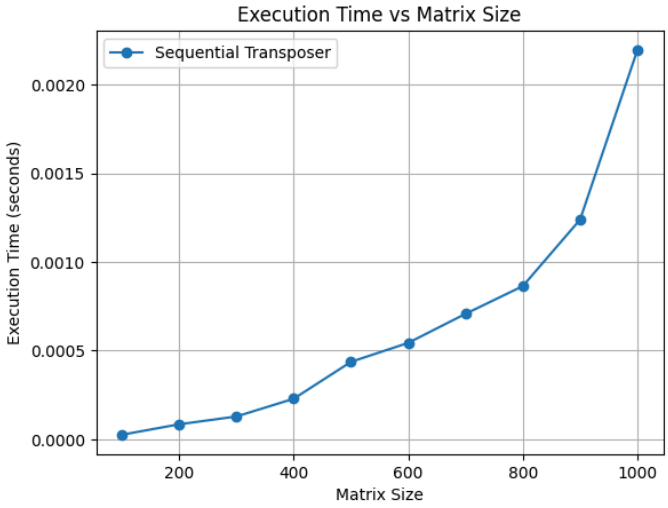
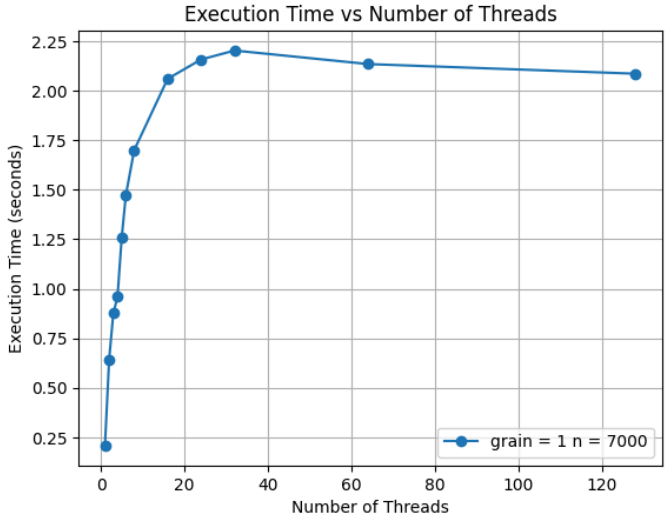
Homework 5: Matrix Transposition
CS231P
5/9/2025
TEAM: Destin Wong 64848542, Midhuna Mohanraj 39922268, Eric Huang 59504944

1. 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.



From the chart, it appears that the runtime increases exponentially as matrix size increases linearly. As our sequential matrix transposition function transpose matrices using two nested loops, it finishes in $O(n^2)$ time. So any increase to the matrix by size n increases runtime by n^2. When n=100, the number of elements in the array is 10000, but when you double the size of the array, the number of elements becomes 40000. Even though we only end up iterating over the upper triangular of the matrix, there is still a large amount of elements that need to be swapped.

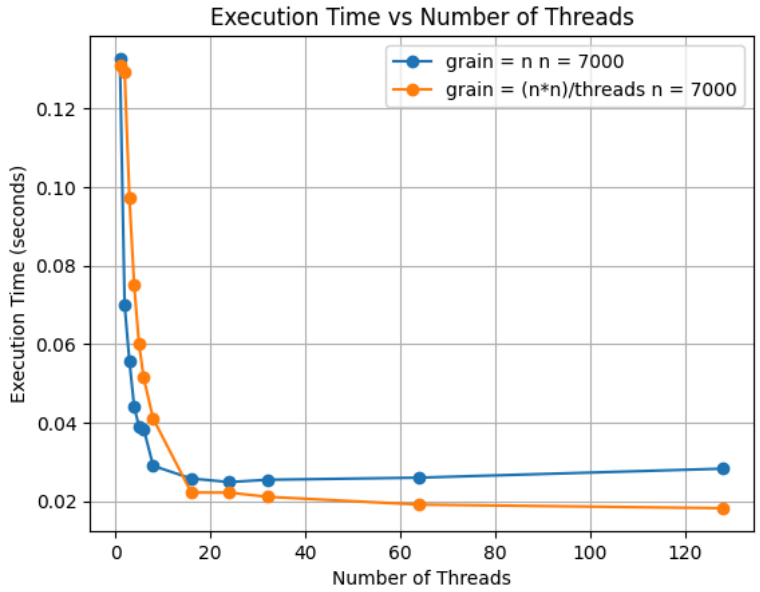
2. A characterization of the multi threaded transposer execution time for a fine-grain configuration.



Upon running the multithreaded transposer on a 7000×7000 matrix while increasing the number of threads, we observed that the runtime initially rises quickly, then plateaus around 32 threads, and gradually decreases as it approaches 128. While one would expect that more threads decrease runtime, the exact opposite occurs. The trend above shows that the overhead of managing additional threads outweighs the benefits of parallelism when each thread handles only a single unit of work. This makes sense as running multithreaded with grain=1 is effectively the same as running sequentially, but

with the added overhead of managing worker threads. The slight decreasing trend in runtime beyond 32 threads is probably due to more efficient CPU scheduling, but overall performance remains less efficient than with fewer threads due to the excessive overhead from fine-grain parallelism.

3. A characterization of the multi threaded transposer execution time for a coarse-grain configuration.



From the above graph, we can see that as the number of threads increases, execution time initially drops significantly due to increased parallelism. Both configurations show optimal performance around 16 to 32 threads. After 32 threads, the speed improvement levels off and even gets slightly worse because too many threads adds more overhead and increases competition for shared resources.

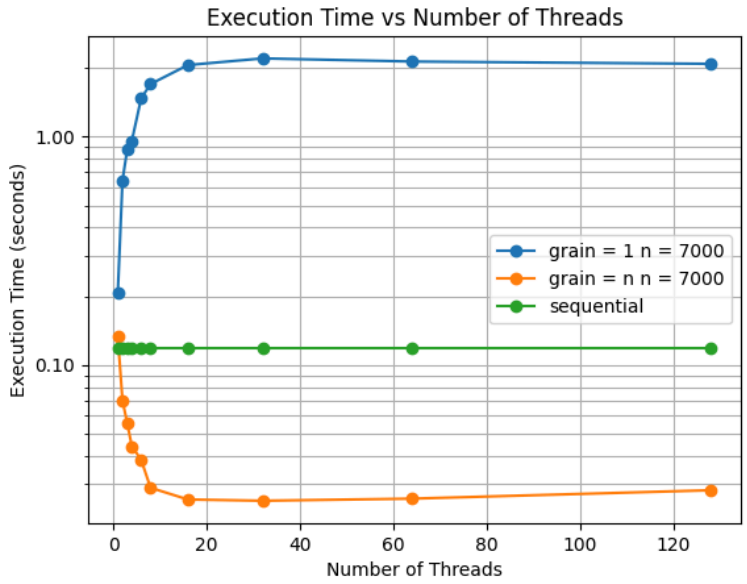
Initially, when the number of threads is low (below 16), grain=n performs better. For example, when threads=2, grain=n performs at half the runtime of grain=n*n/threads. At threads=2, the grain is 7000 for grain=n, while the grain is 24,500,000 for grain=n*n/threads. Because there are only 2 threads and the grain is 24,500,000 for the second system, each thread will have to process a large amount of work before requesting another task, and that ends up taking a longer time compared to the execution when the thread is 7000. The work is balanced better between the two threads in the first setup, as the faster thread can get more work, while the runtime is limited by the slower thread in the second. Then when threads=16, where the grain for grain=n is still 7000, and for grain=n*n/threads is 3,062,500, the second setup begins to outpace the first. This can be attributed to the workload for the second setup approaching a small enough amount to be processed without any significant delays between processes, while competition for shared resources in the first setup starts slowing it down. And generally after threads=16, the (n*n)/threads setup performs better with more threads, likely due to the previous reason, as threads never have to request more work, making better use of CPU.

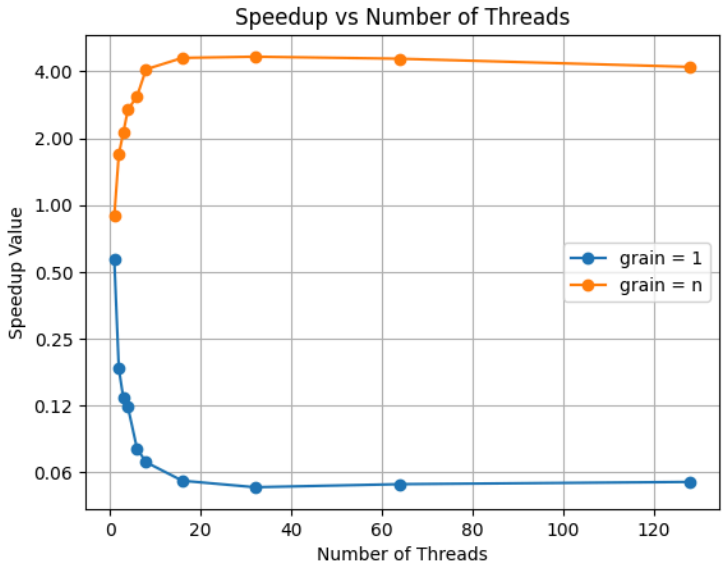
Lastly, if we compare the times in this graph to previously where grain = 1, we can see that increasing the coarseness of the grain tremendously helps with performance, because each

processor has a greater workload and can spend more time performing matrix transpositions (parallelizable work) than waiting on shared resources (non-parallelizable work).

Changing the grain to (n*n)/threads leads to an extremely coarser grain when compared to having grain = n. Comparing the performance between the two, having a more coarse grain leads to a slightly faster execution time as the number of threads increases. This could be because when more threads are added, more time is spent for each thread context switching, but when each thread has an increased amount of work due to the increased coarseness of the grain, it could lead to each thread taking more time on each job, which will lead to less time spent waiting.

4. Fixing n=7,000, grain=1 for finegrain, and grain=n for coarse-grain; graph the speedup.



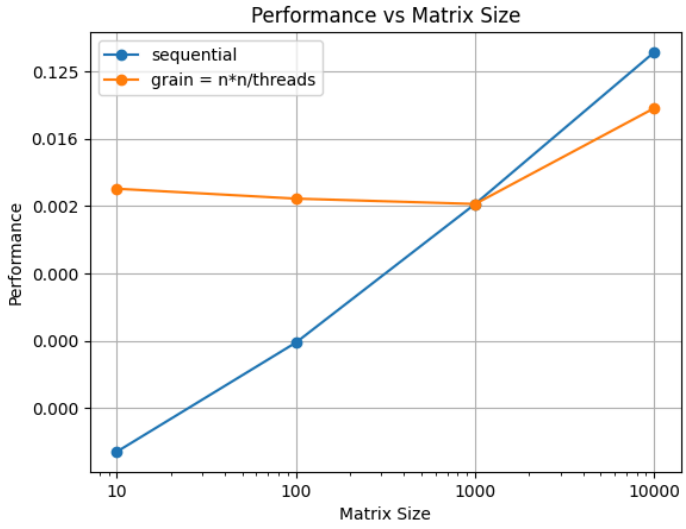


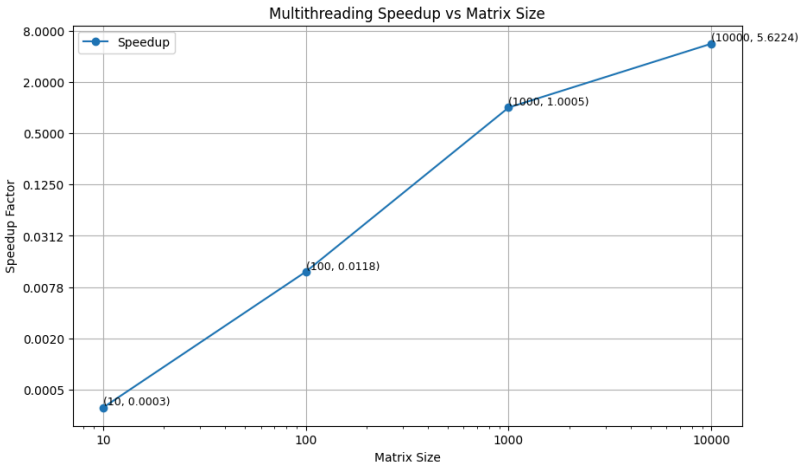
In the above graph, we've graphed the speedup factor achieved by the fine-grained setup (grain=1) vs coarse-grain (grain=n), that is, the runtime of the sequential version divided by the time of the multithreaded version. So a speedup value greater than 1 is faster, while a value less than one is slower than the sequential.

The fine-grain implementation consistently underperforms compared to the sequential execution. As the number of threads increases, the speedup continually decreases, staying below 1 in all cases. This is consistent with our previous observation that fine-grain multithreading effectively functions the same as sequential, but with the added overhead of multi-threading, making it much slower.

In contrast, the coarse-grain approach shows significant speedup, especially when increasing the number of threads from 1 to 16. Initially the performance with a single thread is slightly worse than sequential (likely due to the added time from the thread requesting additional work after each grain), but it immediately improves when we increase to 2 threads, when we begin to reap the benefits of parallelization. The performance improves rapidly up to around 16 threads, then plateaus, as competition for shared resources diminish the benefits of parallelization.

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.





Looking at our results from question 3, we chose to set the grain to (n*n) / number of threads and number of threads to 128, as that is when our program had the fastest execution time. Then we graphed the speedup factor by dividing the sequential runtime by the multithreaded runtime.

We can initially observe that at small matrix sizes (size 10 and 100) there is no speedup, but rather a slowdown compared to the sequential version. This is because in small matrix sizes, just setting up and assigning work to the threads takes more time than doing the actual transposition work, so it dominates the total computation time. Proportionally, the grain size was also very small. According to our grain size formula (grain=(n*n)/threads), when n = 10 our grain is 1, and when n = 100 the grain is 78. Because of the small grain for those matrix sizes, each thread didn't have a lot of work to do before finishing and wasting time waiting to be assigned its next grain of work.

At matrix size 1,000 the speedup factor is around 1, which means the multithreaded version is starting to catch up with the sequential one to being almost equal. According to the formula the grain size for this matrix size is 7,812, so threads would be spending much more time (100 times) doing transposition work than the threads for matrix size n=100. This is the point when the performance gains of parallelization pull equal with the slowdowns from thread scheduling and resource management.

At matrix size 10,000, the grain increases to 781,250, and the speedup increases significantly to 5.6224, meaning that the multithreaded version runs over 5 times faster than the sequential one. At this point the overhead from thread synchronization and work distribution becomes relatively small compared to the useful transposition work being done, so the performance increase from parallelization in this case vastly outweighs the negatives from shared resource management.