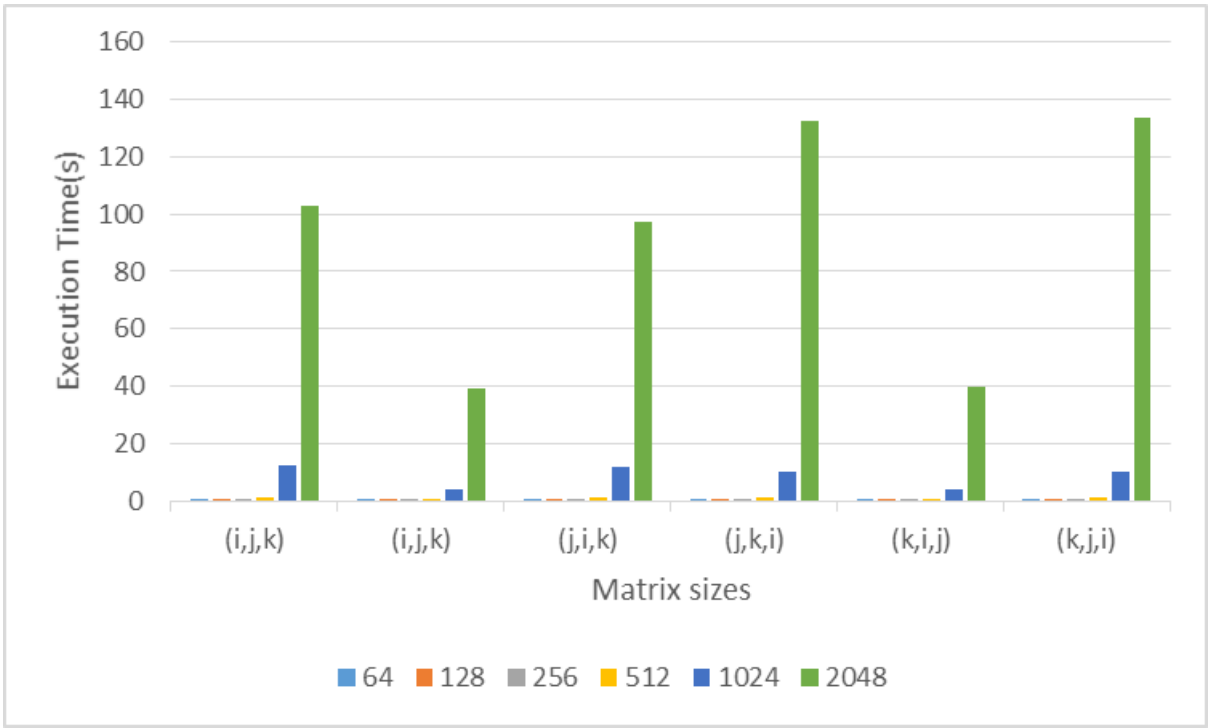
Parallel and Distributed Computing: Homework 4

1. The characterization of the computation progress in a single processor computer, for each of the six mentioned permutations. Add figures as necessary to support your explanation.

Analysis of the data indicates that loop orders i, k, j and k, i, j deliver the quickest processing times. Conversely, the sequences j, k, i and k, j, i lead to the longest durations. Loop orders i, j, k and j, i, k yield average processing times.

Permutation\matrix size	64	128	256	512	1024	2048
(i,j,k)	0.002434	0.018703	0.167512	1.556734	12.43343	102.6857
(i,k,j)	0.001277	0.011718	0.142534	0.912344	4.23566	39.3138
(j,i,k)	0.002204	0.020036	0.167436	1.625442	12.19462	97.1643
(j,k,i)	0.002256	0.012643	0.142436	1.321267	10.43343	132.4732
(k,i,j)	0.002311	0.017849	0.148764	0.914521	4.25265	39.9533
(k,j,i)	0.002297	0.018326	0.144436	1.325442	10.45123	133.4825



Execution Time (seconds) by Loop Order and Matrix Size

Formula Used: C[i][j] += A[i][k]*B[k][j]

For the sequences i, j, k and j, i, k, where k is nested deepest in the loop, the likelihood of retrieving A[i][k] from the cache is higher during iteration, which reduces cache misses due to column-wise traversal. However, since B[k][j] is accessed row-wise, there is an increased possibility of cache misses, leading to generally moderate execution performance for these permutations.

For the permutations i, k, j and k, i, j, the j loop is the innermost. Accessing A[i][k] may still encounter a moderate number of cache misses. The most efficient cache performance is observed while accessing B[k][j] because it utilizes column-wise access, which takes advantage of contiguous memory locations, leading to the most efficient execution times among the discussed variations.

In contrast, for the orders j, k, i and k, j, i, where i is the most rapidly changing index, accessing A[i][k] is less cache-friendly due to non-contiguous data access, leading to more cache misses. While the fetching of B[k][j] is not directly impacted by the innermost loop i, the overall execution times are higher because of the way B is accessed based on the positioning of the outer loops j and k.

2. Discussion of the mapping of the problem to a n-processors Ring. Add figures as necessary to support your explanation.

Unfolding on i (i, j, k and i, k, j): If you unfold the computation on i, each processor in an n-processor ring would compute a complete row of the result matrix C. This is because every element in a row of C only requires the corresponding row from A and all of the columns from B. This distribution can be efficient since it minimizes the required communication if B can be broadcast to all processors or circulated around the ring.

Unfolding on j (j, i, k and j, k, i): Unfolding on j implies that each processor is responsible for computing a complete column of the result matrix C. For this to happen, every processor needs one entire column of B and the complete matrix A. In the permutation j, i, k, after a processor completes a multiplication for a cell in C, it moves to the next row cell, whereas j, k, i implies a column-wise partial fill before completing the cell in multiple iterations. This may not be as communication-efficient since each processor requires access to the entire matrix A.

Unfolding on k (k, i, j and k, j, i): When unfolding on k, each processor in the ring would repeatedly contribute partial sums to every element of the result matrix C. This means that during each iteration of k, the processor must access the full matrices A and B, resulting in considerable communication overhead. It would require either broadcasting both A and B to all processors or a complex communication pattern to distribute the necessary elements of A and B around the ring.

3. Discussion of the gain in performance of the multi-thread implementation as a function of the number of threads concurrently used. Add plots and figures as necessary to explain your results.

Single-Thread Baseline: The line representing a single thread serves as a baseline to compare the performance gains of multi-threading. As matrix size increases, the execution time for the single-threaded case increases significantly, exhibiting a non-linear growth which suggests an algorithmic complexity that scales worse than linearly with matrix size.

Initial Gains with Multi-Threading: When introducing multiple threads, there's a stark decrease in execution time for all matrix sizes. For instance, the transition from 1 to 2 threads results in a clear reduction in execution time, indicating that the workload is being effectively distributed between two processors.

Continued Gains with More Threads: As the number of threads doubles from 2 to 4, and then to 8, we continue to see a decline in execution time, though the rate of decrease is not as dramatic as the initial gain from single to dual threading. This trend continues but with diminishing returns as the number of threads increases to 8 and 16.

Impact of Matrix Size: Larger matrix sizes show more significant performance improvements with the addition of threads. This is likely due to the larger workload being able to be divided more effectively across multiple threads. For small matrices (e.g., 64x64), the benefit of adding more than 2 threads is less noticeable, possibly because the overhead of managing threads can outweigh the benefits for smaller workloads.

Diminishing returns: There is a point of diminishing returns as the number of threads increases. This is evident when comparing the performance of 8 and 16 threads. Despite doubling the thread count, the execution times are remarkably similar, suggesting that the overhead of managing additional threads negates some of the parallel execution benefits. This overhead includes factors such as locking and unlocking of a mutex by each thread to safely read the values of the ith row before proceeding to compute, which can be a significant overhead in multi-threading environments.

To achieve loop unfolding, we have unfolded the outermost loop of the i, j, k combination in the matrix multiplication. All threads have unrestricted access to the entire A and B matrices. On each iteration, a thread locks the mutex to read the value of the current row (i), and after reading, it unlocks the mutex. This locking mechanism ensures that the read operation is thread-safe but introduces latency that becomes more apparent as the number of threads scales up.

In addition to thread management overhead, cache misses also play a role in performance scaling. With each thread having access to the full matrices, the cache may not always contain the required data, leading to costly memory accesses. As the number of concurrent threads increases, so does the likelihood of cache misses, resulting in an increased number of accesses to slower main memory.

thread/matrix size	64	128	256	512	1024	2048
1 Thread	0.002434	0.018703	0.167512	1.556734	12.43343	102.6857
2 Threads	0.001448	0.012378	0.107182	0.789804	6.230271	52.07068
4 Threads	0.001749	0.010989	0.065399	0.424674	3.151321	35.43699
8 Threads	0.00105	0.009294	0.062187	0.453808	3.075083	27.74505
16 Threads	0.001974	0.010466	0.056699	0.430425	3.110455	27.28733