Project 2

Due: October 10, 2025, at 11:59 pm (See BB for official due)

Please read the Common Projects Instructions first.

Problem 2.1: Algorithm Descriptions: For each of the following MPI collective operations, provide a concise algorithmic description using only point-to-point communications (e.g., MPI_Send, MPI_Recv). Include the function's purpose, a step-by-step algorithm (e.g., linear or tree-based), and one scalability consideration.

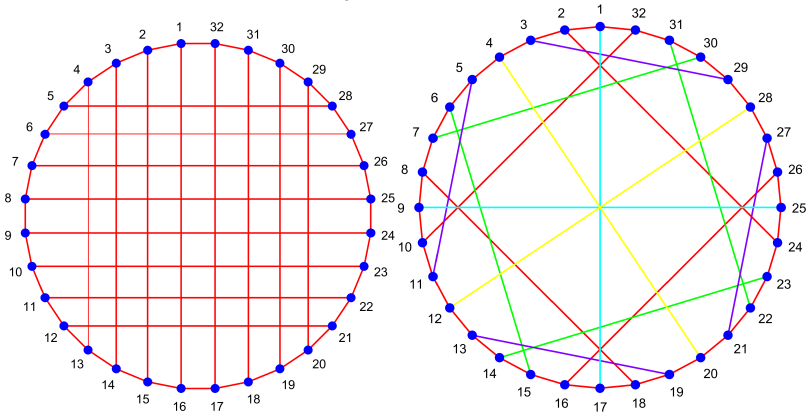
- MPI_Bcast: Broadcasts data from a root to all processes.
- MPI_Scatter: Distributes unique data portions from a root to all processes.
- MPI_Allgather: Gathers data from all processes and distributes the result to all.
- MPI_Alltoall: Each process sends/receives distinct data to/from all others.
- MPI_Reduce: Combines data (e.g., sum, min) from all processes to a root.

Implement MY_Global_Min_Loc: Implement a function MY_Global_Min_Loc that identifies the processor ranks holding the minimum values across distributed arrays, using only point-to-point communications. This implementation involves a few constraints/checks**:** (1) Do not use MPI_Reduce or other collectives. (2) Handle edge cases: $P = 1$, varying N, ties in minima (return any valid rank). (3) Ensure scalability for $P \geq 4$, $N \geq 100$. (4) Note the minima being computed is not within each array, e.g., $\min\{A_0\}$.

- **Input**: Each of Pprocesses has a local array $A_p$of Nintegers.
- **Output**: The root process returns an array of length N, where the i-th element is the rank of the process with the minimum value among $A_0[i], A_1[i], \dots, A_{P-1}[i]$.
- **Example**: For $P = 3, N = 4$:
  - Process 0: $A_0 = \{1, 9, 3, 4\}$
  - Process 1: $A_1 = \{5, 6, 7, 2\}$
  - Process 2: $A_2 = \{9, 8, 6, 1\}$
  - Output on root: {0,1,0,2}, where:
    - Index 0: min{1, 5, 9} = 1 (Process 0).
    - Index 1: min{9, 6, 8} = 6 (Process 1).
    - Index 2: min{3, 7, 6} = 3 (Process 0).
    - Index 3: min{4, 2,)} = 1 (Process 2).

- **Verify** correctness of MY_Global_Min_Loc using the example above and additional cases (e.g., $P = 4, 8$; $N = 10, 100$).
- **Measure** execution time with MPI_Wtime for varying P and N.
- **Output** results to a file (e.g., results.txt) with correctness checks and timing data.
- **Comment** on performance

Problem 2.2: You design parallel algorithms to perform an all-gather operation on two network topologies, **Bid32** (left) and **Opt32**, each with 32 vertices (representing compute nodes or cores) connected by undirected edges (representing bidirectional network links). The topologies are depicted in provided graphs where overlapping edges in the graphs are not connected; each edge starts and ends at distinct vertices. Additionally, in Opt32, various colors of edges do not concern us.



Each node starts with N random floating-point numbers, and the all-gather operation ensures every node receives the $32 \times N$ numbers from all nodes (including its own).

(1) **Algorithm Design:** Design a custom all-gather operation on both Bid32 and Opt32 topologies, using MPI point-to-point communications (e.g., MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv). MPI_Allgather may and should be used for comparison. For each topology:
- Describe the communication pattern (e.g., ring, tree, or topology-specific).
- Account for the undirected edges in the provided graphs, ensuring efficient data exchange.
- Discuss communication bottlenecks, link load unbalancing, if any.

(2) **Implementation and evaluation:** Implement the all-gather algorithms for both topologies.
- **Input**: Each of the 32 nodes starts with an array of N random floating-point numbers.
- **Output**: Each node ends with $32 \times N$ numbers (all nodes' data, including its own).
- **Requirements**:
  - Use your custom all-gather.
  - Map the Bid32 and Opt32 topologies to your MPI processes (e.g., via adjacency lists or matrices based on the provided graphs).
  - Test for $N = 10, 20, 30, 40$.
  - Handle edge cases (e.g., ensuring no deadlocks, verifying data integrity).
- **Verify** correctness and measure execution time using MPI_Wtime. Construct a table to record and compare performances:

	On Bid32	On Opt32
$N = 10$		
$N = 20$		
$N = 30$		
$N = 40$		

Problem 2.3: Design a parallel matrix multiplication using row-wise, column-wise, or block decomposition. This assignment requires implementing a parallel version using MPI, assuming square matrices for simplicity. Tasks are

(1) **Algorithm Design:** Design a parallel matrix multiplication algorithm using MPI.
- Describe the data distribution.
- Discuss load balancing, communication overhead, and scalability for P processes.
- Assume matrices are square (size $N \times N$) and P divides N.

(2) **Implementation:** Implement the algorithm in a program.
- **Input**: Root process generates two random N × N matrices A and B.
- **Output**: Root process computes and verifies C against a serial version.
- **Requirements**:
  - Support $P = 2^2, 2^3, 2^4$.
  - Test for $N = 2^6, 2^8, 2^{10}, 2^{12}$.
  - Include a serial baseline for verification.

(3) **Performance Evaluation:** Create a test harness to:
- Verify correctness.
- Measure execution time.
- Compute speedup.
- Record the timing results and make relevant comments on Amdahl's law:

N	$P = 2^2$	$P = 2^3$	$P = 2^4$
2^6			
2^8			
2^{10}			
2^{12}			

Some leftover space allows me to paste a picture that's relevant: