**Submission Guidelines:**

* Code Submission: Submit your MPI code as a separate file, clearly labeled with your name and assignment details.
* Report Submission: Provide a separate report detailing your approach, algorithm, parallelization strategy, and any challenges faced during implementation.
* Submission Format: Submit both the code and report online using rollno\_section\_assignmentno format on GC.
* Deadline: Ensure that your submission is made before the specified deadline. Late submissions will not be accepted.

**Chapter 5**

**Question 1:** Iso-efficiency and Scalability Analysis

Consider a parallel algorithm for matrix multiplication where the iso-efficiency function is  Suppose you double the number of processing elements (p) while maintaining constant efficiency.

1. Derive the required growth rate of the problem size (W) to preserve efficiency.
2. If the communication overhead for this algorithm is modeled as , show whether the system remains cost-optimal under this scaling.
3. Critically evaluate: Why might iso-efficiency fail to fully capture scalability in systems with non-uniform memory access (NUMA) architectures?

**Question 2:** Cost-Optimality and Overhead Trade-offs

A parallel system solves a problem with a serial runtime . When parallelized, it incurs:

* Excess computation overhead: ,
* Communication overhead: ,
* Idling time: .

1. Derive the parallel runtime Tp and efficiency *E* as functions of n and p.
2. Prove whether this system is cost-optimal for p = O(n).
3. If the system is not cost-optimal, propose modifications to the algorithm (e.g., granularity adjustment, load balancing) to achieve cost-optimality, and re-derive the conditions.

**Chapter 6**

you are required to implement a parallel solution to the Laplace Equation using MPI (Message Passing Interface) for distributed memory systems. The Laplace equation is a widely used partial differential equation (PDE) for modeling steady-state heat distribution, and is commonly solved numerically using iterative methods like the Jacobi or Gauss-Seidel method. Your task is to apply the Jacobi iteration method to compute the final temperature values on a 2D metal plate given fixed boundary condition, and parallelize this process using MPI.

**Problem Statement:**

You are provided with a 2D grid of size N x N representing a metallic plate, where:

* The boundary cells are fixed with known temperature values.
* The inner cells are updated iteratively using the average of their four neighboring cells until convergence.

Your objective is to perform this computation in parallel, using domain decomposition (row

wise or block-wise) with MPI.

**Input:**

* Size of the 2D grid N (assume a square grid).
* Initial temperature values on the boundaries.
* Convergence threshold ε (epsilon), e.g., 0.001.

**Output:**

* Final temperature values for the entire grid (after convergence).
* Number of iterations performed.
* (Optional) A heat map visualization (if supported in your environment).

**Here are the key steps for the assignment:**

1. Understand the serial implementation of the Jacobi iteration provided above.
2. Identify the parallelizable portions of the code (mainly the 2D update loop and the convergence check).
3. Initialize MPI and determine the total number of processes and the rank of each process.
4. Distribute the 2D grid among MPI processes, preferably row-wise.
5. Exchange halo (ghost) rows with neighboring processes using MPI\_Sendrecv or non-blocking communication.
6. Perform the Jacobi iteration in parallel, ensuring all processes synchronize global convergence using MPI\_Allreduce.
7. Gather the final temperature grid on the root process.
8. Verify correctness by comparing the output with the serial version.

**Ensure that you:**

* Properly handle edge cases (e.g., small grids, uneven splits).
* Avoid deadlocks in communication.
* Optimize communication to reduce overhead and maximize parallel speedup.
* Validate your implementation with various values of N and ε.

**Implementation Guidelines:**

* MPI Parallelization: Use MPI to split the grid across processes.
* Data Communication: Carefully exchange boundary rows (top and bottom) between
* neighboring processes.
* Error Handling: If N is not divisible by the number of processes, handle extra rows
* appropriately.
* Convergence Check: Use global reduction (MPI\_Allreduce) to compute the maximum
* difference across processes.

Question 02:

**Title: Implementing Deadlock-Free Point-to-Point Communication using Blocking and Non-Blocking MPI Operations**

You are required to:

1. **Implement two MPI programs** in C using the MPI library:
   * One using **blocking send and receive (MPI\_Send and MPI\_Recv)**
   * One using **non-blocking send and receive (MPI\_Isend, MPI\_Irecv, and MPI\_Wait)**

Both programs should simulate a **ring topology** (i.e., each process sends a message to the next process and receives from the previous one). Assume wrap-around communication (i.e., process 0 receives from the last process).

1. Your implementation should:
   * Use **MPI\_COMM\_WORLD** as the communicator.
   * Correctly handle the ranks of sending and receiving processes.
   * Prevent **deadlocks** in both cases.
   * Use an array of integers as the message buffer.
   * Demonstrate the difference in performance behavior or idling between the two approaches (comment on this).
2. **Report Requirements:**
   * Clearly explain how your implementation avoids **deadlocks**.
   * Highlight differences between blocking and non-blocking behavior.
   * Include **code snippets**, **execution results**, and your **observations** on program behavior.