Introduction

This lab implements and evaluates distributed algorithms using Python and the **`mpi4py`** library to gain practical experience with collective communication and custom inter-process message-passing patterns. The exercises demonstrate how to decompose work across ranks, coordinate partial results, and measure the trade-offs between communication overhead and parallel computation.

Question 1: Parallel Prefix Sum

Algorithm Description

- Step 1 (Parallel): Each process computes the sum of its local block (local_sum)
- Step 2 (Sequential, root): All local sums are gathered at the root; the root computes offsets (prefix of block sums) to determine how much to add to each block's local prefix.
- Step 3 (Parallel): The root scatters offsets back; each process computes a local prefix (**`np.cumsum`**) and adds the received offset to produce the final local prefix.

Implementation (in **`prefix_sum.py`**)

- The program uses **`MPI.Scatter`** to distribute contiguous slices of the input array to each process and **`MPI.Gather`** to reassemble local prefix results at the root.
- The root uses **`comm.gather`** to collect block sums and computes offsets, then **`comm.scatter`** to distribute offsets.
- Local prefix computation uses **`numpy.cumsum`**, and final arrays are typed consistently (int32) for MPI communication.

Results

- The implementation performs a correctness check on the root: the MPI-computed prefix sum is compared to **`numpy.cumsum`** of the original array. The test reported a match (i.e., the MPI result equals the sequential Numpy result), confirming correctness for the tested inputs.

Question 2: Tree-Based Reduction

Algorithm Description

- A manual binary-tree reduction is implemented instead of using **`MPI.Reduce`**.
- Each process determines its children at indices **`2*rank + 1`** and **`2*rank + 2`**. Children send their partial results upward to their parent at **`(rank - 1) // 2`**.
- Internal nodes sum received child data with their own local data and forward the accumulation upward. The root (rank 0) receives from children and holds the final reduced result.

Implementation (in **`tree_reduce.py`**)

- Each process initializes a local matrix (here, 500×500 of ones).
- Non-root processes send their accumulated matrix to their parent using **`comm.Send(...)`** with consistent MPI datatypes; processes that have children use **`comm.Recv(...)`** to receive and add those child matrices into their accumulator.
- The root verifies the final result and measures elapsed time using **`MPI.Wtime()`**.

Performance Analysis

- Experiment context: timings obtained from running **`tree_reduce.py`** with **`mpiexec -n 4`** on square matrices of increasing size.

Size	Processes	Parallel (s)	Sequential (s)	Speedup
500 × 500	4	0.003655	0.002530	0.69×
2000 × 2000	4	0.041267	0.016785	0.41×
4000 × 4000	4	0.199959	0.044463	0.22×

Analysis

- 500 × 500 case: The sequential run is faster (speedup < 1). This is explained by the MPI overhead (message startup latency, buffer handling, multiple Send/Recv operations and synchronization) which dominates when the per-process computational work is small; the cost of communication outweighs the modest compute savings from parallelization for this problem size.
- 4000 × 4000 case: Although the measured parallel time is still larger than the sequential time in this dataset, the gap reduces as size grows. Intuitively, the compute work grows roughly with N^2 (number of matrix elements), while the number of communication rounds or synchronization steps grows much more slowly (and per-element messaging cost grows more slowly relative to total computation in many practical settings). Therefore, for sufficiently large N (or with higher per-element computational intensity), the parallel tree reduction will eventually approach — and then surpass — the sequential performance as computation increasingly amortizes communication costs.
- Tree topology efficiency: The binary-tree reduction reduces communication contention compared to naïve all-to-root schemes by combining partial results in log(P) rounds and limiting simultaneous sends to parents/children. This makes the topology efficient for large payloads and many processes. However, for small matrices the per-message latency and setup cost still make the parallel approach less efficient; the tree approach becomes advantageous as matrix size or computational complexity increases.

Conclusion

This lab highlights the classic trade-off in distributed computing: parallel algorithms can reduce wall-clock time for sufficiently large workloads, but for small problems the overhead of message passing and synchronization may make sequential execution faster. Implementing a binary-tree reduction and a block-wise parallel prefix sum in **`mpi4py`** provided practical insight into how collective behavior can be composed from point-to-point primitives and how to reason about performance in terms of communication cost vs. computation.