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

- To evaluate the scalability of the Tree-Based Reduction, we implemented the algorithm in C++ and simulated it using **SimGrid** on three different network topologies: Cluster Crossbar (high speed), Fat Tree, and Torus (grid-based). The results are presented in the following table:

Matrix Dimensions (Total Elements)	Sequential Time (s)	Parallel Time (s)	Number Of Processes	Topology	Notes
500	0.00124	0.00194	32	Cluster Crossbar	Sequential is faster (Small load)
500	0.00213	0.00233	64	Cluster Crossbar	Sequential is faster
500	0.00075	0.00172	32	Torus	Sequential is faster
500	0.00119	0.00206	64	Torus	Sequential is faster
500	0.00141	0.00311	32	Fat Tree	Sequential is faster
500	0.00209	0.00364	64	Fat Tree	Sequential is faster
1000	0.00176	0.00232	32	Cluster Crossbar	Sequential is faster
1000	0.00318	0.00280	64	Cluster Crossbar	Parallel starts winning
1000	0.00098	0.00206	32	Torus	Sequential is faster
1000	0.00169	0.00248	64	Torus	Sequential is faster
1000	0.00185	0.00368	32	Fat Tree	Sequential is faster
1000	0.00302	0.00435	64	Fat Tree	Sequential is faster
10000	0.01555	0.00793	32	Cluster Crossbar	Parallel is 2x faster
10000	0.03114	0.00960	64	Cluster Crossbar	Parallel is 3.2x faster
10000	0.00748	0.00734	32	Torus	Speeds are equal
10000	0.01653	0.00947	64	Torus	Parallel is faster
10000	0.01512	0.01042	32	Fat Tree	Parallel is faster
10000	0.02971	0.01259	64	Fat Tree	Parallel is faster
100000	0.11436	0.04520	32	Cluster Crossbar	Parallel is 2.5x faster
100000	0.23110	0.05534	64	Cluster Crossbar	Parallel is 4.1x faster
100000	0.05575	0.04361	32	Torus	Parallel is faster
100000	0.12484	0.05811	64	Torus	Parallel is 2x faster
100000	0.11093	0.05369	32	Fat Tree	Parallel is 2x faster
100000	0.22098	0.07076	64	Fat Tree	Parallel is 3x faster
1000000	1.13453	0.35646	32	Cluster Crossbar	Parallel is 3.1x faster
1000000	2.30188	0.43492	64	Cluster	Parallel is 5.2x

				Crossbar	faster
1000000	0.55531	0.36042	32	Torus	Parallel is 1.5x faster
1000000	1.25575	0.50339	64	Torus	Parallel is 2.5x faster
1000000	1.09865	0.37555	32	Fat Tree	Parallel is 2.9x faster
1000000	2.20332	0.52812	64	Fat Tree	Parallel is 4.1x faster
5000000	5.67286	1.73630	32	Cluster Crossbar	Parallel is 3.2x faster
5000000	CRASH	CRASH	64	Cluster Crossbar	Sequential caused OOM Error
5000000	2.77723	1.77844	32	Torus	Parallel is 1.5x faster
5000000	6.25996	2.53623	64	Torus	Parallel is 2.4x faster
5000000	5.49263	1.81468	32	Fat Tree	Parallel is 3x faster
5000000	11.11381	2.68551	64	Fat Tree	Parallel is 4.1x faster

Experimental Setup:

The performance of the Parallel Tree-Based Reduction was compared against a Sequential Gather-and-Sum implementation using the **SimGrid** simulator. The simulation was conducted using a C++ implementation on three distinct network topologies: Cluster Crossbar (high bandwidth), Torus (grid/ring latency), and Fat Tree (hierarchical). We varied the data size per process from 500 to 5,000,000 integers and the number of processes (P) from 32 to 64.

1. Impact of Data Size and Overhead (Small Data)

For small data sizes (500 to 1,000 integers), the Sequential implementation consistently outperformed the Parallel Tree.

- Observation: At 500 integers on a Cluster Crossbar (P=32), Sequential took 0.0012s while Parallel took 0.0019s.
- Reasoning: The $O(\log P)$ advantage of the tree algorithm is negligible when the computation load is tiny. In these cases, the overhead of establishing the manual tree communication structure dominates. Additionally, MPI_Gather is a highly optimized collective operation, whereas the manual tree implementation uses blocking MPI_Send/MPI_Recv, incurring synchronization latency that outweighs the benefits for small payloads.

2. Scalability and the Crossover Point (Medium to Large Data)

As the data size increased to 10,000 integers and beyond, the Parallel Tree implementation began to significantly outperform the Sequential version.

- The Crossover: At 10,000 integers, the Parallel implementation became approximately 2x faster than Sequential on the Cluster Crossbar (P=32).

- Large Scale Dominance: At 5,000,000 integers, the performance gap widened drastically. On the Fat Tree topology with 64 processes, the Sequential method took 11.11s, while the Parallel Tree took only 2.68s, representing a 4.1x speedup.
- Reasoning: As the payload grows, the bottleneck shifts from network latency to bandwidth and serialization at the root. The Sequential algorithm forces the root node to receive $P-1$ messages sequentially ($O(P)$), causing a bottleneck. The Tree algorithm distributes this load across the network ($O(\log P)$), allowing multiple nodes to perform reduction simultaneously.

3. Memory Efficiency and System Stability

A critical finding occurred during the largest test case (5,000,000 integers) on the Cluster Crossbar with 64 processes.

- The Crash: The Sequential implementation failed to complete, terminating with an error (likely Out Of Memory/OOM).
- Analysis: The Sequential approach requires the root node to allocate a receive buffer large enough to hold data from all processes simultaneously ($64 \times 5M \times 4$ bytes≈ 1.28 GB). In a constrained simulation environment, this caused a memory allocation failure.
- Tree Advantage: The Parallel Tree implementation completed successfully. Since each node only receives data from its two children and sums it immediately into its local buffer, the memory requirement remains constant per node regardless of the total number of processes. This proves the Tree algorithm is strictly superior for memory-constrained distributed systems.

4. Topology Comparison

- Cluster Crossbar: Provided the fastest absolute times due to direct links, but highlighted the memory bottleneck of the sequential approach most sharply.
- Fat Tree: Showed the highest speedup factor (4.1x at 64 processes), demonstrating that hierarchical network structures benefit significantly from hierarchical algorithms like the Tree Reduce.
- Torus: The Sequential method remained competitive longer on the Torus topology (winning up to 1,000 integers). This is likely because the Torus relies on neighbor-to-neighbor passing, and MPI_Gather implementations often optimize for this by using ring algorithms. However, at 5,000,000 elements, the Parallel Tree still achieved a 2.4x speedup.

Conclusion

The results confirm that while optimized Sequential collectives are faster for small messages due to lower overhead, Tree-Based Reduction is essential for scalability. It outperforms sequential approaches by over 400% on large workloads and prevents critical memory failures at the root node.