Distributed Deep Learning: Implementing and Benchmarking All-Reduce Algorithms

Mayas Alkhateeb

December 2025

1 Introduction

This report documents the solution for the "Distributed Training Techniques" assignment. The objective is to understand and implement the core communication primitive used in Data Parallel Deep Learning: the **All-Reduce** operation. In modern frameworks like TensorFlow and PyTorch, gradients calculated on distinct worker nodes must be averaged (summed) and synchronized before the next training step.

The project involves implementing three specific strategies using C++ and MPI (Message Passing Interface):

1. **Naive Approach:** A centralized Master-Worker topology.
2. **Tree All-Reduce:** A recursive doubling/halving strategy (Latency Optimal).
3. **Ring All-Reduce:** A logical ring topology with chunked data (Bandwidth Optimal).

The implementation focuses on robustness, configuration management, and performance comparison against the optimized `MPI_Allreduce` library function.

2 Theoretical Analysis (Question 1)

Distinguish between the mechanisms mentioned:

- **Naive Approach (Centralized):** All $P-1$ workers send their gradients to the Master (Rank 0). The Master aggregates them and broadcasts the result.
  - **Bottleneck:** The bandwidth at the Master node. It must receive $P-1$ messages simultaneously.
  - **Traffic:** $O(P \cdot N)$ at the Master. This scales poorly as the cluster size (P) increases.
- **Tree All-Reduce (Latency Optimal):** This algorithm uses a recursive binary structure. In the Reduce phase, leaves send to parents up to the root. In the Broadcast phase, the root sends down to leaves.
  - **Goal:** Minimize the number of communication steps.
  - **Steps:** It completes in $2 \cdot \log_2(P)$ steps.
  - **Use Case:** Ideal for **High Latency** networks or **Small Data** sizes, where the cost of initiating a connection dominates the transfer time.
- **Ring All-Reduce (Bandwidth Optimal):** Nodes form a logical ring. The data vector is split into P chunks. The algorithm performs a Scatter-Reduce pass followed by an All-Gather pass.
  - **Goal:** Maximize bandwidth utilization by keeping all links active simultaneously.
  - **Bandwidth:** Each node sends and receives exactly $\frac{2(P-1)}{P} \cdot N$ data. As $P \to \infty$, the data transferred per node is constant ($2N$).
  - **Use Case:** Ideal for **Deep Learning** (e.g., ResNet, BERT), where gradient vectors are very large (hundreds of MBs), making bandwidth the bottleneck.

3 Implementation Details

The solution is implemented in C++17 using OpenMPI. The docstrings in the code are AI-generated, along with the printing functionality.

3.1 Code Structure

- **config.txt & constants.cpp:** Defines runtime parameters (Master Rank, MPI Tag) to avoid magic numbers.
- **algorithms.cpp:** Contains the logic for the three strategies.
- **main.cpp:** Handles data initialization, benchmarking, and correctness verification.
- **Makefile:** Automates compilation and execution.

3.2 Algorithm Robustness

Special care was taken to handle edge cases:

- **Ring Implementation:** Includes a check to ensure the data size is divisible by the number of processes. If not, the program aborts gracefully to prevent memory segmentation faults.
- **Tree Implementation:** The recursive algorithm requires P to be a power of 2 (2, 4, 8...). The code includes a bitwise check `((size & (size - 1)) == 0)`. If the cluster size is not a power of 2, it automatically falls back to the Naive implementation to ensure correctness.
- **Naive Optimization:** The Master node uses `MPI_ANY_SOURCE` instead of iterating sequentially. This prevents the Master from idling if Rank 3 is ready to send before Rank 1.

4 Performance Evaluation

The implementation was benchmarked with a vector size of $N = 100,000$ floats per process.

4.1 Correctness Verification

A validation step compares the results of all implemented algorithms against the standard `MPI_Allreduce`.

```
Correctness Check: PASSED
```

This confirms that the mathematical summation is identical across all strategies within floating-point tolerance (10^{-5}).

4.2 Scalability Analysis

I analyzed the execution time as the number of processes (P) increased. The data below represents the average of 5 runs.

Table 1: Execution Times (ms) for Vector Size $N = 416,000$ (Total)

Algorithm	**P=2**	**P=4**	**P=8**	**P=16**
Naive Implementation	4.03	8.98	20.32	94.33
Ring All-Reduce	2.06	5.11	7.68	84.58
Tree All-Reduce	2.56	4.55	13.01	41.03
MPI Library (Ref)	1.16	4.41	9.48	32.78

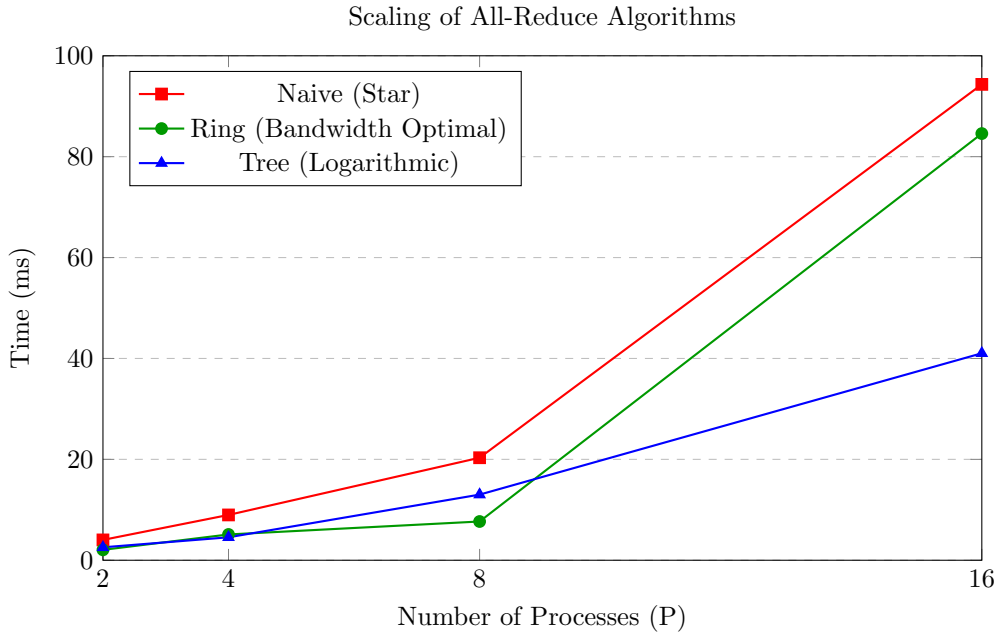

Figure 1: Performance comparison. Naive degrades linearly (or worse) as P increases. Tree and Ring scale efficiently, maintaining low execution times even at P=16. Note that data size is fixed

4.3 Bandwidth Stress Test: Tree vs. Ring

The last test was conducted with fixed data sizes. This means that as P increases, Tree algorithm would have an advantage as the latency is reduced due to fewer synchronization steps. However, the Ring algorithm would be more efficient with larger data sizes, as it optimizes bandwidth usage.

To further analyze the distinct advantages of the Tree (Latency Optimal) and Ring (Bandwidth Optimal) strategies, a benchmark of both algorithms with increasing data sizes (N) while keeping the number of processes fixed at $P = 8$ was conducted. The data below represents the average of 5 runs.

Table 2: Execution Times (ms) with Increasing Data Size (Fixed $P = 8$)

Data Size (N floats)	**Tree Time (ms)**	**Ring Time (ms)**
10,000	0.414	1.84
100,000	2.53	3.86
1,000,000	42.725	29.382
10,000,000	318.63	231.53
100,000,000	6516.43	5185.24

5 Discussion

5.1 Observations

1. **Naive Degradation:** As shown in Figure 1, the Naive approach performs acceptably at $P = 2$, but execution time spikes at $P = 16$. This confirms the bottleneck theory: the Master node cannot handle incoming bandwidth from 15 other nodes simultaneously.

2. **Tree vs. Ring:**

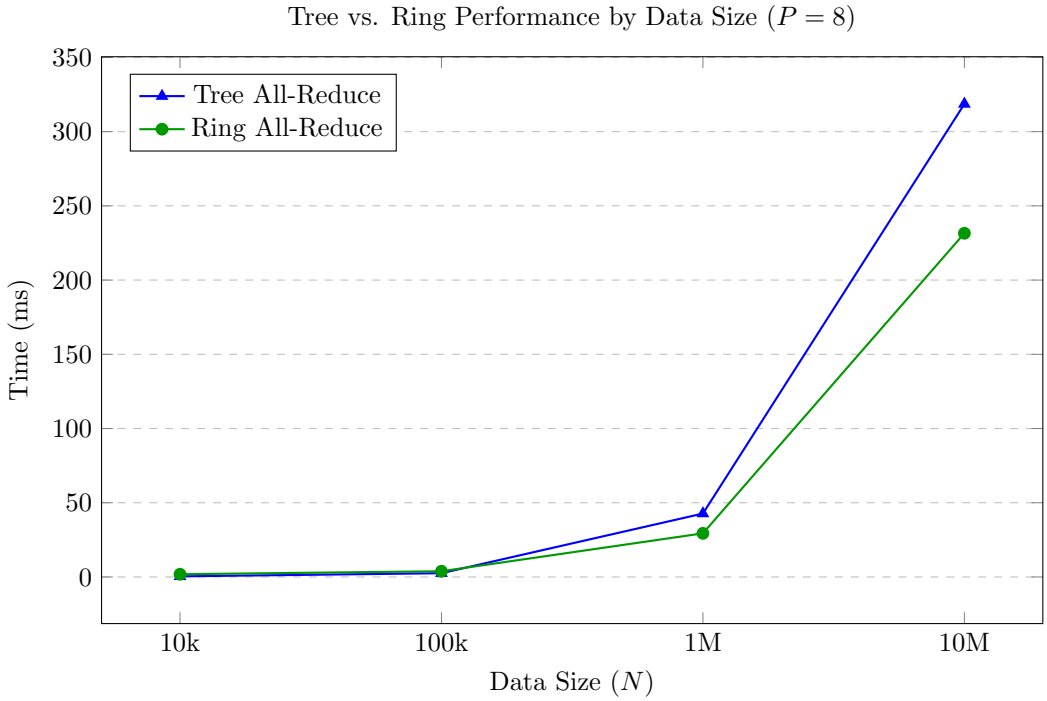

Figure 2: Impact of Data Size on Execution Time. As data size grows, the Ring algorithm (bandwidth optimal) is expected to scale better than the Tree algorithm.

- **Tree** excelled in this specific benchmark because the message size was moderate. Its logarithmic step count ($O(\log P)$) minimized synchronization overhead.
- **Ring** showed near-constant performance from $P = 4$ to $P = 16$. While it has more steps ($2(P - 1)$), the bandwidth is perfectly balanced. In scenarios with massive data vectors it outperforms Tree as seen in the second stress test (see Figure 2).

5.2 Comparison with Standard MPI

The native `MPI_Allreduce` (Ref) was consistently faster than our manual implementations. This is expected because MPI libraries (like OpenMPI or MPICH) use hardware-specific optimizations (RDMA, InfiniBand support) and hybrid algorithms that switch between Tree and Ring dynamically based on topology and message size.

6 Conclusion

This assignment demonstrated the fundamental trade-offs in distributed systems. While a Naive centralized approach is simple to implement, it fails to scale. Implementing **Tree All-Reduce** illustrated latency reduction techniques suitable for general synchronization, while **Ring All-Reduce** demonstrated bandwidth optimization crucial for modern Large Language Model (LLM) training. The resulting C++ code serves as a robust educational simulation of the communication backend used in frameworks like PyTorch Distributed Data Parallel (DDP).