CS406-531 Parallel Computing: Homework 3 Report

MPI All-to-All Implementation

Kerem Tufan
ID: 32554

January 2, 2026

1 Implementation Details

1.1 Algorithm Logic

The goal of this assignment was to implement the `MPI_Alltoall` collective behavior using only point-to-point communication primitives (`MPI_Send` and `MPI_Recv`).

My implementation utilizes a **Linear Shift (Ring-like)** algorithm combined with a manual deadlock avoidance mechanism. The logic iterates through a loop where each process P communicates with a peer process determined by an offset i.

The core logic for determining the target and source for each step i (where $0 \leq i <$ size) is:

$$\text{send\_to} = (\text{rank} + i) \pmod{\text{size}}$$
$$\text{receive\_from} = (\text{rank} - i + \text{size}) \pmod{\text{size}}$$

This ensures that over `size` iterations, every process sends exactly one message to every other process (and itself) and receives exactly one message from every other process.

1.2 Deadlock Avoidance Strategy

A naive implementation where every process calls `MPI_Send` followed by `MPI_Recv` in the same order would lead to a circular dependency (Deadlock), as all processes would block waiting for their send buffer to be read.

To solve this without using `MPI_Sendrecv` or non-blocking calls (`MPI_Isend`), I implemented an ordered handshake mechanism based on process ranks:

- **Case 1 (Self):** If `send_to == rank`, a local memory copy is performed directly.
- **Case 2 (Rank > Source):** If the current process rank is greater than the source rank, it performs **Send then Receive**.
- **Case 3 (Rank < Source):** If the current process rank is lower than the source rank, it performs **Receive then Send**.

This logic breaks the dependency cycle. For any pair of communicating processes, one will always enter the receive state first, allowing the other to complete its send, thus preventing deadlock.

2 Build and Execution

The project was compiled and run on the `gandalf.sabanciuniv.edu` cluster.

2.1 Compilation

The code is compiled using the standard MPI C++ wrapper with optimization level 3:

```
mpicxx main.cpp -O3 -o all2all
```

2.2 Running

The executable is run using `mpirun` for various process counts ($P \in \{2, 4, 8, 16, 60\}$), as requested:

```
mpirun -np 2  ./all2all
mpirun -np 4  ./all2all
mpirun -np 8  ./all2all
mpirun -np 16 ./all2all
mpirun -np 60 ./all2all
```

3 Performance Results

The following tables summarize the runtime comparison between the optimized library implementation (`MPI_Alltoall`) and my custom implementation (`Custom`).

3.1 Test Case 1: 2 Processes

Msg Size	Baseline (ms)	Custom (ms)	Ratio (Cust/Base)	Result
1	**0.0025**	**0.0009**	**0.36**	**PASS**
128	0.0014	0.0015	1.04	PASS
4,096	0.0087	0.0124	1.42	PASS
100,000	0.1053	0.2166	2.05	PASS
500,000	0.5214	1.0534	2.02	PASS

Table 1: Performance for NP=2. **Custom is nearly 3x faster** for the smallest message size.

3.2 Test Case 2: 4 Processes

Msg Size	Baseline (ms)	Custom (ms)	Ratio (Cust/Base)	Result
1	**0.012**	**0.007**	**0.59**	**PASS**
128	0.009	0.014	1.52	PASS
4,096	0.049	0.096	1.97	PASS
100,000	0.623	1.241	1.99	PASS
500,000	2.253	4.314	1.91	PASS

Table 2: Performance for NP=4. Custom remains faster for very small messages.

3.3 Test Case 3: 8 Processes

Msg Size	Baseline (ms)	Custom (ms)	Ratio (Cust/Base)	Result
1	**0.063**	**0.020**	**0.32**	**PASS**
128	0.027	0.043	1.58	PASS
4,096	0.125	0.376	3.00	PASS
100,000	1.305	3.886	2.98	PASS
500,000	7.386	30.448	4.12	PASS

Table 3: Performance for NP=8. Significant speedup observed at minimal size.

3.4 Test Case 4: 16 Processes

Msg Size	Baseline (ms)	Custom (ms)	Ratio (Cust/Base)	Result
1	0.036	0.108	3.00	PASS
128	**0.123**	**0.115**	**0.93**	**PASS**
4,096	0.298	1.207	4.04	PASS
100,000	4.948	16.736	3.38	PASS
500,000	23.757	65.631	2.76	PASS

Table 4: Performance for NP=16. Custom is competitive/faster at medium-small sizes.

3.5 Test Case 5: 60 Processes

Msg Size	Baseline (ms)	Custom (ms)	Ratio (Cust/Base)	Result
1	0.231	0.677	2.93	PASS
128	0.524	0.741	1.41	PASS
4,096	2.134	10.436	4.88	PASS
100,000	51.695	107.905	2.08	PASS
500,000	242.220	489.036	2.01	PASS

Table 5: Performance for NP=60. The ratio stabilizes around 2.0x for large messages.

4 Discussion

4.1 Performance Analysis

The results demonstrate distinct performance characteristics based on process count and message size.

- **Low Process Counts ($NP \leq 8$):** The custom implementation significantly outperforms the MPI library for minimal message sizes (1 integer). For example, at $NP = 8$, the custom code is roughly **3x faster** (Ratio 0.32), and at $NP = 2$, it is similarly dominant (Ratio 0.36). This suggests that the optimized collective algorithm in MPI has a higher initialization overhead that dominates when communication volume is trivial.

- **Medium Scale ($NP = 16$):** At 16 processes, the advantage for minimal messages disappears, but surprisingly, at a message size of 128 integers, the custom code again outperforms the baseline (Ratio 0.93).

- **Large Scale/Message ($NP = 60$):** As message size increases (100k - 500k ints), the network bandwidth becomes the bottleneck. The custom implementation stabilizes at roughly **2x the runtime** of the optimized baseline. This is an expected result, as blocking send/recv calls cannot fully utilize the network fabric's parallel capabilities or overlap computation with communication as effectively as the non-blocking or topology-aware algorithms used by `MPI_Alltoall`.

4.2 Pros and Cons of the Approach

Positives:

- **Low Overhead for Small Data:** As shown in the $NP = 2, 4, 8$ cases, the simple loop avoids the heavy setup costs of complex collective algorithms, making it faster for trivial payloads.
- **Simplicity & Correctness:** The logic is easy to reason about and the rank-based ordering guarantees deadlock freedom without needing auxiliary memory or request tracking.

Drawbacks:

- **Blocking Latency:** Using blocking `MPI_Send/Recv` forces sequential execution, preventing the overlap of communication steps.
- **Scalability Limits:** While efficient for small P, the linear nature of the algorithm means it scales worse than logarithmic algorithms used in optimized libraries for very high P.