Practical Work 3: MPI File Transfer System

Nguyen Tuan Dung
Student ID: 23BI14113

December 3, 2025

1 Introduction

This report documents the implementation of a file transfer system using the **Message Passing Interface (MPI)** standard, specifically utilizing the `mpi4py` library in Python. The approach shifts the communication model from the previous Client/Server or RPC paradigm to a **Peer-to-Peer** message passing paradigm suitable for parallel execution, where two distinct processes (Rank 0 and Rank 1) coordinate the transfer.

2 MPI File Transfer Design

2.1 Design Rationale

Unlike RPC, MPI does not have inherent Client/Server roles. Instead, it uses **Ranks** to define the task of each parallel process.

- **Rank 0 (Sender):** Responsible for reading the file from the disk and sending metadata and data chunks.
- **Rank 1 (Receiver):** Responsible for receiving the data and assembling the file on the disk.

2.2 Communication Protocol (Explicit Message Passing)

We use Point-to-Point communication routines, distinguished by their **Tag** value:

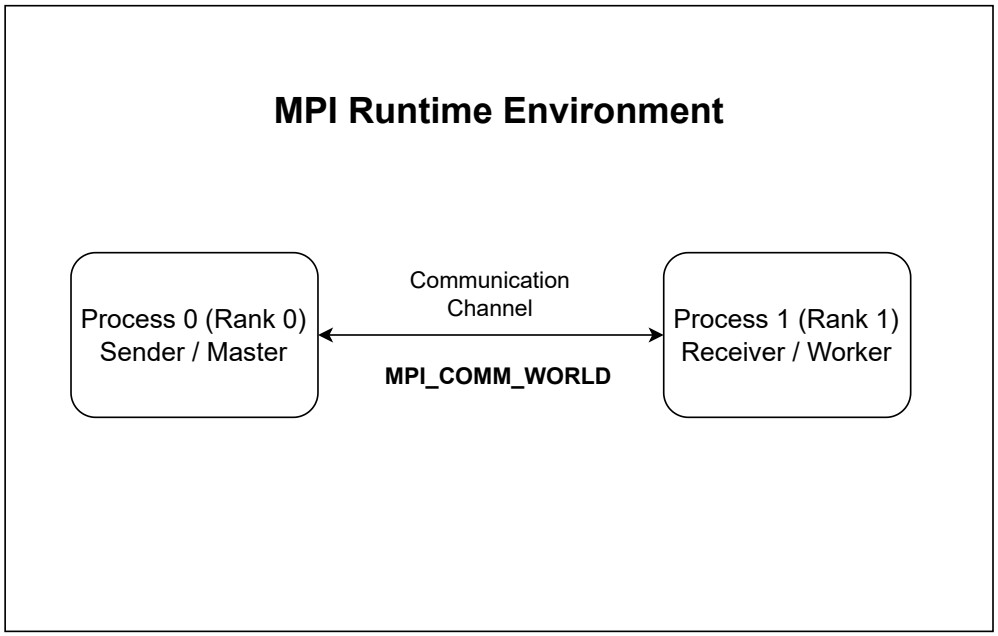
Table 1: MPI Communication Protocol

Step	Sender	Receiver	Tag	Content
1	`comm.send`	`comm.recv`	1	Filename (string)
2	`comm.send`	`comm.recv`	2	Filesize (int)
3	`comm.recv`	`comm.send`	3	Acknowledgment ("READY")
4 (Loop)	`comm.send`	`comm.recv` / `Probe`	4	File Chunk (bytes)
5 (End)	`comm.send`	`comm.recv`	5	Termination Signal (`None`)

3 System Organization and Execution

3.1 System Organization

The entire system is contained within a single program (`mpi_file_transfer.py`) executed by the MPI runtime environment.



3.2 Who Does What

- **MPI Runtime (`mpiexec -n 2`):** Spawns two instances of the program and initializes the communication context (`MPI_COMM_WORLD`).
- **Process 0 (Sender):** Executes the `sender_process` function. Reads file data and uses explicit `comm.send()` calls to push data packets to Process 1.
- **Process 1 (Receiver):** Executes the `receiver_process` function. Uses explicit `comm.recv()` calls to poll for incoming messages and writes them to the output file.

4 Implementation of File Transfer

The implementation replaces the complex socket setup and teardown with the simple, explicit MPI communication commands.

4.1 Sender Side Implementation (Data Transfer)

The sender uses the Rank and Tag to ensure the data reaches the correct destination process.

Listing 1: MPI Sender: Sending Metadata and Data Chunks (Rank 0)

```
1
2  # 1. Get communicator object
3  comm = MPI.COMM_WORLD
4  SENDER_RANK = 0 # This process is Rank 0
5
6  # 2. Send Metadata
7  comm.send(filename, dest=RECEIVER_RANK, tag=1)
8  comm.send(filesize, dest=RECEIVER_RANK, tag=2)
9
10 # 3. Send Data Loop
11 with open(filepath, 'rb') as f:
12     while True:
13         chunk = f.read(BUFFER_SIZE)
14         if not chunk:
```

```
            break

        # Explicitly send the chunk (MPI handles serialization of bytes)
        comm.send(chunk, dest=RECEIVER_RANK, tag=4)

# 4. Send termination signal
comm.send(None, dest=RECEIVER_RANK, tag=5)
```

4.2 Receiver Side Implementation (Data Reception)

The receiver must know the source process (Rank 0) and uses tags to determine the type of incoming message.

Listing 2: MPI Receiver: Receiving Data Chunks (Rank 1)

```
# 1. Get communicator object
comm = MPI.COMM_WORLD
RECEIVER_RANK = 1 # This process is Rank 1

# 2. Receive Metadata
filename = comm.recv(source=SENDER_RANK, tag=1)

# 3. Receive Data Loop
with open(output_filename, 'wb') as f:
    while bytes_received < filesize:

        # Use Probe to check for incoming message and tag
        status = MPI.Status()
        comm.Probe(source=SENDER_RANK, tag=MPI.ANY_TAG, status=status)
        incoming_tag = status.Get_tag()

        if incoming_tag == 5: # Termination signal
            comm.recv(source=SENDER_RANK, tag=5)
            break

        # Receive the chunk (blocking until data is available)
        chunk = comm.recv(source=SENDER_RANK, tag=4)
        f.write(chunk)
        bytes_received += len(chunk)
```

5 Conclusion

The MPI implementation successfully achieved file transfer using a parallel, peer-to-peer communication model. This architecture is distinctly different from both TCP (low-level explicit protocol) and RPC (high-level function abstraction) in that it requires explicit communication calls but benefits from the highly optimized data handling inherent to the MPI standard. The core logic relies on differentiating roles based on `rank` and using `tag` values to order the communication sequence.