Practical Work 3: MPI File Transfer System

Group ID: 18

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 within the `MPI_COMM_WORLD` communicator.

- **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

We use **Point-to-Point communication** routines, distinguished by their **Tag** value to manage the flow of information. The protocol follows a sequential message passing strategy:

The table below outlines the specific operations and content for each step of the MPI Communication Protocol.

Operation	Tag	Step	Content
comm.send / comm.recv	1	Metadata	Metadata (Filename, Filesize)
comm.send / comm.recv	2	Data Chunk	Data Chunk (Binary Bytes)
comm.send / comm.recv	2	Termination Signal	Termination Signal (Empty bytes)

3 System Organization and Execution

3.1 System Organization

The entire system is contained within a single program (`mpi_transfer.py`) executed by the MPI runtime environment. The environment spawns multiple instances of the program which communicate over a channel abstracted by the MPI standard.

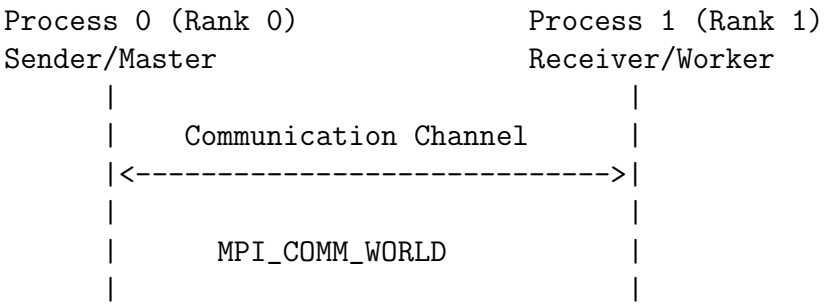

Figure 1: System Organization Diagram

3.2 Who Does What

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

4 Implementation of File Transfer

The implementation replaces complex socket setup with 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. It first packages the filename and size into a dictionary for the metadata step.

```
# 1. Send Metadata (Tag 1)
metadata = {'filename': os.path.basename(filename), 'filesize':
    filesize}
comm.send(metadata, dest=dest_rank, tag=1)

# 2. Send Data Loop (Tag 2)
with open(filename, 'rb') as f:
    while True:
        chunk = f.read(CHUNK_SIZE)
```

```
        if not chunk:
            break
        # Explicitly send the chunk
        comm.send(chunk, dest=dest_rank, tag=2)

# 3. Send termination signal
comm.send(b'', dest=dest_rank, tag=2)
```

Listing 1: MPI Sender (Rank 0)

4.2 Receiver Side Implementation (Data Reception)

The receiver waits for the metadata to prepare the output file, then enters a loop to receive chunks until the EOF signal (empty bytes) is detected.

```
# 1. Receive Metadata (Tag 1)
metadata = comm.recv(source=source_rank, tag=1)
filename = "recv_" + metadata['filename']

# 2. Receive Data Loop (Tag 2)
with open(filename, 'wb') as f:
    while True:
        # Blocking receive for data chunks
        chunk = comm.recv(source=source_rank, tag=2)
        # Check for termination signal (empty bytes)
        if not chunk:
            break
        f.write(chunk)
```

Listing 2: MPI Receiver (Rank 1)

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.