Practical Work 3: MPI File Transfer

Nguyen The Xuan

December 5, 2025

1 Introduction

The objective of this practical work is to implement a file transfer system using the Message Passing Interface (MPI) standard. Unlike the previous client-server architecture using sockets or RPC, MPI allows for parallel processing where distinct processes communicate by sending and receiving messages directly within a communicator.

2 MPI Implementation Choice

We chose **mpi4py** (MPI for Python) as our implementation.

- **Python Integration:** It provides an object-oriented approach to MPI that fits naturally with Python, allowing the transmission of arbitrary Python objects (via pickling) and binary data (buffers).
- **Standard Compliance:** It wraps the standard MPI-2 implementations (like MPICH or Open-MPI), ensuring that the core concepts (Communicators, Ranks, Tags) are consistent with the theoretical course material.
- **Ease of Use:** It abstracts low-level memory management while maintaining high performance for data transfer.

3 Service Design

The system is designed as a single program that behaves differently based on the process Rank.

- **Rank 0 (Sender):** Reads the file and sends it in chunks.
- **Rank 1 (Receiver):** Listens for data and reconstructs the file.

The communication flow relies on blocking `send` and `recv` calls with specific **Tags** to distinguish between the filename, file content, and the end-of-transmission signal.

4 System Organization

The system consists of a single script, `mpi_file_transfer.py`. When executed via `mpiexec`, the MPI runtime spawns multiple instances of this script.

- **Process Separation:** Logic is separated using `if rank == 0` and `elif rank == 1`.
- **Tags:** We defined constants (`TAG_FILENAME`, `TAG_DATA`, `TAG_END`) to synchronize the state between sender and receiver.

5 Implementation

Below are the code snippets showing how we differentiate roles and handle data transmission.

5.1 Differentiating Roles

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    sender(filename, dest_rank=1)
elif rank == 1:
    receiver(source_rank=0)
```

Listing 1: Role Assignment by Rank

5.2 Sending Data (Rank 0)

```
with open(filename, 'rb') as f:
    while True:
        chunk = f.read(CHUNK_SIZE)
        if not chunk:
            break
        comm.send(chunk, dest=dest_rank, tag=TAG_DATA)
comm.send(None, dest=dest_rank, tag=TAG_END)
```

Listing 2: Sender Loop

5.3 Receiving Data (Rank 1)

```
while True:
    status = MPI.Status()
    # Check incoming message before receiving to know the tag
    comm.probe(source=source_rank, tag=MPI.ANY_TAG, status=status)
    tag = status.Get_tag()

    if tag == TAG_DATA:
        chunk = comm.recv(source=source_rank, tag=TAG_DATA)
        f.write(chunk)
    elif tag == TAG_END:
        comm.recv(source=source_rank, tag=TAG_END)
        break
```

Listing 3: Receiver Loop with Probe

6 Experimental Results

We executed the transfer using the command:

`mpiexec -n 2 python mpi_file_transfer.py image.jpg`

1. **Setup:** Ubuntu environment with `mpich` and `mpi4py` installed.
2. **Observation:** Process 0 printed "Sending data..." and Process 1 printed "Saving to: received_image.jpg".
3. **Result:** The file was successfully duplicated without corruption.