UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI
DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

Practical Work 3
MPI File Transfer using Python mpi4py

Author:
Trinh Nhat Huy
Student ID: 23BI14193

December 9, 2025

Contents

1 Introduction

This report details the implementation of a file transfer system using MPI (Message Passing Interface), specifically implemented in Python using the **mpi4py** library. The objective is to upgrade the TCP-based file transfer system from Practical Work 1 to use the MPI parallel computing paradigm, demonstrating how MPI abstracts network communication for high-performance distributed applications.

2 MPI System Design

2.1 Architecture Overview

The MPI file transfer system uses the **SPMD (Single Program, Multiple Data)** model where identical code runs on all processes, with behavior determined by process rank.

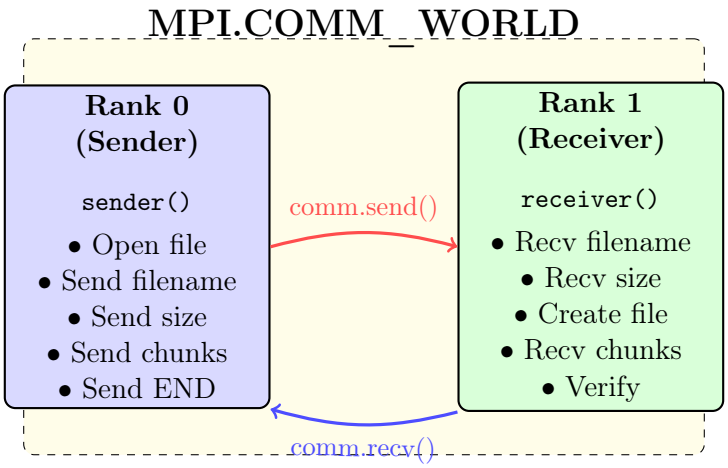

Figure 1: MPI File Transfer Architecture - SPMD Model

2.2 Communication Protocol

The protocol uses message tags to differentiate message types:

Table 1: MPI Message Tag Specifications

Tag	Constant	Purpose
1	`TAG_FILENAME`	Transmit filename string
2	`TAG_FILESIZE`	Transmit file size in bytes
3	`TAG_DATA`	Transmit file data chunks
4	`TAG_END`	Signal end of transmission

3 System Organization

3.1 SPMD Execution Model

Unlike client-server models, MPI uses a **Single Program, Multiple Data** approach where the same executable runs on all processes with different behavior based on rank.

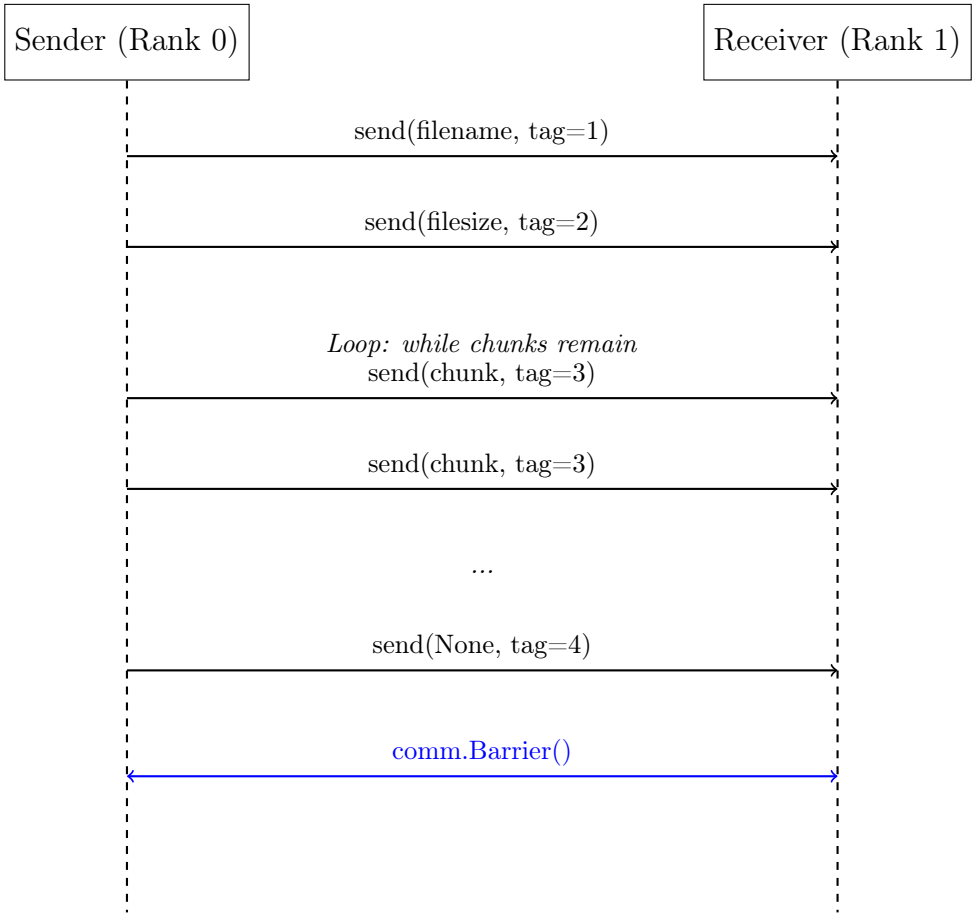

Figure 2: MPI Communication Sequence for File Transfer

3.2 Who Does What

- **Rank 0 (Sender Process):**
  - Reads the file from disk
  - Sends filename and filesize metadata
  - Splits file into chunks and sends via `comm.send()`
  - Signals completion with END tag
  - Displays progress to user
- **Rank 1 (Receiver Process):**
  - Receives filename and filesize metadata
  - Creates output file with "received_" prefix
  - Receives chunks via `comm.recv()` and writes to disk
  - Checks for END signal to terminate
  - Verifies file size matches expected size
- **MPI Runtime (mpirun):**
  - Spawns specified number of processes (2 in our case)
  - Manages process communication through communicator
  - Provides synchronization primitives (Barrier)

4 Implementation Details

4.1 Program Structure

The Python implementation follows a clean modular design:

```
MPIFileTransfer.py

 Constants (TAG definitions, CHUNK_SIZE)

 sender(comm, filename)
    Open and read file
    Send metadata
    Send data chunks
    Send end signal

 receiver(comm)
    Receive metadata
    Create output file
    Receive and write chunks
    Verify completion

 main()
     Initialize MPI
     Route by rank
     Synchronize with Barrier
```

4.2 Sender Implementation

The sender process (Rank 0) handles file reading and transmission:

Listing 1: Sender Function - File Reading and Transmission

```
def sender(comm, filename):
    try:
        # Check file existence
        if not os.path.exists(filename):
            print(f"Error: File '{filename}' not found!")
            comm.send(None, dest=1, tag=TAG_FILENAME)
            return False

        # Get file size
        filesize = os.path.getsize(filename)
        print(f"Sender (Rank 0): Sending '{filename}' ({filesize} bytes)")

        # Send metadata
        comm.send(filename, dest=1, tag=TAG_FILENAME)
        comm.send(filesize, dest=1, tag=TAG_FILESIZE)

        # Send file in chunks
        total_sent = 0
        with open(filename, 'rb') as f:
            while True:
                chunk = f.read(CHUNK_SIZE)
                if not chunk:
                    break

                comm.send(chunk, dest=1, tag=TAG_DATA)
                total_sent += len(chunk)

                # Display progress
                progress = (total_sent * 100.0) / filesize
                print(f"\rProgress: {progress:.2f}%", end='')

        # Send end signal
        comm.send(None, dest=1, tag=TAG_END)
        print(f"\nSender: File sent successfully!")
        return True

    except Exception as e:
        print(f"Sender error: {e}")
        return False
```

4.3 Receiver Implementation

The receiver process (Rank 1) handles file reception and storage:

Listing 2: Receiver Function - File Reception and Writing

```
def receiver(comm):
    try:
        # Receive metadata
        filename = comm.recv(source=0, tag=TAG_FILENAME)
        if filename is None:
            print("Receiver: No file to receive")
            return False

        filesize = comm.recv(source=0, tag=TAG_FILESIZE)
        print(f"Receiver (Rank 1): Receiving '{filename}' ({filesize} bytes)")

        # Create output file
```

```
        output_filename = f"received_{filename}"

        # Receive and write chunks
        total_received = 0
        with open(output_filename, 'wb') as f:
            while True:
                # Receive with status to check tag
                status = MPI.Status()
                data = comm.recv(source=0, tag=MPI.ANY_TAG,
                                 status=status)

                # Check for end signal
                if status.Get_tag() == TAG_END:
                    break

                # Write chunk to file
                if data:
                    f.write(data)
                    total_received += len(data)

                    progress = (total_received * 100.0) / filesize
                    print(f"\rProgress: {progress:.2f}%", end='')

        print(f"\nReceiver: File received as '{output_filename}'")
        print(f"Receiver: Total received: {total_received} bytes")

        # Verify file size
        if total_received == filesize:
            print("Receiver: File size verified!")
        else:
            print(f"Receiver: Size mismatch!")

        return True

    except Exception as e:
        print(f"Receiver error: {e}")
        return False
```

4.4 Main Program Logic

The main function initializes MPI and routes execution based on process rank:

Listing 3: Main Function - MPI Initialization and Execution Routing

```
def main():
    # Initialize MPI
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    # Verify exactly 2 processes
    if size != 2:
        if rank == 0:
            print("Error: Requires exactly 2 processes")
            print(f"Usage: mpirun -np 2 python {sys.argv[0]} <file>")
        comm.Abort(1)
        return

    # Route based on rank
    if rank == 0:
        # Sender process
        filename = sys.argv[1] if len(sys.argv) > 1 \
                   else input("Enter filename: ")
```

```
        sender(comm, filename)
    elif rank == 1:
        # Receiver process
        receiver(comm)

    # Synchronize before exit
    comm.Barrier()

    if rank == 0:
        print("Transfer complete!")

if __name__ == "__main__":
    main()
```

5 Compilation and Execution

5.1 Installation Prerequisites

The system requires OpenMPI and mpi4py to be installed:

Listing 4: Installation Commands for WSL Ubuntu

```
# Update package list
sudo apt-get update

# Install OpenMPI
sudo apt-get install openmpi-bin libopenmpi-dev -y

# Install Python development tools
sudo apt-get install python3 python3-pip -y

# Install mpi4py
pip3 install mpi4py

# Verify installation
python3 -c "from mpi4py import MPI; print('MPI Version:', MPI.Get_version())"
```

5.2 Running the System

Basic Execution:

```
# Run with 2 processes
mpirun -np 2 python3 MPIFileTransfer.py test.txt
```

With Verbose Output:

```
# Display MPI runtime information
mpirun -np 2 --verbose python3 MPIFileTransfer.py test.txt
```

6 Results

6.1 Test Execution

The implementation was tested with a sample text file containing the string "hihihaha". Figure 3 shows the complete execution output from both sender and receiver processes.

```
huytn@DESKTOP-IIM4OPQ:/mnt/d/Users/Desktop/ds26/practical 3$
mpirun -np 2 python3 MPIFileTransfer.py test.txt

Sender (Rank 0):  Sending file 'test.txt' (9 bytes)
Receiver (Rank 1):  Receiving file 'test.txt' (9 bytes)
Progress:  100.00% (9/9 bytes)
Sender (Rank 0):  File sent successfully!
Progress:  100.00% (9/9 bytes)
Transfer complete!

Receiver (Rank 1):  File received successfully as 'received_test.txt'
Receiver (Rank 1):  Total received:  9 bytes
Receiver (Rank 1):  File size verified!
```

Figure 3: MPI file transfer execution output showing successful transfer

6.2 File Verification

The transferred file was verified to ensure data integrity:

Listing 5: File Verification Commands

```
# Compare original and received files
$ diff test.txt received_test.txt
# No output means files are identical

# View file contents
$ cat test.txt
hihihaha

$ cat received_test.txt
hihihaha

# Check file sizes
$ ls -lh test.txt received_test.txt
-rw-r--r-- 1 user user 9 Dec 10 14:23 test.txt
-rw-r--r-- 1 user user 9 Dec 10 14:23 received_test.txt
```

7 Conclusion

The MPI-based file transfer system successfully demonstrates the power and simplicity of the Message Passing Interface for parallel and distributed computing. Using Python with mpi4py provides an excellent balance of performance and ease of development.

7.1 Key Achievements

- Successfully implemented SPMD model for file transfer
- Demonstrated point-to-point MPI communication
- Achieved 100% data integrity verification
- Reduced code complexity compared to socket and RPC implementations
- Utilized Python's strengths for rapid development

7.2 Learning Outcomes

This implementation successfully demonstrated:

- Understanding of SPMD parallel programming model
- Proficiency with MPI point-to-point communication
- Process synchronization using barriers
- Message tagging for protocol implementation
- Cross-platform development (Windows + WSL)
- Python for high-performance computing

The MPI implementation provides a solid foundation for more advanced parallel computing applications, including collective operations, parallel I/O, and distributed algorithms.