University Of Science And Technology Of Hanoi

ICT Department

LAB REPORT

Title: Practical Work 3 – MPI File Transfer

Subject: Distributed Systems (DS2026)

Student: Le Nam Anh

StudentID: 23BI14029

Major: Cyber Security

1 Introduction

Following the exploration of Socket (Lab 1) and RPC (Lab 2), this practical work aims to implement a file transfer system using the **Message Passing Interface (MPI)**.

MPI is a standardized and portable message-passing standard designed to function on parallel computing architectures. The goal is to migrate the file transfer logic to an MPI-based environment, where distinct processes communicate by sending and receiving messages rather than establishing direct socket connections. The system is implemented in Python using the `mpi4py` library.

2 Implementation Choice

For this project, I selected **mpi4py (MPI for Python)** as the implementation wrapper. The reasons for this selection are:

- **Python Integration:** It provides an object-oriented approach to message passing which integrates seamlessly with Python structures.
- **Serialization:** Unlike C-based MPI which requires strict data typing, `mpi4py` automatically handles the serialization (pickling) of arbitrary Python objects (such as dictionaries for metadata), simplifying the development process.
- **Standard Compliance:** It is built on top of standard MPI implementations (like MS-MPI on Windows or OpenMPI on Linux), ensuring high performance and compliance with the MPI specifications.

3 System Design

3.1 Design Principle: SPMD

The system follows the **SPMD (Single Program, Multiple Data)** model. A single script (`mpi_transfer.py`) is executed, but it spawns multiple independent processes. The behavior of each process is determined by its **Rank** (ID) within the `MPI_COMM_WORLD` communicator.

3.2 Roles and Protocol

- **Rank 0 (Sender):** Acts as the source. It reads the file and initiates Point-to-Point communication.
- **Rank 1 (Receiver):** Acts as the destination. It waits for messages and writes data to the disk.

Communication Flow: The transfer relies on MPI Tags to distinguish between message types:

- **Metadata (Tag 1):** Rank 0 sends a Python dictionary containing `{'filename': ..., 'filesize': ...}`.

- **Data Chunks (Tag 2):** Rank 0 reads the file in 4KB chunks and sends them sequentially.
- **Termination (Tag 2):** Rank 0 sends a `None` object to signal End-Of-File (EOF).

4 System Organization

The system operates within the MPI global communicator:

- **Sender (Rank 0):** Uses `comm.send()` to push data.
- **Receiver (Rank 1):** Uses `comm.recv()` to pull data.
- **Synchronization:** The protocol is synchronous; the receiver blocks until data arrives.

5 Implementation Details

5.1 Environment Setup

Before implementing the logic, the development environment required specific configurations to support MPI on the Windows operating system. The setup process involved two main steps executed via the command line:

1. **Installing the MPI Backend:** First, the Microsoft MPI (MS-MPI) library was installed to provide the necessary runtime and SDK for parallel processing. This was done using the Windows Package Manager:

   ```
   winget install Microsoft.MSMPI
   ```

2. **Installing Python Bindings:** Once the backend was established, the `mpi4py` library was installed to allow Python scripts to interface with the MPI runtime:

   ```
   pip install mpi4py
   ```

This configuration ensures that the Python interpreter can successfully link against the system's MPI binaries to manage parallel processes.

5.2 Code Implementation

The core logic is implemented in a single Python script (`mpi_transfer.py`), leveraging the `mpi4py` library. Below are the critical code segments handling the logic based on process ranks. (`mpi_transfer.py`)

```
1 from mpi4py import MPI
2 import os
3 import sys
4
5 UPLOAD_DIR = "received_files"
6 CHUNK_SIZE = 4096
7 TAG_METADATA = 1
8 TAG_DATA = 2
```

```
9
10 def run_sender(filename, dest_rank, comm):
11     if not os.path.isfile(filename):
12         print(f"[Sender] Error: File '{filename}' not found.")
13         comm.send(None, dest=dest_rank, tag=TAG_METADATA)
14         return
15
16     file_basename = os.path.basename(filename)
17     filesize = os.path.getsize(filename)
18     metadata = {'filename': file_basename, 'filesize': filesize}
19
20     print(f"[Sender] Sending metadata: {metadata} to Rank {dest_rank}")
21     comm.send(metadata, dest=dest_rank, tag=TAG_METADATA)
22     print(f"[Sender] Transmission started...")
23     sent_bytes = 0
24     with open(filename, 'rb') as f:
25         while True:
26             chunk = f.read(CHUNK_SIZE)
27             if not chunk:
28                 break
29
30             comm.send(chunk, dest=dest_rank, tag=TAG_DATA)
31             sent_bytes += len(chunk)
32
33     comm.send(None, dest=dest_rank, tag=TAG_DATA)
34     print(f"[Sender] Transfer complete. Total sent: {sent_bytes} bytes."
       )
35
36
37 def run_receiver(source_rank, comm):
38     print(f"[Receiver] Waiting for connection from Rank {source_rank}...
       ")
39     metadata = comm.recv(source=source_rank, tag=TAG_METADATA)
40
41     if metadata is None:
42         print("[Receiver] Error: Sender aborted the operation.")
43         return
44
45     filename = metadata['filename']
46     filesize = metadata['filesize']
47
48     if not os.path.exists(UPLOAD_DIR):
49         os.makedirs(UPLOAD_DIR)
50
51     dest_path = os.path.join(UPLOAD_DIR, filename)
52     print(f"[Receiver] Incoming file: {filename} ({filesize} bytes)")
53
54     received_bytes = 0
55     with open(dest_path, 'wb') as f:
56         while True:
57             chunk = comm.recv(source=source_rank, tag=TAG_DATA)
58             if chunk is None:
59                 break
60
61             f.write(chunk)
62             received_bytes += len(chunk)
63
64     print(f"[Receiver] File saved to: {dest_path}")
```

```
65      print(f"[Receiver] Verification: {received_bytes}/{filesize} bytes
        received.")
66
67  if __name__ == "__main__":
68      comm = MPI.COMM_WORLD
69      rank = comm.Get_rank()
70      size = comm.Get_size()
71
72      if size < 2:
73          if rank == 0:
74              print("Error: This program requires at least 2 MPI processes
        .")
75              print("Usage: mpiexec -n 2 python mpi_transfer.py <filename>
        ")
76          sys.exit(1)
77
78      if rank == 0:
79          if len(sys.argv) < 2:
80              print("Usage: mpiexec -n 2 python mpi_transfer.py <filename>
        ")
81              comm.send(None, dest=1, tag=TAG_METADATA)
82          else:
83              filename = sys.argv[1]
84              run_sender(filename, dest_rank=1, comm=comm)
85
86      elif rank == 1:
87          run_receiver(source_rank=0, comm=comm)
```

Listing 1: MPI Transfer Script

6 Testing and Results

The validation of the MPI-based file transfer system was conducted on a local Windows environment using PowerShell. The testing process involved spawning multiple processes to simulate a parallel computing environment.

Process Spawning: The `mpiexec` command was used to launch the application with 2 separate processes (`-n 2`). The target file for transmission was specified as a command-line argument:

```
mpiexec -n 2 python mpi_transfer.py test.txt
```

Sender (Rank 0) Initialization: Upon startup, the process assigned Rank 0 assumed the role of the Sender. The terminal output indicated that it successfully located the file `test.txt` (size: 214 bytes) and transmitted the metadata dictionary to Rank 1. It then proceeded to stream the file content.

Receiver (Rank 1) Operation: Simultaneously, the process with Rank 1 initialized in the waiting state. The logs confirm that it successfully received the metadata and prepared the destination path. Subsequently, it received the binary data stream and wrote it to the `received_files\test.txt` location.

Completion and Verification: The logs from both processes confirmed a successful synchronization:

- **Sender:** Reported "Transfer complete. Total sent: 214 bytes".

- **Receiver:** Confirmed "Verification: 214/214 bytes received".

This exact match in byte count verifies that the file was transmitted without data loss or corruption.

```
PS D:\GrindingNal\USTH\DistributedSystem\ds2026\Practical 3> mpiexec -n 2 python mpi_transfer.py test.txt
[Sender] Sending metadata: {'filename': 'test.txt', 'filesize': 214} to Rank 1
[Sender] Transmission started...
[Sender] Transfer complete. Total sent: 214 bytes.
[Receiver] Waiting for connection from Rank 0...
[Receiver] Incoming file: test.txt (214 bytes)
[Receiver] File saved to: received_files\test.txt
[Receiver] Verification: 214/214 bytes received.
PS D:\GrindingNal\USTH\DistributedSystem\ds2026\Practical 3>
```

Figure 1: MPI Execution and Result Verification