Practical Work 3: MPI File Transfer

Doan Dinh Khai - 22BA13167

December 24, 2025

MPI Implementation Choice

For this practical work, we chose to use `mpi4py`, which is a Python binding for the Message Passing Interface (MPI) standard. The underlying MPI implementation can be OpenMPI, MPICH, or MS-MPI depending on the operating system.

Operating System and Environment: The implementation was developed on Windows 10 using Python 3.x. The system requires an MPI runtime library (such as MS-MPI for Windows or OpenMPI/MPICH for Linux) to be installed on the system.

Installation Method: The `mpi4py` package can be installed using pip:

```
pip install mpi4py
```

On Windows, MS-MPI must be installed separately from Microsoft's website. On Linux, the MPI runtime is typically available through package managers (e.g., `apt-get install openmpi-bin` or `apt-get install mpich`).

Advantages:

- `mpi4py` provides a clean Python interface to MPI, making it easy to prototype and develop MPI applications.
- It supports both point-to-point and collective communication operations.
- Python's built-in data structures (dictionaries, lists) can be directly sent via MPI, simplifying metadata exchange.
- Cross-platform compatibility when using standard MPI implementations.

Limitations:

- Performance overhead compared to native C/C++ MPI implementations due to Python's interpreted nature.
- For very large files, loading the entire file into memory may not be feasible; chunked transfer would be more appropriate.
- Requires both MPI runtime and Python environment to be properly configured.

MPI Service Design

Our MPI file transfer system uses a simple two-process architecture:

Rank Roles:

- **Rank 0 (Sender):** Reads a file from the local filesystem and sends it to rank 1 using MPI point-to-point communication.

- **Rank 1 (Receiver):** Receives the file metadata and data from rank 0, then writes it to disk with a prefixed filename.

Metadata Exchange: The system exchanges metadata as a Python dictionary containing:

- `name`: The original filename (basename only).
- `size`: The file size in bytes.

This metadata is sent first using tag 0, allowing the receiver to prepare for the incoming file transfer.

File Transfer Strategy: The current implementation sends the entire file as a single message after loading it into memory. This approach is simple and works well for small to medium-sized files. For production use with large files, a chunked transfer strategy would be more memory-efficient, sending the file in fixed-size blocks (e.g., 64KB or 1MB chunks) and using a loop to transfer all chunks sequentially.

Data Flow:

1. Rank 0 reads the file from disk into memory.
2. Rank 0 sends metadata dictionary to rank 1 (tag 0).
3. Rank 1 receives metadata and prepares for file reception.
4. Rank 0 sends file bytes to rank 1 (tag 1).
5. Rank 1 receives file bytes and writes to disk as `MPI_RECV_<filename>`.

System Organization

Directory Structure: The project is organized as follows:

```
ds2026-main/
  practice_3/
    mpi_file_transfer.py
    03.mpi.file.transfer.tex
```

The main implementation file `mpi_file_transfer.py` contains three functions:

- `sender()`: Handles file reading and sending (rank 0).
- `receiver()`: Handles file reception and writing (rank 1).
- `main()`: Initializes MPI, checks process count, and routes execution based on rank.

Command to Start the System: The system is launched using `mpiexec` (or `mpirun` on some systems) with exactly 2 processes:

```
mpiexec -n 2 python mpi_file_transfer.py <path_to_file>
```

The file path is provided as a command-line argument and is only used by rank 0. Rank 1 automatically waits for incoming data.

Testing: Testing can be performed by:

- Creating a test file of various sizes (small text file, medium binary file, etc.).
- Running the transfer command and verifying the received file matches the original.
- Checking that the received filename is prefixed with `MPI_RECV_`.
- Verifying file integrity using checksums (e.g., MD5 or SHA256) if needed.

File Transfer Implementation

Rank 0 (Sender) Actions:

1. Validates that the file exists and is accessible.
2. Extracts the filename (basename) and file size.
3. Reads the entire file into memory as bytes.
4. Sends metadata dictionary `{"name": filename, "size": file_size}` to rank 1 using `comm.send()` with tag 0.
5. Sends the file bytes to rank 1 using `comm.send()` with tag 1.
6. Prints confirmation messages for debugging.

Rank 1 (Receiver) Actions:

1. Waits to receive metadata from rank 0 using `comm.recv()` with tag 0.
2. Extracts filename and expected file size from metadata.
3. Waits to receive file bytes from rank 0 using `comm.recv()` with tag 1.
4. Constructs output filename as `MPI_RECV_<original_filename>`.
5. Writes received bytes to disk.
6. Prints confirmation messages showing the received file path and size.

MPI Communication Primitives Used:

- `MPI.COMM_WORLD`: The default communicator containing all processes.
- `comm.rank`: The rank (process ID) of the current process (0 or 1).
- `comm.size`: Total number of processes (must be 2).
- `comm.send(obj, dest, tag)`: Sends a Python object to the destination rank with a tag for message identification.
- `comm.recv(source, tag)`: Receives a Python object from the source rank matching the specified tag.

The use of tags (0 for metadata, 1 for file data) ensures that messages are received in the correct order and prevents confusion between different message types.

Listing 1: Core MPI file transfer implementation

```python
def sender(comm: MPI.Comm, filepath: str) -> None:
    filename = os.path.basename(filepath)
    file_size = os.path.getsize(filepath)

    with open(filepath, "rb") as f:
        file_bytes = f.read()

    meta = {"name": filename, "size": file_size}
    comm.send(meta, dest=1, tag=0)
    comm.send(file_bytes, dest=1, tag=1)

def receiver(comm: MPI.Comm, output_dir: str = ".") -> None:
    meta = comm.recv(source=0, tag=0)
    filename = meta["name"]
    file_bytes = comm.recv(source=0, tag=1)

    recv_name = f"MPI_RECV_{os.path.basename(filename)}"
    output_path = os.path.join(output_dir, recv_name)

    with open(output_path, "wb") as f:
        f.write(file_bytes)
```

Group Work and Responsibilities

Design and Choice of MPI Implementation: The decision to use `mpi4py` was made collaboratively, considering the team's familiarity with Python and the need for rapid prototyping. The two-process architecture (sender/receiver) was chosen for simplicity and clarity.

Implementation of Sender and Receiver: The sender and receiver functions were implemented following the MPI point-to-point communication pattern. The code handles file I/O, metadata packaging, and MPI message passing. Error handling includes file existence checks and process count validation.

Testing and Debugging: Testing involved transferring files of various sizes and types to ensure correct data transfer. Debugging focused on verifying message ordering using tags and ensuring file integrity after transfer. Print statements were added to track the transfer progress.

Writing and Formatting the LaTeX Report: The report was written in LaTeX to provide a professional document structure. Code snippets were formatted using the `listings` package for syntax highlighting. The report covers all required sections: implementation choice, design, organization, implementation details, and group responsibilities.