USTH Master Spoiler Alert

03.mpi.file.transfer

Nguyen Quynh Trang - 22BA13303

1 Protocol Design

This practical work involved upgrading the previous file transfer system from the Client-Server (TCP/RPC) model to the **Message Passing Interface (MPI)** model. The fundamental shift is from client-server network connections to a parallel execution environment where multiple processes communicate via explicit messages.

1.1 Why MPI?

The transition to MPI is driven by the need for high-performance, low-overhead communication, typical in parallel and high-performance computing environments (as detailed in the lecture slides, slide 51):

- **Abstraction:** MPI abstracts the underlying network details (sockets, shared memory, Infiniband) into simple function calls (`MPI_Send`, `MPI_Recv`).
- **Low Complexity:** Compared to raw sockets, MPI handles connection management automatically.
- **Communication Model:** It naturally supports explicit **Point-to-Point** communication using process Ranks, which perfectly suits the sender-receiver file transfer task.

1.2 MPI Communication Protocol (Tags and Ranks)

Instead of an IDL defining procedures, MPI uses Ranks to identify processes and **Tags** to identify the message type. Our protocol defines a 3-phase communication flow between Rank 0 (Sender) and Rank 1 (Receiver).

Constant	Value	Purpose
`FILENAME_TAG`	100	Send the filename string
`DATA_TAG`	200	Send a chunk of file content
`END_TAG`	300	Signal End-of-File (EOF)

1. **Phase 1: Filename Transfer** (Tag 100).
2. **Phase 2: Data Transfer** (Tag 200) - Loop until file is fully read.
3. **Phase 3: EOF Signal** (Tag 300) - Final zero-byte message to ensure Rank 1 closes the file handle.

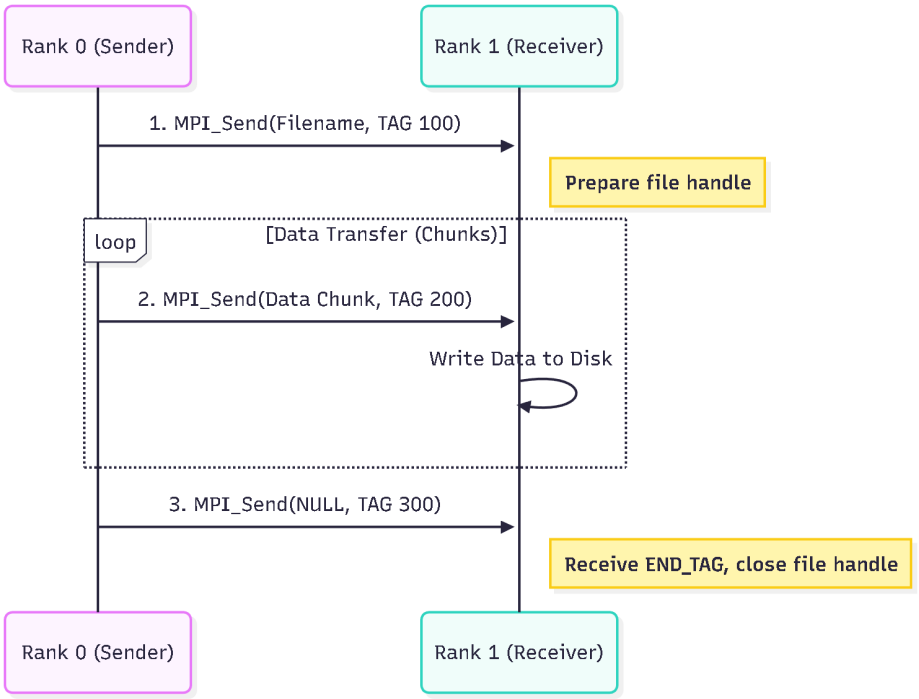

Figure 1: MPI Point-to-Point File Transfer using Tags

2 System Organization

MPI applications operate under the Single Program Multiple Data (SPMD) model. All processes execute the same binary file, but their behavior is differentiated by their assigned **Rank** within the `MPI_COMM_WORLD` communicator.

2.1 Component Roles

Component	Role	Key Mechanism	Entry Point
Rank 0	Sender (Client)	Read from Disk, `MPI_Send` (Dest: Rank 1)	`rank_0_sender` function
Rank 1	Receiver (Server)	`MPI_Recv` (Source: Rank 0), Write to Disk	`rank_1_receiver` function
MPI Runtime	Process Manager	Initialization (`MPI_Init`), Process Spawning (`mpiexec -n 2`)	OS Shell Command
Other Ranks (≥ 2)	Idle	Print status message, `MPI_Finalize`	`main` function's `else` block

2.2 Organization Flow Diagram

The architecture bypasses the need for an RPC compiler, relying entirely on the MPI library and the runtime environment spawned by `mpiexec`.

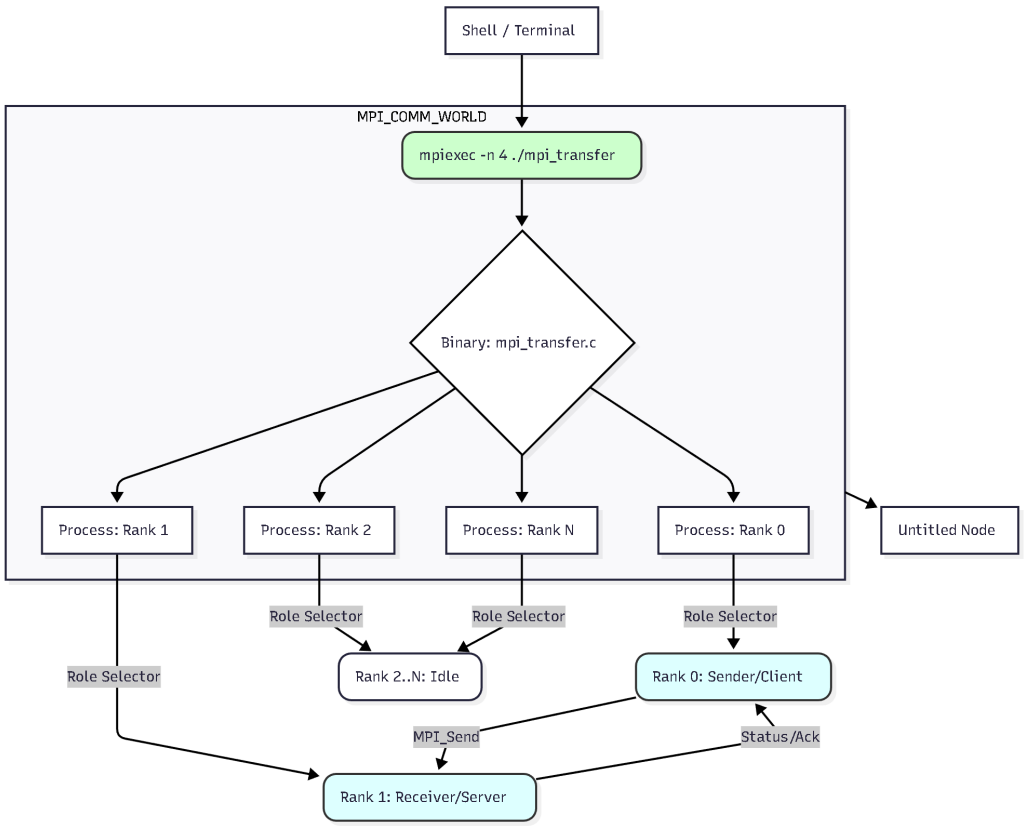

Figure 2: MPI System Architecture: Single Executable, Rank-based Execution

3 File Transfer Implementation

The file transfer implementation uses standard C file I/O (`open`, `read`, `write`) integrated with MPI point-to-point communication calls.

3.1 Code Snippet: Rank 0 (Sender Logic)

Rank 0 handles the two main outgoing messages: the filename and the repeated data chunks, concluding with the end-of-file signal.

```
int filename_len = strlen(filename);
MPI_Send((void*)filename, filename_len + 1, MPI_CHAR, dest_rank, FILENAME_TAG, MPI_CO

while ((bytes_read = read(file_fd_read, data_buffer, BUFFER_SIZE)) > 0) {
    MPI_Send(data_buffer, (int)bytes_read, MPI_CHAR, dest_rank, DATA_TAG, MPI_COMM_WO
    data_sent_total += (int)bytes_read;
}

MPI_Send(NULL, 0, MPI_CHAR, dest_rank, END_TAG, MPI_COMM_WORLD);
```

Figure 3: Rank 0 Code Snippet: Sender Logic using MPI_Send

3.2 Code Snippet: Rank 1 (Receiver Logic)

Rank 1 uses `MPI_ANY_TAG` to listen for both `DATA_TAG` and `END_TAG`, allowing a flexible and non-blocking approach to handling the stream. The `MPI_Get_count` is crucial for determining the exact number of bytes received in the last chunk.

```
MPI_Recv(client_filename, BUFFER_SIZE, MPI_CHAR, source_rank, FILENAME_TAG, MPI_COMM_
// File creation logic follows here...

while (1) {
    MPI_Recv(data_buffer, BUFFER_SIZE, MPI_CHAR, source_rank, MPI_ANY_TAG, MPI_COMM_WO

    if (status.MPI_TAG == END_TAG) {
        break;
    }

    MPI_Get_count(&status, MPI_CHAR, &count);

    if (write(file_fd_write, data_buffer, count) < 0) {
```

```
        perror("Rank 1 ERROR: Failed to write to file");
        break;
    }
    data_received_total += count;
}
```

Figure 4: Rank 1 Code Snippet: Receiver Logic using MPI_Recv