Practical Work 3: MPI File Transfer

Distributed Systems - DS2026

Nguyen Viet Hung
Student ID: 23BI14188
Class: ICT Department

December 12, 2025

Contents

1 Introduction **3**

2 MPI Implementation Choice **3**

3 Service Design **3**
3.1 Communication Flow 3

4 System Organization **4**

5 Implementation Details **4**
5.1 Initialization and Rank Checking 4
5.2 Sender Logic (Rank 0) 5
5.3 Receiver Logic (Rank 1) 5

6 Task Distribution **5**

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 approach using RPC (Remote Procedure Call) or TCP Sockets, MPI focuses on parallel computing where data is exchanged between distinct processes within a communicator.

In this implementation, we simulate a file transfer between two nodes:

- **Rank 0 (Sender):** Reads a file from the local storage and transmits it in chunks.
- **Rank 1 (Receiver):** Receives data chunks and reconstructs the file on the destination storage.

2 MPI Implementation Choice

For this project, we selected **OpenMPI** as our MPI implementation. The reasons for this choice are as follows:

1. **Standard Compliance:** OpenMPI is a high-performance, open-source implementation that fully supports the MPI-3.1 standard, which is widely used in both academic and industrial environments.
2. **Availability Ease of Use:** It is available in the default repositories of most Linux distributions (including Kali Linux and Ubuntu), making the installation process straightforward via `apt`.
3. **Development Tools:** It provides a robust wrapper compiler (`mpicc`) and a versatile runtime environment (`mpirun`), which simplifies the compilation and execution of parallel programs.

3 Service Design

We adopted the **SPMD (Single Program, Multiple Data)** model. A single source code (`transfer.c`) is compiled into one executable. When executed, multiple instances (processes) of this program are launched. The behavior of each process is determined at runtime by its **Rank** ID.

3.1 Communication Flow

The file transfer is designed as a synchronous, blocking communication process:

- The file is split into 4KB buffers (chunks) to handle large files efficiently without exhausting memory.
- A specific "End-of-File" (EOF) signal is implemented by sending a 0-byte message.

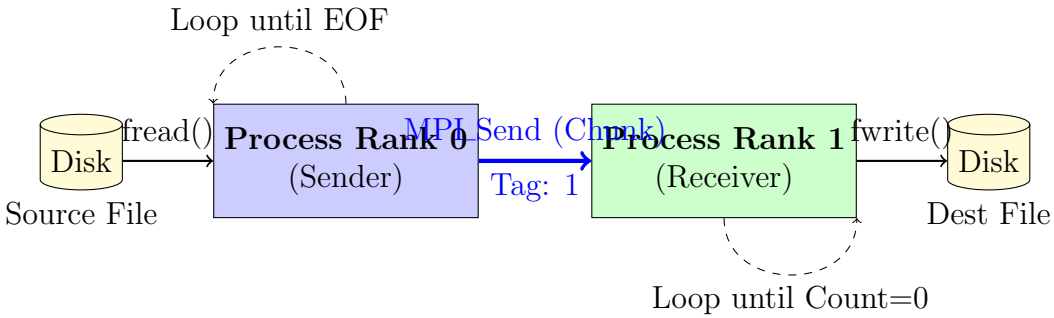

Figure 1: Design of MPI File Transfer System

4 System Organization

The system is organized into a minimalist structure to facilitate easy compilation and execution on Linux environments.

- **transfer.c**: The main C source code containing both the Sender and Receiver logic. It uses `MPI_Comm_rank` to distinguish between the two roles.
- **Makefile**: An automation script to compile the code using `mpicc`.
- **test.txt**: A sample text file used for testing the transmission integrity.

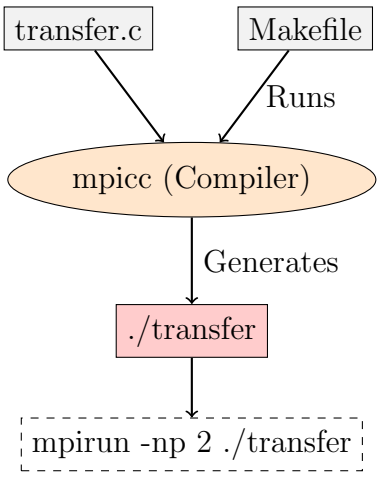

Figure 2: Compilation and Execution Process

5 Implementation Details

The implementation leverages the basic MPI blocking communication functions. Below are the key components of the code.

5.1 Initialization and Rank Checking

We first initialize the MPI environment and determine the process ID (rank). The program ensures at least 2 processes are running.

```c
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (size < 2) {
    if (rank == 0) fprintf(stderr, "Requires at least 2 processes.\n");
    MPI_Finalize();
    return 1;
}
```

5.2 Sender Logic (Rank 0)

The sender reads the file in 4KB chunks and sends them to Rank 1. When the file ends, a 0-byte message is sent as a termination signal.

```c
// Inside if (rank == 0)
while ((bytes_read = fread(buffer, 1, CHUNK_SIZE, f)) > 0) {
    // Send actual data
    MPI_Send(buffer, bytes_read, MPI_BYTE, 1, TAG_DATA, MPI_COMM_WORLD);
}
// Send EOF signal (Empty message)
MPI_Send(buffer, 0, MPI_BYTE, 1, TAG_DATA, MPI_COMM_WORLD);
fclose(f);
```

5.3 Receiver Logic (Rank 1)

The receiver enters an infinite loop, waiting for messages. It uses `MPI_Get_count` to check the size of the received message. If the size is 0, it breaks the loop.

```c
// Inside else if (rank == 1)
while (1) {
    MPI_Recv(buffer, CHUNK_SIZE, MPI_BYTE, 0, TAG_DATA, MPI_COMM_WORLD, &
    status);

    // Check how many bytes were actually received
    MPI_Get_count(&status, MPI_BYTE, &count);

    if (count == 0) {
        break; // Stop if 0-byte message received
    }
    fwrite(buffer, 1, count, f_dest);
}
fclose(f_dest);
```

6 Task Distribution

- **Nguyen Viet Hung:**
  - Researched MPI communication primitives (`MPI_Send`/`MPI_Recv`).
  - Implemented the core logic in C using OpenMPI headers.
  - Created the Makefile for automated building.
  - Tested the application on Kali Linux environment.
  - Wrote this report.