Practical Work

Pham Huu Minh - 23BI14302

2025-12-05

Goal of the Practical Work

The objective of this practical work is to implement a one-to-one file transfer system using the Message Passing Interface (MPI). This involves upgrading the existing TCP file transfer implementation to utilize MPI's communication primitives for distributed file transfer between processes.

Specific Objectives

- Install and configure an MPI implementation (OpenMPI)
- Design an MPI-based architecture for file transfer
- Implement the file transfer mechanism using MPI communication primitives
- Compare the MPI approach with previous TCP Socket and RPC implementations
- Test and verify the correctness of the file transfer system

Environment Setup

The development environment used for this practical work is **Kali Linux**. Kali Linux provides a stable platform with excellent support for development tools and MPI implementations.

OpenMPI Installation

MPI Implementation Used: OpenMPI

OpenMPI was chosen for its wide availability, excellent documentation, and cross-platform support. The installation was performed using the following commands:

```
sudo apt update
sudo apt install openmpi-bin openmpi-common libopenmpi-dev
```

To verify the installation, the following commands were used:

```
mpicc -v
mpirun --version
```

The output confirmed that OpenMPI was successfully installed and ready for use.

MPI Design for File Transfer

Rank-based Architecture

The MPI file transfer system uses a two-process model with distinct roles:

- **Rank 0 (Server/Sender)**:
  - Reads the input file from disk
  - Determines the file size
  - Sends the file size and file content to Rank 1
- **Rank 1 (Client/Receiver)**:
  - Receives the file size from Rank 0
  - Allocates memory for the file data
  - Receives the file content from Rank 0
  - Writes the received data to the output file

Command-line Arguments:

- `input_file`: Path to the source file to be transferred (used by Rank 0)
- `output_file`: Path where the received file will be saved (used by Rank 1)

Message Format

The communication protocol uses two distinct message tags to differentiate between file size and file data:

- `MSG_TAG_SIZE` (value: 100): Used for sending/receiving the file size as an `MPI_INT`
- `MSG_TAG_FILE` (value: 200): Used for sending/receiving the file content as `MPI_BYTE`

The protocol follows a two-step process:

1. First, Rank 0 sends the file size (integer) with tag `MSG_TAG_SIZE`
2. Then, Rank 0 sends the file data (byte array) with tag `MSG_TAG_FILE`
3. Rank 1 receives both messages in the same order

System Organization

The system consists of a single source file: `mpi_file_transfer.c`

The behavior of the program is determined by the MPI rank:

- When `MPI_Comm_rank()` returns 0, the process executes the sender logic
- When `MPI_Comm_rank()` returns 1, the process executes the receiver logic

The code is organized into modular functions:

- `read_file_to_buffer()`: Handles file reading operations
- `write_buffer_to_file()`: Handles file writing operations
- `sender_process()`: Implements the sender logic (Rank 0)
- `receiver_process()`: Implements the receiver logic (Rank 1)
- `main()`: Initializes MPI, validates arguments, and routes to appropriate process function

Implementation

Main Structure and Argument Checking

The main function includes necessary headers, defines constants, and performs initialization:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define MAX_FILE_SIZE 1048576
#define MSG_TAG_SIZE  100
#define MSG_TAG_FILE  200

int main(int argc, char *argv[])
{
    int process_rank, total_processes;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &total_processes);

    if (total_processes != 2) {
        if (process_rank == 0) {
            fprintf(stderr, "Error: This program requires exactly 2 MPI pr
            fprintf(stderr, "Usage: mpirun -np 2 %s <input_file> <output_f
        }
        MPI_Finalize();
        return EXIT_FAILURE;
    }

    if (argc != 3) {
        if (process_rank == 0) {
            fprintf(stderr, "Usage: %s <input_file> <output_file>\n", argv
        }
        MPI_Finalize();
        return EXIT_FAILURE;
    }

    if (process_rank == 0) {
        sender_process(argv[1]);
    } else {
        receiver_process(argv[2]);
    }

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Server Side: Rank 0

The sender process (Rank 0) reads the file and transmits it to the receiver:

```c
static void sender_process(const char *input_filename)
{
    unsigned char *file_data;
    int file_size;

    if (read_file_to_buffer(input_filename, &file_data, &file_size) != 0)
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        return;
    }

    printf("[Sender] Read %d bytes from '%s', transmitting to receiver...\
           file_size, input_filename);

    MPI_Send(&file_size, 1, MPI_INT, 1, MSG_TAG_SIZE, MPI_COMM_WORLD);
    MPI_Send(file_data, file_size, MPI_BYTE, 1, MSG_TAG_FILE, MPI_COMM_WOR

    printf("[Sender] Transmission completed successfully.\n");
    free(file_data);
}
```

The sender process:

1. Opens the input file and reads its entire content into memory
2. Validates the file size (must be within the 1 MB limit)
3. Sends the file size using `MPI_Send()` with tag `MSG_TAG_SIZE`
4. Sends the file data using `MPI_Send()` with tag `MSG_TAG_FILE`
5. Frees allocated memory and completes

Client Side: Rank 1

The receiver process (Rank 1) receives the file and writes it to disk:

```c
static void receiver_process(const char *output_filename)
{
    int file_size;
    unsigned char *file_data;
    MPI_Status status;

    MPI_Recv(&file_size, 1, MPI_INT, 0, MSG_TAG_SIZE, MPI_COMM_WORLD, &sta

    if (file_size <= 0 || file_size > MAX_FILE_SIZE) {
        fprintf(stderr, "[Receiver] Invalid file size received: %d\n", fil
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        return;
    }

    file_data = (unsigned char *)malloc(file_size);
    if (file_data == NULL) {
        perror("[Receiver] Memory allocation failed");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        return;
    }

    MPI_Recv(file_data, file_size, MPI_BYTE, 0, MSG_TAG_FILE, MPI_COMM_WOR

    if (write_buffer_to_file(output_filename, file_data, file_size) != 0)
        free(file_data);
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        return;
    }

    printf("[Receiver] Successfully received %d bytes and saved to '%s'\n"
           file_size, output_filename);
    free(file_data);
}
```

The receiver process:

1. Receives the file size using `MPI_Recv()` with tag `MSG_TAG_SIZE`
2. Validates the received file size
3. Allocates memory for the file data
4. Receives the file data using `MPI_Recv()` with tag `MSG_TAG_FILE`
5. Writes the received data to the output file using `fwrite()`
6. Frees allocated memory and completes

Build Commands

The compilation command used to build the MPI file transfer program:

```
mpicc -Wall -g -o mpi_file_transfer mpi_file_transfer.c
```

Where:

- `mpicc`: OpenMPI's C compiler wrapper
- `-Wall`: Enable all compiler warnings
- `-g`: Include debugging information
- `-o mpi_file_transfer`: Specify the output executable name
- `mpi_file_transfer.c`: Source file

Execution and Test Results

Creating a Test File

A test file was created with dummy data for testing:

```
echo "This is a test file for MPI file transfer." > test_mpi.txt
```

Additional test files of various sizes were also created to test the system with different file sizes.

Running the MPI Program

The MPI program was executed using the following command:

```
mpirun -np 2 ./mpi_file_transfer test_mpi.txt test_mpi_copy.txt
```

Output logs:

```
[Sender] Read 45 bytes from 'test_mpi.txt', transmitting to receiver...
[Sender] Transmission completed successfully.
[Receiver] Successfully received 45 bytes and saved to 'test_mpi_copy.txt'
```

The output shows that:

- Rank 0 successfully read 45 bytes from the input file
- Rank 0 completed sending the data
- Rank 1 successfully received 45 bytes and wrote them to the output file

Verifying the Result

To verify that the files are identical, the `diff` command was used:

```
diff test_mpi.txt test_mpi_copy.txt
```

No output from `diff` confirms that the files are identical. Alternatively, the files can be compared using:

```
cat test_mpi.txt
cat test_mpi_copy.txt
```

Both commands show identical content, confirming the successful file transfer.

Discussion

MPI vs. TCP Socket Implementation

Advantages of MPI:

- **Simplified Communication**: MPI provides high-level communication primitives (`MPI_Send`/`MPI_Recv`) that abstract away low-level socket programming details
- **Process Management**: MPI handles process creation, synchronization, and cleanup automatically
- **Portability**: MPI code is portable across different platforms and network configurations
- **Performance**: MPI implementations are highly optimized and can utilize high-

performance interconnects (InfiniBand, etc.)

- **No Manual Network Programming**: No need to handle socket creation, binding, listening, or connection management

Disadvantages of MPI:

- **Requires MPI Runtime**: Both sender and receiver must have MPI installed and running
- **Less Flexibility**: Less control over low-level network behavior compared to raw sockets
- **Learning Curve**: Requires understanding of MPI concepts (ranks, communicators, tags)

MPI vs. RPC Implementation

Advantages of MPI:

- **Direct Communication**: MPI provides direct point-to-point communication without the overhead of RPC marshalling/unmarshalling
- **Simpler Protocol**: No need to define interface specifications (like XDR files in RPC)
- **Better Performance**: Lower latency for simple data transfers without RPC overhead
- **No Code Generation**: No need for code generation tools like `rpcgen`

Disadvantages of MPI:

- **Less Abstraction**: RPC provides a more abstract interface (function calls) compared to MPI's message-passing model
- **No Interface Definition**: RPC's interface definition files provide better documentation and type safety
- **Platform Dependency**: MPI is primarily used in HPC environments, while RPC is more general-purpose

Personal Work

All tasks for this practical work were completed individually by Pham Huu Minh (23BI14302):

- Research and selection of MPI implementation (OpenMPI)
- Installation and configuration of OpenMPI on Kali Linux
- Design of the MPI-based file transfer architecture
- Implementation of the sender process (Rank 0) logic
- Implementation of the receiver process (Rank 1) logic
- Implementation of file I/O helper functions
- Code compilation and debugging
- Testing with various file sizes and types
- Comparison analysis with TCP Socket and RPC implementations
- Documentation and report writing

Conclusion

This practical work successfully demonstrated the implementation of a file transfer system using the Message Passing Interface (MPI). The system was able to transfer files between two MPI processes using OpenMPI, with Rank 0 acting as the sender and Rank 1 as the receiver.

The implementation showed that MPI provides a clean and efficient way to implement distributed file transfer, with advantages in terms of code simplicity and performance compared to low-level socket programming. However, it requires the MPI runtime environment and has a different programming model compared to RPC.

The lab successfully demonstrated three different approaches to distributed file transfer:

- **TCP Sockets**: Low-level, flexible, but requires manual network programming
- **RPC**: High-level abstraction with interface definitions, but with marshalling overhead
- **MPI**: Efficient message-passing model, ideal for HPC environments, but requires MPI runtime

Each approach has its own strengths and is suitable for different use cases and environments.