

Practical Work #3 – MPI File Transfer

Nguyen Tien Dung – 23BI14116

December 4, 2025

1 Goal of the practical work

The goal of this practical work is to implement a one-to-one file transfer using the Message Passing Interface (MPI). Instead of sockets (Practical Work #1) or RPC (Practical Work #2), the communication is expressed in terms of message passing between MPI processes.

The main objectives are:

- Install and configure an MPI implementation on the working environment.
- Design a simple “server–client” architecture using MPI ranks.
- Implement a file transfer where one process sends a file and another process receives and stores it.
- Compare this MPI-based solution with the previous TCP and RPC implementations.

2 Environment setup

All MPI experiments were done on the same Kali Linux virtual machine used in the previous labs.

2.1 OpenMPI installation

I used OpenMPI as the MPI implementation. On Kali Linux, it is available in the distribution repositories. The following commands were used to install and verify it:

Listing 1: Installing and checking MPI on Kali

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

# Check that MPI is available
mpicc -v
mpirun --version
```

The output confirms that `mpicc` is available as a wrapper around `gcc`, and that OpenMPI 5.x is installed.

3 MPI design for file transfer

3.1 Rank-based architecture

The MPI program uses exactly two processes:

- **Rank 0** acts as the “server”:
 - Reads the entire input file into memory.
 - Sends the size of the file to rank 1.

- Sends the actual file data as a byte array.
- **Rank 1** acts as the “client”:
 - Receives the file size from rank 0.
 - Allocates a buffer of the appropriate size.
 - Receives the file data and writes it to an output file.

The program is launched with two processes and takes two arguments:

- `input_file`: path to the file that rank 0 will read and send.
- `output_file`: path to the file that rank 1 will create and write.

3.2 Message format

The communication protocol is very simple:

1. Rank 0 sends an `int` representing the file size to rank 1 using tag `TAG_SIZE`.
2. Rank 0 sends the file data using type `MPI_BYTE` and tag `TAG_DATA`.
3. Rank 1 receives the size, then receives exactly that many bytes and writes them to disk.

For safety, I limit the maximum file size to `MAXFILESIZE = 1 MB`.

4 System organization

The implementation is contained in a single C source file:

- `mpi_file_transfer.c`: main MPI program. The behavior is selected based on the rank (0 or 1).

The program is compiled with `mpicc` and executed with `mpirun`:

```
mpicc -Wall -g mpi_file_transfer.c -o mpi_file_transfer
mpirun -np 2 ./mpi_file_transfer input.txt output.txt
```

5 Implementation

5.1 Main structure and argument checking

The main function initializes MPI, checks the number of processes, and parses the command line arguments:

Listing 2: Initialization and argument checking

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

#define MAXFILESIZE 1048576 /* 1 MB */
#define TAG_SIZE 0
#define TAG_DATA 1

int main(int argc, char *argv[])
{
    int rank, size;

    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,
            "Please run with exactly 2 processes.\n"
            "Example: mpirun -np 2 ./mpi_file_transfer in.txt out.txt\n");
    }
    MPI_Finalize();
    return EXIT_FAILURE;
}

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

const char *input_file = argv[1];
const char *output_file = argv[2];

```

If either the number of processes or the arguments are incorrect, only rank 0 prints an error message and the program terminates cleanly.

5.2 Server side: rank 0

Rank 0 acts as the server: it reads the input file, checks its size, and sends the data to rank 1.

Listing 3: Server logic on rank 0

```

if (rank == 0) {
    FILE *fp = fopen(input_file, "rb");
    if (!fp) {
        perror("fopen input_file");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (fseek(fp, 0, SEEK_END) != 0) {
        perror("fseek");
        fclose(fp);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    long filesize = ftell(fp);
    if (filesize < 0 || filesize > MAXFILESIZE) {
        fprintf(stderr,
            "File too large or ftell error (size=%ld)\n", filesize);
        fclose(fp);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    rewind(fp);

    int len = (int)filesize;
    unsigned char *buffer = malloc(len);
    if (!buffer) {

```

```

        perror("malloc");
        fclose(fp);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (fread(buffer, 1, len, fp) != (size_t)len) {
        perror("fread");
        free(buffer);
        fclose(fp);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    fclose(fp);

    printf("[Rank_0]_Read_%d_bytes_from_%s_sending_to_rank_1...\n",
           len, input_file);

    MPI_Send(&len, 1, MPI_INT, 1, TAG_SIZE, MPI_COMM_WORLD);
    MPI_Send(buffer, len, MPI_BYTE, 1, TAG_DATA, MPI_COMM_WORLD);

    printf("[Rank_0]_Done_sending.\n");
    free(buffer);

```

I use `MPI_Abort` for error handling so that both processes terminate consistently if something goes wrong on one side.

5.3 Client side: rank 1

Rank 1 receives the size and the data, then writes the received bytes to the output file:

Listing 4: Client logic on rank 1

```

} else if (rank == 1) {
    int len = 0;
    MPI_Status status;

    MPI_Recv(&len, 1, MPI_INT, 0, TAG_SIZE, MPI_COMM_WORLD, &status);
    if (len <= 0 || len > MAXFILESIZE) {
        fprintf(stderr,
                "[Rank_1]_Invalid_size_received:%d\n", len);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    unsigned char *buffer = malloc(len);
    if (!buffer) {
        perror("malloc");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    MPI_Recv(buffer, len, MPI_BYTE, 0, TAG_DATA, MPI_COMM_WORLD, &status);

    FILE *out = fopen(output_file, "wb");
    if (!out) {
        perror("fopen_output_file");
        free(buffer);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    if (fwrite(buffer, 1, len, out) != (size_t)len) {
        perror("fwrite");
    }
}

```

```

        free(buffer);
        fclose(out);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    fclose(out);
    printf("[Rank 1] Received %d bytes from rank 0 and wrote to %s\n",
        len, output_file);
    free(buffer);
}

MPI_Finalize();
return EXIT_SUCCESS;
}

```

After `MPI_Finalize()`, both processes exit cleanly.

6 Build commands

The program is compiled with `mpicc` (which automatically links against the MPI libraries):

Listing 5: Compilation command

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

7 Execution and test results

7.1 Creating a test file

To test the file transfer, I first create a small text file:

```
echo "Hello from MPI server" > test_mpi.txt
```

7.2 Running the MPI program

I run the program with two processes as required:

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

Typical console output is:

```

[Rank 0] Read 23 bytes from test_mpi.txt, sending to rank 1...
[Rank 0] Done sending.
[Rank 1] Received 23 bytes from rank 0 and wrote to test_mpi_copy.txt

```

7.3 Verifying the result

To confirm that the file was correctly transferred, I compare the contents:

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

Both files contain the same string “Hello from MPI server”, which shows that the MPI file transfer works as expected.

8 Discussion

This MPI-based implementation can be compared to the previous TCP and RPC solutions:

- **Compared to TCP sockets:**
 - MPI hides explicit socket creation and connection management.
 - Processes are started together using `mpirun`, and communication uses ranks instead of IP addresses and ports.
- **Compared to RPC:**
 - MPI expresses communication at the level of messages, while RPC expresses it as remote procedure calls.
 - The MPI implementation is shorter for this simple point-to-point example, but does not provide automatic type marshalling like `rpcgen`.

MPI is mainly designed for parallel scientific applications, not for generic client/server services, but for a simple lab it provides a clean and explicit way to implement file transfer between two processes.

9 Personal work

For this practical work I:

- Installed and verified OpenMPI on the Kali virtual machine.
- Designed the rank-based architecture with rank 0 as server and rank 1 as client.
- Implemented the file transfer logic in `mpi_file_transfer.c`.
- Tested the program with sample files and verified the correctness of the transfer.
- Prepared this report describing the design, implementation details, commands used, and test results.

10 Conclusion

In this practical work I implemented a simple MPI-based file transfer between two processes. Using OpenMPI, the entire protocol could be written with a few calls to `MPI_Send` and `MPI_Recv`. Combined with the previous TCP and RPC labs, this exercise shows three different ways to build distributed file transfer systems: sockets, remote procedure calls, and message passing with MPI.