Practical Work 3: MPI File Transfer

Pham Hoang Viet
ID: 23BI14452

1 Goal

The goal of this practical work is to upgrade the TCP file transfer system into a file transfer system using MPI. The new system must:

- use MPI as the communication layer;
- keep a one-to-many file transfer model (one sender, multiple receivers);
- transfer real file data, not only messages or numbers;
- be controlled from the command line.

2 Choice of MPI Implementation

The implementation uses the standard MPI-3 interface provided by Open MPI (or MPICH). The reasons for this choice are:

- MPI is widely available on Unix-like systems and clusters;
- `mpic++` integrates easily with C++ code;
- collective communication primitives such as `MPI_Bcast` fit naturally with the file broadcast requirement (one sender, multiple receivers);
- the same code can run on a single machine or on a cluster without modification.

3 MPI Service Design

Communication Pattern

The file transfer is designed as a broadcast service:

- process with rank 0 is the **root** (sender);
- all processes with rank > 0 are **receivers**;

- the root reads a file from disk and broadcasts:
  1. the file name;
  2. the file size;
  3. the file content.
- each receiver writes a local copy of the file.

Design Figure

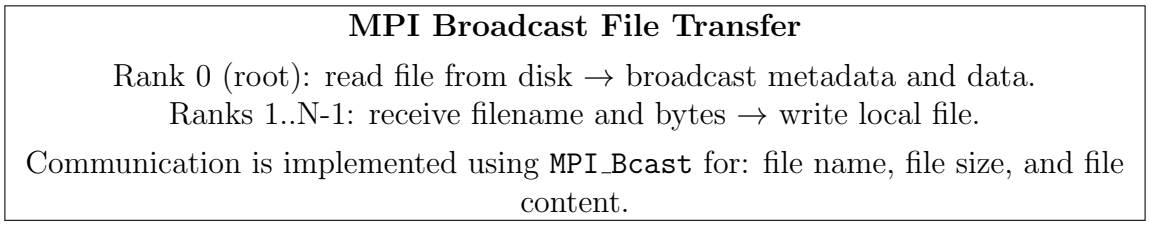

Figure 1: High-level design of the MPI broadcast file transfer.

4 System Organization

The system is organized into a single C++ program `mpi.cc` that runs with multiple MPI processes. Its behavior depends on the rank obtained from `MPI_Comm_rank()`.

- **Root process (rank 0)**:
  - parses the input file name from the command line;
  - reads the whole file into memory;
  - broadcasts the file name as a character array;
  - broadcasts the file size as an integer;
  - broadcasts the file content as a byte array.
- **Receiver processes (rank > 0)**:
  - receive the file name, size, and content;
  - write the content into a file named `received_rankX_filename`;
  - confirm completion by printing a local message.

System Organization

Single source file: `mpi.cc`.
MPI rank 0 executes "sender role".
MPI ranks 1..N-1 execute "receiver role".
All processes share the same executable, but the control flow is separated by rank.

Figure 2: Organization of the MPI file transfer program.

5 Implementation Details

5.1 Core File Transfer Logic (Code Snippet)

The essential code reads the file (on the root), broadcasts metadata, and writes the file (on the receivers). The following snippet shows the main logic in `mpi.cc`:

Listing 1: Core MPI broadcast file transfer snippet.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
const int ROOT = 0;

// --- Step 1: broadcast file name ---
std::string filename;
std::vector<char> nameBuf;
int nameLen = 0;

if (rank == ROOT) {
    filename = argv[1];
    nameBuf.assign(filename.begin(), filename.end());
    nameBuf.push_back('\0');
    nameLen = static_cast<int>(nameBuf.size());
}
MPI_Bcast(&nameLen, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
if (rank != ROOT) {
    nameBuf.resize(nameLen);
}
MPI_Bcast(nameBuf.data(), nameLen, MPI_CHAR, ROOT, MPI_COMM_WORLD);
if (rank != ROOT) {
    filename = std::string(nameBuf.data());
}

// --- Step 2: root reads file ---
int fileSize = 0;
std::vector<char> buffer;

if (rank == ROOT) {
    std::ifstream in(filename, std::ios::binary);
    in.seekg(0, std::ios::end);
    fileSize = static_cast<int>(in.tellg());
    in.seekg(0, std::ios::beg);
    buffer.resize(fileSize);
    in.read(buffer.data(), fileSize);
```

```
    in.close();
}

MPI_Bcast(&fileSize, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
if (rank != ROOT) {
    buffer.resize(fileSize);
}

// --- Step 3: broadcast file content ---
MPI_Bcast(buffer.data(), fileSize, MPI_CHAR, ROOT, MPI_COMM_WORLD);

// --- Step 4: receivers write file ---
if (rank != ROOT) {
    std::string outName =
        "received_rank" + std::to_string(rank) + "_" + filename;
    std::ofstream out(outName, std::ios::binary);
    out.write(buffer.data(), fileSize);
    out.close();
}
```

The complete program also includes initialization and finalization:

Listing 2: Program initialization and finalization.

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    // ... logic shown above ...
    MPI_Finalize();
    return 0;
}
```

6 Execution and Testing

The program is compiled using `mpic++` and executed with `mpirun`.

Compilation

```
mpic++ mpi.cc -o mpi_file_transfer
```

Running the Program

The root process (rank 0) takes the file name as a command-line argument:

```
mpirun -np 4 ./mpi_file_transfer example_file.txt
```

After execution:

- rank 0 prints how many bytes were read from the original file;
- each receiver rank ($1 \ldots 3$ in this example) writes a file named `received_rankX_example_file.txt`;
- the contents of each output file match the original file.

7 Conclusion

This practical work implements a simple yet complete MPI-based file transfer system using broadcast. The design uses collective communication to distribute a file from one root process to all other processes in the communicator. The implementation demonstrates:

- reading and writing binary files in C++;
- broadcasting metadata (file name and size) with MPI;
- broadcasting arbitrary binary content using `MPI_Bcast`;
- organizing a single MPI program into sender and receiver roles based on process rank.

The broadcast approach is straightforward and scalable, and can be extended with features such as checksum verification, chunked transfer, and selective distribution to a subset of processes.