Practical Work 3 - MPI File Transfer

Student name: Do Nhat Thanh
Studen ID: BI12-416

Explain MPI implementation choice

The MPI implementation provided in the example is designed to be simple, focusing on the fundamental aspects of MPI communication to achieve a file transfer between two processes. Here's why this specific approach was chosen:

1. Simplicity and Clarity:

The primary goal was to demonstrate a straightforward use case of MPI for file transfer, making the example accessible to those new to MPI programming. By using basic MPI functions like `MPI_Init`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Send`, and `MPI_Recv`, the example illustrates the core mechanics of initiating MPI, determining the roles of different processes based on their ranks, and performing point-to-point communication.

2. Portability and Accessibility:

MPI is a widely adopted standard in high-performance computing, with implementations available on a variety of platforms. The example code avoids complex or less commonly used features that might not be supported uniformly across different MPI implementations. This ensures that the example is as portable as possible, capable of running on different systems and MPI distributions with minimal modifications.

3. Educational Value:

For educational purposes, it's beneficial to start with the basics. The example covers key concepts like initializing the MPI environment, determining process ranks, sending and receiving messages, and finalizing the MPI environment. Understanding these concepts is crucial for anyone learning MPI and serves as a foundation for exploring more advanced topics and capabilities of MPI.

4. Demonstrating Point-to-Point Communication:

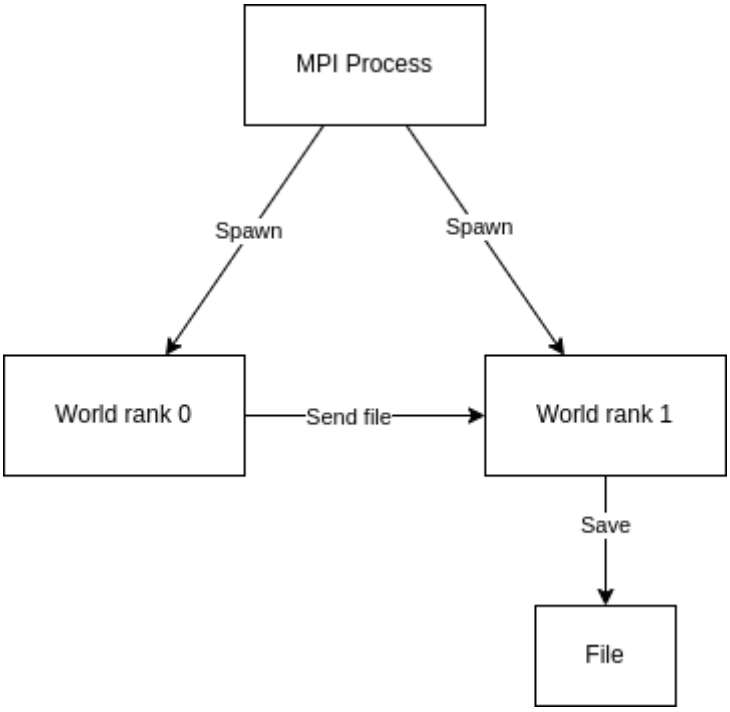
The file transfer task is an excellent way to demonstrate point-to-point communication, one of the fundamental communication paradigms in MPI. The sender (rank 0) and receiver (rank 1) roles clearly show how processes can exchange messages. This example can easily be

extended or modified to explore other MPI communication patterns, such as collective communication or non-blocking communication.

5. Basis for Extension:

The provided implementation serves as a basic framework that can be expanded. For instance, users can modify it to include error handling for MPI calls, implement non-blocking communication to improve efficiency, or incorporate collective operations for scalability. It provides a starting point from which more complex and tailored MPI-based applications can be developed.

System design



Implementation

Init MPI

```
MPI_Init(&argc, &argv);

int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

File sender with rank 0

```
if (world_rank == 0) {
        // Sender
        if (argc < 2) {
                fprintf(stderr, "Usage: mpiexec -n 2 %s <filename>\n",
argv[0]);
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        char* filename = argv[1];
        FILE* file = fopen(filename, "rb");
        if (!file) {
                perror("Failed to open file");
                MPI_Abort(MPI_COMM_WORLD, 1);
        }

        char buffer[BUFFER_SIZE];
        while (!feof(file)) {
                int bytes_read = fread(buffer, 1, BUFFER_SIZE, file);
                MPI_Send(&bytes_read, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
                if (bytes_read > 0) {
                        MPI_Send(buffer, bytes_read, MPI_BYTE, 1, 0,
MPI_COMM_WORLD);
                }
        }

        // Send a zero to signal the end of the file
        int end_signal = 0;
        MPI_Send(&end_signal, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

        fclose(file);
    }
```

File receiver with rank 1

```
else if (world_rank == 1) {
        // Receiver
        FILE* file = fopen("received_file", "wb");
        if (!file) {
                perror("Failed to open file for writing");
                MPI_Abort(MPI_COMM_WORLD, 1);
        }
```

```
        int message_size;
        char buffer[BUFFER_SIZE];
        MPI_Status status;

        do {
                MPI_Recv(&message_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
&status);
                if (message_size > 0) {
                        MPI_Recv(buffer, message_size, MPI_BYTE, 0, 0,
MPI_COMM_WORLD, &status);
                        fwrite(buffer, 1, message_size, file);
                }
        } while (message_size > 0);

        fclose(file);
    }
```