Distributed System Labwork 2

Group 1 - ICT

University of Science and Technology of Hanoi

January, 2022

Contents

1 Introduction

1.1 Overview

MPI(Message Passing Interface) is a library designed to allow users to create programs that can run efficiently on most parallel architectures. In the message-passing model of parallel computation, the processes executing in parallel have separate address spaces. Communication occurs when a portion of one process's address space is copied into another process's address space. This operation is cooperative and occurs only when the first process executes a send operations and the second process executes a receive operation.[1]
In this labwork, we try to build a file transfer system using MPI. We will use C in this labwork.

1.2 Protocol

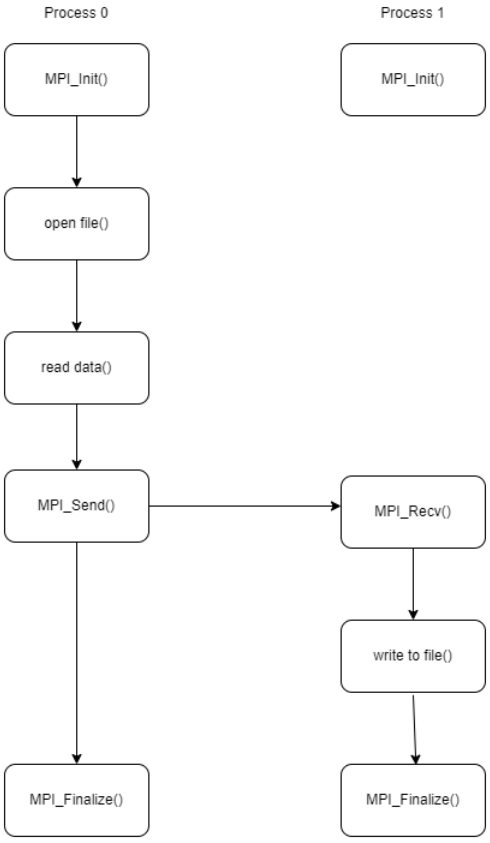

Figure 1: Protocol diagram

2 Implementation

This is the implementation of the labwork. In order to run the file, please type "mpicc mpi_ftp.c -o mpi" and then "mpirun -np 2 ./mpi"

```
#include <stdio.h>
#include <mpi.h>
#include <string.h>
#include <stdlib.h>
#define MAX_SIZE 1024
#define size_tag 2001
#define char_tag 2002

void send_file(FILE* fp, char* filename, char* buffer, int rank_des) {
  fp = fopen(filename, "r");
  if (fp == NULL) {
    perror("Error in reading file..\n");
    exit(1);
  }
  else {
    printf("Reading file successfully..\n");
  }
  int buffer_size;
  while (1) {
    int buffer_size = fread(buffer, 1, MAX_SIZE, fp);
    MPI_Send(&buffer_size, 1, MPI_INT, rank_des, size_tag, MPI_COMM_WORLD);
    MPI_Send(buffer, buffer_size, MPI_CHAR, rank_des, char_tag, MPI_COMM_WORLD);
    if (buffer_size < MAX_SIZE) {
      printf("\nUpload finished.\n");
      break;
    }
  }
  fclose(fp);

}

void receive_file(FILE* fp, char* filename, char* buffer, int rank_from) {

  fp = fopen(filename, "a");
  if (fp == NULL) {
    perror("Error in reading file..\n");
```

```
37      exit(1);
38    }
39    else {
40      printf("Reading file received successfully..\n");
41    }
42    int buffer_size;
43    while (1) {
44      MPI_Recv(&buffer_size, 1, MPI_INT, rank_from, size_tag, MPI_COMM_WORLD,
        ↪  MPI_STATUS_IGNORE);
45      MPI_Recv(buffer, buffer_size, MPI_CHAR, rank_from, char_tag, MPI_COMM_WORLD,
        ↪  MPI_STATUS_IGNORE);
46      fwrite(buffer, 1, buffer_size, fp);
47      if (buffer_size < MAX_SIZE) {
48        printf("\nWrite finished.\n");
49        break;
50      }
51    }
52    fclose(fp);
53  }
54
55  int main(int argc, char* argv[]) {
56    FILE* fp;
57    int my_id, numprocs, len;
58    char buffer[MAX_SIZE];
59    char name[MPI_MAX_PROCESSOR_NAME];
60    int client_to_server = 1;
61    MPI_Init(&argc, &argv);
62
63    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
64    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
65
66    if (numprocs < 2) {
67      printf("Need at least 2 processes");
68    }
69    MPI_Get_processor_name(name, &len);
70    char* root_send = "send.txt";
71    char* slave_received = "received.txt";
72    memset(&buffer, 0, sizeof(buffer));
73    if (client_to_server) {
74      if (my_id == 0) {
```

```
75          send_file(fp, root_send, buffer, 1);
76        }
77        else if (my_id == 1) {
78          receive_file(fp, slave_received, buffer, 0);
79        }
80      }
81      else {
82        if (my_id == 0) {
83          receive_file(fp, root_send, buffer, 1);
84        }
85        else if (my_id == 1) {
86          send_file(fp, slave_received, buffer, 0);
87        }
88      }
89      MPI_Finalize();
90  }
```

This is the result:

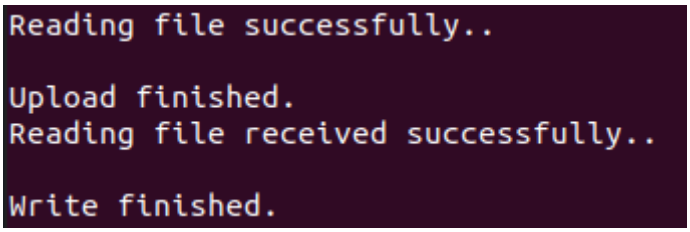

Figure 2: Result after file transfer

3 Contribution

Member	**Contribution**
Nguyen Xuan Tung	Send fIle code
Nguyen Quang Anh	Receive file code
Lu Khanh Huyen	Design Protocol
Tran Hong Quan	MPI concept
Vu Duc Chinh	Report

Table 1: Contribution Table

References

[1] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.