

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

DISTRIBUTED SYSTEM

FINAL REPORT

Remote shell MPI

Group 1

Truong Si Thi Vu

Ta Duc Anh

Dinh Nhu Minh Phuong

Praise Oketola

Nguyen Tuan Duy

April 16, 2020

Contents

| | | |
|----------|-------------------------------------|-----------|
| 1 | Introduction | 2 |
| 1.1 | Message Passing Interface | 2 |
| 1.2 | MPI commands | 2 |
| 1.3 | Remote shell | 2 |
| 2 | Objectives | 3 |
| 3 | State of the art | 3 |
| 4 | Method | 3 |
| 4.1 | MPI | 3 |
| 4.2 | Architecture | 4 |
| 4.3 | Installing MPI | 5 |
| 4.4 | Server side | 5 |
| 4.5 | Client side | 7 |
| 4.6 | Running the program | 8 |
| 5 | Evaluation | 10 |
| 6 | Conclusion | 11 |
| 6.1 | Results | 11 |
| 6.2 | References | 11 |

1 Introduction

1.1 Message Passing Interface

Message Passing Interface (MPI) [1] is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

MPI consists of

- a header file `mpi.h`
- a library of routines and functions, and
- a runtime system

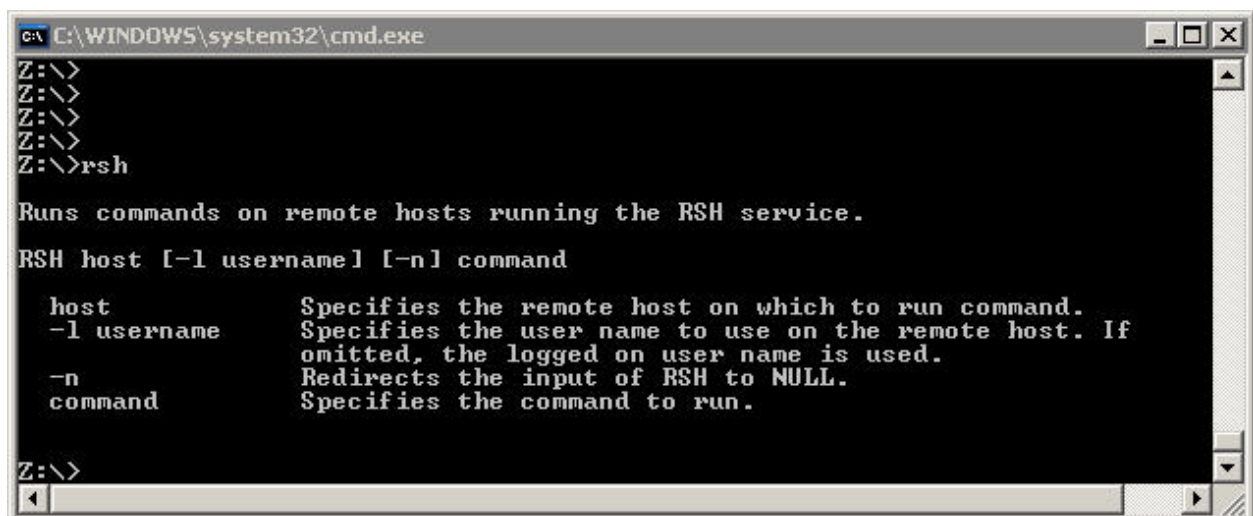
1.2 MPI commands

Seven Basic MPI commands

- `MPI Init()` – Allow command arguments to be modified
- `MPI Finalize()`
- `MPI Comm size()` – Number of MPI processes
- `MPI Comm rank()` – Internal process number
- `MPI Get_processor_name()` – External processor name
- `MPI Send()`
- `MPI Recv()`

1.3 Remote shell

The remote shell (`rsh`) is a command line computer program that can execute shell commands as another user, and on another computer across a computer network.



```
C:\WINDOWS\system32\cmd.exe
Z:\>
Z:\>
Z:\>
Z:\>
Z:\>rsh

Runs commands on remote hosts running the RSH service.

RSH host [-l username] [-n] command

  host          Specifies the remote host on which to run command.
  -l username    Specifies the user name to use on the remote host. If
                  omitted, the logged on user name is used.
  -n            Redirects the input of RSH to NULL.
  command       Specifies the command to run.

Z:\>
```

The remote system to which rsh connects runs the rsh daemon (rshd). The daemon typically uses the well-known Transmission Control Protocol (TCP) port number 514.

As an example of rsh use, the following executes the command `mkdir testdir` as user `remoteuser` on the computer `host.example.com` running a UNIX-like system:

```
rsh -l remoteuser host.example.com "mkdir testdir"
```

2 Objectives

We want to create a remote shell to communicate between client and server, but through message passing interface

Besides that, we want to learn more about the Message Passing Interface and its overall usage.

3 State of the art

During our process of researching, we found very few projects related to MPI in general and even fewer projects related to MPI shell. It is hard to compare our solution to others since the community of this protocol is rather small.

The closest and probably the only project that is related to our own is called MPI-Bash [2] - A Parallel version of the Bash shell written by Spakin (<https://github.com/lanl/MPI-Bash>)

Relating Spakin's project, MPI-Bash includes various MPI functions for data transfer and synchronization, it is not limited to embarrassingly parallel workloads but can incorporate phased operations (i.e., all workers must finish operation X before any worker is allowed to begin operation Y).

Being new to MPI this is a truly difficult concept to grasp and his solution is clearly way more advanced and modern, while ours is naive.

4 Method

Similar to the Hello World example, we created a server-client model based on the functions provided by MPI to pass messages and execute commands. We use client-server chat program as our skeleton structure.

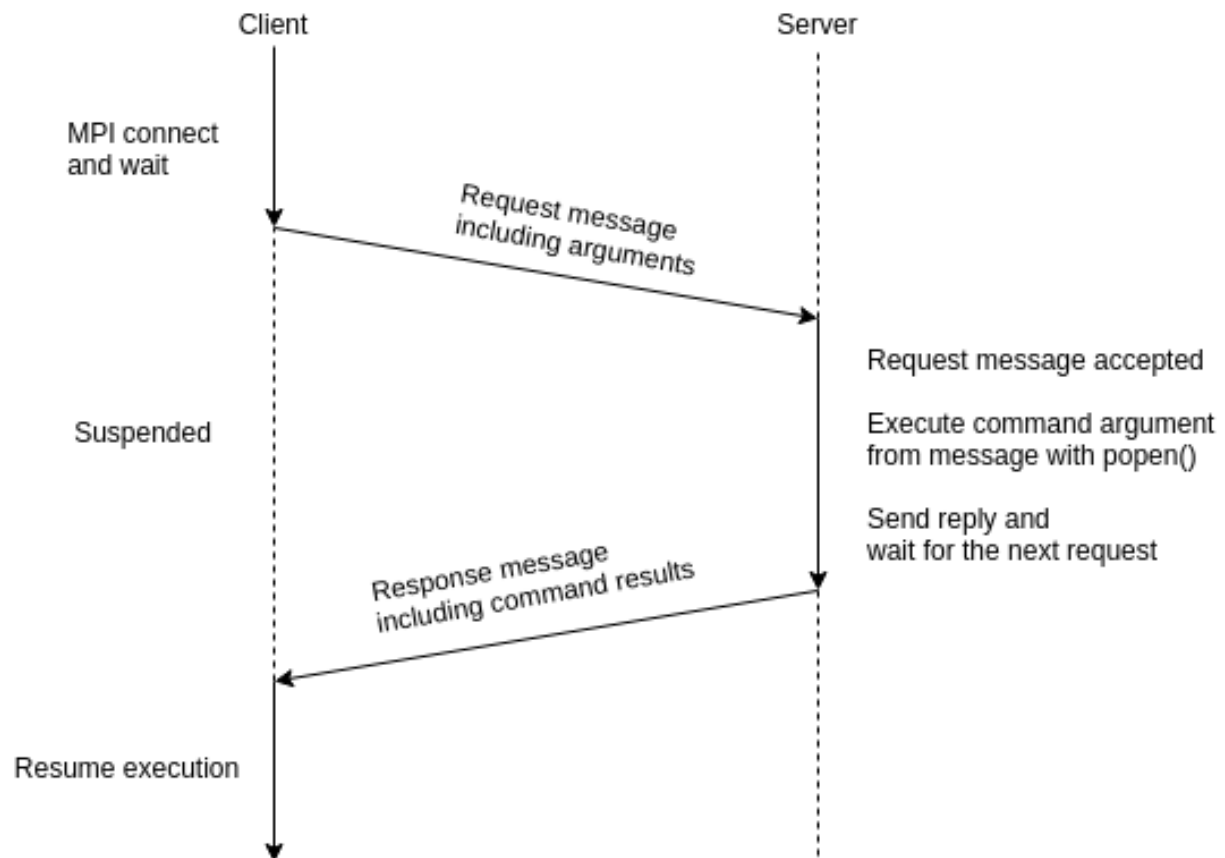
4.1 MPI

We used MPICH which is an MPI implementation from Argonne National Lab and Mississippi State University. The reason is because its free, easy to point ot new machines, well performed, easy to configure, build.

We choose the way to implement called Eager: Send data immediately; Use pre-allocated or dynamically allocated remote buffer space.

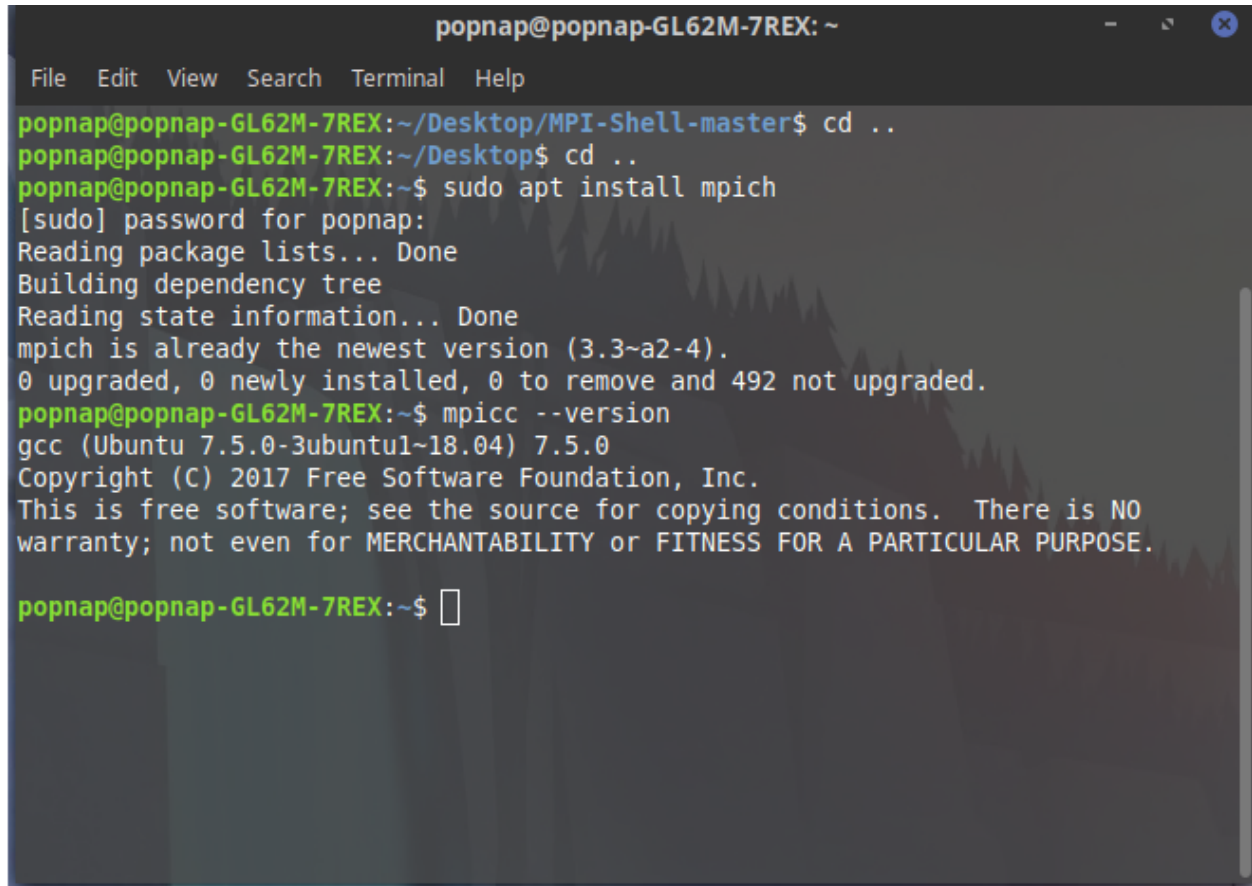
- One-way communication (fast)
- Requires buffer management
- Requires buffer copy
- Does not synchronize processes (good)

4.2 Architecture



4.3 Installing MPI

First of all let's install MPI. We will see if mpicc - the compiler of MPI programs is active.

A terminal window titled 'popnap@popnap-GL62M-7REX: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
popnap@popnap-GL62M-7REX:~/Desktop/MPI-Shell-master$ cd ..
popnap@popnap-GL62M-7REX:~/Desktop$ cd ..
popnap@popnap-GL62M-7REX:~$ sudo apt install mpich
[sudo] password for popnap:
Reading package lists... Done
Building dependency tree
Reading state information... Done
mpich is already the newest version (3.3~a2-4).
0 upgraded, 0 newly installed, 0 to remove and 492 not upgraded.
popnap@popnap-GL62M-7REX:~$ mpicc --version
gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

popnap@popnap-GL62M-7REX:~$
```

4.4 Server side

The server side is composed of 7 main components:

- `MPI_Init(&argc, &argv);` - Initializes MPI
- `MPI_Comm_size(MPI_COMM_WORLD, &size);` Opens a communicator that accepts 1 process only.
- `MPI_Open_port(MPI_INFO_NULL, port_name);` Opens the port that client connects to (port_name)
- `MPI_Recv(&text, 1000, MPI_CHAR, MPI_ANY_SOURCE, TAG_CLIENT_MESSAGE, client, &status);` Listens for the client messages
- `MPI_Send(&path, strlen(path) + 1, MPI_CHAR, 0, TAG_SERVER_RESULT, client);` Sends the results back
- `MPI_Comm_free(&client); MPI_Close_port(port_name); MPI_Finalize();` Closes and ends connections

Full server.c code:

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

#define TAG_CLIENT_MESSAGE 2
#define TAG_SERVER_RESULT 3
#define SIZE_RESULT 1024

int main(int argc, char **argv){
    MPIComm client;
    MPI_Status status;
    char port_name[MPLMAX_PORT_NAME];
    int size, again;
    char text[SIZE_RESULT];
    char path[SIZE_RESULT];
    FILE *fp;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPLCOMM_WORLD, &size);
    if (size != 1) {
        fprintf(stderr, "Server too big");
        exit(EXIT_FAILURE);
    }

    MPI_Open_port(MPI_INFO_NULL, port_name);
    printf("Server available at port: %s\n", port_name);
    while (1) {
        MPI_Comm_accept(port_name, MPI_INFO_NULL,
            0, MPLCOMM_WORLD, &client);

        again = 1;
        while (again) {
            // Server receives here
            MPI_Recv(&text, SIZE_RESULT, MPL_CHAR,
                MPL_ANY_SOURCE, TAG_CLIENT_MESSAGE, client, &status);
            switch (status.MPITAG) {
                case 0:
                    MPI_Comm_free(&client);
                    MPI_Close_port(port_name);
                    MPI_Finalize();
                    return 0;
                case 1:
                    MPI_Comm_disconnect(&client);
                    again = 0;
                    break;
                case 2: // server job here
                    printf(">server: %s\n", text);

                    // Server run command
                    char result[SIZE_RESULT] = "";
                    fp = popen(text, "r");
```

```

        // Read the output a line at a time - output it
        while (fgets(path, sizeof(path), fp) != NULL) {
            printf("%s", path);
            strcat(result, path);
        }

        // Server send output to client
        MPI_Send(&result, strlen(result) + 1,
        MPLCHAR, 0, TAG_SERVER_RESULT, client);

        // Close fp
        pclose(fp);

        break;
    default:
        // Unexpected message type
        MPI_Abort(MPLCOMM_WORLD, 1);
    }
}
}
}

```

4.5 Client side

The client side is more simple, it just needs to make a connection to the server and sends and listens for results:

- `MPI_Init(&argc, &argv);` - Initializes MPI
- `MPI_Comm_connect(port_name, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &server);` Connects to the port argument
- `MPI_Send(&text, strlen(text) + 1, MPI_CHAR, 0, TAG_CLIENT_MESSAGE, server);` Sends commands
- `MPI_Recv(&result, 1000, MPI_CHAR, MPI_ANY_SOURCE, TAG_SERVER_RESULT, server, MPI_STATUS_IGNORE);` Receives results
- `MPI_Comm_disconnect(&server);` and `MPI_Finalize();` Disconnects and ends connections

Full client.c code:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#define TAG_CLIENT_MESSAGE 2
#define TAG_SERVER_RESULT 3
#define SIZE_RESULT 1024

int main( int argc, char **argv ){

```



```

MPIComm server;
MPI_Status status;
char port_name[MPLMAXPORTNAME];
char text[SIZE_RESULT];
char result[SIZE_RESULT];
double t1, t2;

if (argc < 2) {
    fprintf(stderr, "server_port_name_required.\n");
    exit(EXIT_FAILURE);
}

MPI_Init(&argc, &argv);
strcpy(port_name, argv[1]);
MPI_Comm_connect(port_name, MPIINFO_NULL,
0, MPLCOMM_WORLD, &server);

while (1) {
    printf(">client:_");
    scanf("%[^\\n]%%c", text);

    // Client sends here
    t1 = MPI_Wtime();
    MPI_Send(&text, strlen(text) + 1, MPLCHAR,
0, TAG_CLIENT_MESSAGE, server);

    // Client receives here
    MPI_Recv(&result, SIZE_RESULT, MPLCHAR,
MPLANY_SOURCE, TAG_SERVER_RESULT, server, MPI_STATUS_IGNORE);

    t2 = MPI_Wtime();
    printf("(%.28fms)_\\n%s\\n", t2 - t1, result);
}

MPI_Comm_disconnect(&server);
MPI_Finalize();
return 0;
}

```

4.6 Running the program

Compile the c files with:

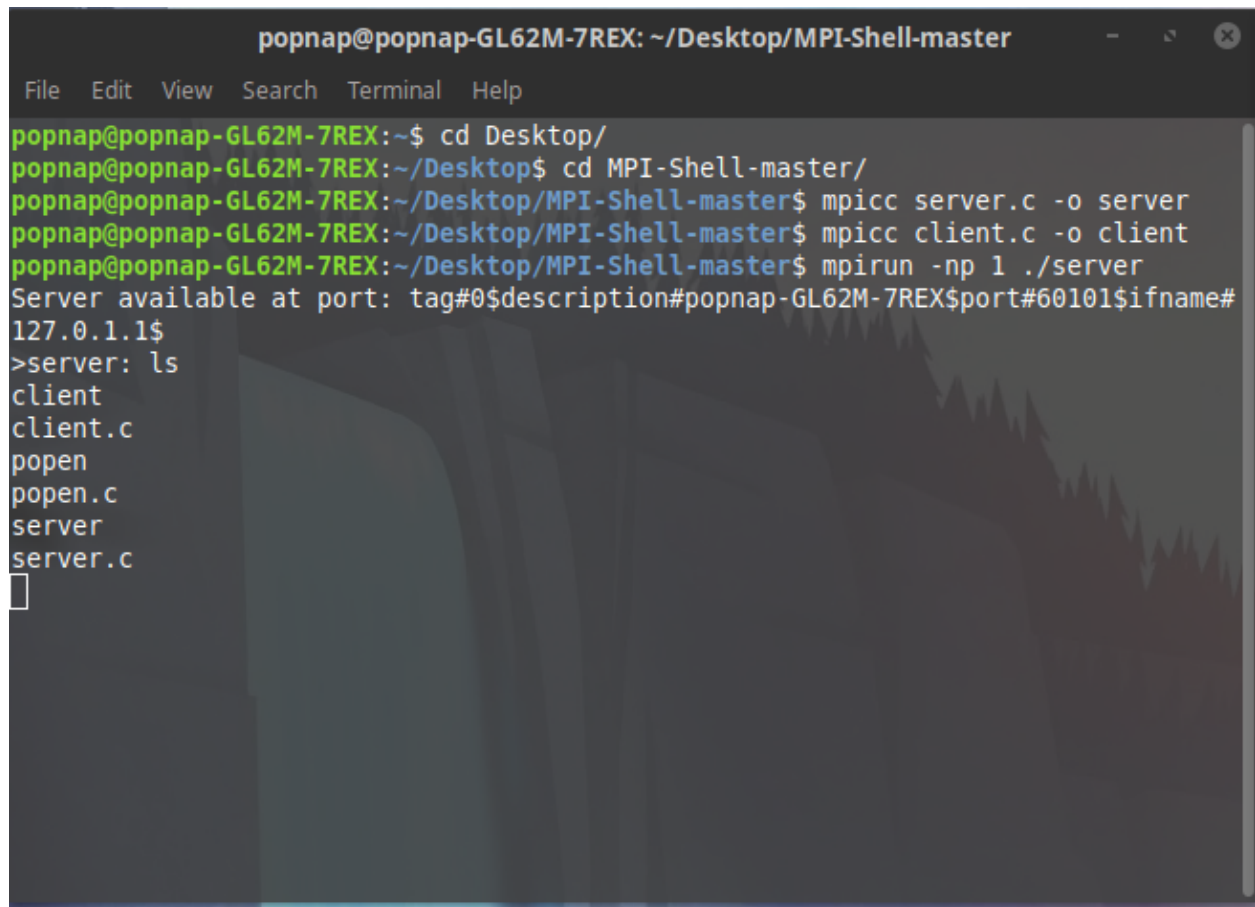
```

mpicc server.c -o server
mpicc client.c -o client

```

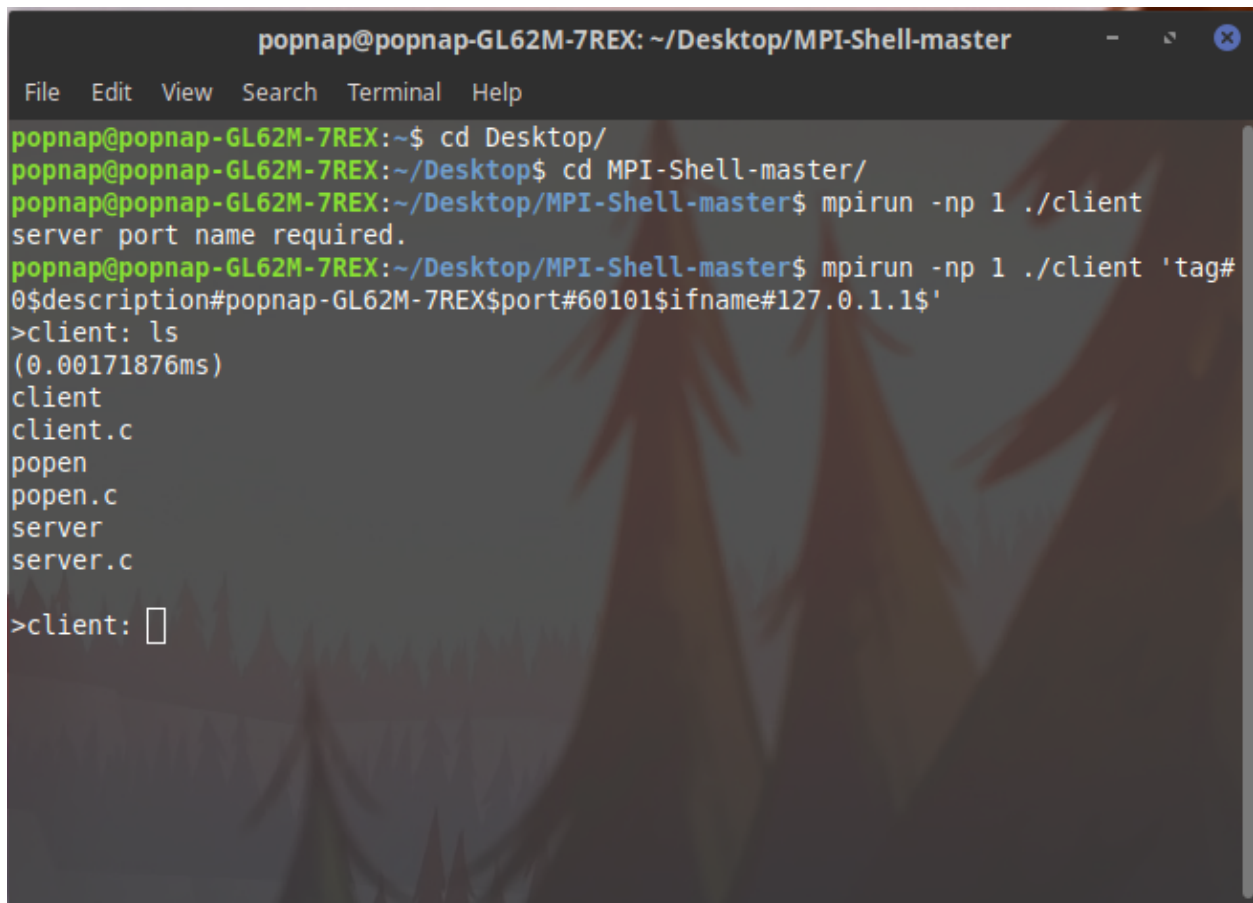
Run the files with mpirun

Server side:

A terminal window titled "popnap@popnap-GL62M-7REX: ~/Desktop/MPI-Shell-master" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the following commands and output:

```
popnap@popnap-GL62M-7REX:~$ cd Desktop/  
popnap@popnap-GL62M-7REX:~/Desktop$ cd MPI-Shell-master/  
popnap@popnap-GL62M-7REX:~/Desktop/MPI-Shell-master$ mpicc server.c -o server  
popnap@popnap-GL62M-7REX:~/Desktop/MPI-Shell-master$ mpicc client.c -o client  
popnap@popnap-GL62M-7REX:~/Desktop/MPI-Shell-master$ mpirun -np 1 ./server  
Server available at port: tag#0$description#popnap-GL62M-7REX$port#60101$ifname#  
127.0.1.1$  
>server: ls  
client  
client.c  
popen  
popen.c  
server  
server.c  
█
```

Client side:

A terminal window titled 'popnap@popnap-GL62M-7REX: ~/Desktop/MPI-Shell-master'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the following commands and output:

```
popnap@popnap-GL62M-7REX:~$ cd Desktop/
popnap@popnap-GL62M-7REX:~/Desktop$ cd MPI-Shell-master/
popnap@popnap-GL62M-7REX:~/Desktop/MPI-Shell-master$ mpirun -np 1 ./client
server port name required.
popnap@popnap-GL62M-7REX:~/Desktop/MPI-Shell-master$ mpirun -np 1 ./client 'tag#
0$description#popnap-GL62M-7REX$port#60101$ifname#127.0.1.1$'
>client: ls
(0.00171876ms)
client
client.c
popen
popen.c
server
server.c
>client: 
```

5 Evaluation

The factors which and affect MPI's performance is numerous, complex and interrelated. Because of this, generalizing the overall performance is difficult.

We will compare the program's response time on two different machines. The response time is calculated by subtracting two time stamps.

```
t1 = MPI_Wtime();
MPI_Send(&text, strlen(text) + 1, MPLCHAR, 0, TAG_CLIENT_MESSAGE, server);
MPI_Recv(&result, SIZE_RESULT, MPLCHAR, MPLANY_SOURCE,
```

```
TAG_SERVER_RESULT, server, MPI_STATUS_IGNORE);
t2 = MPI_Wtime();
Response time = t2 - t1
```

The specifications are:

MSI Laptop:

- Graphic card: GTX 1050Ti

- CPU: i7 HQ 7700
- RAM: 16 GB

Phuong's Laptop

- Graphic card: Intel HD Graphic 4000
- CPU: i5 3337U
- RAM: 4 GB

Table 1: Performance comparision

| | MSI | HP |
|---------|------------|-----------|
| ls | 0.001 s | 0.004 s |
| df -h | 0.02 s | 0.05 s |
| echo | 0.009 s | 0.01 s |
| mkdir | 0.001 s | 0.004 s |
| free -m | 0.03 s | 0.05 s |

6 Conclusion

6.1 Results

We did manage to mimic a remote shell through MPI however, the program runs locally on one machine. We did not take use of the multi-process structure of MPI and used, perhaps, a shortcut to perform this project, which is the use of `popen()`.

To improve this, in the future, we would like to build a shell that uses multiple processes on different nodes to truly take advantage of MPI.

6.2 References

References

- [1] Wikipedia
https://en.wikipedia.org/wiki/Message_passing_interface
- [2] Scott Pakin: MPI-Bash: Parallel scripting right from the Bourne-Again Shell (Bash).
<https://github.com/lanl/MPI-Bash>