UNIVERSIDAD DE COLIMA

Facultad de Telemática
Ing. en Software

Práctica #7
"Uso de funciones Send y Receive con MPI "

Ramírez García Gilberto Felipe
Valencia Sandoval Alfonso
Velasco Munguía Roberto Antonio
5°- "K"

Prof. Ramirez Alcaraz Juan Manuel

1 de Noviembre del 2019

Índice

Introducción

Para conocer más acerca del comportamiento de los clusters con la programación paralela, haremos la siguiente práctica en la cual se desarrollará y ejecutará un código/programa paralelo con MPI. El programa encadena el envío y recepción de un mensaje. El mensaje será el identificador del proceso multiplicado por 100. Los mensajes se enviarán de forma encadenada (pero no secuencial), lo que quiere decir que el primero enviará un mensaje al segundo, el segundo recibirá uno del primero y enviará uno al tercero, y así sucesivamente para todos los procesos lanzados. Todo proceso que reciba un mensaje debe imprimirlo de la forma “Hola, soy el proceso m y he recibido el número x de parte de n".

Desarrollo

Conexión con el cluster

El primer paso para realizar esta práctica será establecer conexión al cluster, tal como lo hemos visto en prácticas anteriores.

```
C:\Users\alfon>ssh curso13@148.228.4.17
curso13@148.228.4.17's password:
Last login: Wed Oct 30 09:59:52 2019 from 201.175.157.236
[curso13@rogueone ~]$
```

Para mantener orden en la práctica, creamos un directorio de trabajo para almacenar todos nuestros archivos y códigos de programación de este programa.

```
[curso13@rogueone messages-chain]$ pwd
/mnt/zfs-pool/home/curso13/Poncho/messages-chain
[curso13@rogueone messages-chain]$
```

Elaboración del programa

Escribimos nuestro programa en un archivo de texto con extensión “*.c*”. En nuestro caso utilizamos un editor de código local para poder hacer uso de herramientas de corrección y resaltado de código, por lo que posteriormente subiremos el archivo al cluster utilizando *scp*.

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

int main (int argc, char* argv[]) {
  int this_proc, total_procs;
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &total_procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &this_proc);

  int value = this_proc * 100;
  int incoming_message;

  if (this_proc != total_procs -1) {
    MPI_Send(&value, 1, MPI_INT, this_proc + 1, 0, MPI_COMM_WORLD);
  }
  if (this_proc != 0) {
    MPI_Status status;
    MPI_Recv(&incoming_message, 1, MPI_INT, this_proc - 1, 0, MPI_COMM_WORLD, &status);
    printf("Hola, soy el proceso %d y he recibido el número %d de parte de %d\n",
this_proc, incoming_message, this_proc - 1);
  }

  MPI_Finalize();
}
```

Explicación del código

La forma en la que resolvemos el problema planteado en esta práctica es la siguiente.
Habiendo inicializado la comunicación entre nodos con la librería MPI tendremos las variables *this_proc* y *total_procs*, los cuales indican el proceso actual y el total de procesos respectivamente.

El siguiente paso es crear el mensaje, que como el problema nos lo indica debe ser el identificador del proceso multiplicado por 100. Por lo que creamos la variable *value* la cual contendrá nuestro mensaje, almacenando el valor de *this_proc * 100*.

El siguiente paso es enviar el mensaje creado. Cada proceso debe enviar el mensaje al proceso posterior al actual, lo que significa que todos los procesos, menos el que tiene el índice (identificador) más alto, enviarán el mensaje. Para esto, la sentencia que se encarga de enviar el mensaje se encuentra dentro de una condición que compara que el índice del proceso actual sea desigual al último: *if (this_proc != total_procs - 1)*. Si la condición es verdadera entonces el proceso actual enviará el mensaje. Para enviar el mensaje utilizamos la función *MPI_Send* la cual pide 6 argumentos:

- Buffer: Referencia de memoria donde se almacena el o los elementos a ser enviados.
- Count: Número entero que indica la cantidad de elementos que están siendo enviados en el buffer.
- DataType: Indicador de que tipo de dato es el que está contenido en el buffer.
- Destiny: Número entero que indica el índice que identifica al proceso destino que recibirá el mensaje.

- Tag: Número entero para etiquetar el mensaje. El significado de esta etiqueta queda a necesidad del desarrollador.
- Communicator: Indicador del nombre de la comunicación en la que se encuentra el proceso destino.

Conociendo como funciona *MPI_Send* lo utilizamos de la siguiente forma *MPI_Send(&value, 1, MPI_INT, this_proc + 1, 0, MPI_COMM_WORLD);*
Pasando una referencia de *value* como mensaje e indicamos que si enviamos un solo elemento de tipo entero, el proceso destino será *this_proc + 1* ya que el mensaje debe ser enviado al proceso con el índice posterior al actual. Colocamos la etiqueta 0 (o cualquier número) ya que no hace falta el uso de etiquetas para este ejercicio, y por último indicamos que el comunicador será *MPI_COMM_WORLD* ya que en este serán inicializado los procesos.

El siguiente paso es recibir el mensaje, para posteriormente imprimirlo en pantalla. Cada proceso recibirá un mensaje del proceso con el índice anterior al actual, lo que quiere decir que todos los procesos menos el primero recibirán un mensaje. Para esto, la sentencia que se encarga de recibir el mensaje se encuentra dentro de una condición que compara que el índice del proceso actual sea desigual al primero, es decir desigual a 0: *if (this_proc != 0)*. Si la condición es verdadera entonces el proceso actual estará a la escucha para recibir un mensaje. Para recibir un mensaje utilizamos la función *MPI_Recv* de forma muy similar a la que enviamos mensajes, esta función pide 7 argumentos:

- Buffer: Referencia de memoria donde se almacenarán los datos recibidos.
- Count: Número entero que indica la cantidad de elementos que serán siendo recibidos en el buffer.
- DataType: Indicador de que tipo de dato es el que está contenido en el buffer.
- Source: Número entero que indica el índice que identifica al proceso que envía el mensaje que debe esperarse.
- Tag: Número entero para etiquetar el mensaje. El significado de esta etiqueta queda a necesidad del desarrollador.
- Communicator: Indicador del nombre de la comunicación en la que se encuentra el proceso destino.
- Status: Variable que almacena el estado de la recepción del mensaje.

Conociendo como funciona *MPI_Recv* lo utilizamos de la siguiente forma:
MPI_Recv(&incoming_message, 1, MPI_INT, this_proc - 1, 0, MPI_COMM_WORLD, &status);
Pasamos una referencia de la variable *incoming_message,* declarada anteriormente, donde se almacenará el mensaje esperado, indicamos que el mensaje solo contiene un elemento de tipo entero, y que este proviene del procesador con el índice anterior al del proceso actual, el mensaje que esperamos lleva la etiqueta 0, el mensaje será transmitido por el comunicador *MPI_COMM_WORLD* y el estado del mensaje se almacenará en la variable *status* definida justo antes de la función *MPI_Recv.*

La función *MPI_Recv* detendrá la ejecución del resto de instrucciones del proceso hasta no recibir el mensaje, por lo que al imprimir la variable *incomming_message* dicha instrucción no se ejecutará hasta que el mensaje recibido sea almacenado dentro de esta variable.

Desplegando nuestro trabajo en el cluster

Para subir nuestro programa al cluster utilizamos el comando *scp* de la siguiente forma:

```
scp ./message.c curso13@148.228.4.17:~/Poncho/messages-chain/
```

```
PS D:\Development\C> scp ./message.c curso13@148.228.4.17:~/Poncho/messages-ch
ain/
curso13@148.228.4.17's password:
message.c
                                              100%  698    12.9KB/s   00:00
PS D:\Development\C>
```

Y comprobamos que el archivo esté dentro de nuestro directorio de trabajo en el cluster.

```
[curso13@rogueone messages-chain]$ ls *.c
message.c
[curso13@rogueone messages-chain]$
```

Posteriormente compilamos nuestro programa para que pueda ser ejecutado:

```
[curso13@rogueone messages-chain]$ module load Compilers/Parallel-Studio-XE-2018
[curso13@rogueone messages-chain]$ mpiicc message.c -o message
[curso13@rogueone messages-chain]$ ls
machines.LINUX  message  message.c  mpi-messagechain.log  mpi-messagechain.out  mpi-run.sh
[curso13@rogueone messages-chain]$
```

Ejecución del programa

Esta parte se realizará igual que en prácticas anteriores, por lo que no entraremos en detalle del script, ya que lo único diferente en este son los nombres del directorio de trabajo y los archivos de salidas de datos. El punto importante a mencionar es que lo correremos 3 veces:

- Utilizando 4 nodos y 3 procesadores.
- Utilizando 4 nodos y 4 procesadores.
- Utilizando 4 nodos y 5 procesadores.

12 procesadores

```
[curso13@rogueone messages-chain]$ cat mpi-messagechain.out
Hola, soy el proceso 10 y he recibido el número 900 de parte de 9
Hola, soy el proceso 1 y he recibido el número 0 de parte de 0
Hola, soy el proceso 2 y he recibido el número 100 de parte de 1
Hola, soy el proceso 11 y he recibido el número 1000 de parte de 10
Hola, soy el proceso 5 y he recibido el número 400 de parte de 4
Hola, soy el proceso 6 y he recibido el número 500 de parte de 5
Hola, soy el proceso 7 y he recibido el número 600 de parte de 6
Hola, soy el proceso 9 y he recibido el número 800 de parte de 8
Hola, soy el proceso 3 y he recibido el número 200 de parte de 2
Hola, soy el proceso 4 y he recibido el número 300 de parte de 3
Hola, soy el proceso 8 y he recibido el número 700 de parte de 7
```

16 procesadores

```
[curso13@rogueone messages-chain]$ cat mpi-messagechain.out
Hola, soy el proceso 11 y he recibido el número 1000 de parte de 10
Hola, soy el proceso 15 y he recibido el número 1400 de parte de 14
Hola, soy el proceso 3 y he recibido el número 200 de parte de 2
Hola, soy el proceso 8 y he recibido el número 700 de parte de 7
Hola, soy el proceso 7 y he recibido el número 600 de parte de 6
Hola, soy el proceso 12 y he recibido el número 1100 de parte de 11
Hola, soy el proceso 4 y he recibido el número 300 de parte de 3
Hola, soy el proceso 10 y he recibido el número 900 de parte de 9
Hola, soy el proceso 9 y he recibido el número 800 de parte de 8
Hola, soy el proceso 14 y he recibido el número 1300 de parte de 13
Hola, soy el proceso 13 y he recibido el número 1200 de parte de 12
Hola, soy el proceso 2 y he recibido el número 100 de parte de 1
Hola, soy el proceso 1 y he recibido el número 0 de parte de 0
Hola, soy el proceso 6 y he recibido el número 500 de parte de 5
Hola, soy el proceso 5 y he recibido el número 400 de parte de 4
```

20 procesadores

```
[curso13@rogueone messages-chain]$ cat mpi-messagechain.out
Hola, soy el proceso 11 y he recibido el número 1000 de parte de 10
Hola, soy el proceso 19 y he recibido el número 1800 de parte de 18
Hola, soy el proceso 3 y he recibido el número 200 de parte de 2
Hola, soy el proceso 7 y he recibido el número 600 de parte de 6
Hola, soy el proceso 15 y he recibido el número 1400 de parte de 14
Hola, soy el proceso 4 y he recibido el número 300 de parte de 3
Hola, soy el proceso 8 y he recibido el número 700 de parte de 7
Hola, soy el proceso 12 y he recibido el número 1100 de parte de 11
Hola, soy el proceso 16 y he recibido el número 1500 de parte de 15
Hola, soy el proceso 9 y he recibido el número 800 de parte de 8
Hola, soy el proceso 18 y he recibido el número 1700 de parte de 17
Hola, soy el proceso 17 y he recibido el número 1600 de parte de 16
Hola, soy el proceso 2 y he recibido el número 100 de parte de 1
Hola, soy el proceso 1 y he recibido el número 0 de parte de 0
Hola, soy el proceso 6 y he recibido el número 500 de parte de 5
Hola, soy el proceso 5 y he recibido el número 400 de parte de 4
Hola, soy el proceso 10 y he recibido el número 900 de parte de 9
Hola, soy el proceso 13 y he recibido el número 1200 de parte de 12
Hola, soy el proceso 14 y he recibido el número 1300 de parte de 13
```

Conclusión

Un punto para observar es que los mensajes en consola no se imprimen en orden. No obstante, la ejecución de ellos es ordenada, porque cada procesador depende del anterior, a excepción del procesador 0 que inicia los mensajes. Nosotros pensamos que esto se debe a que la instrucción de imprimir, la función printf, tarda diferente tiempo en cada procesador. Se imprimen en el orden en estos que llegan a consola. Por este motivo, se imprimen en desorden.

En conclusión, en esta práctica aprendimos a comunicar a nuestros distintos procesos, lo que nos abre las puertas a poder resolver problemas de mayor complejidad. Este ejercicio resuelve un problema sencillo, pero deja en claro en cómo funciona el pase de mensajes entre procesos.