Paradigmas y Lenguajes de Programación
Trabajo Práctico Número 3
– Práctica –

Ulises C. Ramirez [uli.r19@gmail.com]
Héctor Chripczuk
González Verónica

28 de Septiembre, 2018

Código

Todo el código que esté expresado en el documento como respuesta a algún ejercicio está contenido en la carpeta `codigo`, junto a el se podrá encontrar la salida del compilador.

Versionado

Para el corriente documento se está llevando un versionado a fin de mantener un respaldo del trabajo y además proveer a la cátedra o a cualquier interesado la posibilidad de leer el material en la última versión disponible.

Repositorio: *https://github.com/ulisescolina/UC-PYLP*

–Ulises

Índice de Contenído

1 Ejercicio 1 **1**

2 Ejercicio 2 **2**

3 Anexo **8**

3.1 Acerca de la compilación 8

1 Ejercicio 1

CONSIGNA: **resolver el ejercicio planteado en la página de la [Universidad de Granada], Tutorial 01 Hello World.**

- **Explique brevemente como funciona el código fuente.**
- **Describa las instrucciones propias de MPI utilizadas en el ejercicio. ¿Cuáles son los parámetros de entrada y cuáles son los parámetros de salida?**

En primera instancia, accedemos a la página brindada por la cátedra, y obtenemos un archivo base con el cual trabajar.

Código 1: Código base para Hello World desde [Universidad de Granada]

```
1  /*
2   ============================================================
3   Name: holamundo1.cpp
4   Author: Sergio Rodríguez Lumley & Daniel Guerrero Martínez
5   Version:
6   Copyright: GNU Open Source and Free license
7   Description: Tutorial 1. ¡Hola Mundo! Inicial
8   ============================================================
9   */
10
11 #include <iostream>
12 #include <mpi.h>
13 using namespace std;
14
15 int main(int argc, char *argv[])
16 {
17     cout<<"¡Hola Mundo soy proceso unico! "<<endl;
18     return 0;
19 }
```

De ahora en adelante se eliminan las primeras 10 lineas del código expuesto en `Código 1`, dado a que simplemente son comentarios, la razon de que se los haya presentado es porque estos demuestran los datos del creador.

La página tiene desde ya una ayuda que permite conocer que sentencias se necesitan para la correcta implementacion de lo que solicita el ejercicio. Se menciona que los necesario para un funcionamiento minimo de `OpenMPI` son las siguientes instrucciones, según [Clase 3 PyLP - OpenMPI, 2018]:

- **MPI_Init**: este comando inicializa el entorno MPI.
- **MPI_Finalize**: finaliza el entorno de MPI.
- **MPI_Comm_size**: determina la cantidad de procesos del comunicador.
- **MPI_comm_rank**: representa el id de un proceso.

La implementación de lo solicitado se puede ver en el `Código 2`, la explicación de cada linea se presenta luego.

Código 2: Solución ejercicio 1

```
1  /*
2    NUM_HILOS=4
3  */
4  #include <iostream>
5  #include <mpi.h>
6  using namespace std;
7
8  int main(int argc, char *argv[])
9  {
10   int tamanio, id;
11   MPI_Init(&argc, &argv);
12   MPI_Comm_size(MPI_COMM_WORLD, &tamanio);
13   MPI_Comm_rank(MPI_COMM_WORLD, &id);
14   cout<<"¡Hola Mundo soy el proceso "<< id << " de "
15            << tamanio << endl;
16   MPI_Finalize();
17   return 0;
18 }
```

Para información acerca de la compilación del archivo, junto a una explicación relacionada a la `Línea 2` del `Código 2` puede revisar lo presentado en el `Anexo 3.1`

Las diferencias con el `Código 1` inician en la linea `10`, aquí definimos variables en las cuales vamos a guradar información acerca de lo que va sucediendo a medida que se ejecuta el código, es importante aclarar que estas variables son globales y estan disponibles para todas las instancias de los procesos, de esta forma los mismos pueden comunicarse. En la línea `11` iniciamos con lo relacionado a MPI, aquí se inicializa el entorno, mediante `MPI_Init`, en la línea `12` se establece un tamaño para el *comunicador* (lugar que contiene, por defecto, todos los procesos), consecuentemente, esto dará la cantidad de procesos a ser instanciados. en la siguiente línea, lo que se hace es obtener el `id` del proceso actual, teniendo esta información ya podemos empezar a imprimir lo que nos solicita el ejercicio, esto se hace en las líneas `14` y `15`, y por último se finaliza en la línea `16` con el entorno MPI.

2 Ejercicio 2

Consigna: **Queremos hacer un programa paralelo que encadene el envío y recepción de un mensaje, en nuestro caso el mensaje sera el rango (identificador) del proceso que envía. Los mensajes se enviarán de forma encadenada, 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 "Soy el proceso X y he recibido M", siendo X el rango del proceso y M el mensaje recibido.**

Antes de presentar el código cabe mencionar que la consigna no se sigue al pie de la letra, se modifica un poco en cuanto al valor que envia, siendo este la variable `mensaje` la cual es un entero que se define a la hora de mandarla la primera vez. Y luego se reenvía el mismo entero pero se le adiciona una unidad es decir `mensaje + 1`.

Código 3: Solución ejercicio 2

```
/*
  NUM_PROC=4
*/
#include <iostream>
#include <mpi.h>

using namespace std;
int main(int argc, char *argv[])
{
  int tamanio, id, buzon, mensaje;
  bool realiza_envio = false;
  MPI_Status estado;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &tamanio);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

  if (id == 0) {
    mensaje = 1000;
    MPI_Send(&mensaje, /* Mensaje a enviar */
      1, /*Cantidad de elementos*/
      MPI_INT, /* Tipo de dato del mensaje*/
      id+1, /*Proceso receptor del mensaje*/
      0, /* Etiqueta del mensaje */
      MPI_COMM_WORLD /* Comunicador */);
    cout << "Soy el proceso " << id << ", no recibo nada pero "
    <<"estoy enviando el valor "<< mensaje << " a " << id+1 << endl;
  } else {
    MPI_Recv(&buzon, /* Almacenamiento de msj */
       1, /* Cantidad de elementos
        que se esta recibiendo */
       MPI_INT, /* Tipo de dato a recibir*/
       id-1, /* id del proceso origen */
       0, /* Etiqueta esperada para msj */
       MPI_COMM_WORLD, /* Comunicador utilizado */
       &estado /* Info del estado */);
    if (id != tamanio-1) {
      mensaje = buzon+1;
      MPI_Send(&mensaje, /* Mando mensaje que recibo */
        1, /*Cantidad de elementos*/
        MPI_INT, /* Tipo de dato del mensaje*/
        id+1, /*Proceso receptor del mensaje*/
        0, /* Etiqueta del mensaje */
        MPI_COMM_WORLD /* Comunicador */);
      realiza_envio = true;
    }

    if (realiza_envio) {
      cout << "Soy el proceso " << id << ", recibi el valor "
```

```
50        << buzon <<" del proceso " << id-1<< " y estoy enviando el valor "
51        << mensaje << " al proceso " << id+1 << endl;
52      } else {
53        cout << "Soy el proceso " << id << ", recibi el valor "
54        << buzon <<" del proceso "<< id-1 <<" pero no tengo nada "
55        << "que enviar"<< endl;
56      }
57    }
58    MPI_Finalize();
59    return 0;
60  }
```

Los conceptos principales dentro del entorno MPI vistos en este ejercicio son dos, `MPI_Send` y `MPI_Recv`, estos complementan a los comandos introductorios vistos en la `Sección 1`, y la utilidad que proveen es la de enviar y recibir mensajes respectivamente. La utilización con respecto a la sintáxis y errores que se pueden esperar arrojen los comandos en cuestión están explicados en la página de la [Universidad de Granada] en el apartado correspondiente para MPI_Send y MPI_Recv, y [Clase 3 PyLP - OpenMPI, 2018]. Para no ser redundante, acá solo se va a explicar que está haciendo cada uno de ellos en el ámbito que propone el ejercicio.

Código 4: Fragmento de `Código 3`

```
18  if (id == 0)
```

Lo expuesto anteriormente en el `Código 4` es una enseñanza tomada de cátedras como Sistemas Operativos, en la cual tratabamos con procesos y era de vital importancia conocer su `id`, aqui es igual de importante, comenzando por lo solicitado en el ejercicio se sabe que no se desea tener un `anillo` de procesos, más bien se prefiere una simple `cola` de ellos como se representa en `Figura 1`.

Figure 1: Concepto de lo requerido en el Ejercicio 2

Entonces, es vital que conozcamos cual es el proceso `P0` en este caso, dado a que este no debera recibir nada, unicamente enviará un mensaje al siguiente proceso y eso concluirá con su tarea, que es exactamente lo que está pasando en el `Código 5`

Código 5: MPI_Send para P0 - Fragmento de `Código 3`

```
19    mensaje = 1000;
20    MPI_Send(&mensaje, /* Mensaje a enviar */
21      1, /*Cantidad de elementos*/
22      MPI_INT, /* Tipo de dato del mensaje*/
23      id+1, /*Proceso receptor del mensaje*/
```

```
24        0, /* Etiqueta del mensaje */
25        MPI_COMM_WORLD /* Comunicador */);
```

Allí evidenciamos lo que realiza el proceso `P0` (conceptualización presentada en la `Figura 2`), inicializamos una variable `mensaje=1000` que será lo que enviaremos, al siguiente proceso que sera el proceso `P1` ya que está representado por `id+1` en el proceso con id=0, luego este proceso concluirá diciendonos que envió dicho valor que declaramos, junto a su id, el de su destinatario, y que no debe realizar ninguna acción más.

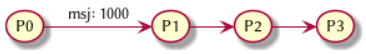

Figure 2: Conceptualización de `Código 5`

Siguiendo por el `else` del `if` presentado en el `Código 4` tendremos el siguiente código que lo vamos a presentar en 2 partes, en la primera vamos a ver otro de los conceptos que nos ocupan en la sección y en la segunda vamos a tratar con lo mismo que tratamos cuando hablamos del proceso `P0`.

Código 6: 1ra parte MPI_Recv - Fragmento de `Código 3`

```
29    MPI_Recv(&buzon, /* Almacenamiento de msj */
30       1, /* Cantidad de elementos
31        que se esta recibiendo */
32       MPI_INT, /* Tipo de dato a recibir*/
33       id-1, /* id del proceso origen */
34       0, /* Etiqueta esperada para msj */
35       MPI_COMM_WORLD, /* Comunicador utilizado */
36       &estado /* Info del estado */);
```

El `Código 6` presenta la aplicación de `MPI_Recv`, y lo que se está haciendo es guardando lo que envió P0 como se demuestra en la `Figura 2`, y lo está almacenando en una variable que se denomió `buzon`, se representa el curso de acción en la `Figura 3`.

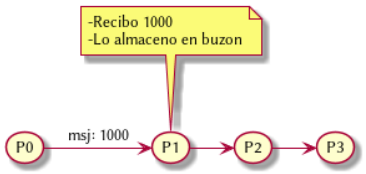

Figure 3: Conceptualización de `Código 6`

Ahora se prosigue con la segunda parte de la explicación e iniciamos preguntándonos que significa la condición en el `Código 7`

Código 7: Condición para detectar último proceso

```
37   if (id != tamanio-1)
```

Recordando la Figura 1, y el hecho de que no se necesita un anillo para el ejercicio, es necesario saber cual es el último nodo, además de conocer el primero, esto se logra mediante el Código 7 en este caso, y a diferencia del primero, el último nodo recibe un mesaje del nodo predecesor, pero no comunica ninguno. Esta acción se codificó de la siguiente manera

Código 8: 2da parte MPI_Recv - Fragmento de Código 3

```
38       mensaje = buzon+1;
39       MPI_Send(&mensaje, /* Mando mensaje que recibo */
40         1, /*Cantidad de elementos*/
41         MPI_INT, /* Tipo de dato del mensaje*/
42         id+1, /*Proceso receptor del mensaje*/
43         0, /* Etiqueta del mensaje */
44         MPI_COMM_WORLD /* Comunicador */);
45       realiza_envio = true;
```

Conceptualizando el Código 8, se concluye con un escenario como lo presentado en la Figura 4.

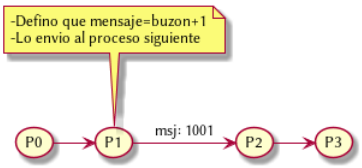

Figure 4: Conceptualización de Código 8

realiza_envio es una variable booleana que permite tener idea de si, el proceso que se está ejecutando envió algo o solo recibió algo, de esta manera se puede mostrar un mensaje mas personalizado a la hora de la ejecución del código.

Para finalizar se muestra una captura con el código ejecutado en 4 procesos.

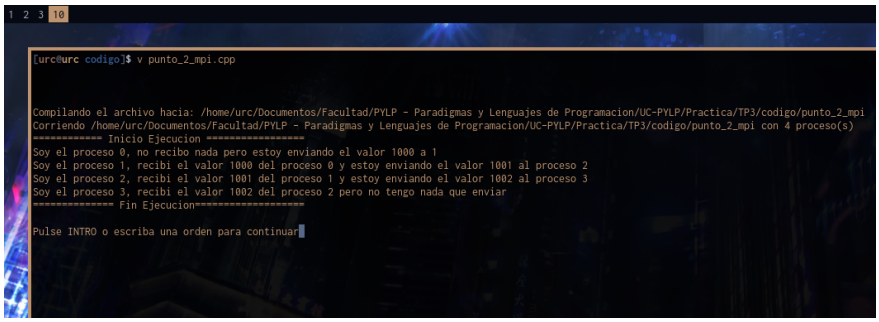

Figure 5: Captura de pantalla de la ejecución del `Código 3`

3 Anexo

3.1 Acerca de la compilación

para la compilación de todo código relacionado con OpenMPI/OpenMP se utilizó el siguiente script:

Código 9: Script de compilación

```
#!/bin/sh

#Variables globales
# @param $1 archivo con el codigo fuente
# @param $2 nombre base del archivo, sin la extension
archivo=$1
base=$2

# Compila codigo OpenMPI en C++ y lo ejecuta con la cantidad
# de recursos que se le soliciten
compi() {\
 txtcpp=$(sed 10q "$archivo")
 if [[ $txtcpp =~ (NUM_PROC=)([0-9]) ]];
 then
    nproc=${BASH_REMATCH[2]};
 else
    nproc=1;
 fi
 echo && echo
 echo "Compilando el archivo hacia: $base"
 mpicxx "$archivo" -o "$base"
 echo "Corriendo $base con $nproc proceso(s)"
 echo "============ Inicio Ejecucion ================="
 mpirun -np "$nproc" "$base"
 echo "============== Fin Ejecucion==================="
}

# Compila codigo OpenMP utilizando el compilador para C++,
# luego lo ejecuta
comp(){
 g++ -fopenmp "$archivo" -o "$base" && "$base"
}
```

Se aprecian dos funciones una `copmi()` específica para la compilación de código que necesite los wrappers para OpenMPI, y una segunda denominada `comp()` que realiza algo similar, pero para OpenMP.

Una peculiaridad a tener en cuenta, es la forma en la que `compi()` trata al archivo, este busca en las primeras 10 lineas una cadena que contenga lo siguiente: "NUM_PROC=*numero_de_procesos*" y lo recuerda. Luego de ralizar la compilación del código, ejecuta al archivo resultante de la compilación mediante `mpirun -np «numero_de_procesos» «nombre_archivo»` (de no encontrar una expresión con esas características, procede a ejecutar el archivo compilado con un solo proceso), lo cual hace que compilación y ejecución se realicen en un solo paso para el usuario.

Referencias

[Universidad de Granada] Universidad de Granada. https://lsi.ugr.es/.

[Clase 3 PyLP - OpenMPI, 2018] Paradigmas y Lenguajes de Programación. *Presentacionde de la Clase 3: OpenMPI.*