

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 Contenido

1	Ejercicio 1	1
2	Ejercicio 2	2
3	Ejercicio 3	7
4	Ejercicio 4	9
5	Ejercicio 5	11
6	Ejercicio 6	14
7	Anexo	15
7.1	Acerca de la compilación	15

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 líneas del código expuesto en Código 1, dado a que simplemente son comentarios, la razón 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 implementación de lo que solicita el ejercicio. Se menciona que lo necesario para un funcionamiento mínimo 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 línea 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 tamano, id;
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &tamano);
13     MPI_Comm_rank(MPI_COMM_WORLD, &id);
14     cout<<"Hola Mundo soy el proceso "<< id << " de "
15         << tamano << 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 7.1

Las diferencias con el Código 1 inician en la línea 10, aquí definimos variables en las cuales vamos a guardar información acerca de lo que va sucediendo a medida que se ejecuta el código, es importante aclarar que estas variables son globales y están 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 será 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 envía, 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

```

1  /*
2     NUM_PROC=4
3  */
4  #include <iostream>
5  #include <mpi.h>
6
7  using namespace std;
8  int main(int argc, char *argv[])
9  {
10     int tamano, id, buzon, mensaje;
11     bool realiza_envio = false;
12     MPI_Status estado;
13
14     MPI_Init(&argc, &argv);
15     MPI_Comm_size(MPI_COMM_WORLD, &tamano);
16     MPI_Comm_rank(MPI_COMM_WORLD, &id);
17
18     if (id == 0) {
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 */);
26         cout << "Soy el proceso " << id << ", no recibo nada pero "
27             <<"estoy enviando el valor " << mensaje << " a " << id+1 << endl;
28     } else {
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 */);
37         if (id != tamano-1) {
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;
46         }
47
48         if (realiza_envio) {
49             cout << "Soy el proceso " << id << ", recibí 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 << ", recibí 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 sintaxis 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 tratábamos con procesos y era de vital importancia conocer su `id`, aquí 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 deberá recibir nada, únicamente 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 denominó `buzon`, se representa el curso de acción en la Figura 3.

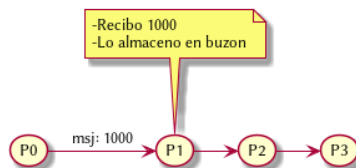


Figure 3: Conceptualización de Código 6

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 != tamaño-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 mensaje 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 = buzón+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.

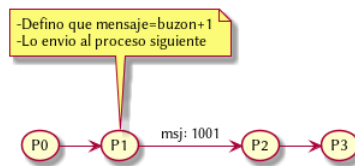


Figure 4: Conceptualización de Código 8

`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.


```

1 2 3 10
[urc@urc codigo]$ v punto_2_mpi.cpp

Compilando el archivo hacia: /home/urc/Documentos/Facultad/PYLP - Paradigmas y Lenguajes de Programacion/UC-PYLP/Practica/TP3/codigo/punto_2_mpi
Corriendo /home/urc/Documentos/Facultad/PYLP - Paradigmas y Lenguajes de Programacion/UC-PYLP/Practica/TP3/codigo/punto_2_mpi con 4 proceso(s)
===== Inicio Ejecucion =====
Soy el proceso 0, no recibo nada pero estoy enviando el valor 1000 a !
Soy el proceso 1, recibi el valor 1000 del proceso 0 y estoy enviando el valor 1001 al proceso 2
Soy el proceso 2, recibi el valor 1001 del proceso 1 y estoy enviando el valor 1002 al proceso 3
Soy el proceso 3, recibi el valor 1002 del proceso 2 pero no tengo nada que enviar
===== Fin Ejecucion=====

Pulse INTRO o escriba una orden para continuar

```

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

3 Ejercicio 3

CONSIGNA: adapte el ejercicio 2, al siguiente planteamiento, el proceso 0, envía dos mensajes un mensaje 11111 al proceso 1 y un mensaje 22222 al proceso 2, luego el mensaje 11111 debe ir pasando por los procesos impares y el mensaje 22222 debe ir pasando por los procesos pares.

Código 9: Solución al Ejercicio 3

```

1  /*
2  NUM_PROC=8
3  */
4  #include <iostream>
5  #include <mpi.h>
6
7  using namespace std;
8  int main(int argc, char *argv[])
9  {
10     int tamano, id, buzon, mensaje=0, i, p_destino;
11     MPI_Status estado;
12
13     MPI_Init(&argc, &argv);
14     MPI_Comm_size(MPI_COMM_WORLD, &tamano);
15     MPI_Comm_rank(MPI_COMM_WORLD, &id);
16
17     if (id == 0) {
18         for (i=0; i<2; i++) {
19             mensaje += 11111;
20             p_destino = i+1;
21             MPI_Send(&mensaje, /* Mensaje a enviar */
22                     1, /*Cantidad de elementos*/
23                     MPI_INT, /* Tipo de dato del mensaje*/
24                     p_destino, /*Proceso receptor del mensaje*/
25                     0, /* Etiqueta del mensaje */
26                     MPI_COMM_WORLD /* Comunicador */);
27             cout << "Soy el proceso 0, estoy enviando el valor "
28                  << mensaje << " al proceso " << p_destino << endl;
29         }

```

```

30 } else {
31     if (id==1 || id==2) {
32         /*Recibo el mensaje de P0*/
33         /*y lo guardo en buzón*/
34         MPI_Recv(&buzon, 1, MPI_INT, 0,
35                 0, MPI_COMM_WORLD, &estado);
36         cout<<"Soy el proceso "<<id<<" recibí el valor "
37         <<buzon<<" del proceso 0."<<endl;
38         for (i=id+2; i < tamaño; i+=2) {
39             /*Envío los mensajes a procesos*/
40             /*pares e impares*/
41             MPI_Send(&buzon,1,MPI_INT,i,0,MPI_COMM_WORLD);
42             cout<<"P"<<id<<": envío el valor " <<
43             buzon<<" a " <<i<<endl;
44         }
45     } else {
46         if (id % 2 != 0) {
47             /*Recibo el mensaje enviado*/
48             /*por el proceso P1*/
49             MPI_Recv(&buzon, 1, MPI_INT, 1,
50                     0, MPI_COMM_WORLD, &estado);
51             cout<<"P"<<id<<": recibo el valor " <<
52             buzon<<" de 1."<<endl;
53         } else {
54             /*Recibo el mensaje enviado*/
55             /*por el proceso P2*/
56             MPI_Recv(&buzon, 1, MPI_INT, 2,
57                     0, MPI_COMM_WORLD, &estado);
58             cout<<"P"<<id<<": recibo el valor " <<
59             buzon<<" de 2."<<endl;
60         }
61     }
62 }
63 MPI_Finalize();
64 return 0;
65 }

```

A continuación se presenta una captura de pantalla para la ejecución del Código 9.

```

1 2 3 10
[urc@urc codigo]$ v punto_3_mpi.cpp

Compilando el archivo hacia: /home/urc/Documentos/Facultad/PYLP - Paradigmas y Lenguajes de Programacion/UC-PYLP/Practica/TP3/codigo/punto_3_mpi
Corriendo /home/urc/Documentos/Facultad/PYLP - Paradigmas y Lenguajes de Programacion/UC-PYLP/Practica/TP3/codigo/punto_3_mpi con 8 proceso(s)
===== Inicio Ejecucion =====
Soy el proceso 0, estoy enviando el valor 11111 al proceso 1
Soy el proceso 0, estoy enviando el valor 22222 al proceso 2
Soy el proceso 1 recibí el valor 11111 del proceso 0.
P1: envío el valor 11111 a 3
Soy el proceso 2 recibí el valor 22222 del proceso 0.
P2: envío el valor 22222 a 4
P1: envío el valor 11111 a 5
P2: envío el valor 22222 a 6
P3: recibo el valor 11111 de 1.
P4: recibo el valor 22222 de 2.
P5: recibo el valor 11111 de 1.
P6: recibo el valor 22222 de 2.
P7: recibo el valor 11111 de 1.
P1: envío el valor 11111 a 7
===== Fin Ejecucion=====
Pulse INTRO o escriba una orden para continuar

```

Figure 6: Captura de pantalla de la ejecución del Código 9

4 Ejercicio 4

CONSIGNA: escribir un algoritmo que lanzando N procesos se genere un vector de tamaño N. Este vector debe ir pasandose entre los procesos y cada proceso deberá rellenar en la posición del vector de su número de rango, su número de rango si el proceso tiene rango impar y con su número de rango negativo si el proceso tiene número de rango par.

En primera instancia, a la hora de la realización del ejercicio una cuestión no quedó del todo clara, esta cuestión tiene que ver con que si hay que imprimir el estado del vector en cada proceso, es decir, que cada proceso reporte el estado del vector cuando éste opere en él, así que se decidió dar la responsabilidad de la comunicación del estado del vector al último proceso que manipule a dicho arreglo.

A continuación se presenta el código para la solución del Ejercicio 4 junto con una captura de pantalla de la ejecución del código.

Código 10: Solución al Ejercicio 4

```

1  /*
2   NUM_PROC=10
3  */
4  #include <iostream>
5  #include <mpi.h>
6  using namespace std;
7  /*
8   * Utilidad para imprimir el vector
9   * @param *vec puntero con la posición
10  * del arreglo en memoria.
11  * @param t cantidad de procesos solicitados
12  * al entorno MPI
13  */
14 void imprimir_vector(int *vec, int t) {
15     int i=0;

```

```

16     cout << "["<<flush;
17     for (i; i <= t-2;i++) {
18         cout<<vec[i]<< " , "<<flush;
19     }
20     cout<<vec[t-1]<<"]"<<endl;
21 }
22
23 int main(int argc, char *argv[])
24 {
25     int tamano, id;
26     MPI_Status estado;
27
28     MPI_Init(&argc, &argv);
29     MPI_Comm_size(MPI_COMM_WORLD, &tamano);
30     MPI_Comm_rank(MPI_COMM_WORLD, &id);
31
32     int vec[tamano];
33     int cant_elementos = id;
34     int p_destino = id+1;
35     /*
36      El primer proceso, P0, no espera recibir nada
37      entonces a este no se le solicita que realice
38      MPI_Recv().
39     */
40     if (id != 0) {
41         MPI_Recv(&vec, cant_elementos, MPI_INT, id-1,
42             0, MPI_COMM_WORLD, &estado);
43     }
44     /*
45      Si el rango es par se carga -rango
46      si el rango es impar se carga rango
47     */
48     if (id % 2 == 0) {
49         vec[id] = -id;
50     } else {
51         vec[id] = id;
52     }
53     /*
54      El ultimo proceso, PN, no espera enviar nada
55      entonces a este no se le solicita que realice
56      MPI_Send().
57      Sin embargo, se hace a este proceso, el ultimo,
58      responsable de la impresion de la version
59      final del vector.
60     */
61     if (id < tamano-1) {
62         MPI_Send(&vec, /* Mensaje a enviar */
63             cant_elementos+1, /*Cantidad de elementos*/
64             MPI_INT, /* Tipo de dato del mensaje*/
65             p_destino, /*Proceso receptor del mensaje*/
66             0, /* Etiqueta del mensaje */
67             MPI_COMM_WORLD /* Comunicador */);
68     } else {
69         cout<<"Soy el ultimo proceso.\n"<<
70             "Este es el vector final: "<<flush;
71         imprimir_vector(vec, tamano);
72     }

```

```

73 |
74 |     MPI_Finalize();
75 |     return 0;
76 | }

```

```

[urc@urc codigo]$ v punto_4_mpi.cpp
Compilando el archivo hacia: /home/urc/Documentos/Facultad/PVLP - Paradigmas y Lenguajes de Programacion/UC-PVLP/Practica/TP3/codigo/punto_4_mpi
Corriendo /home/urc/Documentos/Facultad/PVLP - Paradigmas y Lenguajes de Programacion/UC-PVLP/Practica/TP3/codigo/punto_4_mpi con 10 proceso(s)
===== Inicio Ejecucion =====
Soy el ultimo proceso.
Este es el vector final: [0, 1, -2, 3, -4, 5, -6, 7, -8, 9]
===== Fin Ejecucion=====
Pulse INTRO o escriba una orden para continuar

```

Figure 7: Captura de pantalla de la ejecución del Código 10

5 Ejercicio 5

CONSIGNA: se tiene un algoritmo que envía un mensaje 100 haia los procesos superiores y al llegar al último nodo reenvía al proceso 0, tal como se muestra en la Figura 8 y se solicita que se adapte el código para que aparte del mensaje 100, se envíe un mensaje 200 que se dirija en dirección contraria, como se demuestra en la Figura 9.

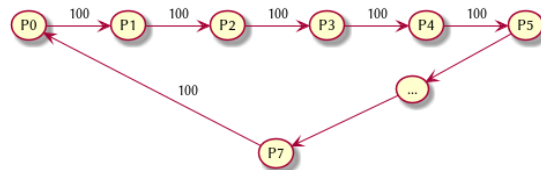


Figure 8: Algoritmo brindado por la cátedra para el Ejercicio 5

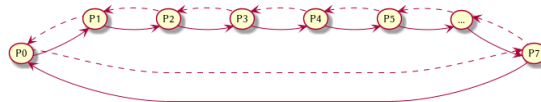


Figure 9: Transformación objetivo teniendo como base Figura 8

Teniendo en cuenta los siguientes valores para las líneas continuas y discontinuas respectivamente



Figure 10: Referencia de valores para Figura 9

En el Código 11 se brinda la implementación del algoritmo para la implementación de la Figura 8

Código 11: Implementación del algoritmo conceptualizado en Figura 8

```

1  /*
2    NUM_PROC=8
3  */
4  #include "mpi.h"
5  #include <stdio.h>
6  using namespace std;
7
8  int main (int argc, char *argv[]) {
9      int    numtasks,
10         rank,
11         next,
12         prev,
13         msj_forward=100,
14         bzn_forward,
15         tag=1;
16     MPI_Status estado;
17
18     MPI_Init(&argc, &argv);
19     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
20     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
21
22     /*Se definen los vecinos*/
23     next = rank+1;
24     if (rank == (numtasks-1))
25         next=0;
26
27     prev = rank-1;
28     if (rank == 0)
29         prev=numtasks-1;
30
31     /*Se envian los mensajes*/
32     MPI_Send(&msj_forward,
33             1,
34             MPI_INT,
35             next,
36             tag,
37             MPI_COMM_WORLD);
38
39     /*Se reciben los mensajes*/
40     MPI_Recv(&bzn_forward,
41             1,
42             MPI_INT,
43             prev,
44             tag,
45             MPI_COMM_WORLD,
46             &estado);
47

```

```

48     cout << "Soy "<<rank<<" y recibí"
49     <<" el msj "<<bzn_forward<<" de "
50     <<estado.MPI_SOURCE<<endl;
51
52     MPI_Finalize();
53     return 0;
54
55 }

```

la próxima acción será realizar la transformación de Código 11 para que cumpla con lo requerido en la Figura 9.

Código 12: Transformación del Código 11 a lo planteado por la Figura 9

```

1  /*
2   NUM_PROC=8
3  */
4  #include "mpi.h"
5  #include <stdio.h>
6  using namespace std;
7
8  int main (int argc, char *argv[]) {
9      int    numtasks, rank,
10         next, prev,
11         r_next, r_prev,
12         msj_forward=100, bzn_forward,
13         r_msj_forward=200, r_bzn_forward,
14         tag=1, r_tag=2;
15     MPI_Status estado, r_estado;
16
17     MPI_Init(&argc, &argv);
18     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
19     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21     /*Se definen los vecinos*/
22     next = rank+1;
23     r_prev = rank+1;
24     if (rank == (numtasks-1)){
25         next=0;
26         r_prev=0;
27     }
28     prev = rank-1;
29     r_next = rank-1;
30     if (rank == 0){
31         prev=numtasks-1;
32         r_next=numtasks-1;
33     }
34     /*Se envían/reciben los mensajes*/
35     /*      Ascendentes      */
36     MPI_Send(&msj_forward, 1, MPI_INT, next, tag,
37             MPI_COMM_WORLD);
38     MPI_Recv(&bzn_forward,1,MPI_INT,prev,tag,
39             MPI_COMM_WORLD,&estado);
40
41     /*Se envían/reciben los mensajes*/
42     /*      Descendentes      */
43     MPI_Send(&r_msj_forward,1,MPI_INT,r_next,r_tag,
44             MPI_COMM_WORLD);

```

```

45 MPI_Recv(&r_bzn_forward,1,MPI_INT,r_prev,r_tag,
46 MPI_COMM_WORLD,&r_estado);
47
48 cout << "Soy "<<rank<<" y recibí"
49 <<" el msj "<<bzn_forward<<" de "
50 <<estado.MPI_SOURCE<<" , además , "
51 <<"recibí el msj "<<r_bzn_forward
52 <<" de "<<r_estado.MPI_SOURCE<<endl;
53
54 MPI_Finalize();
55 return 0;
56
57 }

```

Se adjunta además una captura con la salida de la transformación realizada en el Código 12.

```

urc@urc: código$ v punto_5_t_mpi.cpp
Compilando el archivo hacia: /home/urc/Documentos/Facultad/PVLP - Paradigmas y Lenguajes de Programacion/UC-PVLP/Practica/TP3/codigo/punto_5_t_mpi
Corriendo /home/urc/Documentos/Facultad/PVLP - Paradigmas y Lenguajes de Programacion/UC-PVLP/Practica/TP3/codigo/punto_5_t_mpi con 8 proceso(s)
===== Inicio Ejecución =====
Soy 2 y recibí el msj 100 de 1, además, recibí el msj 200 de 3
Soy 3 y recibí el msj 100 de 2, además, recibí el msj 200 de 4
Soy 4 y recibí el msj 100 de 3, además, recibí el msj 200 de 5
Soy 5 y recibí el msj 100 de 4, además, recibí el msj 200 de 6
Soy 7 y recibí el msj 100 de 6, además, recibí el msj 200 de 0
Soy 0 y recibí el msj 100 de 7, además, recibí el msj 200 de 1
Soy 1 y recibí el msj 100 de 0, además, recibí el msj 200 de 2
Soy 6 y recibí el msj 100 de 5, además, recibí el msj 200 de 7
===== Fin Ejecución =====
Pulse INTRO o escriba una orden para continuar

```

Figure 11: Salida de la ejecución de Código 12

MPI_Reduce, MPI_Gather, MPI_Scatter, MPI_Broadcast

6 Ejercicio 6

CONSIGNA: resuelva el ejercicio planteado en la página de la [Universidad de Granada] con respecto al tutorial 4 - Producto Escalar:

- Mostrar las capturas de pantallas de la ejecución del ejercicio.
- Explique brevemente como funciona el código fuente.
- Describa las instrucciones propias de MPI utilizadas para el presente ejercicio. ¿Cuáles son los parámetros de entrada? ¿Cuáles son los parámetros de salida?

7 Anexo

7.1 Acerca de la compilación

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

Código 13: Script de compilación

```
1  #!/bin/sh
2
3  #Variables globales
4  # @param $1 archivo con el codigo fuente
5  # @param $2 nombre base del archivo, sin la extension
6  archivo=$1
7  base=$2
8
9  # Compila codigo OpenMPI en C++ y lo ejecuta con la cantidad
10 # de recursos que se le soliciten
11 compi() {\
12     txtcpp=$(sed 10q "$archivo")
13     if [[ $txtcpp =~ (NUM_PROC=)([0-9]) ]];
14     then
15         nproc=${BASH_REMATCH[2]};
16     else
17         nproc=1;
18     fi
19     echo && echo
20     echo "Compilando el archivo hacia: $base"
21     mpicxx "$archivo" -o "$base"
22     echo "Corriendo $base con $nproc proceso(s)"
23     echo "===== Inicio Ejecucion ====="
24     mpirun --oversubscribe -np "$nproc" "$base"
25     echo "===== Fin Ejecucion ====="
26 }
27
28 # Compila codigo OpenMP utilizando el compilador para C++,
29 # luego lo ejecuta
30 comp(){
31     g++ -fopenmp "$archivo" -o "$base" && "$base"
32 }
```

Se aprecian dos funciones una `compi()` 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 líneas una cadena que contenga lo siguiente: “NUM_PROC=*numero_de_procesos*” y lo recuerda. Luego de realizar la compilación del código, ejecuta al archivo resultante de la compilación mediante `mpirun --oversubscribe -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*.