CDAP – Práctica 1
Introducción al entorno MPI

Introducción a MPI:

MPI (Message Passing Interface) consiste en una serie de funciones que se incorporan a los lenguajes de programación C, C++ o Fortran que permiten programación paralela con el paradigma de paso de mensajes. En esta práctica se va a utilizar la implementación incluída en la distribución Linux ubuntu y los lenguajes de programación C y python. Para ello deben estar instalados los paquetes `openmpi-bin`, `openmpi-dev` (o `libopenmpi-dev`) y `python-mpi4py`.

Al utilizar un programa que usa la librería MPI, se deberá especificar el número de procesos en paralelo que se van a ejecutar. El módulo de ejecución examinará la lista de máquinas en las cuales se ejecutarán los procesos en paralelo. En el caso de un sistema Linux, esto es configurable mediante un fichero en el directorio **/etc**, y cuyos detalles serán dependientes de la implementación en concreto. La asignación de procesos se hará por orden: el número 0 a la primera de la lista, el 1 a la segunda y así sucesivamente. Si hay más procesos que máquinas, se continúa la asignación empezando otra vez por el principio de la lista.

Primer programa en MPI:

Copiar el programa hola.c:

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char* argv[])
{
   int my_rank;
   int p;
   int source; int dest;
   int tag=0;
   char message[100];

   MPI_Status status;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   if (my_rank != 0) {
      sprintf(message,"Saludos del proceso %d",my_rank);
      dest = 0;
      MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
   }
   else {
      printf("Hola, soy el proceso %d (hay %d procesos) y recibo:\n",my_rank,p);
      for(source=1;source<p;source++) {
         MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
         printf("%s\n",message);
```

```
      }
   }

   MPI_Finalize();

}
```

Compilarlo mediante:

```
$ mpicc -o hola hola.c
```

y ejecutarlo:

```
$ mpirun -np 4 hola
```

El número que sigue a `-np` especifica el número de procesos a ejecutar. Los detalles de lo que ocurre en este programa son los siguientes:

1. La llamada a **mpirun** tal como se especifica más arriba ejecuta una copia en cada uno de los procesadores especificados.

2. Los diferentes procesos ejecutan diferentes sentencias basándose en lo que se denomina el rango de los procesos. Esto se logra con las sentencias:

```
if (my_rank != 0) { } else { }
```

El rango del proceso se obtiene en la línea:

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Todas las funciones de MPI del programa deberán estar comprendidas entre las líneas

```
MPI_Init(&argc, &argv);
```

y

```
MPI_Finalize();
```

El paso de mensajes se logra mediante las funciones `MPI_Send` y `MPI_Recv`. Los primeros argumentos de estas funciones son inmediatos:

1. Un puntero a los datos a enviar.

2. La longitud de los datos a enviar (`MPI_Send`) y la longitud máxima de los datos a recibir (`MPI_Recv`).

3. El tipo de los datos: MPI proporciona una serie de tipos predefinidos que comienzan todos por la cadena `MPI_`.

4. El rango del proceso al que se le envían los datos (`MPI_Send`) y del que se esperan los datos (`MPI_Recv`). En el caso de `MPI_Recv`, el argumento

source puede ser la constante predefinida `MPI_ANY_SOURCE (en C)` o `MPI.ANY_SOURCE (en Python)`.

Los otros argumentos son un poco más complejos.

El argumento `tag` consiste en una etiqueta entera que se añade al mensaje que se pasa entre los procesos. Lo que hace es ponerle una marca al mensaje que puede ser usada a voluntad del programador (por ejemplo, para diferenciar mensajes de distintos tipos). Hay un comodín llamado `MPI_ANY_TAG (en C)` o `MPI.ANY_TAG (en Python)`.

El siguiente argumento es lo que se denomina el comunicador. Permite crear grupos de procesos que intercambian información. Por ahora vamos a usar el comunicador predefinido `MPI_COMM_WORLD`, que incluye a todos los procesos que están en ejecución.

El último argumento de `MPI_Recv`, source es un puntero a una estructura que contiene información sobre el mensaje. Por ejemplo, en caso de que se use `MPI_ANY_SOURCE` en una llamada a `MPI_Recv`, se puede identificar el proceso que ha enviado el mensaje examinando `status->MPI_SOURCE`.

En python, el programa equivalente es hola.py:

```
#!/usr/bin/env python

from mpi4py import MPI

comm = MPI.COMM_WORLD

my_rank = comm.rank
num_processes = comm.size

if my_rank != 0:
      message = "Saludos del proceso %d" % my_rank
      dest_rank = 0
      comm.send(message,dest=dest_rank)
else:
      print "Hola, soy el proceso %d (hay %d procesos) y recibo:" %
(my_rank,num_processes)
      for source_rank in range(1,num_processes):
            message = comm.recv(source=source_rank)
            print message
```

Para ejecutarlo:

```
$ mpiexec -np 4 ./hola.py
```

Ejercicios:

- Compilar y ejecutar el programa cambiando el número de procesos. Observar la salida.

- En el código C, modificar el programa de tal forma que use comodines en

`tag` y `source`.

- Modificar el programa de tal forma que el proceso que recibe los mensajes sea el de rango `p-1`.

- Modificar el programa de tal forma que el proceso de rango `i` mande el mensaje al proceso `(i+1)%p`. Primero hacer una versión que use `MPI_ANY_SOURCE` y después una que no los use. ¿Qué pasa si el programa se ejecuta solamente con un proceso?.