Tutorial complementario de MPI

Este tutorial complementa los apuntes sobre MPI con el objetivo de practicar las funciones básicas y ayudar en el manejo de MPI.

Envio y recepción de mensajes
En primer lugar estudia y copia el siguiente programa en un fichero `dados-1.c` que envía un valor aleatorio, de tipo entero desde un proceso al resto de procesos. Se puede ver el programa como que los procesos “tiran un dado” y el maestro recoge los resultados para procesarlos (mostrarlos por pantalla en este caso).

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TAG 999

void main(int argc, char **argv)
{
  int rank;
  int size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  srand(time(NULL) + rank);

  if (rank != 0) {
    int num = rand() % 6 + 1;
    MPI_Send(&num, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
  } else {
    for(int i = 1; i < size; i++) {
      int num;
      MPI_Recv(&num, 1, MPI_INT, i, TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Proceso %d ha sacado un %d.\n", i, num);
    }
  }

  MPI_Finalize();
}
```

Para compilar el programa utiliza `mpicc` que es un wrapper sobre `gcc` que se encarga de asegurar que las rutas a la librería MPI sean correctas. Admite los mismos parámetros que `gcc`.

```
$ mpicc dados-1.c -o dados-1
```

Para ejecutar el programa hay que utilizar mpirun, que se encarga lanzar tantas instancias del programa como nodos de ejecución se indiquen. Por ejemplo, para 4 nodos de ejecución:

```
$ mpirun -np 4 ./dados-1
```

Ejercicio 1: Envio de varios elementos (dados-2.c)
Extiende el programa anterior para que cada proceso envíe 10 números aleatorios. El proceso maestro se encargará de elegir el mejor de ellos.

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define TAG 999

int max(int *numeros, int total) {
   int max = INT_MIN;
   for(int i = 0; i < total; i++) {
      if (numeros[i] > max) {
         max = numeros[i];
      }
   }
   return max;
}

void main(int argc, char **argv)
{
    // TODO: Adaptar
    // El proceso maestro usará la función max para imprimir el mejor máximo
}
```

Ejercicio 2: Uso de MPI_Status (dados-3.c)
En el ejemplo inicial el bucle del maestro recibe los mensajes en orden.

```
    for(int i = 1; i < size; i++) {
      int num;
      MPI_Recv(&num, 1, MPI_INT, i, TAG,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Proceso %d ha sacado un %d.\n", i, num);
    }
```

En este ejercicio se propone utilizar MPI_ANY_SOURCE para recibir en cualquier orden. El problema es que entonces no se sabe qué proceso ha enviado el mensaje. Utiliza el parámetro MPI_Status para obtener el número de proceso que ha realizado cada envío.

Ejercicio 3: Deshacer empates (dados-4.c)
Extiende el ejemplo inicial para que en caso de empate los procesos que han empatado sigan “tirando el dado” hasta que haya un ganador. El objetivo es idear un protocolo similar al siguiente:

- El maestro notifica a los procesos si deben el dado (al principio todos).
- Los procesos lanzan el dado y pasan el resultado al maestro.
- El maestro comprueba los que han empatado y envía mensajes indicando cuáles deben lanzar el dado o no. Si no deben lanzar el dado, podrán terminar.
- Utiliza MPI_Barrier antes de finalizar para asegurar que todos los procesos llegan a MPI_Finalize al mismo tiempo y evitar problemas.