PRÁCTICA 2 – PROGRAMACIÓN PARALELA MEDIANTE PASO DE MENSAJES AMPLIACIÓN DE SISTEMAS OPERATIVOS

Descripción de la práctica

El objetivo de la práctica es el desarrollo de una aplicación paralela mediante el paso de mensajes sobre un cluster de ordenadores personales siguiendo una arquitectura maestro/esclavo. Para ello se utilizará la interfaz de paso de mensajes Open MPI y se emulará el cluster utilizando procesos que se comunicarán entre ellos. Es decir, la ejecución del código será idéntica a utilizar un cluster, pero se utilizará solo un computador que ejecutará todos los procesos.

Esta arquitectura responde a un modelo de programación centralizado en el que uno de los procesos lleva el peso del control, mientras que el resto se encargan de realizar el trabajo de cómputo. Por lo tanto, existen dos tipos de procesos. Los procesos slave son los que realmente se encargan de hacer el trabajo. Están replicados en todos los nodos que componen el cluster y en general su estructura es muy sencilla. Reciben los datos del proceso master, los procesan según el algoritmo que implementen y devuelven los resultados al proceso master. Después esperan recibir nuevas órdenes del master, que pueden ser más trabajo o una orden de FIN.

Por su parte el proceso master, si bien no realiza ningún trabajo de cálculo, tiene una estructura generalmente más compleja. Su misión es recibir la orden del usuario, repartir el trabajo entre los procesos slaves y recibir los resultados de todos los slaves, componiendo de forma adecuada el resultado final, que se presenta al usuario. También es el encargado de mantener la sincronización entre los slaves y en su caso la carga de trabajo equilibrada entre los distintos nodos de cómputo, en función de su capacidad en cada ejecución de la aplicación.

Open MPI

Como se ha dicho anteriormente, se utilizará el estándar Open MPI para la compilación y ejecución de la práctica. Dado que el alumno va a tener que ejecutar varios procesos en su ordenador, si se utiliza una máquina virtual (ej. VirtualBox) el alumno tendrá que aumentar el número de procesadores utilizados por dicha máquina.

Para ello, lo primero es apagar la máquina virtual si estaba corriendo. Luego, desde Configuración > Sistema > Procesador, se deberán asignar tantas CPUs (por defecto estará en 1) como procesos se quieran crear (8 máximo).

Ahora ya se puede arrancar la máquina. Con esta modificación podremos lanzar varios procesos de forma simultánea con Open MPI en VirtualBox.

Para instalar Open MPI en Ubuntu se pueden seguir dos opciones: una opción rápida mediante terminal, o una opción manual mediante la descarga desde la página web. Se recomienda seguir la instalación rápida, aunque se indican las dos a continuación por si se experimenta algún problema con ésta.

Instalación rápida de Open MPI en Ubuntu

- Para que Open MPI funcione correctamente es necesario tener instalado los compiladores gcc y g++, así como make
- Ejecutar el siguiente comando desde un terminal:

```
sudo apt-get install openmpi-bin openmpi-common openssh-client openssh-server libopenmpi*
```

- Si se realizó correctamente la instalación se podrá ejecutar el siguiente comando:

```
mpirun -version
```

Obteniendo:

```
mpirun (Open MPI) 4.0.3

Report bugs to http://www.open-mpi.org/community/help
```

Instalación manual de Open MPI en Ubuntu

- **En caso de no funcionar la instalación rápida**, habrá que recurrir a la instalación manual de Open MPI. Para ello será necesario ir a su web:

  https://www.open-mpi.org/

  Y realizar los siguientes pasos:

1. Descargar y descomprimir el fichero: `tar –xvf openmpi-*`
2. Ir dentro de la carpeta: `cd openmpi-*`
3. Configurar el fichero de instalación:
   `./configure –prefix=”/home/$USER/.openmpi”`
4. Instalar:
   `make`
   `sudo make install`

5. Incluir los paths lib y bin en “/home/<user>/.bashrc”

```
export PATH=”$PATH:/home/$USER/.openmpi/bin”
export LD_LIBRARY_PATH=”$LD_LIBRARY_PATH:/home/$USER/.openmpi/lib/”
```

Compilación y ejecución de programas con Open MPI

Una vez instalado Open MPI y configurada la máquina virtual, se expone a continuación unos casos de ejemplo para verificar la instalación. Es conveniente destacar que si el programa está escrito en lenguaje C, se compilará usando el comando “mpicc”, mientras que si el programa está escrito en lenguaje C++, se deberá compilar utilizando el comando “mpiCC” (con las C en mayúsculas). Es decir, se sustituye gcc/g++ por mpicc/mpiCC.

Así, el siguiente programa de ejemplo escrito en C:

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

int main(int argc, char **argv)
{
   int rank, nprocs;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hola mundo, soy el proceso %d de los %d que se están ejecutando\n", rank, nprocs);

   MPI_Finalize();
   return 0;
}
```

Se compilaría con el comando:

```
mpicc holamundo.c –o holamundo
```

Y se ejecutaría utilizando 4 procesos de la siguiente forma:

```
mpirun –np 4 holamundo
```

Se puede usar el depurador GDB en cada proceso mediante el siguiente comando:

```
mpirun –np 4 xterm –e gdb holamundo
```

A continuación se presenta un ejemplo de paso de mensajes, en este caso escrito en lenguaje C++:

```
#include "mpi.h"
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
   int rank, size, contador;
   MPI_Status estado;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   //Sólo el proceso 0
   if(rank == 0){
      MPI_Send(&rank //referencia al vector de elementos a enviar
            ,1 // tamaño del vector a enviar
            ,MPI_INT // Tipo de dato que envias
            ,rank+1 // pid del proceso destino
            ,0 //etiqueta
            ,MPI_COMM_WORLD); //Comunicador por el que se manda
   } else{
      MPI_Recv(&contador // Referencia al vector donde se almacenara lo recibido
            ,1 // tamaño del vector a recibir
            ,MPI_INT // Tipo de dato que recibe
            ,rank-1 // pid del proceso origen de la que se recibe
            ,0 // etiqueta
            ,MPI_COMM_WORLD // Comunicador por el que se recibe
            ,&estado); // estructura informativa del estado

      cout<<"Soy el proceso "<<rank<<" y he recibido "<<contador<<endl;
      contador++;
      //El último proceso no envía, sólo recibe
      if(rank != size-1)
         MPI_Send(&contador, 1 ,MPI_INT ,rank+1 , 0 ,MPI_COMM_WORLD);
   }
   MPI_Finalize();
   return 0;
}
```

Como se trata de código C++ se compilaría con el comando:

```
mpiCC ejemplo2.cpp –o ejemplo2
```

Y se ejecutaría utilizando 4 procesos de la siguiente forma:

```
mpirun –np 4 ejemplo2
```

Descripción concreta de la práctica

Se propone la creación de un programa utilizando Open MPI que gestione las entradas y salidas de vehículos a un parking. Los posibles vehículos (procesos) que pueden entrar a dicho parking son automóviles y camiones. Un automóvil ocupará una plaza de aparcamiento dentro del parking, mientras que un camión ocupará dos plazas contiguas de aparcamiento. Las plazas del parking se sitúan de manera contigua de la siguiente forma:

Planta 0:

0	1	2	3	...	N

Planta 1:

0	1	2	3	...	N

El parking dispondrá de una única entrada y una única salida, por las que entrarán y saldrán (respectivamente) los distintos tipos de vehículos permitidos. En la entrada de vehículos habrá un dispositivo de control que permitirá o impedirá el acceso de los mismos al parking dependiendo del estado actual del mismo (plazas de aparcamiento disponibles). En caso de que el dispositivo de control permita el acceso a un vehículo, se le asignará el número de plaza de aparcamiento donde debe aparcar. Queda a decisión del control de acceso qué plaza asigna de entre todas las que haya libres.

Los tiempos de espera de los vehículos dentro del parking son aleatorios (para ello, se puede utilizar la función rand). En el momento en el que un vehículo sale del parking, notifica al dispositivo de control el número de plaza que tenía asignada y se libera la plaza o plazas de aparcamiento que estuviera ocupando, quedando así éstas nuevamente disponibles.

Un vehículo que ha salido del parking esperará un tiempo aleatorio para volver a entrar nuevamente en el mismo. Por tanto los vehículos estarán entrando y saliendo indefinidamente del parking. Es importante que se diseñe el programa de tal forma que se asegure que, antes o después, un vehículo que permanece esperando a la entrada del parking entrará en el mismo.

El programa gestor del parking debe ser capaz de recibir como argumentos el número de plazas por cada planta (PLAZAS) y el número de plantas del parking (PLANTAS). Si no va a haber más de una planta se escribirá 1 en el argumento PLANTAS. El número de vehículos dependerá de la ejecución de mpirun a la hora de su ejecución con el comando –np.

Un ejemplo de funcionamiento de un parking con 6 plazas, una planta y 8 vehículos sería:

ENTRADA: Coche 1 aparca en 0. Plazas libres: 5
Parking: [1] [0] [0] [0] [0] [0]
ENTRADA: Coche 2 aparca en 1. Plazas libres: 4
Parking: [1] [2] [0] [0] [0] [0]
ENTRADA: Coche 3 aparca en 2. Plazas libres: 3
Parking: [1] [2] [3] [0] [0] [0]
ENTRADA: Coche 4 aparca en 3. Plazas libres: 2
Parking: [1] [2] [3] [4] [0] [0]
ENTRADA: Coche 5 aparca en 4. Plazas libres: 1
Parking: [1] [2] [3] [4] [5] [0]
SALIDA: Coche 2 saliendo. Plazas libres: 2
Parking: [1] [0] [3] [4] [5] [0]
SALIDA: Coche 3 saliendo. Plazas libres: 3
Parking: [1] [0] [0] [4] [5] [0]
ENTRADA: Coche 6 aparca en 2. Plazas libres: 2
Parking: [1] [0] [6] [4] [5] [0]
SALIDA: Coche 1 saliendo. Plazas libres: 3
Parking: [0] [0] [6] [4] [5] [0]
ENTRADA: Camión 101 aparca en 1. Plazas libres: 1
Parking: [101] [101] [6] [4] [5] [0]
SALIDA: Coche 4 saliendo. Plazas libres: 2
…

Puntuación de la práctica

- **(4 puntos)** Realizar un programa gestor de las entradas y salidas del parking que funcione únicamente para automóviles. Un automóvil podrá acceder al parking si hay, al menos, una plaza de aparcamiento libre. En el caso de que el parking esté lleno, el automóvil deberá esperar en la barrera a que se libere una plaza. Para dar por correcto este apartado, el código deberá estar libre de interbloqueos y no producir inanición.

- **(5 puntos)** Ampliar el programa anterior incluyendo también la gestión de camiones. Un camión sólo podrá acceder al parking si hay, al menos, dos plazas de aparcamiento contiguas libres. En el caso de que no haya, el vehículo deberá esperar en la barrera hasta que el control le permita entrar. Para dar por correcto este apartado, el código deberá estar libre de interbloqueos y no producir inanición.

- **(1 punto)** Ampliar el programa anterior permitiendo que funcione para varias plantas sabiendo que todas las plantas tendrán el mismo número de plazas y que un camión no puede aparcar entre dos plantas.