Programación paralela mediante paso de mensajes

Ampliación de Sistemas Operativos

Descripción general

El objetivo de la práctica es el desarrollo de una aplicación paralela mediante paso de mensajes sobre un cluster de ordenadores personales siguiendo una arquitectura *master/slave*.

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.

Configuración de la cuenta

La práctica se desarrollará en el *cluster* del laboratorio 103 del Edificio de Laboratorios II. Para poder ejecutar programas basados en MPI es necesario configurar la cuenta de usuario para que no solicite la passwd cada vez que se ejecuta un comando MPI. Para esto hay que seguir los siguientes pasos:

1. Verificar si en el directorio .ssh existen un par de claves pública-privada denominados id_dsa.pub y id dsa.

2. Si no existen, generalos con el comando:

```
ssh-keygen -t dsa
```

Si al ejecutar dicho comando pide alguna contraseña, dejarla en blanco.

3. Verificar que en cada máquina del *cluster* tienes un fichero llamado .ssh/authorized_keys y que tiene permisos `-rw--` En principio basta comprobarlo en un sólo nodo pues el directorio .ssh está en la partifición /home exportada por NFS. Si no existe se crea

```
touch .ssh/authorized_keys
```

```
chmod 600 .ssh/authorized_keys
```

4. Se añade la clave pública generada (id_dsa.pub) anteriormente a dicho fichero

```
cat .ssh/id_dsa.pub >> .ssh/authorized_keys
```

Compilación y Ejecución de Programas

Se utilizará el estándar OpenMPI para la compilación y ejecución de la práctica. Para su utilización se den seguir los siguientes pasos:

1. Compilar el/los programas con el comando `mpicc`. Es decir se sustituye los correspondientes comandos gcc/g++ por mpicc/mpiCC

```
mpicc fuente.c -o ejecutable
```

2. Escribir un fichero de configuración del *cluster* denominado `hostfile` en el que deben figurar los nombres de los nodos en los que se va a ejecutar la aplicación paralela. Por ejemplo para ejecutar en 4 nodos se debe escribir el siguiente fichero:

```
nodo1
nodo2
nodo3
nodo4
```

Nota: Es necesario comprobar previamente que los nodos están activos y funcionan correctamente. Para ello puede hacerse un `ping` a cada uno de ellos.

3. Una vez configurado el *cluster* hay que ejecutar el programa, para lo que se utiliza el comando `mpirun`. En este comando hay varias formas de indicar el número y distribución de los procesos entre los nodos:

a) Con la opción `--host` (lista de *host* separados por coma se especifica los hosts en los que ejecutaremos la aplicación

```
mpirun --host node0,node0,node0,node0 ...
```

De manera implícita se puede indicar en el fichero `hostfile` con la opción `--hostfile`. Hay que tener cuidado de no asignar un número de slots a un *host* mayor al de su número

b) Con la opción `-np` N procesos, MPI se encarga de ejecutar tantas copias del proceso como se indique y lo reparte entre los nodos disponibles en la configuración. Por ejemplo se puede ejecutar el siguiente comando con la salida asociada:

`mpirun --hostfile hostfile -n 4 app` (los cuatro procesos ejecutan app)

`mpirun --hostfile hostfile -np 2 app1 : -np 2 app2` (dos procesos ejecutan app1 y otros dos app2)

c) Se pueden mirar más opciones con `man mpirun`

Descripción concreta de la práctica

El programa a implementar debe gestionar las entradas y salidas de vehículos a un parking. Los posibles vehículos 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 permita o impida 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 principal 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). Es decir, el programa tendría que entender `parking PLAZAS PLANTAS`. Si no se realiza la creación de plantas, se deberá escribirse 1 como argumento de número de PLANTAS.

El número de automóviles y camiones dependerá de la ejecución del mandato *mpirun* a la hora de su ejecución. Si se realiza en varios ficheros una posible ejecución sería:

```
mpirun --hostfile hostfile -np 1 parking PLAZAS PLANTAS : -np 10 coche : -np 3 camion
```

La salida del programa debería ser algo del tipo:

```
ENTRADA: Coche 1 aparca en 0. Plazas libre: 5
Parking: [1] [0] [0] [0] [0] [0]
```

```
ENTRADA: Coche 2 aparca en 1. Plazas libre: 4
Parking: [1] [2] [0] [0] [0] [0]
ENTRADA: Coche 3 aparca en 2. Plazas libre: 3
Parking: [1] [2] [3] [0] [0] [0]
ENTRADA: Coche 4 aparca en 3. Plazas libre: 2
Parking: [1] [2] [3] [4] [0] [0]
ENTRADA: Coche 5 aparca en 4. Plazas libre: 1
Parking: [1] [2] [3] [4] [5] [0]
SALIDA: Coche 2 saliendo. Plazas libre: 2
Parking: [1] [0] [3] [4] [5] [0]
SALIDA: Coche 3 saliendo. Plazas libre: 3
Parking: [1] [0] [0] [4] [5] [0]
ENTRADA: Coche 7 aparca en 2. Plazas libre: 2
Parking: [1] [0] [7] [4] [5] [0]
SALIDA: Coche 1 saliendo. Plazas libre: 3
Parking: [0] [0] [7] [4] [5] [0]
ENTRADA: Camion 101 aparca en 1. Plazas libre: 1
Parking: [101] [101] [7] [4] [5] [0]
SALIDA: Coche 4 saliendo. Plazas libre: 2
….
```

Objetivos parciales

- Gestionar correctamente las entradas y salidas del parking únicamente para automóviles. Un automóvil podrá acceder al parking si hay, al menos, una plaza de aparcamiento libre. En el caso en el que el parking esté lleno, el automóvil deberá esperar en la barrera de entrada a que se libere una plaza de aparcamiento. Para dar por correcto el funcionamiento de este apartado, el programa tiene que estar libre de interbloqueos y no producir inanición (**3 puntos**).

- Gestionar correctamente las entradas y salidas del parking para todo tipo de vehículos permitidos (tanto automóviles como camiones). Un camión sólo podrá acceder al parking si hay, al menos, dos plazas contiguas de aparcamiento libre. En el caso de que no haya plazas disponibles, el vehículo deberá esperar en la barrera de entrada hasta que el control de acceso le permita entrar. Para dar por correcto el funcionamiento de este apartado, el programa tiene que estar libre de interbloqueos y no producir inanición (**5 puntos**).

- Implementar un algoritmo para gestionar un parking con varias plantas, todas ellas con el mismo número de plazas sabiendo que un camión no puede aparcar entre dos plantas (**1 punto**). Este apartado implicar reescribir el método de escritura del parking para mostrar el estado del parking para el número de plantas especificado.

- Gestionar la ejecución del parking separando en diferentes ejecutables la funcionalidad de cada uno de los elementos que forman el parking: gestor, coche y camión (**1 punto**). El número de automóviles y camiones dependerá entonces de la ejecución del mandato *mpirun* a la hora de su ejecución cómo se ha visto anteriormente.

Nota: las puntuaciones para cada objetivo parcial son las puntuaciones máximas que se pueden obtener si se cumplen esos objetivos.

Nota: Para evitar que se produzcan interbloqueos o inanición, no se considerarán válidas soluciones a medida implementadas por el alumno, que solucionen casos concretos.

Entrega de prácticas

La entrega se realizará a través del Campus Virtual en las fechas anunciadas en el mismo. Se debe entregar un único archivo llamado *prac2.zip* que incluya todos los ficheros .c debidamente comentados necesarios con el código de toda la práctica (en caso de utilizar tres ficheros estos se deben llamar *parking.c, coche.c* y *camion.c*) y una memoria de la misma en formato pdf. La memoria debe incluir:

- Índice de contenidos
- Autores
- Opciones de diseño utilizadas con su correspondiente decisión. Incluir diseños rechazados y explicación del rechazo
- Subsistema de comunicaciones de las distintos programas, justificando el empleo de las diferentes funciones de MPI utilizadas.
- Conclusiones extraídas en cuanto a la programación basada en *clusters*
- Comentarios personales: incluyendo problemas encontrados, críticas constructiva, propuesta de mejoras y evaluación del tiempo dedicado.
- No incluir código fuente, a menos que sea indispensable para explicar alguna mejora concreta
- NO DESCUIDE LA CALIDAD DE LA MEMORIA DE SU PRÁCTICA. Aprobar la memoria es tan imprescindible para aprobar la práctica, como el correcto funcionamiento de la misma. Si al evaluar la memoria se considera que no alcanza el mínimo admisible, la práctica se considerará SUSPENSA.

Evaluación de la práctica

La práctica se evaluará comprobando el correcto funcionamiento de los distintos objetivos, y valorando la simplicidad del código, los comentarios, la óptima gestión de recursos, la gestión de errores y la calidad de la memoria. El profesor podrá solicitar una defensa oral de la práctica para su correspondiente defensa de la misma.

Autoría de la práctica

La práctica se debe realizar en grupos de 3/4 personas como máximo.

El hecho de detectar copia en las prácticas expondrá a los alumnos a la posibilidad de una apertura de expediente disciplinario. En caso de detectar copia, los alumnos afectados serán suspendidos en la TOTALIDAD de la asignatura. Una práctica será considerada copia en caso de que contenga la totalidad o una parte de la práctica de otro alumno. Se considerará copia en caso de:

- Archivos que contengan la totalidad o fragmentos de código de otro alumno
- Memorias con la totalidad o fragmentos de frases e imágenes de otro alumno