UNIVERSIDAD CARLOS III DE MADRID
DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA.
ARQUITECTURA DE COMPUTADORES II
13 de junio de 2008

Para la realización del presente examen se dispondrá de **2 horas y media. NO** se podrán utilizar libros ni apuntes. Entregar cada ejercicio en hojas separadas.

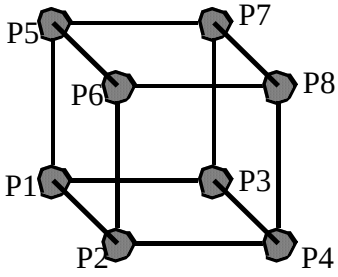
Ejercicio 1 (0.5 puntos)

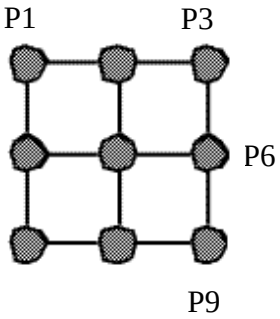
Uno de los objetivos básicos de cómputo paralelo es conseguir reducir el tiempo de ejecución de una aplicación. ¿Con qué otra finalidad resulta útil ejecutar una aplicación en paralelo?

SOLUCIÓN:
Otro tópico de interés consiste en poder ejecutar aplicaciones que por sus elevados requisitos de memoria no pueden ser ejecutadas en un único computador. Sin embargo, sí pueden serlo en una arquitectura paralela (por ejemplo, un multiprocesador de memoria distribuida) haciendo que cada nodo de cómputo sólo tenga una fracción de los datos.

Ejercicio 2 (0.5 puntos)

Considérense dos sistemas multiprocesadores, A y B, con 8 y 9 procesadores respectivamente. El sistema A tiene interconectados sus procesadores formando un cubo, mientras que el sistema B los tiene dispuestos en forma de malla.





Se pide:

a) ¿Cuál es el diámetro de la red de interconexión en ambos casos? ¿Y en el caso general para ambas configuraciones? Justifica la respuesta.
b) ¿Cuál es la bisección en el caso general para ambas configuraciones? Justifica la respuesta.

SOLUCIÓN

a) En el caso del cubo es 3 y en el de la malla es 4. En general, siendo n el número de procesadores, para un hipercubo el diámetro es log(n) y en una malla cuadrada 2(n^1/2 -1)
b) La bisección de un hipercubo es n/2 y la de una malla n^1/2

Ejercicio 3 (0.5 puntos)

Describe en qué consiste el *Test&Set* con *Backoff.*

SOLUCIÓN:

Mediante el *Test&Set* se está constantemente leyendo y escribiendo en una posición de memoria. Esta estrategia tiene como inconveniente generar un alto tráfico a través de la red/bus de comunicaciones.

Mediante el *Backoff* Cuando hay fallo, tarda "un poco" antes de reintentarlo. Este retardo puede ser constante o exponencial. De este modo se reduce el tráfico de red.

Ejercicio 4 (0.5 puntos)

Con qué propósito se utiliza la tecnología Grid en el *Large Hadron Collider* (LHC) del CERN.

SOLUCIÓN:

Mediante el empleo de la tecnología grid se tiene una plataforma que permite distribuir los datos producidos por el LHC entre varios centros distribuidos por el mundo para realizar su procesado de forma distribuida.

Ejercicio 5 (1.5 puntos)

Dado el siguiente programa que se ejecuta sobre una arquitectura NUMA compuesta por tres nodos de cómputo, cada uno de los cuales tiene dos procesadores y un banco de memoria local. Los procesadores no tienen memoria caché. Considérese que el único factor que determina la carga computacional es el número de multiplicaciones con un coste de 1ms. La variable `i` se almacena únicamente en registros del procesador.

```
for (i=0;i<120;i++)
{
    a[i] = a[i] * i

    if(i<60)
    {
      a[i] = a[i] * 2
    }

    b[i]=a[i+1]
 }
```

Se pide describir cómo realizarías las cuatro etapas del proceso de paralelización incluyendo la distribución de los datos (`a` y `b`) más adecuada. Realiza el diseño para que la ejecución del código sea la más eficiente posible, teniendo en cuenta utilizar todos los procesadores, balancear la carga y que **el resultado final del programa paralelo coincida con el del secuencial.**

SOLUCIÓN:

1. **Etapas:**
    1. **Descomposición: cada tarea se corresponde a una iteración del lazo. Existen dos tipos de tareas: aquellas para i<60 con una carga de 2 ms y aquellas con i>=60 con 1 ms.**

2. **Asignación: la asignación de las iteraciones coincide con la de los datos, es decir, el procesador que tiene asignada la iteración i, tendrá asociado los datos a[i] y b[i].**

   La carga total de la aplicación es 60*2 + 60 = 180 ms. El máximo número de hilos es de 6, por lo que cada hilo tendrá (en condiciones de buen balanceo de carga) 30 ms. Para conseguir un buen balanceo de carga existen múltiples soluciones, por ejemplo:
   - **Cada hilo con 10 iteraciones de i <60 y 10 iteraciones de i>=60.**
   - **4 hilos con 15 iteraciones de i<40 y 2 tienen 30 iteraciones i>=60.**

3. **Orquestación: es necesario asegurar que los accesos sobre b[i] se realicen sobre un correcto valor de a[i+1]. Notar que en el programa secuencial, el valor almacenado en b[i] se corresponde a el valor antiguo (antes de ser modificado) de a[i+1]. Por este motivo, un modo adecuado de asegurar el mismo resultado que el programa secuencial es mediante una copia privada del valor a[i+1] (antiguo) de cada procesador.**

4. **Mapeo: Cada proceso de asocia al nodo cuyo banco de memoria contiene los datos que tiene asociado. Dado que cada nodo tiene dos procesadores, ejecutará dos procesos.**

Ejercicio 6 (3 puntos)

El siguiente programa paralelo se ejecuta en 2 procesadores. Sus variables están almacenadas en la memoria compartida e inicializadas a cero.

Procesador 1	Procesador 2
`1a: X=1`	`2a:Y=2`
`1b: Y=X`	`2b:X=3`

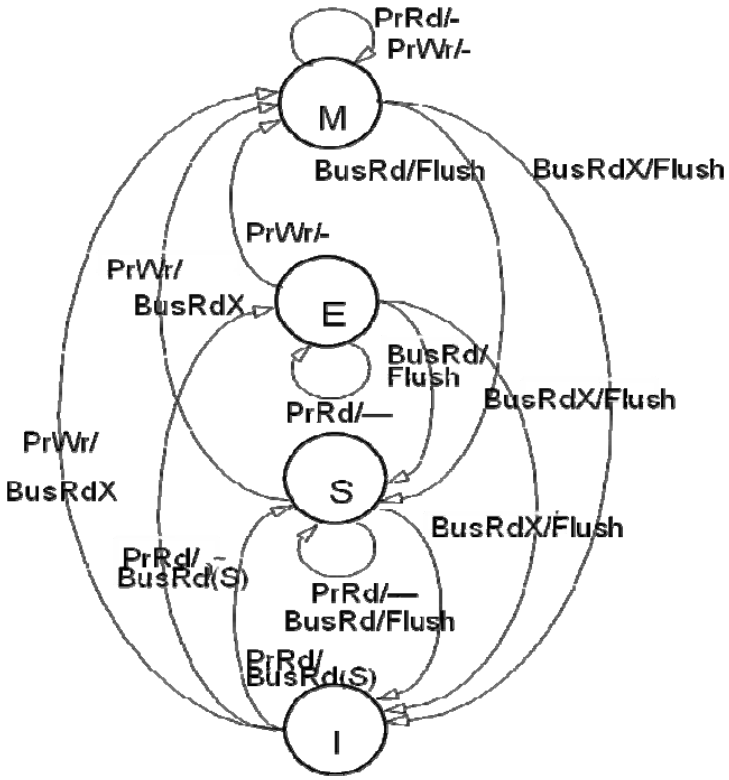
- **Parte A:** se pide responder, de forma razonada, a las siguientes preguntas:

  a) Qué valores finales de X e Y son posibles en un sistema con consistencia secuencial.

  b) ¿Hay algún valor final de X e Y que **no sea** aceptable bajo consistencia secuencial pero que sí lo sea bajo otros modelos de consistencia relajada? Indica el modelo de consistencia relajada en el que es aceptable.

- **Parte B:** Asumiendo que los procesadores acceden a los datos manteniendo una consistencia secuencial con un protocolo de coherencia MESI y que **inicialmente las memorias cachés están vacías**. Asumiendo también que primero se ejecuta P1 y cuando éste finaliza, se ejecuta P2. Se pide:
  a) Indicar qué transacciones del diagrama y que acciones del bus se realiza por el protocolo de coherencia cache cuando X e Y están **en bloques caché diferentes**.
  b) Indicar qué transacciones del diagrama y que acciones del bus se realiza por el protocolo de coherencia cache cuando X e Y están **el mismo bloque caché.**

Instruction	P1 transition	P1 action	P2 transition	P2 action	Bus actions



For lying in different cache blocks

SOLUCIÓN:

- **Consistencia secuencial: (3,2), (3,3),(3,1), (1,1)**
- **R: (1,2) no es acceptable bajo consistencia secuencial 2b, 1a, 2a, 1b. Pero sí es acceptable bajo weak consistency, processor consistency y release consistency.**
  La relación 1b -> 1a no es válida debido a que viola las dependencias de datos de un programa secuencial.
- **X e Y en distintas líneas caché**

Instruction	P1 transition	P1 action	P2 transition	P2 action	Bus actions
P1: X=1	I->M	PrWr			BusRdX
P1: Y=X	I->M	PrRd PrWr			BusRdX
P2: Y=2	M->I		I->M	PrWr	BusRdX Flush
P2: X=3	M->I		I->M	PrWr	BusRdX Flush

- **X e Y en la misma líneas caché**

Instruction	P1 transition	P1 action	P2 transition	P2 action	Bus actions
P1: X=1	I->M	PrWr			BusRdX
P1: Y=X		PrRd PrWr			
P2: Y=2	M->I		I->M	PrWr	BusRdX Flush
P2: X=3				PrWr	

Ejercicio 7 (1.5 puntos)

Dada una arquitectura de memoria compartida con *p* procesadores en la que se desea implementar las siguientes funciones de paso de mensajes.

```
send (int destproc, char *msg)
recv (int myrank, char *msg)
```

Ambas rutinas son bloqueantes: `send` se bloquea hasta que el parámetro `msg` es enviado y `recv` se bloquea hasta recibir el mensaje.

Se pide escribir el pseudo-código para las rutinas send y recv empleando las siguientes funciones POSIX:

```
lock(var) y unlock(var)
cond_wait(cond_var) y cond_signal(cond_var)
```

Pistas:

- Se deben emplear buffers intermedios para enviar/recibir los datos entre los procesadores.
- Es necesario utilizar una función init() para inicializar las variables necesarias.

SOLUCIÓN:

```
char netbuf[p];
int empty[p];
```

```
cond_var full_cond[p];
cond_var empty_cond[p];

init() {
  for (i=0;i<n;i++)
     empty[i]=TRUE;
}

send (int destproc, char *msg) {
   lock(empty[destproc]);
   while (!empty[destproc])
        cond_wait(empty_cond[destproc]);
   empty[destproc]=FALSE;
   netbuf[destproc] = msg;
   cond_signal(full_cond[destproc]);
   unlock(empty_cond[destproc]);
}

recv(int myrank, char* msg){
   lock(empty[myrank]);
   while (empty[myrank])
        cond_wait( full_cond[myrank]);
   empty[myrank]=TRUE;
   msg=netbuf[myrank] ;
   cond_signal(empty_cond[myrank]);
   unlock(empty[myrank]);
}
```

Ejercicio 8 (2 puntos)

Suponga que el comunicador `MPI_COMM_WORLD` está compuesto por tres procesos 0, 1, y 2 y sobre ellos se ejecuta el siguiente código en MPI:

```
int x, y, z;
Int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &mi_rank);
switch (mi_rank) {
      case 0:
            x = 0; y = 1; z = 2;
            MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
(*)         MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);
            MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD);
(…)         ……..
            break;
      case 1:
            x = 3; y = 4; z = 5;
            MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
            break;
      Case 2:
            x = 6; y = 7; z = 8;
            MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD);
            MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
            break;
}
printf (“ x = %d  y = %d  z = %d\n”,x,y,z);
```

1. ¿Cuáles son los valores de x, y, z para cada proceso después de cada llamada? Indícalo de forma justificada.
2. Se desea realizar una modificación del código anterior que consiste en trasladar la línea marcada por (*) y situarla sobre la línea punteada. ¿Es aún posible la ejecución del código? ¿Se obtendrían los mismos resultados? Justifique su respuesta.

Apéndice: Rutinas MPI.

```
MPI_Send(buffer,count,type,dest,tag,comm)

MPI_Recv(buffer,count,type,source,tag,comm,status)

MPI_Bcast (&buffer,count,datatype,root,comm)
```

SOLUCIÓN:

1.

```
case 0:
                x = 0; y = 1; z = 2;
                MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
                x = 0; y = 1; z = 2;
                MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);
                x = 0; y = 1; z = 2;
                MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD);
                x = 0; y = 1; z = 4;
                ……..
                break;
        case 1:
                x = 3; y = 4; z = 5;
                MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
                x = 0; y = 4; z = 5;
                MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
                x = 0; y = 4; z = 5;
                break;
        Case 2:
                x = 6; y = 7; z = 8;
                MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
                x = 6; y = 7; z = 0;
                MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD);
                x = 1; y = 7; z = 0;
                MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
                x = 1; y = 4; z = 0;
                break;
```

1. **No es posible la ejecución del problema por un estado de interbloqueo. En el proceso 2 se ejecuta la rutina MPI_Recv y nunca va a recibir ningún valor porque en el proceso 0 nunca se lleva a ejecutar la rutina MPI_Send. El segundo MPI_Bcast nunca llega a terminar debido a que es una llamada colectiva y por lo tanto, hasta que no se ejecute en todos los procesos no se procede a ejecutar la rutina MPI_Send.**