UNIVERSIDAD CARLOS III DE MADRID
DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA.
ARQUITECTURA DE COMPUTADORES II
19 de junio de 2007

Para la realización del presente examen se dispondrá de **2 1/2 horas. NO** se podrán utilizar libros ni apuntes.
Entregar cada ejercicio en hojas separadas.

Ejercicio 1 (1 punto)
¿Qué es el TOP500?. ¿Cómo ha sido la evolución de la posición de los multiprocesadores de memoria distribuida dentro del TOP500 en estos últimos 15 años?. ¿A qué se debe esta evolución?.

SOLUCIÓN:

El TOP 500 es un ranking de los 500 computadores más potentes (en términos de MFLOPS) del mundo. La potencia de los mismos se evalúa sobre un benchmark paralelo que simula una aplicación real.

Hace 15 años los multiprocesadores de memoria distribuida encabezaban las primeras posiciones del TOP 500. Sin embargo, tras la introducción de las arquitecturas en cluster han sido desplazados y en la actualidad apenas aparecen en el TOP500. El motivo es que los sistemas cluster ofrecen soluciones de más potencia computacional con menores costes.

Ejercicio 2 (1 punto)
Compara una arquitectura de memoria compartida basada en bus frente a una que emplea una red de hipercubo 2D. ¿Qué diferencias principales encuentras?. Cita las principales ventajas e inconvenientes de cada una de ellas.

SOLUCIÓN:

Las principales diferencias se encuentran en distintos niveles:

1. Respecto al mantenimiento de la coherencia cache. La arquitectura en bus es mucho más simple, dado que se puede aplicar un mecanismo de snoopy basado en el tráfico del bus (dado que una señal de bus es detectada por todos los procesadores). En el caso de una red de hipercubo, es necesario emplear un sistema de directorios para mantener la coherencia, con políticas más complejas para almacenar el estado de todas las líneas caché (directorios encadenados, por ejemplo).
2. Respecto a la escalabilidad, una arquitectura bus es menos escalable que una hipercubo. El motivo principal es que el bus representa un importante cuello de botella cuando se emplean un gran número de procesadores (>100), cosa que no sucede en una arquitectura hipercubo.
3. Respecto a la fiabilidad, el bus es menos fiable dado que supone un único punto de fallo. En el caso de una topología hipercubo, dado que tiene una mayor bisección, esta red aumenta significativamente su fiabilidad en el caso que alguna parte de la misma deje de funcionar.
4. Respecto al tipo de arquitectura. En el bus se puede implementar una arquitectura UMA mientras que en un hipercubo, se tiene una arquitectura NUMA.

Ejercicio 3 (2 puntos)
El objetivo de un algoritmo de ordenamiento es reordenar un conjunto de números en orden creciente (o decreciente) según su valor numérico.

Supongamos tener un conjunto de *n* procesos P_0, P_1, …, P_{n-1} organizados como un array lineal, donde P_0 recibe una secuencia de *n* números enteros diferentes entre sí, y P_i (i Є [1,n-1]) recibe datos de P_{i-1} únicamente.

Una solución paralela al problema de ordenamiento consiste en hacer que el proceso P_0, acepte una serie de *n* números, almacene el mayor número recibido hasta ese momento, y deje pasar aquellos números menores que el número almacenado. Cada proceso P_i realiza el mismo algoritmo, almacenando el número mayor recibido hasta ahora. Cuando todos los números hayan sido procesados, P_0 tendrá el mayor de los números, P_1 el siguiente número más grande, P_2 el próximo más grande y así sucesivamente. Por último, P_0 deberá escribir la secuencia de números en orden decreciente. La implementación paralela del algoritmo descrito es la versión paralela del algoritmo de ordenamiento por inserción. El algoritmo básico para el proceso P_i es:

```
der_procNum = n – i – 1;  /* numero de procesos a la derecha de Pi */

recv(&x, Pi-1);
for ( j = 0; j < der_procNum; j++ ) {
  recv(&numero, Pi – 1);
  if (numero > x) {
    send(&x, Pi+1);
    x = numero;
  } else
  send(&numero, Pi+1);
}
/* Enviar los resultados hacia P0 */
send(&x, Pi-1);
for ( j = 0; j < der_procNum; j++ ) {
  recv(&numero, Pi +1);
  send(&numero, Pi-1);
}
```

Nota: Los *send* y *recv* son asíncronos.

Asuma que la operación de comparación e intercambio se lleva a cabo en una etapa de tiempo y una comunicación lleva una unidad de tiempo. ¿Cuánto tiempo requiere la implementación paralela del algoritmo?.

Solución

¿Cuánto tiempo requiere la implementación paralela del algoritmo?.
Ciclos del *pipeline*: n + n – 1 = 2n -1. En cada ciclo tenemos una operación de comparación y otra de intercambio. Además, se necesitan *n* pasos de comunicación para enviar a P_0 los números ordenados (fase_final).
La comunicación consiste de un send() y un recv(), excepto para el último proceso en donde solo se produce un recv(); esta es una diferencia menor y la ignoramos. Por lo tanto, cada ciclo de pipeline requiere de al menos:

$$T_{cómputo} = 1$$
$$T_{comunicación\_pipeline} = 2\ (T_{inicio\_comunicación} + T_{datos})$$

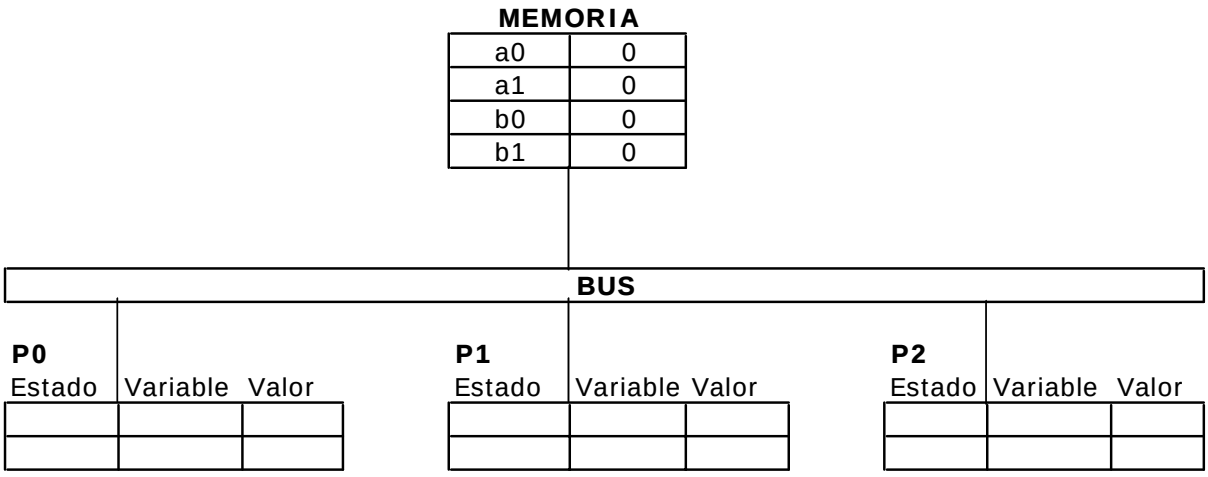
Además, se necesitan *n* pasos de comunicación para enviar a P_0 los números ordenados (fase_final), y en cada paso se tiene un send() y un recv().

$$T_{comunicación\_fase\_final} = 2\ (T_{inicio\_comunicación} + T_{datos})$$

El $T_{paralelo}$ es:

$$T_{paralelo} = (T_{cómputo} + T_{comunicación\_pipeline}\ )(2n\ -1) + T_{comunicación\_fase\_final}$$

Ejercicio 4 (2 puntos)
Un computador paralelo consta de una memoria compartida que tiene 4 bloques, cada bloque consta de una variable entera inicializada en cero. La memoria está conectada a 3 procesadores a través de un bus. Cada procesador tiene asociada una memoria caché de 2 líneas. Las cachés utilizan correspondencia directa. La siguiente figura muestra el estado inicial del sistema.

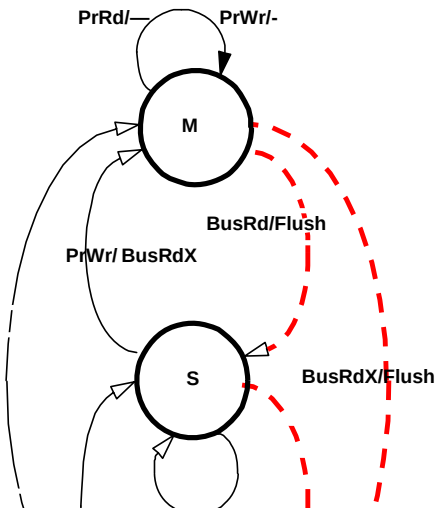


Utilice el protocolo MSI e indique el contenido de:

- La memoria.
- Las tres cachés (para cada línea de cada caché muestre el estado, el nombre de la variable que contiene y el valor de la misma)
- La acción asociada al evento (señal de bus) indicando qué procesador/procesadores la activan.

Después de ejecutar cada una de las siguientes secuencias de instrucciones.

Secuencia 1
t0: P0 lee a0; P1 lee a0; P2 lee b0



Secuencia 2
t1: P0 a0 = 5;

Secuencia 3
t2: P0 a0 = 10, P2 lee b0;

Secuencia 4
t3: P1 lee a0.

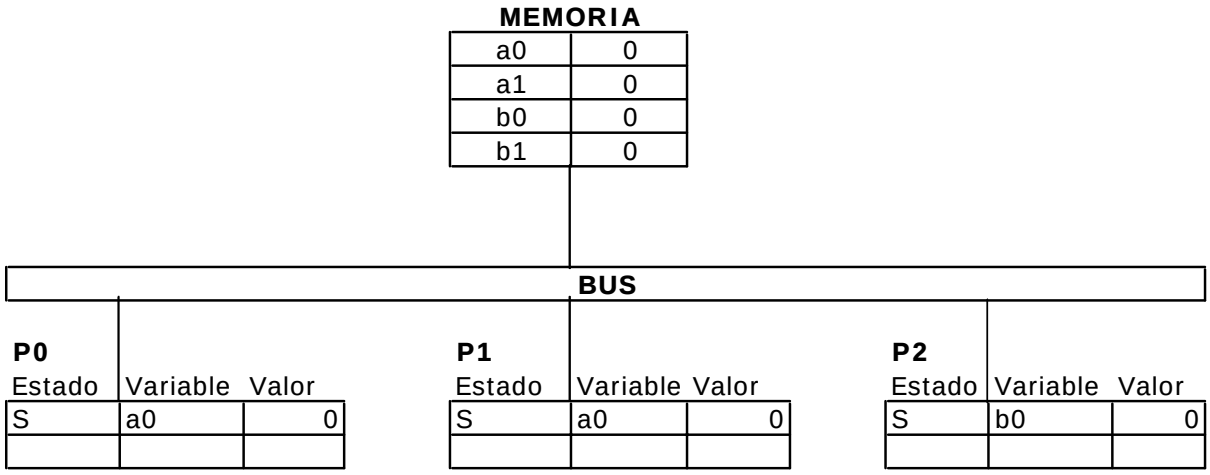
Secuencia 5
t4: P2 a1 = 20

Solución

Secuencia 1

t0: P0 lee a0; P1 lee a0; P2 lee b0 (en el tiempo 0 el proceso P0 lee a0, el proceso P1 lee a0 y P2 lee b0)

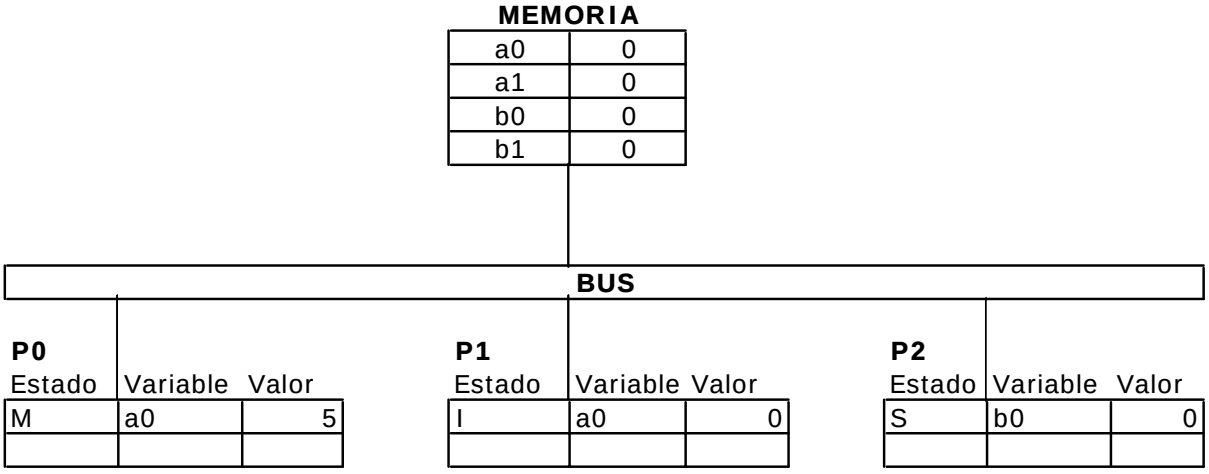
Señal de bus: BusRd (por procesador 0, 1 y 2).



Secuencia 2

t2: P0 a0 = 5.

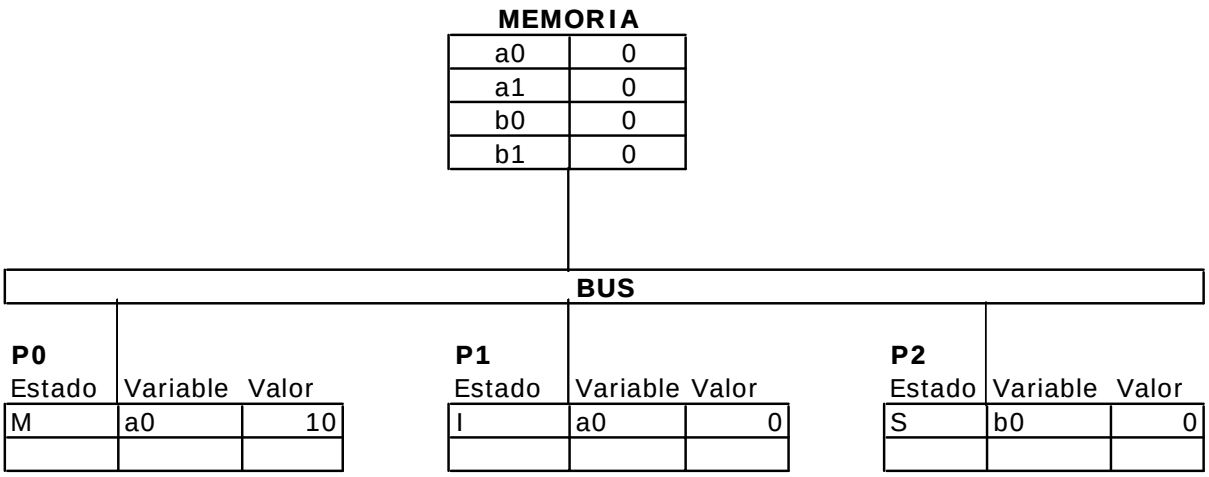
Señal de bus: BusRdX (por procesador P0).



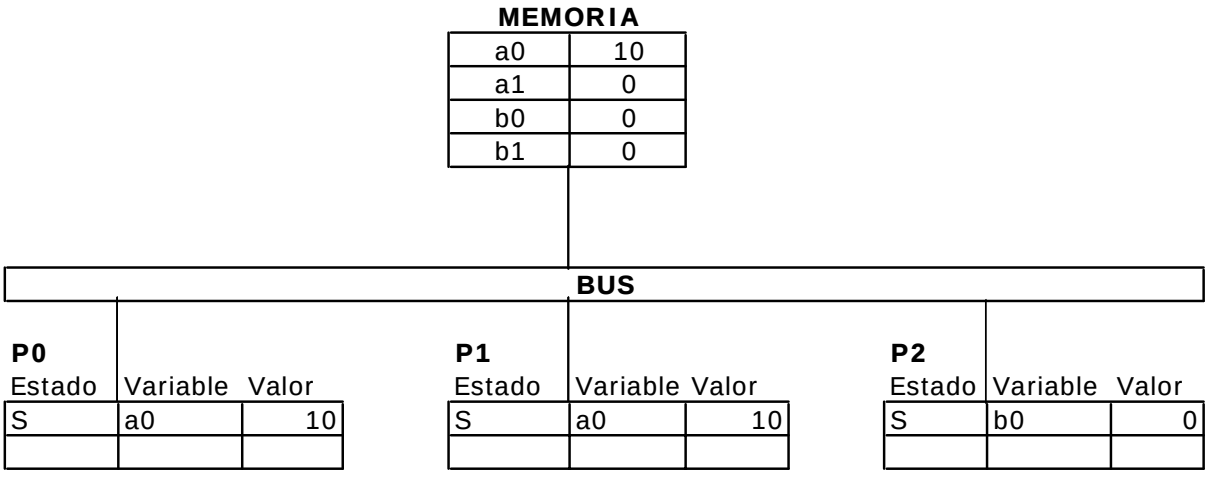
Secuencia 3

t2: P0 a0 = 10, P2 lee b0;

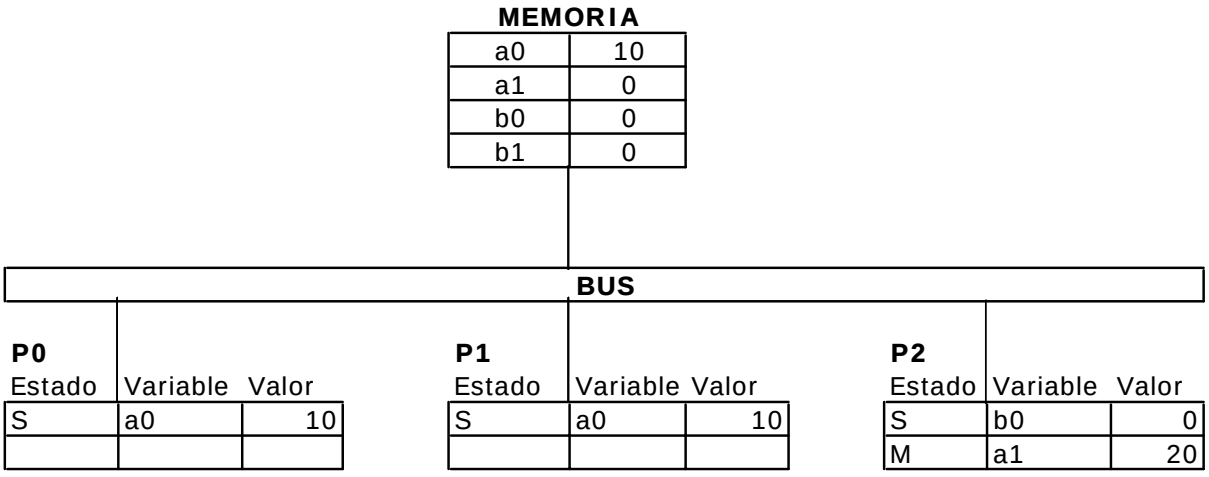
Señal de bus: Ninguna.



Secuencia 4
t3: P1 lee a0.
Señal de bus: BusRd (Procesador P1).



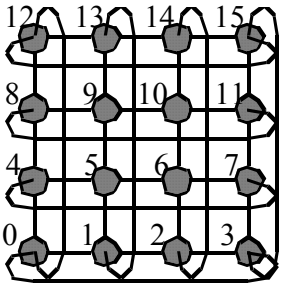
Secuencia 5
t4: P2 a1 = 20
Señal de bus: BusRdx (Procesador P2).



Ejercicio 5 (2 puntos)
La siguiente figura muestra una arquitectura paralela de memoria distribuida compuesta por 16 nodos conectados mediante una red 2D Torus tal y como se muestra en la figura. Cada nodo está compuesto por un procesador, su adaptador de red y un switch de comunicaciones.

La red tiene las siguientes características:

- Protocolo de encaminamiento es por conmutación de paquetes (*Store-and-forward*). Cada switch tiene capacidad de almacenar un único paquete y de enviar simultáneamente 4 (uno a cada switch vecino).
- *Overhead*: Retardo de envío y recepción del adaptador de red: 1 ms para cada caso.
- *Routing delay* (retardo de encaminamiento del *switch*): 2 ms.
- El ancho de banda es tan alto que se puede suponer despreciable el tiempo de transmisión.

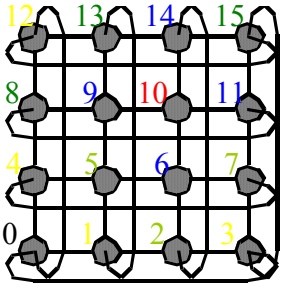


Se pide:

1. Calcular la latencia mínima y máxima de comunicación entre dos nodos de la red. Para cada caso, indicar en la figura entre qué nodos se alcanzan estas latencias.
2. Indicar el tiempo de realizar una operación de broadcast, asumiendo que la inicia el procesador de rank=0 y que éste envía a cada nodo un único paquete. Indicar en cada paso de tiempo qué nodo envía un paquete a qué otro nodo. Usar la notación: Num_nodo_emisor -> (Num_nodo_destino, Num_nodo_receptor)

SOLUCIÓN:

1. La latencia mínima se obtiene entre dos nodos consecutivos. En ese caso tiene de valor: 1ms (retardo envío) + 2ms (retardo encaminamiento) + 1ms (retardo recepción) = 4ms.
   La latencia máxima se obtiene entre aquellos nodos separados por el diámetro. Para este tipo de topología tiene de valor SQRT(16)=4. Asumiendo una nomenclatura (x,y), sería entre el procesador de coordenadas (0,0) y el (2,2). En este caso tiene de valor: 1ms (retardo envío) + 4*2ms (retardo encaminamiento a través de 4 switches) + 1ms (retardo recepción) = 10ms.
2. Para carcular el coste, es necesario obtener la distancia de cada nodo respecto al que inicia la operación de broadcast. La siguiente figura muestra estas distancias de forma coloreada: el nodo más alejado es el 10 (retardo 10ms), los siguientes son los nodos (6,9,11,14) de latencia 8, los siguientes son los nodos (2,5,7,8,13,15) de latencia 6 y los últimos los nodos (1,3,4,12) de latencia 4ms). El procedimiento sería:



T0: 0 -> (10,1) (14,12) (11,3) (6,4)
T1: 0 -> (9,4) (15,3) (13,12) (5,1)
1 -> (10,5)
12 -> (14,13)
3 -> (11,7)
4 -> (6,5)
T2: 0 -> (7,3) (2,1) (8,4)
5 -> (10,9)
13 -> (14,14) *
7 -> (11,11) *
5 -> (6,6) *
4 -> (9,8)
3 -> (15,15)*
12 -> (13,13)*
1 -> (5,5) *
T3: 0 -> (4,4) * (3,3)* (1,1)* (12,12)*
9 -> (10,10) *

8 -> (9,9) *
3 -> (7,7) *
1 -> (2,2) *
4 -> (8,8) *

Ejercicio 6 (2 puntos)
El siguiente algoritmo permite detectar la terminación de un programa paralelo.
Sean n procesos P_0, P_1,…, P_{n-1} conectados con una topología de anillo (i.e. P_i envía datos a $P_{(i+1) \bmod n}$). El programa paralelo tiene la siguiente estructura:

1. Cada procesador realiza en paralelo un conjunto de operaciones englobadas dentro de la función *acciones1()*.
2. Cuando P_0 termina de ejecutar la función anterior, genera un testigo (*token*) que es pasado a P_1. Tras enviar este testigo, pasa a realizar otro conjunto de operaciones dadas por la función *acciones2()*.
3. Cuando P_i ($1 <= i < n$) recibe el testigo
    a. Si ha finalizado la ejecución de *acciones1()*, pasa el testigo a $P_{(i+1) \bmod n}$.
    b. Si no, P_i espera a que finalice esta función y posteriormente pasa el testigo a $P_{(i+1) \bmod n}$.

    Tras enviar el testigo pasa a realizar otro conjunto de operaciones dadas por la función *acciones2()*.
4. Cuando P_0 recibe el testigo (de P_{n-1}), sabe que todos los procesos han terminado. Cuando la función *acciones2()* finaliza este procesador envía al resto de los procesos una señal de terminación (*sigend*) indicándoles que el programa ha terminado e inicia la ejecución de otro conjunto de operaciones dadas por la función *acciones3()*.

Programe en C, utilizando las primitivas MPI_Send() y MPI_Recv() de la librería de comunicaciones de MPI, el algoritmo antes descrito. Las variables *token* y *sigend* se implementan mediante una variable entera que toma de valor 1 para *token*, y 2 para *sigend*.

Synopsis:
```
#include "mpi.h
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
int MPI_Comm_size ( MPI_Comm comm, int *size )
int MPI_Send( void *buf, int count, MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm )
int MPI_Recv( void *buf, int count, MPI_Datatype datatype, int source,
        int tag, MPI_Comm comm, MPI_Status *status )
```

Solución

```
MPI_Init (&argc, &argv);            /* comienza MPI */
MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* obtiene process id */
MPI_Comm_size (MPI_COMM_WORLD, &size);

testigo    = 1; /* el proceso anterior ha terminado, esperamos que el resto termine */
proximo = (rank + 1) % size;
anterior  = (rank – 1) % size;

acciones1();

if ( rank == 0) {
    MPI_Send(&testigo, 1, MPI_INT, proximo, 0, MPI_COMM_WORLD);
    acciones2();
    /* recibe el testigo con valor == 1,
       significa que todos los procesos terminaron acciones1() */
    MPI_Recv(&testigo, 1, MPI_INT,anterior,0,MPI_COMM_WORLD, &status);
    acciones3();
    testigo = 2; /* para avisar a todos que hemos terminado */
    MPI_Send(&testigo, 1, MPI_INT, proximo, 0, MPI_COMM_WORLD);
} else {
    /* recibe el testigo con valor == 1 */
    MPI_Recv(&testigo, 1, MPI_INT,anterior,0,MPI_COMM_WORLD, &status);
    MPI_Send(&testigo, 1, MPI_INT,proximo, 0, MPI_COMM_WORLD);
    acciones2();
    /* recibe el testigo con valor == 2 */
    MPI_Recv(&testigo, 1, MPI_INT,anterior,0,MPI_COMM_WORLD, &status);
    acciones3();
}

MPI_Finalize();
```