UNIVERSIDAD CARLOS III DE MADRID
INGENIERÍA EN INFORMÁTICA.
ARQUITECTURA DE COMPUTADORES II
11 de Septiembre de 2010

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 (3 puntos)

Contesta de forma justificada a las siguientes cuestiones:

1. Usted es ingeniero informático de una compañía y se encarga de evaluar la compra de una arquitectura paralela. Dispone de dos alternativas.

   Opción A: Arquitectura NUMA con 128 nodos y con dos redes: una de gestión de memoria compartida (protocolo de coherencia caché) y otra para las sincronizaciones.

   Opción B: Cluster con 128 nodos. Con una red de comunicaciones.

   Ambas alternativas emplean el mismo procesador y tienen el mismo ancho de banda de red.

   Evalúe los pros y contras de cada una de ellas considerando los siguientes factores: coste, tiempo de desarrollo de aplicaciones paralelas y escalabilidad en número de nodos.

   SOLUCIÓN:

   - Coste: la opción A (NUMA) es a priori más cara dado que el soporte del protocolo de coherencia implica el desarrollo de un hardware específico para la plataforma (con un coste añadido).
   - Tiempo de desarrollo: la opción A tiene típicamente un entorno de desarrollo de OpenMP o Hilos mientras que la B es de MPI. La Opción A permite un menor tiempo de desarrollo.
   - Escalabilidad: la escalabilidad de la arquitectura NUMA (Opción A) será menor que la Opción B, dado que las plataformas de memoria compartida tienen un límite menor en el número de nodos (debido al mayor coste de gestión del protocolo de coherencia).

2. Por qué es preferible usar en MPI comunicaciones asíncronas en vez de las síncronas. Describe un escenario donde las comunicaciones asíncronas sean más eficientes.

3. Explica cuándo la implementación de cerrojos (*locks*) mediante T&T&S (*Test and Test and Set*) produce un pico en el tráfico de bus.

4. Define brevemente el paralelismo de **datos** y el **funcional**. ¿Qué se puede decir acerca de la escalabilidad y balanceo de carga de cada uno de ellos?

Ejercicio 2 (2 puntos)

El siguiente código de OpenMP se ejecuta en una arquitectura CC-NUMA con las siguientes características: tamaño de palabra de 4 bytes y tamaño de bloque (línea caché) de 16 bytes. El vector **a** está alineado en memoria, es decir, la posición de **a[0]** está en la primera entrada de un bloque. El mecanismo de coherencia caché es de actualización.

```
#pragma omp parallel shared(sum, a, n) private(i, valor)

#pragma omp for schedule(static)
for (i = 0; i < 1000; i++) {
   a[i]=2*i;
   valor = función_compleja(a[i]);

   #pragma omp critical
   sum = sum + valor;
}

#pragma omp barrier

 #pragma omp for schedule(static,10)
  for (i=0; i < 1000; i++){
    a[i] = sum;
  }
}
```

La función **función_compleja(i)** tiene un tiempo de cómputo (T) dependiente linealmente del valor del argumento. Es decir, mientras mayor es el valor de **i,** mayor es el tiempo de ejecución.

Se pide introducir en el código modificaciones (y justificarlas) con el fin de:

1. Mejorar el balanceo de carga.
2. Aumentar el grado de paralelismo.
3. Reducir el número de actualizaciones.

SOLUCIÓN

1.- Balanceo de la carga:

La distribución original (por bloques) resulta ineficiente dado que las últimas iteraciones tardan mucho más que las primeras. Una alternativa sería una distribución cíclica de los datos, sin embargo, esta originaría problemas de falsa compartición. Como una solución adecuada se puede emplear una distribución cíclica por bloques, donde cada bloque tiene tantas entradas como palabras hay en una línea caché. Es decir: 16/4=4 palabras por línea.

De este modo, el pragma del primer lazo sería:

```
#pragma omp for schedule(static,4)
```

2.- Aumentar el grado de paralelismo.

El código tiene dos fuentes de ineficiencia. Por una parte, el empleo de una sección crítica dentro del lazo limita el grado de paralelismo, dado que el accesos al a variable sum se realizan secuencialmente (mediante una sección crítica). Para solucionarlo se puede emplear una operación de reducción.

De este modo se eliminaría el pragma crítical y el pragma del primer lazo sería:

```
#pragma omp for schedule(static,4) reduction(+:sum)
```

La segunda fuente de ineficiencia es la operación de sincronización de barrera: esta sincronización es innecesaria dado que existe una sincronización implícita al finalizar el primer lazo y empezar el siguiente.

3.- Reducir el número de actualizaciones.

Existen dos factores en este apartado: el primero es la eliminación de la falsa compartición de los datos (false sharing) ya citado en el primer apartado.

El segundo factor consiste en que cada hilo opere sobre el mismo conjunto de datos en ambos lazos. Para lograr esto, es necesario que el scheduling del segundo lazo coincida con el del primero. Es decir:

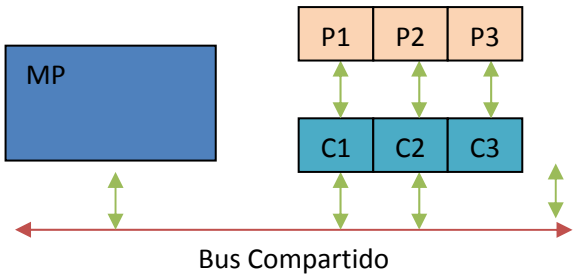
```
#pragma omp for schedule(static,4)
```

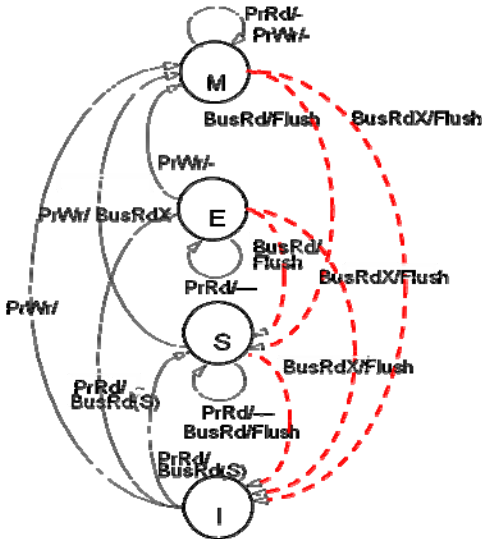
El código resultante es:

```
#pragma omp parallel shared(sum, a, n) private(i, valor)

#pragma omp for schedule(static,4) reduction(+:sum)
for (i = 0; i < 1000; i++) {
   a[i]=2*i;
   valor = función_compleja(a[i]);

   sum = sum + valor;
}


 #pragma omp for schedule(static,4)
  for (i=0; i < 1000; i++){
    a[i] = sum;
  }
}
```

Ejercicio 3 (2 puntos)

Dada la siguiente arquitectura multiprocesador donde existe un protocolo de coherencia caché **MESI**, en la que intervienen 3 procesadores actuando sobre una misma variable compartida **a**.





Se pide rellenar la siguiente tabla para las siguientes secuencias de instrucciones. **Cada una de estas tres secuencias son independientes y en cada una de ellas todas las cachés están inicialmente vacías.**

Procesador: Acción	**Transacción de bus**	**Quién proporciona el bloque (MP,C1,C2,C3)**	**Estado Caché1**	**Estado Caché2**	**Estado Caché3**
P1-read(**a**)					
P1-write(**a**)					
P2-read(**a**)					
P3-write(**a**)					
P1-read(**a**)					

Procesador: Acción	**Transacción de bus**	**Quién proporciona el bloque (MP,C1,C2,C3)**	**Estado Caché1**	**Estado Caché2**	**Estado Caché3**
P1-read(b)					
P3-read(b)					
P3-write(b)					
P1-read(b)					
P2-read(b)					

Procesador: Acción	Transacción de bus	Quién proporciona el bloque (MP,C1,C2,C3)	Estado Caché1	Estado Caché2	Estado Caché3
P2-read(c)					
P2-write(c)					
P2-write(c)					
P3-read(c)					
P1-write(c)					

SOLUCIÓN

1.- P1-read(a); P1-write(a); P2-read(a); P3-write(a); P1-read(a)

Procesador: Accion	Bus Transaction	Quien Proporciona el bloque (MP,C1,C2,C3)	Estado Caché1	Estado Caché2	Estado Caché3
P1:read(a)	BusRd(S)	MP	E	I	I
P1:write(a)	-		M	I	I
P2:read(a)	BusRd(S)	C1	S	S	I
P3:write(a)	BusRdX	MP* (ó una caché con copia)	I	I	M
P1:read(a)	BusRd(S)	C3	S	I	S

2.- P1-read(b); P3-read(b); P3-write(b); P1-read(b); P2-read(b)

Procesador: Accion	Bus Transaction	Quien Proporciona el bloque (MP,C1,C2,C3)	Estado Caché1	Estado Caché2	Estado Caché3
P1:read(b)	BusRd(S)	MP	E	I	I
P3:read(b)	BusRd(S)	C1	S	I	S
P3:write(b)	BusRdX ó BusUpgrade	MP* (ó cache con copia)	I	I	M
P1:read(b)	BusRd(S)	C3	S	I	S
P2:read(b)	BusRd(S)	MP* (ó caché con copia)	S	S	S

BusUpgrade implica acceso exclusivo a bloque, sin lectura

3.-P2-read(c); P2-write(c); P2-write(c); P3-read(c); P1-write(c)

Procesador: Accion	Bus Transaction	Quien Proporciona el bloque (MP,C1,C2,C3)	Estado Caché1	Estado Caché2	Estado Caché3
P2:read(c)	BusRd(S)	MP	I	E	I
P2:write(c)	-	-	I	M	I
P2:write(c)	-	-	I	M	I
P3:read(c)	BusRd(S)	C2	I	S	S
P1:write(c)	BusRdX	MP* (ó caché con copia)	M	I	I

Ejercicio 4 (1 punto)

En una arquitectura que implementa Consistencia Secuencial se pide indicar qué valores finales de A, B y C son posibles tras la ejecución de los códigos. Incialmente A, B y C están inicializadas a 0.

Procesador 1	Procesador 2	Procesador 3
`A=4`	`C = 2`	`if (A >0)`
`if (B==0)`	`if (B ==3)`	`B=3`
`C = 1`	`C = A`	

A	B	C

SOLUCIÓN

A	B	C
4	3	4
4	3	2
4	3	1
4	0	1
4	0	2

Ejercicio 5 (2 puntos)

El siguiente programa en MPI se ejecuta sobre una máquina de cuatro procesadores. Se pide indicar en las tablas los valores de **x** e **y** en cada proceso en cinco puntos distintos del programa.

```
void main(argc,argv)

int argc;

char *argv[];

{
        int p, i, x, y[4];
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &x);
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        for (i=0; i<p; i++) {

             y[i] = 0;

        }

/********** Punto 1 **********/

        MPI_Gather(&x, 1, MPI_INT, y, 1, MPI_INT, 1, MPI_COMM_WORLD);

/********** Punto 2 **********/

        MPI_Reduce(&x, &y[2], 1, MPI_INT, MPI_SUM, 3, MPI_COMM_WORLD );

/********** Punto 3 **********/

        MPI_Scatter(&y, 1, MPI_INT, &y[1], 1, MPI_INT, 1, MPI_COMM_WORLD);

/********** Punto 4 **********/

        MPI_Bcast(&y[2],1,MPI_INT, 3, MPI_COMM_WORLD);

/********** Punto 5 **********/

        MPI_Finalize();

}
```

Nota: Las funciones MPI tienen los siguientes argumentos.

```
int MPI_Scatter ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int
recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm ) ;

int MPI_Gather ( void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int
recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm ) ;


int MPI_Reduce ( void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op
op, int root, MPI_Comm comm )

int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm
)
```

Punto 1

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0					
Proceso 1					
Proceso 2					
Proceso 3					

Punto 2

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0					
Proceso 1					
Proceso 2					
Proceso 3					

Punto 3

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0					
Proceso 1					
Proceso 2					
Proceso 3					

Punto 4

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0					
Proceso 1					
Proceso 2					
Proceso 3					

Punto 5

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0					
Proceso 1					
Proceso 2					

Proceso 3					

SOLUCIÓN

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0	0	0	0	0	0
Proceso 1	1	0	0	0	0
Proceso 2	2	0	0	0	0
Proceso 3	3	0	0	0	0

POINT 2:

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0	0	0	0	0	0
Proceso 1	1	0	1	2	3
Proceso 2	2	0	0	0	0
Proceso 3	3	0	0	0	0

POINT 3:

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0	0	0	0	0	0
Proceso 1	1	0	1	2	3
Proceso 2	2	0	0	0	0
Proceso 3	3	0	0	6	0

POINT 4:

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0	0	0	0	0	0
Proceso 1	1	0	1	2	3
Proceso 2	2	0	2	0	0
Proceso 3	3	0	3	6	0

POINT 5:

	x	y[0]	y[1]	y[2]	y[3]
Proceso 0	0	0	0	6	0
Proceso 1	1	0	1	6	3
Proceso 2	2	0	2	6	0
Proceso 3	3	0	3	6	0