Universidad Carlos III de Madrid	**Departamento de Informática** **Ingeniería Informática** **Arquitectura de computadores II** **Examen final** **8 de septiembre de 2011**	ARCOS

Nombre:-……

- Dispone de dos horas y quince minutos para realizar la prueba.
- No se podrán utilizar libros ni apuntes, ni calculadoras de ningún tipo.
- Los teléfonos móviles deberán permanecer desconectados durante la prueba

Pregunta 1 (1 punto).

- Define el modelo de programación paralela de paralelismo sobre datos (*data parallel*).
- De las siguientes tres arquitecturas: UMA, NUMA, memoria distribuida. ¿Sobre cuáles ellas se puede aplicar este modelo?

SOLUCIÓN

Paralelismo sobre datos:

Concurrencia que se obtiene cuando se aplica la misma operación sobre todos los elementos de un conjunto. Se corresponde con SIMD donde cada procesador ejecuta el mismo código sobre diferentes datos. Dos alternativas de programación: concurrencia explícita e implícita. El programador aporta: Algoritmo. Directivas para distribuir datos. Directivas que guíen la paralelización.

Este modelo es aplicable a las tres arquitecturas mencionadas.

Pregunta 2 (1.25 puntos).

¿En qué se diferencian la consistencia secuencial y la relajada? Cita algún modelo de consistencia relajada.

SOLUCIÓN

Interpretación de Culler et al.: condiciones suficientes para consistencia secuencial:

Cada proceso “lanza” (issues) las operaciones de memoria en el orden del programa.

Después de lanzar una operación de escritura, el proceso espera que se complete la escritura antes de lanzar la siguiente operación.

Después de lanzar una operación de lectura, el proceso espera a que se complete la lectura y a que se realice la escritura cuyo valor va a ser devuelto por la lectura, antes de arrancar la siguiente operación: escritura atómica.

Modelos de Consistencia Relajada (Relaxed Consistency, RC) son menos exigentes que la Consistencia Secuencial, pero más eficiente: permite optimizaciones jugando con el orden de los accesos a memoria. Todos los modelos de RC suministran algún tipo de mecanismo que permite forzar explícitamente la coherencia secuencial.

Modelos:

Processor Consistency (PC), Weak Consistency (WC) yRelease Consistency (RC).

Pregunta 3 (1 punto).

Dado un programa secuencia que se quieres paralelizar con MPI, ¿Qué aspectos hay que considerar en la fase de particionamiento y orquestación para conseguir la máxima aceleración (*speedup*) posible? Enuméralos y defínelos brevemente.

SOLUCIÓN

- Aspectos:
  - Balanceo de carga.
  - Sincronización mínima.
  - Comunicación mínima.
  - Trabajo extra mínimo.

Pregunta 4 (1.5 puntos).

El siguiente código de lock/unlock está implementado con instrucciones estándares (es decir, instrucciones convencionales que no tienen ninguna propiedad especial). Se pide:

1. ¿Qué inconveniente tiene esta implementación que hace que el lock/unlock no funcionen de forma correcta?
2. ¿Qué modificaciones se pueden introducir para que el lock/unlock sí funcionen adecuadamente?

```
lock:     ld   .R1, /dir_cerrojo
          cmp  .R1, #0
          bnz  $lock
          st   #1, /dir_cerrojo
          ret

unlock:   st   #0, /dir_cerrojo
          ret
```

SOLUCIÓN

El problema es que las instrucciones de carga no son atómicas por lo que dos hilos pueden ejecutar a la vez el lock y entrar simultáneamente en la sección crítica.

La solución es el empleo de instrucciones atómicas como T&S o LL y SC.

Pregunta 5 (1.25 puntos).

Enumera los factores que forman parte en la expresión de la latencia de comunicaciones en una red de computadores.

SOLUCIÓN

Evaluación de la latencia de comunicación:
latencia (n)o d = overhead envío + overead recepción
+ ocupación del canal
+ retardo de encaminamiento
+ retardo por "contención"
Overhead: tiempo necesario para iniciar la operación de envío/recepción de un mensaje.

Ocupación del canal = (n+ne)/b
n: data payload
ne: packet envelope
b: ancho de banda
Retardo de encaminamiento: asociado al retardo del switch.
Contención: conflicto en el acceso a un mismo recurso (switch o nodo).

Pregunta 6 (2 puntos).

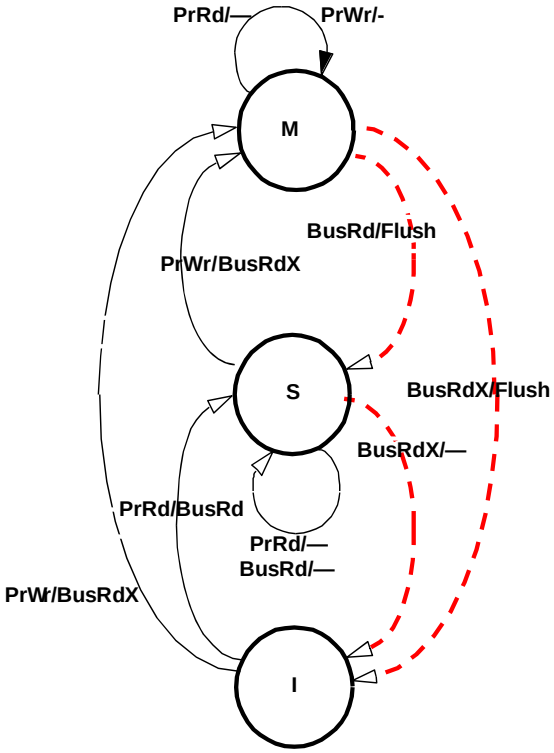
Dado el siguiente código ejecutado sobre una arquitectura de memoria compartida sobre la que ejecuta el siguiente programa empleando 2 hilos.

```
#pragma omp parallel private(myrank i), shared(a)
{
        myrank=omp_get_thread_num()

        if(myrank==0){
                    a=10;
        }
        else{
             sleep(1000);
             for(i=0;i<10;i++){
                a=10;
             }
        }
}
```

La arquitectura tiene las siguientes características:

- 2 procesadores conectados a través de un bus y un único banco de memoria. Cada procesador tiene una memoria caché local.
- Inicialmente todos los procesadores tienen la caché vacía.
- El tamaño de una palabra es de 8 bytes y el tamaño de bloque caché es de 32 bytes.
- El coste asociado a un fallo caché de lectura es de 8 bytes para notificar el fallo y 32 bytes para traer el dato a la memoria caché.
- Existen tres posibles implementaciones de esta arquitectura:
    1. Mediante un protocolo de invalidación MSI. El tráfico asociado a la operación de invalidación es de 2 bytes.
    2. Mediante un protocolo de actualización. El coste de una actualización es de 32 bytes.



Se pide calcular de forma justificada el tráfico de bus en cada uno de las dos posibles implementaciones anteriores cuando el código de ejecuta empleando 2 hilos en total. El índice i del bucle así como la variable myrank se almacenan en un registro y no en la caché. **Nótese que el hilo con myrank=0 se ejecuta y finaliza antes que el resto de los hilos.**

SOLUCIÓN

Invalidación:

- **Hilo rank=0.**
  - **La primera escritura en a produce un fallo caché. La escritura produce una invalidación. Tráfico: MISS(8 + 32) + INV(2)**
- **Hilo rank=1.**
  - **La primera escritura en a produce un fallo caché. La escritura produce una invalidación. Tráfico: MISS(8 + 32) + INV(2)**
  - **El resto del as escrituras son aciertos y no producen invalidaciones. Tráfico: MISS(8 + 32) + INV(2)**
- **Tráfico total: MISS(8 + 32) + INV(2) + MISS(8 + 32) + INV(2)**

Actualización:

- **Hilo rank=0.**
  - **La primera escritura en a produce un fallo caché y una actualización. Tráfico: MISS(8 + 32) + ACT(32)**
- **Hilo rank=1.**
  - **La primera escritura en a produce un fallo caché (la caché estaba vacía) y una actualización Tráfico: MISS(8 + 32) + ACT(32)**
  - **El resto del as escrituras no producen fallos y sí actualizaciones: Tráfico: ACT(32)*9**

Tráfico total: MISS(8 + 32) + ACT(32) + MISS(8 + 32) + ACT(32) + ACT(32)*9

Pregunta 7 (2 puntos).

El siguiente programa MPI se ejecuta sobre una máquina de cuatro procesadores. Se pide rellenar las tablas en cada uno de los puntos.

```
int main(int argc, char** argv)
{
	int pid, npr;
	int i;
	int A[6], B[2];
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &pid);
	MPI_Comm_size(MPI_COMM_WORLD, &npr);

	/* Inicializaciones en todos los procesos */
	for(i=0; i<6; i++) A[i] = -1;
	for(i=0; i<2; i++) B[i] = -1;

	if (pid == 0)
	{
		for(i=0; i<6; i++) A[i] = i;
	}

	MPI_Scatter(&A[0], 2, MPI_INT, &B[0], 2, MPI_INT, 0, MPI_COMM_WORLD);

	/* Punto 1 */

	MPI_Reduce(&B[0], &A[0], 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

	/* Punto 2 */

	MPI_Bcast(&B[0], 2, MPI_INT, 0, MPI_COMM_WORLD);

	/* Punto 3 */

	MPI_Gather(&B[0], 2, MPI_INT, &A[0], 2, MPI_INT, 0, MPI_COMM_WORLD);

	/* Punto 4 */

	MPI_Allgather(&B[0], 2, MPI_INT, &A[0], 2, MPI_INT, MPI_COMM_WORLD);

	/* Punto 5 */

	MPI_Allreduce(&B[1], &A[0], 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

	/* Punto 6 */

	MPI_Finalize();
	return 0;
} /* main */
```

Punto 1

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0								

Proceso 1								
Proceso 2								

Punto 2

	A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**B[0]**	**B[1]**
Proceso 0								
Proceso 1								
Proceso 2								

Punto 3

	A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**B[0]**	**B[1]**
Proceso 0								
Proceso 1								
Proceso 2								

Punto 4

	A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**B[0]**	**B[1]**
Proceso 0								
Proceso 1								
Proceso 2								

Punto 5

	A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**B[0]**	**B[1]**
Proceso 0								
Proceso 1								
Proceso 2								

Punto 6

	A[0]	**A[1]**	**A[2]**	**A[3]**	**A[4]**	**A[5]**	**B[0]**	**B[1]**
Proceso 0								
Proceso 1								

Proceso 2								

SOLUCIÓN

Punto 1

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0	0	1	2	3	4	5	0	1
Proceso 1	-1	-1	-1	-1	-1	-1	2	3
Proceso 2	-1	-1	-1	-1	-1	-1	4	5

Punto 2

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0	6	1	2	3	4	5	0	1
Proceso 1	-1	-1	-1	-1	-1	-1	2	3
Proceso 2	-1	-1	-1	-1	-1	-1	4	5

Punto 3

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0	6	1	2	3	4	5	0	1
Proceso 1	-1	-1	-1	-1	-1	-1	0	1
Proceso 2	-1	-1	-1	-1	-1	-1	0	1

Punto 4

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0	0	1	0	1	0	1	0	1
Proceso 1	-1	-1	-1	-1	-1	-1	0	1
Proceso 2	-1	-1	-1	-1	-1	-1	0	1

Punto 5

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0	0	1	0	1	0	1	0	1
Proceso 1	0	1	0	1	0	1	0	1
Proceso 2	0	1	0	1	0	1	0	1

Punto 6

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	B[0]	B[1]
Proceso 0	3	1	0	1	0	1	0	1
Proceso 1	3	1	0	1	0	1	0	1
Proceso 2	3	1	0	1	0	1	0	1