**UNIVERSIDAD CARLOS III DE MADRID**
**DEPARTAMENTO DE INFORMÁTICA**
**INGENIERÍA EN INFORMÁTICA.**
**ARQUITECTURA DE COMPUTADORES II**
**10 de septiembre de 2009**

Para la realización del presente examen se dispondrá de **2 1/2 horas. NO** se podrán utilizar libros ni apuntes.

---

## Ejercicio 1 (4 puntos).

1. Define la diferencia entre la ejecución concurrente y paralela de procesos. Cita un ejemplo para cada caso. **(0.5 puntos)**

2. ¿Por qué protocolo *snoopy* de coherencia caché está especialmente orientado a arquitecturas basadas en bus? **(0.5 puntos)**

3. ¿Se puede ejecutar en paralelo una aplicación en MPI sobre un multiprocesador de memoria compartida? ¿y una aplicación de OpenMP en una arquitectura de memoria distribuida (no compartida)? **(0.75 puntos).**

4. Desde el punto de vista de eficiencia de un código MPI (en términos de tiempo de ejecución), ¿qué resulta más conveniente, emplear primitivas de comunicación síncronas o asíncronas? **(0.75 puntos).**

5. ¿Qué diferencia existe entre un protocolo de encaminamiento (*routing*) determinista y uno adaptativo? **(0.75 puntos).**

6. Define qué es una arquitectura de cómputo tipo ***cluster***. **(0.75 puntos).**

## Ejercicio 2 (3 puntos)

Dada una plataforma de memoria compartida con dos procesadores conectados a través de un bus y un único banco de memoria. Los procesadores tienen memoria caché. Existen dos posibles implementaciones de esta arquitectura:

1. Mediante un protocolo de invalidación con *write once*. El tráfico asociado a la operación de invalidación es de 16 bits.
2. Mediante un protocolo de actualización. El coste de una actualización es de 64 bits.

El coste asociado a un fallo caché de lectura es de 32 bits para notificar el fallo y 64 bits para traer el dato a la memoria caché.

Los dos procesadores ejecutan un programa paralelo en *Threads* POSIX que tiene una operación de *lock* implementada mediante **Test&Set**. En el caso de acceder a la sección crítica se ejecuta un *sleep* de 10 segundos (sin impacto en la caché) y se realiza un *unlock*, tras lo cual el programa finaliza.

Los dos procesadores alcanzan esta operación en el mismo instante de tiempo. Es decir, ambos ejecutan el LOCK a la vez. Inicialmente, la variable de *lock* (dirección `0xFFFF0001`) está únicamente en memoria y tiene un valor 0.

Se pide:

1. Describir la secuencia temporal en la ejecución del programa.

2. Si los procesadores operan a 200MIPS (millones de instrucciones por segundo) y todas las instrucciones tienen el mismo tiempo de ejecución, se pide calcular para cada implementación el tráfico de bus (en bits transferidos) asociado a la ejecución del *lock* y *unlock* por los dos procesadores.

3. **Sólo para el caso de actualizaciones**: si el bus tiene un ancho de banda de 3Gbits/seg. ¿Se logra saturar el bus mientras el primer procesador está ejecutando el *sleep*?

4. ¿Cómo afecta al rendimiento del programa que el bus llegue a saturarse?

```
IMPLEMENTACIÓN DE LOCK
LABEL:
lock:       t&s    .R1, 0x FFFF0001
            bnez   .R1, $lock
            sleep(10)
            ret

IMPLEMENTACIÓN DE UNLOCK
unlock:     st    0 0x FFFF0001
            ret
```

**SOLUCIÓN:**

**La secuencia temporal es la siguiente:**

1. **Los procesadores alcanzan el T&S y lo ejecutan en orden, por ser atómica. El procesador A obtiene el lock.**
2. **El procesador A ejecuta el sleep mientras que B ejecuta T&S**
3. **El procesador A finaliza y ejecuta el Unlock.**
4. **El procesador B obtiene el lock.**
5. **El procesador B ejecuta el sleep.**
6. **El procesador B finaliza y ejecuta el Unlock.**

**En el caso de invalidaciones tenemos:**

1. **El primer T&S (procesador A): realiza el test (fallo caché (32+64) bits) y realiza el set (1 invalidación sobre memoria) de 16 bits.**
2. **El procesador B: realiza el test y tiene un fallo (32 + 64 bits) el set (1 invalidación sobre la caché de A) de 16 bits.**
3. **El A entra en el sleep durante 10 segundos (no hay tráfico).**
4. **El B realiza T&S durante 10 segundos. Debido a que emplea write once, no realiza invalidaciones obteniendo aciertos caché sin tráfico de bus.**

5. **El A ejecuta Unlock: tiene un fallo (32 + 64 bits) y emite una invalidación (sobre la caché de B) de 16 bits.**
6. **El procesador B: realiza T&S tiene un fallo (32 + 64 bits) en la lectura y luego una invalidación de 16 bits en la escritura (sobre la caché de A).**
7. **B entra en el sleep.**
8. **El B ejecuta Unlock: tiene un acierto caché (no realiza invalidación).**

**TOTAL: 4 fallos, 4 invalidaciones**

**En el caso de actualizaciones tenemos:**

1. **El primer T&S (procesador A): realiza el test (fallo caché (32+64) bits) y realiza el set (1 actualización) de 64 bits.**
2. **El procesador B: realiza el test y tiene un fallo (32 + 64 bits) el set (1 actualización) de 64 bits.**
3. **El A entra en el sleep durante 10 segundos (no hay tráfico)**
4. **El B realiza T&S durante 10 segundos. Tiene aciertos de lectura pero emite una actualización. Debido a que el procesador es de 200MIPS y que ejecuta dos instrucciones (t&s y bnez) el número de t&s ejecutados será 100*(10^6)*10. Cada una realizará una actualización, por lo que el tráfico será de 64*100*(10^6)*10**
5. **El A ejecuta Unlock: emite una actualización de 64 bits.**
6. **El procesador B: realiza T&S y emite una actualización de 64 bits.**
7. **B entra en el sleep.**
8. **El B ejecuta Unlock: tiene un acierto caché y emite una actualización de 64 bits.**

**TOTAL: 2 fallos, 5 + 100*(10^6)*10 actualizaciones**

**Mientras que A ejecuta el sleep, el ancho de banda demandado por B es de 100*(10^6)*64 bits / segundo = 6400Mbits/seg. >> 3000Mbits/seg del bus, por lo que el bus se satura.**

**Una saturación del bus conduce a una mayor latencia en la realización de las actualizaciones. Lo cual hace que el programa de B se ejecute más lentamente.**

**Ejercicio 3 (3 puntos)**

Dado el siguiente programa MPI que se ejecuta sobre una arquitectura de memoria distribuida compuesta por dos nodos de cómputo. Considérese dos factores en la carga computacional:

1.- Las operaciones aritméticas en punto flotante, con un coste de 1ns.

2- Las comunicaciones, son un coste del 2ns por cada palabra transmitida. Las comunicaciones son síncronas (tanto el **`send`** como el **`revc`** se bloquean en la comunicación).

Se pide:

1. Calcular el tiempo de ejecución de cada proceso del programa paralelo.

2. Calcular la aceleración (asumiendo que el programa secuencial ejecuta todo el código sin las operaciones de comunicación).

3. Comente posibles mejoras que contribuyan a:
   a) El aumento del grado de paralelismo y reducción del número de comunicaciones.
   b) La mejora del balanceo de carga.

4.- Escriba el código paralelo resultante (optimizado) y calcule la aceleración.

```
float a[500],b[100]

MPI_Comm_rank(MPI_COMM_WORLD, &mi_rank);
switch (mi_rank) {

      case 0:

            for (i=0;i<500;i++){
                a[i] = i*340;
            }
            for (i=0;i<100;i++){
                b[i] = i*400;
            }

            MPI_Send(&b, 100, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

            for (i=0;i<500;i++){
                a[i] = a[i] * a[i];
            }

            MPI_Recv(&b, 100, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

            Sum=0;
            for (i=0;i<500;i++){
                Sum = Sum + a[i];
            }
            for (i=0;i<100;i++){
                Sum = Sum + b[i];
            }
            Printf("El resultado acumulado es: %d",Sum);

      case 1:

            MPI_Recv(&b, 100, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);

            for (i=0;i<100;i++){
                b[i] = b[i] / 10;
            }

            MPI_Send(&b, 100, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
```

**SOLUCIÓN:**

**1.-**

**Tiempo de ejecución de Rank 0:**
**500 + 100 + 200 (com) + 500 + 200 (comm) + 500 + 100 = 2100ns.**

**Tiempo de ejecución de Rank 1:**
**600 (espera) + 200 (com) + 100 + 200 (comm) = 1100 ns**

**2.-**

**Tiempo secuencial:**
**500 + 100 + 500 + 100 + 500 + 100 = 1800ns.**

**Speedup = Ts/Tp = (500 + 100 + 500 + 100 + 500 + 100) / (500 + 100 + 200 (com) + 500 + 200 (comm) + 500 + 100) = 1800 / 2100.**

**3.a El grado de paralelismo se puede mejorar inicializando las matrices y sumado sus valores en paralelo. Las comunicaciones se pueden reducir: si las matrices se inicializan en paralelo, no es necesario enviarlas. Enviando la suma, sólo es necesario enviar un dato.**

**3.b El grado de balanceo de puede mejorar distribuyendo asignando también 200 entradas de A al Rank 1.**

```
MPI_Comm_rank(MPI_COMM_WORLD, &mi_rank);
switch (mi_rank) {
      case 0:

            for (i=0;i<300;i++){
                a[i] = i*340;
            }

            for (i=0;i<300;i++){
                a[i] = a[i] * a[i];
            }

            Sum=0;
            for (i=0;i<300;i++){
                Sum = sum + a[i];
            }

            MPI_Recv(&Sum2, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);

            Sum = sum + Sum2;

            Printf("El resultado acumulado es: %d",Sum);

      case 1:

            for (i=300;i<200;i++){
                a[i] = i*340;
            }
            for (i=0;i<100;i++){
                b[i] = i*400;
            }

            for (i=0;i<100;i++){
                b[i] = b[i] / 10;
            }
            for (i=300;i<200;i++){
                a[i] = a[i] * a[i];
            }
            Sum=0;
            for (i=0;i<100;i++){
                Sum = sum + b[i];
            }
            for (i=300;i<200;i++){
                Sum = sum + a[i];
            }

            MPI_Send(&sum, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}
```

**Tiempo de ejecución de Rank 0:**
**300 + 300 + 300 + 2 (comm)**

**Tiempo de ejecución de Rank 1:**

**200 + 100 + 100 + 200 + 100 + 200 + 2(comm.)**

**2.-**

**Tiempo secuencial:**
**500 + 100 + 500 + 100 + 500 + 100**

**Speedup = Ts/Tp = (500 + 100 + 500 + 100 + 500 + 100) / (300 + 300 + 300 + 2 ) = 1800 / 902.**