UNIVERSIDAD CARLOS III DE MADRID
DEPARTAMENTO DE INFORMÁTICA
INGENIERÍA EN INFORMÁTICA.
ARQUITECTURA DE COMPUTADORES II
17 de septiembre de 2008

Para la realización del presente examen se dispondrá de **2.5 horas. NO** se podrán utilizar libros ni apuntes. Entregar cada ejercicio en hojas separadas.

Ejercicio 1 (1 punto)

Enumera las principales diferencias existentes entre una arquitectura UMA, NUMA y CC-NUMA.

SOLUCIÓN:
En una arquitectura UMA el coste de acceso a memoria (expresado en latencia) es el mismo para cualquier posición de memoria (y cualquier procesador).

En una arquitectura NUMA, el coste no es homogéneo, siendo menor en aquellos bancos de memoria próximos al procesador, la latencia de acceso a memoria será menor que aquellos otros más alejados al procesador.

En una arquitectura CC-NUMA no existe memoria principal habiendo únicamente memoria caché (distribuida).

Ejercicio 2 (1 punto)

Dado el siguiente código, describe a grandes rasgos cómo se realizaría su ejecución paralela empleando:
1. Modelo de programación *data-parallel*.
2. Modelo de programación paralela funcional

```
for(i=0;i<100;i++)
{
  a[i]=10;
 }

for(j=0;j<500;j++)
{
  b[j]=sin(10*j);
}
```

SOLUCIÓN:
1. **Mediante un modelo data-parallel, primero se ejecuta el primer bucle en paralelo, repartiendo las iteraciones entre los procesadores. Posteriormente se ejecuta el segundo lazo en paralelo del mismo modo, repartiendo las iteraciones entre los procesadores.**
2. **Mediante un modelo de paralelismo funcional, se ejecutan los dos lazos en paralelo, asignando cada uno de ellos a un proceso.**

Ejercicio 3 (2 puntos)

Dada una plataforma con las siguientes características: dos procesadores conectados a través de un bus y un único banco de memoria. Cada procesador tiene una memoria caché local. Existen dos posibles implementaciones de esta arquitectura:

1. Mediante un protocolo de invalidación. 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.

Sobre los cuatro procesadores del sistema se ejecuta un programa paralelo en Threads POSIX que tiene una operación de lock implementada mediante Test&Test&Set. Después de la cual se ejecuta un código nulo (sin impacto en la caché) y se realiza un unlock.

Los dos procesadores alcanzan esta operación en el mismo instante de tiempo. Es decir, ambos ejecutan el while al mismo tiempo. Inicialmente, la variable de lock (dirección 0x00AF341F) está únicamente en memoria y tiene un valor 0.

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.

```
            IMPLEMENTACIÓN DE LOCK
LABEL:      while (load(0x00AF341F) = 1) do
                        nothing;
            if (TEST&SET(0x00AF341F) = 0)
                        {
                          Código nulo;
                        }
            else goto LABEL;


            IMPLEMENTACIÓN DE UNLOCK
            unlock:     st   0 0x00AF341F
            ret
```

SOLUCIÓN:

La secuencia temporal es la siguiente:

1. **Los dos procesadores (A y B) alcanzan el lock y realizan la operación de LOAD.**
2. **Los procesadores alcanzan el T&S y lo ejecutan en orden, por ser atómica. El procesador A obtiene el lock.**
3. **El procesador A ejecuta el código nulo mientras que B ejecuta el load.**
4. **El procesador A finaliza y ejecuta el Unlock.**
5. **El procesador B ejecuta el T&S y obtiene el lock.**
6. **El procesador B ejecuta el código nulo.**
7. **El procesador B finaliza y ejecuta el Unlock.**

En el caso de invalidaciones tenemos:

1. **2 loads -> 2 fallos caché: 2*(32+64) bits.**

2. **El primer T&S (procesador A): realiza el test (acierto caché) y realiza el set (1 invalidación) de 16 bits. El procesador B: realiza el test y tiene un fallo (32 + 64 bits) el set (1 invalidación) de 16 bits.**
3. **El B realiza varios loads (acierto caché).**
4. **El A ejecuta Unlock: tiene una invalidación de 16 bits.**
5. **El procesador B: realiza el load y tiene un fallo (32 + 64 bits). Luego ejecuta el test (acierto caché) el set (invalidación) de 16 bits.**
6. **Nada**
7. **El B ejecuta Unlock: tiene un acierto caché (no realiza invalidación).**

TOTAL: 4 fallos, 4 invalidaciones

En el caso de actualizaciones tenemos:

1. **2 loads -> 2 fallos caché: 2*(32+64) bits.**
2. **El primer T&S (procesador A): realiza el test (acierto caché) y realiza el set (1 actualización) de 64 bits. El procesador B: realiza el test y tiene un acierto el set (1 actualización) de 64 bits.**
3. **El B realiza varios loads (acierto caché).**
4. **El A ejecuta Unlock: tiene un acierto y una actualización de 64 bits.**
5. **El procesador B: realiza el load (acierto) y el test (acierto) y el set (1 actualización) de 64 bits.**
6. **Nada**
7. **El B ejecuta Unlock: tiene un acierto caché y una (1 actualización) de 64 bits.**
8. **TOTAL: 2 fallos, 5 actualizaciones**

Ejercicio 4 (2 puntos)

Dado el siguiente programa secuencial:

```
int main (int argc, char *argv[]) {
int i, n;
float a[100], b[100], sum;

/* Inicializaciones */
n = 100;
for (i=0; i < n; i++)
a[i] = b[i] = i * 1.0;
sum = 0.0;

for (i=0; i < n; i++)
sum = sum + (a[i] * b[i]);

printf("Sum = %f\n",sum);
}
```

Se pide proporcionar dos versiones optimizadas, haciendo los cambios que sean necesarios al código. Una basada en hilos (POSIX threads) y otra usando una única cláusula OpenMP (no importa la planificación).

Solución:

Hilos:

```
#define SIZE 100
```

```
//#define SIZE 100000000
#define NUM_HILOS 2

int n, nh;
float a[SIZE], b[SIZE], sum;
pthread_t *th;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void mul(int *num) {
  int i, ini=*num;
  float aux=0;
  for (i=ini*n/nh; i < (ini+1)*n/nh; i++)
    aux = aux + (a[i] * b[i]);

  pthread_mutex_lock(&m);
  sum=sum+aux;
  pthread_mutex_unlock(&m);
}

int main (int argc, char *argv[]) {
  int i;

  /* Inicializaciones */
  n = SIZE;
  for (i=0; i < n; i++)
    a[i] = b[i] = i * 1.0;
  sum = 0.0;

  nh = NUM_HILOS;
  th = malloc(nh*sizeof(pthread_t));

  for (i=0;i<nh;i++) {
    th[i]=(pthread_t)malloc(sizeof(pthread_t));
    pthread_create(&th[i], NULL, (void * (*)(void *))mul,
(void *)(&i));
  }

  for (i=0;i<nh;i++)
    pthread_join(th[i], NULL);

  printf("Sum = %f\n",sum);
}
```

OpenMP:

```
int main (int argc, char *argv[]) {
  int i, n;
  float a[100], b[100], sum;

  /* Inicializaciones */
  n = 100;
  for (i=0; i < n; i++)
    a[i] = b[i] = i * 1.0;
  sum = 0.0;

  #pragma omp parallel for reduction(+:sum)
    for (i=0; i < n; i++)
      sum = sum + (a[i] * b[i]);
```

```
    printf("Sum = %f\n",sum);
}
```

Ejercicio 5 (2 puntos)

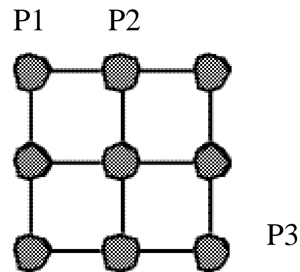
El siguiente programa paralelo se ejecuta en 3 procesadores y sus variables están inicializadas a 0.

Process 1	Process 2	Process 3
1a: A=1	2a: u=A	3a: v=B
	2b: B=1	3b: w=A

Se pide:

1. ¿Cuál de las siguientes salidas para (u, v, w) son posibles bajo consistencia secuencial: (1,0,0), (1,1,1), (1,1,0)? Justifica dando el orden de ejecución y explica por qué las soluciones imposibles no se pueden producir.
2. Asume que los procesos P1, P2 y P3 se mapean en una malla de 3x3 tal y como se muestra en la figura. El sistema no tiene coherencia caché. Explica cómo las soluciones del apartado anterior se pueden producir violando el principio de consistencia secuencial.

.



Solución:

1. (1,0,0) : 3a, 3b, 1a, 2a, 2b
   (1,1,1) : 1a, 2a, 2b, 3a, 3b
   (1,1,0) : no se puede producir: asumamos que se produce. Para W=0, A debería ser 0, V=B debe haberse ejecutado antes, y como V=1 => B debe ser 1 => B=1 fue acabado lo que implica que u=A fue acabado => u=0 lo que es una contradicción+

2. A=1 es ejecutado por P1, como la caché del sistema no garantiza consistencia el proceso P2 puede ver el valor modificado antes que P3. De este modo, P2 ejecuta u=A=1 y B=1. P3 ejecuta v=B=1 y w=A=0. (1,1,10) es una solución possible en este caso.

Ejercicio 6 (2 puntos)

Indique el coste en tiempo de la operación ***MPI_gather*** de un vector de 8 Mbytes sobre un sistema multiprocesador con 8 nodos cada uno y configuración hipercubo. Asuma las siguientes características en la red:

- No existe restricción en el tamaño de paquete (cualquier conjunto de datos puede ser enviado en un único paquete).
- La red opera síncronamente
- Cada switch puede recibir o enviar simultáneamente por todas sus conexiones
- Protocolo de encaminamiento del *switch*: *store and forward*.
- Ancho de banda: 100 Mbits/seg.
- *Routing delay* (retardo de encaminamiento del *switch*): 20 ms.
- Retardo de envío y recepción del procesador de comunicaciones: 0 ms.

Solución:

Cualquier comunicación entre dos nodos implica la intervención de dos switches y un enlace. El tiempo, por tanto, vendrá dado por:

- **T = 2 x 20ms + (n*1Mbits)/100Mbps**

Suponiendo que la operación gather se realiza sobre el procesador 1 y las características de la red, un posible patrón de comunicación sería:

1. **Comunicación de P8 con P6, P7 con P5, P4 con P2, P3 con P1. En este caso se hacen 4 envíos en paralelo que pasan por un enlace y dos switches. (T1= 0,04s + 0,01s = 0,05s)**
2. **Comunicación de P5 con P1, P6 con P2. Aquí se realizan dos envíos en paralelo por un solo enlace y dos switches pero con el doble de datos cada uno (T2= 0,04s + 0,02 = 0,06)**
3. **Comunicación de P2 con P1. Un solo envío, un enlace y dos switches, triple volumen de datos (T3 = 0,04s + 0,03s = 0,07s)**

El tiempo total en este caso es:

Ttotal = T1 + T2 + T3 = 0,05 + 0,04 + 0,07 = 0,16s