



ATENCIÓN:

- Lea atentamente todo el enunciado antes de comenzar a contestar.
 - No se podrán utilizar libros ni apuntes, ni calculadoras de ningún tipo.
 - Los teléfonos móviles deberán permanecer desconectados durante la prueba (apagados, no silenciados).
 - Solamente se corregirán los ejercicios contestados con bolígrafo. Por favor no utilice lápiz.
 - **La duración del examen es de 150 minutos.**
 - **Presente cada ejercicio en una hoja distinta. Si no hace un ejercicio entregue una hoja en blanco con el número del ejercicio.**
-

NOMBRE:

APELLIDOS:

NIA:

Ejercicio 1 [3 puntos]:

Se dispone de un computador que dispone de una caché L1 para datos asociativa por conjuntos de 2 vías y con un tamaño de 32 KB y una caché L1 para instrucciones de iguales características. También dispone de una caché L2 unificada asociativa por conjuntos de 4 vías con un tamaño de 256 KB. En ambos casos el tamaño de línea es de 64 bytes. Se asume que un acierto en la caché L1 requiere 3 ciclos, y un acierto en la caché L2 requiere 9 ciclos adicionales y la penalización por traer un bloque de memoria principal a la caché de nivel 2 es de 60 ciclos. Todas las cachés tienen una política de escritura write-back.

Sobre esta máquina se tiene el siguiente fragmento de código:

```
struct body {  
    double x, y, z;  
    double vx, vy, vz;  
    double ax, ay, az;  
    double m;  
};  
void f(std::vector<body> & v, double dt) {  
    const size_t sz = v.size();  
    for (int i=0; i< sz; ++i) {  
        v[i].vx += v[i].ax * dt;  
        v[i].vy += v[i].ay * dt;  
        v[i].vz += v[i].az * dt;  
        v[i].x += v[i].vx * dt;  
        v[i].y += v[i].vy * dt;  
        v[i].z += v[i].vz * dt;  
    }  
}
```

1.1 [0.5 puntos]: Determine cuál debería ser el tiempo medio de acceso para datos asumiendo que tiene las siguientes tasas de fallo:

- L1 datos: 3%
- L2: 1%



1.2 [1.5 puntos]: Determine el número de fallos en acceso a datos que se producen durante la ejecución del bucle de la función $f()$ del enunciado para la caché L1 (inicialmente vacía) si el tamaño del vector es de 64 elementos. Asuma que las variables i y sz se asignan a registros y no generan accesos de datos. El bloque de datos del vector v se asume que se encuentra alineado a una dirección múltiplo de 64.

1.3 [1 punto]: Proponga una modificación de código usando arrays paralelos (es decir eliminando la fusión de arrays, y usando un array independiente para cada campo de la estructura original) en vez de fusión de arrays y determine cuáles serían las nuevas tasas de fallo, asumiendo que todos los bloques de datos de los arrays se almacenan de forma consecutiva en memoria.

SOLUCIÓN:

1.1:

$$T_a = t_h(L1) + m(L1) * (t_h(L2) + m(L2) * t_p(L2))$$

$$T_a = 3 + 0.03 * (9 + 0.01 * 60) = 3 + 0.03 * 9.6 = 3,228$$

1.2:

Cada elemento del array ocupa 10 valores de 8 bytes, por lo que ocupa 80 bytes. Cada línea de la caché ocupa 64 bytes. Esto hace que los elementos del array no encajen perfectamente en las entradas de la caché. El mínimo común múltiplo de 80 y 64 es 320. Por tanto, se tiene que se produce un encaje para cada 320 bytes, que equivale a 5 líneas de caché y 4 posiciones del array.

La caché L1 tiene 32 KB. El tamaño por vía es de 16 KB. Como el tamaño de línea es 64 B el número de líneas por vía es de $2^{14} / 2^6 = 2^8$.

Como cada elemento del vector necesita 80 bytes. El número de elementos que se puede tener en una vía es $2^{14} / (2^5 * 5) = 2^9 / 5 > 102$. Por tanto, los 64 elementos del vector caben en una única vía de la caché L1.

Para analizar el número de fallos y aciertos, se analizarán los accesos a las primeras 4 posiciones del array, puesto que los resultados son extrapolables al resto. Su disposición en memoria es

L0: $v[0].x, v[0].y, v[0].z, v[0].vx, v[0].vy, v[0].vz, v[0].ax, v[0].ay$

L1: $v[0].az, v[0].m, v[1].x, v[1].y, v[1].z, v[1].vx, v[1].vy, v[1].vz$

L2: $v[1].ax, v[1].ay, v[1].az, v[1].m, v[2].x, v[2].y, v[2].z, v[2].vx$

L3: $v[2].vy, v[2].vz, v[2].ax, v[2].ay, v[2].az, v[2].m, v[3].x, v[3].y$

L4: $v[3].z, v[3].vx, v[3].vy, v[3].vz, v[3].ax, v[3].ay, v[3].az, v[3].m$



En cada iteración se producen un total de 18 accesos a memoria (dos lecturas y una escritura por sentencia, al usarse el operador +=). En cuatro iteraciones, se producen un total de $4 \cdot 18 = 72$ accesos. De ellos 5 serán fallos en caché. En total se producen $5 \cdot 64 / 4 = 80$ fallos

Por tanto la tasa de fallos de la caché es de $5/72 = 0,06944 \rightarrow 6,94\%$

1.3

Si se usan arrays paralelos se tiene:

```
void f(std::vector<double> & x, std::vector<double> & y, std::vector<double> & z,  
    std::vector<double> & vx, std::vector<double> & vy, std::vector<double> & vz,  
    std::vector<double> & ax, std::vector<double> & ay, std::vector<double> & az,  
    double dt) {  
    const size_t sz = x.size();  
    for (int i=0; i<sz; ++i) {  
        vx[i] += ax[i] * dt;  
        vy[i] += ay[i] * dt;  
        vz[i] += az[i] * dt;  
        x[i] += vx[i] * dt;  
        y[i] += vy[i] * dt;  
        z[i] += vz[i] * dt;  
    }  
}
```

Cada uno de los nueve arrays (el array m no se usa) necesita $8 \cdot 64 \text{ bytes} = 2^3 \cdot 2^6 \text{ bytes} = 2^9 \text{ bytes}$, que equivalen a 8 líneas de caché consecutivas. El número total de líneas usado por todos los arrays es de $8 \cdot 9 = 72$, que es muy inferior a las 256 líneas de la caché L1 por vía.

En este caso es necesario analizar lo que ocurre en 8 iteraciones, puesto que en una línea caben 8 elementos de array por línea.

Iteración 0: 9 fallos (uno por array) y 9 aciertos.

Iteraciones 1 a 7: 18 aciertos por iteración.

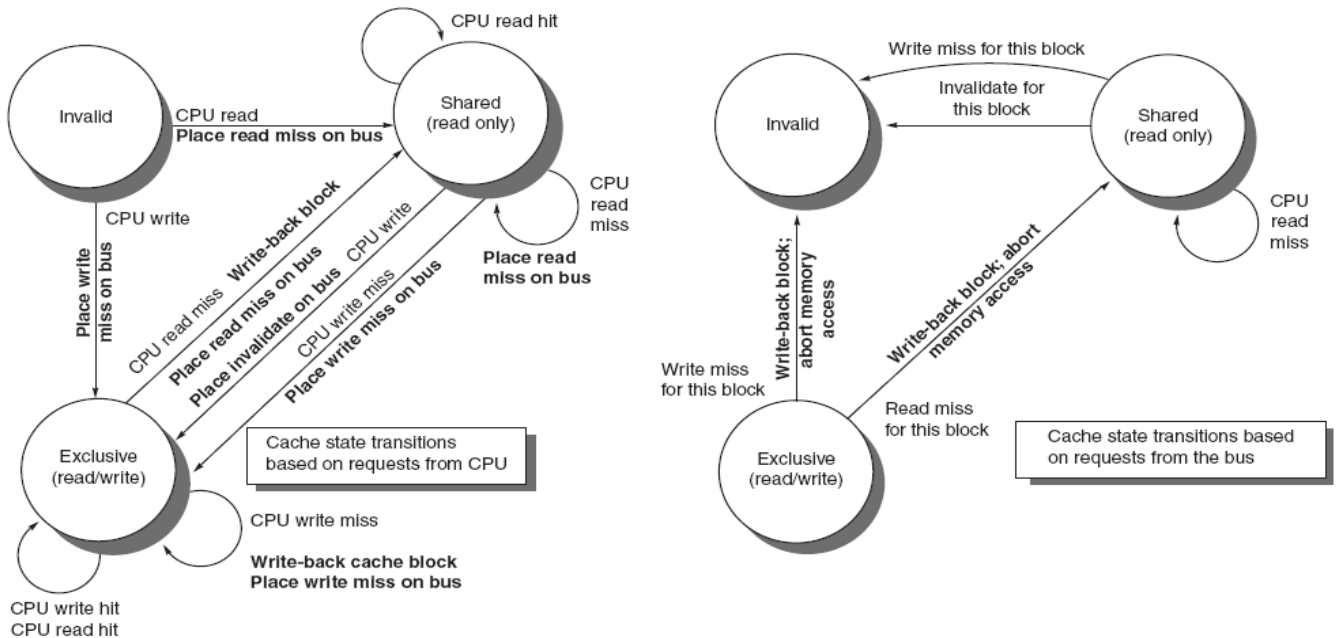
En total para 8 iteraciones, se tiene 9 fallos y 135 aciertos ($9 + 18 \cdot 7$), por lo que la tasa de fallos es $9/144 = 0.0625 \rightarrow 6,25\%$

Ejercicio 2 [2.5 puntos]:

Sea un procesador con tres núcleos con arquitectura de memoria compartida simétrica basado en bus con protocolo de espionaje o *snooping*. Cada núcleo tiene un único nivel de memoria caché que es privada para el núcleo y cuya coherencia se mantiene usando el protocolo MSI. Las memorias cachés utilizan correspondencia directa y cada línea caché puede almacenar bloques de dos palabras.



Examen Ordinario
20 de enero de 2015



Los siguientes escenarios muestran distintas situaciones de ejecución de distintos hilos en cada núcleo. Asíumase que:

- Cada escenario es independiente y para cada uno las memorias caché están inicialmente vacías.
- Los vectores a y b tienen dos entradas, están alineados en memoria almacenándose cada uno de ellos en único bloque.
- Debido al algoritmo de correspondencia, los vectores a y b se almacenan en la misma línea caché.
- Se pide: indicar para cada escenario el contenido a lo largo del tiempo de las memorias caché de cada núcleo siguiendo el formato de la tabla adjunta (hacer una para cada escenario). **Además, debe justificar la respuesta.**

Tiempo	Núcleo 1		Núcleo 2		Núcleo 3		Tráfico bus
	Estado	Contenido (a o b)	Estado	Contenido (a o b)	Estado	Contenido (a o b)	



- **Escenario 1:**

Tiempo	Hilo 1 (ejecutado en núcleo 1)	Hilo 2 (ejecutado en núcleo 2)	Hilo 3 (ejecutado en núcleo 3)
0	Read a[0]		
1		Read a[0]	
2			Write a[0]

- **Escenario 2:**

Tiempo	Hilo 1 (ejecutado en núcleo 1)	Hilo 2 (ejecutado en núcleo 2)	Hilo 3 (ejecutado en núcleo 3)
0		Write a[0]	
1			Write a[0]
2			Write a[0]
3			Read a[0]



- Escenario 3:

Tiempo	Hilo 1 (ejecutado en núcleo 1)	Hilo 2 (ejecutado en núcleo 2)	Hilo 3 (ejecutado en núcleo 3)
0		Read a[0]	
1			Read a[0]
2	Write a[1]		
3			Read a[0]

- Escenario 4:

Tiempo	Hilo 1 (ejecutado en núcleo 1)	Hilo 2 (ejecutado en núcleo 2)	Hilo 3 (ejecutado en núcleo 3)
0		Write a[0]	
1		Read b[0]	
2			Write a[0]
3		Write b[0]	
4		Write a[0]	



Solución:

Escenario 1

Tiempo	Núcleo 1		Núcleo 2		Núcleo 3		Tráfico bus
	Estado	Contenido (a o b)	Estado	Contenido (a o b)	Estado	Contenido (a o b)	
	S	a					Read miss Envío a
	S	a	S	a			Read miss Envío a
	I	a	I	a	E	a	Write miss Envío a

Escenario 2

Tiempo	Núcleo 1		Núcleo 2		Núcleo 3		Tráfico bus
	Estado	Contenido (a o b)	Estado	Contenido (a o b)	Estado	Contenido (a o b)	
			E	a			Write miss Envío a
			I	a	E	a	Write miss Write back a
			I	a	E	a	
			I	a	E	a	



Escenario 3

Tiempo	Núcleo 1		Núcleo 2		Núcleo 3		Tráfico bus
	Estado	Contenido (a o b)	Estado	Contenido (a o b)	Estado	Contenido (a o b)	
			S	a			Read miss Envío a
			S	a	S	a	Read miss Envío a
	E	a	I	a	I	a	Write miss Envío a
	S	a	I	a	S	a	Read miss Write-back a

Escenario 4

Tiempo	Núcleo 1		Núcleo 2		Núcleo 3		Tráfico bus
	Estado	Contenido (a o b)	Estado	Contenido (a o b)	Estado	Contenido (a o b)	
0			E	a			Write miss a Envío a
1			S	b			Read miss b Write-back a Envío b
2			S	b	E	a	Write miss a Envío a
3			E	b	E	a	Invalidate
4			E	a	I	a	Write back b Write miss a Write back a

Ejercicio 3 [1 punto]:

Dado el siguiente código, conteste a las preguntas a continuación:

```
01 void exec(int a){
```



```
02  /*
03   * This code needs "a" seconds of CPU to be executed
04   */
05  }
06
07  int main(){
08
09   #pragma omp parallel for
10   for(int i = 0; i < 100; i++){
11       exec(i);
12   }
13
14   return 0;
15 }
16
```

3.1 [0.5 puntos] ¿Qué hace el la directiva de la línea 9? Considere que ninguna variable de entorno relativa a OpenMP está asignada en la sesión que ejecuta el programa.

3.2 [0.5 puntos] ¿Cuál es el mejor planificador de openmp para la directiva de la línea 9 utilizando un tamaño de *chunk* de 10? Justifique su respuesta comparando el comportamiento de todos los planificadores posibles.

SOLUCIÓN

a) La directiva de la línea 9 realiza dos operaciones.

1.- Genera tantos hilos como núcleos (reales o virtuales) tenga la máquina sobre la que se está ejecutando. Se hará un join de los hilos al final del bucle.

2.- Reparte el bucle entre los hilos creados. Dado que no se indica ningún planificador, utilizará por defecto el planificador *static* y repartirá de forma equitativa las iteraciones entre el número de hilos que haya.

b) Si ponemos de ejemplo una ejecución con 4 hilos:

Dado que la carga de trabajo aumenta a medida que aumenta el índice de iteración, el peor planificador será "static" sin tamaño de chunk, ya que, repartirá 25 iteraciones a cada hilo y las 25 últimas iteraciones tardan mucho más que las 25 primeras. El tiempo total estimado con 4 hilos es el tiempo de las últimas 25 iteraciones: $99 + 98 + 97 + 96 \dots + 76 + 75 = 2175$.

Los planificadores dynamic y static con chunk 10 se comportarán de forma similar. El dynamic repartirá 10 iteraciones a cada hilo y, cuando terminen con ellas, les repartirá 10 más. El estático repartirá de 10 en 10 entre los hilos en un orden similar al dinámico. Un posible reparto sería:

Hilo 0	0 a 9	36	
	40 a 49	435	
	80 a 89	835	1306 segundos



Hilo 1	10 a 19	135	
	50 a 59	535	
	90 a 99	935	1605 segundos

Hilo 2	20 a 29	235	
	60 a 69	635	
			870 segundos

Hilo 3	30 a 39	335	
	70 a 79	735	
			1070 segundos

Por lo que el tiempo total bajaría hasta 1605 segundos.

El último caso sería el planificador “guided”. En este caso se va ajustando el número de iteraciones a medida que avanza la el programa para adaptarse a la carga restante. Por tanto, al final de la ejecución que es cuando más influye el reparto, tendrá un grano más fino y conseguirá el mejor reparto. Cercano a la mejor planificación posible.

Ejercicio 4 [1.5 puntos]:

Dado el siguiente código, conteste a las preguntas que se encuentran a continuación.

```
01 #include <iostream>
02 #include <iomanip>
03 #include <thread>
04 #include <vector>
05
06 double sum;
07 double step;
08 int n_threads;
09
10 void calculate_steps(int thread_id, long nsteps) {
11     for (int i=thread_id; i<nsteps; i=i+n_threads) {
12         double x = (i+0.5) * step;
13         sum += 4.0 / (1.0 + x * x);
14     }
15 }
16
17 int main(){
18     using namespace std;
19     // Init global variables
```



```
20  n_threads = 4;
21  long nsteps = 100000000;
22  step = 1.0 / double(nsteps);
23  sum = 0.0;
24  std::vector<std::thread> threads;
25
26  // Create threads
27  for(int i = 0; i < n_threads; i++){
28      threads.push_back(std::thread(calculate_steps, i, nsteps));
29  }
30  for (int i = 0; i < n_threads; ++i) {
31      threads[i].join();
32  }
33
34  // Calculate reduction
35  double pi = step * sum;
36  cout << "PI= " << setprecision(10) << pi << endl;
37
38  return 0;
39 }
```

4.1 [0.25 puntos]: Indique las líneas del código que forman parte de una sección crítica. Justifique su respuesta.

4.2 [0.25 puntos]: Proponga una solución basada en cerrojos a la sección crítica. Justifique su respuesta.

4.3 [0.5 puntos]: Proponga una solución libre de cerrojos a la sección crítica. Justifique su respuesta.

4.4 [0.5 puntos]: Si se ejecuta el programa en una máquina con dos núcleos, ¿cuál de las dos soluciones tendría mejor rendimiento utilizando dos hilos ($n_threads = 2$)? ¿Y utilizando 32 hilos ($n_threads = 32$)? Justifique ambas respuestas.



SOLUCION

- a) La única parte conflictiva del código se encuentra en la línea 12, dónde todos los hilos intentan modificar de forma concurrente una variable compartida sum.

b)

```
#include <iostream>
#include <iomanip>
#include <thread>
#include <vector>
#include <mutex>
```

```
double sum;
double step;
int n_threads;
std::mutex mtx;
```

```
void calculate_steps(int thread_id, long nsteps) {
    for (int i=thread_id; i<nsteps; i=i+n_threads) {
        double x = (i+0.5) * step;
        std::unique_lock<std::mutex> l{mtx};
        sum += 4.0 / (1.0 + x * x);
    }
}
```

```
int main(){
    using namespace std;

    // Init global variables
    n_threads = 16;
    long nsteps = 10000000;
    step = 1.0 / double(nsteps);
    sum = 0.0;
    std::vector<std::thread> threads;
```

```
// Create threads
for(int i = 0; i < n_threads; i++){
    threads.push_back(std::thread(calculate_steps,i,nsteps));
}
for (int i = 0; i < n_threads; ++i) {
    threads[i].join();
}
```

```
// Calculate reduction
double pi = step * sum;
```



```
cout << "PI= " << setprecision(10) << pi << endl;
```

```
return 0;
```

```
}
```

```
c) #include <iostream>
#include <iomanip>
#include <thread>
#include <vector>
#include <atomic>
```

```
double sum;
```

```
double step;
```

```
int n_threads;
```

```
std::atomic_flag f;
```

```
void calculate_steps(int thread_id, long nsteps) {
    for (int i=thread_id; i<nsteps; i=i+n_threads) {
        double x = (i+0.5) * step;
        while(f.test_and_set());
        sum += 4.0 / (1.0 + x * x);
        f.clear();
    }
}
```

```
int main(){
```

```
    using namespace std;
```

```
    // Init global variables
```

```
    n_threads = 4;
```

```
    long nsteps = 100000000;
```

```
    step = 1.0 / double(nsteps);
```

```
    sum = 0.0;
```

```
    std::vector<std::thread> threads;
```

```
    // Create threads
```

```
    for(int i = 0; i < n_threads; i++){
```

```
        threads.push_back(std::thread(calculate_steps,i,nsteps));
```

```
    }
```

```
    for (int i = 0; i < n_threads; ++i) {
```

```
        threads[i].join();
```

```
    }
```

```
    // Calculate reduction
```

```
    double pi = step * sum;
```

```
    cout << "PI= " << setprecision(10) << pi << endl;
```



Universidad
Carlos III de Madrid

Departamento de Informática
Grado en Ingeniería Informática
Arquitectura de Computadores

Examen Ordinario
20 de enero de 2015



```
return 0;  
}
```

- d) Se puede aceptar como respuesta válida que los atómicos funcionan mejor si hay suficientes recursos disponibles para realizar la espera activa (dos núcleos y dos hilos) pero los mutex se comportarán mejor en caso de que haya más hilos realizando espera activa que recursos disponibles.
Se valorará positivamente aludir a los conceptos de: mutex implica llamadas al sistema, atómicos funcionan bien si hay soporte hardware.