

2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores: Exámenes y Controles

## Examen Final AC 20/06/2012 resuelto

Material elaborado por los profesores responsables de la asignatura:  
Mancia Anguita, Julio Ortega

Licencia Creative Commons 

### 1 Enunciado Examen del 20/06/2012

**Cuestión 1. (0.5 puntos)** ¿Cuál de los siguientes modelos de consistencia permite conseguir mejores tiempos de ejecución? Justifique su respuesta.

- (a) modelo de ordenación débil
- (b) modelo implementado en los procesadores de la línea x86
- (c) modelo de consistencia secuencial

**Cuestión 2. (0.5 puntos)** Suponga un CC-NUMA con protocolo MSI basado en directorios distribuidos. Suponiendo que un bloque se encuentra en el directorio en estado inválido indique cual sería el contenido del directorio, las transiciones de estados (en cache y en el directorio) y la secuencia de paquetes generados por el protocolo de coherencia si un procesador que no tiene el bloque en su cache escribe en una dirección de dicho bloque. Razone su respuesta.

**Ejercicio 1. (1 punto)** En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicional es mejor:

- ALT1: Una instrucción COMPARE actualiza un código de condición y es seguida por una instrucción BRANCH que comprueba esa condición.
- ALT2: Una sola instrucción incluye la funcionalidad de las instrucciones COMPARE y BRANCH.

Hay que tener en cuenta que hay un 25% de saltos condicionales en ALT1; que las instrucciones BRANCH en ALT1 y COMPARE+BRANCH en ALT2 necesitan 6 ciclos mientras que todas las demás necesitan sólo 2; y que el ciclo de reloj de la ALT1 es un 80% del ciclo de ALT2, dado que con esta última alternativa la mayor funcionalidad de la instrucción COMPARE+BRANCH ocasiona una mayor complejidad en el procesador.

**Ejercicio 2. (0.5 puntos)** Implemente un cerrojo simple para un procesador que no garantiza el orden W->R usando Fetch\_and\_Or.

**Ejercicio 3. (1.5 puntos)** Considere el bucle:

```
do {  
    b[i]=a[i]*c;  
    c=c+1;  
    if (c>10) then goto etiqueta;  
    while (i<10);  
    etiqueta:.....
```

Indique cuál es la penalización efectiva debida a los saltos para c=5 considerando que el procesador utiliza:

- (a) Predicción fija (siempre se considera que se va a producir el salto)



- (b) Predicción estática (si el desplazamiento es negativo se toma y si es positivo no)
- (c) Predicción dinámica con dos bits usando el diagrama de estados visto en clase.

Nota: La penalización por saltos incorrectamente predichos es de 4 ciclos y para los saltos correctamente predichos es 0 ciclos.

**Ejercicio 4. (2 puntos)** Se quiere realizar un programa paralelo con paso de mensajes que obtenga el valor máximo de los componentes de un vector,  $v[]$ , con  $N$  elementos en coma flotante.

- (a) Escriba un programa secuencial con notación algorítmica (podría escribirlo en C) para obtener el máximo. ¿Cuántas operaciones de comparación realiza el código en total?
- (b) Escriba ahora un programa paralelo con notación algorítmica (podría escribirlo en C) teniendo en cuenta que: **(1)** El proceso 0 tiene inicialmente el vector  $v$  en su memoria e imprimirá en pantalla el resultado. **(2)** La herramienta de programación proporciona las funciones `send()`/`receive()` que permiten la comunicación uno-a-uno asíncrona:

`send(buffer, count, datatype, idproc, group)` no bloqueante y  
`receive(buffer, count, datatype, idproc, group)` bloqueante,

donde `group` es el identificador del grupo de procesos que intervienen en la comunicación; `idproc` es el identificador del proceso al que se envía o del que se recibe; `buffer` es la dirección a partir de la que se almacenan los datos que se envían o reciben; `datatype` es el tipo de datos a recibir o enviar; y `count` es el número de datos a transferir del tipo `datatype`.

- (c) Dibuje **(1)** el grafo de dependencias de tareas para este ejemplo a partir del código secuencial y **(2)** la estructura (grafo) de los procesos para su código paralelo. ¿Con qué tipo de estructuras de las estudiadas en AC se corresponde la estructura de procesos? Razone su respuesta.
- (d) ¿Cuántas operaciones de comparación realiza cada proceso (procesador/core) si el número de procesos  $p$  divide a  $N$ ? Razone su respuesta.
- (e) Estime el tiempo de ejecución del código secuencial  $T_s$  considerando que coincide con el tiempo de cálculo que suponen las operaciones de comparación (llame  $t$  al tiempo que supone una comparación).
- (f) Estime la ganancia en prestaciones conseguida con la paralelización para un número de procesadores/cores de 10 y  $N=10^6$  **(1)** usando para estimar el tiempo de cálculo paralelo  $T_c$  sólo el tiempo que suponen las operaciones de comparación (llame  $t$  al tiempo que supone una comparación), **(2)** usando para estimar el tiempo de sobrecarga (*overhead*)  $T_o$  sólo el tiempo de comunicación  $T_{c/s}$ , **(3)** considerando el tiempo de transferencia de un mensaje (comunicación entre procesos) igual a  $t_1 + n t_2$  ( $t_1=10000t$  y  $t_2=2t$  son constantes y  $n$  es el número de flotantes del mensaje), **(4)** considerando que las transferencias uno-a-uno se realizan en paralelo en la arquitectura sin conflicto entre transferencias distintas aunque su origen y/o su destino sean el mismo, y **(5)** considerando que cada proceso se asigna a un procesador/core distinto.



## 2 Solución Examen del 20/06/2012

**Cuestión 1.(0.5 puntos)** ¿Cuál de los siguientes modelos de consistencia permite conseguir mejores tiempos de ejecución? Justifique su respuesta.

- a) modelo de ordenación débil
- b) modelo implementado en los procesadores de la línea x86
- c) modelo de consistencia secuencial

### Solución

El modelo de consistencia débil relaja todos los órdenes entre operaciones de acceso a memoria independientes ( $W \rightarrow R$ ,  $W \rightarrow W$ ,  $R \rightarrow W, R$ ), el modelo de consistencia secuencial no relaja ningún orden y el de los procesadores x86 sólo relaja en operaciones escalares el orden  $W \rightarrow R$ .

El mejor es el modelo de consistencia débil (la opción (a)) porque, al relajar todos los órdenes en los accesos a memoria, permite mejores tiempos de ejecución que el resto ya que ningún acceso tiene que esperar a que termine otro anterior en el código siempre que los dos sean accesos a distintas posiciones de memoria.



**Cuestión 2.(0.5 puntos)** Suponga un CC-NUMA con protocolo MSI basado en directorios distribuidos. Suponga un bloque B que se encuentra en memoria principal en estado inválido, indique (a) cual sería el contenido de la entrada del directorio para ese bloque en esta situación y (b) las transiciones de estados (en cache y en el directorio), la secuencia de paquetes generados por el protocolo de coherencia y el contenido al que pasa la entrada del directorio para ese bloque si un procesador que no tiene el bloque en su cache y no está en el nodo origen del bloque escribe en una dirección de dicho bloque. Razone su respuesta.

### Solución

(a) Si el bloque está inválido en memoria principal habrá una copia del bloque válida en un única cache. El contenido en la entrada del bloque del directorio en esa situación tendrá en inválido (a 0) el bit de estado del bloque en memoria y, en su vector de bits de presencia, tendrá un único bit activo, el resto estarán a 0. El bit activo será el que corresponda a la cache que tiene la única copia válida del bloque en el sistema.

Vector de bits de presencia					Estado
0	...	0	1	0	0

(b) Si un procesador de un nodo que no es el origen del bloque escribe en una dirección del bloque:

1. Se produce un fallo de escritura que provoca la emisión desde este nodo del procesador que escribe (o nodo solicitante S) al nodo origen del bloque (o nodo que tiene el bloque en un módulo de memoria principal) de un paquete de petición de lectura con acceso exclusivo al bloque  $PtLecEx(B)$ .

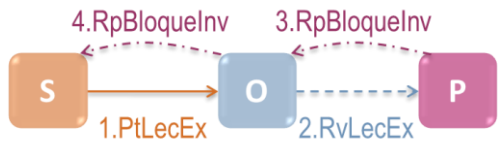
2. El nodo origen (O) al recibir este paquete, como tiene el bloque inválido, re-envía la petición al nodo propietario (P) del bloque  $RvLecEx(B)$ . Obtiene el nodo propietario del directorio, el bit activo de la entrada del directorio para el bloque identifica al único nodo propietario.

3. El nodo propietario (P), al recibir la petición de lectura y acceso exclusivo a B,  $RvLecEx(B)$ , invalida la copia que tiene del bloque (porque pide acceso exclusivo) y responde con el bloque (porque pide lectura) al origen confirmando acceso exclusivo, es decir, confirma la invalidación del bloque en su cache. Por tanto, envía al origen un paquete de respuesta con el bloque confirmando la invalidación de la copia del bloque en su cache,  $RpBloqueInv(B)$ .

4. El nodo origen (O), cuando recibe la respuesta la re-envía al nodo solicitante  $RpBloqueInv(B)$ . Además mantiene el estado del bloque en inválido, porque lo va a modificar el nodo solicitante, y en el vector de bit

de presencia activará el bit que se corresponde con el nodo solicitante (que es donde estará la única copia válida del bloque), el resto de bits de presencia estarán a 0.

5. El nodo solicitante (S), cuando recibe el paquete con el bloque, RpBloqueInv (B), lo introduce en su cache, lo modifica y pone estado modificado en la entrada del bloque en el directorio de su cache (no confundir con el directorio de memoria principal).

Estado inicial	Evento	Estado final	Diagrama de paquetes
D) Inválido S) Inválido P) Modificado Acceso remoto	Fallo de escritura	D) Inválido S) Modificado P) Inválido	



**Ejercicio 5. (1 punto)** En un procesador sin segmentación de cauce, determine cuál de estas dos alternativas para realizar un salto condicional es mejor:

- ALT1: Una instrucción COMPARE actualiza un código de condición y es seguida por una instrucción BRANCH que comprueba esa condición.
- ALT2: Una sola instrucción incluye la funcionalidad de las instrucciones COMPARE y BRANCH.

Hay que tener en cuenta que hay un 25% de saltos condicionales en ALT1; que las instrucciones BRANCH en ALT1 y COMPARE+BRANCH en ALT2 necesitan 6 ciclos mientras que todas las demás necesitan sólo 2; y que el ciclo de reloj de la ALT1 es un 80% del ciclo de ALT2, dado que en con esta última alternativa la mayor funcionalidad de la instrucción COMPARE+BRANCH ocasiona una mayor complejidad en el procesador.

### Solución

Datos del ejercicio (considerando 25% de instrucciones BRANCH en ALT1 para el conjunto de programas de prueba):

$$\text{Tiempo de ciclo de ALT1} = 0,8 * \text{Tiempo de ciclo de ALT2} \quad (T_{\text{ciclo}}^1 = 0,8 * T_{\text{ciclo}}^2)$$

Datos para ALT1 (considerando 25% de BRANCH en ALT1;  $NI_i$ : nº de instrucciones del tipo i en el conjunto de programas de prueba;  $NI^1$ : nº de instrucciones en total en el conjunto de programas de prueba en ALT1):

Instrucciones i	ciclos	$NI_i/NI^1$	Proporción (%)	Comentarios
BRANCH en ALT1	6	0,25	25	
RESTO en ALT1	2	0,75	75	Incluye instrucciones COMPARE
TOTAL		1	100	

Datos para ALT2 (considerando 25% de instrucciones BRANCH en ALT1;  $NI^2$ : número de instrucciones en total en el conjunto de programas de prueba en ALT2):

Instrucciones i	ciclos	$NI_i/NI^1$	$NI_i/NI^2$	Comentarios
COMPARE+BRANCH en ALT2	6	0,25	$25/75 = 1/3$	
RESTO en ALT2	2	$0,75 - 0,25 = 0,5$	$(75-25)/75 = 2/3$	Se quitan las instrucciones COMPARE que se han añadido a BRANCH, que son un 25% de $NI^1$
TOTAL		0,75	1	$NI^2 = 0,75 * NI^1$ (al no tener ALT2 instr. COMPARE aparte)

El tiempo de CPU se puede expresar como:

$$T_{CPU} = CPI * NI * T_{CICLO}$$

donde CPI es el número medio de ciclos por instrucción, NI es el número de instrucciones en total del conjunto de programas de prueba, y  $T_{CICLO}$  el tiempo de ciclo del procesador.

Para la alternativa *ALT1* se tiene:

$$T_{CPU}^1 = CPI^1 * NI^1 * T_{ciclo}^1 = (0.25 * 6 + 0.75 * 2) * NI^1 * T_{ciclo}^1 = 3 * NI^1 * 0.8 * T_{ciclo}^2 = 2.4 * NI^1 * T_{ciclo}^2$$

Para la alternativa *ALT2* se tiene:

$$CPI^2 = (0.25 * NI^1 * 6 + 0.5 * NI^1 * 2) / (0.75 * NI^1) = (0.25 * 6 + 0.5 * 2) / 0.75$$

$$T_{CPU}^2 = CPI^2 * NI^2 * T_{ciclo}^2 = [(0.25 * 6 + 0.5 * 2) / 0.75] * 0.75 * NI^1 * T_{ciclo}^2 = (0.25 * 6 + 0.5 * 2) * NI^1 * T_{ciclo}^2 = 2.5 * NI^1 * T_{ciclo}^2$$

Los dos tiempos se han calculado en función de las mismas variables. Como se puede ver es menor el tiempo de la alternativa 1:  $T_{CPU}^1 < T_{CPU}^2$ . Es decir, aunque un repertorio con instrucciones más complejas puede reducir el número de instrucciones de los programas, disminuyendo el número de instrucciones a ejecutar, esto no tiene que suponer una mejora de prestaciones, si al implementar esta opción la mayor complejidad del procesador lo hace algo más lento.



**Ejercicio 6. (0.5 puntos)** Implemente usando `Fetch_and_Or` un cerrojo simple para un procesador que no garantiza el orden W->R.

#### Solución

```
Lock(k): while (Fetch_and_Or(k, 1) == 0) {};
```

Aclaración: el modelo de consistencia garantiza que ningún acceso de lectura y escritura a las variables compartidas (se accede a las variables compartidas después del `lock`) adelantará la escritura del cerrojo `k` porque esta escritura acompaña a una lectura en una operación de lectura-modificación-escritura atómica (`Fetch_and_Or` es atómica) y el modelo de consistencia no permite que se adelante a lecturas anteriores en el código.

```
Unlock(k) : k=0;
```

Aclaración: el modelo de consistencia garantiza que la escritura del cerrojo `k` no adelanta a ningún acceso anterior en el código (se accede a las variables compartidas antes del `unlock`).



**Ejercicio 7. (1.5 puntos)** Considere el bucle:

```
i=0;
do {
    b[i]=a[i]*c;
    c=c+1;
    if (c>10) then goto etiqueta;
    i=i+1;
while (i<10);
etiqueta:.....
```

Indique cuál es la penalización efectiva debida a los saltos para un valor inicial de `c` igual a 5 considerando que el procesador utiliza:

- (a) Predicción fija (siempre se considera que se va a producir el salto)
- (b) Predicción estática (si el desplazamiento es negativo se toma y si es positivo no)

(c) Predicción dinámica con dos bits usando el diagrama de estados visto en clase (considerar que el estado inicial es 11 para todos los saltos).

Nota: La penalización por saltos incorrectamente predichos es de 4 ciclos y para los saltos correctamente predichos es 0 ciclos.

### Solución

#### Datos del ejercicio:

Hay dos instrucciones de salto. Identificaremos con 1 a la instrucción de salto que hay dentro del bucle (salto condicional hacia delante) e identificaremos con 2 a la que controla el final de las iteraciones del bucle (salto condicional hacia atrás). El bucle se repite mientras  $i$  es menor que 10 y mientras  $c$  no supere 10. Las dos variables  $i$  y  $c$  se incrementan en 1 cada iteración. Para  $c=5$  la primera condición de salida del bucle que se cumple es la correspondiente al salto que hay dentro del bucle porque  $c$  llega a 11 antes de que  $i$  llegue a 10. Así que, para  $c=5$ , la secuencia de Saltos y No Saltos de las instrucciones 1 y 2 es  $(N_1 S_2)^5 S_1$  (N significa que no se salta y S que se salta,  $()^5$  significa que se repite lo que hay dentro del paréntesis 5 veces). Por tanto, para la instrucción 1 tenemos  $N_1 N_1 N_1 N_1 N_1 S_1$  y para la instrucción 2  $S_2 S_2 S_2 S_2 S_2$ . Esta ejecución de los dos saltos queda reflejada en la siguiente tabla:

Iteración del bucle	1	2	3	4	5	6
$i$ al final de la iteración del bucle (valor inicial 0)	1	2	3	4	5	6
$c$ al final de la iteración del bucle (valor inicial 5)	6	7	8	9	10	11
Salto del if Ejecución	N	N	N	N	N	S
Salto del while Ejecución	S	S	S	S	S	

(a) Predicción fija (siempre se salta): hay 5 penalizaciones para la instrucción 1 (20 ciclos), como ilustra la siguiente tabla (P denota penalización):

Iteración del bucle	1	2	3	4	5	6	Total ciclos penalización
Predicción salto if y salto while	S	S	S	S	S	S	
Salto del if Ejecución	N	N	N	N	N	S	
Salto del if Penalización	P	P	P	P	P		$5 * 4 \text{ ciclos} = 20 \text{ ciclos}$
Salto del while Ejecución	S	S	S	S	S		
Salto del while Penalización							$0 * 4 \text{ ciclos} = 0 \text{ ciclos}$

(b) Predicción estática: hay una penalización para la instrucción 1 (4 ciclos) como ilustra la siguiente tabla:

Iteración del bucle	1	2	3	4	5	6	Total ciclos penalización
Predicción salto if	N	N	N	N	N	N	
Salto del if Ejecución	N	N	N	N	N	S	
Salto del if Penalización					P		$1 * 4 \text{ ciclos} = 4 \text{ ciclos}$
Predicción salto while	S	S	S	S	S		
Salto del while Ejecución	S	S	S	S			
Salto del while Penalización							$0 * 4 \text{ ciclos} = 0 \text{ ciclos}$

(c) Predicción dinámica: para la instrucción 1 hay tres penalizaciones (los dos primeros  $N_1$   $N_1$  y el último  $S_1$ ), por lo que hay 15 ciclos de penalización. Para la instrucción 2 no hay penalizaciones en este caso. La siguiente tabla ilustra las afirmaciones anteriores:

Iteración del bucle	1	2	3	4	5	6	Total ciclos penalización
Estado para if	1	1	0	0	0	0	
	1	0	1	0	0	0	
Predicción salto if	S	S	N	N	N	N	
Salto del if Ejecución	N	N	N	N	N	S	
Salto del if Penalización	P	P				P	3 * 4 ciclos = 12 ciclos
Estado para while	1	1	1	1	1	1	
	1	1	1	1	1	1	
Predicción salto while	S	S	S	S	S	S	
Salto del while Ejecución	S	S	S	S	S	S	
Salto del while Penalización							0 * 4 ciclos = 0 ciclos



**Ejercicio 8. (2 puntos)** Se quiere realizar un programa paralelo con paso de mensajes que obtenga el valor máximo de los componentes de un vector,  $V[]$ , con  $N$  elementos en coma flotante positivos.

(a) Escriba un programa secuencial con notación algorítmica (podría escribirlo en C) para obtener el máximo. ¿Cuántas operaciones de comparación realiza el código en total?

(b) Escriba ahora un programa paralelo con notación algorítmica (podría escribirlo en C) teniendo en cuenta que: **(1)** el proceso 0 tiene inicialmente el vector  $v$  en su memoria e imprimirá en pantalla el resultado; **(2)** el número de procesos (procesadores/cores)  $p$  divide a  $N$ ; **(3)** la herramienta de programación proporciona las funciones `send()`/`receive()` que permiten la comunicación uno-a-uno asíncrona:

`send(buffer, count, datatype, idproc, group)` no bloqueante y  
`receive(buffer, count, datatype, idproc, group)` bloqueante,

donde `group` es el identificador del grupo de procesos que intervienen en la comunicación; `idproc` es el identificador del proceso al que se envía o del que se recibe; `buffer` es la dirección a partir de la que se almacenan los datos que se envían o reciben; `datatype` es el tipo de datos a recibir o enviar; y `count` es el número de datos a transferir del tipo `datatype`.

(c) Dibuje **(1)** el grafo de dependencias de tareas para este ejemplo (parta del código secuencial) y **(2)** la estructura (grafo) de los procesos para su código paralelo. ¿Con qué tipo de estructuras de las estudiadas en AC se corresponde la estructura de procesos? Razone su respuesta.

(d) ¿Cuántas operaciones de comparación realiza cada proceso (procesador/core) si el número de procesos  $p$  divide a  $N$ ? Razone sus respuestas.

(e) Estime el tiempo de ejecución del código secuencial  $T_s$  considerando que coincide con el tiempo de cálculo que suponen las operaciones de comparación (llame  $t_c$  al tiempo que supone una comparación).

(f) Estime la ganancia en prestaciones conseguida con la paralelización para un número de procesadores/cores de 10 y  $N=10^6$  **(1)** usando para estimar el tiempo de cálculo paralelo  $T_c$  sólo el tiempo

que suponen las operaciones de comparación (llame  $t$  al tiempo que supone una comparación), **(2)** usando para estimar el tiempo de sobrecarga (*overhead*)  $T_0$  sólo el tiempo de comunicación  $T_{c/s}$ , **(3)** considerando el tiempo de transferencia de un mensaje (comunicación entre procesos) igual a  $t_1 + nt_2$  ( $t_1 = 10000t$  y  $t_2 = 2t$  son constantes y  $n$  es el número de flotantes del mensaje), **(4)** considerando que las transferencias uno-a-uno se realizan en paralelo en la arquitectura sin conflicto entre transferencias distintas aunque su origen y/o su destino sean el mismo, y **(5)** considerando que cada proceso se asigna a un procesador/core distinto.

### Solución

(a)

#### Pre-condición

$x$ : contendrá el máximo

$N$ : número de componentes del vector

$V[]$ : vector con los  $N$   $n^\circ$  de entrada

#### Pos-condición

Imprime en pantalla,  $x$ , el máximo encontrado en  $V[]$

#### Máximo secuencial

```
x = V[0];
for ( i=1 ; i<N; i++)
    if (V[i]>x) x=V[i];

printf("Máximo = %f \n", x);
```

**Comparaciones:** El número de instrucciones de comparación, sin contar las instrucciones que controlan la finalización del bucle, es igual a  $N-1$ , y  $2N-1$  contando las instrucciones que controlan la finalización del bucle

(b)

#### Pre-condición

$x$ : contendrá el máximo,  $x_{aux}$ : recepción de máximos

$N$ : número de componentes del vector

$V[]$ : vector con los  $N$   $n^\circ$  de entrada, todos los procesos tendrán un vector  $V$  pero el proceso 0 será el único que al principio de la ejecución tendrá en  $V$  los datos, el resto no los tiene (así lo establece el enunciado del ejercicio).

$grupo$ : identificador del grupo de procesos que intervienen en la comunicación.

$nproc$ : número de procesos en  $grupo$ .

$idproc$ : identificador del proceso (dentro del grupo de  $num\_procesos$  procesos) que ejecuta el código

$tipo$ : tipo de los datos a enviar o recibir

$count$ : es  $N/nproc$

#### Pos-condición

Imprime en pantalla,  $x$ , el máximo encontrado en  $V[]$



**Máximo paralelo**

```
//Difusión V, se supone que nproc divide a N
if (idproc==0)
    for (i=1; i<nproc; i++) send(V[i*count],count,tip0,i,grupo);

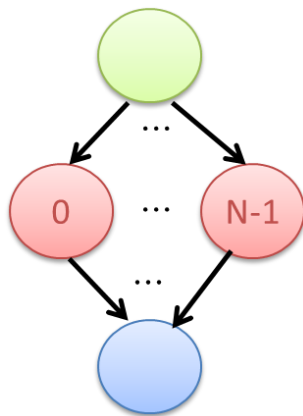
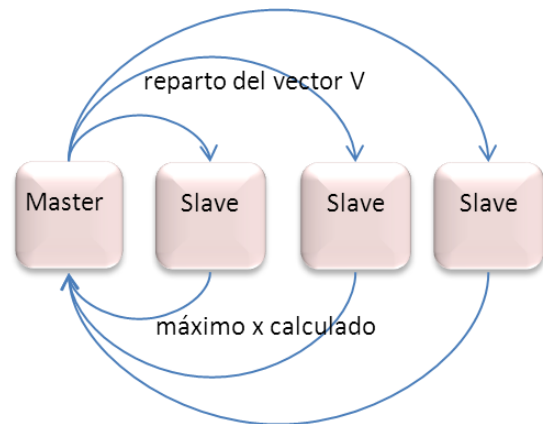
else receive(V,count,tip0,0,grupo);

//Cálculo paralelo, asignación estática
x = V[0];
for ( i=1 ; i<count; i++)
    if (V[i]>x) x=V[i];

//Comunicación resultados
if (idproc==0)
    for (i=1; i<nproc) {
        receive(xaux,1,tip0,i,grupo);
        if (xaux>x) x = xaux;
    }
else send(x,1,tip0,0,grupo);

//Proceso 0 imprime resultado
if (idproc==0) printf("Máximo = %f \n", x);
```

(c)

**Grafo de tareas****Grafo de procesos**

(d) El número de instrucciones de comparación que se realizan en paralelo, sin contar las comparaciones que controlan el final del bucle, es igual a  $(N/n_{\text{proc}}) - 1$ . Cada procesador realiza  $(N/n_{\text{proc}}) - 1$  comparaciones ya que el bucle en todos los procesos se ejecuta desde 1 a  $(N/n_{\text{proc}}) - 1$ . A cada proceso se trasfiere  $N/n_{\text{proc}}$  números que se recogen en su vector  $V$ . El proceso 0 realiza  $n_{\text{proc}} - 1$  comparaciones más que el resto porque calcula el máximo de los máximos parciales calculados por todos los procesos. Si se cuentan las instrucciones de comparación que se realizan para controlar el final del bucle, el total de comparaciones ascendería a  $2(N/n_{\text{proc}}) - 1$ .

(e)  $T_s(N) = (N-1) * t$  (teniendo en cuenta sólo las comparaciones con  $x$ ; no se tienen en cuenta, por tanto, las comparaciones que controlan el final del bucle)



(f) Llamando  $p$  a  $n_{proc}$  y teniendo en cuenta sólo las comparaciones con  $x$  (no se tienen en cuenta, por tanto, las comparaciones que controlan el final del bucle):

$$T_p(p, N) = T_c(p, N) + T_{c/s}(N) = [ (N/p-1) * t + (p-1) * t ] + [ 10000 * t + N/p * 2 * t + 10000 * t + 2 * t ]$$

El tiempo de comunicación depende del tiempo que supone el reparto de  $v$  antes de hacer los cálculos (se envían  $N/p$  datos a cada procesador) más el tiempo que supone la transferencia de los máximos parciales calculados por los procesos (se envía un dato por cada proceso).

$$T_p(p, N) = \{ [(10^6/10-1) + 9] + [2*10^4 + 2*10^6/10 + 2] \} * t$$

$$S(p, N) = (10^6-1) * t / \{ [(10^6/10-1) + 9] + [2*10^4 + 2*10^6/10 + 2] \} * t$$

$$S(p, N) \sim 10^6 / (10^5 + 2*10^4 + 2*10^5)$$

$$S(p, N) \sim 100 / (10 + 2 + 20)$$

$$S(p, N) \sim 100 / 32 = 3,125$$



2º curso / 2º cuatr.

Grado en  
Ing. Informática

# Arquitectura de Computadores: Exámenes y Controles

## Examen de Prácticas 20/06/2012 resuelto

Material elaborado por los profesores responsables de la asignatura:  
Mancia Anguita, Julio Ortega

Licencia Creative Commons



### 1 Enunciado Examen de Prácticas del 20/06/2012

**Cuestión 1.**(1 punto) Considere el programa de la siguiente figura:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char **argv) {
    int i, n=20, tid, a[n], suma=0, sumalocal;
    if(argc < 2) {
        fprintf(stderr, "\nFalta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) n=20;

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

    #pragma omp parallel private(sumalocal,tid)
    { sumalocal=0;
      tid=omp_get_thread_num();
      #pragma omp for schedule(static)
      for (i=0; i<n; i++)
      { sumalocal += a[i];
        printf(" thread %d suma de a[%d]=%d sumalocal=%d \n", tid,i,a[i],sumalocal);
      }
      suma += sumalocal;
    }
    #pragma omp barrier
    #pragma omp master
    printf("thread master=%d imprime suma=%d\n", tid,suma);
}
```

- (a) ¿Permite calcular correctamente la suma de todos los elementos del array `a[]`? (Indique cómo solucionaría el problema en el caso de que lo hubiera).
- (b) ¿Qué se pretende al incluir `#pragma omp master` en este programa?
- (c) ¿Se puede prescindir de `#pragma omp barrier`? (Justifique su respuesta).
- (d) ¿Cuál es la utilidad de la clausula `schedule(static)` en la directiva `#pragma omp for schedule(static)`?



- (e) Qué diferencias habría en lo que imprime en pantalla el programa si se sustituyera `master` por `single` en el programa? Razone su respuesta. ¿Para qué sirve la función `omp_get_thread_num()`?

**Cuestión 2. (1 punto)** Considere el programa de la siguiente figura

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

main(int argc, char **argv) {
    int i, n = 7, chunk, a[n], suma=0;

    if(argc < 2) {
        fprintf(stderr, "\nFalta chunk \n");
        exit(-1);
    }
    chunk = atoi(argv[1]);

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

    #pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(static, chunk)
    for (i=0; i<n; i++)
    { suma = suma + a[i];
      printf(" thread %d suma a[%d] suma=%d \n",
            omp_get_thread_num(), i, suma);
    }

    printf("Fuera de 'parallel for' suma=%d\n", suma);
}
```

- (a) ¿Permite calcular la suma de los componentes del vector `a[]`? Justifique su respuesta.
- (b) ¿Qué imprime el código cada vez que se ejecuta la función `printf` del bucle?
- (c) ¿Qué imprime el código cuando se ejecuta la segunda función `printf`?
- (d) ¿Para qué sirve el parámetro `chunk` en la construcción `#pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(static, chunk)`?
- (e) ¿Para qué sirven las clausulas `firstprivate(suma)` y `lastprivate(suma)`?
- (f) ¿qué imprime el programa si se eliminan `firstprivate(suma)` y `lastprivate(suma)` y se incluye `reduction(+:suma)` en la construcción `#pragma omp parallel for`?



## 2 Solución Examen de Prácticas del 20/06/2012

**Cuestión 1.**(1 punto) Considere el programa de la siguiente figura:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char **argv) {
    int i, n=20, tid, a[n], suma=0, sumalocal;
    if(argc < 2) {
        fprintf(stderr, "\nFalta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) n=20;

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

    #pragma omp parallel private(sumalocal,tid)
    { sumalocal=0;
      tid=omp_get_thread_num();
      #pragma omp for schedule(static)
      for (i=0; i<n; i++)
      { sumalocal += a[i];
        printf(" thread %d suma de a[%d]=%d sumalocal=%d \n", tid,i,a[i],sumalocal);
      }
      suma += sumalocal;
    }
    #pragma omp barrier
    #pragma omp master
    printf("thread master=%d imprime suma=%d\n", tid,suma);
}
```

- (a) ¿Permite calcular correctamente la suma de todos los elementos del array `a[]`? (Indique cómo solucionaría el problema en el caso de que lo hubiera).
- (b) ¿Qué se pretende al incluir `#pragma omp master` en este programa?
- (c) ¿Se puede prescindir de `#pragma omp barrier`? (Justifique su respuesta).
- (d) ¿Cuál es la utilidad de la clausula `schedule(static)` en la directiva `#pragma omp for schedule(static)`?
- (e) Qué diferencias habría en lo que imprime en pantalla el programa si se sustituyera `master` por `single` en el programa? Razone su respuesta. ¿Para qué sirve la función `omp_get_thread_num()`?

### Solución

(a) No, porque los threads acceden a la variable compartida `suma` en paralelo y todos leen, modifican y escriben en la variable (R-M-W). Tendrían que acceder un thread detrás de otro, es decir, los threads tendrían que acceder secuencialmente a realizar la operación `suma+=sumalocal` (R-M-W) para evitar que más de un thread puedan leer el mismo valor de `suma`. Para resolver el problema se podría, por ejemplo, utilizar la directiva `atomic` o la directiva `critical` (esta última es menos eficiente):

**atomic**

```
#pragma omp atomic
suma += sumalocal;
```

**critical**

```
#pragma omp critical
suma += sumalocal;
```

**(b)** Se pretende que el thread 0 (el master) ejecute la función `printf` que hay justo después de esta directiva (en el bloque estructurado de la directiva). Esta función imprime `tid`, que es el identificador del thread (0 en este caso) que ejecuta `printf` y el contenido de la variable compartida `suma`. Como se ha reflexionado previamente, `suma` puede que no contenga la suma de todos los componentes del vector debido al acceso sin exclusión mutua que realizan los threads previamente en el código.

**(c)** No. Con esta directiva los threads se esperan en el punto del código donde se encuentra, cuando todos han llegado a ese punto, continúan la ejecución. Es necesario mantener esta barrera para que el thread 0 imprima el contenido de la variable `suma` cuando todos los threads han acumulado en esta variable el resultado parcial de suma que han almacenado en su variable local `sumalocal`. Si se elimina, el thread 0 podría imprimir la suma parcial que él mismo ha calculado o la suma de los valores calculados por alguno de los threads incluido el mismo, no habría, por tanto, garantía de haber acumulado en `suma` todas las sumas parciales calculadas por los threads.

**(d)** Esta cláusula fija qué tipo de asignación de iteraciones del bucle (planificación) se va a realizar. En este caso fija que se realice una planificación por parte del compilador (estática). Como no se especifica el tamaño de los trozos (es decir, el número de iteraciones consecutivas del bucle) que se van a asignar a los threads en turno rotatorio, se tomará el que fije por defecto la implementación particular de OpenMP que se use.

**(e)** Como se ha comentado en **(b)** la función `printf` imprime `tid`, que contiene el identificador del thread que ejecuta el `printf`. Si se usa `single`, la función `printf` la ejecutará el thread que llegue en primer lugar a ese punto del código, podría ser el 0 o cualquier otro. Por tanto, el valor que se imprime como `tid` ahora, usando `single`, puede ser 0, 1,... `nthread-1`, donde `nthread` es el identificador del thread que ejecuta el código. La variable `tid` contiene el identificador del thread que ejecuta el código porque se inicializa con el valor que devuelve la función de la biblioteca OpenMP `omp_get_thread_num()`.



**Cuestión 2.**(1 punto) Considere el programa de la siguiente figura

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

main(int argc, char **argv) {
    int i, n = 7, chunk, a[n], suma=0;

    if(argc < 2) {
        fprintf(stderr, "\nFalta chunk \n");
        exit(-1);
    }
    chunk = atoi(argv[1]);

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

    #pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(static, chunk)
    for (i=0; i<n; i++)
    { suma = suma + a[i];
      printf(" thread %d suma a[%d] suma=%d \n",
             omp_get_thread_num(), i, suma);
    }

    printf("Fuera de 'parallel for' suma=%d\n", suma);
}
```

- (a) ¿Permite calcular la suma de los componentes del vector `a[]`? Justifique su respuesta.
- (b) ¿Qué imprime el código cada vez que se ejecuta la función `printf` del bucle?
- (c) ¿Qué imprime el código cuando se ejecuta la segunda función `printf`?
- (d) ¿Para qué sirve el parámetro `chunk` en la construcción `#pragma omp parallel for firstprivate(suma) lastprivate(suma) schedule(static, chunk)`?
- (e) ¿Para qué sirven las cláusulas `firstprivate(suma)` y `lastprivate(suma)`?
- (f) ¿qué imprime el programa si se eliminan `firstprivate(suma)` y `lastprivate(suma)` y se incluye `reduction(+:suma)` en la construcción `#pragma omp parallel for`?

### Solución

(a) No. La cláusula `firstprivate(suma)` fuerza a que haya una variable `suma` privada a cada thread y a que esta variable se inicialice al valor que tiene la variable compartida `suma` declarada en el thread master. Por tanto, el valor de la variable compartida `suma` declarada en el thread master se copia a todas las variables privadas `suma` de los threads que ejecutan la región paralela. En ningún momento en el código se suman las sumas parciales obtenidas por los threads en sus respectivas variables privadas `suma` con el fin de obtener la suma total.



(b) El `printf` que se ejecuta cada iteración del bucle imprime el identificador del thread que ejecuta la iteración del bucle (porque es el valor que devuelve `omp_get_thread_num()`), la iteración que se ejecuta (valor de `i`), y el valor de la variable privada del thread `suma` en esa iteración. Los threads imprimen en pantalla el valor de su variable privada `suma` cada vez que añade un nuevo componente del vector a dicha variable.

(c) Imprime el contenido de la variable compartida `suma` que se ha obtenido en la última iteración del bucle, independientemente de cual sea el thread que ha ejecutado esa iteración.

(d) Se utiliza `chunk` para fijar el número de iteraciones consecutivas del bucle que va a contener los trozos de código que se van a usar como unidades de asignación a los threads. Estos trozos se asignan por turno rotatorio. Si, por ejemplo, `chunk` fuese 1, entonces la unidad de asignación sería una iteración y se asignaría entonces al thread 0, las iteraciones 0, `nthreads`, `2nthreads`, ..., al thread 1 las iteraciones 1, `nthreads+1`, `2nthreads+1`, ..., y así sucesivamente. Si, por ejemplo, fuese 2, entonces se asignaría al thread `i` las iteraciones, `i, i+1, 2nthreads+2i, 2nthreads+2i+1, 4nthreads+2i, 4nthreads+2i+1, ...`

(e) La cláusula `firstprivate(suma)` fuerza a que haya una variable `suma` privada a cada thread y a que esta variable se inicialice al valor que tiene la variable compartida `suma` declarada en el thread master. Por tanto, el valor de la variable compartida `suma` declarada en el thread master se copia a todas las variables privadas `suma` de los threads que ejecutan la región paralela.

La cláusula `lastprivate(suma)` fuerza a que haya una variable `suma` privada a cada thread (lo fuerza también `firstprivate`) y a que al valor que tiene la variable privada `suma` en la última iteración del bucle se copie a la variable compartida `suma` declarada en el thread master. En este caso se devuelve el valor de la variable `suma` del thread que ejecute la iteración `n-1=6`.

(e) La cláusula `reduction(+:suma)` fuerza a que haya una variable `suma` privada a cada thread inicializada a 0 y a que los threads, una vez ejecutadas las iteraciones que tienen asignados, sumen (por usar "+") en exclusión mutua el contenido de su variable privada `suma` a la variable compartida `suma` declarada en el thread master. Por tanto, si se sustituyen `firstprivate(suma)` y `lastprivate(suma)` por `reduction(+:suma)`, la suma se va a calcular correctamente y el programa imprimirá en el segundo `printf` la suma de todos los componentes del vector. El vector de 7 componentes se ha inicializado a 0, 1, 2, 3, 4, 5, 6 (`a[i]=i`); por tanto, se imprimirá 21.

