2º curso / 2º cuatr.

**Grado en Ing. Informática**

**Doble Grado en Ing. Informática y Matemáticas**

# Arquitectura de Computadores: Exámenes y Controles

## Examen Final AC 05/07/2013 resuelto

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



## 1 Enunciado Examen del 05/07/2013

**Cuestión 1.**(0.5 puntos) Diferencias entre un core de procesamiento con multithread temporal y un core de procesamiento con multihread simultáneo.

**Cuestión 2.** (0.5 puntos) Diferencias entre un core de procesamiento VLIW y un core de procesamiento superescalar.

**Ejercicio 1.** (1 punto) La empresa Hardahí, dedicada al desarrollo de microprocesadores (cores o núcleos de procesamiento), ha recibido dos propuestas de proyectos viables de sus equipos de ingenieros, pero sólo tiene presupuesto para invertir en uno de ellos. El primero tiene un coste de un millón de euros y se basa en la mejora de la microarquitectura para reducir los CPI de las instrucciones a la mitad, mientras que el segundo consiste en la aplicación de nuevas técnicas de optimización de código para conseguir que el compilador que genera código ejecutable para su familia de microprocesadores reduzca, en el conjunto de programas de prueba que utiliza la empresa, el número de instrucciones LOAD un 10%, de STORE un 10 % y de BRANCH un 10%. Este último proyecto tiene un coste de seiscientos mil euros. ¿Qué proyecto debería seleccionar por ser más rentable (por tener mejor relación entre mejora de prestaciones y euros invertidos)?
**NOTA:** La familia de procesadores de esta empresa tiene una arquitectura de tipo LOAD/STORE en las que las operaciones sólo utilizan como operandos registros de la CPU. El conjunto de programas de prueba tiene que el 30% de las instrucciones son operaciones con la ALU (2 CPI), el 20% son operaciones con números en coma flotante (8 CPI), el 20% LOADs (4 CPI), el 10% STOREs (4 CPI) y el 20% BRANCHs (4 CPI).

**Ejercicio 2.** (1.75 puntos) Se dispone del código siguiente para calcular, en la variable `sum` ,la suma de los `N` componentes del vector `a[]` en un multiprocesador usando el estilo de programación de variables compartidas (todos los threads ejecutarán el mismo código):

```
sump = 0;
for (i=ithread  ; i<N  ; i=i+nthread) {
     sump = sump + a[i];
}
while (Fetch_&_Or(k,1)==1)  {};
sum = sum + sump;
k=0;
```

Conteste a las siguientes preguntas:

**(a)** (0.25 puntos) ¿Qué variables deberían ser necesariamente variables privadas a un thread? Razone su respuesta, indique, para cada una de ellas, qué consecuencias habría si fuese compartida.

**(b)** (0.25 puntos) ¿Qué variables deberían ser necesariamente variables compartidas por todos los threads? Razone su respuesta, indique, para cada una de ellas, qué consecuencias habría si fuese privada.

**(c)** (0.25 puntos)

**(d)** (1 punto)

**Ejercicio 3.** (1 puntos)

**Ejercicio 4.** (1.25 puntos)


**(a)** (0.1 puntos)

**(b)** (0.15 puntos)

**(c)** (0.25 puntos)

**(d)** (0.25 puntos)

**(e)** (0.5 puntos)

# 2 Solución Examen del 05/07/2013

**Cuestión 1.**(0.5 puntos) Diferencias entre un core de procesamiento con multithread temporal y un core de procesamiento con multihread simultáneo.

**Solución**

Un núcleo con multithread simultánea es un núcleo superescalar que puede emitir a unidades funcionales instrucciones de múltiples threads distintos por lo que puede ejecutar instrucciones de distintos threads en paralelo; es decir, ejecutar distintos threads en paralelo. Para conseguir emitir instrucciones de distintos threads en paralelo procesa los threads en paralelo en las distintas etapas del cauce superescalar, para ello las etapas del cauce se comparten por los threads o se reparten entre los threads, los registros de la arquitectura se replican (hay un conjunto propio de cada thread que es capaz de ejecutar en paralelo el núcleo) y el almacenamiento entre etapas se comparte por los threads, se reparte entre ellos o se replica. Mientras que un núcleo con multithread temporal es un núcleo segmentado, superescalar o VLIW que procesa varios threads concurrentemente, pero no en paralelo (sólo emite instrucciones de un thread a unidades funcionales), es decir, ejecuta los threads multiplexando en el tiempo el uso de las etapas del cauce por los distintos threads. Para la multiplexación tiene hardware que se encarga de conmutar entre threads. Las etapas del cauce se multiplexan (se asignan temporalmente a un thread), excepto la etapa de captación que se reparte o comparte por los thread, los registros de la arquitectura se replican (hay un conjunto propio de cada thread que es capaz de ejecutar en paralelo el núcleo) y el almacenamiento entre etapas se multiplexa (se asignan temporalmente a un thread), se comparte por los threads, se reparte entre ellos o se replica (hay un conjunto propio de cada thread que es capaz de ejecutar en paralelo el núcleo).

◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀

**Cuestión 2.**(0.5 puntos). Diferencias entre un core de procesamiento VLIW y un core de procesamiento superescalar.

**Solución**

Tanto un núcleo superescalar como uno VLIW aprovechan el paralelismo a nivel de instrucción, ambos ejecutan instrucciones concurrentemente por su arquitectura segmentada, y en paralelo, por su arquitectura con múltiples unidades funcionales. Ambos son capaces además de emitir más de una instrucción a la vez a las unidades funcionales. La diferencia principal entre ellos está en que los procesadores superescalares disponen de hardware para extraer paralelismo en las instrucciones que se captan de memoria mientras que en los procesadores VLIW este paralelismo lo extrae el compilador. Es decir, en los procesadores superescalares se incluye hardware que se encarga de planificar, en tiempo de ejecución, la ejecución de las instrucciones que se captan de memoria (planificación dinámica) mientras que en los procesadores VLIW esta planificación la realiza el compilador (planificación estática). Las instrucciones que se van a ejecutar en paralelo se captan juntas de memoria en los procesadores VLIW. Estas instrucciones que se captan juntas no presentan dependencia de datos (RAW, WAW, WAR), ni de control (debida a la presencia de saltos) y se ejecutan en distinta unidad funcional, de todo ello se ha asegurado el compilador. En el caso de los procesadores superespalares se incluye hardware que elimina las dependencias de datos WAR y WAW (buffer de renombrado, en caso de utilizar buffer de reorden se usa como buffer de renombrado), hardware de emisión que evita las dependencias RAW (ventana de emisión, estaciones de reserva, buffer de reorden), y hardware de predicción de saltos, que reduce la penalización que supone la ejecución de los saltos. De lo comentado se deduce que los cores VLIW tienen un hardware más sencillo, lo que les hace ser más pequeños y consumir menos energía.

◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀◀

**Ejercicio 1.** (1 punto) La empresa Hardahí, dedicada al desarrollo de microprocesadores (cores o núcleos de procesamiento), ha recibido dos propuestas de proyectos viables de sus equipos de ingenieros, pero sólo tiene presupuesto para invertir en uno de ellos. El primero tiene un coste de un millón de euros y se basa en

la mejora de la microarquitectura para reducir los CPI de las instrucciones a la mitad, mientras que el segundo consiste en la aplicación de nuevas técnicas de optimización de código para conseguir que el compilador que genera código ejecutable para su familia de microprocesadores reduzca, en el conjunto de programas de prueba que utiliza la empresa, el número de instrucciones LOAD un 10%, de STORE un 10 % y de BRANCH un 10%. Este último proyecto tiene un coste de seiscientos mil euros. ¿Qué proyecto debería seleccionar por ser más rentable (por tener mejor relación entre mejora de prestaciones y euros invertidos)?
**NOTA:** La familia de procesadores de esta empresa tiene una arquitectura de tipo LOAD/STORE en las que las operaciones sólo utilizan como operandos registros de la CPU. El conjunto de programas de prueba tiene que el 30% de las instrucciones son operaciones con la ALU (2 CPI), el 20% son operaciones con números en coma flotante (8 CPI), el 20% LOADs (4 CPI), el 10% STOREs (4 CPI) y el 20% BRANCHs (4 CPI).

## Solución

Datos del ejercicio:

Alternativa inicial:

| $I_i$ | $CPI_i$ | $NI_i$ | Comentarios |
|---|---|---|---|
| **LOAD** | 4 | 0,2×NI | |
| **STORE** | 4 | 0,1×NI | |
| **ALU** | 2 | 0,3×NI | |
| **FP** | 8 | 0,2×NI | |
| **BR** | 4 | 0,2×NI | |
| **TOTAL** | | $NI^1$ | |

Alternativa 1 (coste 1000000€)

| $I_i$ | $CPI_i^1$ | $NI_i^1$ | Comentarios |
|---|---|---|---|
| **LOAD** | 2 | 0,2×NI | |
| **STORE** | 2 | 0,1×NI | |
| **ALU** | 1 | 0,3×NI | |
| **FP** | 4 | 0,2×NI | |
| **BR** | 2 | 0,2×NI | |
| **TOTAL** | | $NI=NI^1$ | |

Alternativa 2 (coste 600000€)

| $I_i$ | $CPI_i^2$ | $NI_i^2$ | Comentarios |
|---|---|---|---|
| **LOAD** | 4 | $(0,2-0,1\times0,2)\times NI^1=0,18\times NI^1$ | Se reducen las instrucciones LOAD en un 10% |
| **STORE** | 4 | $(0,1-0,1\times0,1)\times NI^1=0,09\times NI^1$ | Se reducen las instrucciones STORE en un 10% |
| **ALU** | 2 | $0,3\times NI^1$ | |
| **FP** | 8 | $0,2\times NI^1$ | |
| **BR** | 4 | $(0,2-0,1\times0,2)\times NI^1=0,18\times NI^1$ | Se reducen las instrucciones BR en un 10% |
| **TOTAL** | | $\mathbf{NI^2=0,95\times NI^1}$ | **Es decir, $NI^2$ es igual a $0,95\times NI^1$** |

Solución 1:

Vamos a llamar $T_{CPU}$ al tiempo de CPU que requiere la alternativa de partida y CPI a los ciclos por instrucción que se consiguen con esta alternativa.

En primer lugar se obtiene el tiempo de CPU para la alternativa de mejora 1:

$$T^1_{CPU} = CPI^1\times NI\times T_{ciclo} = (0.2\times2+0.1\times2+0.2\times2+0.3\times1+0.2\times4)\times NI\times T_{ciclo} = (1+0.3+0.8)\times NI^1\times T_{ciclo} = 2.1\times NI\times T_{ciclo}$$

Ganancia prestaciones= $T_{CPU}/T^1_{CPU}= CPI\times NI\times T_{ciclo}/2.1\times NI\times T_{ciclo}=CPI/2.1$ (CPI: ciclos por instrucción inicial)

Relación ganancia prestaciones/coste = CPI/2100000

En segundo lugar se calcula el tiempo de CPU para la alternativa de mejora 2:

$$T^2_{CPU} = CPI^2 \times NI^2 \times T_{ciclo} = (0.18\times4+0.09\times4+0.18\times4+0.3\times2+0.2\times8) \times NI^1 \times T_{ciclo} = (0.45\times4+ 0.6 + 1.6) \times NI \times T_{ciclo} = (1.8+ 0.6 + 1.6) \times NI \times T_{ciclo} = 4 \times NI \times T_{ciclo}$$

Ganancia prestaciones= $T_{CPU}/ T^2_{CPU}= CPI\times NI\times T_{ciclo} / 4\times NI\times T_{ciclo}=CPI/4$

Relación ganancia prestaciones/coste = CPI/2400000

Es mejor la primera opción, ya que presenta una mayor relación ganancia en prestaciones por euro invertido por tener el resultado una denominador menor. Como se puede observar, en realidad no hace falta calcular el tiempo de CPU de la alternativa original para contestar.

Solución 2:

Vamos a llamar $T_{CPU}$ al tiempo de CPU que requiere la alternativa de partida y CPI a los ciclos por instrucción que se consiguen con esta alternativa:

$$T_{CPU} = CPI\times NI\times T_{ciclo} = (0.2\times4+0.1\times4+0.2\times4+0.3\times2+0.2\times8) \times NI \times T_{ciclo} = (2+ 0.6 + 1.6) \times NI^1 \times T_{ciclo} = 4.2\times NI\times T_{ciclo}$$

En primer lugar se calcula el tiempo de CPU para la alternativa de mejora 1:

$$T^1_{CPU} = CPI^1\times NI\times T_{ciclo} = (0.2\times2+0.1\times2+0.2\times2+0.3\times1+0.2\times4) \times NI \times T_{ciclo} = (1 + 0.3 + 0.8) \times NI^1 \times T_{ciclo} = 2.1\times NI\times T_{ciclo}$$

Ganancia prestaciones= $T_{CPU}/T^1_{CPU}= 4.2\times NI\times T_{ciclo}/2.1\times NI\times T_{ciclo}=2$

Relación ganancia prestaciones/coste = 2/1000000 = 0.000002

En segundo lugar se calcula el tiempo de CPU para la alternativa de mejora 2:

$$T^2_{CPU} = CPI^2 \times NI^2 \times T_{ciclo} = (0.18\times4+0.09\times4+0.18\times4+0.3\times2+0.2\times8) \times NI^1 \times T_{ciclo} = (0.45\times4+ 0.6 + 1.6) \times NI \times T_{ciclo} = (1.8+ 0.6 + 1.6) \times NI \times T_{ciclo} = 4 \times NI \times T_{ciclo}$$

Ganancia prestaciones= $T_{CPU}/ T^2_{CPU}= 4.2\times NI\times T_{ciclo} / 4\times NI\times T_{ciclo}=1.05$

Relación ganancia prestaciones/coste = 1.05/600000 = 0.00000175

Es mejor la primera opción, ya que presenta una mayor relación ganancia en prestaciones por euro invertido 0.000002 frente a 0.00000175.

**Ejercicio 2.** (1.75 puntos) Se dispone del código siguiente para calcular, en la variable `sum` ,la suma de los `N` componentes del vector `a[]` en un multiprocesador usando el estilo de programación de variables compartidas (todos los threads ejecutarán el mismo código):

```
sump = 0;
for (i=ithread  ; i<N  ; i=i+nthread) {
     sump = sump + a[i];
}
while (Fetch_&_Or(k,1)==1)  {};
sum = sum + sump;
k=0;
```

Conteste a las siguientes preguntas:

**(e)** (0.25 puntos) ¿Qué variables deberían ser necesariamente variables privadas a un thread? Razone su respuesta, indique, para cada una de ellas, qué consecuencias habría si fuese compartida.

**(f)** (0.25 puntos) ¿Qué variables deberían ser necesariamente variables compartidas por todos los threads? Razone su respuesta, indique, para cada una de ellas, qué consecuencias habría si fuese privada.

**(g)** (0.25 puntos) ¿Qué variables podrían ser indistintamente privadas o compartidas? Razone su respuesta. ¿Qué diferencia(s) habría?

**(h)** (1 punto) ¿Para qué modelos de consistencia de memoria se puede asegurar que se obtendría correctamente en la variable `sum` la suma de los `N` componentes del vector `a[]` en cualquier ejecución del código y para qué modelos de consistencia no se puede asegurar un resultado correcto en todas las ejecuciones? Razone su respuesta.

**Solución**

**(a)** Las variables `i`, `ithread` y `sump` deben ser privadas. La primera porque cada thread debe ejecutar su propio bucle, la segunda porque cada thread debe tener en `ithread` su identificador dentro del grupo de thread que está colaborando en la ejecución paralela, y la tercera porque deben acumular cada thread en una variable distinta la suma de los componentes del vector que tiene asignados sumar. Cada thread, en definitiva, tiene que sumar un subconjunto distinto de los componentes del vector, y lo deben hacer con su propio bucle y en su propia variable acumulador.

Si `i` fuera compartida (y las demás variables tuvieran el ámbito esperado) todos los threads usarían la misma variable como índice del bucle, leyendo y escribiendo en ella. Como resultado varios threads podrían sumar el mismo componente del vector `a[]` (porque podrían ejecutar la misma iteración del bucle), podrían no sumarse todos los componentes del vector (se podría saltar alguna iteración del bucle) e incluso se podrían acceder a posiciones de memoria no reservadas al leer el vector (porque algún thread puede encontrar un `i<n` y al acceder a `a[]` podría encontrarse un `i>=N`, como resulado del último incremento de `i` de algún otro thread).

Si `ithread` fuese compartida (y las demás variables tuvieran el ámbito esperado) todos los threads al final calcularían el mismo valor en `sump` porque todos sumarían los mismos componentes del vector `a[]`.

Si `sump` fuera compartida (y las demás variables tuvieran el ámbito esperado) todos los threads al final tendrían el mismo valor en `sump` y podría ocurrir que no se sumaran todos los componentes del vector en `sump`, ya que varios threads podrían leer el mismo valor de `sump` en alguna de las iteraciones del bucle. Además al final se obtendría en `sum` el valor resultante en `sump` multiplicado por el número de threads.

**(b)** Las variables `sum` y `k` deben ser compartidas. La primera porque es la variable en la que se pretende obtener la suma final de los componentes del vector, y la segunda porque es una variable cerrojo.

Si `sum` fuese privada (y las demás variables tuvieran el ámbito esperado) todos los threads obtendrían en su variable privada `sum` el valor que han calculado en `sump`, no se obtendría la suma de todos los `sump` en ninguna variable `sum`.

Si `k` fuera privada (y las demás variables tuvieran el ámbito esperado) no habría sincronización en el acceso a la variable compartida `sum`. Suponiendo que la `k` privada pudiera valer 1 en algunos threads y 0 en otros, en los threads en los que su `k` fuese 0 entrarían a modificar `sum` todos a la vez (no uno detrás de otro, secuencialmente) y en los que tuviera valor de uno se quedarían indefinidamente bloqueados sin terminar en el bucle que lee y escribe 1 en `k` (porque nunca podrían leer un 0 en `k`, ya que ningún otro thread puede acceder a su variable `k` al ser privada)

**(c)** Las variables `N` (número de componentes del vector a sumar), `nthread` (número total de threads) y `a[]` (vector cuyos componentes se van a sumar) pueden ser compartidas o privadas. Ninguna de ellas va a modificar su contenido durante la ejecución, por lo que pueden ser compartidas. En el caso del vector además cada thread va a leer distintos componentes del vector.

Si se hacen privadas, supondrían ocupar más almacenamiento en memoria innecesario y hacer copias innecesarias para cada threads, lo que añadiría tiempo extra de ejecución. En el caso del vector `a[]`,si al acceder a un nuevo componente del mismo se produce un fallo en el último nivel de cache compartido entre cores, el bloque de memoria trasferido a la cache contendría datos que no va a usar ningún core (thread) de los que comparten la cache. Esto ocurriría para todos los fallos de cache de todos los threads que comparten la cache. Si se hacen estas variables compartidas hay un ahorro del almacenamiento en memoria como se ha mencionado. En el caso del vector `a[]`, además permitiría que, si los threads se ejecutan en cores que comparten el último nivel de cache, los fallos de cache disminuyan, porque al traer un bloque de memoria (cuando se produce un fallo) se traen datos que usan distintos threads (en un bloque hay componentes del vector que usan distintos threads).

(d) En la siguiente tabla se puede ver la secuencia de lecturas y escrituras en memoria de variables compartidas en el código:

| Accesos a variables compartidas | Código |
|---|---|
| | `sump = 0;`<br>`for (i=ithread ; i<N ; i=i+nthread) {`<br>`  sump = sump + a[i];`<br>`}` |
| RW(k) | `while (Fetch & Or(k,1)==1) {};` |
| R(sum), W(sum) | `sum = sum + sump;` |
| W(k) | `k=0;` |

Para que el código obtenga la suma de forma correcta en la variable compartida `sum` se debe cumplir lo siguiente:

(1). Que la lectura de la variable `sum`, `R(sum)`, y la escritura, `W(sum)`, no adelanten a la operación atómica de lectura y escritura de `k`, `RW(k)`. De esta forma no será posible que más de un flujo de control acceda a la vez a la variable compartida `sum`; no habrá posibilidad, por ejemplo, de que dos flujos de control (threads) lean el mismo valor de `sum` y, por tanto, no podrán ver varios un 0 en `sum`.

(2). Que la escritura del cerrojo `k`, `W(k)`, no adelante a los accesos a memoria de la variable `sum` anteriores en el código, `R(sum)`, `W(sum)`. De esta forma no será posible que el flujo de control que está accediendo a `sum` abra el cerrojo (escriba 0 en `k`) permitiendo que otro flujo de control acceda a `sum`, antes de que termine su acceso a la variable compartida `sum`.

Para un modelo de consistencia secuencial o para un modelo de consistencia que relaje el orden W->R se obtendría siempre en `sum` la suma de los componentes del vector `a[]`, porque en ambos casos se cumple lo indicado en los dos puntos anteriores. El modelo de consistencia secuencial garantiza siempre todos los órdenes, es decir, garantiza que todos los accesos a memoria de un flujo de control se llevan a cabo en el orden en el que se encuentran esos accesos en el código que ejecuta el flujo de control. En el caso del modelo que relaja el orden W->R, esta relajación no impide que en el código anterior todos los accesos se realicen ordenadamente porque la única lectura que podría adelantar a una escritura, `R(sum)`, no podrá adelantarla porque la escritura anterior es una operación atómica que también incluye lectura, `RW(k)`, y una lectura no puede adelantar a una lectura anterior (no se relaja R->R, sólo W->R).

Para (1) los modelos que relajan W->R y W->W y para (2) los que relajan todos los órdenes, como el modelo de ordenación débil y el de consistencia de liberación, el resultado de la suma no siempre se obtendrá correctamente en todas las ejecuciones debido a que relajan W->W y a que, en el segundo caso (modelo de ordenación débil y consistencia de liberación), además relajan R->R. Al no asegurar el orden W->W, la escritura de `k` de 0, `W(k)`, o apertura del cerrojo puede producirse antes de la escritura de `sum`, `W(sum)`, por parte de un flujo de control (A). Al abrir el cerrojo otro flujo de control (B) puede leer la variable compartida, `R(sum)`, e incluso modificarla, `W(sum)`, antes de que (A) la haya modificado. Como consecuencia el valor de `sump` calculado por (A) no se observaría en el resultado `sum`. Además, en

**Ejercicio 3.** (1 puntos)

**Solución**

| | | |
|---|---|---|
| | | lw r1,(a) ;se carga en r1 lo que hay en la dirección a |
| | p1, p2 | cmp.eq r1,0 |
| (p1) | | lw r2,(b) ;se carga en r2 lo que hay en la dirección b |
| (p1) | | add r3,r1,r2 |
| (p2) | | add r3,r1,#1 |
| | | sw (c),r3 ;se guarda r3 en la dirección c |
| | | lw r1,4(a) ;se carga en r1 lo que hay en a+4 |
| | p1, p2 | cmp.eq r1,0 |
| (p1) | | lw r2,4(b) ;se carga en r2 lo que hay en b+4 |
| (p1) | | add r3,r1,r2 |
| (p2) | | add r3,r1,#1 |
| | | sw 4(c),r3 ;se guarda r3 en la dirección c+4 |

**Ejercicio 4.**

**(f)** (0.1 puntos) ¿Qué tiempo tarda en procesarse una imagen de **m** bloques de 8x8 píxeles si se implementa para la aplicación un código secuencial?

**(g)** (0.15 puntos) ¿Cuál sería el máximo *grado de paralelismo* para una implementación paralela teniendo en cuenta el grafo de tareas? Razone su respuesta.

**(h)** (0.25 puntos) ¿Qué directivas OpenMP utilizaría para conseguir, usando sólo directivas, que las cuatro tareas, T1, T2, T3 y T4, se puedan ejecutar en paralelo? ¿Para qué usaría cada una de ellas? Razone sus respuestas.

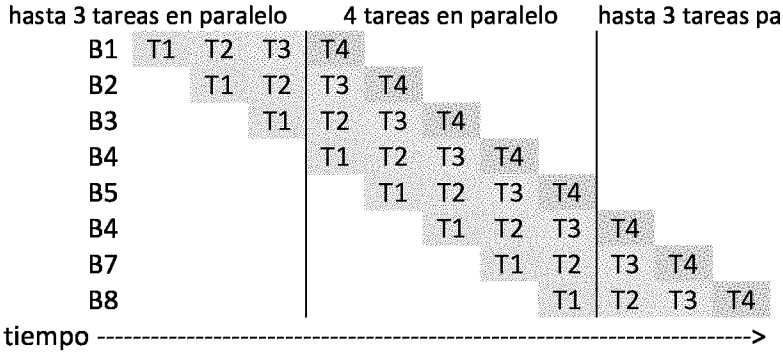
**(i)** (0.25 puntos) ¿Qué tiempo tarda en procesarse una imagen de **m** bloques de 8x8 píxeles en un código paralelo teniendo en cuenta el grafo de tareas?

**(j)** (0.5 puntos) ¿Qué ganancia en prestaciones y qué eficiencia se podría llegar a conseguir como máximo en una implementación paralela teniendo en cuenta el grafo de tareas?

**Solución**

**(a)** El tiempo de ejecución secuencial, $T_S$, de la aplicación suponiendo que se procesa una imagen de m bloques de 8x8 píxeles sería: $T_S = (80ms+40ms+80ms+20ms) \times m = 220ms \times m$; es decir, el tiempo que supone el procesamiento de un bloque, 220ms, multiplicado por el número de bloques a procesar.

**(b)** El *grado de paralelismo* es el número máximo de tareas que se pueden ejecutar en paralelo. En este caso se podrían estar procesando en un momento dado hasta 4 bloques de 8x8 píxeles en paralelo, cada uno de ellos estaría en una etapa distinta del cauce segmentado, es decir, a cada uno de ellos se le estaría aplicando una tarea distinta. El grado de paralelismo sería entonces de 4, ya que se pueden ejecutar las 4 tareas a la vez, cada una de ellas procesando un bloque distinto. En consecuencia, el número máximo de procesadores/cores que se podrían aprovechar sería 4 y, por tanto, como máximo usaríamos 4 threads. Ejemplo de secuencia de ejecución para 8 bloques (B1 a B8) en la que se puede ver que, como máximo, puede haber 4 bloques ejecutándose en paralelo y que pueden estar ejecutándose las 4 tareas en paralelo:




En cualquier caso, dados los tiempos de ejecución las tareas T2 y T4 se podrían asignar al mismo procesador. Si se asignan al mismo procesador se podría usar un grado de paralelismo de 3.

**(c)** Usaríamos `parallel` para crear y terminar los 4 threads que se pueden ejecutar en paralelo, y las directivas `senctions` y `section` para distribuir las tareas entre los threads, cada section ejecutaría una función/tarea distinta, la barrera implícita de `senctions` se usaría para la sincronización entre threads. En uno de los `section` se ejecutaría `DescodificadorEntropia()` (T1), en un segundo `section` `CuantificacionInversa()` (T2), en un tercero pondríamos `TrasformadaCosenoInversa()` (T3) y, en un cuarto, `ConversionRGB()`. El código podría tener la siguiente estructura (los puntos suspensivos indican que puede haber de otro código en ese punto):

```
#include <omp.h>
...
#pragma omp parallel
{ ...
  for (i=0;i<m;i++) {
    ...
    #pragma omp sections
    {
     #pragma omp section
        {… DescodificadorEntropia(); …}
     #pragma omp section
        {… CuantificacionInversa(); …}
     #pragma omp section
        {… TrasformadaCosenoInversa(); …}
     #pragma omp section
        {… ConversionRGB(); …}
    }
    ...
  }
...
} ...
```

**(d)**

**(e)**