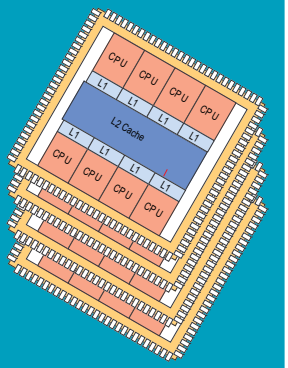
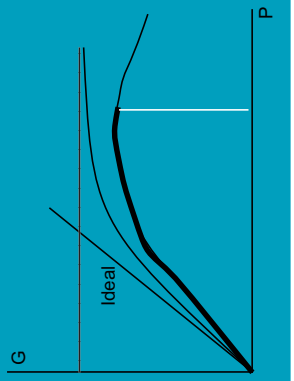
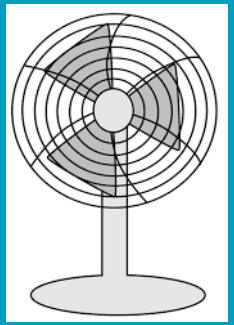






Multiprocesadores

$$P = C_e \cdot v^2 \cdot f + v \cdot I_f$$

J.M. Llabería

Contenido

Capítulo 2
Sistema multiprocesador

La necesidad de incrementar el rendimiento de un sistema de cómputo es siempre un horizonte en arquitectura de computadores. El objetivo es reducir el tiempo de ejecución de un programa o incrementar la productividad.

Una técnica ampliamente utilizada para mejorar la productividad es el paralelismo. En este contexto se implementa utilizando varios procesadores que ejecutan de forma autónoma instrucciones, pero con mecanismos que permiten comunicar información entre ellos y sincronizar esa comunicación. Estos sistemas de cómputo se denominan multiprocesadores.

Por otro lado, restricciones tecnológicas favorecen la utilización de multiprocesadores, ya que dedicar más recursos en mejorar el rendimiento de un procesador no representa un incremento sustancial del mismo y empeora otras métricas como el consumo de energía, necesidades de disipación térmica y relaciones entre rendimiento y estas últimas métricas.

Factores tecnológicos favorables por un lado, como la escala de integración y la necesidad de mayor capacidad de procesado por otro lado, teniendo en cuenta las restricciones tecnológicas, han determinado la introducción masiva en el mercado de chips multiprocesador.

Ejemplos de paralelismo son el paralelismo de datos y el paralelismo de función (Figura 2.1). En el primero de ellos varios procesadores efectúan la misma tarea sobre conjuntos de datos disjuntos. Un ejemplo del paralelismo de datos es un bucle cuyas iteraciones son independientes entre si. Cada iteración se puede estar calculando en un procesador. Otro ejemplo son búsquedas independientes en una base de datos. Un ejemplo de paralelismo de función es el procesado de imagen. En el procesado se distinguen varias fases que se encadenan, siendo cada fase una función. Los datos de entrada de una fase son los datos de salida de la fase previa.

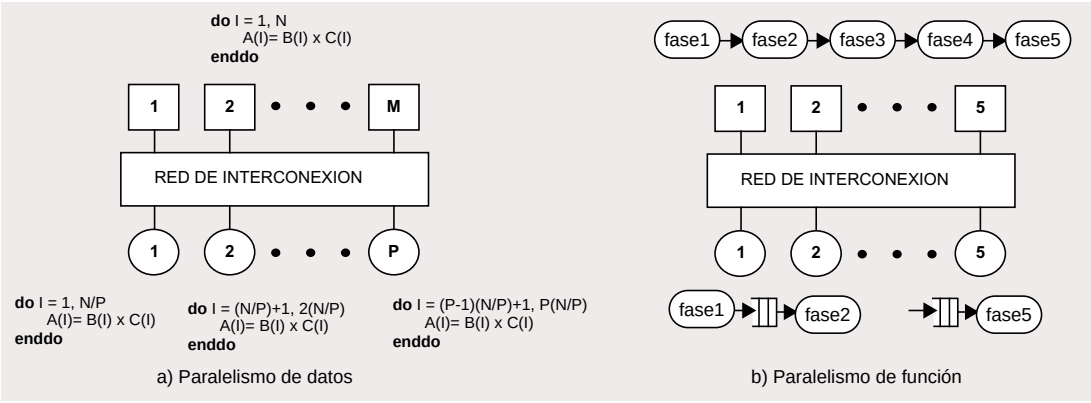


Figura 2.1 Ejemplos de paralelismo.

MODELOS DE PROGRAMACION

La explotación de la capacidad de procesado de un multiprocesador no es transparente al programador, como pueden ser mejoras a nivel de microarquitectura en un procesador. Por modelo de programación entendemos la forma de expresar, en un lenguaje de alto nivel, el paralelismo.

Podemos distinguir dos modelos básicos de programación paralela, que se diferencian en la forma de comunicar información y sincronizar la utilización de la misma entre varias tareas paralelas (Figura 2.2). Esto es, la coordinación del trabajo efectuado en cada tarea.

- Memoria compartida: se utiliza un único espacio de direcciones. La comunicación y sincronización se efectúa mediante instrucciones load y store que ejecutan los hilos[1] para acceder a las posiciones de memoria.
- Paso de mensajes: los espacios de direcciones de cada proceso son disjuntos. La comunicación y sincronización entre los procesos se efectúa mediante el intercambio explícito de mensajes, utilizando un canal de comunicación.

El modelo de memoria compartida facilita partir de un programa serie y paralelizarlo. Todos los hilos tienen acceso directo, mediante instrucciones load y store, a las estructuras de datos. Es necesario utilizar primitivas explí-

1. En el contexto de memoria compartida utilizaremos el término hilo para identificar una tarea de un programa paralelo, ya que el espacio lógico es el mismo. Ahora bien, varios procesos, cada uno con su espacio lógico, pueden acceder al mismo espacio físico utilizando el mecanismo de traducción de direcciones de lógica a física (Figura 2.2). En el contexto de paso de mensajes utilizaremos el término proceso.

citas para sincronizar la comunicación de información entre los hilos. La escalabilidad de estos multiprocesadores está limitada por la tecnología disponible.

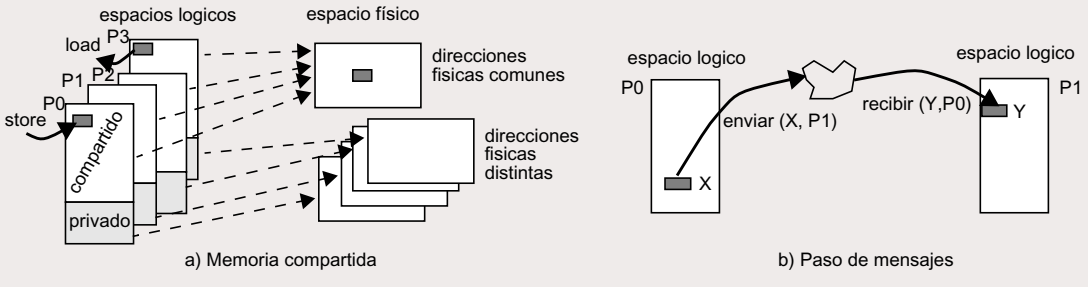


Figura 2.2 Modelos de programación paralela.

El modelo de paso de mensajes requiere particionar explícitamente o distribuir las estructuras de datos del programa entre procesos. Al finalizar hay que recolectar los resultados de cada proceso. Facilita la escalabilidad del multiprocesador, ya que la comunicación se explicita mediante primitivas donde se identifica el emisor y los receptores. La utilización de estas primitivas identifica los puntos de comunicación y sincronización.

En este capítulo y en los restantes nos centraremos en multiprocesadores con memoria compartida.

ORGANIZACIÓN DE MULTIPROCESADORES CON MEMORIA COMPARTIDA

Distinguimos dos organizaciones básicas en función de la latencia de acceso a memoria (Figura 2.3).

- Latencia de acceso uniforme: El tiempo de acceso a memoria de cualquier procesador a cualquier módulo de memoria es el mismo. Los nodos o elementos conectados a la red de interconexión son módulos de memoria o procesadores. La escalabilidad del diseño está limitada por la necesidad de que la latencia de acceso a memoria esté dentro de ciertos rangos. Esta latencia está determinada principalmente por la red de interconexión utilizada.
- Latencia de acceso no uniforme: El tiempo de acceso a memoria desde un procesador a los módulos de memoria es distinto. Los nodos son pares procesador-memoria. La latencia de acceso de un procesador al módulo de memoria con el que está emparejado es mucho menor que la latencia de acceso a otros módulos de memoria. Esta característica facilita la escalabilidad del multiprocesador, aunque requiere una distribución

adecuada de las estructuras de datos, para aprovechar la menor latencia de acceso al módulo de memoria con el que está emparejado un procesador.

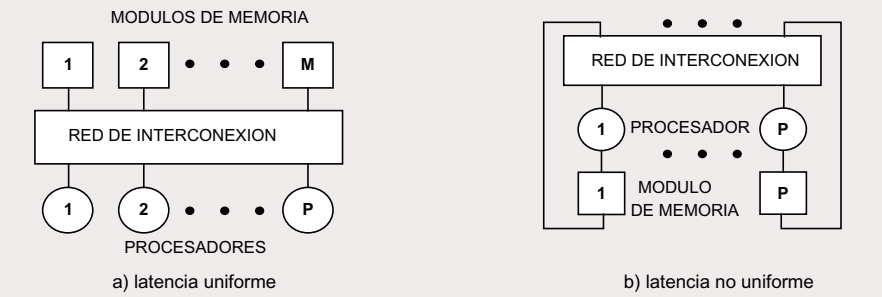


Figura 2.3 Organizaciones del multiprocesador.

En los multiprocesadores se explota la propiedad de localidad de los programas para reducir la latencia de acceso a memoria. La memoria se organiza de forma jerárquica, siendo transparente al programador. Por otro lado, para reducir la demanda de ancho de banda de los procesadores a la red de interconexión, se interponen niveles de la jerarquía entre el procesador y la red de interconexión.

Red de interconexión

La función de la red de interconexión es transportar información entre los nodos conectados a la misma. La red se utiliza siempre que se produce un fallo de cache. Entonces, la latencia de atravesar la red puede ser una fracción significativa de la latencia de acceso a memoria.

Por otro lado, en un multiprocesador la demanda para utilizar la red de interconexión es proporcional al número de procesadores. Entonces, la serialización de los accesos a memoria, que está determinada por el encaminamiento de las peticiones de los procesadores y las repuestas de los módulos de memoria, incrementa la latencia efectiva de acceso a memoria.

Seguidamente se describen algunas de las redes que se utilizan típicamente.

Bus. Es una extensión natural de un sistema uniprocesador. Al elemento que conecta el procesador con memoria se conectan más procesadores. La escalabilidad[2] del bus está limitada por la carga eléctrica que representan los procesadores y la longitud necesaria del bus para conectar los procesadores (Figura 2.4).

2. Número de procesadores y frecuencia de funcionamiento del bus.

Crossbar. Se distinguen puertos de entrada y puertos de salida. En los puertos de entrada se conectan los procesadores y en los puertos de salida los módulos de memoria (Figura 2.4). Puede observarse como p buses que están conectados a los procesadores y m buses que están conectados a los módulos de memoria. Cada uno de los p buses está conectado a cada uno de los m buses mediante un conmutador. En un intervalo de tiempo uno de los m buses sólo puede conectarse a uno de los p buses[3]. Si cada procesador solicita acceso a un módulo de memoria distinto se pueden efectuar p accesos en paralelo (p menor igual que m)[4]. La escalabilidad de la red está limitada por el área que ocupa, la cual también influye en la latencia necesaria para atravesar la red de interconexión.

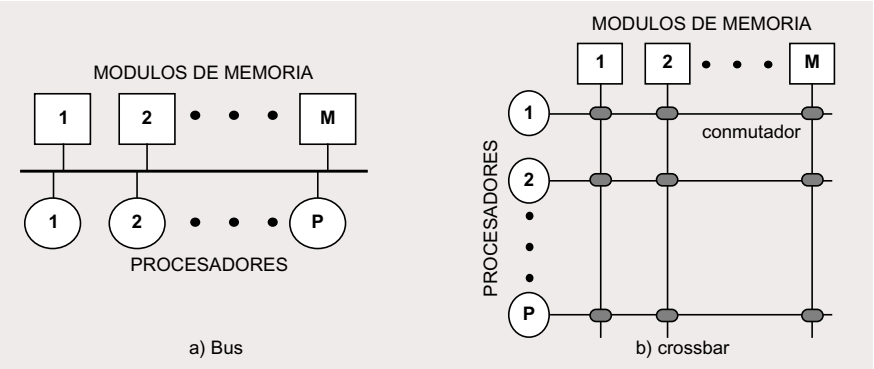


Figura 2.4 Redes de interconexión: a) bus y b) crossbar.

Anillo. La red puede observarse como conexiones punto a punto entre nodos del multiprocesador (Figura 2.5). Los nodos pueden ser módulos de memoria, procesadores o el emparejamiento de ambos. Entre dos nodos adyacentes en la red existe comunicación punto a punto. Sin embargo, para establecer una comunicación entre dos nodos no adyacentes hay encaminar la información pasando por los nodos ubicados entre el nodo fuente y el nodo destino. Entonces, cada nodo debe disponer de un componente que encamine los mensajes cuyo destinatario es otro nodo. En un anillo, pueden producirse tantas comunicaciones en paralelo como conexiones punto a punto hay en el anillo. La escalabilidad del anillo está limitada por la latencia de comunicación entre los dos nodos más alejados entre sí[5].

3. Diremos que es una conexión punto a punto. Una red crossbar puede utilizarse en otros contextos, donde interesa difundir información desde un emisor a varios receptores. Para ello, uno de los p buses se conecta a varios de los m buses. Entonces diremos que es una conexión para difundir información.
4. Un bus puede considerarse un caso degenerado de crossbar (m = 1).
5. Diámetro (número de nodos), el cual es N/2 para N nodos.

Malla. Para reducir la latencia de comunicación debido a la lejanía entre nodos, la red de interconexión se organiza en dos dimensiones. Los nodos están en los cruces de los enlaces de comunicación. Un nodo, no periférico, puede estar comunicándose con los cuatro nodos a los que está conectado[6].

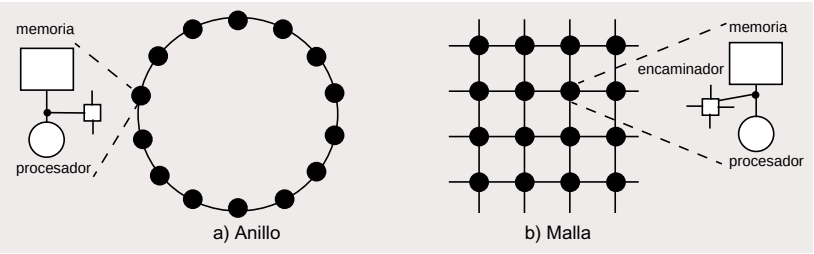


Figura 2.5 Redes de interconexión: a) anillo y b) malla.

OPERACIONES DE SINCRONIZACIÓN

Por comunicación entendemos la transmisión de información entre hilos, la cual, en un multiprocesador de memoria compartida, se efectúa mediante posiciones de almacenamiento en memoria y la ejecución de instrucciones load y store. Las sincronizaciones se efectúan utilizando posiciones de almacenamiento en memoria. Entonces, una sincronización puede observarse como una forma especial de comunicación en la que los datos son información de control.

En un multiprocesador la sincronización entre hilos se utiliza para satisfacer, en tiempo de ejecución, las restricciones impuestas por las dependencias entre hilos. Esto es, un objetivo de una operación de sincronización es proveer un mecanismo para que un hilo acceda a una variable después de conocer que ha sido actualizada (dependencia verdadera) o viceversa, esperar a que una variable haya sido leída para actualizarla (antidependencia). Otro objetivo de una operación de sincronización es garantizar que sólo un hilo, entre varios, está actualizando un conjunto de variables en un instante determinado. El primer caso se denomina sincronización mediante eventos y el segundo acceso excluyente.

Finalmente se describe la operación de sincronización barrera que fuerza las dependencias entre fases de cálculo de grupos de operaciones. Todos los hilos ejecutan la operación barrera y esperan hasta que el último hilo ejecuta la operación.

6. El diámetro de una malla cuadrada es 2 x (N -1), siendo N^2 el número total de nodos.

Soporte de la arquitectura

La característica clave que debe tener una instrucción, para ser utilizada en operaciones de sincronización, es que el acceso a memoria sea una operación atómica.

A nivel de lenguaje máquina, la unidad de ejecución indivisible para acceder a memoria son las instrucciones load y store. Si dos instrucciones acceden concurrentemente a la misma posición de memoria el hardware serializa el acceso.

Las dos operaciones de sincronización descritas, evento y acceso exclusivo, se pueden implementar mediante instrucciones load y store. Seguidamente se presentan ejemplos para mostrar la necesidad de sincronización y el mecanismo para realizarla.

En la Figura 2.6 se muestra una sincronización mediante evento y en la Figura 2.10 se muestra el acceso exclusivo a un conjunto de variables utilizando instrucciones load y store.

Sincronización mediante evento

Los hilos que se muestran en la parte izquierda de la Figura 2.6 se comunican un valor utilizando la variable A. En concreto, el hilo H1 escribe un valor en la variable A y el hilo H2 lee el valor.

Hilo H1	Hilo H2	Comentarios
. . .	. . .	R1 = R6 = dirección de la variable A
store R3, 0(R1)	load R4, 0(R6)	
. . .	. . .	

Hilo H1	Hilo H2	Inicialización
A = 3.14	While (aviso = 0) { };	aviso = 0
aviso = 1	T = A	
. . .	. . .	

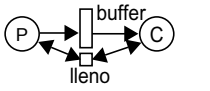
Figura 2.6 Sincronización mediante evento.

Para garantizar que el hilo H2 lee la variable A después de que el hilo H1 la haya actualizado hay que incluir una operación de sincronización, denominada por evento. Para ello se utiliza otra variable, denominada aviso, que el hilo H1 actualiza después de escribir en la variable cuyo valor quiere comunicar. El hilo H2 consulta el valor de la variable aviso. Mientras el valor de variable aviso no tiene un valor predeterminado no prosigue la ejecución (parte derecha de la Figura 2.6).

Ejercicio

Suponga el siguiente código que se utiliza para comunicar información entre dos hilos. Uno de los hilos se denomina productor y el otro hilo se denomina consumidor.

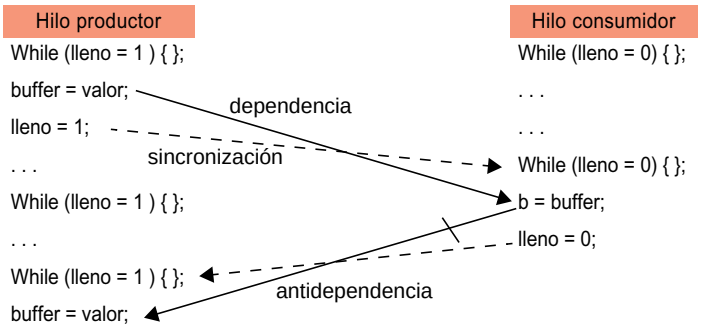
Hilo productor	Hilo consumidor
While (lleno = 1) { };	While (lleno = 0) { };
buffer = valor;	b = buffer;
lleno = 1;	lleno = 0;



Indique las operaciones de sincronización que se utilizan y el tipo de dependencia de datos entre hilos que se quieren garantizar en tiempo de ejecución.

Respuesta

La operación de sincronización es de tipo evento. Se utiliza únicamente una variable para efectuar dos operaciones de sincronización y la variable es lleno. En el sentido productor-consumidor garantiza que el hilo consumidor no lee el contenido de la variable buffer hasta que el hilo productor ha escrito un valor. Esto es, una dependencia de datos verdadera. En el sentido consumidor-productor garantiza que el hilo productor no actualice el valor de la variable buffer antes de que el hilo productor la haya leído. Esto es, una antidependencia.



Comunicación entre dos hilos

Una forma de paralelismo es el paralelismo a nivel de tarea o paralelismo funcional, en el cual funciones independientes se ejecutan en procesadores distintos. Por ejemplo, se pueden encadenar varias funciones de forma segmentada y el objetivo es procesar un flujo de datos.

Para conectar un hilo productor y un hilo consumidor usualmente se utiliza un buffer con varias entradas. Ello permite que el hilo productor pueda producir ráfagas de información más rápidamente que la rapidez de consumo del hilo consumidor, aunque en media la rapidez de producción sea menor igual que la rapidez de consumo.

El buffer se utiliza para acomodar las ráfagas de producción con la rapidez media del hilo consumidor. El hilo productor almacena la información en el buffer mientras el hilo consumidor procesa la información previa. En la Figura 2.7 se muestra un diseño de los dos hilos con un buffer que se gestiona de forma circular. El hilo productor almacena la información en entradas consecutivas del buffer, utilizando la variable cola que indica la primera entrada libre. De forma similar, el hilo consumidor extrae información de entradas consecutivas del buffer, utilizando la variable cabeza que indica la primera entrada con información.

Hilo productor	Hilo consumidor	Inicialización
while (cola = (cabeza + N -1) mod N) { };	While (cabeza = cola) { };	cabeza = cola = 0
buffer(cola) = item;	item = buffer(cabeza);	
cola = (cola + 1) mod N;	cabeza = (cabeza + 1) mod N;	
	PROCESADO (item);	

Figura 2.7 Buffer circular que utilizan un hilo productor y un hilo consumidor.

Para determinar la condición de buffer lleno o vacío se utilizan las variables cola y cabeza[7]. Observemos que estas variables sólo son actualizadas por un hilo. Por tanto, no es necesario establecer una serialización al acceder a ellas.

Acceso exclusivo

En el ejemplo del buffer circular, cuando la rapidez con que el hilo productor produce datos es mayor que la rapidez con la cual las consume el hilo consumidor, una alternativa es utilizar varios hilos consumidores, hasta equiparar la rapidez de producción con la de consumo[8].

Ahora bien, un entrelazado de las instrucciones de dos hilos consumidores (Figura 2.7), como el mostrado en la Figura 2.8, puede producir resultados no esperados. Por ejemplo, el entrelazado muestra que los dos hilos leen la misma entrada del buffer.

tiempo	Hilo consumidor 1	Hilo consumidor 2
1	while (cabeza = cola) { };	While (cabeza = cola) { };
2	item = buffer(cabeza);	
3		item = buffer(cabeza);

Figura 2.8 Entrelazado de sentencias de dos hilos consumidores que se ejecutan concurrentemente.

7. Observe que el buffer como máximo puede contener N-1 elementos sin extraer antes de que el consumidor se bloquee.
8. Paralelismo de datos accediendo a una única estructura de datos para obtener la información.

Para que el entrelazado sea correcto hay que garantizar que el acceso a algunas variables compartidas está serializado. Esto es, hasta que un hilo no ha efectuado todas las operaciones de lectura y escritura, ningún otro hilo consumidor puede acceder a las variables compartidas. Esta acción de sincronización se denomina garantizar acceso exclusivo. Notemos que un hilo consumidor necesita incrementar el valor de la variable cabeza, para indicar que puede leerse información de la siguiente entrada. Por tanto, sólo un hilo consumidor puede acceder al buffer. En la Figura 2.9 se muestra un ejemplo de código donde se indican las instrucciones que acceden de forma exclusiva a las variables compartidas por los hilos productores. Notemos que una vez obtenido el acceso exclusivo se vuelve a comprobar si existe alguna entrada que debe procesarse.

Hilo productor	Hilo consumidor	Inicialización
while (cola = (cabeza + N -1) mod N) { };	repeat	cabeza = cola = 0
buffer(cola) = item;	While (cabeza = cola) { };	
cola = (cola + 1) mod N;	volver = 0	
	Adquirir acceso exclusivo	
	if (not (cabeza = cola)) then	
	item = buffer(cabeza);	
	cabeza = (cabeza + 1) mod N;	
	else	
	volver = 1	
	endif	
	Liberar acceso exclusivo	
	until volver = 0	
	PROCESADO (item);	

Figura 2.9 Buffer circular que utilizan un hilo productor y varios hilos consumidores.

Implementación del acceso exclusivo mediante instrucciones load y store

En este apartado se muestra una forma de garantizar acceso exclusivo mediante instrucciones load y store. Notemos que se utilizan dos sincronizaciones tipo evento (aviso1 y aviso2) y una variable turno que determina el hilo que accede a la zona de exclusión cuando intentan entrar los dos hilos concurrentemente. En este caso, obtiene el acceso exclusivo el hilo que no ejecuta la sentencia “turno = “ el último.

Comentarios	Hilo H1	Hilo H2	Inicialización
	aviso1 = 1;	aviso2 = 1;	aviso1 = aviso2 = 0
adquirir el acceso	turno = 1;	turno = 2;	
	while ((aviso2 = 1) and (turno = 1)) {};	while ((aviso1 = 1) and (turno = 2)) {};	
	acceso exclusivo	acceso exclusivo	
liberar el acceso	aviso1 = 0;	aviso2 = 0;	

Figura 2.10 Obtención de acceso exclusivo, algoritmo de Peterson, utilizando instrucciones load y store.

La implementación de acceso excluyente mediante instrucciones load y store tiene deficiencias y no escala cuando se incrementa el número de procesadores. Por ello el lenguaje máquina de los procesadores se extiende con instrucciones atómicas que realizan las siguientes operaciones de forma indivisible o atómica: a) lectura, b) modificación y c) escritura. Estas instrucciones facilitan la implementación de acceso exclusivo a variables compartidas y las denominaremos de forma genérica instrucciones LME.

Instrucciones atómicas de lectura-modificación-escritura

Una característica clave de la ejecución de una instrucción LME, que accede a una posición de memoria, es que la secuencia de lectura, modificación y escritura es atómica. Esto es, además de que no se gestionan cambios del flujo de ejecución, forzados[9] o no, durante la ejecución de una instrucción LME el hardware serializa el acceso de otros procesadores a la posición de memoria que referencia.

En la Figura 2.11 se muestra la especificación de varias instrucciones atómicas. Una operación típica es un intercambio atómico. Una instrucción que implementa esta operación intercambia el contenido de un registro con el contenido de una posición de memoria. Cuando dos procesadores en un multiprocesador ejecutan concurrentemente una instrucción intercambio, estas serán ordenadas por el mecanismo de serialización de escrituras.

intercambio (llave, R)	Test&set (llave)	fetch&inc (dir)	Comentario
tmp = llave	tmp = llave	tmp = dir	Secuencia de instrucciones que se ejecutan de forma atómica.
llave = R	llave = 1	dir = tmp + 1	
return R = tmp	return tmp	return tmp	

Figura 2.11 Ejemplos de instrucciones atómicas.

9. Por ejemplo, una interrupción.

Otras operaciones atómicas son: a) test&set y b) fetch&inc. En una implementación de la instrucción test&set se lee el valor de una posición de memoria y posteriormente se escribe un uno. En una instrucción fetch&inc se lee el valor de una posición de memoria y atómicamente se almacena el valor leído incrementado en una unidad.

En la Figura 2.12 se muestra la utilización de la instrucción atómica intercambio para implementar la primitiva adquirir, que junto con la primitiva liberar permite garantizar el acceso exclusivo a un conjunto de datos compartidos de una secuencia de instrucciones.

adquirir (llave)	liberar (llave)	acceso exclusivo	Inicialización
R = 1	llave = 0	. . .	llave = 0
repeat	return	adquirir (llave)	
intercambio (llave, R)		acceso exclusivo	
until R = 0		liberar (llave)	
return		. . .	

Figura 2.12 Primitivas adquirir y liberar y su utilización para garantizar acceso exclusivo.

En la Figura 2.13 se muestra la utilización de la instrucción fetch&inc para obtener el número de iteración que debe ejecutar un hilo, perteneciente al conjunto de hilos que ejecutan un bucle paralelo.

Bucle paralelo	Autoplanificación	Comentarios
doall I = 1, N	LI = fetch&inc (GI)	GI: variable compartida
. . .	while (LI =< N)	LI: variable local. Número de iteración
endoall	. . .	
	LI = fetch&inc (GI)	Inicialmente GI = 1
	endwhile	

Figura 2.13 Obtención del número de iteración por parte de los hilos que calculan un bucle paralelo. Utilización de la instrucción atómica fetch&inc para autoplanificación.

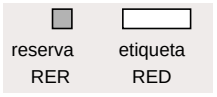
Instrucciones load con enlace y almacenamiento condicional

La implementación de una instrucción de lectura-modificación-escritura de forma atómica no es sencilla. Por ello se han añadido en algunos lenguaje maquina dos instrucciones que implementan individualmente las operaciones de lectura y actualización, pero conjuntamente implementan una operación LME. Para ello se incrementa la semántica las instrucciones load y store conocidas[10]. Estas instrucciones se denominan load con enlace (LL) y almacenamiento condicional (SC)[11].

Para garantizar que la ejecución de la secuencia de instrucciones LL y SC es atómica el hardware dispone de recursos para: a) identificar la secuencia de instrucciones LL y SC y b) observar las escrituras de otros procesadores a la posición de memoria a la que acceden. En estas condiciones, la operación de actualización no se efectúa si, en el lapso de tiempo transcurrido entre la ejecución de la instrucción LL y la instrucción SC, se ha detectado una escritura a la posición de memoria.

Para la implementación de las instrucciones LL y SC se utiliza un registro denominado registro de enlace (RE). El registro contiene la dirección de la posición de memoria que lee la instrucción LL. El contenido del registro RE se invalida cuando se observa una escritura, en el lapso de tiempo transcurrido entre la ejecución de la instrucción LL y la ejecución de la instrucción SC[12]. Al interpretar la instrucción SC se analiza si el registro RE es inválido. Si es el caso, la instrucción SC se convierte en una instrucción nop (se anula). En caso contrario se actualiza la posición de memoria.

En la Figura 2.14 se especifican las instrucciones load con enlace y almacenamiento condicional.

load con enlace: LL R_d, X(R_f)	Descripción	Registro de enlace (RE)
$RED^v = X + R_f^v$ $RER^v = 1$ $R_d^v = M [X + R_f^v]$ el superíndice v indica valor	Almacena en R_d el contenido de la posición de memoria cuya dirección se calcula como $X + R_f^v$ y efectúa la acción de reserva almacenando la dirección $X + R_f^v$ en un registro denominado registro de enlace (RE).	

almacenamiento condicional: SC R_{fd}, X(R_f)	

if ($RED^v = X + R_f^v$ and $RER^v = 1$) then
cancelar la reserva de otros procesadores de la dirección $X + R_f^v$
$M [X + R_f^v] = R_{fd}^v$
$R_{fd}^v = 1$
else
$R_{fd}^v = 0$
	Comprueba si la reserva de la dirección de memoria en el registro RE es válida. Si es el caso, el contenido del registro R_{fd} se almacena en la posición de memoria cuya dirección se calcula como $X + R_f^v$ y en el registro R_{fd} se almacena el valor uno. En otro caso, la instrucción se anula y en el registro R_{fd} se almacena un cero

Figura 2.14 Especificación de las instrucciones load con enlace y almacenamiento condicional.

10. En una instrucción atómica la operación de lectura no es observable por otros procesadores y como no modifica el estado de memoria no es necesario excluirla en el lapso de tiempo requerido para ejecutar una operación atómica.
11. En inglés, usualmente, load linked y store conditional.
12. También debe invalidarse, por ejemplo, en operaciones de cambio de contexto del procesador. En los capítulos de coherencia de cache se entra en mayor detalle.

En la Figura 2.15 se muestra la utilización de las instrucciones LL y SC para implementar la operación atómica intercambio. Después de ejecutar las instrucciones LL y SC, para efectuar el intercambio, se comprueba si se ha ejecutado la instrucción SC de forma efectiva. Si este no es el caso, se vuelve a intentar, ya que la ejecución atómica de un intercambio no se ha realizado.

intercambio (llave, R)		Intercambio (0(R1), R4)	comentario
tmp = llave	1$:	mov R3, R4	mover valor de intercambio
llave =R		LL R2, 0(R1)	Load Linked
return R = tmp		SC R3, 0(R1)	Store condicional
		beq R3, 1$	repetir si se anula
		mov R4, R2	R1: dirección de la variable llave
		return	

Figura 2.15 Operación intercambio implementada mediante instrucciones LL y SC.

Observemos que entre las instrucciones LL y SC se pueden ejecutar instrucciones que utilizan como operandos valores almacenados en el banco de registros y que almacenan el resultado en un registro del banco de registros[13]. Una recomendación usual es que entre una instrucción LL y una instrucción SC no se efectúen accesos a memoria. También, no es usual soportar el entrelazado o imbricado del secuencia de instrucciones LL y SC.

Consideraciones en una operación de sincronización

Seguidamente se efectúan algunas consideraciones sobre una operación de sincronización. En una sincronización distinguimos tres fases:

- Fase de adquisición u obtención. Se establece un valor en una variable compartida para indicar la adquisición de una sincronización.
- Fase de espera. Nos centraremos en lo que se denomina espera activa. Esto es, la espera se efectúa mediante la ejecución de instrucciones cuyo fin es esperar la adquisición de la sincronización.
- Fase de liberación. Se establece un valor en una variable compartida para indicar la liberación de una sincronización.

Cada una de las fases anteriores presenta características distintas en función de la operación de sincronización. Sin embargo, de forma genérica interesa tener en cuenta las siguientes características:

- Fase de adquisición: la latencia debe ser reducida cuando no existen otros hilos que efectúan la misma fase.

13. Para que no se incremente la probabilidad de que otro procesador escriba en la posición de memoria accedida, el número de instrucciones entre una instrucción LL y una instrucción SC tiene que ser reducido.

- Fase de espera: el tráfico que se inyecte en la red de interconexión debe ser bajo.
- Necesidades de almacenamiento: el número de posiciones de memoria utilizadas debe ser reducido.
- Escalabilidad: tanto la latencia como el tráfico y el almacenamiento necesario en las fases de adquisición y espera debe incrementarse linealmente con el número de hilos.
- Equitativo: hay que garantizar que no se produce inanición. Esto es, un hilo no se queda relegado de efectuar una adquisición de forma indefinida.

Ejercicio

En la operación de sincronización por evento, mostrada en la Figura 2.6, indique las fases de adquisición y espera y las necesidades de almacenamiento. En una operación de acceso exclusivo, como la mostrada en la Figura 2.12, indique las fases de adquisición y espera y las necesidades de almacenamiento.

Respuesta

En la operación de sincronización que se muestra seguidamente, la fase de adquisición es la ejecución de la sentencia aviso = 1, la fase de espera es el bucle while y la necesidad de almacenamiento es una posición de memoria.

Hilo H1	Hilo H2	Inicialización
A = 3.14	While (aviso = 0) { };	aviso = 0
aviso = 1	T = A	
. . .	. . .	

En una operación de sincronización, para garantizar acceso exclusivo, la operación intercambio en la primitiva adquirir y la actualización de la variable llave en la primitiva liberar son respectivamente las fases de adquisición y liberación. La necesidad de almacenamiento es una posición de memoria (variable llave). La fase de espera es el bucle repeat.

```
adquirir (llave)          liberar (llave)      acceso exclusivo
R = 1                     llave = 0            llave = 0
repeat                    return               . . .
   intercambio (llave,R)                       adquirir (llave)
until R = 0                                       acceso exclusivo
return                                         liberar (llave)
```

Operacion barrera

Una barrera es una operación de sincronización que fuerza las dependencias entre fases de cálculo de grupos de operaciones. Todos los hilos ejecutan la operación barrera y esperan hasta que el último hilo ejecuta la operación. Podemos decir que una operación barrera es un sincronización por evento.

En la Figura 2.16 se muestra un ejemplo de utilización de la operación BARRERA. Los cálculos efectuados en el primer bucle se utilizan en el segundo bucle. Esto es, hay dependencias de datos. La planificación de iteraciones entre los procesadores no garantiza que un procesador ejecute las mismas iteraciones en los dos bucles. Entonces, antes de empezar a ejecutar el segundo bucle hay que esperar a que finalicen todas las iteraciones del primer bucle[14].

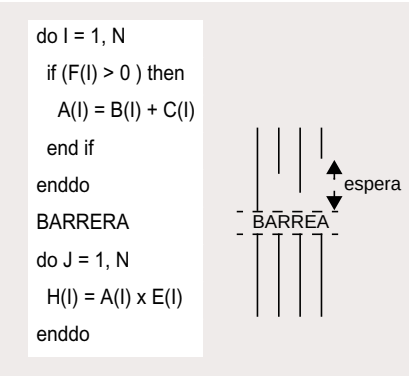


Figura 2.16 Ejemplo de utilización de la operación de sincronización BARRERA.

En la implementación de la operación BARRERA se utiliza una llave y un evento. La llave se utiliza para garantizar el acceso exclusivo a un contador, que contabiliza los hilos que han ejecutado la operación barrera. El evento se utiliza para que los hilos esperen hasta que el contador toma el valor cero.

En la parte izquierda de la Figura 2.17 se muestra un esquema del código de una operación barrera. Para contabilizar el número de hilos que han ejecutado la operación se utiliza un contador. En la parte derecha se muestra una implementación, utilizando las primitivas adquirir y liberar, para obtener acceso exclusivo al contador. Para efectuar el bloqueo se utiliza la variable liberar, mediante la cual se implementa una sincronización tipo evento.

14. Podrían utilizarse sincronizaciones por evento individualizadas por dependencia. Ahora bien, el número de eventos necesarios es proporcional al número de iteraciones.

BARRERA
b = b+1
if b < N **then**
bloquear hilo en la cola
else
desbloquear todos los hilos de la cola
b := 0
endif
return

	BARRERA	comentario
1$:	adquirir (llave)	acceso exclusivo
	if (contador = 0) liberar =0	inicializar evento de finalización
	contador = contador +1	
	liberar (llave)	
	if (contador = N) then	evento de finalización
	contador =0	
	liberar =1	activar evento de finalización
	else	
	repeat	esperar a la activación
	until (liberar = 1)	del evento de finalización
	endif	
	return	

Figura 2.17 Operación de sincronización B RRER .N es el número de procesadores.

Si el valor de la variable contador es cero se desactiva el evento liberar, ya que se inicia la ejecución de la operación atómica barrera. Previamente se ha adquirido el acceso exclusivo a la variable contador. Después de incrementar el contenido de la variable contador se libera el acceso exclusivo. Si el valor de la variable contador es menor que N los hilos se esperan, utilizando espera activa, en el bucle repeat. En caso contrario se establece el valor cero en la variable contador y se habilita la liberación[15].

La operación barrera descrita tiene contención, tanto en la fase de contabilizar el número de procesadores que han llegado a ella como en la fase de dejar la barrera. En la fase de contar el número de procesadores, la contención es importante ya que hay que obtener acceso exclusivo a la variable contador. La fase de dejar la barrera solo tiene contención de lecturas. Una alternativa es utilizar una estructura de datos en árbol para implementar la operación barrera.

En la Figura 2.18 se muestra un esquema. Las hojas son los hilos y la llegada de los hilos a la barrera se contabiliza de forma distribuida; en sentido ascendente. En la Figura 2.18 cada nodo del primer nivel del árbol contabiliza a 2 hilos. En el siguiente nivel, se contabilizan 2 grupos de 2 hilos. En el último nivel, que es la raíz, se contabilizan 4 grupos de 2 hilos. Cuando el último de

15. Esta implementación de la operación barrera (parte derecha de la Figura 2.17) no funciona correctamente si se utiliza varias veces en un código. Por ejemplo, entre varios bucles paralelos consecutivos con dependencias. En un ejercicio se propone su modificación.

los 2 hilos llega a un nodo se lo notifica al nodo del siguiente nivel. Posteriormente la liberación también puede efectuarse de forma distribuida siguiendo el árbol en sentido inverso.

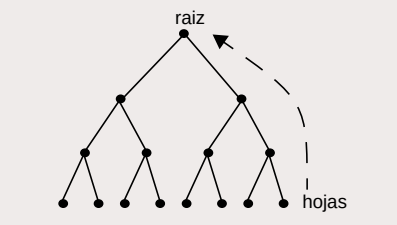


Figura 2.18 Barrera con estructura en árbol para reducir la contención.

LEY DE AMDAHL

La ley de Amdahl permite calcular la ganancia que se obtiene al ejecutar un programa partiendo de: a) la fracción de tiempo en la cual puede explotarse una mejora y b) la ganancia que se obtiene cuando se utiliza la mejora.

En la Figura 2.19 se muestra un diagrama donde, en el tiempo de ejecución de una programa en el procesador original, se distingue la proporción de tiempo en la cual se pueden utilizar los P procesadores. Para simplificar el dibujo, los instante de tiempos en los cuales se pueden utilizan los P procesadores se han agrupado de forma contigua (t_2). La ganancia depende de dos factores: a) la fracción de tiempo en el procesador original donde pueden utilizarse los P procesadores ($f_m = t_2/T_o$)[16] y b) la ganancia cuando se utilizan los P procesadores el 100% ($P = t_2/t_3$).

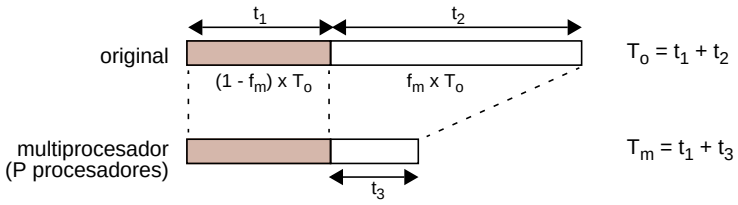


Figura 2.19 Relación entre el tiempo de ejecución en el procesador original y el tiempo de ejecución en el multiprocesador.

Teniendo en cuenta los tiempos de ejecución, la ganancia se calcula como

$$G = \frac{T_o}{T_m} = \frac{T_o}{t_1 + t_3}$$

16. A esta fracción de tiempo también se la denomina, en ocasiones, fracción de código que se paraleliza.

Como $f_m = t_2/T_o$ y $P = t_2/t_3$, efectuando manipulaciones algebraicas obtenemos.

$$G = \frac{1}{(1 - f_m) + \frac{f_m}{P}}$$

Cuando el número de procesadores tiende a infinito obtenemos la siguiente expresión, que indica la máxima ganancia que se puede obtener.

$$G|_{P \to \infty} = \frac{1}{1 - f_m}$$

Revisión I

La ley de Amdahl supone un tamaño fijo del problema que se resuelve o unas funcionalidades que no se modifican. Sin embargo, cuando se dispone de más capacidad de procesado, es típico añadir más funcionalidades o mejorar la precisión del cálculo, lo cual incrementa el tamaño del problema[17].

En este contexto la carga de trabajo se incrementa pero se espera que, al disponer de más procesadores, el tiempo de ejecución sea el mismo que antes o se reduzca (Figura 2.20). Adicionalmente se espera que la carga de trabajo adicional se pueda ejecutar en paralelo.

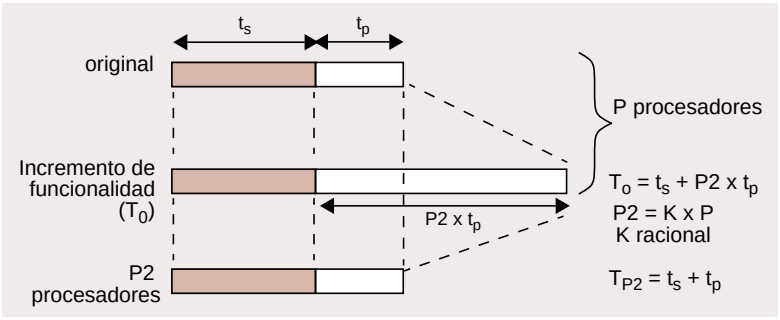


Figura 2.20 Ganancia teniendo en cuenta un incremento de las funcionalidades de la aplicación.

En el contexto descrito tenemos que

$$G = \frac{T_o}{T_{P2}} = \frac{t_s + t_p \times P_2}{t_s + t_p} = (1 - \beta) + \beta \times P_2 \qquad \text{siendo} \qquad \beta = \frac{t_p}{t_s + t_p}$$

Entonces, si la funcionalidad añadida[18] también utiliza los procesadores adicionales ($P_2 > P$), se obtiene una ganancia que es lineal con la parte que utiliza el incremento de capacidad de procesado.

17. Este hecho no es exclusivo del contexto multiprocesador. Se ha estado produciendo al incrementar las prestaciones de un sistema uniprocesador.

Desde el punto de vista del usuario se dispone de mayor funcionalidad y el tiempo de respuesta es el mismo.

Revisión II

En una ejecución paralela la inicialización de los hilos es una tarea serie. Por otro lado, las operaciones de sincronización no son necesarias en una ejecución serie del programa. Estos tiempos representan una penalización al ejecutar un programa paralelo, respecto su ejecución serie, y pueden reducir la ganancia esperada al ejecutar el programa en paralelo.

Para que el efecto de la penalización de una sincronización sea reducido, una operación de sincronización debe permitir realizar un número significativo de cálculos. Si además esta sincronización determina mucha comunicación de información, puede ser otro factor que reduzca la ganancia. Por ejemplo, no es lo mismo efectuar 10 cálculos con 2 procesadores, antes de efectuar una sincronización, que utilizar 20 procesadores, donde cada uno de ellos realiza un cálculo por operación de sincronización.

En la Figura 2.21 se muestra una representación del tiempo de ejecución de un programa en un procesador y el tiempo de ejecución en un multiprocesador, donde se explicita el tiempo de sincronización.

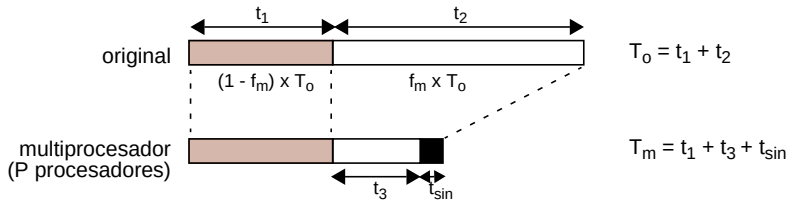


Figura 2.21 Relaciones de tiempos de ejecución teniendo en cuenta la sincronización.

La expresión de la ganancia cuando se tiene en cuenta la penalización debida a la comunicación y sincronización es

$$G = \frac{1}{(1 - f_m) + \frac{f_m}{P} + \frac{t_{sin}}{T_0}}$$

18. El tiempo requerido por el incremento de funcionalidades es $(P_2 - 1) \times t_p$.

En la Figura 2.22 se muestra una gráfica con valores representativos de f_m y t_{sin}. Observemos que cuando t_{sin} es distinto de cero, un incremento del número de procesadores puede reducir la ganancia.

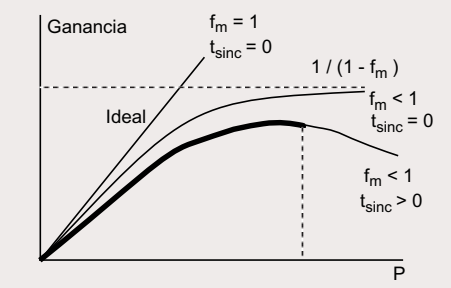
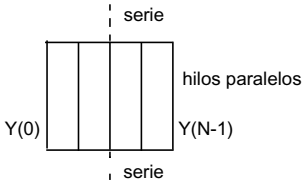


Figura 2.22 Ganancia teniendo en cuenta el tiempo de comunicación y sincronización. El valor $1 / (1 - f_m)$ se calcula suponiendo un número ilimitado de procesadores, siendo $f_m < 1$ y $t_{sin} = 0$.

Como ejemplo en la Figura 2.23 se muestra la ejecución del producto matriz por vector. En la parte izquierda, cada procesador calcula el producto de una fila de la matriz por el vector, para calcular un elemento del vector resultado. Se pueden estar calculando todos los elementos del vector resultado en paralelo. En la parte derecha se utilizan todos los procesadores para calcular un elemento del vector. Se produce una serialización debido a la operación de reducción que hay que realizar[19]. La operación de reducción se efectúa en paralelo. Para ello los cálculos, en concreto las sumas, se efectúan en un orden distinto al especificado por el programador.

Cálculo de un elemento del vector Y en cada procesador

```
doall i = 0, N-1
  do j = 0, N-1
    Y(i) = Y(i) + A(i, j) * X (j)
```



Cálculo de un elemento del vector Y utilizando todos los procesadores

```
do i = 0, N-1
  S = Y(i)
  doall j = 0, N-1
    S = S + A(i, j) * X (j)
  Y(i) = S;
```

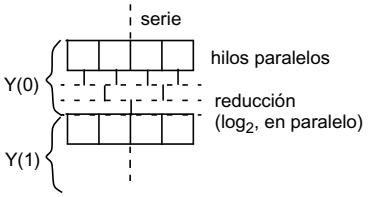


Figura 2.23 Cálculo del producto matriz por vector en paralelo. El bucle en negrita es el que se ejecuta en paralelo. Los dibujos no están en la misma escala.

19. Una operación de reducción requiere que los procesadores accedan (lectura y escritura) a variables compartidas. Por tanto, hay que garantizar acceso exclusivo.

Otro ejemplo es la operación de sincronización barrera que hay que utilizar entre dos bucles paralelos, cuando el primero calcula datos que se utilizan en el segundo y no se utiliza una planificación específica para asignar procesadores a hilos, si ello es posible.

Eficiencia

Se entiende por eficiencia la capacidad de utilizar los recursos disponibles. En este contexto, los recursos son los procesadores. Entonces, la eficiencia se calcula mediante la siguiente expresión.

$$E = \frac{G}{P}$$

EJERCICIOS

Ejercicio 2.1

Considere la ejecución paralela de un programa, con 200.000 instrucciones, en un multiprocesador de memoria compartida con 4 procesadores que funcionan a una frecuencia de 40 Mhz. El programa consiste de 4 tipos de instrucciones. La frecuencia de los tipos de instrucciones y el número de ciclos (CPI) necesarios para cada tipo de instrucción se han obtenido de una traza experimental del programa.

tipo de instrucción	CPI	frecuencia
aritméticas y lógicas	1	60 %
load/store con acierto en cache	2	18 %
secuenciamiento explícito	4	12 %
referencias a memoria con fallo de cache	8	10%

Pregunta 1: *Calcule el CPI medio cuando el programa se ejecuta en un solo procesador.*

Pregunta 2: *Calcule los MIPS cuando el programa se ejecuta en un solo procesador.*

Para la ejecución con 4 procesadores el programa se divide en 4 partes. Suponga que la frecuencia de instrucciones en cada parte es la misma que en el programa original. El CPI para las instrucciones que referencian a memoria y producen un fallo de cache se ha incrementado de 8 a 12 ciclos, debido a colisiones en la red de interconexión o en los módulos de memoria. El CPI para el resto de instrucciones no se modifica.

El programa se divide en cuatro partes de 30.000, 40.000, 60.000 y 70.000 instrucciones cada una.

Pregunta 3: *Calcule el CPI cuando el programa se ejecuta en el multiprocesador con 4 procesadores.*

Pregunta 4: *Calcule la ganancia cuando el programa se ejecuta en un multiprocesador con 4 procesadores respecto a una ejecución serie.*

Pregunta 5: *Calcule la eficiencia del multiprocesador con 4 procesadores.*

Pregunta 6: *¿Por qué no se obtiene máxima eficiencia?*

Para balancear la carga de trabajo en los 4 procesadores, el programa se divide en cuatro parte iguales (50.000 instrucciones en cada parte). Debido a las necesidades de sincronización entre las cuatro partes, en el tiempo de

ejecución debe considerarse el equivalente a la ejecución de 5000 instrucciones extras en cada parte individualmente (son instrucciones que no pertenecen al cálculo original, tiempos de espera, etc.).

Pregunta 7: *Calcule el CPI cuando el programa se ejecuta en un multiprocesador con 4 procesadores.*

Pregunta 8: *Calcule los MIPS cuando el programa se ejecuta en un multiprocesador con 4 procesadores.*

Pregunta 9: *Calcule la ganancia al ejecutar el programa en un multiprocesador con 4 procesadores respecto de una ejecución en un sólo procesador.*

Pregunta 10: *Calcule la eficiencia del multiprocesador con 4 procesadores.*

Pregunta 11: *¿Por qué no se obtiene máxima eficiencia?.*

Ejercicio 2.2

Se define ganancia de rendimiento como

$$G(P) = \frac{T_1(n)}{T_P(n)}$$

siendo $T_1(n)$ el tiempo de ejecución del programa en 1 procesador y $T_P(n)$ el tiempo de ejecución con P procesadores. Se define eficiencia como

$$E(P) = \frac{G(P)}{P} = \frac{T_1(n)}{T_P(n) \times P}$$

lo cual es una indicación del grado actual de ganancia que se obtiene comparado con el máximo valor de la ganancia.

Pregunta 1: *Determine el valor máximo de la ganancia.*

Pregunta 2: *Determine los valores extremos de la eficiencia.*

Pregunta 3: *¿Cómo debe variar la ganancia en función de P para que la eficiencia sea constante?.*

Ejercicio 2.3

El tiempo de ejecución de un programa en un procesador es

$$T_1 = T_{seq} + T_{par}$$

donde T_{seq} es el tiempo empleado en cálculos no paralelizables y T_{par} es el tiempo empleado en cálculos perfectamente paralelizables entre el número de procesadores disponibles.

El tiempo de ejecución en un multiprocesador con P procesadores es

$$T_p = T_{seq} + \frac{T_{par}}{P}$$

Entonces, el incremento de rendimiento que se obtiene al ejecutar el programa en un multiprocesador respecto de una ejecución serie se puede calcular utilizando la ley de Amdahl.

$$G = T_1 / T_p = \frac{1}{(1-\alpha) + \alpha/P}$$

donde α es la fracción de tiempo en una ejecución serie que se paraleliza.

Un multiprocesador puede operar en modo serie o paralelo. Suponga que en modo paralelo se utilizan 9 procesadores. Posteriormente se observa que el 25% de T_p se atribuye al modo paralelo. Durante el tiempo restante el programa se ha ejecutado en modo serie.

Pregunta 1: *Calcule la ganancia efectiva en las condiciones anteriores cuando se compara con una ejecución exclusivamente en modo serie. Además, calcule la fracción α de código que ha sido paralelizada.*

Pregunta 2: *Suponga que doblamos el número de procesadores. Calcule la ganancia efectiva que se obtiene.*

Pregunta 3: *Suponga que mediante técnicas de compilación y 9 procesadores se puede obtener la misma ganancia efectiva que en el apartado anterior, donde se había duplicado el número de procesadores. Determine la fracción α de código que ha sido paralelizada por el compilador.*

Ejercicio 2.4

El siguiente código resuelve un sistema triangular de ecuaciones.

```
do J=1,N
    X(J) = B(J) / A(J, J)
    do I = J+1, N                          bucle paralelo
        B(I) = B(I) - A(I,J) * X(J)
    enddo
enddo
```

En las siguientes preguntas considere sólo las dos sentencias de cálculo al evaluar el tiempo. Suponga que una sentencia de cálculo tarda un ciclo en ejecutarse.

Pregunta 1: *Calcule la fracción α de código que no se paraleliza.*

Pregunta 2: *Dibuje un perfil del paralelismo. Esto es, numero de cálculos paralelos en función de la variable de iteración J.*

Pregunta 3: *Suponga que se dispone de N procesadores. Calcule la ganancia respecto de una ejecución serie. Calcule también la eficiencia.*

Ejercicio 2.5

Al ejecutar una aplicación en un multiprocesador se distinguen 3 modos: a) se utilizan todos los procesadores, b) se utiliza la mitad de los procesadores y c) se utiliza un procesador. Suponga que el 2% del tiempo se ejecuta en un solo procesador y que hay 100 procesadores.

Pregunta 1: *Queremos obtener una ganancia de 80. Calcule la fracción máxima de tiempo en la que se solo se utilizan la mitad de los procesadores.*

Ejercicio 2.6

Considere la ejecución del siguiente código en un multiprocesador con 8 procesadores.

```
doall I = 1, N
    A(I) = A(I) + C(I)
enddo
```

Cada procesador tiene una cache que podemos considerar de tamaño infinito y con un tamaño de bloque de 32 bytes, ocupando cada elemento de los vectores 8 bytes. Solo existen fallos de carga o forzosos y no existen fallos de capacidad ni de conflicto debidos a la función de mapeo. En otras palabras, solo existe fallo de cache cuando se referencia por primera vez un dato si el bloque no está en la cache.

La red de interconexión entre la memoria y las caches privadas de los procesadores soporta un número ilimitado de transferencias en paralelo. En cuanto a la memoria es capaz de suministrar en paralelo un número ilimitado de peticiones de acceso. Es decir, no existen retardos debido a que varios procesadores soliciten acceso a memoria de forma concurrente.

Suponga un multiprocesador ideal donde las acciones de coherencia de cache se realizan en tiempo cero. Además las referencias de un procesador a memoria no están interferidas por los accesos de otros procesadores a variables almacenadas en el mismo bloque.

El tiempo de acierto en cache es de un ciclo de procesador. La penalización por fallo de cache son 100 ciclos de procesador.

Suponga que el tiempo de ejecución de las operaciones aritméticas y de las instrucciones de control del bucle es despreciable. Es decir, considere que el cuerpo del bucle son las instrucciones

```
load A(I)
load C(I)
store A(I)
```

Pregunta 1: *Calcule el tiempo de ejecución en un procesador.*

El bucle anterior se ejecuta en el multiprocesador utilizando 2 algoritmos de planificación estática.

A) Planificación simple: El bucle doall se convierte en los siguientes bucles

```
doall p=1, NP
  do I = (p-1)×⌈N/(NP)⌉+1, min(N, p×⌈N/(NP)⌉)
```

B) Planificación entrelazada: El bucle doall se convierte en los siguientes bucles

```
doall p=1, NP
    do I = p , N, NP
```

donde NP es el número de procesadores

Suponga que N es múltiplo del número de procesadores y que NP es igual a 8.

Pregunta 2: *Calcule, en el caso A, el tiempo de ejecución y la ganancia que se obtiene respecto a la ejecución en un procesador.*

Pregunta 3: *Calcule, en el caso B, el tiempo de ejecución y la ganancia que se obtiene respecto a la ejecución en un procesador.*

Pregunta 4: *Calcule la relación entre la ganancia en el caso A y el caso B y justifique el valor haciendo referencia a algún parámetro del multiprocesador.*

Ejercicio 2.7

Suponga un sistema multiprocesador con cinco procesadores. Al ejecutar un programa en el multiprocesador comprobamos que durante 1/6 del tiempo se han utilizado los 5 procesadores.

Pregunta 1: *Calcule la ganancia respecto a una ejecución serie.*

Pregunta 2: *Calcule el porcentaje de código (α) que ha sido paralelizado.*

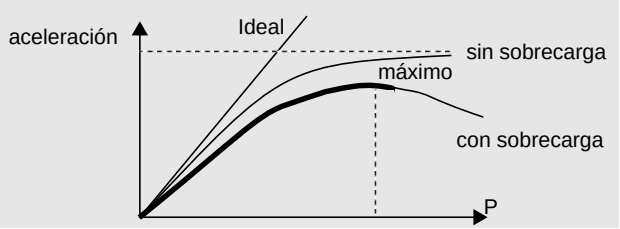
Pregunta 3: *Calcule la ganancia si suponemos que el código paralelo se puede ejecutar con un número ilimitado de procesadores.*

Suponga que se añaden 5 procesadores más. En estas condiciones el 80% del código paralelizable se puede ejecutar en 10 procesadores y el resto de código paralelizable se sigue ejecutando en 5 procesadores.

Pregunta 4: *Calcule la ganancia respecto a una ejecución serie.*

Pregunta 5: *Utilizando solo 5 procesadores, calcule el porcentaje de código (α_1) que debe ser paralelizado (mejora de la capacidad de paralelización del compilador) para obtener la misma ganancia que en el apartado 4).*

Suponga que el coste (sobrecarga) de iniciar los procesos en los procesadores y otros efectos debidos a la sincronización y comunicación determinan que hay que considerar un coste adicional en el tiempo de ejecución paralelo.



Pregunta 6: *En las condiciones de la pregunta 5) (α_1) y suponiendo que la sobrecarga, por el concepto expuesto anteriormente, es 0.005 veces el tiempo de ejecución en serie por el número de procesadores (0.005 x P x T_{serie}), calcule el número de procesadores con el que se obtiene la máxima ganancia.*

Nota: La derivada del producto de dos funciones es igual al primer factor por la derivada del segundo más el segundo factor por la derivada del primero. La derivada del cociente de dos funciones es igual a la derivada del numerador por el denominador menos la derivada del denominador por el numerador, divididas por el cuadrado del denominador.

Ejercicio 2.8

Para dar soporte a la sincronización se han diseñado dos instrucciones específicas denominadas Load Linked (LL) y Store Conditional (SC).

Load Linked (LL rd, (rf)): además de efectuar el load, tiene el efecto lateral de activar un bit denominado bit de enlace (bit LL) que se asocia al bloque de cache en el cual se mapea la dirección.

Este bit forma un enlace rompible entre la instrucción LL y la siguiente instrucción SC (store condicional).

La instrucción SC (SC rfd, (rf)), cuando se ejecuta, efectúa un store simple si el bit de enlace (LL bit) está activado. Si el bit de enlace no está activado, entonces el store no se ejecuta. La ejecución o no del store se indica en el registro (rfd) de la instrucción SC. Por tanto, este registro es fuente y destino.

El bit de enlace (bit LL) se desactiva si, mientras se ejecuta la secuencia de código entre LL y SC, se produce cualquier evento que potencialmente modifica el bloque accedido utilizando LL. Esto puede ser debido, entre otras causas: a) a una actualización externa del bloque que contiene la variable accedida mediante LL, b) a una invalidación externa del bloque que contiene la variable accedida mediante LL.

Codifique las siguientes operaciones atómicas utilizando las instrucciones LL y SC.

Pregunta 1: *Test&set (llave).*

test&set (llave)	comentarios
tmp = llave	llave es una variable global
llave = 1	
return tmp	

Pregunta 2: *Fetch&inc (dir).*

fetch&inc (dir)	comentario
tmp = dir	dir es una variable global
dir = tmp + 1	
return tmp	

Pregunta 3: *Compare&swap (viejo, nuevo, dir).*

Compare&swap (viejo, nuevo, dir)	comentario
tmp = dir	dir es una variable global
if (tmp = viejo) then	nuevo y viejo son valores almacenados en variables locales
dir = nuevo	
z =1	
else	
viejo = dir	
z =0	
endif	
return z	

Pregunta 4: *Fetch&and (dir, a).*

fetch&and (dir, a)	comentario
tmp = dir	dir es una variable global
dir = tmp and a	a es un valor almacenado en una variable local
return tmp	

Pregunta 5: *Sawp (dir, a).*

swap (dir, a)	comentario
tmp = dir	dir es una variable global
dir = a	a es un valor almacenado en una variable local
return tmp	

Ejercicio 2.9

En el siguiente programa paralelo la actualización de la variable global Total se efectúa creando una zona de exclusión mediante las primitivas atómicas obtener y liberar.

```
doall II =1 , P
   do I = II, N, P
      local (II) = local (II) + X(I)
   enddo
   obtener (llave)
      Total = Total + local (II)
   liberar (llave)
enddo
```

Modifique el programa de forma que se utilice la instrucción indivisible compare&swap para efectuar la actualización de la variable Total.

Compare&swap (viejo, nuevo, dir)	comentario
tmp = dir	dir es una variable global
if (tmp = viejo) then	nuevo y viejo son valores almacenados en variables locales
dir = nuevo	
z =1	
else	
viejo = dir	
z =0	
endif	
return z	

Ejercicio 2.10

El siguiente algoritmo efectúa una planificación del bucle por trozos de tamaño fijo.

```
region critica
  LI = GI
  GI = GI + K
end region critica
While (LI < N)
  do I = LI, min (LI+K-1, N)
    . . .
  enddo
  region critica
    LI = GI
    GI = GI + K
  end region critica
endwhile
```

comentario
GI: variable global que indica la primera iteración no ejecutada
K: número fijo de iteraciones asignadas. El valor ha sido determinado antes de iniciar la ejecución paralela.
LI: variable local que indica la primera iteración que ejecuta el hilo
GI está inicializado con el valor de la primera iteración del bucle
N es el número de iteraciones del bucle

Los procesadores disponen de las instrucciones atómicas fetch&add y compare&swap.

fetch&add (dir, a)	comentario
tmp = dir	dir es una variable global
dir = tmp + a	a es un valor almacenado en una variable local
return tmp	

Compare&swap (viejo, nuevo, dir)	comentario
tmp = dir	dir es una variable global
if (tmp = viejo) then	nuevo y viejo son valores almacenados en variables locales
dir = nuevo	
z =1	
else	
viejo = dir	
z =0	
endif	
return z	

Pregunta 1: *Reescriba el algoritmo utilizando la primitiva fetch&add.*

Pregunta 2: *Reescriba el algoritmo utilizando la primitiva compare&swap.*

Ejercicio 2.11

La operación de sincronización barrera se puede implementar utilizando la primitiva fetch&inc mediante el siguiente código.

barrera (bar)	inicialización	comentario
fetch&inc (bar) repeat until bar = N return	variable global bar = 0	N es el número de hilos que ejecutan la operación barrera

Un programador quiere reutilizar la posición de memoria bar en sucesivas utilizaciones de la operación barrera, de la forma que se muestra en los siguientes códigos paralelos (parte izquierda). Para ello ha diseñado el siguiente código de la operación barrera (parte derecha).

código paralelo A	código paralelo B	barrera (bar)
bar = 0	do = . . .	if (fetch&inc (bar) = N-1) then
doall I = 1, N	doall = . . .	bar = 0
. . .	. . .	else
enddoall	endoall	repeat
barrera (bar)	barrera (bar)	until bar = 0
doall I = 1, N	enddo	endif
. . .		return
enddoall		
barrera (bar)		
. . .		

Pregunta 1: *Muestre que esta implementación de la operación barrera no funciona correctamente.*

Nota: suponga que algún hilo llega a la segunda barrera antes de que el resto de hilos hayan acabado de ejecutar la primera barrera.

Pregunta 2: *Proponga una implementación de la operación barrera que funcione correctamente en el ejemplo anterior. Especifique si las variables utilizadas son globales o locales.*

Nota: desacople la espera activa de la variable que se utiliza para contabilizar los hilos que han llegado a la barrera. Además, la nueva variable debe actuar como un conmutador entre dos instancias de la operación barrera.

Ejercicio 2.12

En el diseño de un algoritmo de planificación de bucles se tiene en consideración la siguiente característica: los hilos del bucle paralelo compiten con otros procesos o hilos en el sistema multiprocesador y es posible que no estén disponibles todos los procesadores necesarios en el mismo instante de tiempo. Por tanto, los hilos inician la ejecución en tiempos distintos.

Para balancear la carga de trabajo la idea es asignar un número de iteraciones variable. El primer trozo que se asigna tiene un tamaño $\lceil N/P \rceil$, siendo N el número de iteraciones. Los siguientes trozos se van decrementando hasta que no quedan iteraciones. Este tipo de planificación se denomina guiada.

Una acción de planificación determina la asignación de un número de iteraciones K_i a un hilo, considerando que el resto de hilos (P-1) también se planifica en el mismo instante. Sea R_i el número de iteraciones que quedan por planificar, entonces en la siguiente planificación se asignan $K_i = \lceil R_i/P \rceil$ iteraciones a un hilo, en la siguiente planificación se asignan $K_i = \lceil R_{i+1}/P \rceil$ a otro hilo, donde $R_{i+1} = R_i - K_i$ y así sucesivamente. Inicialmente tenemos $R_0 = N$.

Pregunta 1: *Calcule para los siguientes valores la asignación de iteraciones: a) N=100, P =5 y b) N=1000, P =4.*

Pregunta 2: *Dibuje un diagrama de tiempos para el caso de la primera pregunta, suponiendo que los procesadores empiezan a trabajar en los siguientes instantes de tiempo y que cada iteración equivale a una unidad de tiempo.*

P1	P2	P3	P4	P5
0	5	3.5	10	17.5

Otros algoritmos que se utilizan para planificar bucles paralelos son: autoplanificación y planificación por trozos de tamaño fijo. Por autoplanificación se entiende que un hilo obtiene una iteración para ejecutar y al finalizar la ejecución compite con otros hilos por obtener otra iteración. Por planificación por trozos de tamaño fijo se entiende que el número total de iteraciones se distribuye entre los hilos antes de empezar la ejecución paralela. En este último caso suponga que a un hilo se le asignan iteraciones contiguas.

Pregunta 3: *Compare planificación guiada con autoplanificación y planificación por trozos de tamaño fijo. Razone sobre el número de planificaciones y la distribución de la carga.*

Pregunta 4: *Diseñe un algoritmo que implemente planificación guiada al ejecutar un bucle. Para ello dispone de las primitivas atómicas obtener (llave) y liberar (llave). Los valores de la primera y última iteración son 1 y N respectivamente.*

Ejercicio 2.13

En el diseño de un algoritmo de planificación de bucles se tiene en consideración la siguiente característica: en el cuerpo del bucle existen condicionales y existe una alta probabilidad de que los trozos asignados a los procesadores finalicen antes del tiempo previsto.

La idea, utilizada en la planificación, es dejar suficiente trabajo para los trozos que finalizan antes (alisar). A este tipo de planificación se le denomina factorización.

Las iteraciones se asignan en P trozos de igual tamaño K_i. El tamaño del trozo se calcula mediante la siguiente expresión $K_i = \lceil R_i/(2 \times P) \rceil$, siendo $R_{i+1} = R_i - P * K_i$, donde $R_0 = N$ y N es el número de iteraciones.

Pregunta 1: *Calcule para el siguiente caso la asignación de iteraciones: N=100, P = 5.*

Pregunta 2: *Escriba el esqueleto de un programa que muestra la planificación de un bucle utilizando planificación mediante factorización. Para ello dispone de las primitivas atómicas obtener (llave) y liberar (llave).*

Ejercicio 2.14

Un programa paralelo utiliza P procesadores de un multiprocesador. El encargado de recursos informáticos se plantea, manteniendo el mismo tiempo de ejecución, reducir el consumo de energía, reduciendo la frecuencia y utilizando más procesadores. Supondremos para simplificar que el CPI medio de los procesadores al reducir la frecuencia no se modifica (se mantiene el número de ciclos en fallo).

Sea α la fracción de código paralelizable medida en una ejecución serie del programa. La reducción en frecuencia representa multiplicar por r > 1 el tiempo de ciclo.

Pregunta 1: *Desarrolle una expresión que evalúe el número de procesadores necesarios (PP) en función de α, r y P.*

Pregunta 2: *Calcule la ganancia en energía (C x V^2) en función de P, PP y r. Para ello suponga que la tensión de alimentanción se reduce $\beta = 1 - (1/r)$ al reducir la frecuencia en la misma magnitud. Suponga también, para simplificar, que todos los procesadores consumen la misma energía durante todo el tiempo de ejecución.*

Seguidamente se quiere evaluar la influencia del tiempo de iniciación de los hilos del programa al ejecutarlo en paralelo. Para simplificar, supondremos que el programa es totalmente paralelizable y que el tiempo de iniciación de un hilo es T_i. Notemos que los hilos se inician de forma serie.

Pregunta 3: *Calcule el número máximo de procesadores (P) a partir del cual el tiempo de ejecución en paralelo se incrementa (mínimo tiempo de ejecución). El tiempo de ejecución en serie del programa es T_S.*

Pregunta 4: *Calcule el número de procesadores (P) a partir del cual el tiempo de ejecución en paralelo es mayor que el tiempo en serie (T_S). Suponga que $P >> 1$ y que $T_i << T_S$. El resultado no es infinito.*