Multiprocesadores y multicomputadores

Daniel Jiménez-González

PID_00184815

Universitat Oberta de Catalunya

Índice

Introducción

A pesar de que los uniprocesadores han ido incorporando soporte *hardware* para explotar los diferentes niveles de paralelismo y mejorar el rendimiento de las aplicaciones (paralelismo a nivel de instrucción o ILP, a nivel de datos o DLP, y a nivel de *threads* o TLP), los *speedups* alcanzados en estas aplicaciones no han sido significativamente grandes, ya sea por motivos tecnológicos o bien por las necesidades concretas de las aplicaciones.

Otra vía para conseguir *speedups* más significativos es crear sistemas de computación más grandes aprovechando la mejora tecnológica y combinando múltiples uniprocesadores, los cuales pueden aplicar o no la misma instrucción sobre múltiples datos. A los sistemas que aplican una misma instrucción a multiples datos se les llama computadores SIMD* y, en el caso que apliquen diferentes instrucciones a los datos, reciben el nombre de MIMD**.

*** SIMD:** ***Single Instruction Multiple Data.***
**** MIMD:** ***Multiple Instruction Multiple Data.***

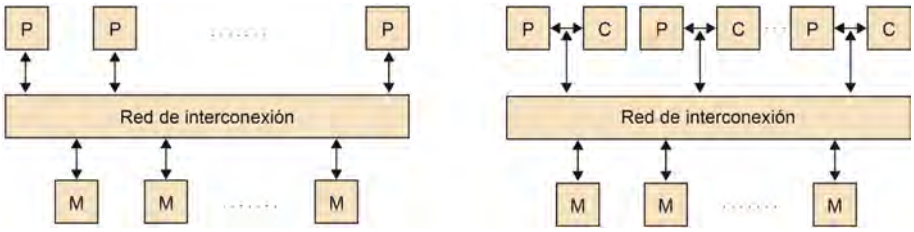
Las máquinas SIMD dieron respuesta a aplicaciones científicas y de ingeniería que realizaban operaciones sobre estructuras muy regulares, como pueden ser *arrays* o vectores. Los computadores paralelos *array processors* y *vector processors* son los más conocidos. La característica más importante de este tipo de máquinas es que aplican una misma operación (instrucción) sobre una secuencia de datos, ya sea teniendo una serie de procesadores o unidades funcionales que operan a la vez (*array processors*), o bien con una unidad funcional que opera secuencialmente sobre una secuencia de datos (*vector processors*).

Las máquinas o computadores MIMD combinan múltiples procesadores que no necesariamente tienen que realizar la misma instrucción u operación sobre los datos. Estos sistemas con múltiples procesadores se suelen clasificar según si tienen memoria compartida o sólo distribuida, siendo los de memoria compartida los multiprocesadores, y los de memoria distribuida, los multicomputadores. A los multiprocesadores también se les suele llamar fuertemente acoplados, y a los multicomputadores, débilmente acoplados.

Los multiprocesadores se caracterizan por el hecho de que todos los procesadores tienen un espacio virtual de direcciones de memoria común. En éstos, los procesadores sólo necesitan realizar un `load`/`store` para acceder a cualquier posición de la memoria. De esta forma, los procesadores pueden comunicarse simplemente a través de variables compartidas, facilitando así la programación. Los multiprocesadores más conocidos son los llamados *symetric multiprocessors* o SMP, que se caracterizan por ser todos los procesadores del mismo tipo y poder acceder de la misma forma a los sistemas de memoria y entrada/salida. La figura 1 muestra dos esquemas básicos de la arquitectura de un multiprocesador, donde todos los procesadores comparten la memoria principal. En la parte izquierda de la figura se muestra un sistema en el que los procesadores no tienen caché y, a la derecha, un sistema en el que sí tienen caché. Las cachés se introdujeron para explotar mejor la localidad de los datos, y como consecuencia, reducir la contención de acceso a la memoria compartida. Sin embargo, estas cachés introducen un problema de coherencia

entre las posibles copias de un dato concreto en las diferentes cachés, que analizaremos en este módulo. En esta misma figura, independientemente de si el sistema tiene cachés o no, podemos observar que hay una red de interconexión que conecta los procesadores con la memoria. Esta red puede ser un simple bus o algo más complejo.

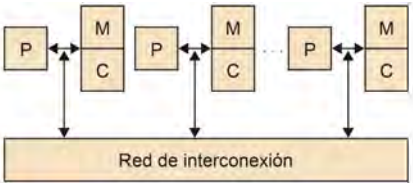
Figura 1. Multiprocesador de memoria compartida sin caché (izquierda) y con caché (derecha).



En los sistemas multiprocesador suele haber un único sistema operativo con una única tabla de páginas y una tabla de procesos. Sin embargo, puede ser que el sistema operativo tenga que trabajar con más de una tabla de páginas. Esto ocurre con los multiprocesadores de tipo *distributed shared memory* o DSM, en los que cada nodo de la máquina tiene su propia memoria virtual y su propia tabla de páginas. En estos sistemas, una página está localizada en una de las memorias asociadas a un nodo. En el momento de un fallo de página, el sistema operativo pide la página al procesador que la tiene local para que se la envíe. En realidad, la gestión es como un fallo de página en la que el sistema operativo tiene que buscar la página en disco.

Los multicomputadores, tal y como hemos comentado, se caracterizan por tener la memoria distribuida. En estos sistemas los procesadores tienen espacios físicos y virtuales diferentes para cada procesador. Por consiguiente, cada procesador puede acceder con *loads* y *stores* a su memoria local, pero no a las memorias de otros procesadores. Así, si un procesador debe acceder a un dato localizado en la memoria de otro procesador, éstos se tendrán que comunicar vía mensaje a través de la red de interconexión. Esto dificulta notablemente la programación de estos sistemas. Sin embargo, aun teniendo esta desventaja significativa con respecto a los multiprocesadores, los sistemas multicomputadores tienen la gran ventaja de que son mucho más económicos de montar que los multiprocesadores, ya que se pueden construir conectando procesadores de carácter general a través de una red de interconnexión estándar, sin necesidad de ser a medida. Esto, además, los hacen más escalables. La figura 2 muestra un sistema multicomputador, donde cada procesador tiene su memoria local cercana y una conexión a la red de interconexión para poder comunicarse con el resto de procesadores, y así, poder acceder a los datos de otras memorias.

Figura 2. Esquema básico de memoria de un multicomputador.



Finalmente, nos podemos encontrar con sistemas MIMD híbridos: un sistema multicomputador con memoria distribuida entre los diferentes nodos que lo forman, conectados a través de una red de interconexión, y dentro de cada nodo, tener un sistema multiprocesador con memoria compartida, normalmente con una conexión basada en bus. Estos sistemas combinan las ventajas antes mencionadas: programabilidad y escalabilidad. La organización del módulo es la siguiente: En el apartado 1 realizaremos una clasificación de los computadores SIMD y MIMD. Entraremos en más detalle en los sistemas multiprocesador y multicomputador en los apartados 2 y 3 respectivamente. En el apartado 2, subapartado 2.1, analizaremos las conexiones típicas a la memoria, que nos podemos encontrar en estos sistemas. Posteriormente, describiremos los problemas derivados de tener diferentes bancos de memoria accesibles en paralelo, y de la incorporación de las cachés en los sistemas multiprocesadores: el problema de la consistencia en el subapartado 2.2 y el de la coherencia en el subapartado 2.3, respectivamente.

En el apartado 3 describiremos los sistemas multicomputador. En el subapartado 3.1 de éste veremos algunas de las redes de interconexión usadas en la conexión de los procesadores en estos sistemas. Finalmente, en el subapartado 3.2, analizaremos algunas de las comunicaciones colectivas utilizando un modelo de comunicación básico.

Objetivos

Los objetivos generales de este módulo didáctico son los siguientes:

1. Conocer las características de las principales arquitecturas SIMD y MIMD.
2. Conocer las diferentes redes de interconexión para los multiprocesadores (memoria compartida) y multicomputadores (memoria distribuida).
3. Saber distinguir entre el problema de consistencia de memoria y de coherencia de memoria.
4. Conocer los diferentes modelos de consistencia de memoria.
5. Conocer los diferentes mecanismos y protocolos para mantener la coherencia de memoria.
6. Conocer y saber utilizar las diferentes métricas de caracterización de las redes de interconexión.
7. Saber medir el coste de colectivas de comunicación en diferentes topologías de red.

1. Clasificación

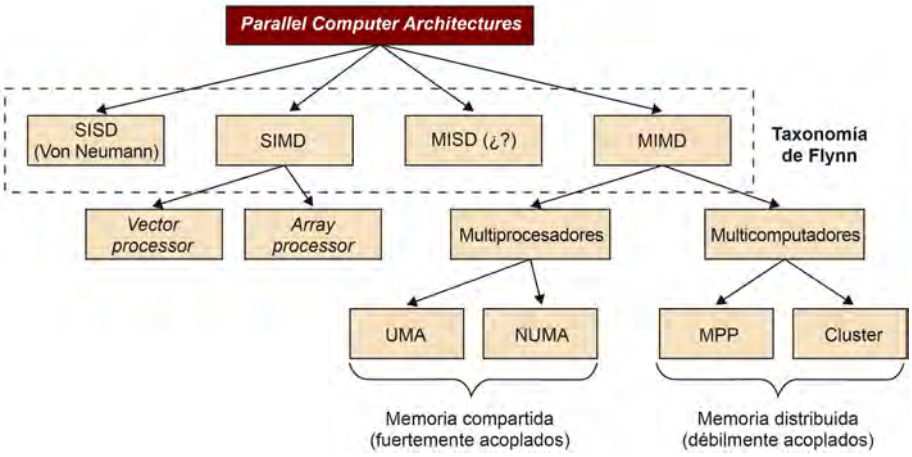
La taxonomía de Flynn clasifica los computadores según cómo y con cuántas instrucciones se procesan los datos. Esta clasificación distingue si un mismo dato o datos diferentes son procesados por una única instrucción o instrucciones diferentes, obteniendo así cuatro tipos diferentes de arquitecturas: SISD, SIMD, MISD y MIMD.

Taxonomía de Flynn

SISD: *single instruction single data*.
SIMD: *Single instruction multiple data*.
MISD: *Multiple instruction single data*.
MIMD: *Multiple instruction multiple data*.

En este apartado nos centraremos en las arquitecturas *Single instruction multiple data* y *Multiple instruction multiple data*. Primero describiremos brevemente las arquitecturas *Single instruction multiple data*, para después entrar en detalle en las arquitecturas *Multiple instruction multiple data*. En la figura 3 se muestra la clasificación realizada por Flynn, y unas subcategorías que clasifican según el procesamiento de los datos y la forma de compartir estos datos, que detallamos a continuación.

Figura 3. Taxonomía de Flynn y otras subcategorías.



1.1. Arquitecturas paralelas SIMD

Las arquitecturas paralelas SIMD las podemos clasificar en *array processors* y *vector processors*.

1.1.1. *Array processors*

Los *array processors* básicamente consisten en un gran número de procesadores idénticos que realizan la misma secuencia de instrucciones en datos diferentes. Cada procesador, en paralelo con el resto de procesadores, realiza la misma instrucción sobre los datos que le toca procesar. El primer *array processor* fue el computador ILLIAC IV de la Universidad de Illinois, aunque no se pudo montar en toda su totalidad por motivos económicos.

ILLIAC IV

Fue creada por la Universidad de Illinois en 1976.

Notad que los *array processors* son la semilla de las extensiones SIMD de los uniprocesadores.

1.1.2. *Vector processors*

Los *vector processors*, desde el punto de vista del programador, son lo mismo que los computadores *array processors*. En cambio, el procesamiento de los datos no se hace en paralelo. Los datos son procesados por una única unidad funcional muy segmentada. La empresa que más *vector processors* ha realizado es Cray Research (ahora parte de SGI). La primera máquina *vector processor* fue la Cray-1.

Cray-1

Fue instalada en los Alamos National Laboratory en 1976.

1.2. Multiprocesadores

Los multiprocesadores son computadores que tienen memoria compartida y se pueden clasificar en UMA (*uniform memory access*), NUMA (*nonuniform memory access*) y COMA (*cache only memory access*).

Los nombres que reciben las arquitecturas multiprocesador UMA y NUMA tienen que ver con el tiempo de acceso a la memoria principal, no teniéndose en cuenta la diferencia de tiempo entre un acierto o un fallo en caché. De lo contrario, también deberíamos considerar arquitectura NUMA a cualquier sistema con jerarquía de memoria, incluidos los uniprocesadores.

En cuanto a la arquitectura COMA, que no tuvo mucho éxito, se basa en tener la memoria compartida como si fuera una gran caché.

A continuación describiremos cada una de estas arquitecturas.

1.2.1. UMA

En este tipo de arquitectura, como bien dice su nombre, todos los accesos a memoria tardan el mismo tiempo. Seguramente podemos pensar que es difícil que tengamos el mismo tiempo de acceso si la memoria, aunque compartida, está dividida en módulos a los que se accede a través de una red de interconexión basada en *switches*. Eso es cierto y, para conseguir esta uniformidad de acceso, se tiene que aumentar el tiempo de los accesos más rápidos.

Switch

Es un dispositivo que interconecta dos o más segmentos de red (*links*), pasando datos de un segmento a otro según la configuración de conexiones y la dirección que deba tomar el dato a transferir.

¿Por qué se busca esta uniformidad de tiempo de acceso? Porque para los programadores es más fácil deducir qué parámetros ayudarán a mejorar sus programas si el tiempo de acceso es igual para cualquier acceso. De lo contrario, como pasa con las arquitecturas NUMA*, deberían ser mucho más conscientes de la arquitectura que tiene el computador y de cómo están distribuidos los datos en la arquitectura en cuestión.

*** Los sistemas NUMA facilitan la programación, ya que el tiempo de acceso siempre es el mismo.**

La figura 4 muestra un sistema multiprocesador donde los procesadores se conectan a la memoria compartida utilizando un bus (dato y direcciones). Este tipo de máquinas son fáciles de construir. Sin embargo, el hecho de que todos los procesadores accedan al mismo bus para acceder a memoria puede significar un cuello de botella. Una alternativa a tener un único bus sería tener varios buses, de tal forma que se distribuirían los accesos a los módulos de memoria. También se podría usar una red de interconexión más compleja, como una red *crossbar*, que permita evitar conflictos entre los accesos a las memorias que no van al mismo módulo, tal y como se muestra en la figura 5.

Conexiones mediante bus

Las conexiones a través de un bus son fáciles de construir pero pueden tener mayor contención de acceso a memoria que otras redes de interconexión.

Figura 4. Multiprocesador de memoria compartida a través de un bus.

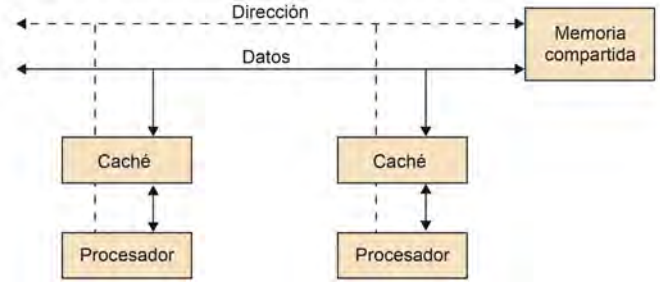
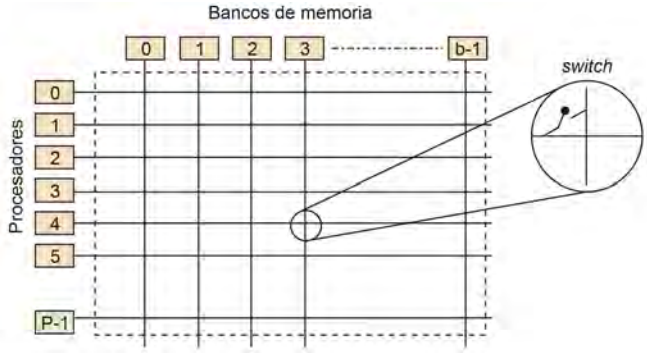


Figura 5. Multiprocesador de memoria compartida a través de un *crossbar*.



Por otra parte, una forma de reducir la contención de los accesos a memoria* es reduciendo la cantidad de estos accesos por parte de los procesadores, y se puede conseguir incorporando jerarquías de memoria en los mismos. De esta forma, si los programas explotan la localidad de datos, los accesos a memoria se reducirán.

*** La jerarquía de memoria ayuda a reducir la contención de acceso a memoria si se explota la localidad de datos.**

Las máquinas UMA se hicieron más y más populares desde aproximadamente el año 2000, año en el que podríamos decir que hubo un punto de inflexión en la tendencia de los diseños de las arquitecturas de los uniprocesadores, debido básicamente al consumo energético y a la consecuente bajada de rendimiento en las aplicaciones. La aparición de los procesadores CMP o *chip multi-processors* fueron el resultado natural de intentar aprovechar las mejoras tecnológicas existentes, y aprovechar así mejor el área del chip, sin aumentar considerablemente el consumo energético.

Los procesadores CMP son procesadores de tipo UMA que incorporan más de una unidad de procesamiento dentro de un único chip. Algunos ejemplos con unidades de procesamiento homogéneas (todas las CPU son iguales) son los Intel Core2 Duo (con 2 CPU), Intel Nehalemi7 (con 4 CPU), AMD Istanbul (con 6 CPU), Sun SPARC64-VIIIfx (con 8 CPU), IBM Power7 (con 8 CPU), etc. Pero también existen con unidades heterogéneas,

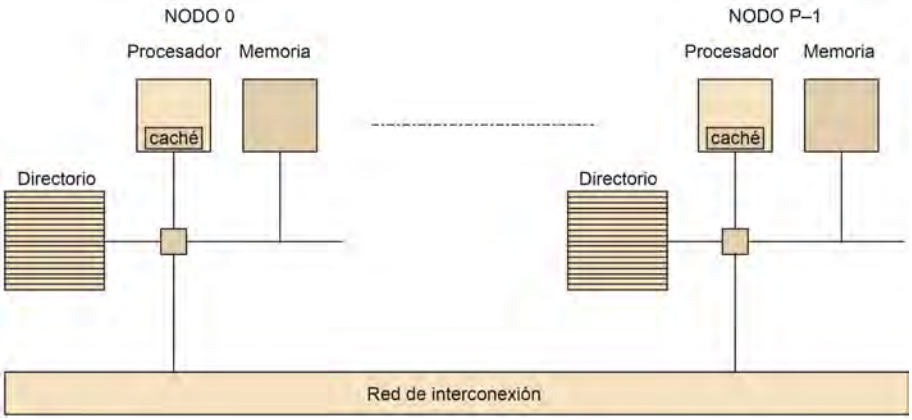
como es el caso del Intel Sandy Bridge, que tiene un procesador de carácter general y una unidad de procesamiento gráfico.

1.2.2. NUMA

En los multiprocesadores NUMA, a diferencia de los UMA, los accesos a memoria pueden tener tiempos distintos. En estas máquinas la memoria también está compartida, pero los módulos de memoria están distribuidos entre los diferentes procesadores con el objetivo de reducir la contención de acceso a memoria.

El módulo de memoria que está junto a un procesador en un mismo nodo recibe el nombre de memoria local a este procesador. Así, los accesos de un procesador a su memoria local suelen ser mucho más rápidos que los accesos a la memoria local de otro procesador (memoria remota). La figura 6 muestra un esquema básico de la distribución de memoria en este tipo de multiprocesadores, donde se observa la distribución de los módulos entre los diferentes nodos. Estos nodos tienen un bus local para acceder a la memoria local, y se conectan a la red de interconexión para acceder a la memoria situada en otros nodos. También observamos un *hardware* llamado directorio que sirve para mantener la coherencia de los datos y que explicaremos, más adelante, en el módulo. La redes de interconexión típicas de estos sistemas son las redes de tipo *tree* y las de bus jerárquico.

Figura 6. Multiprocesador NUMA de memoria compartida.



Desde el punto de vista del programador, éste deberá ser consciente de la arquitectura NUMA para poder acercar los datos a la memoria local del procesador que tenga que operar con esos datos. De esta forma se podrán reducir los accesos a memorias locales de otros procesadores (memorias remotas) y, como consecuencia, se mejorará el tiempo de ejecución de la aplicación.

Mapeo de datos

El programador deberá ser consciente del mapeo de los datos para reducir el número de accesos a la memoria local de otros procesadores.

Por otra parte, al igual que pasaba con las arquitecturas UMA, también se incorporaron las cachés para reducir la contención de memoria. Pero además, con estas cachés también se intenta ocultar la diferencia de tiempo de acceso a memoria entre memoria local y remota. Según tengan o no caché, los multiprocesadores NUMA se clasifican en *cache coherence* NUMA (ccNUMA) o *Non-Coherence* NUMA respectivamente. El término de coherencia o no coherencia proviene de mantenerla o no en los datos duplicados en las cachés.

Coherencia de los datos

Mantener la coherencia de los datos significa que el último valor de un dato debe ser visto por todos los procesadores.

Un ejemplo de multiprocesador ccNUMA es la SGI Altix UV 1000, que está formado por 224 procesadores Intel Xeon X7542 de 64 bits, a 2,66 GHz de 6 núcleos, con un total de 1.344 núcleos de cálculo. Otros ejemplos de máquinas NUMA son la SGI Origin 3000, el Cray T3E y el AMD Opteron.

1.2.3. COMA

Aunque las arquitecturas ccNUMA ayudan a ocultar la diferencia de tiempo entre accesos locales y remotos, esto dependenderá de la localidad de datos que se explote y la capacidad de las cachés. Así, en el caso de que el volumen de los datos a los que se accede sea mayor que la capacidad de la caché, o el patrón de acceso no ayude, volveremos a tener fallos de caché y el rendimiento de las aplicaciones no será bueno.

Arquitecturas COMA

Las arquitecturas *cache only memory access* tratan la memoria principal como una memoria caché grande.

Las arquitecturas COMA (*cache only memory access*) intentan solucionar este problema haciendo que la memorial local a cada procesador se convierta en parte de una memoria caché grande. En este tipo de arquitecturas, las páginas de memoria desaparecen y todos los datos se tratan como líneas de caché. Sin embargo, con esta estrategia surgen nuevos interrogantes: ¿cómo se localizan las líneas de caché? y cuando una línea se tiene que reemplazar, ¿qué sucede si esta es la última? Estas preguntas no tienen fácil solución y pueden requerir de soporte *hardware* adicional. Normalmente se implementan utilizando un mecanismo de directorio distribuido.

Así, aunque este tipo de máquinas prometían mejor rendimiento que las ccNUMA, se han construido pocas. Las primeras COMA que se construyeron fueron la KSR-1 y la *data diffusion machine*.

1.3. Multicomputadores

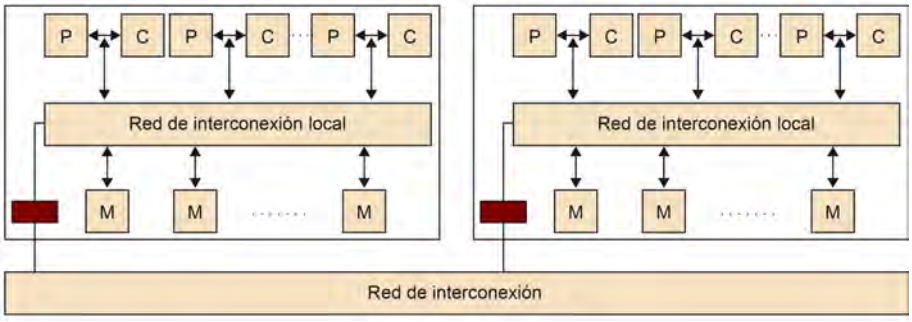
La característica más importante de estos sistemas es que tienen la memoria distribuida. Esta arquitectura también es conocida como arquitectura basada en el paso de mensajes, ya que los procesos deben realizar comunicaciones (mensajes) a través de la red de interconexión para poder compartir datos. Estas máquinas, por consiguiente, no están limitadas por el ancho de banda de memoria, sino más bien por el de la red de interconexión.

La figura 7 muestra el esquema básico de un multicomputador donde cada nodo tiene parte de la memoria distribuida. Los nodos pueden contener desde un simple uniprocesador como un sistema multiprocesador.

1.3.1. MPP

Los sistemas MPP (*massively parallel processors*) son multicomputadores que tienen sus CPU conectadas con una red de interconexión estática de altas prestaciones (baja latencia

Figura 7. Esquema básico de un multicomputador donde cada nodo es un multiprocesador.



y elevado ancho de banda) especialmente diseñada para el sistema. Son sistemas grandes, en comparación con un sistema CMP, pero que no suelen tener un número muy elevado de CPU debido al coste económico que supondría mantener una red de interconexión de altas prestaciones. En general, son sistemas difícilmente escalables.

Las características principales de los sistemas MPP son:

- Utilizan microprocesadores estandard, pero también pueden incorporar *hardware* específico o aceleradores (por ejemplo, ASIC*, FPGA**, GPU).

- La red de interconexión que incorporan es propietaria, especialmente diseñada, con muy baja latencia y un ancho de banda elevado.

- Suelen venir con *software* propietario y librerías para gestionar la comunicación.

- Tienen una capacidad de almacenamiento de entrada/salida elevada, ya que se suelen utilizar para trabajar con grandes volúmenes de datos que se han de procesar y almacenar.

- Tienen mecanismos de tolerancia de fallos del *hardware*.

*** *Hardware* específico para el desarrollo de algun proceso concreto.**
**** *Field programmable gate array*: *hardware* que se puede programar usando lenguajes de descripción *hardware*.**

Estos sistemas MPP normalmente se han utilizado en cálculos científicos, pero también se han usado de forma comercial. La serie de *connection machines* es el ejemplo típico de máquinas masivamente paralelas.

Otras máquinas MPP como el Blue Gene (#1 en el Top500 en el 2006), implementada sobre un Toroide 3D ($32 \times 32 \times 64$), el Red Strom, basada en AMD Opterons, y Anton utilizan algunos ASIC para acelerar cálculos y comunicaciones. En particular, la máquina Anton fue especialmente diseñada para resolver problemas de *molecular dynamics* de forma muy rápida.

En la tabla 1 mostramos algunos ejemplos importantes de las primeras redes de interconexión utilizadas en sistemas MPP y sus características principales.

Tabla 1. Redes de interconexión MPP para diferentes máquinas: número de nodos que permiten, topología de red, ancho de banda *bisection bandwidth* que soportan y año de aparición.

Nombre	# de nodos	Topología	Ancho de banda (*link*)	*Bisection bandwidth* (MB/s)	Año
ncube/ten	1-1024	10-cube	1.2	640	1987
iPSC/2	16-128	7-cube	2	345	1988
MP-1216	32-512	2D grid	3	1.300	1989
Delta	540	2D grid	40	640	1991
CM-5	32-2048	fat tree	20	10.240	1991
CS-2	32-1024	fat tree	50	50.000	1992
Paragon	4-1024	2D grid	200	6.400	1992
T3D	16-1024	3D Torus	300	19.200	1993

1.3.2. *Clusters computers*

Los clusters consisten en cientos (o miles) de PC o estaciones de trabajo autónomas (*stand-alone**), posiblemente heterogéneas, conectadas por medio de una red comercial. Estas redes comerciales, que en un principio eran menos competitivas que las interconexiones propietarias en los sistemas MPP, han ido evolucionando hasta convertise en redes con muy buenas prestaciones. Estas redes normalmente son *TCP/IP packet switched* sobre LAN de cobre, WAN, fibra, *wireless*, etc. La diferencia básica con respecto a los sistemas MPP son que estos tenían nodos de alto rendimiendo (*high profile*), interconectados con redes estáticas de alta tecnología, y especialmente diseñadas.

Las principales ventajas de los clusters son las siguientes: son muy escalables (en coste por procesador), se pueden ampliar gradualmente, tienen mayor disponibilidad por la redundancia en número de procesadores, y salen rentables debido a su buena ratio coste por beneficio, ya que utilizan *comodities**.

Las tecnologías más usadas en las redes de interconexión de los clusters son: la WAN, la MAN, la LAN y la SAN. Las redes WAN se caracterizan por poder conectar ordenadores en grandes territorios y por tener unas ratios de transmisión de datos que van normalmente de 1,200 bit/s a 24 Mbit/s, aunque con ciertos protocolos se han alcanzado los 155 Mbit/s (*ATM protocol*). La tecnología LAN está normalmente limitada a una cierta área, como puede ser una casa, un laboratorio o quizás un edificio, y pueden alcanzar transferencias de datos mas elevadas que las WAN. Las redes MAN o *metropolitan area network* están entre las LAN y las WAN en cuanto a rendimiento y área que pueden abarcar, pudiendo seguir los mismos protocolos que las redes WAN. Finalmente, las redes SAN o *storage area network* están más dedicadas al almacenamiento de datos.

En cuanto a los protocolos de red que normalmente se utilizan, encontramos los de Internet o normalmente conocidos como TCP/IP (de *transmission control protocol* [TCP] e *Internet protocol* [IP]) y los ATM y FDDI (no tan utilizado este último desde que aparecieron las redes de Ethernet rápidas).

*** *Stand-alone* quiere decir que es autónomo.**

LAN

Local area network (LAN) es una red que conecta procesadores en un espacio limitado como la casa, la escuela, el laboratorio de cálculo, o un edificio.

WAN

wide area network (WAN) es una red de comunicaciones que cubre un área de espacio muy grande, como por ejemplo, toda una ciudad, una región o un país

*** *Comodities* signfica que son comerciales.**

ATM protocol

Es una técnica estándar de *switching* diseñada para unificar telecomunicaciones y redes de computadores.

En cuanto a las tecnologías más usadas para interconexión en los clusters son las de: Ethernet desde 1974 (100 Gbit/s Ethernet para LAN y 10 Gbit/s para WAN), Myrinet de Myricom desde 1998 (Myri 10 Gbit/s para LAN), Infiniband desde 2005 (Infiniband QDR 12X con 96 Gbit/s para LAN), QsNet de Quadrics (apareció en el 2002, con 8 Bbit/s para LAN), 802.11n desde el 2009 (Wireless Networks con 600 Mbit/s), etc.

Lectura complementaria

Más información en tecnologías de red:
F. Petrini; O. Lysne; y R. Brightwell (2006). "High Performance Interconnects". *IEEE Micro*, núm 3.

Podemos distinguir dos tipos de clusters: los centralizados y los no centralizados. Reciben el nombre de centralizados los que montan el conjunto de PC que lo forman en un *rack* o armario compacto. En este tipo de clusters los PC son normalmente todos del mismo tipo y los únicos periféricos que tienen, por tema de espacio, son las tarjetas de red y posiblemente disco. Los no centralizados son aquellos sistemas cuyos PC se encuentran, por ejemplo, repartidos en todo un campus, o bien en un edificio.

Como ejemplo más visible de cluster tenemos el cluster de computadores que ha construido Google para poder dar respuesta y espacio a todas las consultas de una forma rápida y eficiente. También cabe mencionar el Marenostrum, un sistema basado en procesadores PowerPC y una red de interconexión Myrinet, que fue el multiprocesador más rápido de Europa en el momento de su instalación en el 2004. En el 2006 dobló su capacidad en número de procesadores al pasar a tener 10.240 procesadores IBM Power PC 970MP a 2.3 GHz (2560 JS21 blades). Otros ejemplos que podemos mencionar son el cluster Columbia (la NASA) y que fue el número 2 del Top500 en 2004, el IBM Roadrunner LANL (número 1 en el 2008 del Top500, y que está basado en 12,960 IBM PowerXCell 8i, más 6480 AMD Opteron, unidos con una Infiniband), y el Avalon Beowulf Cluster (de 1998) situado en el Lawerence Livermore National Laboratory.

Marenostrum

Marenostrum se encuentra situado en el BSC-CNS en Barcelona.

1.4. Top500

El proyecto TOP500 es un ranking de los 500 supercomputadores más poderosos del mundo. Esta lista está recopilada por Hans Meuer, de la Universidad de Mannheim (Alemania), Jack Dongarra, de la Universidad de Tennessee (Knoxville) y Erich Strohmaier y Horst Simon, del centro NERSC/Lawrence Berkeley National Laboratory.

El proyecto se inició en 1993 y publica una lista actualizada cada seis meses. La primera actualización de cada año se hace en junio, coincidiendo con la *International Supercomputer Conference*, y la segunda actualización es en noviembre, en la *IEEE Supercomputer Conference*.

En la tabla 2 se muestra el nombre del supercomputador más potente, según los criterios de ordenación de la lista TOP500, desde el año 1993 hasta el 2011.

Tabla 2. Lista de máquinas más potentes según el criterio utilizado por la lista TOP500.

Nombre de la máquina	País	Período
Fujitsu K computer	Japón	Junio 2011 - junio 2012
NUDT Tianhe-1A	China	Noviembre 2010 - junio 2011
Cray Jaguar	EE. UU.	Noviembre 2009 - noviembre 2010
IBM Roadrunner	EE. UU.	Junio 2008 - noviembre 2009
IBM Blue Gene/L	EE. UU.	Noviembre 2004 - junio 2008
NEC Earth Simulator	Japón	Junio 2002 - noviembre 2004
IBM ASCI White	EE. UU.	Noviembre 2000 - junio 2002
Intel ASCI Red	EE. UU.	Junio 1997 - noviembre 2000
Hitachi CP-PACS	Japón	Noviembre 1996 - junio 1997
Hitachi SR2201	Japón	Junio 1996 - noviembre 1996
Fujitsu Numerical Wind Tunnel	Japón	Noviembre 1994 - junio 1996
Intel Paragon XP/S140	EE. UU.	Junio 1994 - noviembre 1994
Fujitsu Numerical Wind Tunnel	Japón	Noviembre 1993 - junio 1994
TMC CM-5	EE. UU.	Junio 1993 - noviembre 1993

2. Multiprocesador

En este apartado analizaremos en detalle los sistemas multiprocesador, también conocidos como sistemas fuertemente acoplados o de memoria compartida. En el subapartado 2.1 estudiaremos cómo se suelen conectar los procesadores con el sistema de memoria y, en particular, veremos las conexiones basadas en bus, las basadas en una red de interconexión de tipo *crossbar* y las de tipo *multistage*. Las del primer tipo, basadas en bus, son las más fáciles de implementar pero tienen el problema de la contención en los accesos a memoria. Las del segundo tipo (*crossbar*) evitan esta contención, pero son muy costosas. Las terceras surgen como una solución intermedia, menos contención que las basadas en bus, y menos costosas que las *crossbar*.

Una forma de reducir más la contención de accesos a memoria es mediante la utilización de bancos de memoria a los que se pueda acceder en paralelo y la incorporación de memorias cachés. Sin embargo, de resultas de esta división en módulos de memoria y de la integración de las cachés aparecen dos problemas: la consistencia de memoria, que analizaremos en el subapartado 2.2, y la coherencia de caché, que estudiaremos en el subapartado 2.3. La consistencia tiene que ver con el orden relativo de escrituras en memoria, y la coherencia tiene que ver con la visibilidad de una escritura en memoria por parte de todos los procesos. Para ambos casos veremos mecanismos *software* y *hardware* para relajar y/o solucionar el problema.

2.1. Conexiones típicas a memoria

2.1.1. Basadas en un bus

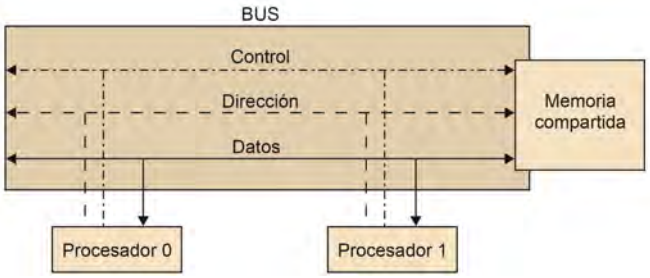
En un sistema multiprocesador basado en bus, todos los procesadores comparten este medio. El bus es la forma más fácil de conectar los procesadores con la memoria y el que mejor escala desde el punto de vista económico con el número de nodos/procesadores conectados al bus.

Coste económico

El coste económico de una red está normalmente asociado a la interfaz de red y al número de dispositivos que la forman.

Otra ventaja de este sistema es que todos los procesadores están conectados con la memoria de "forma directa", y todos están a una distancia de un bus. La figura 8 muestra la conexión de dos procesadores a la memoria a través de un bus. En estas conexiones una parte del bus se utiliza para comunicar la dirección de memoria a la que se quiere acceder, otra para los datos, y otra para las señales de control.

Figura 8. Multiprocesador de memoria compartida a través de un bus (datos, dirección y control).



La gran desventaja de los sistemas basados en bus es la poca escalabilidad que tienen desde el punto de vista de rendimiento. Por ejemplo, si el número de procesadores conectados es muy elevado (más de 20), la contención en el acceso a memoria puede ser significativa. Cada vez que un procesador quiere acceder a memoria, éste debe pedir permiso al árbitro del bus para saber si éste está ocupado con otro acceso. Si el bus está ocupado, el procesador que hizo la petición se tendrá que esperar para intentarlo más adelante. Este protocolo de petición de bus lo tienen que hacer todos los procesadores. Por lo tanto, si el número de procesadores es elevado, la probabilidad de que el bus esté ocupado será más elevada, con la consiguiente pérdida de rendimiento debido a las esperas.

Sistemas basados en bus

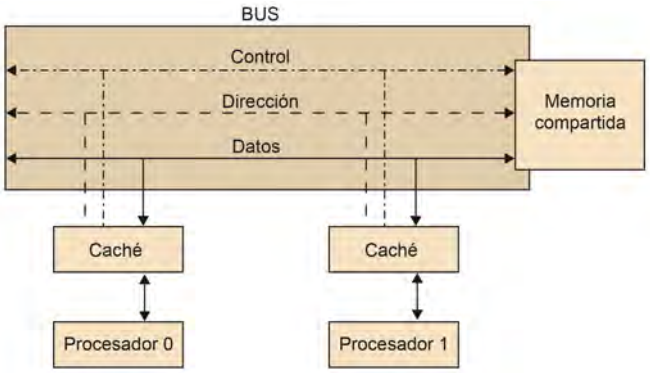
Las redes de interconexión basadas en bus son las más fáciles de montar pero son poco escalables desde el punto de vista del rendimiento.

Una forma de aliviar esta contención cuando tenemos un número relativamente elevado de procesadores es la de incorporar cachés a los procesadores. Con ello, si se explota la localidad temporal, estaremos reduciendo los accesos a memoria. Como contrapartida, tal y como veremos más adelante en este módulo, esto implica un problema de coherencia de los datos al poder tener varias copias del mismo dato en varias cachés del multiprocesador. La figura 9 muestra un esquema básico de un sistema multiprocesador basado en bus con memorias cachés.

Las cachés

Las cachés reducen la contención de acceso a memoria si se explota la localidad de datos, pero introducen el problema de coherencia de caché.

Figura 9. Multiprocesador de memoria compartida a través de un bus (datos, dirección y control), con memoria caché.



Lectura complementaria

Sobre el tema de la implementación de *software caches* podéis leer:
M. Gonzalez; N. Vujic; X. Martorell; E. Ayguade; A. E. Eichenberger; T. Chen; Z. Sura; T. Zhang; K. O'Brien; K. O'Brien (2008)."Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture" (pp. 292-302). Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08).

Finalmente, otra posibilidad de aliviar la contención en sistemas basados en bus, aparte de incorporar cachés, es incorporando memorias privadas en cada procesador. Estas memorias se pueden cargar con código, datos constantes, variables locales, etc. que no afectan al resto de procesadores y que aliviarían también la contención del bus. Sin embargo, en este caso, normalmente es responsabilidad del programador y del procesador realizar la carga de estos datos en esa memoria privada, aunque hay trabajos donde se implementan *software*

caches que intentan quitar esta responsabilidad al programador. Por ejemplo, las tarjetas gráficas del tipo GPGPU (*general purpose Graphic Processing Unit*) tienen una parte de memoria que llaman privada, que está dedicada precisamente a este tipo de datos. Con ello intentan aliviar el acceso a memoria compartida.

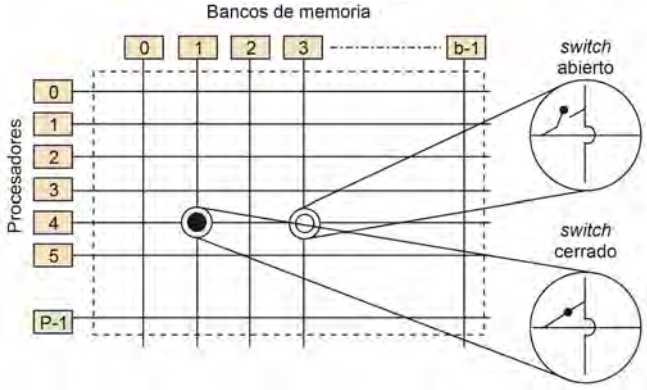
2.1.2. Basadas en *crossbar*

Sistemas basados en *crossbar*

En los sistemas de memoria compartida basados en *crossbar* no hay conflictos de acceso a memoria siempre y cuando no se acceda al mismo módulo de memoria.

Los sistemas basados en bus tienen el problema de escalabilidad a partir de 16 o 32 procesadores. Una forma de evitar este problema de forma sencilla es mediante el uso de *crossbars*. Las redes de interconexión *crossbar* conectan los procesadores del sistema a los bancos de memoria que forman la memoria compartida evitando los conflictos en los accesos. La figura 10 muestra un ejemplo de conexión de P procesadores a b bancos de memoria. Esta consiste en una malla de *switches* que permite la conexión no bloqueante de cualquier procesador a cualquier banco de memoria. Cada *switch* del *crosbar* puede ser electrónicamente cerrado o abierto, permitiendo o no que la línea vertical se conecte a la línea horizontal, respectivamente. En la figura mostramos los dos casos, uno en el que el *switch* está cerrado y otro en el que está abierto.

Figura 10. Multiprocesador de memoria compartida a través de un *crossbar*, enseñando los dos modos en los que se puede encontrar un *switch*.



¿Qué significa que la malla de *switches* permite una conexión no bloqueante? Significa que el acceso de un procesador a un banco de memoria no bloquea la conexión de otro procesador a cualquier otro banco de memoria. Es por eso mismo por lo que normalmente el número de bancos de memoria es mayor que el número de procesadores. En otro caso, habría procesadores que no se podrían conectar a un banco de memoria debido a un conflicto para acceder al mismo banco que otro procesador.

Sistema poco escalable

Las redes de interconexión de tipo *crossbar* evitan los conflictos de acceso a módulos de memoria distintos. Sin embargo, son poco escalables económicamente hablando.

La desventaja de este sistema de conexión con memoria es que se precisan alrededor de P^2 *switches* para poder montar este sistema, siendo este coste importante y, por consiguiente, poco escalable. En cambio, un sistema basado en un bus es escalable en términos de coste, pero no de rendimiento. En el siguiente subapartado mostramos un sistema que tiene un coste menos elevado que los *crossbar* y con mejor rendimiento que los sistemas basados en bus.

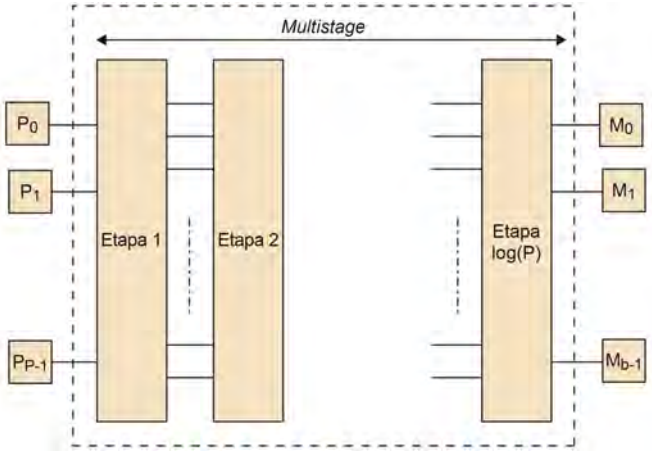
2.1.3. Basadas en *multistage*

Sistemas basados en *multistage*

La red de interconexión *multistage* permite tener menos conflictos de acceso a memoria que los sistemas basados en bus y son menos costosos que las redes de tipo *crossbar*.

Los sistemas basados en *multistage* consisten en una serie de etapas que están unas conectadas a las otras de tal forma que se conecten los procesadores con los bancos de memoria. La figura 11 muestra un esquema general de un sistema de conexión basado en *multistage*.

Figura 11. Red de interconexión *multistage* genérica de P procesadores de entrada y b bancos de memoria de salida.

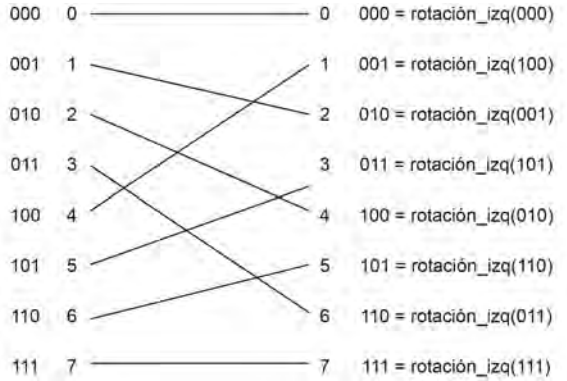


La red *omega* es la *multistage* más común. Ésta tiene log P etapas, donde P es el número de procesadores conectados a la primera etapa, conectadas unas con otras en cadena. El número de salidas de cada etapa es también P, y la última etapa está conectada con P bancos de memoria. La forma de conectar una entrada i con una salida j en una etapa sigue un patrón que se conoce como *perfect shuffle* tal y como indica la siguiente fórmula:

$$j = \begin{cases} 2i & \text{si } 0 <= i <= P/2 - 1 \\ 2i + 1 - P & \text{si } P/2 <= i <= P - 1 \end{cases}$$

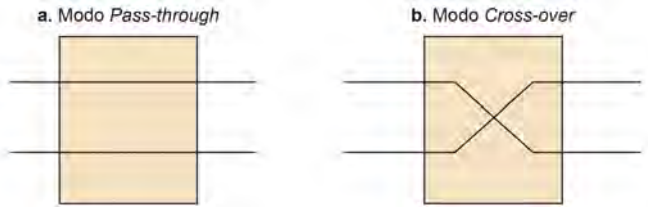
La figura 12 muestra gráficamente cómo se conectan las salidas de una etapa con las entradas de la siguiente etapa en una red *omega*.

Figura 12. *Perfect Shuffle* usado en la conexión de las etapas en la *multistage omega*.



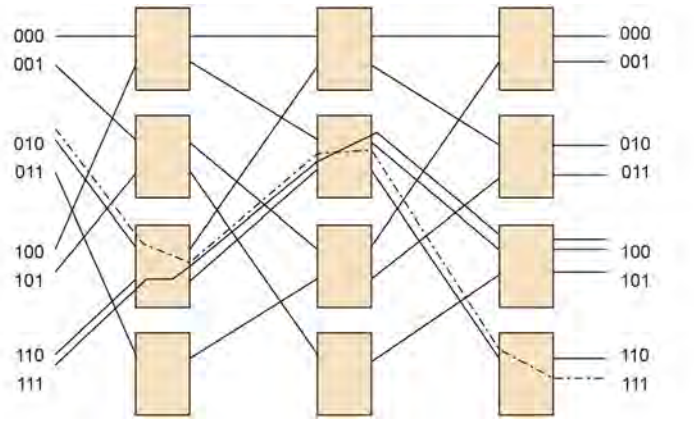
En cada etapa hay $P/2$ *switches*. Un *switch* de la etapa i está configurado de tal forma que para un envío del procesador P al banco de memoria b, comparará el bit i de la codificación del procesador y del banco de memoria al que está accediendo. Si los bits son iguales, el *switch* dejará pasar el dato tal y como muestra la figura 13.a. En caso contrario, si son diferentes, entonces el *switch* cruzará las entradas como se muestra en la figura 13.b.

Figura 13. Modos de empleo de un *switch* de una etapa de la red de interconexión *omega*.



Este tipo de sistema de conexión es menos costoso que un sistema basado en *crossbar*, ya que ahora sólo se necesitan $P/2 \times \log p$ *switches*, en comparación con los $P \times P$ *switches* del *crossbar*. Sin embargo, esta reducción de coste se produce como contrapartida al hecho de que dos accesos desde dos procesadores pueden necesitar compartir un mismo *switch*. En este caso, uno de los accesos se deberá bloquear y esperar a que el otro acabe de hacer el acceso. El conflicto se puede producir tanto en el momento de acceder al dato, o cuando éste está volviendo al procesador. La figura 14 muestra un ejemplo donde dos accesos diferentes se tienen que bloquear, ya que comparten parte del camino.

Figura 14. Ejemplo de conflicto en el acceso de dos procesadores diferentes a dos bancos de memoria diferentes.



2.2. Consistencia de memoria

2.2.1. Definición

El problema de consistencia de memoria aparece cuando una memoria está formada por varios módulos de memoria conectados a través de una red de interconexión a un conjunto de procesadores. En particular, cuando varios procesadores acceden a un dato para escritura se nos plantea la pregunta siguiente: ¿en qué orden se deben ver estas escrituras por parte del resto de procesadores si éstos hacen una lectura de los datos de forma paralela?

Hay muchas respuestas posibles, ya que unos accesos se pueden adelantar a los otros, siendo el resultado de las lecturas impredecible. Todo esto se agrava cuando tenemos cachés, ya que entonces es posible que varios procesadores tengan copias de lecturas previas.

Para que los procesadores que acceden a unos datos sepan, por adelantado, qué valores pueden obtener si hay escrituras y lecturas a la vez, se suelen determinar unas reglas que determinan lo que puede devolver la memoria en esa situación. El conjunto de estas normas es lo que recibe el nombre de modelo de consistencia. Por ejemplo, si una CPU0 realiza un acceso de escritura del valor X en una posición de memoria, la CPU1 realiza también un acceso de escritura del valor Y en la misma posición de memoria, y la CPU2 realiza un acceso de lectura del valor de esa misma posición de memoria, ¿qué debería obtener CPU2? Puede ser que obtuviera el valor X y no el valor Y, siendo esto posiblemente correcto si el modelo de consistencia, que todos los procesadores deben conocer, permite este resultado.

Modelo de consistencia

Un modelo de consistencia es el conjunto de normas que nos indica qué valores deberían ver los procesadores en sucesivas lecturas de una o varias posiciones de memoria.

A continuación detallaremos algunos de los modelos de consistencia conocidos: *strict consistency*, *sequential consistency*, *processor consistency*, *weak consistency* y *release consistency*.

2.2.2. Consistencia estricta o *strict consistency*

En este modelo de consistencia, cualquier lectura sobre una posición de memoria deberá tener como resultado la última escritura realizada en esa posición de memoria. Así, en el ejemplo anterior, la CPU2 debería obtener el valor Y.

Consistencia estricta

La consistencia estricta garantiza que los valores que se ven en memoria estén en estricto orden de escritura.

Esta consistencia se puede alcanzar si hay un único módulo de memoria que trate en estricto orden de llegada los accesos, y sin ningún tipo de caché en el sistema. Esto se puede hacer pero se traducirá en que el acceso a memoria será un cuello de botella.

2.2.3. Consistencia secuencial o *sequential consistency*

El modelo de consistencia secuencial garantiza un mismo orden de las escrituras para todas las CPU. Con este modelo no es posible que dos CPU, que leen un dato de una posición de memoria varias veces al mismo tiempo que otras CPU están escribiendo sobre la misma posición, vean dos secuencias diferentes de valores en esa posición de memoria. El orden en el que se ven las escrituras bajo este modelo no es importante, ya que el que se produzca antes una escritura u otra será aleatorio. Lo importante es que cuando se establezca este orden, todas las CPU vean el mismo orden.

Consistencia secuencial

La consistencia secuencial garantiza que los valores que se ven en memoria se vean en el mismo orden por todas las CPU. No importan si es en un orden diferente al que se hicieron las escrituras.

Por ejemplo, supongamos el caso en el que la CPU0 y la CPU1 realizan un acceso de escritura de los valores X e Y sobre una posición de memoria, separados mínimamente en el tiempo. Muy poco después las CPU2 y CPU3 realizan dos accesos de lectura, cada una sobre la misma posición de memoria en la que la CPU0 y la CPU1 realizan la escritura. El resultado de las lecturas puede variar según la visión que tengan de las escrituras, pero

lo que no puede suceder bajo este modelo de consistencia es que la CPU2 vea X en la primera lectura e Y en la segunda lectura, y que la CPU3 vea Y en la primera y X en la segunda. Esto significaría que ambas CPU tienen una visión diferente de las escrituras. Notad que el hecho de que se haya producido el orden X, y después Y, en vez del orden Y y después X no es importante. Lo importante es que las CPU2 y CPU3 ven el mismo orden.

2.2.4. Consistencia del procesador o *processor consistency*

Consistencia del procesador

La consistencia de procesador garantiza que los valores que se ven en memoria respetan el orden de las escrituras de un procesador. Sin embargo, dos procesadores pueden ver diferentes órdenes de escrituras.

El modelo de consistencia del procesador indica que las escrituras hechas por un procesador sobre una posición de memoria serán vistas por los demás procesadores en el mismo orden en el que se lanzaron. Sin embargo, a diferencia del modelo de consistencia secuencial, no garantiza que los accesos de escrituras de diferentes procesadores se vean igual por los demás procesadores.

Por ejemplo, en el caso de que la CPU0 realice las escrituras $W0_0$, $W0_1$ y $W0_2$ sobre una posición de memoria m, el resto de CPU sólo podrán ver este orden de escrituras sobre la posición de memoria m. Si ahora, de una forma intercalada, la CPU1 realiza las escrituras $W1_0$, $W1_1$ y $W1_2$ sobre la misma posición, el resto de procesadores verán las escrituras de la CPU1 en el orden que se lanzaron. Sin embargo, dos CPU distintas, CPU2 y CPU3, podrían observar un orden diferente de escrituras. Por ejemplo, la CPU2 podría ver $W1_0$, $W0_0$, $W0_1$, $W1_1$, $W0_2$, $W1_2$, y, en cambio, la CPU3 podría ver $W0_0$, $W1_0$, $W1_1$, $W0_1$, $W1_2$, $W0_2$. Como podemos observar, tienen dos visiones globales distintas de las escrituras, pero el orden de las escrituras respeta el orden de cómo se hicieron los accesos de escritura por parte de la CPU0 y la CPU1.

2.2.5. Consistencia *weak*

Consistencia *weak*

La consistencia *weak* no garantiza nigún orden entre las escrituras de un mismo procesador, pero dispone que hay unas operaciones de sincronización que garantizan que todas las operaciones de escrituras se deben acabar antes de que esta operación acabe.

Este modelo es mucho más flexible que el anterior, ya que ni siquiera garantiza el orden de lanzamiento de los accesos de escritura de un mismo procesador. Por consiguiente, para el ejemplo de modelo de consistencia de procesador, podría suceder que la CPU2 viera las escrituras de la CPU0 en este orden: $W0_2$, $W0_1$ y $W0_0$. Con lo que uno puede pensar que ciertamente no hay ningún orden. Sin embargo, el modelo determina que existen operaciones de sincronización que garantizan que todas las escrituras realizadas antes de llamar a esta sincronización se completarán antes de dejar que se inicie otro acceso de escritura.

Las operaciones de sincronización vacían el *pipeline* de escrituras, y siguen un modelo consistente secuencial, de tal forma que los procesadores ven un mismo orden en las operaciones de sincronización. Estas operaciones se deben realizar a nivel *software* y no son a coste cero. Por ejemplo, en el modelo de programación de memoria compartida OpenMP existe la directiva `#pragma omp flush` que realiza esa operación de sincronización.

2.2.6. Consistencia *release*

Consistencia *release*

La consistencia *release* garantiza que si se usan adecuadamente las operaciones de sincronización, las escrituras que se hayan hecho en una sección crítica de datos compartidos ya se habrán acabado antes de entrar otra vez.

El modelo de consistencia *release* intenta reducir el coste de esperar todas las escrituras previas a una operación de sincronización. Este modelo se basa en la idea de las secciones críticas. La filosofía es que un proceso que sale de una sección crítica no necesita esperarse a que todas las operaciones de escritura previas se hayan acabado. Lo que sí que es necesario es que si otro proceso (o éste mismo) tiene que entrar en la sección crítica, se tendrá que esperar a que las operaciones previas a la salida de la sección crítica sí que hayan acabado. De esta forma se evita tener que parar los accesos hasta que realmente sea necesario.

El mecanismo de funionamiento es el siguiente:

- Una CPU que quiere acceder a un dato compartido tiene que realizar una adquisición del bloqueo para poder acceder a los datos compartidos (entrada en la sección crítica).
- Esta CPU puede operar sobre los datos compartidos (dentro de la sección crítica).
- Cuando la CPU acaba, ésta libera la sección crítica utilizando una operación *release*. Esta operación no se espera a que todas las escrituras anteriores acaben, ni tampoco evita que empiecen otras. Pero la operación *release* no se considerará acabada hasta que todas las escrituras previas hayan finalizado.
- Si otra CPU (o la misma) intenta entrar otra vez en una sección crítica, entonces, en el momento de querer adquirir el bloqueo para el acceso compartido, si todas las operaciones *release* ya han acabado, podrá continuar. Si no, se deberá esperar a que todas las operaciones de tipo *release* se hayan efectuado.

Con este mecanismo, es posible que no tengamos que parar ninguna escritura en absoluto, reduciendo así el coste que podía significar tener que mantener el modelo de consistencia *weak*. En este modelo sólo será necesario esperarse a las operaciones si en el momento de entrar en una sección crítica hay alguna operación *release* que todavía no haya acabado.

2.2.7. Comparación de los modelos de consistencia

Fence

Un *fence* es una barrera de sincronización de memoria que fuerza que todas las operaciones de memoria previas se hayan acabado.

En la tabla 3 se muestran los órdenes entre los diferentes accesos a memoria y de sincronización, para los modelos de consistencia *sequential consistency*, *processor consistency*, *weak consistency* y *release ordering*, desde el punto de vista de un único procesador. Así, los modelos más relajados basan su consistencia únicamente en los *fences* creados por operaciones de sincronización (S en la tabla), en contraposición de los modelos más estrictos, donde los *fences* son implícitos en las operaciones de lectura y escritura. Es por eso por lo que los modelos más relajados (*weak consistency* y *release ordering*) no tienen ningún orden definido entre las operaciones de lectura y escritura.

Las operaciones de sincronización S_A y S_R son las operaciones de sincronización `acquire` (adquirir bloqueo) y `release` (liberar bloqueo) que son necesarias para el modelo de con-

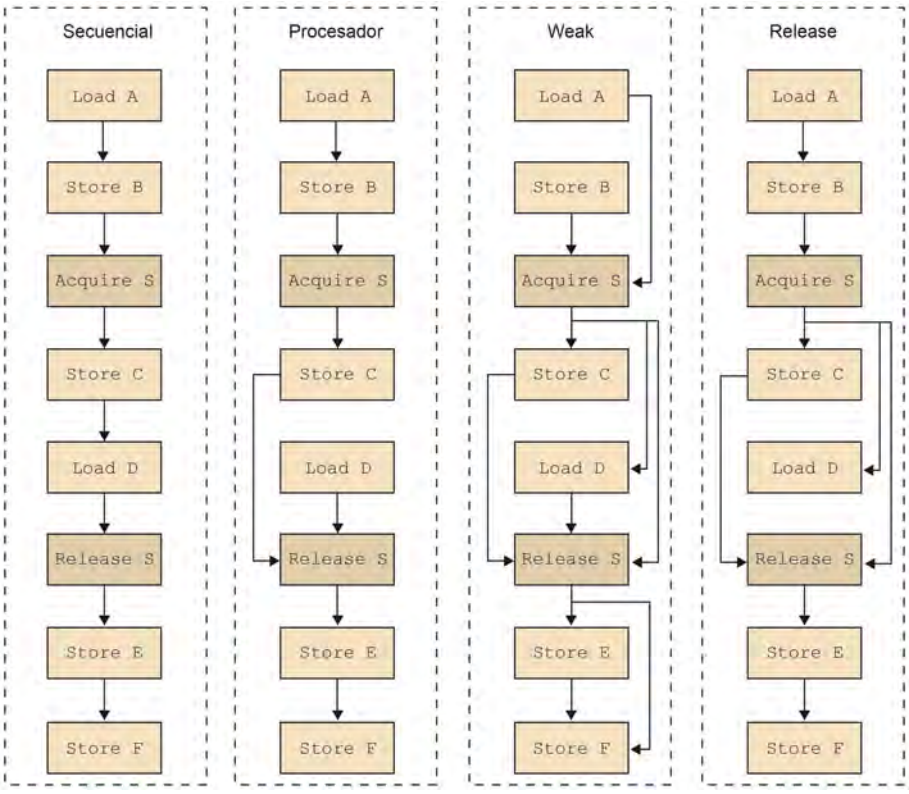
sistencia *release consistency*. En equivalencia, la operación de sincronización S se transformaría en S_A y S_R. Así, por ejemplo, el caso $S \rightarrow R$ se transformaría en $S_A \rightarrow R$ y $S_R \rightarrow R$, y un $S \rightarrow S$ se transformaría en: $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$.

Tabla 3. Orden de ejecución en los modelos de consistencia *sequential*, *processor*, *weak* y *release* de los acesos de lectura y escritura, y las sincronizaciones.

Model	Orden en lecturas y escrituras	Orden en operaciones de sincronización
Sequential consistency	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Processor consistency	$R \rightarrow R$, $R \rightarrow W$,	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
	$W \rightarrow W$	
Weak consistency		$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Release ordering		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_A$, $W \rightarrow S_R$ $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$

La figura 15 muestra gráficamente los órdenes impuestos según el modelo de consistencia usado. En esta figura se muestran los órdenes mínimos entre los diferentes accesos (de sincronización: `acquire`, `release` o de escritura y lectura: `Load`, `Store`), por lo que aquellos que vengan de forma transitiva no se mostrarán, como por ejemplo, la escritura en C, antes del *release* de S en el modelo de consistencia *sequential*. Como se puede observar, a medida que el modelo es más relajado, el número de órdenes necesarios se reduce.

Figura 15. Muestra el orden que deben cumplir las operaciones de acceso a memoria y de sincronización para los diferentes modelos de consistencia, para una misma secuencia de operaciones.



2.3. Coherencia de caché

La diferencia básica entre consistencia de memoria y coherencia de caché es que la primera determina cuándo un valor escrito puede ser visto por una lectura, y la segunda indica qué valores deberían poder ser devueltos por una lectura (sin indicar cuándo).

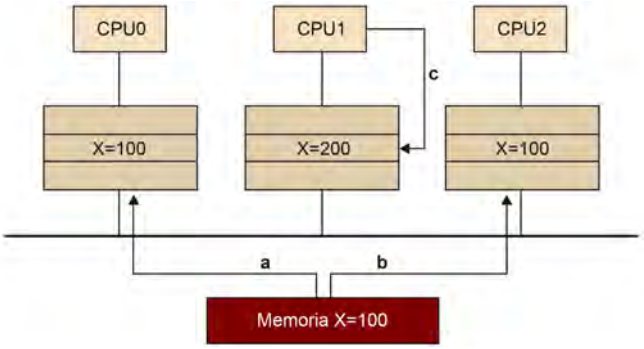
2.3.1. Definición

El problema de la coherencia de caché aparece en el momento en que introducimos la caché en la jerarquía de memoria de los procesadores que forman parte de un multiprocesador.

El objetivo principal de introducir estas cachés es explotar mejor la localidad espacial y temporal de los datos, y así, reducir la contención de memoria. Sin embargo, como consecuencia de la incorporación de las cachés en un sistema multiprocesador, un mismo dato puede estar replicado en varias cachés a la vez. Esto no es un problema si no hay ninguna actualización del dato. En cambio, si se produce una escritura en el dato por parte de un procesador, el resto de copias del valor, en el resto de cachés, quedarán desactualizadas. En este caso hay un problema de coherencia de la caché.

En la figura 16 vemos un ejemplo en el que un *thread* ejecutándose en la CPU0 realiza una lectura de la variable X (figura 16.a) y lo deja en su caché. Luego, otro *thread* desde la CPU2 realiza otra lectura (figura 16.b), y finalmente un *thread* de la CPU1 realiza una escritura (figura 16.c). En este momento, tanto la CPU0 como la CPU2 tienen el dato con un valor antiguo y diferente al de la CPU1. En este caso se dice que estas cachés no tienen coherencia.

Figura 16. Ejemplo de un multiprocesador sin coherencia de caché.



A continuación vamos a ver una serie de protocolos para garantizar la coherencia de caché.

2.3.2. Protocolos de escritura

Para mantener la coherencia de caché en los sistemas multiprocesador se pueden seguir dos políticas en la escritura de un dato*:

*** A veces hablamos de actualización del dato, o coherencia de ese dato, pero en realidad deberíamos hablar de bloque de memoria.**

1) *Write-updated* o *write broadcast*: Mediante este protocolo se mantienen actualizadas las copias del dato (bloque de memoria) en las cachés del resto de procesadores.

2) *Write-invalidate*: Mediante este protocolo se pretende asegurar que el que escribe tiene la exclusividad del dato (bloque de memoria), haciendo que el resto de procesadores invaliden su copia y no puedan tener copias desactualizadas. El dato (bloque de memoria) es actualizado en memoria cuando es *flushed*/reemplazado de la caché o lo pide otro procesador. Con este protocolo, el dato (línea de caché con el bloque de memoria donde está el dato) tiene unos bits de estado para indicar exclusividad, propiedad y si ha sido modificado (*dirty*).

La segunda política es la que se suele usar más, ya que normalmente tiene mejor rendimiento que la primera. A continuación detallamos algunas de las características para entender mejor el rendimiento que se puede alcanzar con cada una de ellas:

1) En el caso de tener el protocolo *write-update* y tener que realizar varias escrituras sobre un mismo dato, sin ninguna lectura intercalada, el sistema tendrá que hacer todas las actualizaciones de memoria y cachés del resto de procesadores. Sin embargo, en el caso del protocolo *write-invalidate*, sólo es preciso realizar una invalidación, ya que una vez que queda una única copia en el sistema, no hay necesidad de enviar más invalidaciones.

2) En el caso de tener que realizar escrituras sobre datos distintos que van al mismo bloque de memoria, si estamos en el protocolo *write-update*, tendremos que actualizar el dato en ese bloque de memoria tantas veces como sea necesario. Sin embargo, mantener todos los datos actualizados puede significar un elevado número de accesos a memoria. En cambio, si estamos en un protocolo de *write-invalidate* se invalidarán las líneas de cachés que contienen copia del dato (bloque de memoria) la primera vez que se escriba en un dato dentro de esta línea, y ya no se hará nada más sobre ella. En contrapartida, se podrían estar provocando fallos de caché en las lecturas, por parte de otros procesadores, de otros datos en la misma línea de caché del dato invalidado.

Write buffers

En el protocolo *write-update* se pueden agrupar las escrituras a un mismo bloque de memoria en los *write buffers* para reducir la contención de acceso a memoria.

3) En el caso de una lectura de un dato, seguramente que la lectura en un sistema con protocolo *write-update* es más rápida que en un sistema *write-invalidate*. Esto es así básicamente porque si el procesador contenía una copia del dato, ésta se habrá mantenido actualizada con el protocolo *write-update*. En cambio, en el protocolo *write-invalidate*, el dato habrá sido invalidado. Sin embargo, si la programación es adecuada, estas invalidaciones por datos que comparten un bloque de memoria se pueden reducir o evitar.

Así, el protocolo *write-update* necesita mayor ancho de banda de acceso a bus y a memoria, y es la razón por la cual el protocolo que normalmente es más usado es el de *write-invalidate*.

2.3.3. Mecanismo *hardware*

La forma de mantener la coherencia de caché en *hardware* es mediante la utilización de bits de estado para cualquier bloque de memoria que se comparte. Hay dos mecanismos *hardware* que mantienen actualizado el estado de los datos compartidos:

- Sistema basado en *Snoopy*: los bits de estado del bloque de memoria están replicados en las cachés del sistema que tienen una copia del dato (bits por cada línea de caché), a diferencia del basado en directorios, que está centralizado. Son sistemas en los que los procesadores suelen estar conectados a la memoria vía bus, y además, cada controlador de caché tiene un *hardware* dedicado que puede leer/sondear (*Snoopy*) qué pasa por el bus. Así, todos los controladores de memoria miran si las peticiones de accesos que se realizan sobre el bus afectan a alguna de sus copias o no.

- Sistema basado en directorios: Los bits de estado de compartición de un bloque de memoria están únicamente en un sitio llamado directorio. Cuando se realizan lecturas y escrituras sobre un dato del bloque de memoria, su estado, centralizado en el directorio, se deberá actualizar de forma adecuada.

Diferencias de los mecanismos

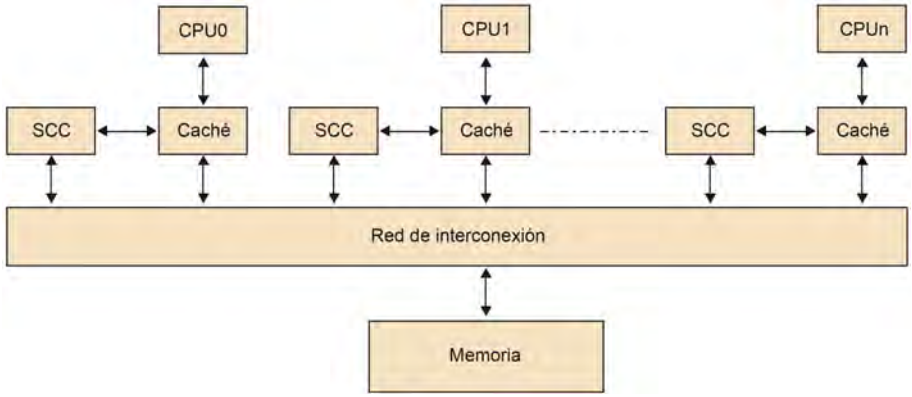
Los sistemas basados en *Snoopy* tienen bits de estado para cada copia del bloque de memoria. En cambio, los sistemas basados en directorios sólo tienen unos bits de estado por cada bloque de memoria.

Sistema basado en *Snoopy*

Los sistemas con cachés con *Snoopy* están normalmente asociadas a sistemas multiprocesadores basados en redes de interconexión tales como un bus o un *ring*.

La figura 17 muestra un esquema básico de un sistema multiprocesador con una serie de procesadores, una caché asociada por procesador, y un *hardware* dedicado de *Snoopy cache coherence* (SCC). Todos ellos están conectados a la red de interconexión (normalmente un bus) que conecta la memoria compartida con los procesadores.

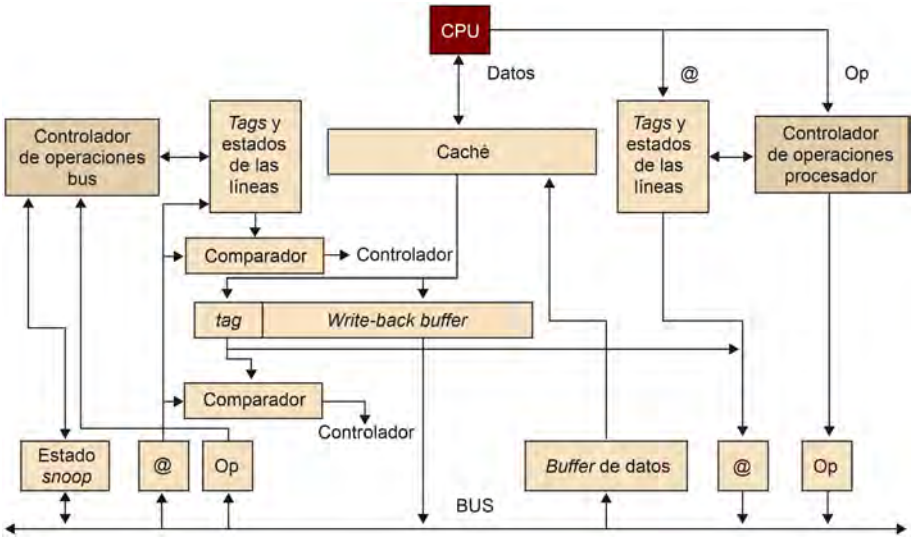
Figura 17. Esquema básico de un sistema multiprocesador con memoria compartida basado en un bus de interconexión y *Snoopy* cache *coherence*.



Tal y como se conectan los SCC al bus, todas las transacciones en el bus son visibles en todos los procesadores. De esta forma, monitorizando estas transacciones, se pueden tomar las acciones necesarias de cambio de estado de las líneas de la caché, según los estados que determine el protocolo de coherencia que se asuma.

La figura 18 muestra un poco más en detalle lo que puede significar un SCC. Si nos fijamos en la parte derecha del esquema, veremos un *hardware* dedicado (controlador del procesador) que recibe comandos del procesador y que está conectado a la información de los *tags* y del estado de las líneas de caché. Así, dependiendo de los comandos u operaciones que se efectúen en el procesador, el controlador del procesador, mirando el estado de las líneas de caché y sus *tags*, decidirá enviar operaciones al bus y/o cambiar el estado de alguna línea de caché.

Figura 18. Detalle del *Snoopy cache coherence hardware*.



Por otro lado, si nos fijamos en el lado izquierdo de la figura, veremos que hay un *hardware* dedicado a sondear/recibir los comandos u operaciones que vienen del bus (Op conectado al BUS) o interconexión con memoria y que se conecta con el controlador del bus para enviarle la información. La dirección que viene del bus se lee y se pasa a los dos comparadores existentes: uno para comparar con los *tags* de la caché y otro para compararlo con los *tags* del *write-back buffer*. El *write-back buffer* es un *buffer* para agrupar escrituras a una misma línea de caché. En caso de tener coincidencia en alguna línea o en el *write-back buffer*, se deberá actuar en consecuencia según el protocolo de coherencia seguido. También observamos que hay una conexión llamada estado *Snoopy* que le sirve al procesador para informar al bus de que tiene una copia del dato del cual se está haciendo una lectura por parte de otro procesador, o viceversa, para que informen al procesador de que no tendrá exclusividad en el dato que está leyendo de memoria.

Los sistemas basados en bus con *Snoopy* son fáciles de construir. Sin embargo, el rendimiento de éstos sólo es bueno si el número de procesadores no crece significativamente (más de 20), y si los accesos a la memoria se mantienen locales en cada procesador, es decir, se accede a la caché de cada procesador. El problema surge cuando se empiezan a realizar muchas escrituras y/o tenemos muchos procesadores conectados. Esto puede significar que todos los procesadores empiecen a compartir datos, con lo que tendríamos un tráfico de operaciones a través del bus significativo, ya que los mensajes se envían a todos los procesadores en forma de *broadcast*.

A continuación veremos un sistema basado en directorios, donde se guarda la información del estado de las líneas de una forma centralizada (o no replicada). Estos directorios también guardan los procesadores que comparten un bloque de memoria. En estos sistemas se evitan los envíos en forma de *broadcast* y sólo se envían las actualizaciones a aquellos procesadores que mantienen copia. Estos sistemas son más escalables que los sistemas basados en bus, pero su construcción es más compleja.

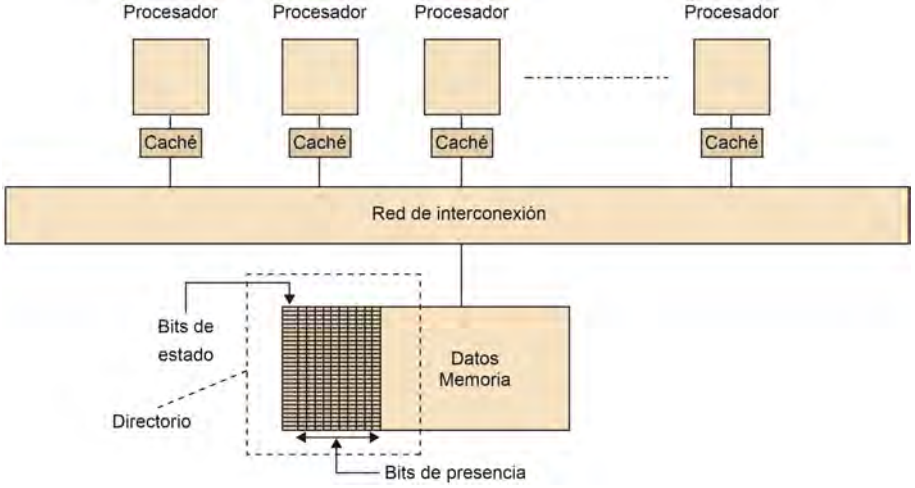
Sistema basado en directorios

Sistema basado en directorios

En un sistema basado en directorios se guarda información de los procesadores que tienen copia de un bloque de memoria para reducir la comunicación a través de la red de interconexión.

Figura 19 muestra un sistema multiprocesador formado por varios procesadores que, a través de la red de interconexión, se comunican con la memoria compartida global. Esta memoria compartida tiene soporte *hardware* para guardar la información de los procesadores que disponen de una copia de ese bloque de memoria en sus cachés. Esta información es clave para que este sistema *hardware* sea más escalable que el de *Snoopy*. Con esta información, ahora se podrán enviar los mensajes de actualización o invalidación sólo a aquellos procesadores que dispongan de una copia del dato.

Figura 19. Sistema multiprocesador formado por una memoria principal compartida con el sistema de directorios de coherencia de caché centralizado.



Hay varias posibilidades de guardar esta información, aunque en la figura mostramos sólo una de ellas:

- Mantener un único identificador del procesador que tiene el valor de ese bloque de memoria. De esta forma sólo se podría tener una copia del dato. Esto no permite lecturas concurrentes sin provocar accesos en la memoria principal para actualizar qué procesador tiene la copia en cada momento. La ventaja es que es escalable desde el momento en que no guardas más que un entero independientemente del número de procesadores.

- Mantener la información de k procesadores. Esto conlleva más esfuerzo en cuanto a que hay que guardar más enteros, y dependiendo del número de bloques de memoria, puede ser relativamente un coste significativo (k enteros por bloque de memoria). En cambio, a diferencia de la anterior opción, ahora sí que se permitirían lecturas de hasta k procesadores sin ningun problema.

- Mantener P* bits, un bit por procesador, para indicar la presencia o no del dato en la caché de un procesador. De esta forma todos podrían tener copias, y según el número de procesadores, sería mucho más rentable que tener k enteros. El único inconveniente es cuando el número de procesadores aumenta considerablemente. En este caso se podría alcanzar un número de bits relativamente elevado en comparación con el tamaño de un bloque de memoria. Ésta es la opción que mostramos en la figura. Una posible solución para reducir este coste sería aumentar el tamaño de los bloques de caché, de tal forma que reduciríamos el sobrecoste de mantener P bits por bloque de memoria. Sin embargo, esto tendría como efecto colateral que el número de datos que comparten un mismo bloque de memoria será mayor y habrá más probabilidades de que haya invalidaciones por actualizaciones de datos diferentes, que caen en el mismo bloque de memoria.

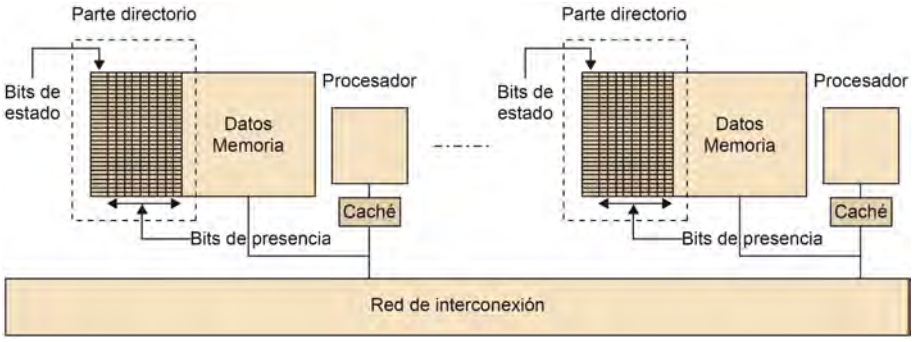
- Mantener una lista encadenada de los procesadores que tienen copia, utilizando memoria para guardar esa información. El *hardware* dedicado sólo requeriría ser un registro para guardar la cabecera de esa lista encadenada.

*** En este módulo nos centraremos en sistemas de directorios con P bits de presencia.**

En este módulo nos centraremos en el caso de directorios donde puede haber tantas copias como procesadores, utilizando un bit de presencia para cada procesador. Esta información se guarda junto con los bits de estado del bloque de memoria en la memoria. El valor de esos bits de estado depende del protocolo de coherencia que se tome. En el subapartado 2.3.4 veremos dos: MSI y MESI, cuyos nombres provienen de los posibles estados de compartición que puede tener un bloque de memoria. Para estos protocolos, los posibles estados de un bloque de memoria son los de *modified*, *exclusive* para el MESI, *shared* e *invalid*.

En un sistema con un protocolo basado en directorio, cada vez que se accede a un dato se tiene que consultar y posiblemente actualizar los bits de presencia y de estado en el directorio de memoria. Por lo tanto, si la memoria está formada por un solo módulo, tanto los accesos a memoria como la actualización del directorio se pueden convertir en cuellos de botella. Una posible solución es distribuir la memoria físicamente en módulos entre los nodos del sistema multiprocesador, repartiendo de la misma forma el directorio entre los diferentes módulos, para aumentar así el paralelismo en la gestión de accesos a la memoria principal y las transacciones a través del bus. La figura 20 muestra un esquema de un sistema multiprocesador con el directorio distribuido, típico de un sistema multiprocesador NUMA.

Figura 20. Sistema multiprocesador formado por una memoria principal compartida, físicamente distribuida, con el sistema de directorios de coherencia de caché distribuido.



Los sistemas basados en directorios, junto con los protocolos de coherencia MSI y MESI, tienen como objetivo evitar que los procesadores deban generar transacciones sobre la red de interconexión tanto para lecturas como para escrituras sobre un bloque de memoria, si éste ya se encuentra en la caché o bien está actualizado en la caché del procesador. Este objetivo es el mismo que se tiene con los sistemas basados en *Snoopy*, que también evitan transacciones innecesarias. La diferencia básica con respecto al sistema basado en *Snoopy* es que ahora, si hay varios procesadores que actualizan un dato, las transacciones en el bus sólo se envían a aquellos que participan de la compartición de ese bloque de memoria.

En un sistema multiprocesador con el sistema de directorios distribuido pueden estar involucrados tres tipos de nodos en las transacciones a través del bus:

1) El *local node* que es el procesador/nodo que genera la petición.

2) El *home node* que es el nodo donde reside el bloque de memoria según su dirección de memoria, y la asignación que haya realizado el sistema operativo a través de la traducción de memoria virtual a física (`MMU`).

3) El *remote node* que es el nodo que tiene una copia del bloque de memoria, ya sea en exclusividad o de forma compartida.

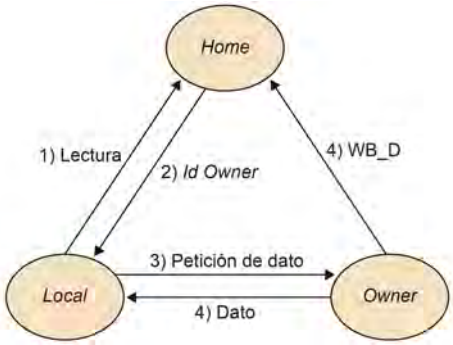
De entre estos tres tipos de procesadores tenemos que darnos cuenta de la importancia del *home node* de un dato. Este procesador mantendrá el estado del bloque de memoria que contiene el dato y la información de los procesadores que tienen una copia de éste. La forma de asignar el *home node* depende de la política de traducción de direcciones virtuales a físicas por parte del sistema operativo. Normalmente, la política consiste en intentar acercar las páginas de memoria al procesador que las ha pedido primero. Por lo tanto, si el programador no es consciente de este detalle podríamos hacer que nuestros programas no obtuvieran el rendimiento esperado. Lo entenderemos en cuanto veamos algunos casos de ejemplos donde se realizan lecturas y escrituras sobre un dato que ya tiene un *home node* asignado.

El primer caso que trataremos es cuando el *local node* realiza una lectura de un dato con una determinada dirección de memoria. El sistema operativo, a través de la traducción de la MMU, podrá determinar qué nodo tiene ese dato en su memoria local, el *home node*, para poder saber el estado de ese bloque de memoria. El *home node* responderá a la petición de dos posibles formas, según haya o no un procesador con el bloque de memoria actualizado:

1) Para el caso en que haya un procesador con una copia en exclusividad modificada: la figura 21 muestra los tres nodos implicados. Las aristas de la figura están numeradas según la secuencia de operaciones, indicando la operación o dato que se envía de un nodo a otro. Así, el *home node* indica al *local node* qué nodo es el *remote* o *owner node*, tras recibir una petición de lectura de un dato. El *local node*, con esta información, pide al *owner node* que le suministre el bloque de memoria en cuestión. Esto provoca que el *owner node* le devuelva el dato y que se actualice la memoria principal *writeback-data* (WB_D). Aunque

no se muestra en la figura, los bits de presencia se deben actualizar adecuadamente en el *home node*, al igual que los bits de estado de la caché.

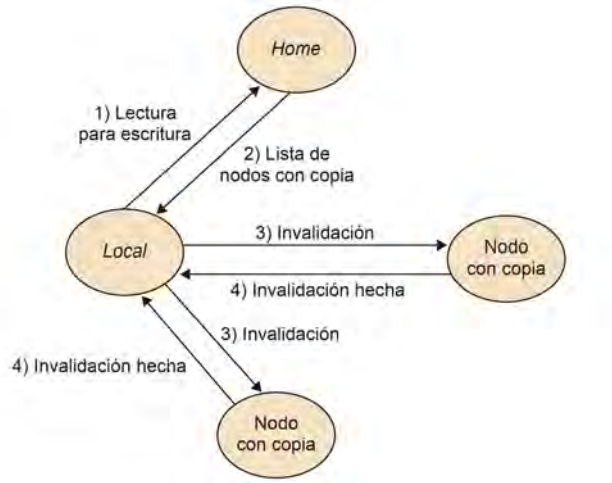
Figura 21. Acceso de lectura a un dato de un bloque de memoria que está modificado.



2) Para el caso en que no haya ningun procesador con una copia en exclusividad modificada: el *home node* suministra el bloque de memoria al *local node* (aunque posiblemente también se la podría dar otro procesador con una copia según el protocolo de coherencia usado) y, en función de si el *local node* es el único o no con este bloque de memoria, asignará el estado *exclusive* (en un protocolo de coherencia MESI) o *shared* (en un protocolo de coherencia MSI) respectivamente a los bits del bloque de memoria.

En el caso de que el *local node* quiera realizar un acceso de lectura con intención de escritura, tal y como muestra la figura 22, el *home node* suministrará la lista de posibles nodos que tengan una copia de este bloque de memoria; dos nodos en el caso de la figura. El *local node*, con esta información, hará una petición a esos nodos para que invaliden las copias del bloque de memoria. El *home node* actualizará los bits de presencia para indicar que sólo el *local node* tiene una copia del bloque de memoria y que éste está modificado.

Figura 22. Acceso de escritura a un dato de un bloque de memoria que está compartido por dos procesadores más.



Con estos dos casos observamos que el *home node* de un dato debe mantener, en todo momento, el estado y los bits de presencia del bloque de memoria que contiene el dato en cuestión, tanto para lecturas como para lecturas con intención de escritura. Esto significa

que si el procesador que realiza el primer acceso no coincide con el procesador que realiza más actualizaciones de ese bloque de memoria, las peticiones tendrán que pasar por el *home node* cada vez, y deteriorarán el rendimiento de la aplicación innecesariamente. Esto suele pasar, en particular, si la política de traducción de direcciones de virtual a física hace esta asignación inicial del *home node** en función de qué nodo toca primero el bloque de memoria.

*** El primer acceso a un dato de memoria determina qué nodo es el *home node*.**

Finalmente, cabe comentar que hay sistemas basados en directorios que incorporan éste en la caché más externa compartida, como puede ser el caso del Intel i7 y la serie 7000 de Xeon.

2.3.4. Protocolos de coherencia

En este apartado analizaremos en detalle dos protocolos de coherencia (MSI y MESI) que se pueden combinar con la política en escritura *write-invalidate* y los mecanismos *hardware* que hemos visto. Estos protocolos determinan los estados de compartición de memoria en los que puede estar un bloque de memoria, y cómo se realiza la transición de uno a otro según los accesos que se produzcan en el sistema multiprocesador. En particular, describiremos los dos protocolos para un sistema basado en *Snoopy*.

MSI

El protocolo MSI de coherencia contempla los siguientes estados:

1) *Dirty* o *modified* (**M**): El estado en el que un bloque de memoria (línea de caché) está en la caché cuando se ha modificado o se ha leído para modificar.

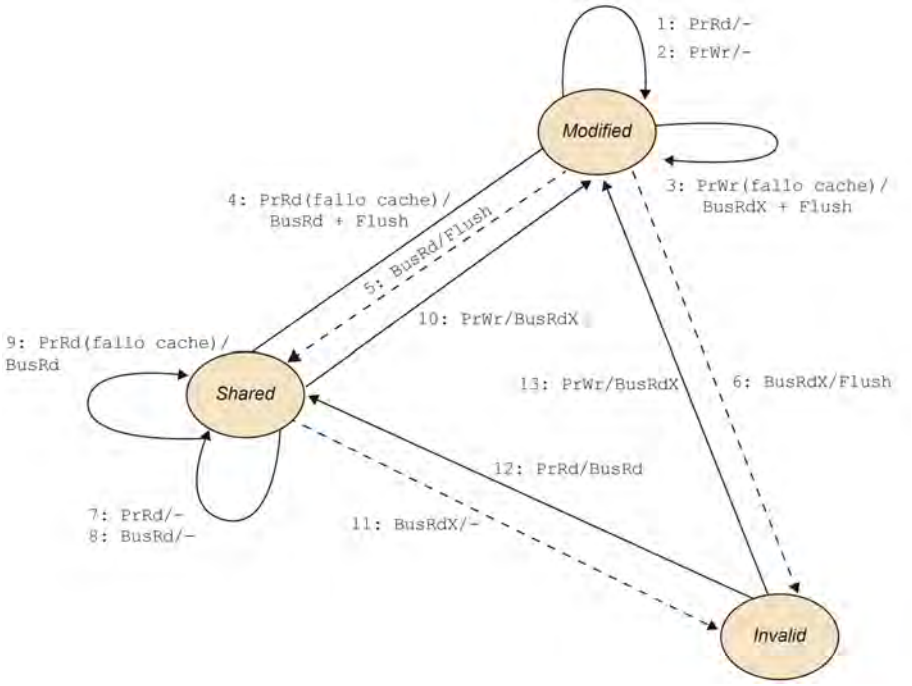
2) *Shared* (**S**): El estado en el que un bloque de memoria está cuando se ha leído y no hay intención de modificarlo.

3) *Invalid* (**I**): El estado en el que un bloque de memoria (línea de caché) está cuando se ha invalidado o bien no existe en la caché.

¿Cómo aplicaríamos estos estados a una máquina con política en escritura *write-invalidate* y un sistema *hardware* basado en bus (*Snoopy*)? El grafo de estados que se muestra en la figura 23 determina los estados en los que puede estar una línea de caché (nodos del grafo) y las acciones del bus o del procesador que provocan los cambios de un estado al otro (aristas del grafo). Las aristas del grafo están etiquetadas con el siguiente formato: <origen de la acción / transacción en el bus realizada como parte de la acción>. Las operaciones o acciones que se pueden realizar por parte del procesador son : `PrRd` (el procesador hace una lectura `Rd`) y `PrWr` (el procesador realiza una escritura `Wr`). Las acciones que se pueden realizar a través del bus son: `BusRd` (se realiza un petición de lectura a memoria), `BusRdX` (se realiza una petición de lectura con intención de escritura), y `flush` se deja en el bus el bloque de memoria que contiene el dato (la línea de caché), para que la memoria

y, si se da el caso, la caché del procesador que la solicitó, puedan actualizar la línea. Estas acciones en el bus se ven en el resto de procesadores gracias a los mecanismos *hardware* para mantener la coherencia. Por ejemplo, en un sistema basado en *Snoopy*, el SCC del sistema podrá ver qué pasa por el bus y actuar en consecuencia.

Figura 23. Grafo de estados del protocolo de coherencia `MSI`.



Desde el punto de vista de la caché de un procesador del sistema, los estados que pueden tener cada una de las líneas de la caché son los siguientes:

1) *Modified*: Si una línea de caché está en el estado de modificada significa que el bloque de memoria al que corresponden los datos de esa línea de caché es la única copia en el sistema multiprocesador y que ni siquiera la memoria tiene el bloque actualizado.

- Peticiones desde el procesador: Si el procesador que contiene este bloque de memoria en su caché realiza una operación de lectura (`PrRd`) (acción 1 en el grafo) o escritura (`PrWr`) (acción 2 en el grafo) sobre algun dato de esta línea, el procesador no tendrá por qué realizar ninguna acción en el bus, y además, la línea de caché mantendrá el mismo estado. En el caso de tener un fallo de escritura en el acceso a un bloque de memoria diferente (acción 3 en el grafo), con reemplazo de línea de caché, la línea se quedaría en el mismo estado, pero se debería emitir una acción de lectura para escritura en el bus (`BusRdX`) para obtener el nuevo bloque de memoria y un `flush` de la línea reemplazada. Si tuviéramos un fallo de lectura en el acceso a un dato de un bloque de memoria diferente (acción 4 en el grafo), pero que hiciera que cayera en una línea de caché con estado modificado (la línea de caché se tiene que reemplazar), la línea de caché pasaría al estado de compartido (con el nuevo bloque de memoria) y el procesador realizaría una petición de lectura en el bus y en un `flush` del contenido de la línea de caché reemplazada.

- Peticiones desde el bus: En el caso de recibir una petición de lectura del bus (`BusRd`) de un bloque de memoria que se contiene en una línea de caché con estado modificado (acción 5 en el grafo), la línea de caché se tendrá que suministrar a la memoria y al procesador que solicitó el bloque de memoria. Esto se realiza mediante un `flush` de la línea en el bus. La línea deberá pasar a estado compartido. Por otro lado, si la petición en el bus es la de leer para modificar (`BusRdX`) (acción 6 en el grafo), también deberemos suministrar la línea de caché mediante un `flush`, pero en este caso el estado de la línea en nuestra caché pasa a ser inválido. Esto es debido a que sólo una caché puede contener un mismo bloque de memoria cuando éste se ha modificado.

2) *Shared*: Una línea de caché en este estado indica que contiene un bloque de memoria que no se ha modificado, y por consiguiente, el bloque puede estar replicado en más de una caché del sistema multiprocesador. En este estado, si el procesador realiza una petición de lectura `PrRd` (acción 7 en el grafo) o bien observa que en el bus se está realizando una petición de lectura (`BusRd`) (acción 8 en el grafo) sobre el bloque de memoria de la línea de caché compartida, ésta no cambiará de estado. Si se realiza una lectura `PrRd` y se produce un fallo de caché (acción 9 en el grafo), la línea no cambiará de estado pero se tendrá que hacer una petición al bus del nuevo bloque de memoria (`BusRd`). En cambio, si se realiza un petición de escritura por parte del procesador (`PrWr`) (acción 10 en el grafo) sobre la línea compartida, éste deberá avisar al bus de que quiere realizar una modificación del bloque de memoria contenido en esa línea (`BusRdX`), de tal forma que se puedan invalidar las líneas de caché del resto de procesadores con ese mismo bloque de memoria. En este caso, el estado de la línea de caché pasará a ser *modified*. Finalmente, en el caso de que se observe una petición de escritura sobre un bloque de memoria (`BusRdX`) (acción 11 en el grafo) contenido en una línea de caché con estado compartido, esta línea deberá cambiar al estado de inválido, ya que sólo un procesador puede tener una copia modificada de un bloque de memoria.

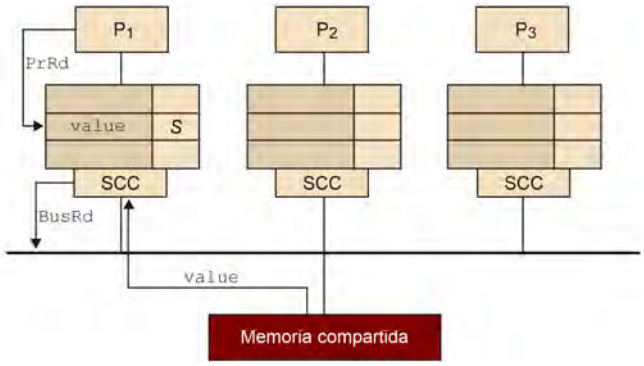
3) *Invalid*: Una línea de caché está en este estado cuando o bien nunca se ha leído un bloque de memoria que vaya a esta línea, o bien se leyó pero después quedó invalidada por una petición de escritura sobre el mismo bloque de memoria por parte de otro procesador. Las únicas acciones que pueden afectar al estado de una línea de caché inválida son las de lectura o escritura por parte del procesador (acciones 12 y 13 en el grafo). Las acciones del bus no le afectan porque esa línea de caché no contiene ningún bloque de memoria válido. Así, si se realiza una petición de lectura por parte del procesador (`PrRd`), la línea de caché pasará al estado *shared* una vez que haya podido dar la orden de petición de lectura a través del bus (`BusRd`), y poder informar al resto de procesadores. Si se efectúa una petición de escritura por parte del procesador (`PrWr`), en este caso la línea de caché pasará al estado *modified*, una vez realiza la petición al bus de que se va a hacer una escritura en el bus (`BusRdX`).

A continuación veremos un ejemplo de cómo cambian los estados de las líneas de caché en las cachés de diferentes procesadores, cuando se accede a un mismo bloque de memoria en un sistema multiprocesador. Supongamos el caso en el que tenemos tres procesadores $(1, 2, 3)$, y que realizan una secuencia de lecturas y escrituras. Cada uno de ellos está leyendo o escribiendo el mismo dato de memoria. $r1$ significa lectura(r de *read*) del procesador 1 y $w3$ significa escritura (w de *write*) del procesador 3. La secuencia de escrituras

y lecturas es la siguiente: $r1$, $r2$, $w3$, $r2$, $w1$, $r3$ y $r1$, y, al empezar, supondremos que las líneas de caché, donde va el bloque de memoria que contiene el dato leído y escrito en cada procesador, están invalidadas.

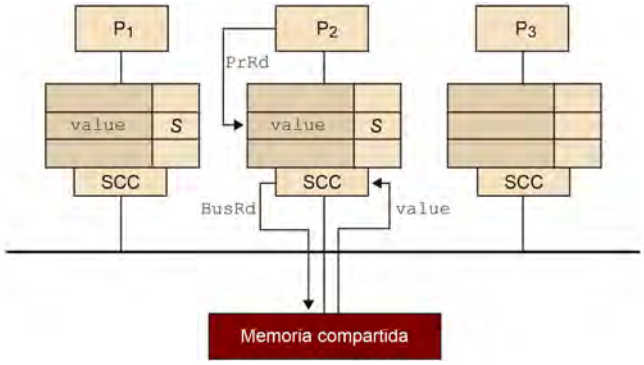
El primer acceso a memoria lo realiza el procesador 1. La figura 24 refleja cada una de las acciones que se deben realizar tanto en el procesador como en el bus, y cómo queda el estado de la línea de caché asociada a este dato, en cada uno de los tres procesadores. Así, el procesador 1 realiza una lectura de un dato, y como su caché no la tiene, tiene que generar una petición de lectura a través del bus (`BusRd`). Al no tenerla ningún otro procesador, el controlador de la memoria principal dejará en el bus el bloque de memoria donde está el dato, permitiendo que el SCC del procesador que pidió el dato lo lea y actualice la línea de caché. El SCC también actualizará el estado de la línea de caché, que ahora pasará a estado compartido (S en la figura).

Figura 24. El procesador 1 realiza una petición de lectura.



El segundo acceso lo realiza el procesador 2, que hace una lectura del mismo valor. La figura 25 muestra el estado de las líneas de caché en los diferentes procesadores una vez efectuada la lectura del dato. Como en la operación anterior, también se generará una petición al bus de lectura (`BusRd`) y el controlador de memoria principal suministrará el bloque de memoria en el bus. El estado de las líneas de caché de los procesadores 1 y 2 queda como compartido (S).

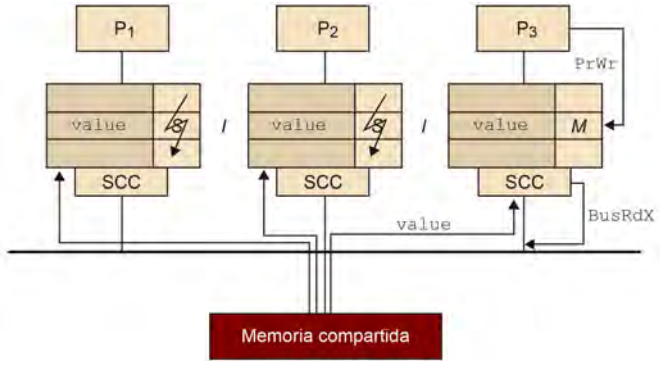
Figura 25. El procesador 2 realiza una petición de lectura.



En la tercera operación ($w3$) del ejemplo el procesador 3 realiza una escritura sobre un bloque de memoria (`PrWr` de la figura 26) que no está en la caché, por lo que se tiene que pedir en el bus.

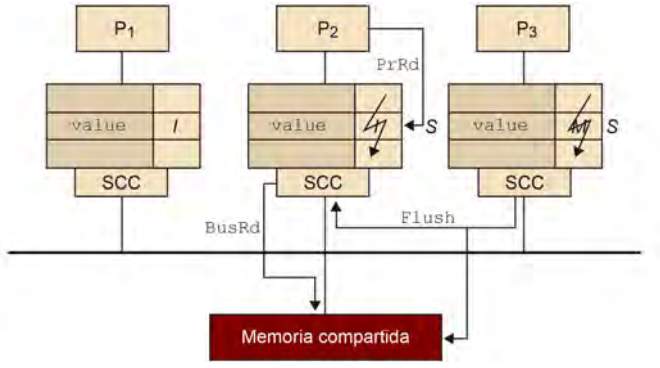
Así, el procesador debe realizar una petición al bus de lectura con intención de escritura `BusRdX` para poder conseguir el bloque de memoria en exclusividad. Los procesadores 1 y 2 observan en el bus que hay una petición de lectura con intención de escritura de un bloque de memoria que tienen en la caché. Por consiguiente, invalidan las líneas de caché que contienen una copia del bloque de memoria que se quiere modificar. ¿Quién suministrará el bloque de memoria al procesador 3? La memoria será la encargada de hacerlo, ya que tiene una copia válida del bloque de memoria que se ha pedido. La figura 26 muestra todas las acciones que se han realizado y el estado en el que quedan las líneas de caché de los procesadores.

Figura 26. El procesador 3 realiza una petición de escritura.



En la siguiente operación ($r2$), el procesador 2 realiza una petición de lectura del bloque de memoria que, al no estar en la caché, deberá realizar una petición de lectura al bus `BusRd`. El procesador 3, que tiene la copia actualizada de la caché, cancelará esta petición de lectura a la memoria principal, y suministrará el bloque de memoria que tiene en su caché mediante una acción de `flush`. Este `flush` del bloque de memoria actualizará tanto la caché del procesador 2, que realizó la petición de lectura, como la memoria principal. La figura 27 muestra la serie de acciones y transacciones a través del bus que se han tenido que efectuar.

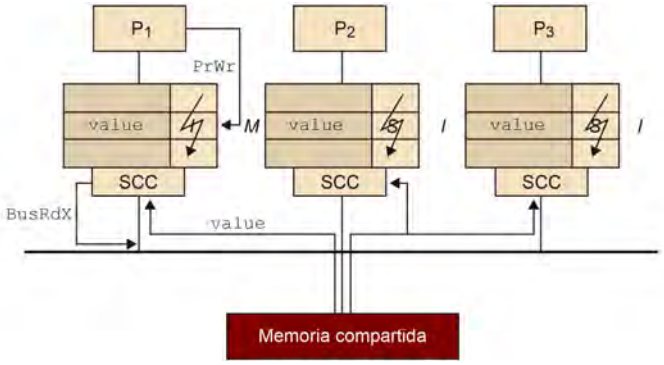
Figura 27. El procesador 2 realiza una petición de lectura.



A continuación, el procesador 1 realiza una escritura del dato (operación $w1$) mediante la petición `PrWr` sobre el bloque de memoria que no se encuentra en caché, teniendo que realizar una petición de lectura con intención de escritura a través del bus (`BusRdX`). La memoria suministrará el bloque de memoria al bus, ya que tiene el bloque actualizado.

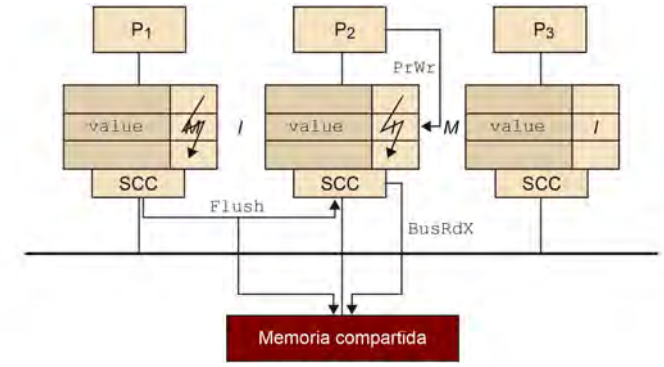
El SCC del procesador 1 lo verá en el bus y actualizará la línea de caché con ese bloque, asignándole el estado de *modified*. El resto de procesadores, al ver en el bus que se está realizando una petición de escritura, invalidarán las líneas de caché que tenían la copias de ese bloque. La figura 28 contempla todas las acciones en el procesador y en el bus.

Figura 28. El procesador 1 realiza una petición de escritura.



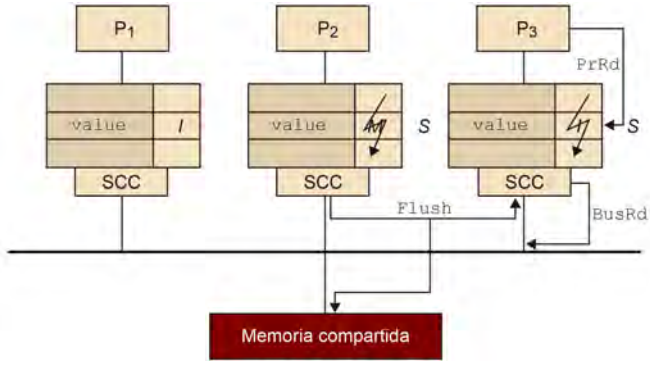
La siguiente operación es $w2$, es decir, el procesador 2 realiza una escritura sobre el bloque de memoria que acaba de modificar el procesador 1 y que él no tiene en caché. El procesador 2, por lo tanto, debe realizar una petición de lectura para modificar el bloque de memoria en el bus (`BusRdX`). El procesador 1, que era el único que disponía de una copia (modificada), observa esta operación que aparece en el bus, cancela la operación de acceso a memoria y deja en el bus el último valor del bloque de datos (`flush` en el procesador 1). Este bloque de memoria se actualizará en memoria y en la caché del procesador 2, quedando la línea de caché donde va el bloque de memoria en el estado de *modified*. La figura 29 visualiza todas estas operaciones mencionadas.

Figura 29. El procesador 2 realiza una petición de escritura.



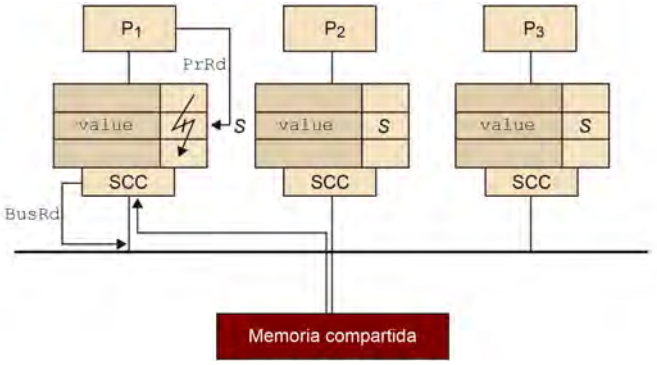
En la penúltima operación del ejemplo, el procesador 3 realiza una lectura del bloque de memoria. Como no tiene ese bloque de memoria en caché, tiene que realizar una petición de lectura en el bus `BusRd`. Será el procesador 2 el que suministre el bloque de memoria, ya que tiene la copia actualizada. Para ello, éste debe cancelar la lectura a memoria principal, y poner el bloque de memoria en el bus con una operación de `flush`. Esta operación actualizará la memoria principal con el último valor y la caché del procesador 3. Las líneas de caché del procesador 3 y del procesador 2 quedan en estado *shared*. La figura 30 muestra las acciones realizadas por el procesador y a través del bus.

Figura 30. El procesador 3 realiza una petición de lectura.



Finalmente, el procesador 1 realiza una lectura del bloque de memoria (operación $r1$), que no está en su caché. Esto hace que deba realizar una petición de lectura del bloque a través del bus `BusRd`. La memoria principal, que está actualizada, suministrará el bloque de memoria. El estado de la línea de caché de cada procesador, que contiene una copia del bloque de memoria, quedará como *shared*. La figura 31 muestra esta última lectura. A partir de esta situación, cualquier lectura sobre ese bloque de memoria no provocará ningún cambio en el estado de las cachés de los procesadores, ni tampoco operaciones en el bus.

Figura 31. El procesador 1 realiza una petición de lectura.



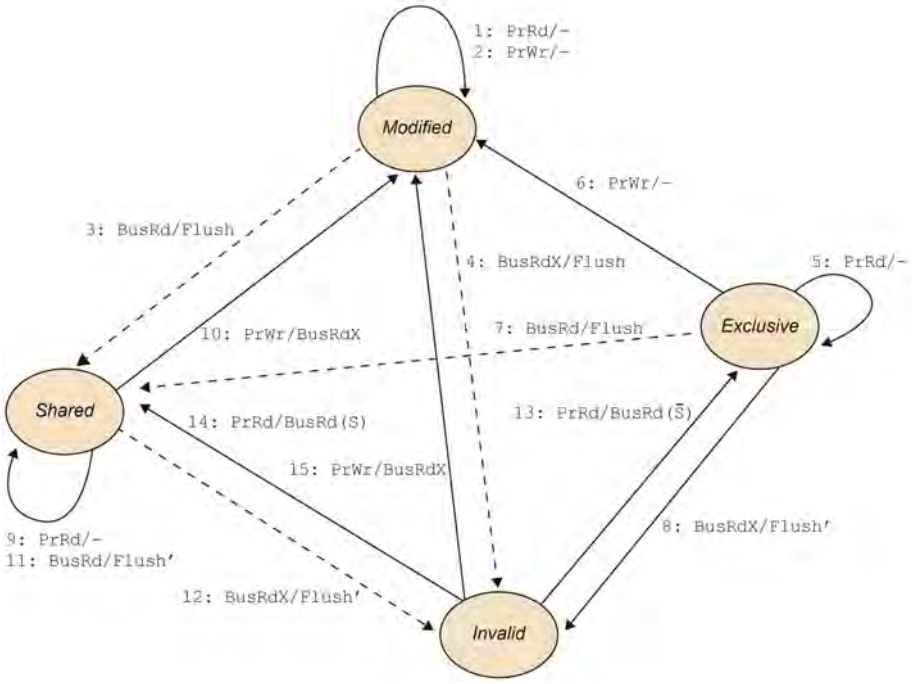
MESI

El protocolo de coherencia MESI tiene un estado más que el protocolo MSI: el de exclusividad. Una línea de caché de un procesador está en este estado si el procesador realiza una lectura de un bloque de memoria y es el único que tiene copia de este bloque. Con este estado se intenta ahorrar operaciones en el bus, ya que al ser la única copia no es necesario informar al resto de procesadores. Por otro lado, a diferencia del protocolo MSI en el que la memoria siempre suministra el bloque de memoria si lo tiene actualizado, en la versión original del protocolo MESI (la versión de Illinois), la memoria sólo suministra un bloque de memoria si ésta es la única copia, independientemente de si está actualizado o no. Por el contrario, si hay una copia del bloque de memoria actualizada en una caché, será ésta la que suministre el bloque de memoria. Esto es lo que se llama la técnica *cache-to-cache sharing*, y su objetivo es suministrar más rápidamente los datos al bus. La técnica *cache-to-cache sharing* requiere que, en caso de que haya más de una caché con una copia del

bloque de memoria actualizado, exista un mecanismo para determinar qué caché dará el bloque de memoria.

El grafo de estados, para un caso donde tenemos *Snoopy* y *write-invalidate*, y utilizando la técnica de *cache-to-cache sharing* de la versión original de Illinois, se muestra en la figura 32. Para simplificar la figura no mostramos las transiciones para operaciones de lectura con fallo de caché.

Figura 32. El grafo de estados de un dato (línea de caché) en un procesador con protocolo de coherencia MESI.



El protocolo MESI tiene los siguientes estados:

1) *Modified*: Este estado indica que la línea de caché contiene un bloque de memoria válido y modificado en la caché. Además, como el bloque de memoria está modificado, no puede haber otra copia de éste en ninguna otra caché de otro procesador ni en la memoria principal.

2) *Exclusive*: Cuando una línea de caché se encuentra en este estado indica que el bloque de memoria es válido en caché, pero ninguna otra caché contiene una copia del bloque de memoria. Además, la memoria principal está actualizada, ya que el bloque de memoria no ha sido modificado. Para poder saber si una lectura de un bloque de memoria se hace en exclusividad, necesitamos una señal (por procesador) que nos permita saber si los procesadores contienen una copia o no del bloque de memoria leído.

3) *Shared*: Una línea de caché con este estado indica que el bloque de memoria es válido, no está modificado, y por consiguiente la memoria principal está actualizada, pero que puede haber una copia del bloque de memoria en otra caché.

4) *Invalid*: En este caso, la línea de caché no tiene ningún bloque de memoria válido, ya sea porque no se cargó nunca o porque se tuvo que invalidar.

Desde el punto de vista de la caché de un procesador del sistema, las transiciones entre los estados que pueden tener cada una de las líneas de la caché son las siguientes:

1) *Modified*: En el caso de que el procesador realice lecturas o escrituras en un bloque de memoria que esté en caché (`PrRd`, acción 1 de la figura, y `PrWr`, acción 2 de la figura, respectivamente), el estado de la línea de caché que lo contiene se mantendrá y, además, no se enviará ninguna operación al bus, ya que sabemos que este bloque de memoria es la única copia existente. Si se observa en el bus una petición de lectura (`BusRd`) sobre el bloque de memoria de la línea de caché (acción 3 de la figura), ésta pasará al estado *shared* y, en caso de una operación de lectura para escritura (`BusRdX`, acción 4 de la figura), la línea de caché pasará al estado inválido. En ambos casos, se hace un `flush` del dato modificado y se cancela el acceso a la memoria, ya que ésta no tiene el bloque de memoria más actualizado.

2) *Exclusive*: En el caso de que el procesador realice una lectura que acierte en la línea de caché (acción 5 de la figura), ésta no cambiará de estado. Además, al ser el bloque de memoria la única copia en el sistema, el procesador no enviará ninguna operación al bus. En cambio, si el procesador realiza una operación de escritura sobre el bloque de memoria de la línea (acción 6 de la figura), ésta pasará al estado *modified* (bloque de memoria modificado). En este caso, el procesador tampoco enviará ninguna operación al bus, ya que sabemos que la línea de caché contiene la única copia de ese dato.

Si el SCC del procesador observa una operación de lectura (`BusRd`, acción 7 de la figura), el procesador, que tiene una copia del bloque de memoria, pondrá el bloque en el bus con una operación (`flush`) para proporcionar el dato al procesador que hizo la petición. La línea de caché pasará al estado de *shared*. En cambio, si se observa una petición de lectura para escritura en el bus (acción 8 de la figura), el procesador suministrará el bloque de memoria, pero en este caso la línea de caché pasará a ser inválida.

3) *Shared*: En el caso de que el procesador realice una petición de lectura del bloque de memoria de la línea (acción 9 de la figura), ésta no cambiará de estado y no se realizará ninguna petición al bus. Si el procesador realiza una petición de escritura sobre esta línea (acción 10 de la figura), el procesador generará una petición de lectura para modificar en el bus (`BusRdX`) y la línea pasará al estado *modified*.

En caso de observar una petición de lectura en el bus (acción 11 de la figura), el procesador, si es el escogido para suministrar el bloque de memoria en el bus (`flush'`), pondrá el bloque de memoria en el bus pero no cambiará el estado de la línea. Este bloque en el bus será leído por el procesador que inició la petición. En el caso de observar una petición de escritura (acción 12 de la figura) se procederá de la misma forma, pero el estado de la línea de caché pasará a ser inválido.

4) *Invalid*: En el caso de que el procesador realice una petición de lectura (acción 13 de la figura), si la petición en el bus de lectura `BusRd` no obtiene como respuesta una señal de alguno de los procesadores indicando que contiene el bloque de memoria (`BusRd(`$\bar{S}$`)`), pasará al estado de exclusividad. En caso contrario (acción 14 de la figura), si se obtiene una señal de alguno de los procesadores `BusRd(`S`)`, el estado pasa a ser *shared*.

Finalmente, si el procesador realiza una escritura sobre el bloque de memoria de la línea (acción 15 de la figura), hará una petición de lectura para escritura `BusRdX` y la línea pasará al estado de *modified*.

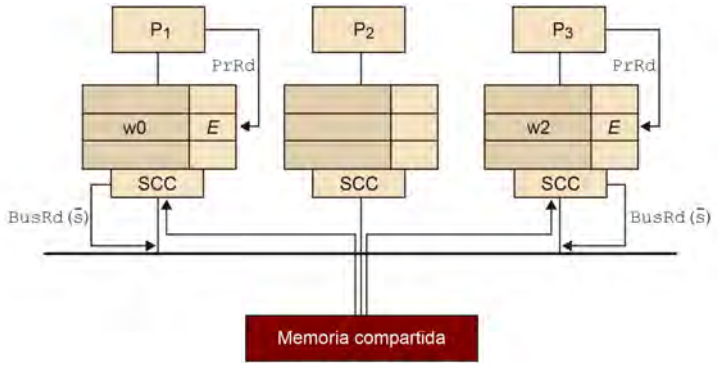
Veremos un ejemplo para acabar de entender el protocolo MESI. En este ejemplo se realizarán los accesos indicados en la tabla 4.

Tabla 4. Secuencia de lecturas y escrituras realizadas por tres procesadores en diferentes tiempos.

Tiempo	Procesador-Operación	Procesador-Operación
1	P_1 - ld w0	P_3 - ld w2
2	P_1 - st w0	P_2 - st w2
3	P_2 - st w2	P_3 - ld w0
4	P_3 - st w0	
5	P_1 - ld w2	
6	P_2 - ld w1	
7	P_3 - ld w1	

Para el primer instante tenemos dos accesos de lectura de dos datos diferentes que van a bloques de memoria diferentes, pero que pueden o no ir a la misma línea de caché. En este caso, si fueran a la caché del mismo procesador, se mapearían en la misma línea de caché. Sin embargo, como lo leen dos procesadores diferentes, van a la misma línea pero de cachés diferentes, tal y como podemos observar en la figura 33. Ambos procesadores, P_1 y P_2, realizan la petición de lectura `PrRd` en el procesador y una petición de lectura en el bus `BusRd`. Como no hay ninguna copia del bloque de memoria en otro procesador, la señal de dato compartido (S) del bus no se activará –la transición será `BusRd(`$\bar{S}$`)`– en el grafo de la figura 32 y, por consiguiente, ambos recibirán el bloque de la memoria (dato) que han pedido y cambiarán el estado de sus respectivas líneas de caché a *exclusive*. Recordemos que los datos estan en bloques diferentes de memoria, de ahí la exclusividad.

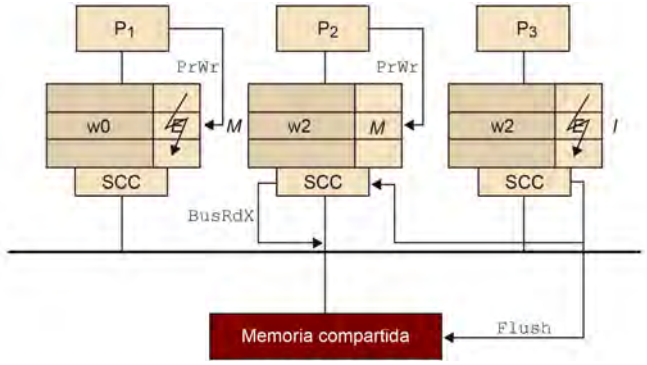
Figura 33. Procesadores 1 y 3 realizan una petición de lectura de los datos w0 y w2, respectivamente.



En el segundo instante (figura 34) se producen dos escrituras sobre los datos leídos anteriormente, w0 y w2, pero ahora son el procesador P_1, que ya dispone del bloque de memoria en su caché, y el P_2, que fallará en su acceso a caché. En el caso del procesador P_1, como ya tiene el bloque de memoria en exclusividad, sólo tendrá que modificar su dato con la petición de escritura del procesador (`PrWr`) y no tendrá que realizar ninguna operación sobre el bus. El estado de la línea de caché pasará a ser *modified*. En cambio, en el

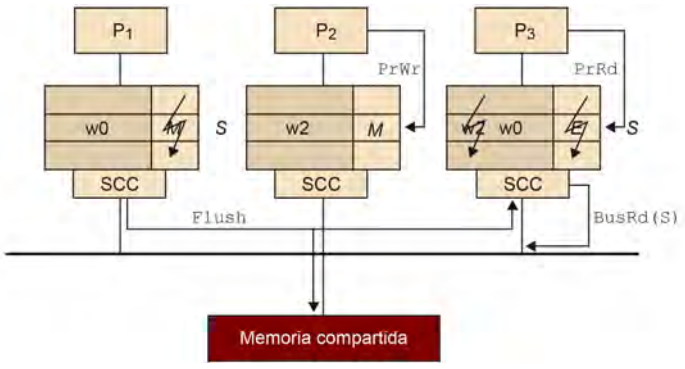
caso del procesador P_2, éste deberá realizar una petición de lectura con intención de escritura `BusRdX` en el bus. Esto hará que el procesador P_3 observe en el bus la petición sobre el bloque de memoria con w2, que él tiene en caché, y por lo tanto, cambiará el estado de su línea de caché a inválido. Además, el procesador P_3 suministrará el bloque de memoria en el bus con una operación de `flush` (técnica *cache-to-cache sharing*).

Figura 34. Procesadores 1 y 2 realizan una petición de escritura de los datos w0 y w2, respectivamente.



Las siguientes operaciones son una escritura del procesador P_2 sobre w2 con acierto en caché, y una lectura del procesador P_3 sobre el dato $w0$ con fallo en caché. Este último dato (bloque de memoria) lo tiene el procesador P_1 en una línea de caché con estado *modified*. En el primer caso, como es una escritura sobre un dato que ya está modificado en la caché del procesador de la petición, sólo se tiene que realizar la operación sobre la línea, sin necesidad de provocar ninguna operación sobre el bus. En el segundo caso, sin embargo, la petición del procesador P_3 necesita realizar una petición en el bus, ya que se ha producido un fallo en caché (línea de caché con estado inválido). Al realizar la petición de lectura sobre el bus (`BusRd`), el procesador P_1, que tiene el bloque de memoria modificado en su caché, cancelará el acceso a memoria, y suministrará el bloque de memoria a la memoria principal y al procesador P_3. Los procesadores P_1 y P_3 cambiarán el estado de sus respectivas líneas de caché a compartido (*shared*). Todas estas operaciones se reflejan en la figura 35.

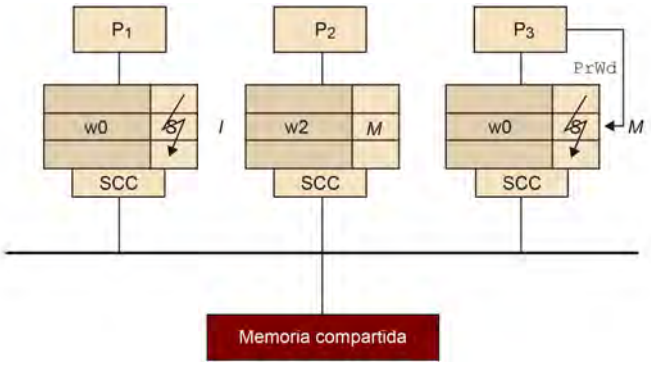
Figura 35. Procesadores 2 y 3 realizan una petición de escritura del dato w2 y de lectura del dato w0, respectivamente.



En el cuarto instante, el procesador P_3 realiza una petición de escritura sobre w0, por lo que el estado de la línea de caché debe cambiar a *modified*. Aparte de esto, ¿qué operaciones a través del bus se deben realizar para que el P_1 pase la línea de caché con ese bloque de

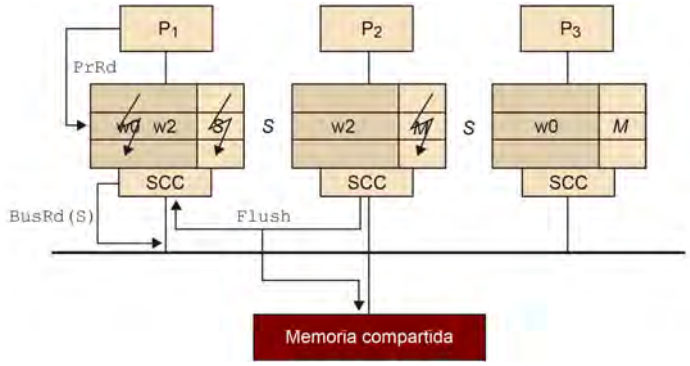
memoria al estado de inválido? En la figura 36 se muestra cómo deberían quedar los estados de los datos leídos hasta el momento. Intentad completar el dibujo con las operaciones sobre el bus.

Figura 36. Procesador 3 realiza una petición de escritura del dato w0.



En el quinto instante, el procesador P_1 realiza una lectura de w2 con fallo en caché (tenía otro bloque de memoria en caché, el del dato w0). Por lo tanto, el procesador deberá realizar una petición de lectura sobre el bus (`BusRd`). La señal de compartición S del bus se activará como respuesta a esta petición de lectura, ya que el procesador P_2 tiene una copia del bloque de memoria. Además, el procesador P_2 suministrará el bloque de memoria en el bus (`flush`) para que el procesador P_1 y la memoria principal puedan actualizarlo. Las líneas de caché con el bloque de memoria de w2 de los procesadores pasan a tener el estado *shared*. La figura 37 refleja todas estas operaciones realizadas en el procesador y en el bus.

Figura 37. Procesador 1 realiza una petición de lectura del dato w2.

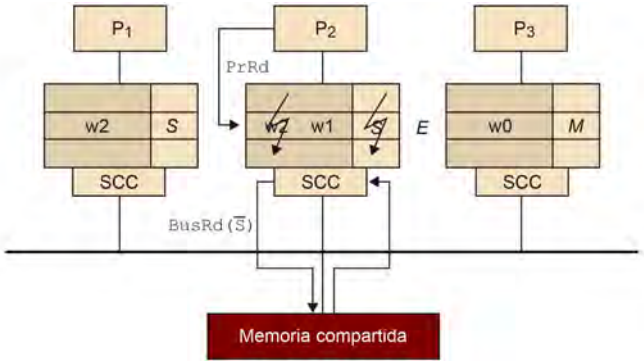


En este momento, tenemos el dato w2 en estado *shared* en las cachés de los procesadores P_1 y P_2, y el dato w0, en estado *modified* en la caché de P_3.

A continuación (sexto instante) el procesador P_2 realizará una lectura de w1 con fallo en caché (tenía otro bloque de memoria en caché, el del dato w2). Por lo tanto, el procesador P_2 deberá realizar una petición de lectura al bus (`BusRd`). Al no haber ninguna otra copia de este bloque de memoria, la señal de compartición (S) restará inactiva en el bus (`BusRd(`$\bar{S}$`)`), y el procesador P_2 pasará la línea de caché con el nuevo bloque de memoria a estado *exclusive*. La figura 38 muestra los estados de las diferentes líneas de cachés de

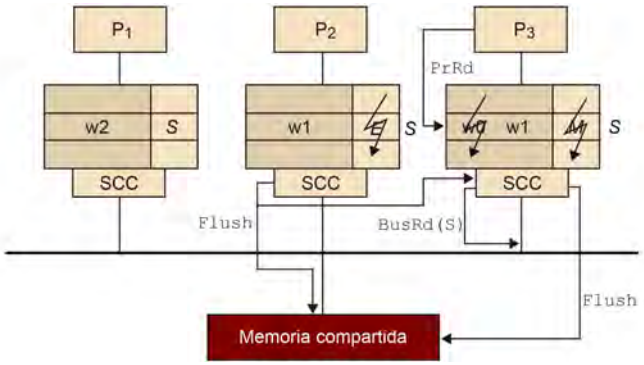
los procesadores. En esta situación, ¿el procesador P_1 debería pasar la línea de caché con el dato w2 a un estado de exclusividad?

Figura 38. El procesador 2 realiza una petición de lectura del dato w1.



Finalmente, el procesador P_3 realiza una lectura del dato w1 con fallo en caché. Como consecuencia de este fallo el procesador P_3 realizará una petición de lectura al bus. El procesador P_2, que tiene el bloque de memoria modificado en su caché, cancelará la lectura de memoria, pondrá el bloque de memoria en el bus (`flush`) para que el procesador P_2 y la memoria principal puedan actualizarse, e indicará que él dispone de una copia con la señal de compartición S. Tanto el procesador P_2 como el procesador P_3 cambiarán el estado de sus respectivas líneas de caché, con el dato w1, a estado compartido. La figura 39 muestra el contenido de las cachés y el estado de éstas.

Figura 39. El procesador 3 realiza una petición de lectura del dato w1.



El Intel i7 usa una variante del protocolo MESI, llamado MESIF que añade un nuevo estado (*Forward*) para indicar qué procesador de los que comparten un dato debería responder una petición.

Protocolo MESIF

El multiprocesador Intel i7 incorpora un sistema basado en directorios en la caché L3 con un variante del protocolo MESI llamado MESIF.

True y *false sharing*

En el momento en que tenemos un sistema multiprocesador con memoria con coherencia de caché podemos observar dos tipos de conflictos en la compartición de los datos: *true sharing* y *false sharing*.

El *true sharing** de ellos es el resultado de estar accediendo al mismo dato desde dos o más procesadores. Esto provoca, tal y como hemos visto en los protocolos de coherencia, que el dato se tenga que actualizar (*write-update*) o invalidar (*write-invalidate*) si se realiza alguna escritura al dato. Si se realizan muchas escrituras al dato desde diferentes procesadores esto puede provocar que los accesos al bus y el mantenimiento del sistema de coherencia de caché se conviertan en un cuello de botella.

*** *True sharing*: se produce cuando se comparte el mismo dato de memoria.**

El segundo de estos conflictos, *false sharing**, surge como consecuencia de mantener la coherencia de caché a nivel de un bloque de memoria (línea de caché) y no de dato. Así, dos datos diferentes, accedidos por procesadores diferentes, pueden compartir un mismo bloque de memoria y, entonces, compartirán la coherencia del mismo bloque. Por ejemplo, si un procesador P_1 lee uno de estos datos que comparten bloque de memoria, el estado de su línea de caché será *exclusive* o *shared* dependiendo de si es el único procesador con copia o no de este bloque. Si ahora otro procesador P_2 escribe en otro dato diferente, pero en el *mismo* bloque de memoria, éste provocará que se tenga que invalidar la línea de caché del procesador P_1, ya que el procesador P_2 va a modificar un dato del mismo bloque memoria. Esto es lo que se llama *false sharing* o compartición falsa, porque lo que se comparte es el bloque de memoria y no el mismo dato.

*** *False sharing*: se produce cuando no se comparte el mismo dato de memoria, pero sí el mismo bloque de memoria, con lo que dos procesadores pueden estar luchando por la exclusividad de ese bloque de memoria.**

La forma de evitar este tipo de situaciones, en el que uno de los procesadores debe escribir y por consiguiente puede haber invalidaciones innecesarias, es intentar que los datos a los que se accede por diferentes procesadores no estén consecutivos en memoria para evitar que estén en el mismo bloque de memoria (línea de caché).

Un ejemplo de la situación mencionada podría ser el caso de tener un vector de enteros, cuyos elementos son actualizados frecuentemente por diferentes procesadores. Esto puede provocar muchos fallos de caché debido a las invalidaciones por compartición falsa.

Analizaremos cómo se puede producir *false sharing* en el momento de realizar la paralelización con `OpenMP` del código 2.1, que es el código secuencial del producto escalar.

```
1  static long num_steps = 100000;
2  void main ()
3  {
4          int i;
5          double x, pi, step, sum = 0.0;
6
7          step = 1.0/(double) num_steps;
8
9          for (i=1;i<= num_steps; i++) {
10                 x = (i-0.5)*step;
11                 sum = sum + 4.0/(1.0+x*x);
12         }
13         pi = step * sum;
14 }
```

Código 2.1: Código secuencial del producto escalar.

Observamos que la variable `sum` debería actualizarse dentro de una exclusión mutua o bien con `atomic`, ya que debería ser actualizada en paralelo por todos los *threads*. Para evitar esta sincronización, se podría pensar en que cada *thread* sumara una varia-

ble global distinta, que después el *thread master* pudiese leer y obtener la suma total, con la suma de las sumas parciales de los resultados obtenidos en cada *thread*.

El código 2.2 es la versión OpenMP, usando un vector de reales para realizar las sumas parciales por parte de cada *thread*.

```
#include "omp.h"
#define NUM_THREADS = 4
static long num_steps = 100000;
void main ()
{
  int i, id;
  double x, pi, step, sum[NUM_THREADS];

  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);

  #pragma omp parallel private(id)
  {
    id = omp_get_thread_num();
    sum[id] = 0.0;

    #pragma omp for private(x,i)
    for (i=1;i<= num_steps; i++) {
       x = (i-0.5)*step;
       sum[id] += 4.0/(1.0+x*x);
    }

    #pragma omp single
    for (i=0, pi=0.0; i<NUM_THREADS; i++)
       pi += step * sum[i];
  }
}
```

Código 2.2: Código `OpenMP` del producto escalar.

Analizando un poco cómo se ejecutará el código, y sabiendo que varios reales del vector `double sum[NUM_THREADS]` puede estar en el mismo bloque de memoria (línea de caché), cada *thread* podrá estar provocando una invalidación de las líneas de caché del resto de procesadores debido a que todos están actualizando el mismo bloque de memoria. Esto provocará fallos de caché y un elevado tráfico a través del bus, que es innecesario.

¿Cómo podemos solucionar el problema de *false sharing* en este ejemplo? Una posible y sencilla solución es añadiendo *padding* al vector, de tal manera que obligue a que los elementos del vector que se usen para calcular la suma parcial estén en bloques de memoria (líneas de caché) diferentes. El código 2.3 muestra la declaración con el *padding* suponiendo que los bloques de memoria son de 64 bytes.

Padding

Padding es una técnica por la que se añaden elementos *basura* que sirven para separar elementos, conseguir un alineamiento concreto de los datos, o conseguir que una estructura ocupe un determinado número de bytes.

```
        ...
        double sum[NUM_THREADS][8];
        ...
```

Código 2.3: Ejemplo de *padding*.

De esta forma podríamos utilizar el primer elemento de `sum[id][0]`, sabiendo que, como tenemos 8 `doubles` por línea de caché (bloque de memoria), no tendremos ahora *false sharing*.

3. Multicomputador

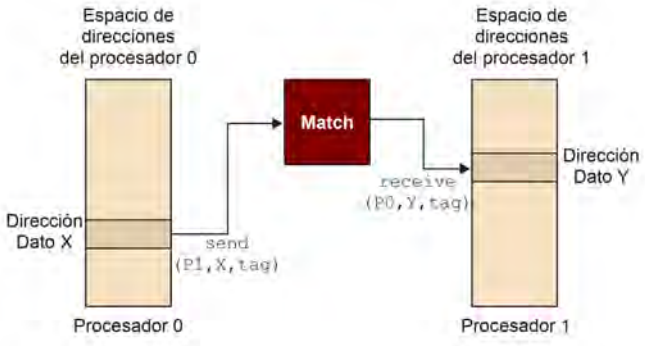
Los multicomputadores, tal y como ya se ha comentado en la introducción, surgieron de la necesidad de poder escalar en número de procesadores sin un elevadísimo coste económico. Estos consisten en un conjunto de procesadores y bancos de memoria que se conectan a través de una red de interconexión con una determinada topología de red.

La principal característica de los multicomputadores es que los procesadores, al más bajo nivel, ya no pueden compartir datos a través de `loads/stores`, sino que lo tienen que hacer por medio de mensajes. Es por eso por lo que en este tipo de sistemas no hay problemas de coherencia de caché ni de consistencia de memoria. Aquí, la sincronización se hace explícita con los mensajes, tal y como muestra la figura 40, con primitivas estilo `send` y `receive`. En el `send` se especifíca el buffer a enviar y a quién se envía. Por el otro lado, el `receive` debe especificar el buffer de recepción y de quién lo recibe. Los buffer del `send` y del `receive` están en espacios de direcciones diferentes, suponiendo que son procesos diferentes. Opcionalmente, también se puede especificar una etiqueta o *tag* al mensaje, para recibir un mensaje concreto de un procesador concreto, cumpliéndose la *matching rule*.

Nota aclarativa

En los sistemas multiprocesadores, al más bajo nivel, también se podría considerar que se están realizando mensajes, un mensaje formado por la línea de caché, pero es generado por una instrucción de acceso a memoria.

Figura 40. Comunicación con primitivas de comunicación `send` y `receive`



Una pregunta que nos podemos hacer es: ¿quién debe realizar la comunicación entre procesos? Normalmente es el programador el que utiliza un *message-passing paradignm*; es decir, el programador usará una API de una librería de usuario, que exporta el modelo de comunicación al programador. Sin embargo, en determinados sistemas, puede haber un *software DSM (distributed-shared memory)*, que consiste en una capa de *software* transparente a los programadores y que oculta la característica de memoria distribuida, implementando una memoria compartida en *software*. En un sistema con DSM, el sistema operativo tiene un papel importante, ya que el funcionamiento se suele basar en los fallos de página en los accesos. Esto implica que los programadores pueden trabajar con accesos `load/store` para acceder a los datos de un proceso que está en otro nodo sin utilizar de forma explícita el modelo de paso de mensajes. A bajo nivel, este *software*, tras detectar un

fallo de página en el acceso a un dato con `load/store` por parte de un procesador, utilizará el modelo de paso de mensajes para comunicar páginas de memoria de una memoria física a otra.

3.1. Redes de interconexión

Las redes de interconexión permiten la comunicación de datos entre nodos (procesadores), o entre procesadores y memoria tal y como vimos en las redes de interconexión para los sistemas multiprocesadores en el apartado anterior. Los procesadores y/o los bancos de memoria se conectarán a las entradas y salidas de la red de interconexión.

Normalmente, la redes de interconexión están formadas por *switches* y *links* conectados los unos a los otros. La capacidad de comunicación de los *links* depende de sus características físicas (*capacity coupling* y fortaleza de señal) que en parte dependen, a su vez, de la longitud del *link*. Los *switches* pueden ser más o menos sofisticados, pudiendo contener *buffering* para desacoplar el puerto de salida del puerto de entrada, encaminamiento para reducir la congestión, y *multicast* para realizar *broadcast* de una misma salida. El mapeo de los puertos de entrada a los puertos de salida puede variar según diferentes mecanismos, y este mecanismo y el grado del *switch* influyen en el coste final de estos *switches*.

Switch

En una red de interconexión, un *switch* consiste en un dispositvo *hardware* con una serie de puertos de entrada y otros de salida. El número total de puertos de un *switch* es el grado de este *switch*. Los puertos de entrada están conectados con los de salida mediante un *crossbar* que se puede configurar.

Para conectar los nodos a los *switches* se necesita una interfaz de red. Esta interfaz debe soportar anchos de banda más elevados que los necesarios para conectarse con entrada/salida, ya que los buses de memoria suelen ser mucho más rápidos. La interfaz de red normalmente tiene la responsabilidad de empaquetar datos, hacer el enrutamiento de los paquetes, hacer *buffering* de los datos de entrada y salida, control de errores, etc.

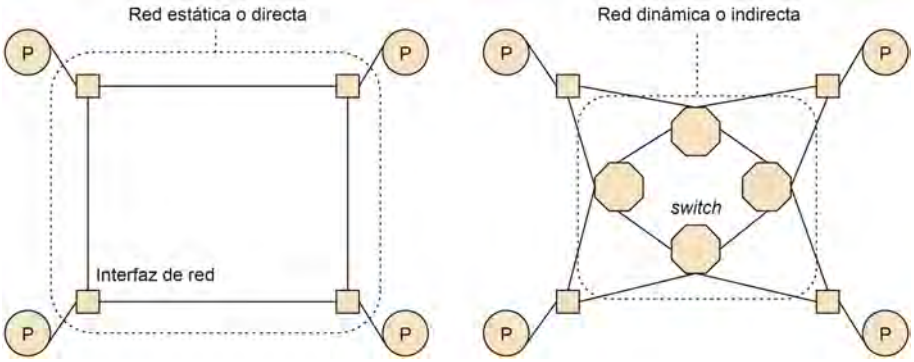
Las redes de interconexión se pueden dividir en *static* o directas, y *dynamic* o indirectas.

- **Directas**: Las redes directas son aquellas en las que los nodos de procesamiento están conectados punto a punto vía *links*.

- **Indirectas**: En este caso los procesadores puede que no estén conectados punto a punto, y los *links* pueden conectar varios *switches* de tal forma que se pueda llegar a establecer diferentes caminos entre los nodos de procesamiento y entre los nodos y los bancos de memoria.

La figura 41 muestra un ejemplo de una red de interconexión directa (izquierda) e indirecta (derecha) formada por 4 procesadores conectados punto a punto y vía *switches* respectivamente.

Figura 41. Ejemplos de una red de interconexión directa y otra de tipo indirecta, ambas formadas por 4 procesadores.



3.1.1. Métricas de análisis de la red

En este subapartado analizaremos algunas de las medidas que se utilizan para evaluar una red de interconexión. En el subapartado siguiente analizaremos algunas redes de interconexión con diferentes formas de conectarse.

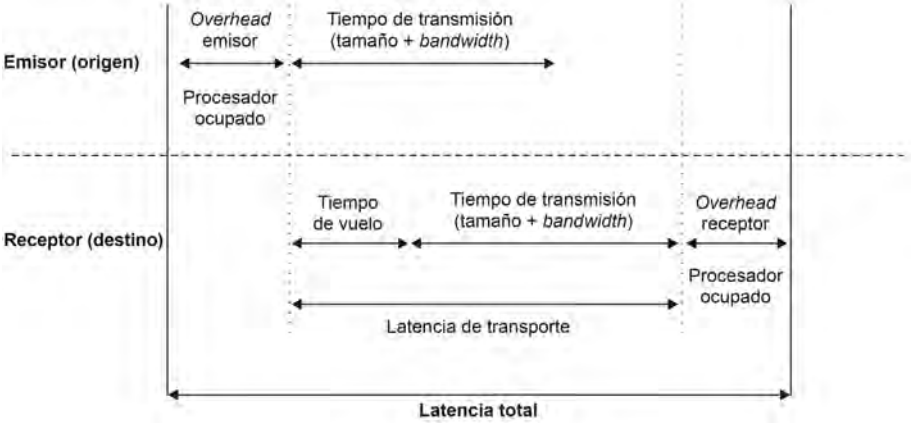
Latencia y ancho de banda de la red

La latencia de la red indica el tiempo que se necesita para realizar la comunicación de un solo dato de un procesador a otro. La latencia de los mensajes en la red cobra importancia en aquellos programas que tienen que realizar un elevado número de mensajes pequeños.

El ancho de banda de la red es la cantidad de datos por unidad de tiempo que se puede mantener en la comunicación de un conjunto de datos. La importancia del ancho de banda de una red es mayor en los programas que realizan un elevado número de mensajes grandes.

En cualquier caso, queremos hacer notar que la latencia máxima que ofrece una red, según el fabricante, muchas veces no se alcanza debido que hay una serie de costes que se pagan al realizar la comunicación. Por lo tanto, la latencia que el programador observará es la latencia de la red de interconexión más la latencia del *software* que se utiliza para realizar la comunicación. La figura 42 refleja las diferentes partes que afectan al envío de un dato.

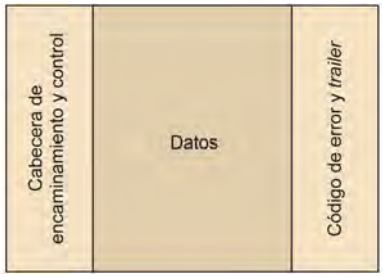
Figura 42. Latencia de comunicación.



Por un lado, vemos que el procesador que envía tiene que pagar un tiempo en preparar el mensaje y colocarlo en la red para que se empiece a transmitir (*overhead* del emisor), después hay un tiempo de transmisión, el tiempo que se necesita para que llegue un mensaje de un procesador al otro (tiempo de vuelo) y, finalmente, hay un coste de recepción del mensaje por parte del procesador destino (*overhead* del receptor).

Por otro lado, el ancho de banda que conseguimos en el envío de datos a través de la red normalmente es menor al máximo que nos ofrece la red de interconexión según el fabricante. Esto es debido a que hay que empaquetar los datos e incluir en ellos el enrutamiento, control de errores, cabecera, etc., tal y como muestra la figura 43. El ancho de banda que realmente conseguiremos es el que se suele llamar el *effective bandwidth*.

Figura 43. Información adicional (*overhead*) para poder enviar un mensaje.



Fanout o *Connectivity* de la red

El número de conexiones en un nodo se llama el grado (matemáticamente), *fanout* o *connectivity* según los ingenieros. El *fanout* puede indicarnos qué tolerancia a fallos tiene el procesador, ya que cuanto mayor sea el grado de un nodo más caminos posibles tendremos en el momento de enrutar la comunicación. Además, tener un grado elevado permite reducir la contención de la red en algunos caminos.

Diameter de la red

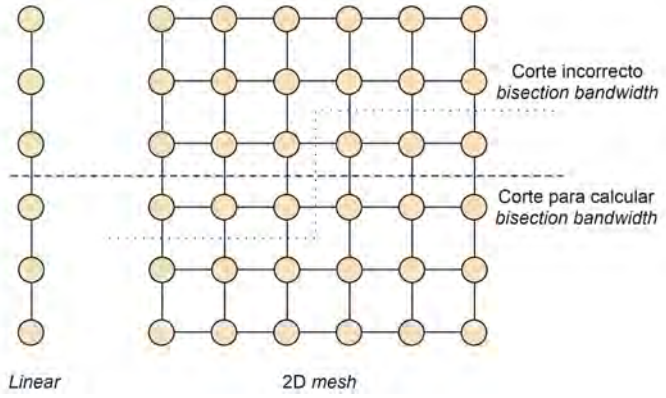
El *diameter* es la distancia máxima entre dos nodos, siendo la distancia entre dos nodos el número de *links* que tenemos que atravesar para ir de un nodo al otro. El *diameter*, por consiguiente, nos da una visión de cuánto nos costará el envío de un nodo a otro, ya que cada *link* que pasamos puede significar un pequeño coste de comunicación adicional. La distancia media entre nodos también nos puede dar una idea de lo que nos cuesta cada comunicación, es decir, de la latencia de un mensaje.

Bisection bandwidth

El *bisection bandwidth* nos da una medida de la cantidad de datos que se pueden enviar en las transmisiones sin tener contención en la red de interconexión. Para calcular el *bisection bandwidth* se divide la red de interconnexión en dos partes, eliminando el mínimo número

de *links* entre ambas partes, de tal forma que las dos partes tengan el mismo número de nodos. El número de *links* eliminados, multiplicado por el ancho de banda de cada uno de ellos, nos determina el *bisection bandwidth*. En la figura 44 mostramos cuál es el corte correcto para calcular el *bisection bandwidth* de una red de interconexión *linear array* (izquierda) y otra de tipo *2D mesh* (derecha). Los puntos son nodos o *switches*, y las líneas que los unen son los *links*. Notad que también mostramos un ejemplo de lo que no sería un corte correcto para calcular el *bisection bandwidth*, ya que se eliminan más *links* de los necesarios.

Figura 44. Ejemplo de corte de una red en el cálculo del *bisection bandwidth* de un *linear array* y de un *2D mesh*.



Muchos diseñadores de redes de interconexión intentan realizar topologías de red que maximicen esta medida de las redes, con la idea de minimizar la contención.

Dimensionalidad

La dimensionalidad es la cantidad de opciones que se tienen para ir de un nodo a otro. Si de un nodo a otro no hay más que una posibilidad de camino, entonces se dice que es cero dimensional. Sin embargo, si podemos movernos hacia el este o hacia el oeste para ir a un determinado nodo, entonces diremos que es uno dimensional. Si podemos, además, ir hacia el norte o hacia el sur, diremos que es dos dimensional.

3.1.2. Topología de red

La topología de red es un patrón en el cual los *switches* están conectados a otros *switches* mediante *links*. Las interconexiones se muestran con grafos donde los nodos representan o bien procesadores o *switches*, y las aristas los *links*. La topología de red influye en la latencia de los mensajes, en el *bandwidth* conseguido, y en la congestión que pueda haber en las comunicaciones, tal y como se ha podido reflejar en las diferentes medidas de rendimiento que se han comentado en el subapartado anterior.

A continuación analizaremos diversas topologías de red, tanto directas como indirectas. En las figuras que aparezcan en este subapartado sólo mostraremos los *links* y los *switches* en-

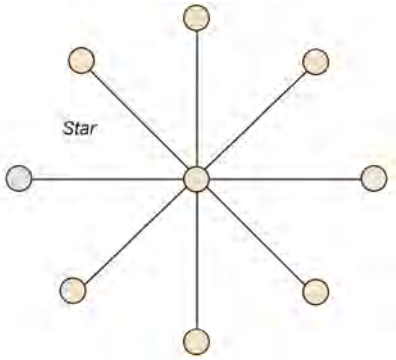
tre ellos. No se mostrarán ni las memorias ni los procesadores, que suelen estar conectados con una interfaz de red a los *switches*.

Las redes de interconexión basadas en un bus, el *crossbar* y el *multistage* se detallaron en el apartado 2, dedicado a los multiprocesadores. Ahora nos centraremos en las siguientes redes de interconexión: *star*, *full interconnect*, *linear and meshes* y *tree interconnect*.

Star

La topología de red *star* es una tipología cero dimensional. Consiste en un nodo central que está conectado al resto de nodos en el exterior. Esta topología de red es muy sencilla de implementar pero tiene dos inconvenientes principalmente: (1) el nodo central se puede convertir en un cuello de botella en las comunicaciones, y (2) si el nodo central falla, el resto de nodos se quedarían desconectados. La figura 45 muestra un ejemplo de topología *star*.

Figura 45. Topología de red de tipo *star*.



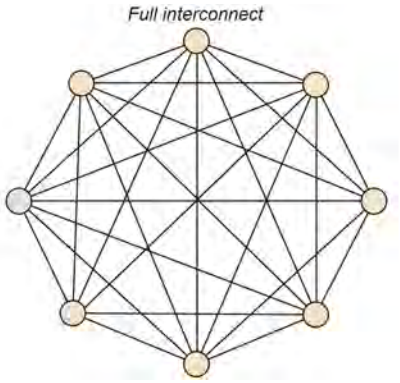
Full interconnect

La *full interconnect* es una cero dimensional también porque para ir de un nodo a otro sólo hay un camino. A diferencia de la anterior, ésta es completamente tolerante a fallos, maximiza el *bisection bandwidth*, y minimiza el *diameter*. Su principal desventaja es que es muy poco escalable. La figura 46 muestra una red de interconexión con topologia *full interconnect* o *completely-connected*. Este tipo de red es la topología de red directa equivalente a la indirecta *crossbar*.

Linear and meshes

Estos tipos de red aparecieron como solución al elevado coste que suponían las redes de interconexión totalmente conectadas. Tenemos de diferentes tipos: *linear arrays*, *meshes* o *grids* sin o con conexiones en las esquinas (*rings*), y finalmente *cubes* e *hypercubes*.

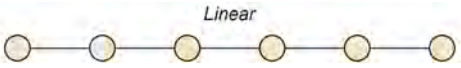
Figura 46. Topología de red de tipo *full interconnect*.



Linear array es cero dimensional, ya que para ir de un nodo a otro sólo lo podemos hacer por un camino. Cada nodo, a excepción de los nodos de los extremos, tienen un vecino a la derecha y otro a la izquierda. El *diameter* viene determinado por la distancia entre los dos nodos de los extremos.

Una posible extensión del *linear array* es la *ring interconnect*. Ésta es uno dimensional al poder ir a la izquieda o a la derecha en el momento de ir de un nodo a otro. El *diameter* en este caso es la mitad que en el *linear array*, ya que la distancia más larga entre nodos es entre uno de los extremos y el nodo central. La figura 47 muestra un ejemplo de *linear array*.

Figura 47. Topología de red de tipo *linear*.

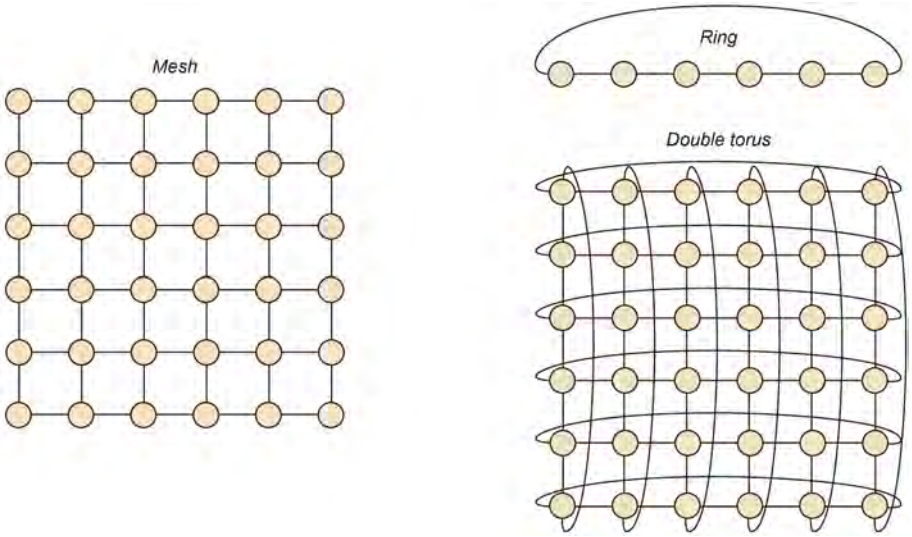


Grid o mesh es la extensión de los *linear arrays* en una o más dimensiones. Este tipo de interconexión es 2 o más dimensional, ya que cada nodo, a excepción de los nodos en los extremos, tiene como mínimo 4 conexiones con 4 nodos vecinos (derecha, izquierda, arriba y abajo). Cada dimensión del *mesh* tiene $\sqrt{P}$ nodos, donde P es el número de nodos. El diámetro de una red de este tipo crece a razón del número de nodos por dimensión, al ser la distancia más larga entre nodos que la que hay entre nodos colocados en esquinas opuestas. Estos sistemas son muy usados, ya que son fácilmente escalables, además de que las aplicaciones con cálculos de una estructura regular son fácilmente mapeadas en estas redes de interconexión.

Una variante del *mesh* es añadir conexiones entre los nodos de los extremos en cada dimensión. Esta variante, *double torus*, es más tolerante a fallos y reduce el diámetro, ya que las esquinas están conectadas y por consiguiente ya no son los nodos más alejados, tal y como pasaba en el caso del *ring* en comparación con el *linear array*. La figura 48 muestra los tipos de red de interconexión *mesh*, *ring* y *double torus*.

Una clase especial de *mesh* es la red de interconexión *cube*. Esta red de interconexión es una *mesh* de 3 dimensiones. En general es una *3D mesh* con k nodos por dimensión. En la figura 49.a se muestra una con 2 nodos por dimensión. Dos cubos replicados, y

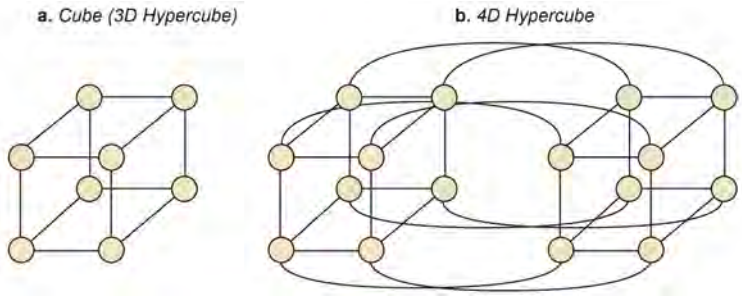
Figura 48. Topologías de red de tipo *mesh*, *ring* y *double torus*.



conectados en cada uno de sus nodos, forman lo que sería un *four-dimensional cube*, y así sucesivamente podríamos formar *five-dimensional cube*, etc. En general, este tipo de cubos se llaman *hypercube*.

Los *hypercubes* son una buena elección para sistemas de alto rendimiento, ya que el diámetro crece linealmente con el número de dimensiones (log_2(el número de nodos)). Es decir, en un 8-dimensional *hypercube*, con 2^8 procesadores, sólo necesitaríamos 8 pasos, como máximo, para ir de un procesador a otro. Este diámetro es la mitad que una *2D mesh* de 16 por 16, que tiene el mismo número de nodos que el *hypercube*, pero que tiene un diámetro de 16. La desventaja de estos *hypercubes* es su coste económico, ya que el *fanout* necesario es muy elevado. La figura 49.b muestra un ejemplo de red de interconexión de tipo *hypercube*.

Figura 49. Topologías de red de tipo *cube* e *hypercube*.

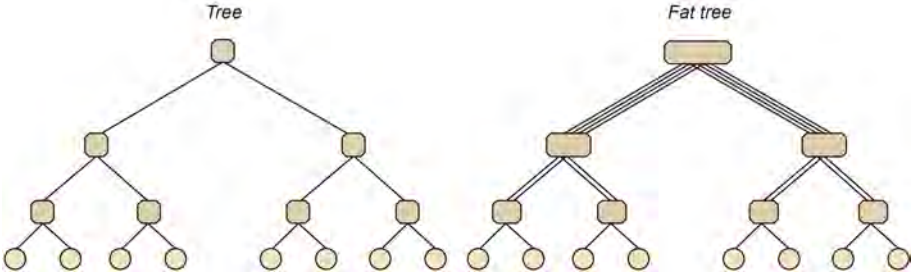


Tree interconnect

La red de interconexión *tree interconnect* tiene un *bisection bandwidth* de uno, que se corresponde con el único *switch* que separa un subárbol del otro, el *switch root* del árbol. Por consiguiente, el nodo raíz del árbol se puede convertir en un cuello de botella en las comunicaciones. Una forma de solucionarlo es por medio de ampliar el número de *links* a medida que nos acercamos al nodo raíz, teniendo tantos *links* como hojas en ese subárbol. Esto aumenta el *bisection bandwidth* hasta $n/2$, reduciendo así el cuello de botella que

suponía el *switch root*. Este tipo de topología de red, donde se aumenta el número de *links* a medida que nos acercamos al nodo raíz, se llama *fat tree*. La figura 50 muestra un ejemplo de topología en árbol, y *fat tree*.

Figura 50. Topologías de red de tipo *tree* y *fat tree*.



Diámetro y *bisection bandwidth* de algunas topologías de red

La tabla 5 muestra el diámetro y el *bisection bandwidth* de algunas de las redes directas (parte superior de la tabla) e indirectas (parte inferior), expresadas en número de *links* que tenemos que eliminar.

Tabla 5. Relación de topologías de red con su *diameter* y *bisection bandwidth* expresado en número de *links*.

Topología	Diámetro	*bisection bandwidth* (*#links*)
1D line	$P-1$	1
1D ring	$\frac{P}{2}$	2
2D mesh	$2 \times (\sqrt{P}-1)$	$\sqrt{P}$
2D torus	$2 \times \lfloor \sqrt{P}/2 \rfloor$	$2 \times \sqrt{P}$
D-hypercube	$d = \log P$	$\frac{P}{2}$
Tree	$2 \times \log P$	1
Fat tree	$2 \times \log P$	$\frac{P}{2}$
Omega	$\log P$	$\frac{P}{2}$

Primero comentaremos las directas. El diámetro de la *linear array* o *1D line* es $P-1$, que consiste en llegar de un extremo al otro de la red. En cambio, si tenemos un *1D ring*, al tener una conexión entre los dos extremos, el camino más largo es justamente al nodo central, estando éste a una distancia de $\frac{P}{2}$. El *bisection bandwidth* de estas dos redes es 1 y 2 respectivamente, que son los *links* que debemos eliminar para desconectar una y otra mitad.

En el caso de la red *2D mesh* ya comentamos el diámetro en su momento, y es resultado de ir de una esquina del *2D mesh* a la esquina opuesta. Por consiguiente, tenemos que recorrer $\sqrt{P}-1$ en una dimensión y otros tantos en la otra, suponiendo que tenemos P nodos. En el caso del *2D torus*, al igual que pasaba al pasar de *linear array* a *1D ring*, los extremos en cada dimensión están conectados. Esto significa que de una esquina a su opuesta sólo necesitamos dos pasos. Por consiguiente, los nodos a mayor distancia son aquellos que están en una esquina y el nodo que se encuentra en medio del *2D torus*. Esto significa que se tiene que recorrer $\lfloor \sqrt{P}/2 \rfloor$ nodos en una dimensión y lo mismo en la otra, hasta llegar al nodo central. Para calcular el *bisection bandwidth* en el caso de *2D mesh* se debe cortar por la mitad el *mesh*. Esto significa desconectar tantos *links* como nodos en una dimensión,

es decir, $\sqrt{P}$. En el caso del *torus* también se deben eliminar los *links* que conectan los extremos. Es decir, su *bisection bandwidth* es el doble del *mesh*.

Finalmente, en el caso de las redes de interconexión directas, para calcular el diámetro de la red *D-hypercube* seguiremos el mismo razonamiento que para un *2D mesh*. Tenemos que ir de un extremo a otro en cada dimensión. En el caso de un *hypercube* como el de la figura, eso significa tener que recorrer 1 nodo en cada dimensión (hay 2 nodos por dimensión). Así, para las d dimensiones, tendremos que recorrer d nodos. En cuanto al cálculo del *bisection bandwidth* podemos verlo como la interconexión de dos $(d-1)$ *hypercube*, tal y como se muestra en la figura 49. Por consiguiente, se tienen que eliminar tantos nodos como los que tenga uno de estos $d-1$ *hypercube*, que son 2^{d-1}, es decir $\frac{P}{2}$.

En cuanto a las redes de interconexión indirectas, la red *tree* tiene un diámetro que corresponde a la distancia existente entre un nodo de un subárbol del *tree* al otro subárbol; es decir, subir $\log P$ niveles y bajar otros tantos. Para el caso del *fat tree* tenemos el mismo camino, y por consiguiente, el mismo diámetro. Por otro lado, el *bisection bandwidth* para el *tree* es sólo 1, ya que sólo necesitamos cortar el *link* de la raíz del árbol. En cambio, para el *fat tree*, al aumentar el número de *links* el *bisection bandwidth* se aumenta hasta $\frac{P}{2}$.

Finalmente, el diámetro de la red de interconexión *Omega* es igual al número de estapas que tiene la red, es decir, $\log_2 P$. En cuanto al *bisection bandwidth*, éste es la mitad del número de *links* que van de la parte superior a la inferior. Es la mitad debido a que sólo la mitad de los *links* pueden estar activos a la vez, por temas de conflictos en el camino que siguen los mensajes o accesos.

3.2. Comunicaciones

En los sistemas multicomputador, tal y como hemos podido observar, necesitamos realizar comunicaciones entre procesos.

El tiempo de comunicación del mensaje de m elementos de un procesador emisor a un procesador destino (comunicación punto a punto), sin contar los costes de inicialización del mensaje en el procesador emisor, y los de recepción en el procesador receptor, es de:

$$T_{comm} = t_s + m \times t_w$$

donde t_s es el tiempo de inicialización del mensaje y t_w es el tiempo de transmisión por palabra enviada (o elemento, para simplificarlo).

Este modelo básico es una simplificación de otro modelo de comunicación que considera otros factores de la red. Los factores de la red que normalmente se distinguen en modelos de comunicación más completos son:

Nota

En este módulo trabajaremos con un modelo simplificado de comunicación punto a punto:

$$T_{comm} = t_s + m \times t_w.$$

En este modelo despreciamos el t_h y consideramos un enrutamiento *cut-through* del mensaje.

* ***Start up*** **en inglés**

1) t_s o tiempo de inicialización* (t_s) consiste en el tiempo de preparar el mensaje para enviarlo a la red, determinar el camino del mensaje a través de la red, y el coste de la comunicación entre el nodo local y el *router*.

2) Tiempo de transmisión del mensaje, que a su vez consta de:

- por *byte* (t_w): tiempo de transmisión de un elemento (word).

- por *hop* (t_h): tiempo necesario para que la cabecera del mensaje se transmita entre dos nodos directamente conectados a la red, pero que no consideraremos en nuestro modelo simplificado, ya que t_h no es significativo para mensajes pequeños ni para mensajes grandes en comparación con t_s y t_w, respectivamente.

Además, estamos suponiendo que tenemos un tipo de red de interconexión que usa un enrutamiento *cut-through* del mensaje, donde todas las partes de un mensaje tienen el mismo enrutamiento e información de error.

Por otra parte, las aplicaciones suelen utilizar comunicaciones colectivas. Éstas son comunicaciones entre más de un procesador con unos determinados patrones de comunicación. La implementación eficiente de estas comunicaciones colectivas en diferentes arquitecturas es importante para obtener aplicaciones paralelas eficientes. A continuación detallamos algunos de los algoritmos de comunicación que se siguen para implementar de forma eficiente estas comunicaciones colectivas, y el coste de comunicación de cada una de ellas, basándonos en el modelo básico de comunicación punto a punto.

3.2.1. Coste de las colectivas

Ahora vamos a analizar los costes de las comunicaciones colectivas más frecuentemente usadas (*one-to-all [broadcast], all-to-one [reduction], scatter, gather*) en tres redes de interconexión: *linear array*, *2D mesh* e *hypercube*. En la elaboración de este análisis vamos a suponer, al igual que para la comunicación punto a punto, que el número de *links* no afecta al coste de un mensaje, además de suponer que no tenemos contención en ninguno de los *links*. En otro caso, el coste de comunicación de la colectiva podría ser mayor. También supondremos que los *links* entre los nodos son bidireccionales, es decir, dos nodos directamente conectados pueden enviar un mensaje a través de ese *link* en paralelo. Sin embargo, un nodo no puede recibir dos mensajes por el mismo *link*.

one-to-all (*broadcast*) y *all-to-one* (*reduction*)

Figura 51 muestra un esquema básico de las dos colectivas, de las que una es la dual de la otra.

La colectiva de comunicación *one-to-all broadcast*, tal y como indica su nombre, realiza la comunicación de un dato de tamaño m a todos los procesos involucrados en la colectiva, desde un único proceso fuente. De esta forma, al final de la comunicación habrá P copias del dato m, siendo P el número de procesos en la comunicación.

Figura 51. Esquema básico de las comunicaciones colectivas *one-to-all* y *all-to-one*.

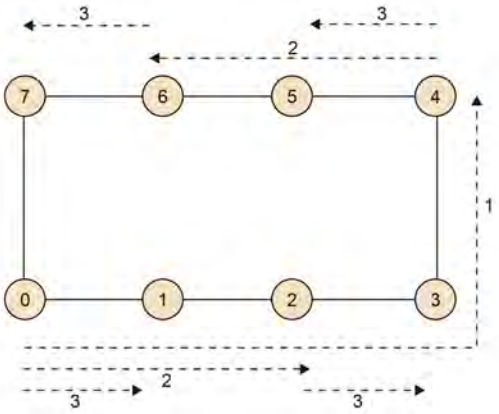
La colectiva de comunicación *all-to-one reduction* es la inversa al *one-to-all broadcast*. En este caso, la colectiva implica también una operación asociativa de los datos a comunicar. Por ejemplo, una posible reducción podría ser la suma de los datos de todos los procesos en la comunicación. Cada proceso haría la suma local y todos llamarían a la colectiva para poder realizar la suma de los resultados de la suma local a cada proceso, y dejar el resultado de esa suma en un único proceso.

1) Coste en un *linear array/ring*

La solución más sencilla de enviar un mensaje a todos es enviando $P-1$ mensajes desde el procesador fuente a los otros $P-1$ procesadores. Sin embargo, ésta no es una solución eficiente si el número de procesadores es muy elevado. Una solución más eficiente es la de intentar conseguir que, a cada paso de comunicación, se vayan doblando los procesadores que reciben los datos en el caso de *one-to-all broadcast* y que se dividan por la mitad en el *all-to-one reduction*. Esta estrategia se llama *recursive doubling*.

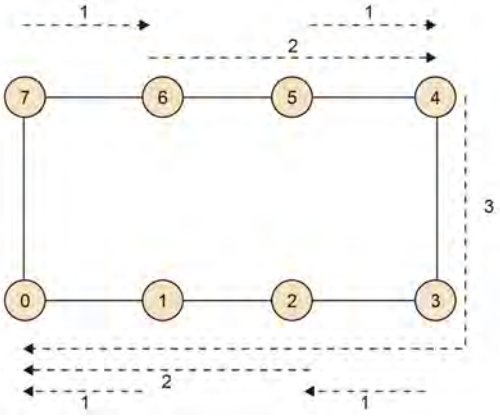
La figura 52 muestra el detalle de cómo se realizarían las comunicaciones en una red de interconexión *ring* de 8 nodos. En la figura se muestran los mensajes con flechas. Y para cada flecha se indica en qué paso del algoritmo se realiza esta comunicación. Así, para la comunicación colectiva *broadcast* observamos que en el paso 1 la comunicación va del procesador 0 al 4, en el siguiente paso son el 0 y el 4 los que envían al 2 y al 6, y finalmente, en el último paso (el paso $\log 8$), los procesadores harán una comunicación a sus respectivos vecinos, completando así el *broadcast*.

Figura 52. Comunicaciones en un *ring* para la realización de un *one-to-all*.



En la figura 53 mostramos el ejemplo de hacer un *all-to-one reduction*. En este caso los nodos impares empiezan realizando una comunicación a su vecino, después hay una comunicación cada dos, y finalmente una comunicación del nodo 4 al nodo 0, acabándose así la reducción. Como se puede observar, son exactamente los pasos inversos a una comunicación *one-to-all broadcast*.

Figura 53. Comunicaciones en un *ring* para la realización de un *all-to-one*.



Por consiguiente, el coste de comunicación del *one-to-all broadcast* y del *all-to-one reduction* es de:

$$T_{comm} = \log P \times (t_s + m \times t_w)$$

siendo P el número de procesadores, y m el tamaño del dato a realizar el *broadcast* o bien el tamaño del dato que tiene cada procesador al inicio de la comunicación colectiva *all-to-one reduction*.

2) Coste en un *mesh*

Las dos comunicaciones colectivas en una red de interconexión *mesh* se pueden realizar en dos pasos:

1) Realizar la comunicación colectiva en una de las filas de la red de interconexión *mesh*.

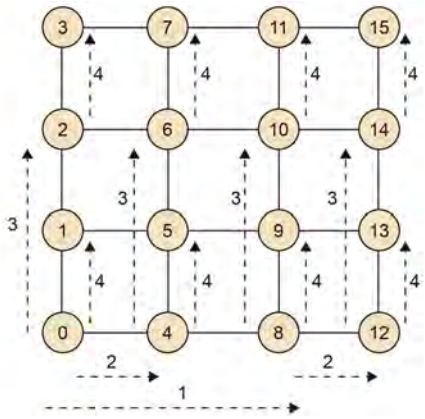
2) Aplicar la comunicación colectiva a cada columna, una vez realizada la anterior colectiva en una de las filas.

Es decir, en dos pasos de *one-to-all broadcast* o bien de *all-to-one reduction*, completaríamos la comunicación colectiva.

La figura 54 muestra los mensajes que se realizan para un *one-to-all broadcast* para una *mesh* de 4×4 nodos, indicando en qué paso se realiza cada uno de ellos. Primero se realiza

el *one-to-all broadcast* en la fila del *mesh* con nodos 0, 4, 8 y 12. Posteriormente se aplica a cada columna en paralelo.

Figura 54. Comunicaciones en un *mesh* para la realización de un *one-to-all*.



El coste de comunicación es dos veces el coste en una red de interconexión *ring* de $\sqrt{P}$ procesadores (el número de procesadores en cada dimensión en una red de tipo *mesh*):

$$T_{comm} = 2 \times \log \sqrt{P} \times (t_s + m \times t_w)$$

siendo P el número de procesadores, y m el tamaño del dato a realizar el *broadcast* o bien el tamaño del dato que tiene cada procesador al inicio de la comunicación colectiva *all-to-one reduction*.

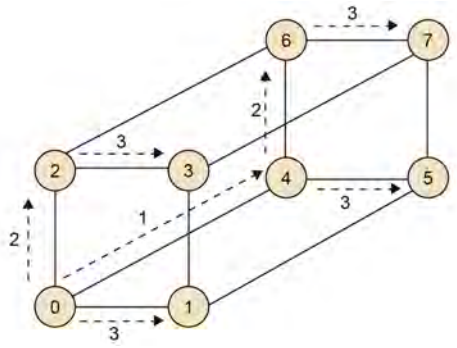
3) Coste en un *hypercube*

El caso del *hypercube* es un caso especial de *mesh* con 2^d nodos, donde d es el número de dimensiones de la *mesh*, y el 2 viene del hecho de tener dos nodos por dimensión. Por consiguiente, podemos aplicar el mismo algoritmo que hemos aplicado en la *mesh*: primero una dimensión, después en paralelo todos los nodos en la siguiente dimensión, después la siguiente, etc. hasta llegar a realizar las d dimensiones. La figura 55 muestra los mensajes en una comunicación colectiva *one-to-all broadcast*. La etiqueta de cada comunicación indica en qué momento se realiza cada comunicación punto a punto.

El coste de comunicación es d veces el coste en una red de interconexión *ring* de 2 procesadores:

$$T_{comm} = d \times \log 2 \times (t_s + m \times t_w) = d \times (t_s + m \times t_w)$$

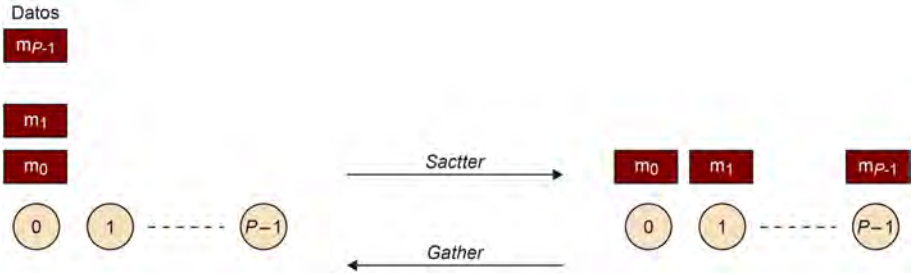
Figura 55. Comunicaciones en un *3D hypercube (cube)* para la realización de un *one-to-all*.



Scatter* y *gather

En una comunicación colectiva *scatter* un nodo distribuye equitativamente un buffer de tamaño $P \times m$ entre todos los procesadores que realizan la comunicación colectiva. Es decir, cada procesador i recibirá m elementos consecutivos del buffer, empezando en la posición $i \times m$ del buffer. La figura 56 muestra un esquema básico de esta comunicación. La comunicación colectiva *gather* es la inversa a la operación *scatter*.

Figura 56. Esquema básico de las comunicaciones colectivas *scatter* y *gather*.



Los pasos de comunicación en la operación *scatter* son iguales que en el *broadcast*, pero en el caso de este último, el tamaño del mensaje se mantiene constante (m), mientras que para el *scatter* va reduciéndose.

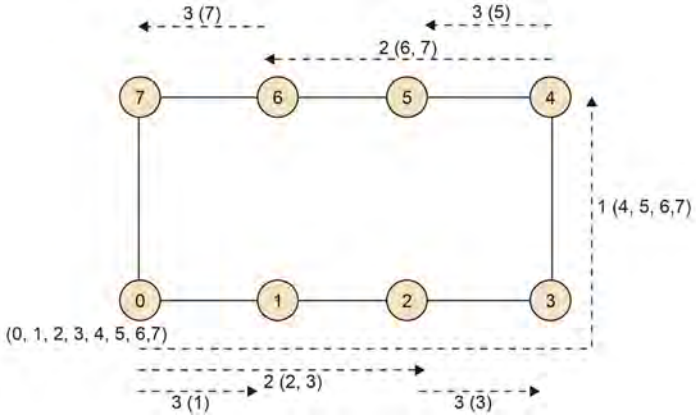
El *gather* es una operación equivalente al *reduction*, pero también varía el tamaño del mensaje. En el caso de la comunicación *gather*, el mensaje va aumentando a medida que vamos haciendo pasos de comunicación.

1) Coste en un *linear array/ring*

Al igual que para el caso del *one-to-all*, la solución más sencilla sería la de enviar la parte del *buffer* que le corresponde a cada procesador, de tamaño m, con un mensaje *point-to-point*. Pero esta solución no es eficiente para un número elevado de procesadores. Una solución más eficiente es seguir la misma estrategia que seguimos para el *one-to-all broadcast*: intentar conseguir que, a cada paso de comunicación, se vaya doblando los procesadores que reciben los datos. Ahora, sin embargo, el mensaje inicial es de tamaño $\frac{P}{2} \times m$, y a cada paso de comunicación se reducirá por la mitad.

La figura 57 muestra al detalle cómo irían las comunicaciones en una red de interconexión *ring* de 8 nodos. En la figura se muestran los mensajes con flechas. Y para cada flecha se indica en qué paso del algoritmo se realiza esta comunicación, y los datos que se comunican. En el ejemplo de la figura, el procesador 0 debe realizar el *scatter* de 8 datos, que se muestran en la figura junto al procesador 0. En el primer paso de comunicación el procesador enviará un mensaje de tamaño $\frac{P}{2} \times m$ al procesador 4. En el siguiente paso, los nodos 0 y 4 enviarán la mitad de los datos $\frac{P}{2^2} \times m$, y así sucesivamente hasta llegar al paso $\log P$, donde el mensaje será de tamaño $\frac{P}{2^{\log P}} \times m$, es decir, m.

Figura 57. Comunicaciones en un *ring* para la realización de un *scatter*.



En el caso de la comunicación *gather* tendríamos el proceso inverso en las comunicaciones.

Por consiguiente, el coste de comunicación de *scatter*/*gather* es de:

$$\begin{aligned} T_{comm} &= \sum_{i=1}^{\log P} (t_s + \frac{P}{2^i} m \times t_w) \\ T_{comm} &= t_s \log P + t_w m(P-1) \end{aligned}$$

siendo P el número de procesadores.

2) Coste en un *mesh*

Para las redes de interconexión *mesh* podemos seguir la misma estrategia que seguimos para las operaciones de *broadcast* y *reduction*: aplicar primero el *scatter*/*gather* en una fila y después en todas las columnas.

El coste de comunicación de ambas operaciones *scatter*/*gather* es dos veces el coste en una red de interconexión *ring* de $\sqrt{P}$ procesadores:

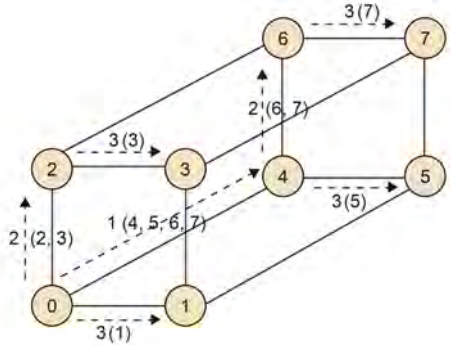
$$T_{comm} = 2 \times (t_s \log \sqrt{P} + t_w m(\sqrt{P} - 1))$$

siendo P el número de procesadores. Para la operación *scatter* m es el tamaño de la parte que se tiene que enviar a cada procesador del total de datos comunicados $P \times m$. Y para la operación *gather* es el tamaño del dato que tiene cada procesador al inicio de la colectiva, para acabar dejando un mensaje de tamaño $P \times m$ en el procesador destino del *gather*.

3) Coste en un *hypercube*

La figura 58 muestra la realización de una operación *scatter* en un *3D hypercube* de 8 nodos, y enviando un único elemento a cada procesador. El caso del *D hypercube* es equivalente a la red de interconexión *mesh*, al ser esta red de interconexión una *mesh* de dimensión d y dos nodos por dimensión. Como se puede observar en la figura, en cada paso se envía la mitad de los datos al nodo vecino que está en su misma dimensión. Así, a cada paso, el tamaño del mensaje que envía cada nodo se divide por dos, hasta conseguir que cada procesador tenga un mensaje de tamaño m, que se corresponde con 1 elemento en el caso de la figura.

Figura 58. Comunicaciones en un *hypercube* para la realización de un *scatter*.



La operación *gather* es la operación inversa equivalente, que se empezaría con los procesadores con un mensaje de tamaño m en cada procesador y, siguiendo el camino inverso de la figura, se iría doblando el tamaño del mensaje a cada paso de comunicación.

El coste de comunicación del *scatter/gather* es d veces el coste en una red de interconexión *ring* de 2 procesadores por dimensión:

$$
\begin{aligned}
T_{comm} &= d \times (t_s \log P + t_w m(P-1)) \\
T_{comm} &= d \times (t_s \log 2 + t_w m(2-1)) \\
T_{comm} &= d \times (t_s + t_w m)
\end{aligned}
$$

Resumen

En este módulo hemos descrito la clasificación según Flynn, ampliándola con las subcategorías que A. Tanenbaum describió en su libro sobre estructuras de computadores, para posteriormente centrarnos en las dos principales subcategorías de las máquinas MIMD: multiprocesadores o máquinas de memoria compartida (fuertemente acoplados) y multicomputadores o de memoria distribuida (débilmente acoplados).

Para los multiprocesadores hemos descrito, principalmente, cómo se conectan con la memoria (bus, *crossbar*, *multistage*, etc.), los problemas de consistencia de memoria que se plantean cuando hay varios procesadores leyendo y escribiendo sobre una misma memoria, y el problema de la coherencia de caché debido a la existencia de varias copias de un mismo dato en el sistema. En el caso del problema de consistencia de memoria, se han analizado diferentes modelos de consistencia que fijan las reglas del orden de las lecturas y escrituras que se deberían ver por parte de todos los procesadores. Para el caso de la coherencia de caché, se ha descrito en detalle el funcionamiento de dos protocolos para mantener la coherencia de caché: MSI y MESI, y los mecanismos *hardware* de los que se dispone para conseguir esa coherencia.

En cuanto a los multicomputadores, éstos no tienen el problema de consistencia ni de coherencia de caché, ya que la compartición de los datos se hace mediante el paso de mensajes. Estos mensajes tienen un coste que nosotros hemos modelado de forma sencilla, y que nos han ayudado a detallar el coste de las comunicaciones en redes de interconexión con diferentes topologías de red, que también hemos analizado en este módulo. En particular, hemos diferenciado entre redes de interconexión directas e indirectas, y hemos descrito las diferentes formas de medir el rendimiento de estas redes, y cómo pueden afectar al rendimiento de nuestras aplicaciones.

Bibliografía

Grama, Ananth; Gupta, Anshul; Karypis, George y Kumar, Vipin (2003). *Introduction to Parallel Computing*. Pearson: Addison Wesley.

Hennessy, John L. y Patterson, David A. (1996) *Computer Architecture: A Quantiative Approach*. Morgan Kaufmann.

Tanenbaum, Andrew S. (2006). *Structured Computer Organization*. Pearson: Addison Wesley.