Multiprocessadors i multicomputadors

Daniel Jiménez-González

PID_00215408

UOC
Universitat Oberta de Catalunya

Els textos i imatges publicats en aquesta obra estan subjectes llevat que s'indiqui el contrari a una llicència de Reconeixement-NoComercial-SenseObraDerivada (BY-NC-ND) v.3.0 Espanya de Creative Commons. Podeu copiar-los, distribuir-los i transmetre'ls públicament sempre que en citeu l'autor i la font (FUOC. Fundació per a la Universitat Oberta de Catalunya), no en feu un ús comercial i no en feu obra derivada. La llicència completa es pot consultar a http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.ca.

Índex

Introducció 5

Objectius 8

1. Classificació 9
1.1. Arquitectures paral·leles SIMD 9
1.1.1. *Array processors* 9
1.1.2. *Vector processors* 10
1.2. Multiprocessadors 10
1.2.1. UMA 10
1.2.2. NUMA 12
1.2.3. COMA 13
1.3. Multicomputadors 13
1.3.1. MPP 14
1.3.2. *Clusters computers* 15
1.4. Top500 16

2. Multiprocessador 18
2.1. Connexions típiques a memòria 18
2.1.1. Basades en un bus 18
2.1.2. Basades en *crossbar* 20
2.1.3. Basades en *multistage* 21
2.2. Consistència de memòria 22
2.2.1. Definició 22
2.2.2. Consistència estricta o *strict consistency* 23
2.2.3. Consistència seqüencial o *sequential consistency* 23
2.2.4. Consistència del processador o *processor consistency* 24
2.2.5. Consistència *weak* 24
2.2.6. Consistència *release* 25
2.2.7. Comparació dels models de consistència 25
2.3. Coherència de memòria cau 27
2.3.1. Definició 27
2.3.2. Protocols d'escriptura 28
2.3.3. Mecanisme de maquinari 29
2.3.4. Protocols de coherència 35

3. Multicomputador 51
3.1. Xarxes d'interconnexió 52
3.1.1. Mètriques d'anàlisi de la xarxa 53
3.1.2. Topologia de xarxa 55
3.2. Comunicacions 59

3.2.1. Cost de les col·lectives 60

Resum 66

Bibliografia 67

Introducció

A pesar que els monoprocessadors han anat incorporant suport maquinari per a explotar els diferents nivells de paral·lelisme i millorar el rendiment de les aplicacions (paral·lelisme a escala d'instrucció o ILP, a escala de dades o DLP, i a escala de *threads* o TLP), els *speedups* assolits en aquestes aplicacions no han estat significativament grans, ja sigui per motius tecnològics o bé per les necessitats concretes de les aplicacions.

Una altra via per a aconseguir *speedups* més significatius és crear sistemes de computació més grans aprofitant la millora tecnològica i combinant múltiples monoprocessadors, els quals poden aplicar o no la mateixa instrucció sobre múltiples dades. Els sistemes que apliquen una mateixa instrucció a múltiples dades anomenen *computadors SIMD** i, en el cas que apliquin diferents instruccions a les dades, reben el nom de MIMD**.

*** SIMD:** ***single instruction multiple data.***
**** MIMD:** ***multiple instruction multiple data.***

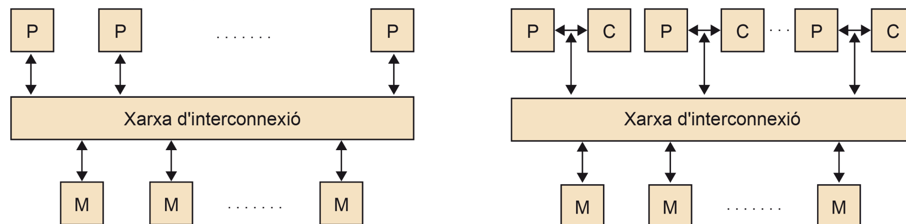
Les màquines SIMD van donar resposta a aplicacions científiques i d'enginyeria que feien operacions sobre estructures molt regulars, com poden ser *arrays* o vectors. Els computadors paral·lels *array processors* i *vector processors* són els més coneguts. La característica més important d'aquest tipus de màquines és que apliquen una mateixa operació (instrucció) sobre una seqüència de dades, ja sigui tenint una sèrie de processadors o unitats funcionals que operen alhora (*array processors*), o bé amb una unitat funcional que opera seqüencialment sobre una seqüència de dades (*vector processors*).

Les màquines o computadors MIMD combinen múltiples processadors que no necessàriament han de fer la mateixa instrucció o operació sobre les dades. Aquests sistemes amb múltiples processadors se solen classificar segons si tenen memòria compartida o només distribuïda: els de memòria compartida són els multiprocessadors, i els de memòria distribuïda, els multicomputadors. Els multiprocessadors també se solen anomenar fortament acoblats, i els multicomputadors, feblement acoblats.

Els multiprocessadors es caracteritzen pel fet que tots els processadors tenen un espai virtual d'adreces de memòria comuna. En aquests, els processadors només necessiten fer un `load`/`store` per a accedir a qualsevol posició de la memòria. D'aquesta manera, els processadors es poden comunicar simplement per mitjà de variables compartides, i faciliten així la programació. Els multiprocessadors més coneguts són els anomenats *symetric multiprocessors* o SMP, que es caracteritzen per ser tots els processadors del mateix tipus i poder accedir de la mateixa manera als sistemes de memòria i entrada/sortida. La figura 1 mostra dos esquemes bàsics de l'arquitectura d'un multiprocessador, en què tots els processadors comparteixen la memòria principal. En la part esquerra de la figura es mostra un sistema en el qual els processadors no tenen memòria cau i, a la dreta, un sistema en el qual sí que tenen memòria cau. Les memòries cau es van introduir per a explotar millor la localitat de les dades, i

com a conseqüència, reduir la contenció d'accés a la memòria compartida. No obstant això, aquestes memòries cau introdueixen un problema de coherència entre les possibles còpies d'una dada concreta en les diferents memòries cau, que analitzarem en aquest mòdul. En aquesta mateixa figura, independentment de si el sistema té memòries cau o no, podem observar que hi ha una xarxa d'interconnexió que connecta els processadors amb la memòria. Aquesta xarxa pot ser un simple bus o quelcom més complex.

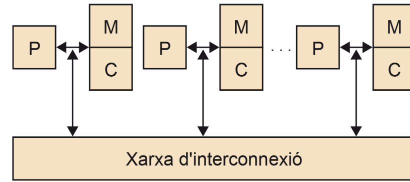
Figura 1. Multiprocessador de memòria compartida sense memòria cau (esquerra) i amb memòria cau (dreta)



En els sistemes multiprocessador hi sol haver un únic sistema operatiu amb una única taula de pàgines i una taula de processos. No obstant això, pot ser que el sistema operatiu hagi de treballar amb més d'una taula de pàgines. Això ocorre amb els multiprocessadors de tipus *distributed shared memory* o DSM, en els quals cada node de la màquina té la seva pròpia memòria virtual i la seva pròpia taula de pàgines. En aquests sistemes, una pàgina està localitzada en una de les memòries associades a un node. En el moment d'un error de pàgina, el sistema operatiu demana la pàgina al processador que la té local perquè la hi enviï. En realitat, la gestió és com un error de pàgina en el qual el sistema operatiu ha de buscar la pàgina en disc.

Els multicomputadors, tal com hem comentat, es caracteritzen per tenir la memòria distribuïda. En aquests sistemes els processadors tenen espais físics i virtuals diferents per a cada processador. Per tant, cada processador pot accedir amb *loads* i *stores* a la seva memòria local, però no a les memòries d'altres processadors. Així, si un processador ha d'accedir a una dada localitzada en la memòria d'un altre processador, aquests s'hauran de comunicar via missatge a través de la xarxa d'interconnexió. Això dificulta notablement la programació d'aquests sistemes. No obstant això, fins i tot tenint aquest desavantatge significatiu pel que fa als multiprocessadors, els sistemes multicomputador tenen el gran avantatge que són molt més econòmics de muntar que els multiprocessador, ja que es poden construir connectant processadors de caràcter general per mitjà d'una xarxa d'interconnexió estàndard, sense necessitat de ser a mesura. Això, a més, els fa més escalables. La figura 2 mostra un sistema multicomputador, en què cada processador té la seva memòria local propera i una connexió a la xarxa d'interconnexió per a poder comunicar-se amb la resta de processadors, i així, poder accedir a les dades d'altres memòries.

Figura 2. Esquema bàsic de memòria d'un multicomputador



Finalment, ens podem trobar amb sistemes MIMD híbrids: un sistema multicomputador amb memòria distribuïda entre els diferents nodes que el formen, connectats per mitjà d'una xarxa d'interconnexió, i dins de cada node, tenir un sistema multiprocessador amb memòria compartida, normalment amb una connexió basada en bus. Aquests sistemes combinen els avantatges esmentats abans: programabilitat i escalabilitat.

L'organització del mòdul és la següent: en l'apartat 1 farem una classificació dels computadors SIMD i MIMD. Entrarem en més detall en els sistemes multiprocessador i multicomputador en els apartats 2 i 3 respectivament. En l'apartat 2, subapartat 2.1., analitzarem les connexions típiques a la memòria que ens podem trobar en aquests sistemes. Posteriorment, descriurem els problemes derivats de tenir diferents bancs de memòria accessibles en paral·lel, i de la incorporació de les memòries cau en els sistemes multiprocessador: el problema de la consistència en el subapartat 2.2. i el de la coherència en el subapartat 2.3., respectivament.

En l'apartat 3 descriurem els sistemes multicomputador. En el subapartat 3.1. d'aquest veurem algunes de les xarxes d'interconnexió usades en la connexió dels processadors en aquests sistemes. Finalment, en el subapartat 3.2., analitzarem algunes de les comunicacions col·lectives utilitzant un model de comunicació bàsic.

Objectius

Els objectius generals d'aquest mòdul didàctic són els següents:

1. Conèixer les característiques de les principals arquitectures SIMD i MIMD.
2. Conèixer les diferents xarxes d'interconnexió per als multiprocessadors (memòria compartida) i multicomputadors (memòria distribuïda).
3. Saber distingir entre el problema de consistència de memòria i de coherència de memòria.
4. Conèixer els diferents models de consistència de memòria.
5. Conèixer els diferents mecanismes i protocols per a mantenir la coherència de memòria.
6. Conèixer i saber utilitzar les diferents mètriques de caracterització de les xarxes d'interconnexió.
7. Saber mesurar el cost de col·lectives de comunicació en diferents topologies de xarxa.

1. Classificació

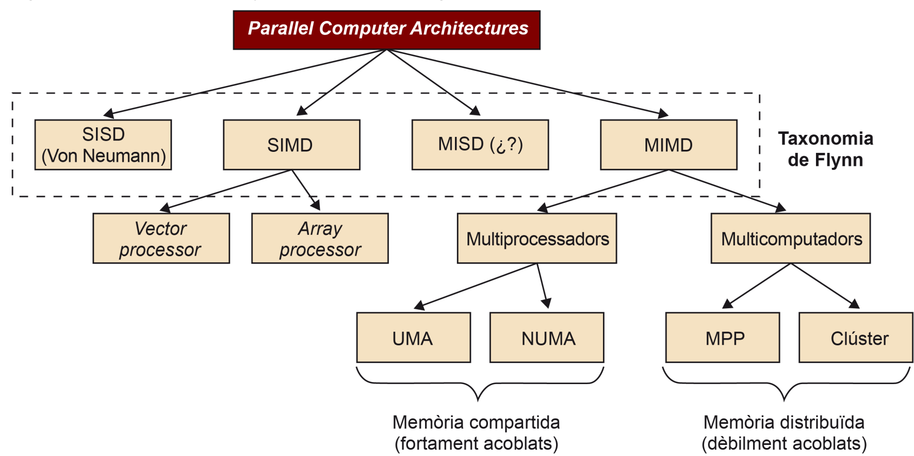
La taxonomia de Flynn classifica els computadors segons com i amb quantes instruccions es processen les dades. Aquesta classificació distingeix si una mateixa dada o dades diferents són processades per una única instrucció o instruccions diferents, i s'obtenen així quatre tipus diferents d'arquitectures: SISD, SIMD, MISD i MIMD.

Taxonomia de Flynn

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

En aquest apartat ens centrarem en les arquitectures *single instruction multiple data* i *multiple instruction multiple data*. Primer descriurem breument les arquitectures *single instruction multiple data*, i després entrarem detalladament en les arquitectures *multiple instruction multiple data*. En la figura 3 es mostra la classificació feta per Flynn, i unes subcategories que classifiquen segons el processament de les dades i la manera de compartir aquestes dades, que detallem a continuació.

Figura 3. Taxonomia de Flynn i altres subcategories



1.1. Arquitectures paral·leles SIMD

Les arquitectures paral·leles SIMD les podem classificar en *array processors* i *vector processors*.

1.1.1. *Array processors*

Els *array processors* bàsicament consisteixen en un gran nombre de processadors idèntics que fan la mateixa seqüència d'instruccions en dades diferents. Cada processador, en paral·lel amb la resta de processadors, fa la mateixa instrucció sobre les dades que li toca processar. El primer *array processor* va ser el computador ILLI-

ILLIAC IV

Va ser creada per la Universitat d'Illinois el 1976.

AC IV de la Universitat d'Illinois, encara que no es va poder muntar tot per motius econòmics.

Noteu que els *array processors* són la llavor de les extensions SIMD dels monoprocessadors.

1.1.2. *Vector processors*

Els *vector processors*, des del punt de vista del programador, són el mateix que els computadors *array processors*. En canvi, el processament de les dades no es fa en paral·lel. Les dades són processades per una única unitat funcional molt segmentada. L'empresa que més *vector processors* ha fet és Cray Research (ara part de SGI). La primera màquina *vector processor* va ser la Cray-1.

Cray-1

Va ser instal·lada en Los Alamos National Laboratory el 1976.

1.2. Multiprocessadors

Els multiprocessadors són computadors que tenen memòria compartida i es poden classificar en UMA (*uniform memory access*), NUMA (*nonuniform memory access*) i COMA (*cache only memory access*).

Els noms que reben les arquitectures multiprocessador UMA i NUMA tenen a veure amb el temps d'accés a la memòria principal, i no es té en compte la diferència de temps entre un encert o un error en memòria cau. En cas contrari, també hauríem de considerar arquitectura NUMA qualsevol sistema amb jerarquia de memòria, incloent-hi els monoprocessadors.

Quant a l'arquitectura COMA, que no va tenir gaire èxit, es basa a tenir la memòria compartida com si fos una gran memòria cau.

A continuació descriurem cadascuna d'aquestes arquitectures.

1.2.1. UMA

En aquest tipus d'arquitectura, com bé diu el nom, tots els accessos a memòria triguen el mateix temps. Segurament podem pensar que és difícil que tinguem el mateix temps d'accés si la memòria, encara que compartida, està dividida en mòduls als quals s'accedeix per mitjà d'una xarxa d'interconnexió basada en *switches*. Això és cert i, per a aconseguir aquesta uniformitat d'accés, s'ha d'augmentar el temps dels accessos més ràpids.

Switch

És un dispositiu que interconnecta dos o més segments de xarxa (*links*), passant dades d'un segment a un altre segons la configuració de connexions i la direcció que hagi de prendre la dada per transferir.

Per què es busca aquesta uniformitat de temps d'accés? Perquè per als programadors és més fàcil deduir quins paràmetres ajudaran a millorar els seus programes si el temps d'accés és igual per a qualsevol accés. En cas contrari, com passa amb les arquitectures

NUMA*, haurien de ser molt més conscients de l'arquitectura que té el computador i de com estan distribuïdes les dades en l'arquitectura en qüestió.

La figura 4 mostra un sistema multiprocessador en què els processadors es connecten a la memòria compartida utilitzant un bus (dada i adreces). Aquest tipus de màquines són fàcils de construir. No obstant això, el fet que tots els processadors accedeixin al mateix bus per a accedir a memòria pot significar un coll d'ampolla. Una alternativa a tenir un únic bus seria tenir diversos busos, de tal manera que es distribuirien els accessos als mòduls de memòria. També es podria usar una xarxa d'interconnexió més complexa, com una xarxa *crossbar*, que permeti evitar conflictes entre els accessos a les memòries que no van al mateix mòdul, tal com es mostra en la figura 5.

*** Els sistemes NUMA faciliten la programació, ja que el temps d'accés sempre és el mateix.**

Connexions mitjançant bus

Les connexions per mitjà d'un bus són fàcils de construir però poden tenir més contenció d'accés a memòria que altres xarxes d'interconnexió.

Figura 4. Multiprocessador de memòria compartida per mitjà d'un bus

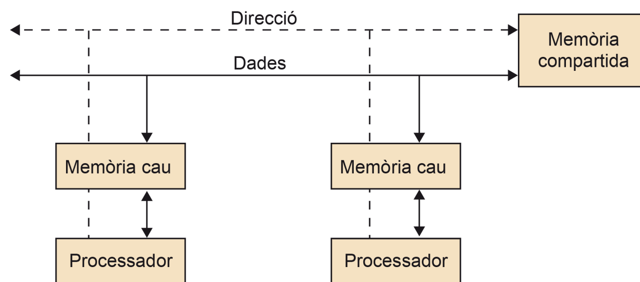
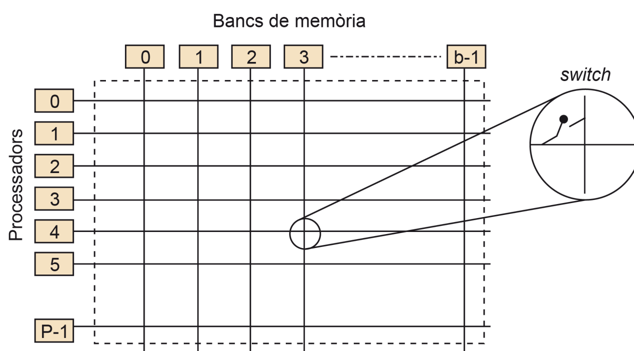


Figura 5. Multiprocessador de memòria compartida per mitjà d'un *crossbar*



D'altra banda, una manera de reduir la contenció dels accessos a memòria* és reduint la quantitat d'aquests accessos per part dels processadors, i es pot aconseguir incorporant jerarquies de memòria en aquests. D'aquesta manera, si els programes exploten la localitat de dades, els accessos a memòria es reduiran.

*** La jerarquia de memòria ajuda a reduir la contenció d'accés a memòria si s'explota la localitat de dades.**

Les màquines UMA es van fer més i més populars des d'aproximadament l'any 2000, any en el qual podríem dir que hi va haver un punt d'inflexió en la tendència dels dissenys de les arquitectures dels monoprocessadors, bàsicament a causa del consum energètic i a la conseqüent baixada de rendiment en les aplicacions. L'aparició dels processadors CMP o *chip multi-processors* va ser el resultat natural d'intentar aprofitar les millores tecnològiques existents, i aprofitar així millor l'àrea del xip, sense augmentar-ne considerablement el consum energètic.

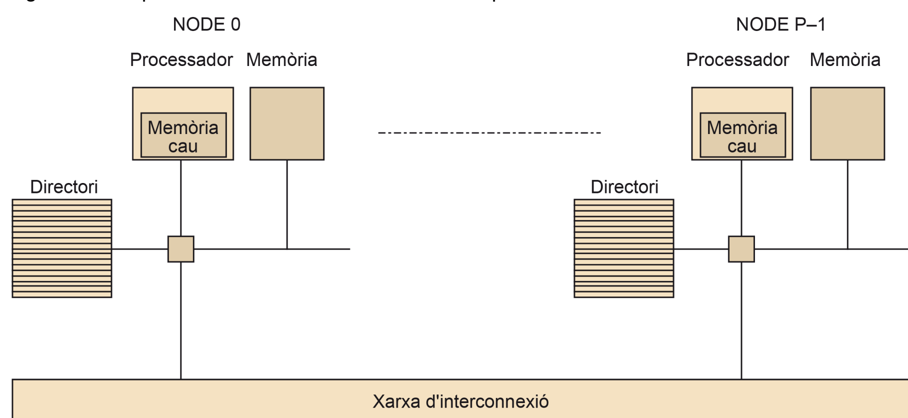
Els processadors CMP són processadors de tipus UMA que incorporen més d'una unitat de processament dins d'un únic xip. Alguns exemples amb unitats de processament homogènies (totes les CPU són iguals) són els Intel Core2 Duo (amb 2 CPU), Intel Nehalemi7 (amb 4 CPU), AMD Istanbul (amb 6 CPU), Sun SPARC64-VIIIfx (amb 8 CPU), IBM Power7 (amb 8 CPU), etc. Però també n'hi ha amb unitats heterogènies, com és el cas de l'Intel Sandy Bridge, que té un processador de caràcter general i una unitat de processament gràfic.

1.2.2. NUMA

En els multiprocessadors NUMA, a diferència dels UMA, els accessos a memòria poden tenir temps diferents. En aquestes màquines la memòria també està compartida, però els mòduls de memòria estan distribuïts entre els diferents processadors amb l'objectiu de reduir la contenció d'accés a memòria.

El mòdul de memòria que està al costat d'un processador en un mateix node rep el nom de memòria local a aquest processador. Així, els accessos d'un processador a la seva memòria local solen ser molt més ràpids que els accessos a la memòria local d'un altre processador (memòria remota). La figura 6 mostra un esquema bàsic de la distribució de memòria en aquest tipus de multiprocessadors, en què s'observa la distribució dels mòduls entre els diferents nodes. Aquests nodes tenen un bus local per a accedir a la memòria local, i es connecten a la xarxa d'interconnexió per a accedir a la memòria situada en altres nodes. També observem un maquinari anomenat *directori* que serveix per a mantenir la coherència de les dades, i que explicarem, més endavant, en el mòdul. La xarxes d'interconnexió típiques d'aquests sistemes són les xarxes de tipus *tree* i les de bus jeràrquic.

Figura 6. Multiprocessador NUMA de memòria compartida



Des del punt de vista del programador, aquest haurà de ser conscient de l'arquitectura NUMA per a poder apropar les dades a la memòria local del processador que hagi d'operar amb aquestes dades. D'aquesta manera es podran reduir els accessos a memòries locals d'altres processadors (memòries remotes) i, com a conseqüència, millorarà el temps d'execució de l'aplicació.

Mapatge de dades

El programador haurà de ser conscient del mapatge de les dades per a reduir el nombre d'accessos a la memòria local d'altres processadors.

D'altra banda, igual que passava amb les arquitectures UMA, també es van incorporar les memòries cau per a reduir la contenció de memòria. Però a més, amb aquestes memòries cau també s'intenta ocultar la diferència de temps d'accés a memòria entre memòria local i remota. Segons tinguin o no memòria cau, els multiprocessadors NUMA es classifiquen en *cache coherence* NUMA (ccNUMA) o *non-coherence* NUMA, respectivament. El terme de coherència o no-coherència prové de mantenir-la o no en les dades duplicades en les memòries cau.

Coherència de les dades

Mantenir la coherència de les dades significa que l'últim valor d'una dada ha de ser vist per tots els processadors.

Un exemple de multiprocessador ccNUMA és l'SGI Altix UV 1000, que està format per 224 processadors Intel Xeon X7542 de 64 bits, a 2,66 GHz de 6 nuclis, amb un total de 1.344 nuclis de càlcul. Altres exemples de màquines NUMA són l'SGI Origin 3000, el Cray T3I i l'AMD Opteron.

1.2.3. COMA

Encara que les arquitectures ccNUMA ajuden a ocultar la diferència de temps entre accessos locals i remots, això dependrà de la localitat de dades que s'exploti i la capacitat de les memòries cau. Així, en el cas que el volum de les dades a què s'accedeix sigui més gran que la capacitat de la memòria cau, o el patró d'accés no ajudi, tornarem a tenir errors de memòria cau i el rendiment de les aplicacions no serà bo.

Les arquitectures COMA (*cache only memory access*) intenten solucionar aquest problema fent que la memorial local a cada processador es converteixi en part d'una memòria cau gran. En aquest tipus d'arquitectures, les pàgines de memòria desapareixen i totes les dades es tracten com a línies de memòria cau. No obstant això, amb aquesta estratègia sorgeixen nous interrogants: com es localitzen les línies de memòria cau? I quan una línia s'ha de reemplaçar, què succeeix si és l'última? Aquestes preguntes no tenen solució fàcil i poden requerir suport maquinari addicional. Normalment s'implementen utilitzant un mecanisme de directori distribuït.

Arquitectures COMA

Les arquitectures *cache only memory access* tracten la memòria principal com una memòria cau gran.

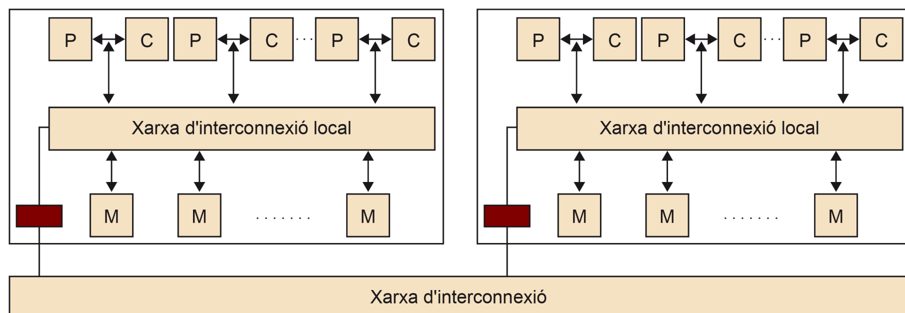
Així, encara que aquest tipus de màquines prometien més bon rendiment que les ccNUMA, se n'han construït poques. Les primeres COMA que es van construir van ser la KSR-1 i la *data diffusion machine*.

1.3. Multicomputadors

La característica més important d'aquests sistemes és que tenen la memòria distribuïda. Aquesta arquitectura també és coneguda com a arquitectura basada en el pas de missatges, ja que els processos han de fer comunicacions (missatges) per mitjà de la xarxa d'interconnexió per a poder compartir dades. Aquestes màquines, per tant, no estan limitades per l'amplada de banda de memòria, sinó més aviat per la de la xarxa d'interconnexió.

La figura 7 mostra l'esquema bàsic d'un multicomputador en què cada node té part de la memòria distribuïda. Els nodes poden contenir des d'un simple monoprocessador a un sistema multiprocessador.

Figura 7. Esquema bàsic d'un multicomputador en què cada node és un multiprocessador



1.3.1. MPP

Els sistemes MPP (*massively parallel processors*) són multicomputadors que tenen les seves CPU connectades amb una xarxa d'interconnexió estàtica d'altes prestacions (baixa latència i amplada de banda elevada) especialment dissenyada per al sistema. Són sistemes grans, en comparació d'un sistema CMP, però que no solen tenir un nombre gaire elevat de CPU a causa del cost econòmic que implicaria mantenir una xarxa d'interconnexió d'altes prestacions. En general, són sistemes difícilment escalables.

Les característiques principals dels sistemes MPP són:

- Utilitzen microprocessadors estàndard, però també poden incorporar maquinari específic o acceleradors (per exemple, ASIC*, FPGA**, GPU).

- La xarxa d'interconnexió que incorporen és de propietat, especialment dissenyada, amb molt baixa latència i una amplada de banda elevada.

- Solen venir amb programari de propietat i biblioteques per a gestionar la comunicació.

- Tenen una capacitat d'emmagatzematge d'entrada/sortida elevada, ja que se solen utilitzar per a treballar amb grans volums de dades que s'han de processar i emmagatzemar.

- Tenen mecanismes de tolerància d'errors de maquinari.

*** Maquinari específic per al desenvolupament d'algun procés concret.**
**** *Field programmable gate array*: maquinari que es pot programar usant llenguatges de descripció maquinari.**

Aquests sistemes MPP normalment s'han utilitzat en càlculs científics, però també s'han usat de manera comercial. La sèrie de *connection machines* és l'exemple típic de màquines massivament paral·leles.

Altres màquines MPP com el Blue Gene (núm. 1 en el Top500 el 2006), implementada sobre un Toroide 3D ($32 \times 32 \times 64$), el Red Strom, basada en AMD Opterons, i Anton, utilitzen alguns ASIC per a accelerar càlculs i comunicacions. En particular, la màquina Anton va ser especialment dissenyada per a resoldre problemes de dinàmica molecular de manera molt ràpida.

En la taula 1 mostrem alguns exemples importants de les primeres xarxes d'interconnexió utilitzades en sistemes MPP i les seves característiques principals.

Taula 1. Xarxes d'interconnexió MPP per a diferents màquines: nombre de nodes que permeten, topologia de xarxa, amplada de banda *bisection bandwidth* que suporten i any d'aparició

Nom	Nre. de modes	Topologia	Amplada de banda (*link*)	*Bisection bandwidth* (MB/s)	Any
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*

Els clústers consisteixen en centenars (o milers) de PC o estacions de treball autònomes (*stand-alone**), possiblement heterogènies, connectades per mitjà d'una xarxa comercial. Aquestes xarxes comercials, que al principi eren menys competitives que les interconnexions de propietat en els sistemes MPP, han anat evolucionant fins a convertir-se en xarxes amb molt bones prestacions. Aquestes xarxes normalment són *TCP/IP packet switched* sobre LAN de coure, WAN, fibra, *wireless* (sense fil), etc. La diferència bàsica pel que fa als sistemes MPP és que aquests tenien nodes d'alt rendiment (*high profile*), interconnectats amb xarxes estàtiques d'alta tecnologia, i especialment dissenyades.

*** *Stand-alone* vol dir que és autònom.**

LAN

local area network (LAN) és una xarxa que connecta processadors en un espai limitat com la casa, l'escola, el laboratori de càlcul, o un edifici.

Els principals avantatges dels clústers són les següents: són molt escalables (en cost per a processador), es poden ampliar gradualment, tenen més disponibilitat per la redundància en nombre de processadors, i surten rendibles a causa de la bona ràtio de cost per benefici, ja que utilitzen *comodities**.

WAN

wide area network (WAN) és una xarxa de comunicacions que cobreix un àrea d'espai molt gran, com per exemple, tota una ciutat, una regió o un país

*** *Comodities* significa que són comercials.**

Les tecnologies més usades en les xarxes d'interconnexió dels clústers són: la WAN, la MAN, la LAN i la SAN. Les xarxes WAN es caracteritzen per poder connectar ordinadors en grans territoris i per tenir unes ràtios de transmissió de dades que van normalment de 1.200 bit/s a 24 Mbit/s, encara que amb certs protocols s'han aconseguit els 155 Mbit/s (*ATM protocol*). La tecnologia LAN està normalment limitada a una certa àrea, com pot ser una casa, un laboratori o potser un edifici, i poden assolir transferències de dades més elevades que les WAN. Les xarxes MAN o *metropolitan area network* estan entre les LAN i les WAN quant a rendiment i àrea que poden abas-

ATM protocol

És una tècnica estàndard de *switching* dissenyada per a unificar telecomunicacions i xarxes de computadors.

tar, i poden seguir els mateixos protocols que les xarxes WAN. Finalment, les xarxes SAN o *storage area network* estan més dedicades a l'emmagatzematge de dades.

Quant als protocols de xarxa que normalment s'utilitzen, trobem els d'Internet, normalment coneguts com a TCP/IP (de *transmission control protocol* [TCP] i *Internet protocol* [IP]) i els ATM i FDDI (no tan utilitzat aquest últim des que van aparèixer les xarxes d'Ethernet ràpides).

Quant a les tecnologies més usades per a interconnexió en els clústers són les següents: Ethernet des de 1974 (100 Gbit/s Ethernet per LAN i 10 Gbit/s per WAN), Myrinet de Myricom des de 1998 (Myri 10 Gbit/s per LAN), Infiniband des de 2005 (Infiniband QDR 12X amb 96 Gbit/s per LAN), QsNet de Quadrics (va aparèixer el 2002, amb 8 Bbit/s per LAN), 802.11n des del 2009 (Wireless Networks amb 600 Mbit/s), etc.

Lectura complementària

Més informació en tecnologies de xarxa:
F. Petrini; O. Lysne; R. Brightwell (2006). "High Performance Interconnects". *IEEE Micro* (núm. 3).

Podem distingir dos tipus de clústers: els centralitzats i els no centralitzats. Reben el nom de centralitzats els que munten el conjunt de PC que el formen en un *rack* o armari compacte. En aquest tipus de clústers els PC són normalment tots del mateix tipus i els únics perifèrics que tenen, per tema d'espai, són les targetes de xarxa i possiblement disc. Els no centralitzats són aquells sistemes els PC dels quals es troben, per exemple, repartits en tot un campus, o bé en un edifici.

Com a exemple més visible de clúster tenim el clúster de computadors que ha construït Google per a poder donar resposta i espai a totes les consultes d'una manera ràpida i eficient. També cal esmentar el Marenostrum, un sistema basat en processadors PowerPC i una xarxa d'interconnexió Myrinet, que va ser el multiprocessador més ràpid d'Europa en el moment de la instal·lació el 2004. El 2006 va doblar la seva capacitat en nombre de processadors en passar a tenir 10.240 processadors IBM Power PC 970MP a 2.3 GHz (*2560 JS21 blades*). Altres exemples que podem esmentar són el clúster Columbia (NASA), que va ser el número 2 del Top500 el 2004, l'IBM Roadrunner LANL (número 1 el 2008 del Top500, i que està basat en 12.960 IBM PowerXCell 8i, més 6.480 AMD Opteron, units amb una Infiniband), i l'Avalon Beowulf Cluster (de 1998) situat en el Lawerence Livermore National Laboratory.

Marenostrum

Marenostrum es troba situat en el BSC-CNS a Barcelona.

1.4. Top500

El projecte Top500 és un rànquing dels 500 supercomputadors més poderosos del món. Aquesta llista està recopilada per Hans Meuer, de la Universitat de Mannheim (Alemanya), Jack Dongarra, de la Universitat de Tennessee (Knoxville), i Erich Strohmaier i Horst Simon, del centre NERSC/Lawrence Berkeley National Laboratory.

El projecte es va iniciar el 1993 i publica una llista actualitzada cada sis mesos. La primera actualització de cada any es fa al juny, coincidint amb la *International Supercomputer Conference*, i la segona actualització és al novembre, en la *IEEE Supercomputer Conference*.

En la taula 2 es mostra el nom del supercomputador més potent, segons els criteris d'ordenació de la llista Top500, des de l'any 1993 fins al 2011.

Taula 2. Llista de màquines més potents segons el criteri utilitzat per la llista Top500.

Nom de la màquina	País	Període
Fujitsu K computer	Japó	Juny 2011 - juny 2012
NUDT Tianhe-1A	Xina	Novembre 2010 - juny 2011
Cray Jaguar	EUA	Novembre 2009 - novembre 2010
IBM Roadrunner	EUA	Juny 2008 - novembre 2009
IBM Blue Gene/L	EUA	Novembre 2004 - juny 2008
NEC Earth Simulator	Japó	Juny 2002 - novembre 2004
IBM ASCI White	EUA	Novembre 2000 - juny 2002
Intel ASCI Red	EUA	Juny 1997 - novembre 2000
Hitachi CP-PACS	Japó	Novembre 1996 - juny 1997
Hitachi SR2201	Japó	Juny 1996 - novembre 1996
Fujitsu Numerical Wind Tunnel	Japó	Novembre 1994 - juny 1996
Intel Paragon XP/S140	EUA	Juny 1994 - novembre 1994
Fujitsu Numerical Wind Tunnel	Japó	Novembre 1993 - juny 1994
TMC CM-5	EUA	Juny 1993 - novembre 1993

2. Multiprocessador

En aquest apartat analitzarem detalladament els sistemes multiprocessador, també coneguts com a sistemes fortament acoblats o de memòria compartida. En el subapartat 2.1. estudiarem com se solen connectar els processadors amb el sistema de memòria i, en particular, veurem les connexions basades en bus, les basades en una xarxa d'interconnexió de tipus *crossbar* i les de tipus *multistage*. Les del primer tipus, basades en bus, són les més fàcils d'implementar però tenen el problema de la contenció en els accessos a memòria. Les del segon tipus (*crossbar*) eviten aquesta contenció, però són molt costoses. Les terceres sorgeixen com una solució intermèdia, menys contenció que les basades en bus, i menys costoses que les *crossbar*.

Una manera de reduir més la contenció d'accessos a memòria és mitjançant la utilització de bancs de memòria als quals es pugui accedir en paral·lel i la incorporació de memòries cau. No obstant això, de resultes d'aquesta divisió en mòduls de memòria i de la integració de les memòries cau apareixen dos problemes: la consistència de memòria, que analitzarem en el subapartat 2.2., i la coherència de memòria cau, que estudiarem en el subapartat 2.3.. La consistència té a veure amb l'ordre relatiu d'escriptures en memòria, i la coherència té a veure amb la visibilitat d'una escriptura en memòria per part de tots els processos. Per a tots dos casos veurem mecanismes de programari i maquinari per a relaxar o solucionar el problema.

2.1. Connexions típiques a memòria

2.1.1. Basades en un bus

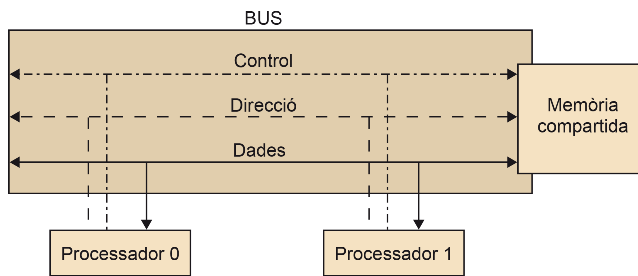
En un sistema multiprocessador basat en bus, tots els processadors comparteixen aquest mitjà. El bus és la manera més fàcil de connectar els processadors amb la memòria i el que millor escala des del punt de vista econòmic amb el nombre de nodes/processadors connectats al bus.

Cost econòmic

El cost econòmic d'una xarxa està normalment associat a la interfície de xarxa i al nombre de dispositius que la formen.

Un altre avantatge d'aquest sistema és que tots els processadors estan connectats amb la memòria de "manera directa", i tots estan a una distància d'un bus. La figura 8 mostra la connexió de dos processadors a la memòria per mitjà d'un bus. En aquestes connexions una part del bus s'utilitza per a comunicar l'adreça de memòria a la qual es vol accedir, una altra per a les dades, i una altra per als senyals de control.

Figura 8. Multiprocessador de memòria compartida per mitjà d'un bus (dades, adreça i control)



El gran desavantatge dels sistemes basats en bus és la poca escalabilitat que tenen des del punt de vista de rendiment. Per exemple, si el nombre de processadors connectats és molt elevat (més de 20), la contenció en l'accés a memòria pot ser significativa. Cada vegada que un processador vol accedir a memòria, ha de demanar permís a l'àrbitre del bus per a saber si aquest està ocupat amb un altre accés. Si el bus està ocupat, el processador que va fer la petició s'haurà d'esperar per a intentar-ho més endavant. Aquest protocol de petició de bus l'han de fer tots els processadors. Per tant, si el nombre de processadors és elevat, la probabilitat que el bus estigui ocupat serà més elevada, amb la conseqüent pèrdua de rendiment a causa de les esperes.

Sistemes basats en bus

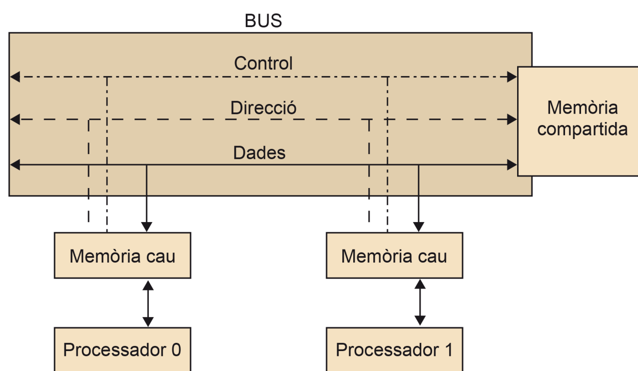
Les xarxes d'interconnexió basades en bus són les més fàcils de muntar però són poc escalables des del punt de vista del rendiment.

Una manera d'alleujar aquesta contenció quan tenim un nombre relativament elevat de processadors és la d'incorporar memòries cau als processadors. Amb això, si s'explota la localitat temporal, estarem reduint els accessos a memòria. Com a contrapartida, tal com veurem més endavant en aquest mòdul, això implica un problema de coherència de les dades en poder tenir diverses còpies de la mateixa dada en diverses memòries cau del multiprocessador. La figura 9 mostra un esquema bàsic d'un sistema multiprocessador basat en bus amb memòries cau.

Les memòries cau

Les memòries cau redueixen la contenció d'accés a memòria si s'explota la localitat de dades, però introdueixen el problema de coherència de memòria cau.

Figura 9. Multiprocessador de memòria compartida per mitjà d'un bus (dades, adreça i control), amb memòria cau



Finalment, una altra possibilitat d'alleujar la contenció en sistemes basats en bus, a part d'incorporar memòries cau, és incorporant memòries privades en cada processador. Aquestes memòries es poden carregar amb codi, dades constants, variables locals, etc., que no afecten la resta de processadors i que alleujarien també la contenció del bus. No obstant això, en aquest cas, normalment és responsabilitat del programador i del processador fer la càrrega d'aquestes dades en aquesta memòria privada, encara que hi ha treballs en què s'implementen *software caches* que intenten llevar aquesta

Lectura complementària

Sobre el tema de la implementació de *software caches* podeu llegir:
Marc Gonzalez; Nikola Vujic; Xavier Martorell; Eduard Ayguade; Alexandre E. Eichenberger; Tong Chen; Zehra Sura; Tao Zhang; Kevin O'Brien; Kathryn O'Brien (2008).“Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture” (pàg. 292-302). Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08).

responsabilitat al programador. Per exemple, les targetes gràfiques del tipus GPG-PU (*general purpose graphic processing unit*) tenen una part de memòria anomenada privada, que està dedicada precisament a aquest tipus de dades. Amb això intenten alleujar l'accés a memòria compartida.

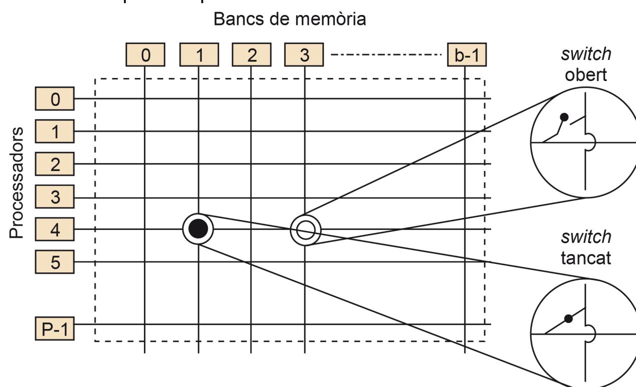
2.1.2. Basades en *crossbar*

Els sistemes basats en bus tenen el problema d'escalabilitat a partir de 16 o 32 processadors. Una manera d'evitar aquest problema de manera senzilla és mitjançant l'ús de *crossbars*. Les xarxes d'interconnexió *crossbar* connecten els processadors del sistema als bancs de memòria que formen la memòria compartida i eviten els conflictes en els accessos. La figura 10 mostra un exemple de connexió de P processadors a b bancs de memòria. Aquesta consisteix en una malla de *switches* que permet la connexió no blocadora de qualsevol processador a qualsevol banc de memòria. Cada *switch* del *crosbar* pot ser electrònicament tancat o obert, i permet o no que la línia vertical es connecti a la línia horitzontal, respectivament. En la figura mostrem tots dos casos, un en el qual el *switch* està tancat i un altre en el qual està obert.

Sistemes basats en *crossbar*

En els sistemes de memòria compartida basats en *crossbar* no hi ha conflictes d'accés a memòria sempre que no s'accedeixi al mateix mòdul de memòria.

Figura 10. Multiprocessador de memòria compartida per mitjà d'un *crossbar*, que ensenya les dues maneres en els quals es pot trobar un *switch*



Què significa que la malla de *switches* permet una connexió no bloquejant? Significa que l'accés d'un processador a un banc de memòria no bloqueja la connexió d'un altre processador a qualsevol altre banc de memòria. És per això mateix pel que normalment el nombre de bancs de memòria és més gran que el nombre de processadors. En un altre cas, hi hauria processadors que no es podrien connectar a un banc de memòria a causa d'un conflicte per a accedir al mateix banc que un altre processador.

El desavantatge d'aquest sistema de connexió amb memòria és que es necessiten entorn de P^2 *switches* per a poder muntar aquest sistema; aquest cost és important i, per tant, poc escalable. En canvi, un sistema basat en un bus és escalable en termes de cost, però no de rendiment. En el subapartat següent mostrem un sistema que té un cost menys elevat que els *crossbar* i amb més bon rendiment que els sistemes basats en bus.

Sistema poc escalable

Les xarxes d'interconnexió de tipus *crossbar* eviten els conflictes d'accés a mòduls de memòria diferents. No obstant això, són poc escalables econòmicament parlant.

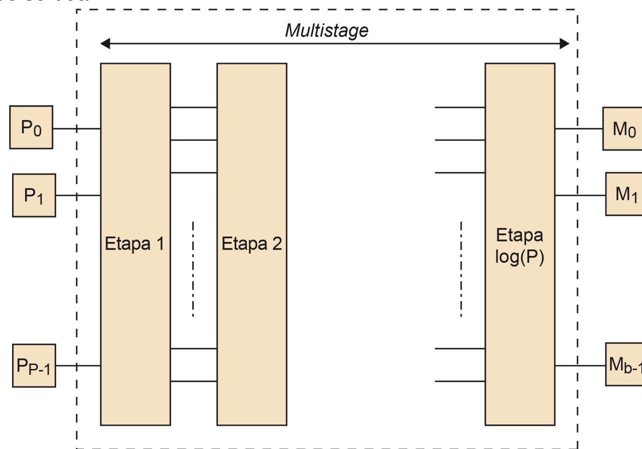
2.1.3. Basades en *multistage*

Els sistemes basats en *multistage* consisteixen en una sèrie d'etapes que estan unes connectades a les altres de tal manera que es connectin els processadors amb els bancs de memòria. La figura 11 mostra un esquema general d'un sistema de connexió basat en *multistage*.

Sistemes basats en *multistage*

La xarxa d'interconnexió *multistage* permet tenir menys conflictes d'accés a memòria que els sistemes basats en bus i són menys costosos que les xarxes de tipus *crossbar*.

Figura 11. Xarxa d'interconnexió *multistage* genèrica de P processadors d'entrada i b bancs de memòria de sortida

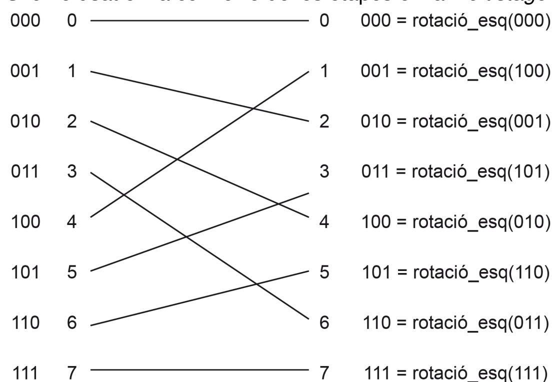


La xarxa *omega* és la *multistage* més comuna. Aquesta té $\log P$ etapes, en què P és el nombre de processadors connectats a la primera etapa, connectats uns amb altres en cadena. El nombre de sortides de cada etapa és també P, i l'última etapa està connectada amb P bancs de memòria. La manera de connectar una entrada i amb una sortida j en una etapa segueix un patró que es coneix com a *perfect shuffle*, tal com indica la fórmula següent:

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

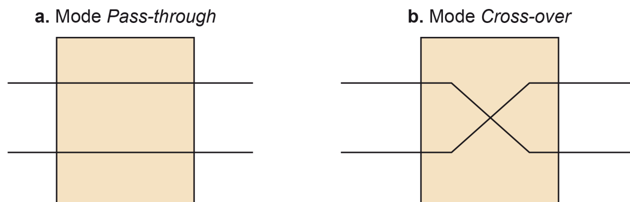
La figura 12 mostra gràficament com es connecten les sortides d'una etapa amb les entrades de l'etapa següent en una xarxa *omega*.

Figura 12. *Perfect Shuffle* usat en la connexió de les etapes en la *multistage omega*



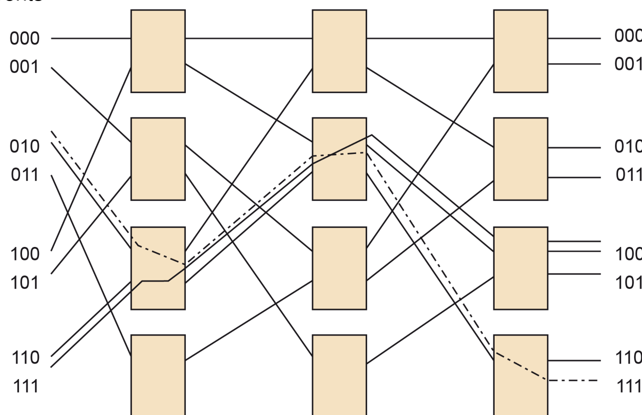
En cada etapa hi ha $P/2$ *switches*. Un *switch* de l'etapa i està configurat de tal manera que per a un enviament del processador P al banc de memòria b, compararà el bit i de la codificació del processador i del banc de memòria al qual està accedint. Si els bits són iguals, el *switch* deixarà passar la dada tal com mostra la figura 13.a. En cas contrari, si són diferents, llavors el *switch* encreuarà les entrades com es mostra en la figura 13.b.

Figura 13. Modes d'ocupació d'un *switch* d'una etapa de la xarxa d'interconnexió *omega*



Aquest tipus de sistema de connexió és menys costós que un sistema basat en *crossbar*, ja que ara només es necessiten $P/2 \times \log p$ *switches*, en comparació dels $P \times P$ *switches* del *crossbar*. No obstant això, aquesta reducció de cost es produeix com a contrapartida al fet que dos accessos des de dos processadors poden necessitar compartir un mateix *switch*. En aquest cas, un dels accessos s'haurà de bloquejar i esperar que l'altre acabi de fer l'accés. El conflicte es pot produir tant en el moment d'accedir a la dada, o quan aquesta està tornant al processador. La figura 14 mostra un exemple en què dos accessos diferents s'han de bloquejar, ja que comparteixen part del camí.

Figura 14. Exemple de conflicte en l'accés de dos processadors diferents a dos bancs de memòria diferents



2.2. Consistència de memòria

2.2.1. Definició

El problema de consistència de memòria apareix quan una memòria està formada per diversos mòduls de memòria connectats per mitjà d'una xarxa d'interconnexió a un conjunt de processadors. En particular, quan diversos processadors accedeixen a una dada per a escriptura se'ns planteja la pregunta següent: en quin ordre han de veure

aquestes escriptures la resta de processadors si aquests fan una lectura de les dades de manera paral·lela?

Hi ha moltes respostes possibles, ja que uns accessos es poden avançar als altres, i el resultat de les lectures és impredictible. Tot això s'agreuja quan tenim memòries cau, ja que llavors és possible que diversos processadors tinguin còpies de lectures prèvies.

Perquè els processadors que accedeixen a unes dades sàpiguen, per endavant, quins valors poden obtenir si hi ha escriptures i lectures alhora, se solen determinar unes regles que determinen el que pot retornar la memòria en aquesta situació. El conjunt d'aquestes normes és el que rep el nom de model de consistència. Per exemple, si una CPU0 fa un accés d'escriptura del valor X en una posició de memòria, la CPU1 fa també un accés d'escriptura del valor Y en la mateixa posició de memòria, i la CPU2 fa un accés de lectura del valor d'aquesta mateixa posició de memòria; què hauria d'obtenir CPU2? Pot ser que obtingui el valor X i no el valor Y, i això és possiblement correcte si el model de consistència, que tots els processadors han de conèixer, permet aquest resultat.

Model de consistència

Un model de consistència és el conjunt de normes que ens indica quins valors haurien de veure els processadors en lectures successives d'una o diverses posicions de memòria.

A continuació detallarem alguns dels models de consistència coneguts: *strict consistency*, *sequential consistency*, *processor consistency*, *weak consistency* i *release consistency*.

2.2.2. Consistència estricta o *strict consistency*

En aquest model de consistència, qualsevol lectura sobre una posició de memòria haurà de tenir com a resultat l'última escriptura feta en aquesta posició de memòria. Així, en l'exemple anterior, la CPU2 hauria d'obtenir el valor Y.

Consistència estricta

La consistència estricta garanteix que els valors que es veuen en memòria estiguin en estricte ordre d'escriptura.

Aquesta consistència es pot aconseguir si hi ha un únic mòdul de memòria que tracti en estricte ordre d'arribada els accessos, i sense cap tipus de memòria cau en el sistema. Això es pot fer però es traduirà que l'accés a memòria serà un coll d'ampolla.

2.2.3. Consistència seqüencial o *sequential consistency*

El model de consistència seqüencial garanteix un mateix ordre de les escriptures per a totes les CPU. Amb aquest model no és possible que dues CPU, que llegeixen una dada d'una posició de memòria diverses vegades al mateix temps que altres CPU estan escrivint sobre la mateixa posició, vegin dues seqüències diferents de valors en aquesta posició de memòria. L'ordre en el qual es veuen les escriptures sota aquest model no és important, ja que que es produeixi abans una escriptura o una altra serà aleatori. L'important és que quan s'estableixi aquest ordre, totes les CPU vegin el mateix ordre.

Consistència seqüencial

La consistència seqüencial garanteix que els valors que es veuen en memòria els vegin en el mateix ordre totes les CPU. No importa si és en un ordre diferent d'aquell en què es van fer les escriptures.

Per exemple, suposem el cas en el qual la CPU0 i la CPU1 fan un accés d'escriptura dels valors X i Y sobre una posició de memòria, separats mínimament en el temps.

Molt poc després les CPU2 i CPU3 fan dos accessos de lectura, cadascun sobre la mateixa posició de memòria en la qual la CPU0 i la CPU1 fan l'escriptura. El resultat de les lectures pot variar segons la visió que tinguin de les escriptures, però el que no pot succeir sota aquest model de consistència és que la CPU2 vegi X en la primera lectura i Y en la segona lectura, i que la CPU3 vegi Y en la primera i X en la segona. Això significaria que totes dues CPU tenen una visió diferent de les escriptures. Noteu que el fet que s'hagi produït l'ordre X, i després Y, en comptes de l'ordre Y i després X, no és important. L'important és que les CPU2 i CPU3 veuen el mateix ordre.

2.2.4. Consistència del processador o *processor consistency*

El model de consistència del processador indica que les escriptures fetes per un processador sobre una posició de memòria seran vistes pels altres processadors en el mateix ordre en el qual es van llançar. No obstant això, a diferència del model de consistència seqüencial, no garanteix que els accessos d'escriptures de diferents processadors els vegin igual els altres processadors.

Consistència del processador

La consistència de processador garanteix que els valors que es veuen en memòria respecten l'ordre de les escriptures d'un processador. No obstant això, dos processadors poden veure diferents ordres d'escriptures.

Per exemple, en el cas que la CPU0 faci les escriptures $W0_0$, $W0_1$ i $W0_2$ sobre una posició de memòria m, la resta de CPU només podran veure aquest ordre d'escriptures sobre la posició de memòria m. Si ara, d'una manera intercalada, la CPU1 fa les escriptures $W1_0$, $W1_1$ i $W1_2$ sobre la mateixa posició, la resta de processadors veuran les escriptures de la CPU1 en l'ordre que es van llançar. No obstant això, dues CPU diferents, CPU2 i CPU3, podrien observar un ordre diferent d'escriptures. Per exemple, la CPU2 podria veure $W1_0$, $W0_0$, $W0_1$, $W1_1$, $W0_2$, $W1_2$, i, en canvi, la CPU3 podria veure $W0_0$, $W1_0$, $W1_1$, $W0_1$, $W1_2$, $W0_2$. Com podem observar, tenen dues visions globals diferents de les escriptures, però l'ordre de les escriptures respecta l'ordre de com es van fer els accessos d'escriptura per part de la CPU0 i la CPU1.

2.2.5. Consistència *weak*

Aquest model és molt més flexible que l'anterior, ja que ni tan sols garanteix l'ordre de llançament dels accessos d'escriptura d'un mateix processador. Per tant, per a l'exemple de model de consistència de processador, podria succeir que la CPU2 veiés les escriptures de la CPU0 en aquest ordre: $W0_2$, $W0_1$ i $W0_0$. Amb això un pot pensar que certament no hi ha cap ordre. No obstant això, el model determina que hi ha operacions de sincronització que garanteixen que totes les escriptures fetes abans de cridar aquesta sincronització es completaran abans de deixar que s'iniciï un altre accés d'escriptura.

Consistència *weak*

La consistència *weak* no garanteix cap ordre entre les escriptures d'un mateix processador, però disposa que hi ha unes operacions de sincronització que garanteixen que totes les operacions d'escriptures s'han d'acabar abans que aquesta operació acabi.

Les operacions de sincronització buiden el *pipeline* d'escriptures, i segueixen un model consistent seqüencial, de tal manera que els processadors veuen un mateix ordre en les operacions de sincronització. Aquestes operacions s'han de fer a escala de programari i no són amb cost zero. Per exemple, en el model de programació de memòria compartida OpenMP hi ha la directiva `#pragma omp flush`, que fa aquesta operació de sincronització.

2.2.6. Consistència *release*

El model de consistència *release* intenta reduir el cost d'esperar totes les escriptures prèvies a una operació de sincronització. Aquest model es basa en la idea de les seccions crítiques. La filosofia és que un procés que surt d'una secció crítica no necessita esperar-se que totes les operacions d'escriptura prèvies s'hagin acabat. El que sí que és necessari és que si un altre procés (o aquest mateix) ha d'entrar en la secció crítica, s'haurà d'esperar que les operacions prèvies a la sortida de la secció crítica sí que hagin acabat. D'aquesta manera s'evita haver de parar els accessos fins que realment sigui necessari.

Consistència *release*

La consistència *release* garanteix que si s'usen adequadament les operacions de sincronització, les escriptures que s'hagin fet en una secció crítica de dades compartides ja s'hauran acabat abans d'entrar una altra vegada.

El mecanisme de funcionament és el següent:

- Una CPU que vol accedir a una dada compartida ha de fer una adquisició del bloqueig per a poder accedir a les dades compartides (entrada en la secció crítica).
- Aquesta CPU pot operar sobre les dades compartides (dins de la secció crítica).
- Quan la CPU acaba, aquesta allibera la secció crítica utilitzant una operació *release*. Aquesta operació no s'espera que totes les escriptures anteriors acabin, ni tampoc evita que en comencin d'altres. Però l'operació *release* no es considerarà acabada fins que totes les escriptures prèvies hagin finalitzat.
- Si una altra CPU (o la mateixa) intenta entrar una altra vegada en una secció crítica, llavors, en el moment de voler adquirir el bloqueig per a l'accés compartit, si totes les operacions *release* ja han acabat, podrà continuar. Si no, s'haurà d'esperar que totes les operacions de tipus *release* s'hagin efectuat.

Amb aquest mecanisme, és possible que no hàgim de parar cap escriptura en absolut, i reduïm així el cost que podia significar haver de mantenir el model de consistència *weak*. En aquest model només serà necessari esperar-se a les operacions si en el moment d'entrar en una secció crítica hi ha alguna operació *release* que encara no hagi acabat.

2.2.7. Comparació dels models de consistència

En la taula 3 es mostren els ordres entre els diferents accessos a memòria i de sincronització, per als models de consistència *sequential consistency*, *processor consistency*, *weak consistency* i *release ordering*, des del punt de vista d'un únic processador. Així, els models més relaxats basen la seva consistència únicament en els *fences* creats per operacions de sincronització (S en la taula), en contraposició dels models més estrictes, en què els *fences* són implícits en les operacions de lectura i escriptura. És per això que els models més relaxats (*weak consistency* i *release ordering*) no tenen cap ordre definit entre les operacions de lectura i escriptura.

Fence

Un *fence* és una barrera de sincronització de memòria que força que totes les operacions de memòria prèvies s'hagin acabat.

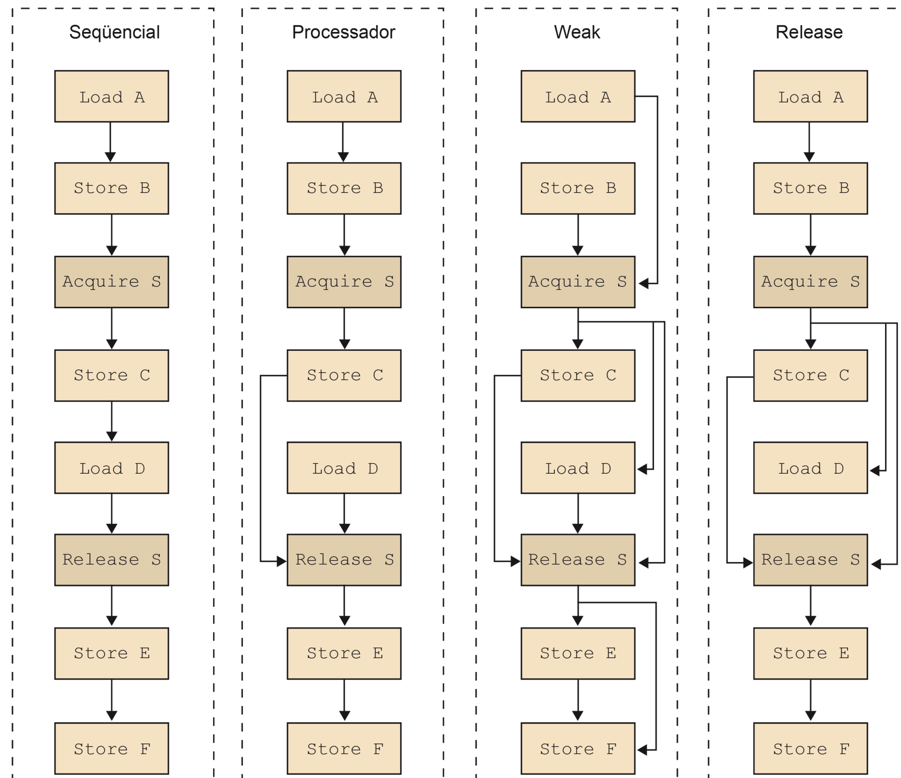
Les operacions de sincronització S_A i S_R són les operacions de sincronització `acquire` (adquirir bloqueig) i `release` (alliberar bloqueig), que són necessàries per al model de consistència *release consistency*. En equivalència, l'operació de sincronització S es transformaria en S_A i S_R. Així, per exemple, el cas $S \rightarrow R$ es transformaria en $S_A \rightarrow R$ i $S_R \rightarrow R$, i un $S \rightarrow S$ es transformaria en: $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$.

Taula 3. Ordre d'execució en els models de consistència *sequential*, *processor*, *weak* i *release* dels accessos de lectura i escriptura, i les sincronitzacions.

Model	Ordre en lectures i escriptures	Ordre en operacions de sincronització
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 mostra gràficament els ordres imposats segons el model de consistència usat. En aquesta figura es mostren els ordres mínims entre els diferents accessos (de sincronització: `acquire`, `release` o d'escriptura i lectura: `load, store`), per la qual cosa aquells que vinguin de manera transitiva no es mostraran, com per exemple, l'escriptura en C, abans del *release* de S en el model de consistència *sequential*. Com es pot observar, a mesura que el model és més relaxat, el nombre d'ordres necessaris es redueix.

Figura 15. Ordre que han de complir les operacions d'accés a memòria i de sincronització per als diferents models de consistència, per a una mateixa seqüència d'operacions



2.3. Coherència de memòria cau

La diferència bàsica entre consistència de memòria i coherència de memòria cau és que la primera determina quan un valor escrit pot ser vist per una lectura, i la segona indica quins valors haurien de poder ser retornats per una lectura (sense indicar quan).

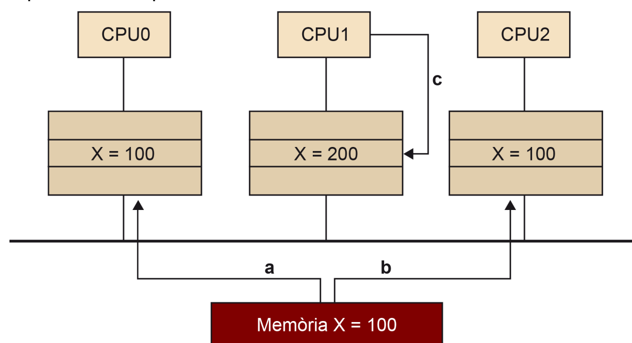
2.3.1. Definició

El problema de la coherència de memòria cau apareix en el moment en què introduïm la memòria cau en la jerarquia de memòria dels processadors que formen part d'un multiprocessador.

L'objectiu principal d'introduir aquestes memòries cau és explotar millor la localitat espacial i temporal de les dades, i així, reduir la contenció de memòria. No obstant això, com a conseqüència de la incorporació de les memòries cau en un sistema multiprocessador, una mateixa dada pot estar replicada en diverses memòries cau alhora. Això no és un problema si no hi ha cap actualització de la dada. En canvi, si es produeix una escriptura en la dada per part d'un processador, la resta de còpies del valor, en la resta de memòries cau, quedaran desactualitzades. En aquest cas hi ha un problema de coherència de la memòria cau.

En la figura 16 veiem un exemple en el qual un *thread* executant-se en la CPU0 fa una lectura de la variable X (figura 16.a) i el deixa en la seva memòria cau. Després, un altre *thread* des de la CPU2 fa una altra lectura (figura 16.b), i finalment un *thread* de la CPU1 fa una escriptura (figura 16.c). En aquest moment, tant la CPU0 com la CPU2 tenen la dada amb un valor antic i diferent del de la CPU1. En aquest cas es diu que aquestes memòries cau no tenen coherència.

Figura 16. Exemple d'un multiprocessador sense coherència de memòria cau



A continuació veurem una sèrie de protocols per a garantir la coherència de memòria cau.

2.3.2. Protocols d'escriptura

Per a mantenir la coherència de memòria cau en els sistemes multiprocessador es poden seguir dues polítiques en l'escriptura d'una dada*:

*** De vegades parlem d'actualització de la dada, o coherència d'aquesta dada, però en realitat hauríem de parlar de bloc de memòria.**

1) *Write-updated* o *write broadcast*. Mitjançant aquest protocol es mantenen actualitzades les còpies de la dada (bloc de memòria) en les memòries cau de la resta de processadors.

2) *Write-invalidate*. Mitjançant aquest protocol es pretén assegurar que el que escriu té l'exclusivitat de la dada (bloc de memòria), i fa que la resta de processadors invalidin la seva còpia i no puguin tenir còpies desactualitzades. La dada (bloc de memòria) és actualitzada en memòria quan és *flushed*/reemplaçada de la memòria cau o la demana un altre processador. Amb aquest protocol, la dada (línia de memòria cau amb el bloc de memòria on hi ha la dada) té uns bits d'estat per a indicar exclusivitat, propietat i si ha estat modificada (*dirty*).

La segona política és la que se sol usar més, ja que normalment té millor rendiment que la primera. A continuació detallem algunes de les característiques per a entendre millor el rendiment que es pot aconseguir amb cadascuna:

1) En el cas de tenir el protocol *write-update* i haver de fer diverses escriptures sobre una mateixa dada, sense cap lectura intercalada, el sistema haurà de fer totes les actualitzacions de memòria i memòries cau de la resta de processadors. No obstant això, en el cas del protocol *write-invalidate*, només cal fer una invalidació, ja que una vegada que queda una única còpia en el sistema, no hi ha necessitat d'enviar més invalidacions.

2) En el cas d'haver de fer escriptures sobre dades diferents que van al mateix bloc de memòria, si estem en el protocol *write-update*, haurem d'actualitzar la dada en aquest bloc de memòria tantes vegades com sigui necessari. No obstant això, mantenir totes les dades actualitzades pot significar un elevat nombre d'accessos a memòria. En canvi, si estem en un protocol de *write-invalidate* s'invalidaran les línies de memòries cau que contenen còpia de la dada (bloc de memòria) la primera vegada que s'escrigui en una dada dins d'aquesta línia, i ja no s'hi farà res més a sobre. En contrapartida, es podrien estar provocant errors de memòria cau en les lectures, per part d'altres processadors, d'altres dades en la mateixa línia de memòria cau de la dada invalidada.

Write buffers

En el protocol *write-update* es poden agrupar les escriptures a un mateix bloc de memòria en els *write buffers* per a reduir la contenció d'accés a memòria.

3) En el cas d'una lectura d'una dada, segurament la lectura en un sistema amb protocol *write-update* és més ràpida que en un sistema *write-invalidate*. Això és així bàsicament perquè si el processador contenia una còpia de la dada, aquesta s'haurà mantingut actualitzada amb el protocol *write-update*. En canvi, en el protocol *write-invalidate*, la dada haurà estat invalidada. No obstant això, si la programació és adequada, aquestes invalidacions per dades que comparteixen un bloc de memòria es poden reduir o evitar.

Així, el protocol *write-update* necessita més amplada de banda d'accés a bus i a memòria, i és la raó per la qual el protocol que normalment és més usat és el de *write-invalidate*.

2.3.3. Mecanisme de maquinari

La manera de mantenir la coherència de memòria cau en maquinari és mitjançant la utilització de bits d'estat per a qualsevol bloc de memòria que es comparteix. Hi ha dos mecanismes maquinari que mantenen actualitzat l'estat de les dades compartides:

- Sistema basat en *Snoopy*: els bits d'estat del bloc de memòria estan replicats en les memòries cau del sistema que tenen una còpia de la dada (bits per cada línia de memòria cau), a diferència del basat en directoris, que està centralitzat. Són sistemes en els quals els processadors solen estar connectats a la memòria via bus, i a més, cada controlador de memòria cau té un maquinari dedicat que pot llegir/sondejar (*Snoopy*) què passa pel bus. Així, tots els controladors de memòria miren si les peticions d'accessos que es fan sobre el bus afecten alguna de les seves còpies o no.

- Sistema basat en directoris: els bits d'estat de compartició d'un bloc de memòria estan únicament en un lloc anomenat *directori*. Quan es fan lectures i escriptures sobre una dada del bloc de memòria, el seu estat, centralitzat en el directori, s'haurà d'actualitzar de manera adequada.

Diferències dels mecanismes

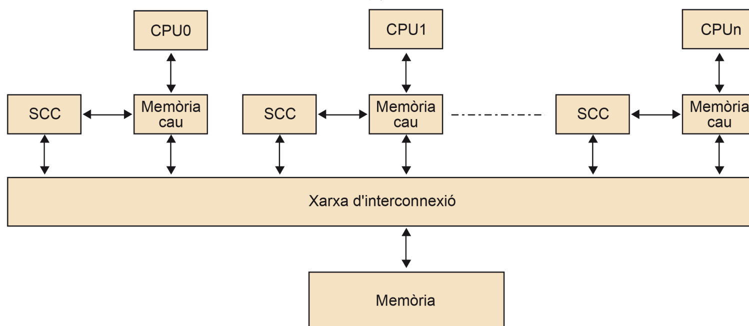
Els sistemes basats en *Snoopy* tenen bits d'estat per a cada còpia del bloc de memòria. En canvi, els sistemes basats en directoris només tenen uns bits d'estat per cada bloc de memòria.

Sistema basat en *Snoopy*

Els sistemes amb memòries cau amb *Snoopy* estan normalment associades a sistemes multiprocessador basats en xarxes d'interconnexió com ara un bus o un *ring*.

La figura 17 mostra un esquema bàsic d'un sistema multiprocessador amb una sèrie de processadors, una memòria cau associada per a processador, i un maquinari dedicat de *Snoopy cache coherence* (SCC). Tots estan connectats a la xarxa d'interconnexió (normalment un bus) que connecta la memòria compartida amb els processadors.

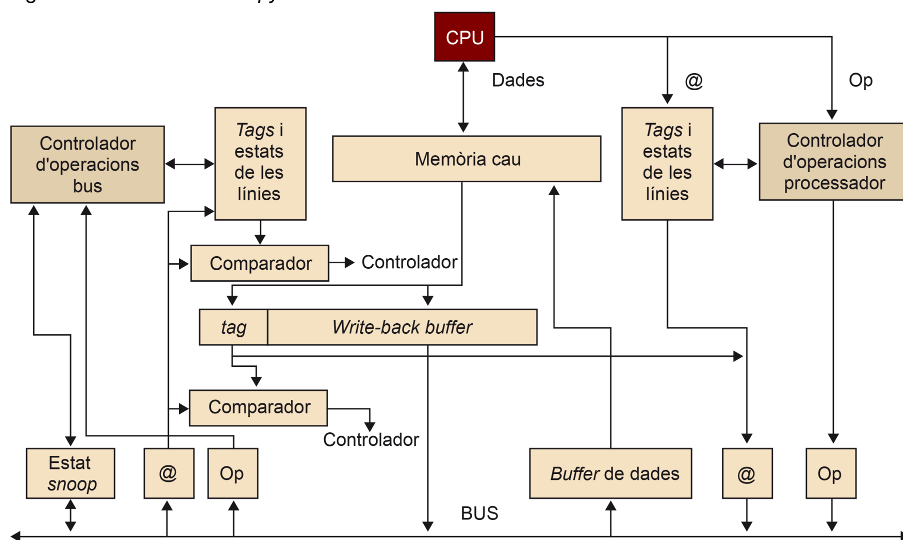
Figura 17. Esquema bàsic d'un sistema multiprocessador amb memòria compartida basat en un bus d'interconnexió i *Snoopy* cache *coherence*



Tal com es connecten els SCC al bus, totes les transaccions en el bus són visibles en tots els processadors. D'aquesta manera, monitorant aquestes transaccions, es poden prendre les accions necessàries de canvi d'estat de les línies de la memòria cau, segons els estats que determini el protocol de coherència que s'assumeixi.

La figura 18 mostra una mica més detalladament el que pot significar un SCC. Si ens fixem en la part dreta de l'esquema, veurem un maquinari dedicat (controlador del processador) que rep instruccions del processador i que està connectat a la informació dels *tags* i de l'estat de les línies de memòria cau. Així, depenent de les ordres o operacions que s'efectuïn en el processador, el controlador del processador, mirant l'estat de les línies de memòria cau i les seves *tags*, decidirà enviar operacions al bus o canviar l'estat d'alguna línia de memòria cau.

Figura 18. Detall de l'*Snoopy cache coherence hardware*



D'altra banda, si ens fixem en el costat esquerre de la figura, veurem que hi ha un maquinari dedicat a sondejar/rebre les ordres o operacions que vénen del bus (Op connectat al bus) o interconnexió amb memòria, que es connecta amb el controlador del bus per a enviar-li la informació. L'adreça que ve del bus es llegeix i es passa als dos comparadors existents: un per a comparar amb els *tags* de la memòria cau i un altre per a comparar-ho amb els *tags* del *write-back buffer*. El *write-back buffer* és un *buffer* per a agrupar escriptures a una mateixa línia de memòria cau. En cas de tenir coincidència en alguna línia o en el *write-back buffer*, s'haurà d'actuar en conseqüència segons el protocol de coherència seguit. També observem que hi ha una connexió anomenada estat *Snoopy* que serveix al processador per a informar al bus que té una còpia de la dada del com està fent una lectura un altre processador, o viceversa, perquè informin al processador que no tindrà exclusivitat en la dada que està llegint de memòria.

Els sistemes basats en bus amb *Snoopy* són fàcils de construir. No obstant això, el rendiment d'aquests només és bo si el nombre de processadors no creix significativament (més de 20), i si els accessos a la memòria es mantenen locals en cada processador, és a dir, s'accedeix a la memòria cau de cada processador. El problema sorgeix quan

es comencen a fer moltes escriptures o tenim molts processadors connectats. Això pot significar que tots els processadors comencin a compartir dades, amb la qual cosa tindríem un trànsit d'operacions a través del bus significatiu, ja que els missatges s'envien a tots els processadors en forma de *broadcast*.

A continuació veurem un sistema basat en directoris, on es desa la informació de l'estat de les línies d'una manera centralitzada (o no replicada). Aquests directoris també desen els processadors que comparteixen un bloc de memòria. En aquests sistemes s'eviten els enviaments en forma de *broadcast* i només s'envien les actualitzacions a aquells processadors que mantenen còpia. Aquests sistemes són més escalables que els sistemes basats en bus, però la construcció és més complexa.

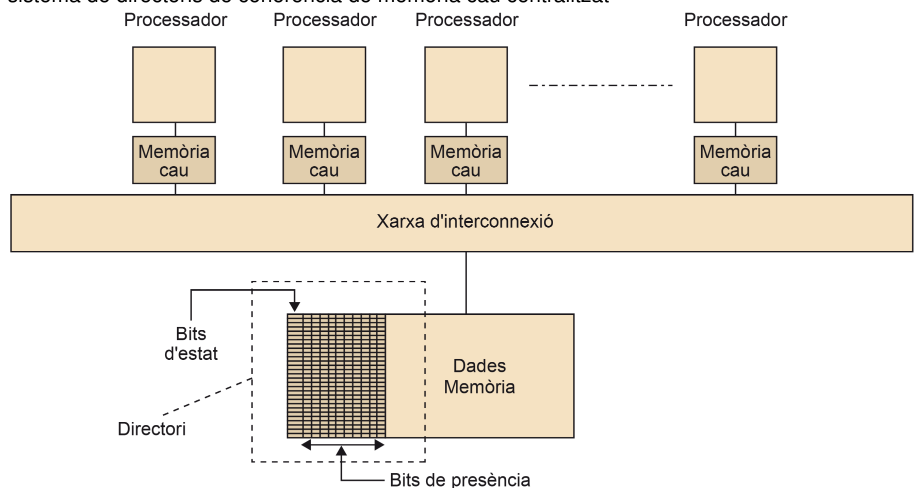
Sistema basat en directoris

La figura 19 mostra un sistema multiprocessador format per diversos processadors que, a través de la xarxa d'interconnexió, es comuniquen amb la memòria compartida global. Aquesta memòria compartida té suport maquinari per a desar la informació dels processadors que disposen d'una còpia d'aquest bloc de memòria en les seves memòries cau. Aquesta informació és clau perquè aquest sistema maquinari sigui més escalable que el de *Snoopy*. Amb aquesta informació, ara es podran enviar els missatges d'actualització o invalidació només a aquells processadors que disposin d'una còpia de la dada.

Sistema basat en directoris

En un sistema basat en directoris es desa informació dels processadors que tenen còpia d'un bloc de memòria per a reduir la comunicació a través de la xarxa d'interconnexió.

Figura 19. Sistema multiprocessador format per una memòria principal compartida amb el sistema de directoris de coherència de memòria cau centralitzat



Hi ha diverses possibilitats de desar aquesta informació, encara que en la figura en mostrem només una:

- Mantenir un únic identificador del processador que té el valor d'aquest bloc de memòria. D'aquesta manera només es podria tenir una còpia de la dada. Això no permet lectures concurrents sense provocar accessos en la memòria principal per a actualitzar quin processador té la còpia a cada moment. L'avantatge és que és

escalable des del moment en què no es desa més que un enter independentment del nombre de processadors.

- Mantenir la informació de k processadors. Això comporta més esforç, ja que cal desar més enters, i depenent del nombre de blocs de memòria, pot ser relativament un cost significatiu (k enters per bloc de memòria). En canvi, a diferència de l'opció anterior, ara sí que es permetrien lectures de fins a k processadors sense cap problema.

- Mantenir P* bits, un bit per processador, per a indicar la presència o no de la dada en la memòria cau d'un processador. D'aquesta manera tots podrien tenir còpies, i segons el nombre de processadors, seria molt més rendible que tenir k enters. L'únic inconvenient és quan el nombre de processadors augmenta considerablement. En aquest cas es podria assolir un nombre de bits relativament elevat en comparació amb la mida d'un bloc de memòria. Aquesta és l'opció que mostrem en la figura. Una possible solució per a reduir aquest cost seria augmentar la mida dels blocs de memòria cau, de tal manera que reduiríem el sobrecost de mantenir P bits per bloc de memòria. Tanmateix, això tindria com a efecte col·lateral que el nombre de dades que comparteixen un mateix bloc de memòria serà més gran i hi haurà més probabilitats que hi hagi invalidacions per actualitzacions de dades diferents, que cauen en el mateix bloc de memòria.

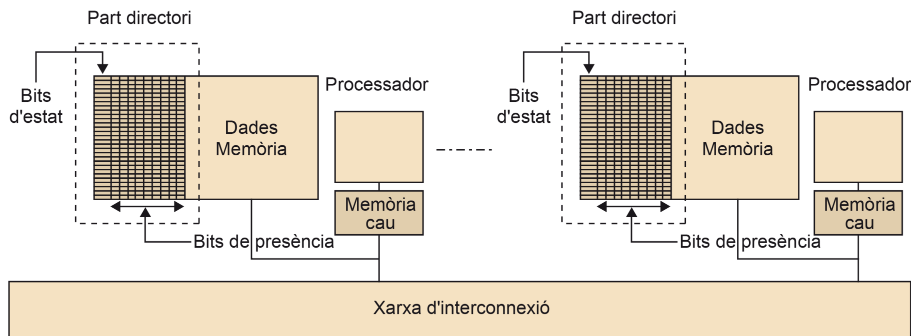
*** En aquest mòdul ens centrarem en sistemes de directoris amb P bits de presència.**

- Mantenir una llista encadenada dels processadors que tenen còpia, utilitzant memòria per a desar aquesta informació. El maquinari dedicat només requeriria ser un registre per a desar la capçalera d'aquesta llista encadenada.

En aquest mòdul ens centrarem en el cas de directoris on hi pot haver tantes còpies com processadors, utilitzant un bit de presència per a cada processador. Aquesta informació es desa juntament amb els bits d'estat del bloc de memòria en la memòria. El valor d'aquests bits d'estat depèn del protocol de coherència que es prengui. En el subapartat 2.3.4. en veurem dos: MSI i MESI, els noms dels quals provenen dels possibles estats de compartició que pot tenir un bloc de memòria. Per a aquests protocols, els possibles estats d'un bloc de memòria són els de *modified*, *exclusive* per al MESI, *shared* i *invalid*.

En un sistema amb un protocol basat en directori, cada vegada que s'accedeix a una dada s'han de consultar i possiblement actualitzar els bits de presència i d'estat en el directori de memòria. Per tant, si la memòria està formada per un sol mòdul, tant els accessos a memòria com l'actualització del directori es poden convertir en colls d'ampolla. Una possible solució és distribuir la memòria físicament en mòduls entre els nodes del sistema multiprocessador, repartint de la mateixa manera el directori entre els diferents mòduls, per a augmentar així el paral·lelisme en la gestió d'accessos a la memòria principal i les transaccions a través del bus. La figura 20 mostra un esquema d'un sistema multiprocessador amb el directori distribuït, típic d'un sistema multiprocessador NUMA.

Figura 20. Sistema multiprocessador format per una memòria principal compartida, físicament distribuïda, amb el sistema de directoris de coherència de memòria cau distribuït



Els sistemes basats en directoris, juntament amb els protocols de coherència MSI i MESI, tenen com a objectiu evitar que els processadors hagin de generar transaccions sobre la xarxa d'interconnexió tant per a lectures com per a escriptures sobre un bloc de memòria, si aquest ja es troba en la memòria cau o bé està actualitzat en la memòria cau del processador. Aquest objectiu és el mateix que es té amb els sistemes basats en *Snoopy*, que també eviten transaccions innecessàries. La diferència bàsica pel que fa al sistema basat en *Snoopy* és que ara, si hi ha diversos processadors que actualitzen una dada, les transaccions en el bus només s'envien a aquells que participen de la compartició d'aquest bloc de memòria.

En un sistema multiprocessador amb el sistema de directoris distribuït hi pot haver involucrats tres tipus de nodes en les transaccions a través del bus:

1) El *local node*, que és el processador/node que genera la petició.

2) El *home node*, que és el node on hi ha el bloc de memòria segons la seva adreça de memòria, i l'assignació que hagi fet el sistema operatiu per mitjà de la traducció de memòria virtual a física (`MMU`).

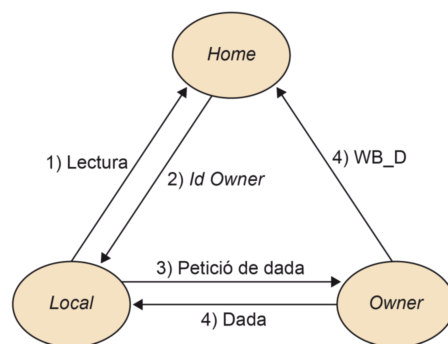
3) El *remote node*, que és el node que té una còpia del bloc de memòria, ja sigui en exclusivitat o de manera compartida.

Entre aquests tres tipus de processadors ens hem d'adonar de la importància del *home node* d'una dada. Aquest processador mantindrà l'estat del bloc de memòria que conté la dada i la informació dels processadors que tenen una còpia d'aquesta. La manera d'assignar el *home node* depèn de la política de traducció d'adreces virtuals a físiques per part del sistema operatiu. Normalment, la política consisteix a intentar apropar les pàgines de memòria al processador que les ha demanades primer. Per tant, si el programador no és conscient d'aquest detall podríem fer que els nostres programes no obtinguessin el rendiment esperat. Ho entendrem quan vegem alguns casos d'exemples en què es fan lectures i escriptures sobre una dada que ja té un *home node* assignat.

El primer cas que tractarem és quan el *local node* fa una lectura d'una dada amb una determinada adreça de memòria. El sistema operatiu, per mitjà de la traducció de l'MMU, podrà determinar quin node té aquesta dada en la seva memòria local, el *home node*, per a poder saber l'estat d'aquest bloc de memòria. El *home node* respondrà a la petició de dues maneres possibles, segons hi hagi o no un processador amb el bloc de memòria actualitzat:

1) Per al cas en què hi hagi un processador amb una còpia en exclusivitat modificada: la figura 21 mostra els tres nodes implicats. Les arestes de la figura estan numerades segons la seqüència d'operacions, i indiquen l'operació o dada que s'envia d'un node a un altre. Així, el *home node* indica al *local node* quin node és el *remote* o *owner node*, després de rebre una petició de lectura d'una dada. El *local node*, amb aquesta informació, demana al *owner node* que li subministri el bloc de memòria en qüestió. Això provoca que l'*owner node* li retorni la dada i que s'actualitzi la memòria principal *writeback-data* (WB_D). Encara que no es mostra en la figura, els bits de presència s'han d'actualitzar adequadament en el *home node*, igual que els bits d'estat de la memòria cau.

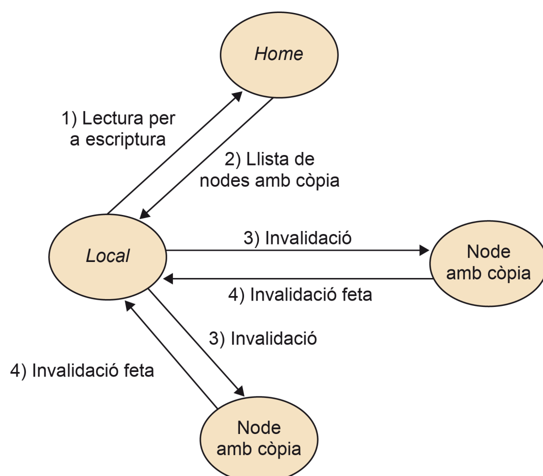
Figura 21. Accés de lectura a una dada d'un bloc de memòria que està modificat



2) Per al cas en què no hi hagi cap processador amb una còpia en exclusivitat modificada: el *home node* subministra el bloc de memòria al *local node* (encara que possiblement també la hi podria donar un altre processador amb una còpia segons el protocol de coherència usat) i, en funció de si el *local node* és l'únic o no amb aquest bloc de memòria, assignarà l'estat *exclusive* (en un protocol de coherència MESI) o *shared* (en un protocol de coherència MSI), respectivament, als bits del bloc de memòria.

En el cas que el *local node* vulgui fer un accés de lectura amb intenció d'escriptura, tal com mostra la figura 22, el *home node* subministrarà la llista de possibles nodes que tinguin una còpia d'aquest bloc de memòria; dos nodes en el cas de la figura. El *local node*, amb aquesta informació, farà una petició a aquests nodes perquè invalidin les còpies del bloc de memòria. El *home node* actualitzarà els bits de presència per a indicar que només el *local node* té una còpia del bloc de memòria i que aquest està modificat.

Figura 22. Accés d'escriptura a una dada d'un bloc de memòria que està compartit per dos processadors més



Amb aquests dos casos observem que el *home node* d'una dada ha de mantenir, en tot moment, l'estat i els bits de presència del bloc de memòria que conté la dada en qüestió, tant per a lectures com per a lectures amb intenció d'escriptura. Això significa que si el processador que fa el primer accés no coincideix amb el processador que fa més actualitzacions d'aquest bloc de memòria, les peticions hauran de passar pel *home node* cada vegada, i deterioraran el rendiment de l'aplicació innecessàriament. Això sol passar, en particular, si la política de traducció d'adreces de virtual a física fa aquesta assignació inicial del *home node** en funció de quin node toca primer el bloc de memòria.

* El primer accés a una dada de memòria determina quin node és el *home node.*

Finalment, cal comentar que hi ha sistemes basats en directoris que l'incorporen en la memòria cau més externa compartida, com pot ser el cas de l'Intel i7 i la sèrie 7000 de Xeon.

2.3.4. Protocols de coherència

En aquesta secció analitzarem detalladament dos protocols de coherència (MSI i MESI) que es poden combinar amb la política en escriptura *write-invalidate* i els mecanismes maquinari que hem vist. Aquests protocols determinen els estats de compartició de memòria en els quals es pot trobar un bloc de memòria, i com es fa la transició d'un a l'altre segons els accessos que es produeixin en el sistema multiprocessador. En particular, descriurem els dos protocols per a un sistema basat en *Snoopy*.

MSI

El protocol MSI de coherència preveu els estats següents:

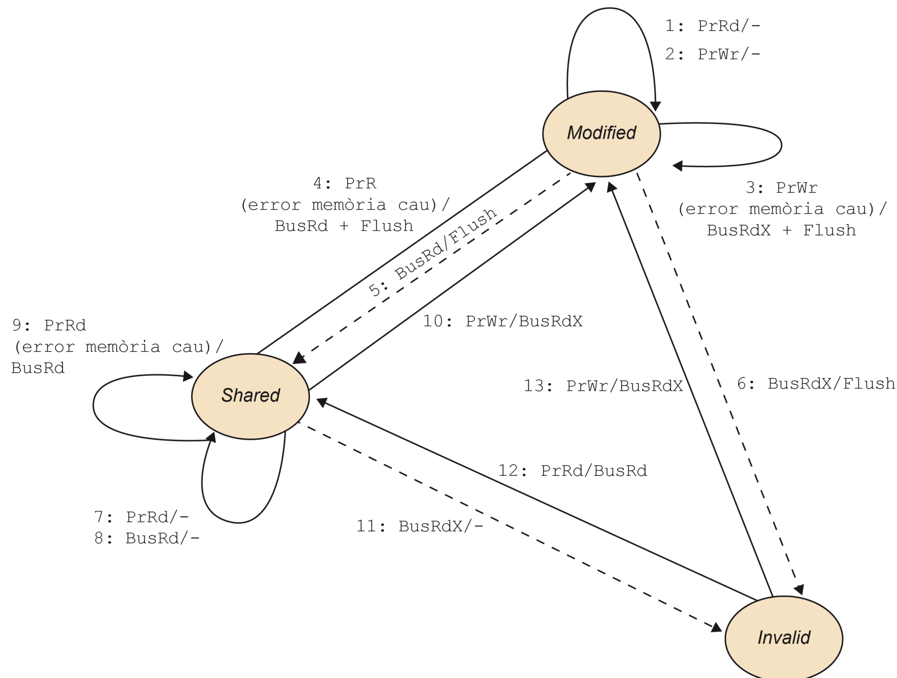
1) *Dirty* o *modified* (**M**): l'estat en el qual un bloc de memòria (línia de memòria cau) està en la memòria cau quan s'ha modificat o s'ha llegit per a modificar.

2) *Shared* (**S**): l'estat en el qual es troba un bloc de memòria quan s'ha llegit i no hi ha intenció de modificar-lo.

3) *Invalid* (**I**): l'estat en el qual es troba un bloc de memòria (línia de memòria cau) quan s'ha invalidat o bé no existeix en la memòria cau.

Com aplicaríem aquests estats a una màquina amb política en escriptura *write-invalidate* i un sistema maquinari basat en bus (*Snoopy*)? El graf d'estats que es mostra en la figura 23 determina els estats en els quals es pot trobar una línia de memòria cau (nodes del graf) i les accions del bus o del processador que provoquen els canvis d'un estat a l'altre (arestes del graf). Les arestes del graf estan etiquetades amb el format següent: <origen de l'acció / transacció en el bus feta com a part de l'acció>. Les operacions o accions que pot fer el processador són : `PrRd` (el processador fa una lectura `Rd`) i `PrWr` (el processador fa una escriptura `Wr`). Les accions que es poden fer a través del bus són: `BusRd` (es fa un petició de lectura a memòria), `BusRdX` (es fa una petició de lectura amb intenció d'escriptura), i `flush`, es deixa en el bus el bloc de memòria que conté la dada (la línia de memòria cau), perquè la memòria i, si es dóna el cas, la memòria cau del processador que la va sol·licitar, puguin actualitzar la línia. Aquestes accions en el bus es veuen en la resta de processadors gràcies als mecanismes maquinari per a mantenir la coherència. Per exemple, en un sistema basat en *Snoopy*, l'SCC del sistema podrà veure què passa pel bus i actuar en conseqüència.

Figura 23. Graf d'estats del protocol de coherència `MSI`



Des del punt de vista de la memòria cau d'un processador del sistema, els estats que poden tenir cadascuna de les línies de la memòria cau són els següents:

1) *Modified*: Si una línia de memòria cau està en l'estat de modificada significa que el bloc de memòria al qual corresponen les dades d'aquesta línia de memòria cau és

l'única còpia en el sistema multiprocessador i que ni tan sols la memòria té el bloc actualitzat.

- Peticions des del processador: si el processador que conté aquest bloc de memòria en la seva memòria cau fa una operació de lectura (`PrRd`) (acció 1 en el graf) o escriptura (`PrWr`) (acció 2 en el graf) sobre alguna dada d'aquesta línia, el processador no necessàriament ha de fer cap acció en el bus, i a més, la línia de memòria cau mantindrà el mateix estat. En cas de tenir un fallada d'escriptura en l'accés a un bloc de memòria diferent (acció 3 en el graf), amb reemplaçament de línia de memòria cau, la línia es quedaria en el mateix estat, però s'hauria d'emetre una acció de lectura per a escriptura en el bus (`BusRdX`) per a obtenir el nou bloc de memòria i un `flush` de la línia reemplaçada. Si tinguéssim una errada de lectura en l'accés a una dada d'un bloc de memòria diferent (acció 4 en el graf), però que fes que caigués en una línia de memòria cau amb estat modificat (la línia de memòria cau s'ha de reemplaçar), la línia de memòria cau passaria a l'estat de compartit (amb el nou bloc de memòria) i el processador realitzaria una petició de lectura en el bus i en un `flush` del contingut de la línia de memòria cau reemplaçada.

- Peticions des del bus: en el cas de rebre una petició de lectura del bus (`BusRd`) d'un bloc de memòria que es conté en una línia de memòria cau amb estat modificat (acció 5 en el graf), la línia de memòria cau s'haurà de subministrar a la memòria i al processador que va sol·licitar el bloc de memòria. Això es fa mitjançant un `flush` de la línia en el bus. La línia haurà de passar a estat compartit. D'altra banda, si la petició en el bus és la de llegir per a modificar (`BusRdX`) (acció 6 en el graf), també haurem de subministrar la línia de memòria cau mitjançant un `flush`, però en aquest cas l'estat de la línia en la nostra memòria cau passa a ser invàlid. Això és perquè només una memòria cau pot contenir un mateix bloc de memòria quan aquest s'ha modificat.

2) *Shared*: Una línia de memòria cau en aquest estat indica que conté un bloc de memòria que no s'ha modificat, i per tant, el bloc pot estar replicat en més d'una memòria cau del sistema multiprocessador. En aquest estat, si el processador fa una petició de lectura `PrRd` (acció 7 en el graf) o bé observa que en el bus s'està fent una petició de lectura (`BusRd`) (acció 8 en el graf) sobre el bloc de memòria de la línia de memòria cau compartida, aquesta no canviarà d'estat. Si es realitza una lectura `PrRd` i es produeix un fallada de memòria cau (acció 9 en el graf), la línia no canviarà d'estat però s'haurà de fer una petició al bus del nou bloc de memòria (`BusRd`). En canvi, si es realitza un petició d'escriptura per part del processador (`PrWr`) (acció 10 en el graf) sobre la línia compartida, aquest haurà d'avisar al bus que vol fer una modificació del bloc de memòria contingut en aquesta línia (`BusRdX`), de tal forma que es poden invalidar les línies de memòria cau de la resta de processadors amb aquest mateix bloc de memòria. En aquest cas, l'estat de la línia de memòria cau passarà a ser *modified*. Finalment, en el cas que s'observi una petició d'escriptura sobre un bloc de memòria (`BusRdX`) (acció 11 en el graf) contingut en una línia de memòria cau amb estat compartit, aquesta línia haurà de canviar l'estat d'invàlid, ja que només un processador pot tenir una còpia modificada d'un bloc de memòria.

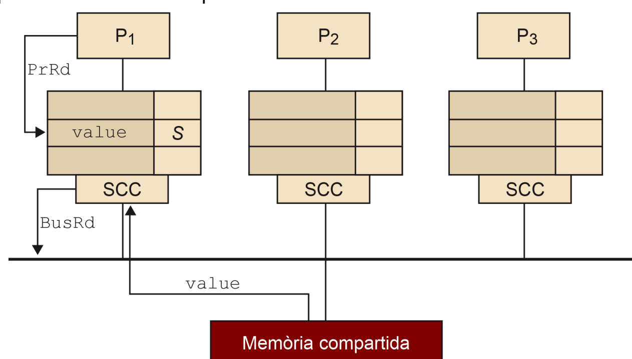
3) *Invalid*: una línia de memòria cau està en aquest estat quan o bé mai no s'ha llegit un bloc de memòria que vagi a aquesta línia, o bé es va llegir però després va quedar

invalidada per una petició d'escriptura sobre el mateix bloc de memòria per part d'un altre processador. Les úniques accions que poden afectar l'estat d'una línia de memòria cau invàlida són les de lectura o escriptura per part del processador (accions 12 i 13 en el graf). Les accions del bus no li afecten perquè aquesta línia de memòria cau no conté cap bloc de memòria vàlid. Així, si fa una petició de lectura el processador (`PrRd`), la línia de memòria cau passarà a l'estat *shared* una vegada que hagi pogut donar l'ordre de petició de lectura a través del bus (`BusRd`), i poder informar la resta de processadors. Si efectua una petició d'escriptura el processador (`PrWr`), en aquest cas la línia de memòria cau passarà a l'estat *modified*, una vegada feta la petició al bus que es farà una escriptura en el bus (`BusRdX`).

A continuació veurem un exemple de com canvien els estats de les línies de memòria cau en les memòries cau de diferents processadors quan s'accedeix a un mateix bloc de memòria en un sistema multiprocessador. Suposem el cas en el qual tenim tres processadors $(1, 2, 3)$, i que fan una seqüència de lectures i escriptures. Cadascun està llegint o escrivint la mateixa dada de memòria. $r1$ significa lectura (r de *read*) del processador 1 i $w3$ significa escriptura (w de *write*) del processador 3. La seqüència d'escriptures i lectures és la següent: $r1$, $r2$, $w3$, $r2$, $w1$, $r3$ i $r1$, i, en començar, suposarem que les línies de memòria cau, on va el bloc de memòria que conté la dada llegida i escrita en cada processador, estan invalidades.

El primer accés a memòria el fa el processador 1. La figura 24 reflecteix cadascuna de les accions que s'han de fer tant en el processador com en el bus, i com queda l'estat de la línia de memòria cau associada a aquesta dada, en cadascun dels tres processadors. Així, el processador 1 fa una lectura d'una dada, i com la seva memòria cau no la té, ha de generar una petició de lectura a través del bus (`BusRd`). En no tenir-la cap altre processador, el controlador de la memòria principal deixarà en el bus el bloc de memòria on es troba la dada, i permetrà que l'SCC del processador que va demanar la dada la llegeixi i actualitzi la línia de memòria cau. L'SCC també actualitzarà l'estat de la línia de memòria cau, que ara passarà a estat compartit (S en la figura).

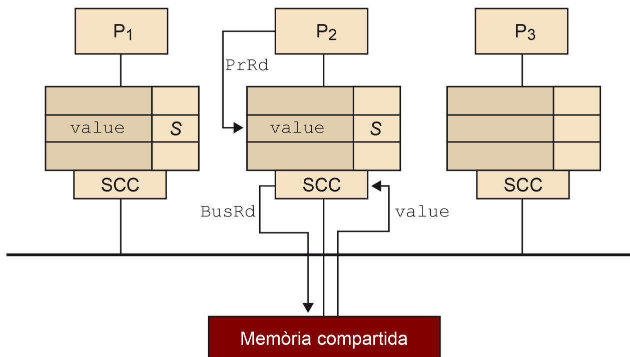
Figura 24. El processador 1 fa una petició de lectura



El segon accés el fa el processador 2, que fa una lectura del mateix valor. La figura 25 mostra l'estat de les línies de memòria cau en els diferents processadors una vegada efectuada la lectura de la dada. Com en l'operació anterior, també es generarà una petició al bus de lectura (`BusRd`) i el controlador de memòria principal subministrarà

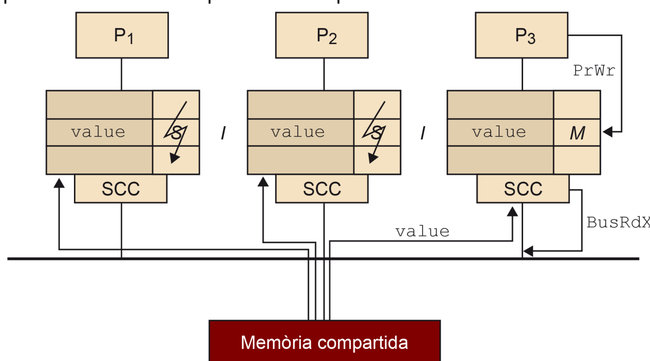
el bloc de memòria en el bus. L'estat de les línies de memòria cau dels processadors 1 i 2 queda com a compartit (S).

Figura 25. El processador 2 fa una petició de lectura



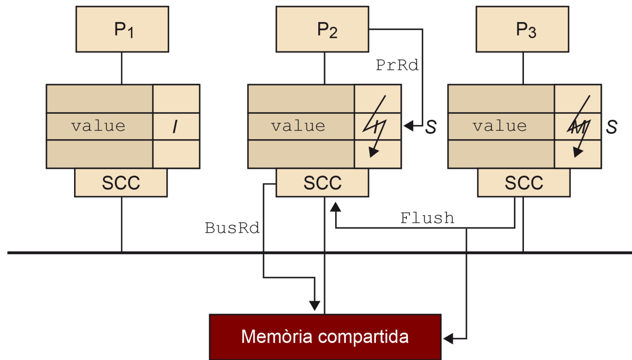
En la tercera operació ($w3$) de l'exemple el processador 3 fa una escriptura sobre un bloc de memòria (PrWr de la figura 26) que no es troba en la memòria cau, per la qual cosa s'ha de demanar en el bus. Així, el processador ha de fer una petició al bus de lectura amb intenció d'escriptura BusRdX per a poder aconseguir el bloc de memòria en exclusivitat. Els processadors 1 i 2 observen en el bus que hi ha una petició de lectura amb intenció d'escriptura d'un bloc de memòria que tenen en la memòria cau. Per tant, invaliden les línies de memòria cau que contenen una còpia del bloc de memòria que es vol modificar. Qui subministrarà el bloc de memòria al processador 3? La memòria serà l'encarregada de fer-ho, ja que té una còpia vàlida del bloc de memòria que s'ha demanat. La figura 26 mostra totes les accions que s'han fet i l'estat en el qual queden les línies de memòria cau dels processadors.

Figura 26. El processador 3 fa una petició d'escriptura



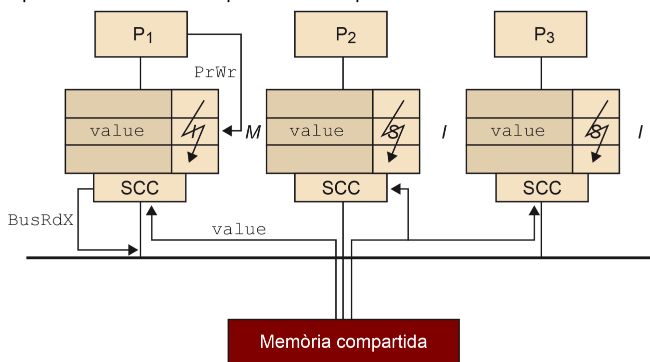
En l'operació següent ($r2$), el processador 2 fa una petició de lectura del bloc de memòria que, en no estar en la memòria cau, haurà de fer una petició de lectura al bus BusRd. El processador 3, que té la còpia actualitzada de la memòria cau, cancel·larà aquesta petició de lectura a la memòria principal, i subministrarà el bloc de memòria que té en la seva memòria cau mitjançant una acció de flush. Aquest flush del bloc de memòria actualitzarà tant la memòria cau del processador 2, que va fer la petició de lectura, com la memòria principal. La figura 27 mostra la sèrie d'accions i transaccions a través del bus que s'han hagut d'efectuar.

Figura 27. El processador 2 fa una petició de lectura



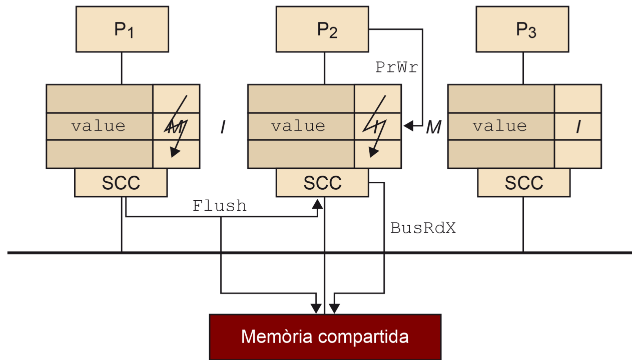
A continuació, el processador 1 fa una escriptura de la dada (operació $w1$) mitjançant la petició `PrWr` sobre el bloc de memòria que no es troba en memòria cau, i ha de fer una petició de lectura amb intenció d'escriptura a través del bus (`BusRdX`). La memòria subministrarà el bloc de memòria al bus, ja que té el bloc actualitzat. L'SCC del processador 1 ho veurà en el bus i actualitzarà la línia de memòria cau amb aquest bloc, i li assignarà l'estat de *modified*. La resta de processadors, en veure en el bus que s'està fent una petició d'escriptura, invalidaran les línies de memòria cau que tenien les còpies d'aquest bloc. La figura 28 preveu totes les accions en el processador i en el bus.

Figura 28. El processador 1 fa una petició d'escriptura



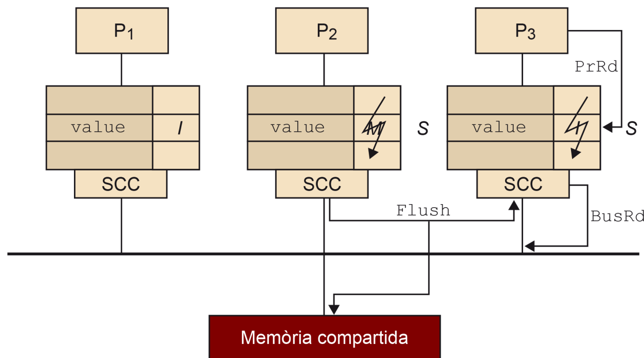
L'operació següent és $w2$, és a dir, el processador 2 fa una escriptura sobre el bloc de memòria que acaba de modificar el processador 1 i que ell no té en memòria cau. El processador 2, per tant, ha de fer una petició de lectura per a modificar el bloc de memòria en el bus (`BusRdX`). El processador 1, que era l'únic que disposava d'una còpia (modificada), observa aquesta operació que apareix en el bus, cancel·la l'operació d'accés a memòria i deixa en el bus l'últim valor del bloc de dades (`flush` en el processador 1). Aquest bloc de memòria s'actualitzarà en memòria i en la memòria cau del processador 2, i quedarà la línia de memòria cau on va el bloc de memòria en l'estat de *modified*. La figura 29 visualitza totes aquestes operacions esmentades.

Figura 29. El processador 2 fa una petició d'escriptura



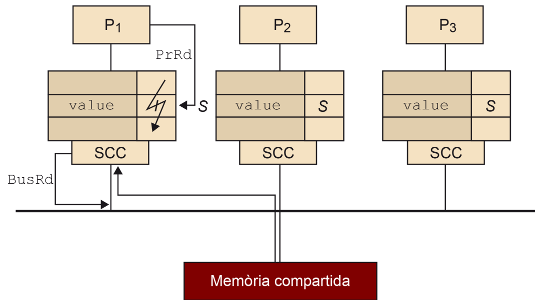
En la penúltima operació de l'exemple, el processador 3 fa una lectura del bloc de memòria. Com no té aquest bloc de memòria en memòria cau, ha de fer una petició de lectura en el bus `BusRd`. Serà el processador 2 el que subministri el bloc de memòria, ja que té la còpia actualitzada. Per a això, aquest ha de cancel·lar la lectura a memòria principal, i posar el bloc de memòria en el bus amb una operació de `flush`. Aquesta operació actualitzarà la memòria principal amb l'últim valor i la memòria cau del processador 3. Les línies de memòria cau del processador 3 i del processador 2 queden en estat *shared*. La figura 30 mostra les accions fetes pel processador i a través del bus.

Figura 30. El processador 3 fa una petició de lectura



Finalment, el processador 1 fa una lectura del bloc de memòria (operació $r1$), que no està en la seva memòria cau. Això fa que hagi de fer una petició de lectura del bloc a través del bus `BusRd`. La memòria principal, que està actualitzada, subministrarà el bloc de memòria. L'estat de la línia de memòria cau de cada processador, que conté una còpia del bloc de memòria, quedarà com a *shared*. La figura 31 mostra aquesta última lectura. A partir d'aquesta situació, qualsevol lectura sobre aquest bloc de memòria no provocarà cap canvi en l'estat de les memòries cau dels processadors, ni tampoc operacions en el bus.

Figura 31. El processador 1 fa una petició de lectura

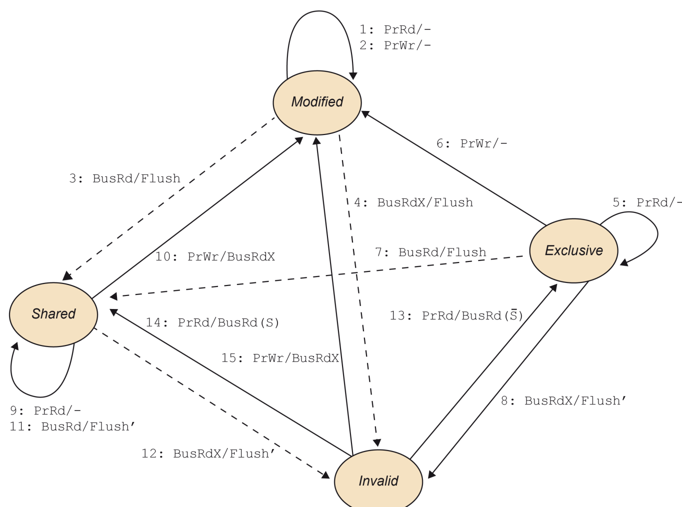


MESI

El protocol de coherència MESI té un estat més que el protocol MSI: el d'exclusivitat. Una línia de memòria cau d'un processador està en aquest estat si el processador fa una lectura d'un bloc de memòria i és l'únic que té còpia d'aquest bloc. Amb aquest estat s'intenten estalviar operacions en el bus, ja que en ser l'única còpia no és necessari informar la resta de processadors. D'altra banda, a diferència del protocol MSI en el qual la memòria sempre subministra el bloc de memòria si el té actualitzat, en la versió original del protocol MESI (la versió d'Illinois), la memòria només subministra un bloc de memòria si aquesta és l'única còpia, independentment de si està actualitzat o no. Per contra, si hi ha una còpia del bloc de memòria actualitzada en una memòria cau, serà aquesta la que subministri el bloc de memòria. Això és el que es diu la tècnica *cache-to-cache sharing*, i el seu objectiu és subministrar més ràpidament les dades al bus. La tècnica *cache-to-cache sharing* requereix que, en cas que hi hagi més d'una memòria cau amb una còpia del bloc de memòria actualitzat, hi hagi un mecanisme per a determinar quina memòria cau donarà el bloc de memòria.

El graf d'estats, per a un cas en què tenim *Snoopy* i *write-invalidate*, i utilitzant la tècnica de *cache-to-cache sharing* de la versió original d'Illinois, es mostra en la figura 32. Per simplificar la figura no mostrem les transicions per a operacions de lectura amb error de memòria cau.

Figura 32. El graf d'estats d'una dada (línia de memòria cau) en un processador amb protocol de coherència MESI



El protocol MESI té els estats següents:

1) *Modified*: aquest estat indica que la línia de memòria cau conté un bloc de memòria vàlid i modificat en la memòria cau. A més, com el bloc de memòria està modificat, no hi pot haver una altra còpia d'aquest en cap altra memòria cau d'un altre processador ni en la memòria principal.

2) *Exclusive*: quan una línia de memòria cau es troba en aquest estat indica que el bloc de memòria és vàlid en memòria cau, però cap altra memòria cau conté una còpia del bloc de memòria. A més, la memòria principal està actualitzada, ja que el bloc de memòria no ha estat modificat. Per a poder saber si una lectura d'un bloc de memòria es fa en exclusivitat, necessitem un senyal (per processador) que ens permeti saber si els processadors contenen una còpia o no del bloc de memòria llegit.

3) *Shared*: una línia de memòria cau amb aquest estat indica que el bloc de memòria és vàlid, no està modificat, i per tant la memòria principal està actualitzada, però hi pot haver una còpia del bloc de memòria en una altra memòria cau.

4) *Invalid*: en aquest cas, la línia de memòria cau no té cap bloc de memòria vàlid, ja sigui perquè no es va carregar mai o perquè es va haver d'invalidar.

Des del punt de vista de la memòria cau d'un processador del sistema, les transicions entre els estats que poden tenir cadascuna de les línies de la memòria cau són les següents:

1) *Modified*: en el cas que el processador faci lectures o escriptures en un bloc de memòria que estigui en memòria cau (`PrRd`, acció 1 de la figura, i `PrWr`, acció 2 de la figura, respectivament), l'estat de la línia de memòria cau que el conté es mantindrà i, a més, no s'enviarà cap operació al bus, ja que sabem que aquest bloc de memòria és l'única còpia existent. Si s'observa en el bus una petició de lectura (`BusRd`) sobre el bloc de memòria de la línia de memòria cau (acció 3 de la figura), aquesta passarà a l'estat *shared* i, en cas d'una operació de lectura per a escriptura (`BusRdX`, acció 4 de la figura), la línia de memòria cau passarà a l'estat invàlid. En tots dos casos, es fa un `flush` de la dada modificada i es cancel·la l'accés a la memòria, ja que aquesta no té el bloc de memòria més actualitzat.

2) *Exclusive*: en el cas que el processador faci una lectura que encerti en la línia de memòria cau (acció 5 de la figura), aquesta no canviarà d'estat. A més, en ser el bloc de memòria l'única còpia en el sistema, el processador no enviarà cap operació al bus. En canvi, si el processador fa una operació d'escriptura sobre el bloc de memòria de la línia (acció 6 de la figura), aquesta passarà a l'estat *modified* (bloc de memòria modificat). En aquest cas, el processador tampoc no enviarà cap operació al bus, ja que sabem que la línia de memòria cau conté l'única còpia d'aquesta dada.

Si l'SCC del processador observa una operació de lectura (`BusRd`, acció 7 de la figura), el processador, que té una còpia del bloc de memòria, posarà el bloc en el bus amb una operació (`flush`) per a proporcionar la dada al processador que va fer la petició. La línia de memòria cau passarà a l'estat de *shared*. En canvi, si s'observa una petició

de lectura per a escriptura en el bus (acció 8 de la figura), el processador subministrarà el bloc de memòria, però en aquest cas la línia de memòria cau passarà a ser invàlida.

3) *Shared*: en el cas que el processador faci una petició de lectura del bloc de memòria de la línia (acció 9 de la figura), aquesta no canviarà d'estat i no es farà cap petició al bus. Si el processador fa una petició d'escriptura sobre aquesta línia (acció 10 de la figura), el processador generarà una petició de lectura per a modificar en el bus (`BusRdX`) i la línia passarà a l'estat *modified*.

En cas d'observar una petició de lectura en el bus (acció 11 de la figura), el processador, si és l'escollit per a subministrar el bloc de memòria en el bus (`flush'`), posarà el bloc de memòria en el bus però no canviarà l'estat de la línia. Aquest bloc en el bus serà llegit pel processador que va iniciar la petició. En el cas d'observar una petició d'escriptura (acció 12 de la figura) es procedirà de la mateixa manera, però l'estat de la línia de memòria cau passarà a ser invàlid.

4) *Invalid*: en el cas que el processador faci una petició de lectura (acció 13 de la figura), si la petició en el bus de lectura `BusRd` no obté com a resposta un senyal d'algun dels processadors que indiqui que conté el bloc de memòria (`BusRd(`$\bar{S}$`)`), passarà a l'estat d'exclusivitat. En cas contrari (acció 14 de la figura), si s'obté un senyal d'algun dels processadors `BusRd(`S`)`, l'estat passa a ser *shared*.

Finalment, si el processador fa una escriptura sobre el bloc de memòria de la línia (acció 15 de la figura), farà una petició de lectura per a escriptura `BusRdX` i la línia passarà a l'estat de *modified*.

Veurem un exemple per a acabar d'entendre el protocol MESI. En aquest exemple es faran els accessos indicats en la taula 4.

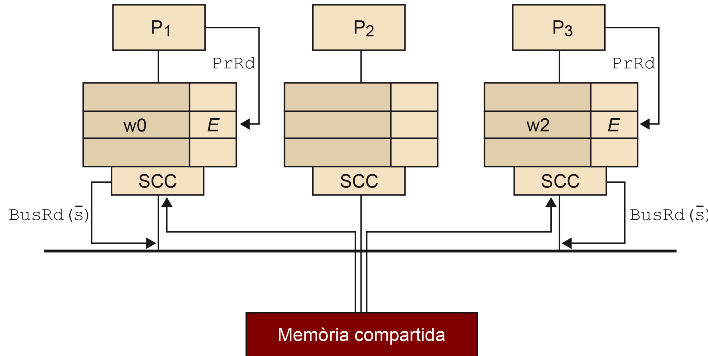
Taula 4. Seqüència de lectures i escriptures fetes per tres processadors en diferents temps

Temps	Processador-operació	Processador-operació
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	

Per al primer instant tenim dos accessos de lectura de dues dades diferents que van a blocs de memòria diferents, però que poden o no anar a la mateixa línia de memòria cau. En aquest cas, si anessin a la memòria cau del mateix processador, es maparien en la mateixa línia de memòria cau. No obstant això, com el llegeixen dos processadors diferents, van a la mateixa línia però de memòries cau diferents, tal com podem observar en la figura 33. Tots dos processadors, P_1 i P_2, fan la petició de lectura `PrRd` en el processador i una petició de lectura en el bus `BusRd`. Com no hi ha cap còpia del bloc de memòria en un altre processador, el senyal de dada compartida (S) del bus no s'activarà –la transició serà `BusRd(`$\bar{S}$`)`– en el graf de la figura 32 i, per tant, tots dos

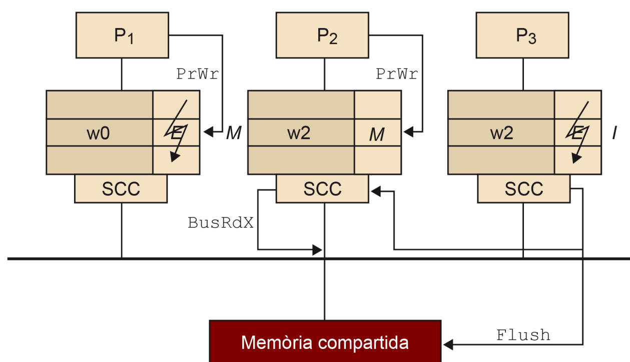
rebran el bloc de la memòria (dada) que han demanat i canviaran l'estat de les seves línies de memòria cau respectives a *exclusive*. Recordem que les dades estan en blocs diferents de memòria, i per això l'exclusivitat.

Figura 33. Els processadors 1 i 3 fan una petició de lectura de les dades $w0$ i $w2$, respectivament



En el segon instant (figura 34) es produeixen dues escriptures sobre les dades llegides anteriorment, $w0$ i $w2$, però ara són el processador P_1, que ja disposa del bloc de memòria en la seva memòria cau, i el P_2, que fallarà en el seu accés a memòria cau. En el cas del processador P_1, com ja té el bloc de memòria en exclusivitat, només haurà de modificar la seva dada amb la petició d'escriptura del processador (`PrWr`) i no haurà de fer cap operació sobre el bus. L'estat de la línia de memòria cau passarà a ser *modified*. En canvi, en el cas del processador P_2, aquest haurà de fer una petició de lectura amb intenció d'escriptura `BusRdX` en el bus. Això farà que el processador P_3 observi en el bus la petició sobre el bloc de memòria amb $w2$, que ell té en memòria cau, i per tant, canviarà l'estat de la seva línia de memòria cau a invàlid. A més, el processador P_3 subministrarà el bloc de memòria en el bus amb una operació de `flush` (tècnica *cache-to-cache sharing*).

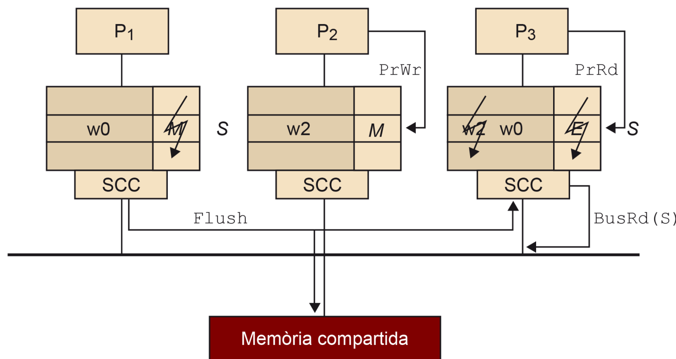
Figura 34. Els processadors 1 i 2 fan una petició d'escriptura de les dades $w0$ i $w2$, respectivament



Les operacions següents són una escriptura del processador P_2 sobre $w2$ amb encert en memòria cau, i una lectura del processador P_3 sobre la dada $w0$ amb error en memòria cau. Aquesta última dada (bloc de memòria) la té el processador P_1 en una línia de memòria cau amb estat *modified*. En el primer cas, com és una escriptura sobre una dada que ja està modificada en la memòria cau del processador de la petició, només s'ha de fer l'operació sobre la línia, sense necessitat de provocar cap operació sobre el

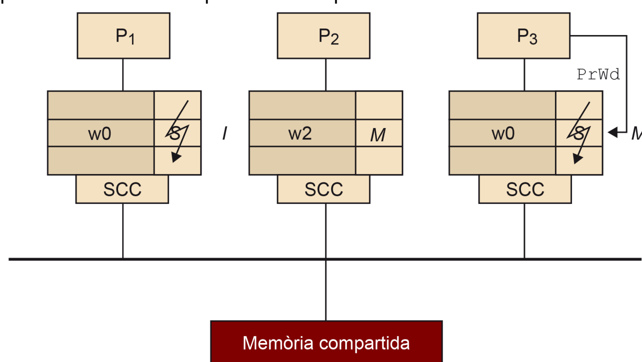
bus. En el segon cas, no obstant això, la petició del processador P_3 necessita fer una petició en el bus, ja que s'ha produït un error en memòria cau (línia de memòria cau amb estat invàlid). En fer la petició de lectura sobre el bus (BusRd), el processador P_1, que té el bloc de memòria modificat en la seva memòria cau, cancel·larà l'accés a memòria, i subministrarà el bloc de memòria a la memòria principal i al processador P_3. Els processadors P_1 i P_3 canviaran l'estat de les seves línies de memòria cau respectives a compartit (*shared*). Totes aquestes operacions es reflecteixen en la figura 35.

Figura 35. Els processadors 2 i 3 fan una petició d'escriptura de la dada $w2$ i de lectura de la dada $w0$, respectivament



En el quart instant, el processador P_3 fa una petició d'escriptura sobre $w0$, per la qual cosa l'estat de la línia de memòria cau ha de canviar a *modified*. A part d'això, quines operacions a través del bus s'han de fer perquè el P_1 passi la línia de memòria cau amb aquest bloc de memòria a l'estat d'invàlid? En la figura 36 es mostra com haurien de quedar els estats de les dades llegides fins al moment. Intenteu completar el dibuix amb les operacions sobre el bus.

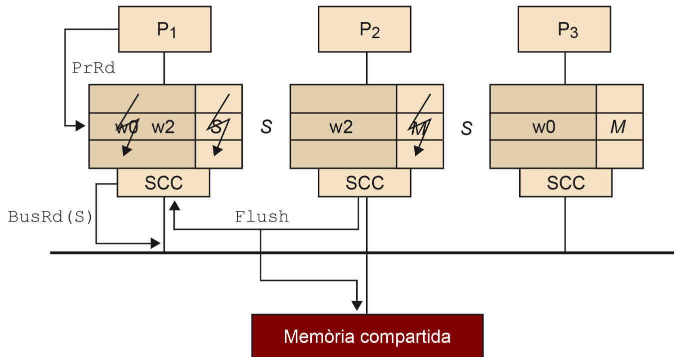
Figura 36. El processador 3 fa una petició d'escriptura de la dada $w0$



En el cinquè instant, el processador P_1 fa una lectura de $w2$ amb error en memòria cau (tenia un altre bloc de memòria en memòria cau, el de la dada $w0$). Per tant, el processador haurà de fer una petició de lectura sobre el bus (BusRd). El senyal de compartició S del bus s'activarà com a resposta a aquesta petició de lectura, ja que el processador P_2 té una còpia del bloc de memòria. A més, el processador P_2 subministrarà el bloc de memòria en el bus (flush) perquè el processador P_1 i la memòria principal el puguin actualitzar. Les línies de memòria cau amb el bloc de

memòria de $w2$ dels processadors passen a tenir l'estat *shared*. La figura 37 reflecteix totes aquestes operacions fetes en el processador i en el bus.

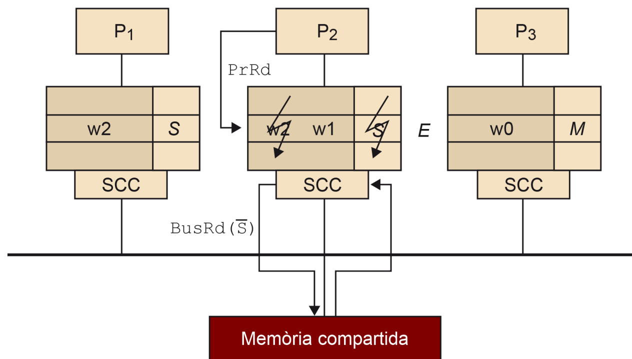
Figura 37. El processador 1 fa una petició de lectura de la dada $w2$



En aquest moment, tenim la dada $w2$ en estat *shared* en les memòries cau dels processadors P_1 i P_2, i la dada $w0$, en estat *modified* en la memòria cau de P_3.

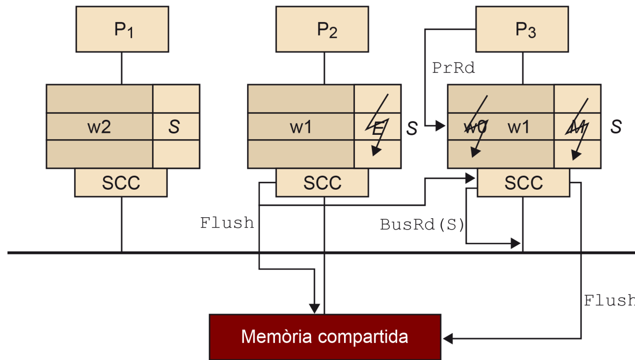
A continuació (sisè instant) el processador P_2 farà una lectura de $w1$ amb error en memòria cau (tenia un altre bloc de memòria en memòria cau, el de la dada $w2$). Per tant, el processador P_2 haurà de fer una petició de lectura al bus (`BusRd`). En no haver-hi cap altra còpia d'aquest bloc de memòria, el senyal de compartició (S) restarà inactiu en el bus (`BusRd(`$\bar{S}$`)`), i el processador P_2 passarà la línia de memòria cau amb el nou bloc de memòria a estat *exclusive*. La figura 38 mostra els estats de les diferents línies de memòries cau dels processadors. En aquesta situació, el processador P_1 hauria de passar la línia de memòria cau amb la dada $w2$ a un estat d'exclusivitat?

Figura 38. El processador 2 fa una petició de lectura de la dada $w1$



Finalment, el processador P_3 fa una lectura de la dada $w1$ amb error en memòria cau. Com a conseqüència d'aquest error el processador P_3 farà una petició de lectura al bus. El processador P_2, que té el bloc de memòria modificat en la seva memòria cau, cancel·larà la lectura de memòria, posarà el bloc de memòria en el bus (`flush`) perquè el processador P_2 i la memòria principal es puguin actualitzar, i indicarà que ell disposa d'una còpia amb el senyal de compartició S. Tant el processador P_2 com el processador P_3 canviaran l'estat de les seves línies de memòria cau respectives, amb la dada $w1$, a estat compartit. La figura 39 mostra el contingut de les memòries cau i l'estat d'aquestes.

Figura 39. El processador 3 fa una petició de lectura de la dada $w1$



L'Intel i7 usa una variant del protocol MESI, anomenat MESIF que afegeix un nou estat (*Forward*) per a indicar quin processador dels que comparteixen una dada hauria de respondre una petició.

Protocol MESIF

El multiprocessador Intel i7 incorpora un sistema basat en directoris en la memòria cau L3 amb una variant del protocol MESI anomenat *MESIF*.

True i *false sharing*

En el moment en què tenim un sistema multiprocessador amb memòria amb coherència de memòria cau podem observar dos tipus de conflictes en la compartició de les dades: *true sharing* i *false sharing*.

El *true sharing** d'aquests és el resultat d'estar accedint a la mateixa dada des de dos o més processadors. Això provoca, tal com hem vist en els protocols de coherència, que la dada s'hagi d'actualitzar (*write-update*) o invalidar (*write-invalidate*) si es fa alguna escriptura a la dada. Si es fan moltes escriptures a la dada des de diferents processadors això pot provocar que els accessos al bus i el manteniment del sistema de coherència de memòria cau es converteixin en un coll d'ampolla.

*** *True sharing*: es produeix quan es comparteix la mateixa dada de memòria.**

El segon d'aquests conflictes, *false sharing**, sorgeix com a conseqüència de mantenir la coherència de memòria cau a escala d'un bloc de memòria (línia de memòria cau) i no de dada. Així, dues dades diferents, a la qual accedeixen dos processadors diferents, poden compartir un mateix bloc de memòria i, llavors, compartiran la coherència del mateix bloc. Per exemple, si un processador P_1 llegeix una d'aquestes dades que comparteixen bloc de memòria, l'estat de la seva línia de memòria cau serà *exclusive* o *shared* depenent de si és l'únic processador amb còpia o no d'aquest bloc. Si ara un altre processador P_2 escriu en una altra dada diferent, però en el *mateix* bloc de memòria, aquest provocarà que s'hagi d'invalidar la línia de memòria cau del processador P_1, ja que el processador P_2 modificarà una dada del mateix bloc de memòria. Això és el que es diu *false sharing* o compartició falsa, perquè el que es comparteix és el bloc de memòria i no la mateixa dada.

*** *False sharing*: es produeix quan no es comparteix la mateixa dada de memòria, però sí el mateix bloc de memòria, amb la qual cosa dos processadors poden estar lluitant per l'exclusivitat d'aquest bloc de memòria.**

La manera d'evitar aquest tipus de situacions, en les quals un dels processadors ha d'escriure i, per tant, hi pot haver invalidacions innecessàries, és intentar que les dades a les quals accedeixen diferents processadors no siguin consecutives en memòria per a evitar que estiguin en el mateix bloc de memòria (línia de memòria cau). Un exemple de la situació esmentada podria ser el cas de tenir un vector d'enters, els elements

dels quals són actualitzats freqüentment per diferents processadors. Això pot provocar molts errors de memòria cau a causa de les invalidacions per compartició falsa.

Analitzarem com es pot produir *false sharing* en el moment de fer la paral·lelització amb OpenMP del codi 2.1, que és el codi seqüencial del producte 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 }
```

Codi 2.1: Código secuencial del producto escalar.

Observem que la variable sum s'hauria d'actualitzar dins d'una exclusió mútua o bé amb atomic, ja que hauria de ser actualitzada en paral·lel per tots els *threads*. Per a evitar aquesta sincronització, es podria pensar que cada *thread* suma una variable global diferent, que després el *thread master* pot llegir i obtenir la suma total, amb la suma de les sumes parcials dels resultats obtinguts en cada *thread*.

El codi 2.2 és la versió OpenMP, usant un vector de reals per a fer les sumes parcials per part de cada *thread*.

```
1  #include "omp.h"
2  #define NUM_THREADS = 4
3  static long num_steps = 100000;
4  void main ()
5  {
6    int i, id;
7    double x, pi, step, sum[NUM_THREADS];
8
9    step = 1.0/(double) num_steps;
10   omp_set_num_threads(NUM_THREADS);
11
12   #pragma omp parallel private(id)
13   {
14     id = omp_get_thread_num();
15     sum[id] = 0.0;
16
17     #pragma omp for private(x,i)
18     for (i=1;i<= num_steps; i++) {
19        x = (i-0.5)*step;
20        sum[id] += 4.0/(1.0+x*x);
21     }
22
23     #pragma omp single
24     for (i=0, pi=0.0; i<NUM_THREADS; i++)
25        pi += step * sum[i];
26   }
27 }
```

Codi 2.2: Código OpenMP del producto escalar.

Analitzant una mica com s'executarà el codi, i sabent que diversos reals del vector `double sum[NUM_THREADS]` poden estar en el mateix bloc de memòria (línia de memòria cau), cada *thread* podrà estar provocant una invalidació de les línies de memòria cau de la resta de processadors a causa que tots estan actualitzant el mateix bloc de memòria. Això provocarà errors de memòria cau i un elevat trànsit per mitjà del bus, que és innecessari.

Com podem solucionar el problema de *false sharing* en aquest exemple? Una solució possible i senzilla és afegint *padding* al vector, de tal manera que obligui que els elements del vector que s'usin per a calcular la suma parcial estiguin en blocs de memòria (línies de memòria cau) diferents. El codi 2.3 mostra la declaració amb el *padding* suposant que els blocs de memòria són de 64 bytes.

Padding

El *padding* és una tècnica per la qual s'afegeixen elements *escombraria* que serveixen per a separar elements, aconseguir un alineament concret de les dades, o aconseguir que una estructura ocupi un determinat nombre de bytes.

```
1        ...
2        double sum[NUM_THREADS][8];
3        ...
```

Codi 2.3: Ejemplo de *padding*.

D'aquesta manera podríem utilitzar el primer element de `sum[id][0]`, sabent que, com tenim 8 `doubles` per línia de memòria cau (bloc de memòria), no tindrem ara *false sharing*.

3. Multicomputador

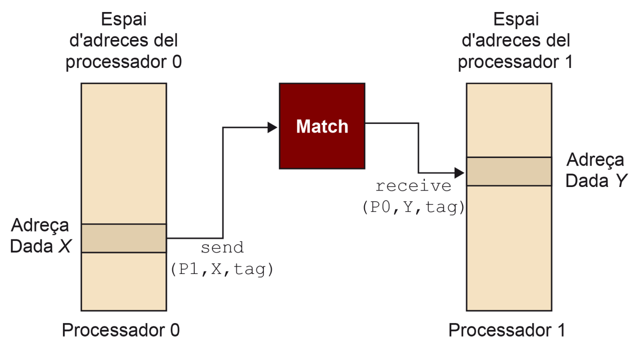
Els multicomputadors, tal com ja s'ha comentat en la introducció, van sorgir de la necessitat de poder escalar en nombre de processadors sense un elevadíssim cost econòmic. Aquests consisteixen en un conjunt de processadors i bancs de memòria que es connecten a través d'una xarxa d'interconnexió amb una determinada topologia de xarxa.

La principal característica dels multicomputadors és que els processadors, al més baix nivell, ja no poden compartir dades per mitjà de `loads/stores`, sinó que ho han de fer per mitjà de missatges. És per això que en aquest tipus de sistemes no hi ha problemes de coherència de memòria cau ni de consistència de memòria. Aquí, la sincronització es fa explícita amb els missatges, tal com mostra la figura 40, amb primitives de l'estil `send` i `receive`. En el `send` s'especifica el *buffer* per enviar i a qui s'envia. Per l'altre costat, el `receive` ha d'especificar el *buffer* de recepció i de qui el rep. Els *buffers* del `send` i del `receive` estan en espais d'adreces diferents, suposant que són processos diferents. Opcionalment, també es pot especificar una etiqueta o *tag* al missatge, per a rebre un missatge concret d'un processador concret, de manera que es compleixi la *matching rule*.

Nota aclaridora

En els sistemes multiprocessador, al més baix nivell, també es podria considerar que s'estan fent missatges, un missatge format per la línia de memòria cau, però és generat per una instrucció d'accés a memòria.

Figura 40. Comunicació amb primitives de comunicació `send` i `receive`



Una pregunta que ens podem fer és: qui ha de fer la comunicació entre processos? Normalment és el programador el que utilitza un *message-passing paradigm*; és a dir, el programador usarà una API d'una biblioteca d'usuari, que exporta el model de comunicació al programador. No obstant això, en determinats sistemes, hi pot haver un *programari DSM (distributed-shared memory)*, que consisteix en una capa de programari transparent als programadors i que oculta la característica de memòria distribuïda, i implementa una memòria compartida en programari. En un sistema amb DSM, el sistema operatiu té un paper important, ja que el funcionament se sol basar en els errors de pàgina en els accessos. Això implica que els programadors poden treballar amb accessos `load/store` per a accedir a les dades d'un procés que està en un altre node sense utilitzar de manera explícita el model de pas de missatges. A baix nivell, aquest programari, després de detectar un error de pàgina en l'accés a una dada amb `load/store` per part d'un processador, utilitzarà el model de pas de missatges per a comunicar pàgines de memòria d'una memòria física a una altra.

3.1. Xarxes d'interconnexió

Les xarxes d'interconnexió permeten la comunicació de dades entre nodes (processadors), o entre processadors i memòria, tal com vam veure en les xarxes d'interconnexió per als sistemes multiprocessador en l'apartat anterior. Els processadors o els bancs de memòria es connectaran a les entrades i sortides de la xarxa d'interconnexió.

Normalment, les xarxes d'interconnexió estan formades per *switches* i *links* connectats uns amb altres. La capacitat de comunicació dels *links* depèn de les seves característiques físiques (*capacity coupling* i fortalesa de senyal) que en part depenen, al seu torn, de la longitud del *link*. Els *switches* poden ser més o menys sofisticats, i poden contenir *buffering* per a desacoblar el port de sortida del port d'entrada, encaminament per a reduir la congestió, i *multicast* per a fer *broadcast* d'una mateixa sortida. El mapatge dels ports d'entrada als ports de sortida pot variar segons diferents mecanismes, i aquest mecanisme i el grau del *switch* influeixen en el cost final d'aquests *switches*.

Switch

En una xarxa d'interconnexió, un *switch* consisteix en un dispositiu maquinari amb una sèrie de ports d'entrada i altres de sortida. El nombre total de ports d'un *switch* és el grau d'aquest *switch*. Els ports d'entrada estan connectats amb els de sortida mitjançant un *crossbar* que es pot configurar.

Per a connectar els nodes als *switches* es necessita una interfície de xarxa. Aquesta interfície ha de suportar amplades de banda més elevades que les necessàries per a connectar-se amb entrada/sortida, ja que els busos de memòria solen ser molt més ràpids. La interfície de xarxa normalment té la responsabilitat d'empaquetar dades, fer l'encaminament dels paquets, fer *buffering* de les dades d'entrada i sortida, control d'errors, etc.

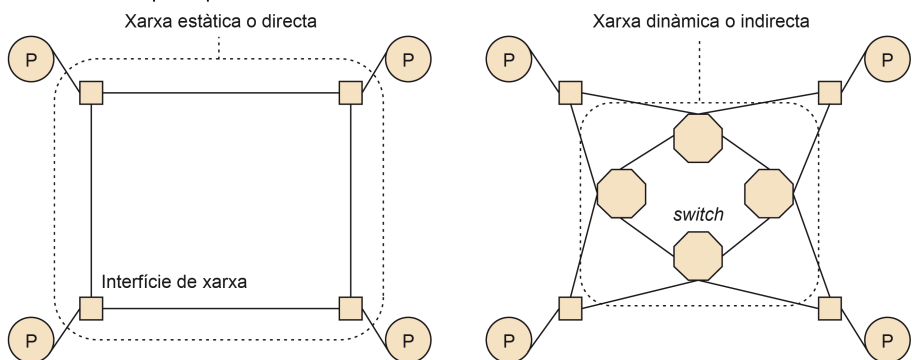
Les xarxes d'interconnexió es poden dividir en *static* o directes, i *dynamic* o indirectes.

- **Directes**: les xarxes directes són aquelles en les quals els nodes de processament estan connectats punt a punt via *links*.

- **Indirectes**: en aquest cas els processadors pot ser que no estiguin connectats punt a punt, i els *links* poden connectar diversos *switches* de tal manera que es puguin arribar a establir diferents camins entre els nodes de processament i entre els nodes i els bancs de memòria.

La figura 41 mostra un exemple d'una xarxa d'interconnexió directa (esquerra) i indirecta (dreta) formada per 4 processadors connectats punt a punt i via *switches*, respectivament.

Figura 41. Exemples d'una xarxa d'interconnexió directa i una altra de tipus indirecta, totes dues formades per 4 processadors



3.1.1. Mètriques d'anàlisi de la xarxa

En aquest subapartat analitzarem algunes de les mesures que s'utilitzen per a avaluar una xarxa d'interconnexió. En el subapartat següent analitzarem algunes xarxes d'interconnexió amb diferents maneres de connectar-se.

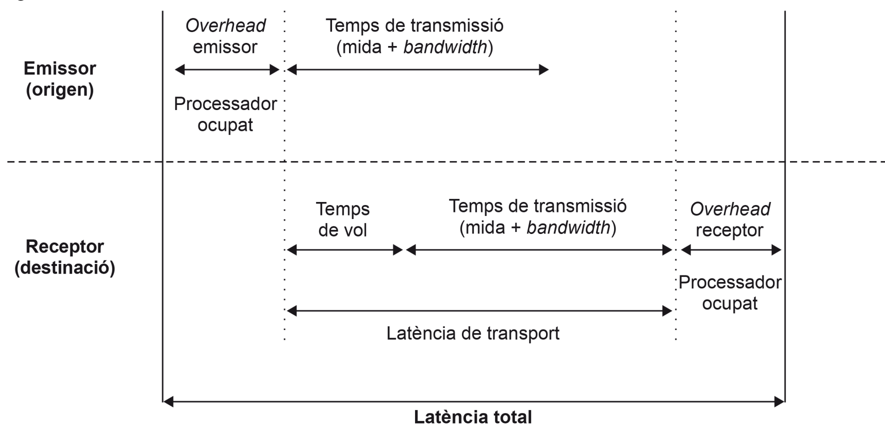
Latència i amplada de banda de la xarxa

La latència de la xarxa indica el temps que es necessita per a fer la comunicació d'una sola dada d'un processador a un altre. La latència dels missatges a la xarxa cobra importància en aquells programes que han de fer un elevat nombre de missatges petits.

L'amplada de banda de la xarxa és la quantitat de dades per unitat de temps que es poden mantenir en la comunicació d'un conjunt de dades. La importància de l'amplada de banda d'una xarxa és més gran en els programes que fan un nombre elevat de missatges grans.

En qualsevol cas, volem fer notar que la latència màxima que ofereix una xarxa, segons el fabricant, moltes vegades no s'aconsegueix perquè hi ha una sèrie de costos que es paguen en fer la comunicació. Per tant, la latència que el programador observarà és la latència de la xarxa d'interconnexió més la latència del programari que s'utilitza per a fer la comunicació. La figura 42 reflecteix les diferents parts que afecten l'enviament d'una dada.

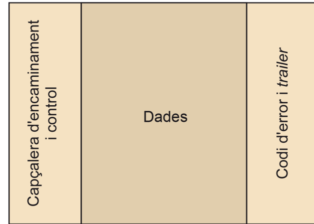
Figura 42. Latència de comunicació



D'una banda, veiem que el processador que envia ha de pagar un temps a preparar el missatge i col·locar-lo a la xarxa perquè es comenci a transmetre (*overhead* de l'emissor), després hi ha un temps de transmissió, el temps que es necessita perquè arribi un missatge d'un processador a l'altre (temps de vol) i, finalment, hi ha un cost de recepció del missatge per part del processador destinació (*overhead* del receptor).

D'altra banda, l'amplada de banda que aconseguim en l'enviament de dades per mitjà de la xarxa normalment és més petita que el màxim que ens ofereix la xarxa d'interconnexió segons el fabricant. Això és a causa que cal empaquetar les dades i incloure-hi l'encaminament, control d'errors, capçalera, etc., tal com mostra la figura 43. L'amplada de banda que realment aconseguirem és el que se sol anomenar *effective bandwidth*.

Figura 43. Informació addicional (*overhead*) per a poder enviar un missatge



Fanout o *connectivity* de la xarxa

El nombre de connexions en un node es diu el grau (matemàticament), *fanout* o *connectivity*, segons els enginyers. El *fanout* ens pot indicar quina tolerància a errors té el processador, ja que com més gran sigui el grau d'un node més camins possibles tindrem en el moment d'encaminar la comunicació. A més, tenir un grau elevat permet reduir la contenció de la xarxa en alguns camins.

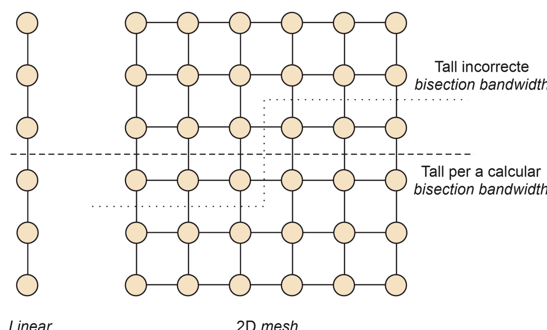
Diameter de la xarxa

El *diameter* és la distància màxima entre dos nodes, i la distància entre dos nodes és el nombre de *links* que hem de travessar per a anar d'un node a l'altre. El *diameter*, per tant, ens dóna una visió de quant ens costarà l'enviament d'un node a un altre, ja que cada *link* que passem pot significar un petit cost de comunicació addicional. La distància mitjana entre nodes també ens pot donar una idea del que ens costa cada comunicació, és a dir, de la latència d'un missatge.

Bisection bandwidth

La *bisection bandwidth* ens dóna una mesura de la quantitat de dades que es poden enviar en les transmissions sense tenir contenció a la xarxa d'interconnexió. Per a calcular la *bisection bandwidth* es divideix la xarxa d'interconnexió en dues parts, eliminant el mínim nombre de *links* entre totes dues parts, de tal manera que les dues parts tinguin el mateix nombre de nodes. El nombre de *links* eliminats, multiplicat per l'amplada de banda de cadascun, ens determina la *bisection bandwidth*. En la figura 44 mostrem quin és el tall correcte per a calcular la *bisection bandwidth* d'una xarxa d'interconnexió *linear array* (esquerra) i una altra de tipus *2D mesh* (dreta). Els punts són nodes o *switches*, i les línies que els uneixen són els *links*. Noteu que també mostrem un exemple del que no seria un tall correcte per a calcular la *bisection bandwidth*, ja que s'eliminen més *links* dels necessaris.

Figura 44. Exemple de tall d'una xarxa en el càlcul de la *bisection bandwidth* d'un *linear array* i d'un *2D mesh*



Molts dissenyadors de xarxes d'interconnexió intenten fer topologies de xarxa que maximitzin aquesta mesura de les xarxes, amb la idea de minimitzar la contenció.

Dimensionalitat

La dimensionalitat és la quantitat d'opcions que es tenen per a anar d'un node a un altre. Si d'un node a un altre no hi ha més que una possibilitat de camí, llavors es diu que és zero dimensional. No obstant això, si ens podem moure cap a l'est o cap a l'oest per a anar a un determinat node, llavors direm que és un dimensional. Si podem, a més, anar cap al nord o cap al sud, direm que és dos dimensional.

3.1.2. Topologia de xarxa

La topologia de xarxa és un patró en el qual els *switches* estan connectats a altres *switches* mitjançant *links*. Les interconnexions es mostren amb grafs en què els nodes representen o bé processadors o *switches*, i les arestes els *links*. La topologia de xarxa influeix en la latència dels missatges, en el *bandwidth* aconseguit, i en la congestió que hi pugui haver en les comunicacions, tal com s'ha pogut reflectir en les diferents mesures de rendiment que s'han comentat en el subapartat anterior.

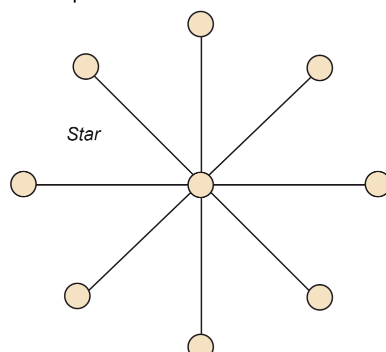
A continuació analitzarem diverses topologies de xarxa, tant directes com indirectes. En les figures que apareguin en aquest subapartat només mostrarem els *links* i els *switches* entre aquests. No es mostraran ni les memòries ni els processadors, que solen estar connectats amb una interfície de xarxa als *switches*.

Les xarxes d'interconnexió basades en un bus, el *crossbar* i el *multistage*, es van detallar en l'apartat 2, dedicat als multiprocessadors. Ara ens centrarem en les xarxes d'interconnexió següents: *star*, *full interconnect*, *linear and meshes* i *tree interconnect*.

Star

La topologia de xarxa *star* és una tipologia zero dimensional. Consisteix en un node central que està connectat a la resta de nodes en l'exterior. Aquesta topologia de xarxa és molt senzilla d'implementar però té dos inconvenients principalment: (1) el node central es pot convertir en un coll d'ampolla en les comunicacions, i (2) si el node central falla, la resta de nodes es quedarien desconnectats. La figura 45 mostra un exemple de topologia *star*.

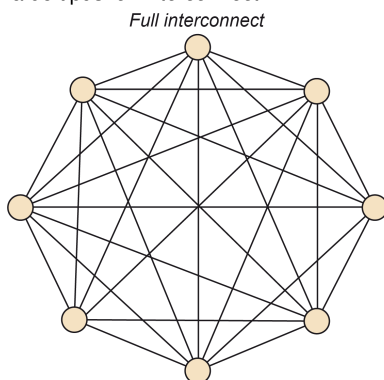
Figura 45. Topologia de xarxa de tipus *star*



Full interconnect

La *full interconnect* és una zero dimensional també perquè per a anar d'un node a un altre només hi ha un camí. A diferència de l'anterior, aquesta és completament tolerant a errors, maximitza la *bisection bandwidth*, i minimitza el *diameter*. El seu principal desavantatge és que és molt poc escalable. La figura 46 mostra una xarxa d'interconnexió amb topologia *full interconnect* o *completely-connected*. Aquest tipus de xarxa és la topologia de xarxa directa equivalent a la indirecta *crossbar*.

Figura 46. Topologia de xarxa de tipus *full interconnect*

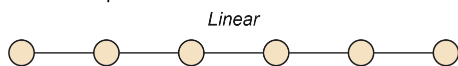


Linear and meshes

Aquests tipus de xarxa van aparèixer com a solució a l'elevat cost que representaven les xarxes d'interconnexió totalment connectades. En tenim de diferents tipus: *linear arrays*, *meshes* o *grids* amb connexions en les cantonades (*rings*) o sense, i finalment *cubes* i *hypercubes*.

Linear array és zero dimensional, ja que per a anar d'un node a un altre només ho podem fer per un camí. Cada node, a excepció dels nodes dels extrems, té un veí a la dreta i un altre a l'esquerra. El *diameter* està determinat per la distància entre els dos nodes dels extrems. Una possible extensió del *linear array* és la *ring interconnect*. Aquesta és u dimensional, en poder anar a l'esquerra o a la dreta en el moment d'anar d'un node a un altre. El *diameter* en aquest cas és la meitat que en el *linear array*, ja que la distància més llarga entre nodes és entre un dels extrems i el node central. La figura 47 mostra un exemple de *linear array*.

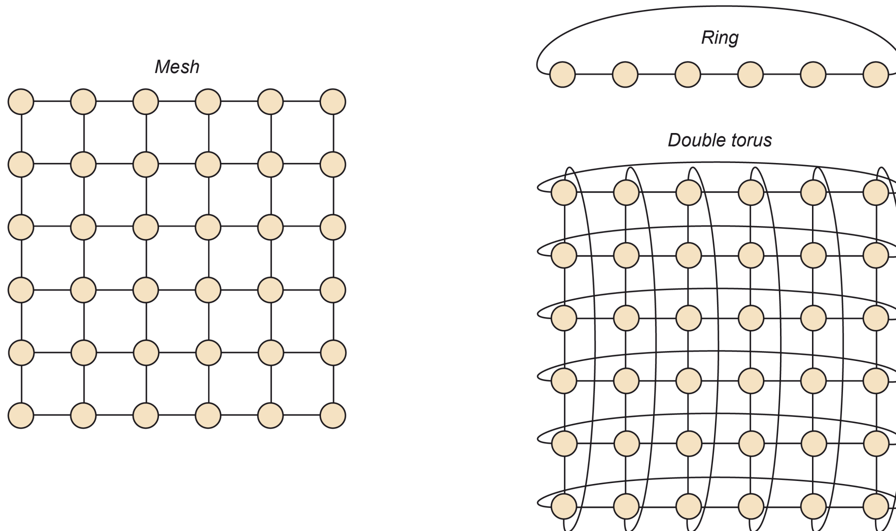
Figura 47. Topologia de xarxa de tipus *linear*



Grid o mesh és l'extensió dels *linear arrays* en una o més dimensions. Aquest tipus d'interconnexió és 2 o més dimensional, ja que cada node, a excepció dels nodes en els extrems, té com a mínim 4 connexions amb 4 nodes veïns (dreta, esquerra, a dalt i a baix). Cada dimensió del *mesh* té $\sqrt{P}$ nodes, en què P és el nombre de nodes. El diàmetre d'una xarxa d'aquest tipus creix a raó del nombre de nodes per dimensió, en ser la distància més llarga entre nodes que la que hi ha entre nodes col·locats en cantonades oposades. Aquests sistemes són molt usats, ja que són fàcilment escalables, a més que les aplicacions amb càlculs d'una estructura regular són fàcilment mapades en aquestes xarxes d'interconnexió.

Una variant del *mesh* és afegir connexions entre els nodes dels extrems en cada dimensió. Aquesta variant, *double torus*, és més tolerant a errors i redueix el diàmetre, ja que les cantonades estan connectades, i per tant ja no són els nodes més allunyats, tal com passava en el cas del *ring* en comparació del *linear array*. La figura 48 mostra els tipus de xarxa d'interconnexió *mesh*, *ring* i *double torus*.

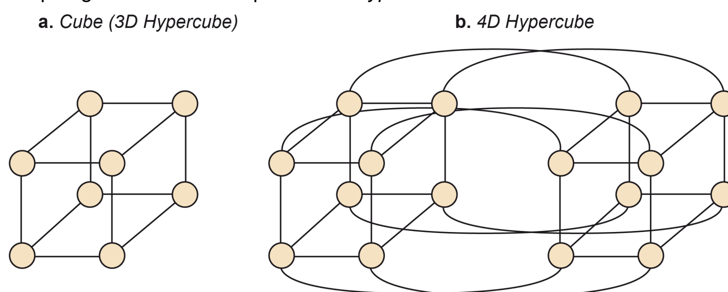
Figura 48. Topologies de xarxa de tipus *mesh*, *ring* i *double torus*



Una classe especial de *mesh* és la xarxa d'interconnexió *cube*. Aquesta xarxa d'interconnexió és una *mesh* de 3 dimensions. En general és una *3D mesh* amb k nodes per dimensió. En la figura 49.a se'n mostra una amb 2 nodes per dimensió. Dos cubs replicats, i connectats en cadascun dels seus nodes, formen el que seria un *four-dimensional cube*, i així successivament podríem formar *five-dimensional cube*, etc. En general, aquest tipus de cubs es diuen *hypercube*.

Els *hypercubes* són una bona elecció per a sistemes d'alt rendiment, ja que el diàmetre creix linealment amb el nombre de dimensions (log_2(el nombre de nodes)). És a dir, en un 8-dimensional *hypercube*, amb 2^8 processadors, només necessitaríem 8 passos, com a màxim, per a anar d'un processador a un altre. Aquest diàmetre és la meitat que una *2D mesh* de 16 per 16, que té el mateix nombre de nodes que l'*hypercube*, però que té un diàmetre de 16. El desavantatge d'aquests *hypercubes* és el seu cost econòmic, ja que el *fanout* necessari és molt elevat. La figura 49.b mostra un exemple de xarxa d'interconnexió de tipus *hypercube*.

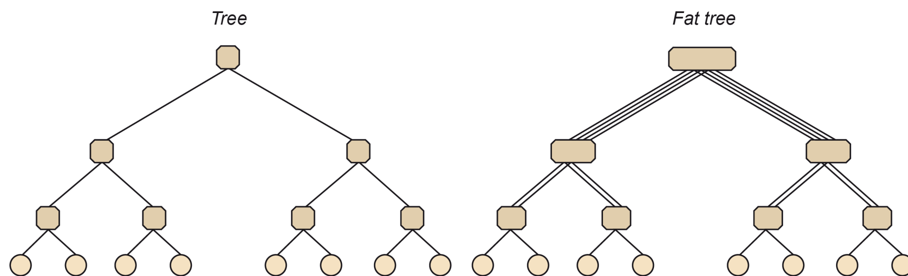
Figura 49. Topologies de xarxa de tipus *cube* i *hypercube*



Tree interconnect

La xarxa d'interconnexió *tree interconnect* té una *bisection bandwidth* d'1, que es correspon amb l'únic *switch* que separa un subarbre de l'altre, el *switch root* de l'arbre. Per tant, el node arrel de l'arbre es pot convertir en un coll d'ampolla en les comunicacions. Una manera de solucionar-ho és per mitjà d'ampliar el nombre de *links* a mesura que ens apropem al node arrel, tenint tants *links* com fulles en aquest subarbre. Això augmenta la *bisection bandwidth* fins a $n/2$, i es redueix així el coll d'ampolla que tenia el *switch root*. Aquest tipus de topologia de xarxa, en què s'augmenta el nombre de *links* a mesura que ens apropem al node arrel, es diu *fat tree*. La figura 50 mostra un exemple de topologia en arbre, i *fat tree*.

Figura 50. Topologies de xarxa de tipus *tree* i *fat tree*



Diàmetre i *bisection bandwidth* d'algunes topologies de xarxa

La taula 5 mostra el diàmetre i la *bisection bandwidth* d'algunes de les xarxes directes (part superior de la taula) i indirectes (part inferior), expressades en nombre de *links* que hem d'eliminar.

Taula 5. Relació de topologies de xarxa amb els seus *diameter* i *bisection bandwidth* expressat en nombre de *links*.

Topologia	Diàmetre	***Bisection bandwidth*** **(nre. (*#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	$\log P$	$\frac{P}{2}$

Primer comentarem les directes. El diàmetre de la *linear array* o *1D line* és $P-1$, que consisteix a arribar d'un extrem a l'altre de la xarxa. En canvi, si tenim un *1D ring*, en tenir una connexió entre els dos extrems, el camí més llarg és justament al node central, i aquest està a una distància de $\frac{P}{2}$. La *bisection bandwidth* d'aquestes dues xarxes és 1 i 2, respectivament, que són els *links* que hem d'eliminar per a desconnectar una meitat i l'altra.

En el cas de la xarxa *2D mesh* ja vam comentar el diàmetre en el seu moment, i és resultat d'anar d'una cantonada del *2D mesh* a la cantonada oposada. Per tant, hem de recórrer $\sqrt{P}-1$ en una dimensió i el mateix en l'altra, suposant que tenim P nodes. En el cas del *2D torus*, igual que passava en passar de *linear array* a *1D ring*, els extrems en cada dimensió estan connectats. Això significa que d'una cantonada a la seva oposada només necessitem dos passos. Per tant, els nodes a més distància són aquells que estan en una cantonada i el node que es troba enmig del *2D torus*. Això significa que s'ha de recórrer $\lfloor\sqrt{P}/2\rfloor$ nodes en una dimensió i el mateix en l'altra,

fins a arribar al node central. Per a calcular la *bisection bandwidth* en el cas de *2D mesh* s'ha de tallar per la meitat el *mesh*. Això significa desconnectar tants *links* com nodes en una dimensió, és a dir, $\sqrt{P}$. En el cas del *torus* també s'han d'eliminar els *links* que connecten els extrems. És a dir, la *bisection bandwidth* és el doble del *mesh*.

Finalment, en el cas de les xarxes d'interconnexió directes, per a calcular el diàmetre de la xarxa *D-hypercube* seguirem el mateix raonament que per a un *2D mesh*. Hem d'anar d'un extrem a un altre en cada dimensió. En el cas d'un *hypercube* com el de la figura, això significa haver de recórrer 1 node en cada dimensió (hi ha 2 nodes per dimensió). Així, per a les d dimensions, haurem de recórrer d nodes. Quant al càlcul de la *bisection bandwidth*, el podem veure com la interconnexió de dues *(*$d-1$*) hypercube*, tal com es mostra en la figura 49. Per tant, s'han d'eliminar tants nodes com els que tingui un d'aquests $d-1$ *hypercube*, que són 2^{d-1}, és a dir $\frac{P}{2}$.

Quant a les xarxes d'interconnexió indirectes, la xarxa *tree* té un diàmetre que correspon a la distància existent entre un node d'un subarbre del *tree* a l'altre subarbre; és a dir, pujar $\log P$ nivells i baixar el mateix. Per al cas del *fat tree* tenim el mateix camí, i per tant, el mateix diàmetre. D'altra banda, la *bisection bandwidth* per al *tree* és només 1, ja que només necessitem tallar el *link* de l'arrel de l'arbre. En canvi, per al *fat tree*, en augmentar el nombre de *links*, la *bisection bandwidth* s'augmenta fins a $\frac{P}{2}$.

Finalment, el diàmetre de la xarxa d'interconnexió *Omega* és igual al nombre d'etapes que té la xarxa, és a dir, $\log_2 P$. Quant a la *bisection bandwidth*, és la meitat del nombre de *links* que van de la part superior a la inferior. És la meitat a causa que només la meitat dels *links* poden estar actius alhora, per temes de conflictes en el camí que segueixen els missatges o accessos.

3.2. Comunicacions

En els sistemes multicomputador, tal com hem pogut observar, necessitem fer comunicacions entre processos.

El temps de comunicació del missatge de m elements d'un processador emissor a un processador destinació (comunicació punt a punt), sense comptar els costos d'inicialització del missatge en el processador emissor, i els de recepció en el processador receptor, és:

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

en què t_s és el temps d'inicialització del missatge i t_w és el temps de transmissió per paraula enviada (o element, per simplificar-ho). Aquest model bàsic és una simplificació d'un altre model de comunicació que considera altres factors de la xarxa. Els factors de la xarxa que normalment es distingeixen en models de comunicació més complets són:

Nota

En aquest mòdul treballarem amb un model simplificat de comunicació punt a punt:

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

En aquest model menyspreem el t_h i considerem un encaminament *cut-through* del missatge.

1) t_s o temps d'inicialització* (t_s) consisteix en el temps de preparar el missatge per a enviar-lo a la xarxa, determinar el camí del missatge a través de la xarxa, i el cost de la comunicació entre el node local i el *router*.

* ***Start up*, en anglès**

2) Temps de transmissió del missatge, que al seu torn consta del següent:

- per *byte* (t_w): temps de transmissió d'un element (*word*).

- per *hop* (t_h): temps necessari perquè la capçalera del missatge es transmeti entre dos nodes directament connectats a la xarxa, però que no considerarem en el nostre model simplificat, ja que t_h no és significatiu per a missatges petits ni per a missatges grans en comparació de t_s i t_w, respectivament.

A més, estem suposant que tenim un tipus de xarxa d'interconnexió que usa un encaminament *cut-through* del missatge, en què totes les parts d'un missatge tenen el mateix encaminament i informació d'error.

D'altra banda, les aplicacions solen utilitzar comunicacions col·lectives. Aquestes són comunicacions entre més d'un processador amb uns determinats patrons de comunicació. La implementació eficient d'aquestes comunicacions col·lectives en diferents arquitectures és important per a obtenir aplicacions paral·leles eficients. A continuació detallem alguns dels algorismes de comunicació que se segueixen per a implementar de manera eficient aquestes comunicacions col·lectives, i el cost de comunicació de cadascuna, basant-nos en el model bàsic de comunicació punt a punt.

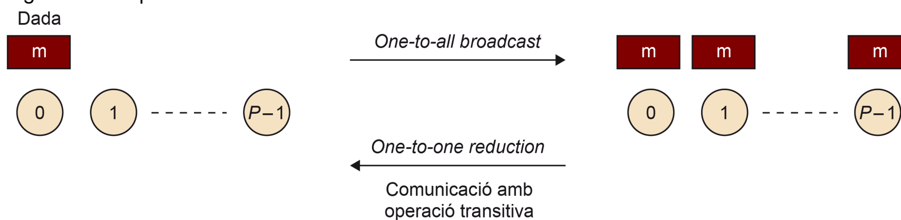
3.2.1. Cost de les col·lectives

Ara analitzarem els costos de les comunicacions col·lectives usades més freqüentment (*one-to-all [broadcast], all-to-one [reduction], scatter, gather*) en tres xarxes d'interconnexió: *linear array*, *2D mesh* i *hypercube*. En l'elaboració d'aquesta anàlisi anem a suposar, igual que per a la comunicació punt a punt, que el nombre de *links* no afecta el cost d'un missatge, a més de suposar que no tenim contenció en cap dels *links*. En un altre cas, el cost de comunicació de la col·lectiva podria ser més gran. També suposarem que els *links* entre els nodes són bidireccionals, és a dir, dos nodes directament connectats poden enviar un missatge a través d'aquest *link* en paral·lel. No obstant això, un node no pot rebre dos missatges pel mateix *link*.

***One-to-all* (*broadcast*) i *all-to-one* (*reduction*)**

La figura 51 mostra un esquema bàsic de les dues col·lectives, de les quals una és la dual de l'altra.

Figura 51. Esquema bàsic de les comunicacions col·lectives *one-to-all* i *all-to-one*



La col·lectiva de comunicació *one-to-all broadcast*, tal com indica el seu nom, fa la comunicació d'una dada de mida m a tots els processos involucrats en la col·lectiva, des d'un únic procés font. D'aquesta manera, al final de la comunicació hi haurà P còpies de la dada m, si P és el nombre de processos en la comunicació.

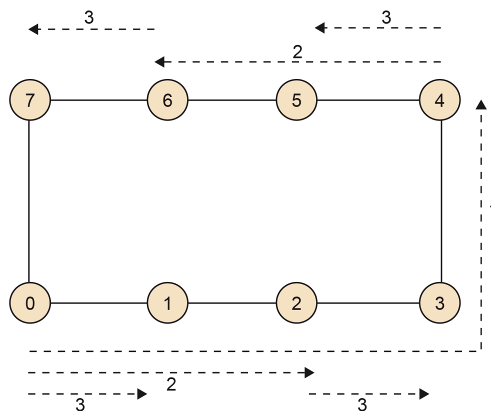
La col·lectiva de comunicació *all-to-one reduction* és la inversa del *one-to-all broadcast*. En aquest cas, la col·lectiva implica també una operació associativa de les dades per comunicar. Per exemple, una possible reducció podria ser la suma de les dades de tots els processos en la comunicació. Cada procés faria la suma local i tots cridarien la col·lectiva per a poder fer la suma dels resultats de la suma local a cada procés, i deixar el resultat d'aquesta suma en un únic procés.

1) Cost en un *linear array/ring*

La solució més senzilla d'enviar un missatge a tots és enviant $P - 1$ missatges des del processador font als altres $P - 1$ processadors. No obstant això, aquesta no és una solució eficient si el nombre de processadors és molt elevat. Una solució més eficient és la d'intentar aconseguir que, a cada pas de comunicació, es vagin doblant els processadors que reben les dades en el cas de *one-to-all broadcast* i que es divideixin per la meitat en el *all-to-one reduction*. Aquesta estratègia es diu *recursive doubling*.

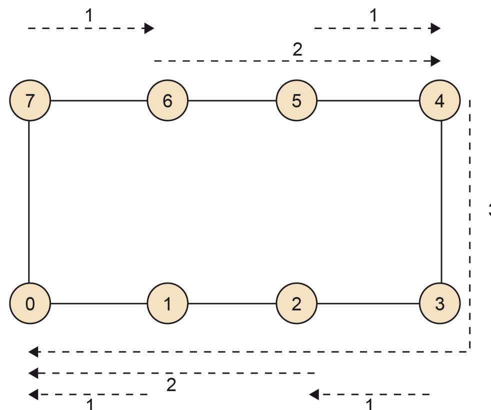
La figura 52 mostra el detall de com es farien les comunicacions en una xarxa d'interconnexió *ring* de 8 nodes. En la figura es mostren els missatges amb fletxes. I per a cada fletxa s'indica en quin pas de l'algorisme es fa aquesta comunicació. Així, per a la comunicació col·lectiva *broadcast* observem que en el pas 1 la comunicació va del processador 0 al 4, en el pas següent són el 0 i el 4 els que envien al 2 i al 6, i finalment, en l'últim pas (el pas $\log 8$), els processadors faran una comunicació als seus veïns respectius, i completaran així el *broadcast*.

Figura 52. Comunicacions en un *ring* per a la realització d'un *one-to-all*



En la figura 53 mostrem l'exemple de fer un *all-to-one reduction*. En aquest cas els nodes imparells comencen fent una comunicació al seu veí, després hi ha una comunicació cada dos, i finalment una comunicació del node 4 al node 0, i s'acaba així la reducció. Com es pot observar, són exactament els passos inversos d'una comunicació *one-to-all broadcast*.

Figura 53. Comunicacions en un *ring* per a la realització d'un *all-to-one*



Per tant, el cost de comunicació del *one-to-all broadcast* i de l'*all-to-one reduction* és de:

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

en què P és el nombre de processadors, i m la mida de la dada per fer el *broadcast* o bé la mida de la dada que té cada processador a l'inici de la comunicació col·lectiva *all-to-one reduction*.

2) Cost en un *mesh*

Les dues comunicacions col·lectives en una xarxa d'interconnexió *mesh* es poden fer en dos passos:

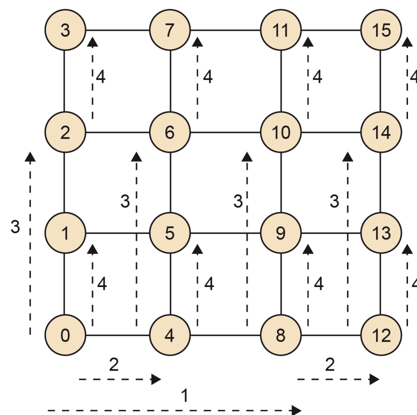
1) fer la comunicació col·lectiva en una de les files de la xarxa d'interconnexió *mesh*.

2) Aplicar la comunicació col·lectiva a cada columna, una vegada feta l'anterior col·lectiva en una de les files.

És a dir, en dos passos de *one-to-all broadcast* o bé d'*all-to-one reduction*, completaríem la comunicació col·lectiva.

La figura 54 mostra els missatges que es fan per un *one-to-all broadcast* en una *mesh* de 4×4 nodes, i s'hi indica en quin pas es fa cadascun. Primer es fa el *one-to-all broadcast* en la fila del *mesh* amb nodes 0, 4, 8 i 12. Posteriorment s'aplica a cada columna en paral·lel.

Figura 54. Comunicacions en un *mesh* per a la realització d'un *one-to-all*



El cost de comunicació és dues vegades el cost en una xarxa d'interconnexió *ring* de $\sqrt{P}$ processadors (el nombre de processadors en cada dimensió en una xarxa de tipus *mesh*):

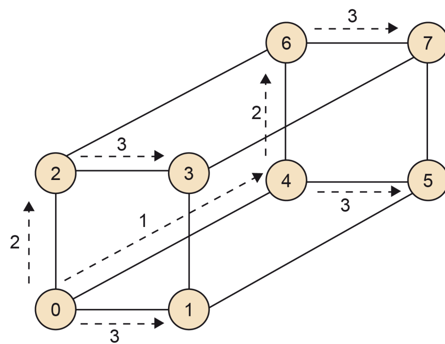
$$T_{comm} = 2 \times \log \sqrt{P} \times (t_s + m \times t_w)$$

en què P és el nombre de processadors, i m la mida de la dada per fer el *broadcast* o bé la mida de la dada que té cada processador a l'inici de la comunicació col·lectiva *all-to-one reduction*.

3) Cost en un *hypercube*

El cas de l'*hypercube* és un cas especial de *mesh* amb 2^d nodes, en què d és el nombre de dimensions de la *mesh*, i el 2 ve del fet de tenir dos nodes per dimensió. Per tant, podem aplicar el mateix algorisme que hem aplicat en la *mesh*: primer una dimensió, després en paral·lel tots els nodes en la dimensió següent, després la següent, etc., fins a arribar a fer les d dimensions. La figura 55 mostra els missatges en una comunicació col·lectiva *one-to-all broadcast*. L'etiqueta de cada comunicació indica en quin moment es fa cada comunicació punt a punt.

Figura 55. Comunicacions en un *3D hypercube (cube)* per a la realització d'un *one-to-all*



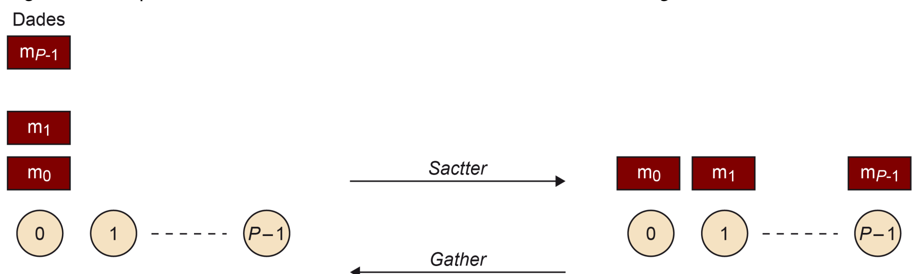
El cost de comunicació és d vegades el cost en una xarxa d'interconnexió *ring* de 2 processadors:

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

Scatter i *gather*

En una comunicació col·lectiva *scatter* un node distribueix equitativament un *buffer* de mida $P \times m$ entre tots els processadors que fan la comunicació col·lectiva. És a dir, cada processador i rebrà m elements consecutius del *buffer*, començant en la posició $i \times m$ del *buffer*. La figura 56 mostra un esquema bàsic d'aquesta comunicació. La comunicació col·lectiva *gather* és la inversa de l'operació *scatter*.

Figura 56. Esquema bàsic de les comunicacions col·lectives *scatter* i *gather*



Els passos de comunicació en l'operació *scatter* són iguals que en el *broadcast*, però en el cas d'aquest últim, la mida del missatge es manté constant (m), mentre que per a l'*scatter* es va reduint.

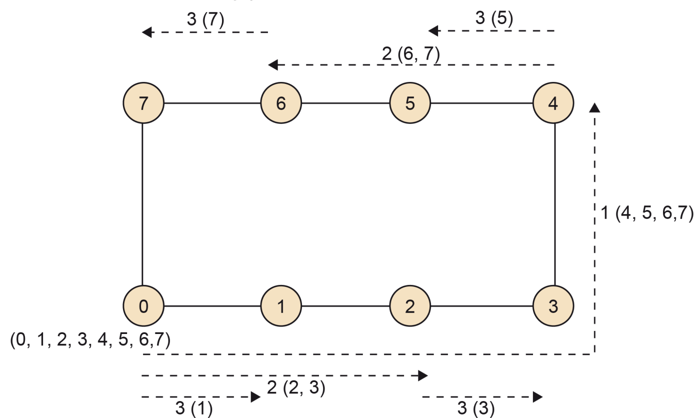
El *gather* és una operació equivalent a la *reduction*, però també varia la mida del missatge. En el cas de la comunicació *gather*, el missatge va augmentant a mesura que anem fent passos de comunicació.

1) Cost en un *linear array/ring*

Igual que per al cas del *one-to-all*, la solució més senzilla seria la d'enviar la part del *buffer* que correspon a cada processador, de mida m, amb un missatge *point-to-point*. Però aquesta solució no és eficient per a un nombre elevat de processadors. Una solució més eficient és seguir la mateixa estratègia que seguim per al *one-to-all broadcast*: intentar aconseguir que, a cada pas de comunicació, es vagin doblant els processadors que reben les dades. Ara, no obstant això, el missatge inicial és de mida $\frac{P}{2} \times m$, i a cada pas de comunicació es reduirà per la meitat.

La figura 57 mostra al detall com anirien les comunicacions en una xarxa d'interconnexió *ring* de 8 nodes. En la figura es mostren els missatges amb fletxes. I per a cada fletxa s'indica en quin pas de l'algorisme es fa aquesta comunicació, i les dades que es comuniquen. En l'exemple de la figura, el processador 0 ha de fer l'*scatter* de 8 dades, que es mostren en la figura al costat del processador 0. En el primer pas de comunicació el processador enviarà un missatge de mida $\frac{P}{2} \times m$ al processador 4. En el pas següent, els nodes 0 i 4 enviaran la meitat de les dades $\frac{P}{2^2} \times m$, i així successivament fins a arribar al pas $\log P$, en què el missatge serà de mida $\frac{P}{2^{\log P}} \times m$, és a dir, m.

Figura 57. Comunicacions en un *ring* per a la realització d'un *scatter*



En el cas de la comunicació *gather* tindríem el procés invers en les comunicacions.

Per tant, el cost de comunicació de *scatter/gather* és de:

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

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

en què P és el nombre de processadors.

2) Cost en un *mesh*

Per a les xarxes d'interconnexió *mesh* podem seguir la mateixa estratègia que vam seguir per a les operacions de *broadcast* i *reduction*: aplicar primer l'*scatter/gather* en una fila i després en totes les columnes.

El cost de comunicació de totes dues operacions *scatter/gather* és dues vegades el cost en una xarxa d'interconnexió *ring* de $\sqrt{P}$ processadors:

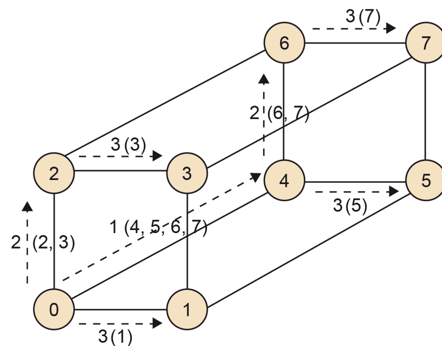
$$T_{comm} = 2 \times (t_s \log \sqrt{P} + t_w m(\sqrt{P} - 1))$$

en què P és el nombre de processadors. Per a l'operació *scatter* m és la mida de la part que s'ha d'enviar a cada processador del total de dades comunicades $P \times m$. I per a l'operació *gather* és la mida de la dada que té cada processador a l'inici de la col·lectiva, per a acabar deixant un missatge de mida $P \times m$ en el processador destinació del *gather*.

3) Cost en un *hypercube*

La figura 58 mostra la realització d'una operació *scatter* en un *3D hypercube* de 8 nodes, i enviant un únic element a cada processador. El cas del *D hypercube* és equivalent a la xarxa d'interconnexió *mesh*, en ser aquesta xarxa d'interconnexió una *mesh* de dimensió d i dos nodes per dimensió. Com es pot observar en la figura, en cada pas s'envia la meitat de les dades al node veí que està en la seva mateixa dimensió. Així, a cada pas, la mida del missatge que envia cada node es divideix per dos, fins a aconseguir que cada processador tingui un missatge de mida m, que es correspon amb 1 element en el cas de la figura.

Figura 58. Comunicacions en un *hypercube* per a la realització d'un *scatter*



L'operació *gather* és l'operació inversa equivalent, que es començaria amb els processadors amb un missatge de mida m en cada processador i, seguint el camí invers de la figura, s'aniria doblant la mida del missatge a cada pas de comunicació.

El cost de comunicació de l'*scatter/gather* és d vegades el cost en una xarxa d'interconnexió *ring* de 2 processadors per dimensió:

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

Resum

En aquest mòdul hem descrit la classificació segons Flynn, i l'hem ampliat amb les subcategories que A. Tanenbaum va descriure en el seu llibre sobre estructures de computadors, i posteriorment ens hem centrat en les dues subcategories principals de les màquines MIMD: multiprocessadors o màquines de memòria compartida (fortament acoblats) i multicomputadors o de memòria distribuïda (feblement acoblats).

Per als multiprocessadors hem descrit, principalment, com es connecten amb la memòria (bus, *crossbar*, *multistage*, etc.), els problemes de consistència de memòria que es plantegen quan hi ha diversos processadors llegint i escrivint sobre una mateixa memòria, i el problema de la coherència de memòria cau a causa de l'existència de diverses còpies d'una mateixa dada en el sistema. En el cas del problema de consistència de memòria, s'han analitzat diferents models de consistència que fixen les regles de l'ordre de les lectures i escriptures que haurien de veure tots els processadors. Per al cas de la coherència de memòria cau, s'ha descrit detalladament el funcionament de dos protocols per a mantenir la coherència de memòria cau: MSI i MESI, i els mecanismes de maquinari dels quals es disposa per a aconseguir aquesta coherència.

Quant als multicomputadors, aquests no tenen el problema de consistència ni de coherència de memòria cau, ja que la compartició de les dades es fa mitjançant el pas de missatges. Aquests missatges tenen un cost que nosaltres hem modelitzat de manera senzilla, i que ens han ajudat a detallar el cost de les comunicacions en xarxes d'interconnexió amb diferents topologies de xarxa, que també hem analitzat en aquest mòdul. En particular, hem diferenciat entre xarxes d'interconnexió directes i indirectes, i hem descrit les diferents maneres de mesurar el rendiment d'aquestes xarxes, i com poden afectar el rendiment de les nostres aplicacions.

Bibliografia

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

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

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