

PAR



Roger Bitlloch • Parallelisme • 2025-26 QP

Teoria Post-parcial

T4. Introducció a les arquitectures Paralleles

T5. Data Decomposition

Basat en les diapositives de la FIB

Índex

Avaluació Paral·lelisme	4
Calendari del curs	4
T4. Introducció al Hardware del Paral·lelisme	5
4.1 El problema: coherència de caché	5
4.2 <i>True sharing</i> vs <i>False sharing</i>	5
4.3 Polítiques d'escriptura i mecanismes de coherència	7
4.4 UMA/SMP amb snooping	8
4.4.1 Protocol MSI: estats d'una línia de caché	8
4.4.2 Esdeveniments i transicions d'estat	8
4.4.3 Optimitzacions del MSI: MESI, MOSI, MSIF	10
4.4.4 Cost de la coherència: bits d'estat per línia	11
4.5 NUMA amb directoris	12
4.5.1 Limitacions del snooping en NUMA	12
4.5.2 Protocol MSU - Coherència basada en directori	12
4.5.3 Bits de presència (Sharers List)	13
4.5.4 Rols dels nodes	13
4.5.5 Comandes del protocol de directori (simplificat)	13
4.5.6 Cost de la coherència: bits del directori	15
4.6 Implicació pràctica: la política <i>first touch</i>	16
4.6.1 Snooping i directori alhora	17
4.7 Resum: tipus d'arquitectures multiprocessador	17
4.8 Sincronització de baix nivell	18
4.8.1 Implementació de Locks: <i>Test-and-Set</i>	18
4.8.2 Optimització: <i>Test-and-Test-and-Set</i>	18
4.8.3 Load-Linked / Store-Conditional (LL/SC)	18
4.8.4 Barreres de sincronització	18
4.9 Consolidació T4 — El Hardware del paral·lelisme	19
T5. Descomposició de dades (<i>Data Decomposition</i>)	20
5.1 Objectius de la descomposició de dades	20
5.2 Estratègies geomètriques de descomposició	21
5.2.1 Block	21
5.2.2 Block amb el residu repartit:	22
5.2.3 Cyclic	23
5.2.4 Block-Cyclic	24
5.2.5 Propietats de l'estratègia Block-Cyclic:	24
5.2.6 Block-Cyclic alineat a línia de caché	25
5.2.7 Comparativa de les estratègies	25
5.3 Padding per solucionar false sharing	26
5.4 Descomposicions 2D (matrius)	27
5.5 Padding en matrius	28
5.6 Consolidació T5 — Data decomposition	29

Avaluació Paral·lelisme

- **Teoria:** 65% – Parcial (P), Final Part 1 (FP1) i Final Part 2 (FP2)
 - **Part 1** (T1, T2 i T3): 32,5% – $\max(P, FP1)$ si $P \geq 5$, sinó $FP1$
 1. Task Dependence
 - (a) Dibuixar TDG
 - (b) Calcular T_1 , T_∞ i $P_{\min}$
 - (c) Calcular T_P + diagrama amb datasharing
 2. Paral·lelitzar bucle amb tasques explícites o implícites
 3. Paral·lelitzar programa recursiu amb cutoff
 - **Part 2** (T4 i T5): 32,5% – FP2
 4. Numa Nodes (nombre de bits / taula) i Aplicar Padding
 5. Paral·lelitzar programa Geometric Data Decomposition
- **Laboratori:** 35% = $\max(SL, FL)$ si $FL > 4$, sinó $0.35 \cdot SL + 0.65 \cdot FL$.
 - **Informes Lab** (SL): OBLIGATORIS ($0.3 \cdot Lab3 + 0.5 \cdot Lab4 + 0.2 \cdot Lab5$)
 - **Final Lab** (FL): Mateix dia del final (pots portar chuletari fet a mà)

Si aprofueu l'assignatura i FP2, us puguen fins a un 10% de la nota fent qüestionaris d'Atenea!!!

Calendari del curs

Dilluns 16h15 - 17h45		Contingut
S7	20 Abril	LAB 4
S8	27 Abril	T4.1 Arquitectura Paralela
S9	4 Maig	T4.2 Arquitectura Paralela
S10	11 Maig	T5 Data decomposition
S11	18 Maig	LAB 5
	25 Maig	NO HI HA CLASSE - Festiu 2a Pasqua
IP1-1		T1 i T2 + Exercicis TDG
IP1-2		T3.1 (Iterative i Reductions)
IP1-3		T3.2 (Recursive Leaf i Tree)
IP2-1		T4 Numa Nodes
IP2-2		T5 Geometric Data Decomposition
IP2-3		LAB
IP2-4		Examen Pràctica 2024-25Q1
IP2-5		Examen Pràctica 2024-25Q2
16 Juny: Final + Exàmen LAB		

4 Introducció al Hardware del Parallelisme

Quan parlem de parallelisme a nivell de hardware, el primer que cal entendre és com estan organitzats els processadors i la memòria, perquè això determina com es comparteixen les dades i quin cost té accedir-hi. Els nostres objectius seran descobrir:

- Com es comuniquen els threads quan comparteixen memòria o necessiten una mateixa dada.
- Quin *overhead* introdueix el fet que cada core tingui cachés pròpies (coherència).

4.1 El problema: coherència de caché

En arquitectures de **memòria compartida** (tant UMA com NUMA), tots els cores veuen el mateix espai d'adreces i es comuniquen simplement llegint i escrivint dades compartides amb instruccions **load/store**.

Per qüestions de rendiment, **cada core té la seva pròpia caché**. Això vol dir que una mateixa dada (o, més exactament, una mateixa **línia de caché**) pot estar **copiada simultàniament** a diverses cachés.

El problema apareix quan dues (o més) cachés tenen una còpia de la **mateixa línia** i **un core hi escriu**: les altres còpies poden quedar **obsoletes** (amb un valor antic). Per evitar que un core llegeixi dades “velles” mentre un altre ja ha escrit el valor nou, el sistema ha de garantir **coherència**: un conjunt de regles perquè totes les còpies mantinguin una visió consistent del valor.

Exemple d'incoherència:

Pas	Acció	Conseqüència	Cache A	Cache B	Mem[X]
0	(estat inicial)	Cap core té còpia de X	—	—	0
1	Core A: load x	Miss → A carrega la línia de memòria	0	—	0
2	Core B: load x	Miss → B carrega la línia de memòria	0	0	0
3	Core A: x = 15	A escriu a la seva caché local	15	0	0
4	Core B: load x	Hit! → B llegeix la seva còpia obsoleta	15	0	0

Al pas 4, B llegeix x de la seva caché i obté **0** en lloc de **15**: **incoherència**. B no sap que A ha modificat la línia.

Abans que B pugui llegir una còpia obsoleta (pas 4), hem d'idear un mecanisme perquè quan A modifiqui la línia **invalidi o actualitzi** la còpia de B. D'aquesta manera podem garantir que tots els cores sempre veuen el valor més recent. Necessitem un **protocol de coherència**!

4.2 True sharing vs False sharing

La coherència **NO** es gestiona a nivell de variable individual, sinó a nivell de **línia de caché (cache line)**. És a dir, el hardware tracta tota la línia com a unitat: si s'ha de fer invalidació o transferència, **afecta tota la línia**, no només el camp o element que ha canviat.

Si el hardware hagués de rastrejar l'estat de **cada byte** de memòria, el nombre de bits de control seria enorme: caldria mantenir informació d'estat i de compartició per cadascuna de les milions de posicions que poden estar a les cachés.

Treballar amb una granularitat de línia de caché simplifica enormement el hardware, però té una conseqüència directa: pot generar tràfic de coherència **innecessari** quan threads diferents modifiquen variables que, tot i ser independents, comparteixen línia.

Això dona lloc a dos tipus de tràfic de coherència amb causes molt diferents:

True sharing (compartició real)

Parlem de **true sharing** quan diversos threads accedeixen a **la mateixa adreça** (la mateixa variable o element) i, per tant, **realment necessiten compartir la mateixa dada**. En aquest cas, és normal i inevitable que hi hagi comunicació de coherència entre processadors, perquè tots han d'estar “al dia” del valor correcte.

```
// Thread A i Thread B escriuen AMBDOS sobre result -> true sharing
#pragma omp atomic
result += parcial; // coherencia inevitable (mateixa variable)
```

Tot i que és inevitable, el **tràfic de coherència** es pot **minimitzar** mitjançant una reducció: cada thread acumula en una variable **local** i fa un sol **atomic** al final, en lloc d'un per iteració.

False sharing (compartició falsa)

El **false sharing** passa quan diferents threads treballen amb **variables diferents**, però aquestes variables estan ubicades tan a prop a memòria que acaben caient dins la **mateixa línia de caché**. Com que la coherència es manté per línia i no per variable, una escriptura sobre un camp pot provocar invalidacions o transferències sobre **tota la línia**, afectant l'altre thread **encara que no hi hagi cap conflicte real de dades**.

```
int v[100]; // elements contigus en memoria
Thread A: v[0]++; // (escriu v[0])
Thread B: v[1]++; // (escriu v[1])
// Tot i ser VARIABLES DIFERENTS, escriuen a la MATEIXA LINIA!
```

Si $v[0]$ i $v[1]$ cauen dins la mateixa línia de caché (p. ex. línia de 16 bytes, `int` de 4 bytes):

False sharing pas a pas — $v[0]$ i $v[1]$ a la mateixa línia de caché

Pas	Acció	Conseqüència	Linea Cache A	Línia Cache B
0	(estat inicial)	Cap core té la línia	—	—
1	A: $v[0]++$	Miss → A carrega línia $[v0, v1, \dots]$	$[1, 0, \dots]$	—
2	B: $v[1]++$	Miss → B carrega línia; invalida A	inv.	$[1, 1, \dots]$
3	A: $v[0]++$	Miss! → A recarrega línia; invalida B	$[2, 1, \dots]$	inv.
4	B: $v[1]++$	Miss! → B recarrega línia; invalida A	inv.	$[2, 2, \dots]$

Cada escriptura provoca un **miss** i una **invalidació** a l'altra caché, tot i que A i B **mai** toquen la mateixa variable. El ping-pong d'invalidacions continua indefinidament, degradant el rendiment.

És a dir: no s'està compartint “la mateixa variable”, però el hardware ha d'actuar com si hi hagués compartició perquè la línia és comuna, i això introdueix **overheads** i pèrdua de rendiment.

Més endavant (secció 5.3) veurem com evitar-ho amb *padding* (afegir espai entre camps) i amb alineació/reorganització de dades perquè les variables modificades per threads diferents quedin en línies de caché separades.

4.3 Polítiques d'escriptura i mecanismes de coherència

Un cop identificat el problema, cal definir quina **política d'escriptura** aplica el hardware quan un processador escriu sobre una línia copiada en altres caches, i quin **mecanisme** utilitza per comunicar-ho.

Polítiques d'escriptura: *Write-Update* vs *Write-Invalidate*

La política estableix *què* cal fer amb les còpies d'una línia quan un core hi escriu: actualitzar-les o invalidar-les.

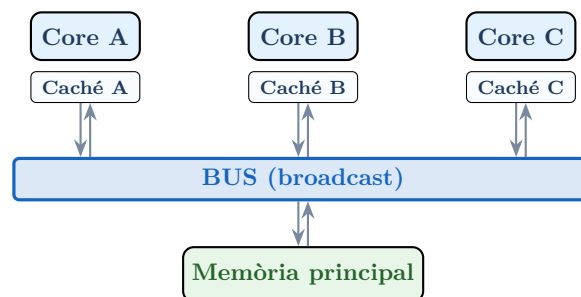
- **Write-Update**: quan un core escriu, fa *broadcast* del nou valor a totes les caches que tenen la línia, forçant una actualització immediata.
Evita *misses* de coherència, però genera molt trànsit si hi ha escriptures consecutives que ningú més llegeix.
- **Write-Invalidate** (estàndard): quan un core escriu, invalida totes les altres còpies; els altres cores hauran de tornar a demanar la línia quan la necessitin.
Redueix el trànsit (només la primera escriptura genera invalidacions) a costa de més *invalidation misses*.

Mecanismes de coherència: Snooping vs Directori

El mecanisme diu **com** es comuniquen les caches per dur a terme la política d'escriptura.

Broadcast-based (Snooping) — sistemes UMA/SMP

S'utilitza un bus de *broadcast* on totes les caches "escolten" (*snoop*) les transaccions i reaccionen. Funciona bé amb pocs cores, però **no escala**: el bus es converteix en un coll d'ampolla.

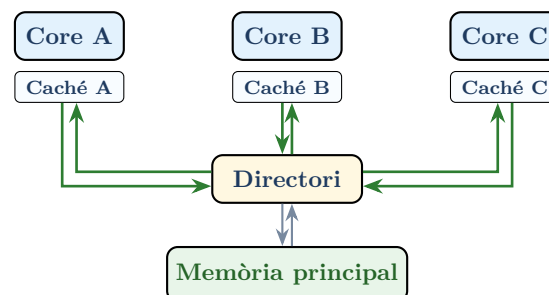


Totes les caches veuen totes les transaccions

Directory-based (Directori) — sistemes NUMA

Un directori centralitzat registra **qui té còpies de cada línia**, permetent enviar missatges **punt-a-punt** només als nodes implicats, eliminant el broadcast.

Escala molt millor per a sistemes grans.



Només els nodes implicats reben missatges

4.4 UMA/SMP amb snooping

En una UMA (*Uniform Memory Access*) o SMP (*Symmetric MultiProcessor*), tots els cores comparteixen **un únic espai d'adreces** i accedeixen a una memòria “central” amb **un temps d'accés uniforme** (idealment, la mateixa latència per a tots).

El mecanisme de snooping és el natural per a aquesta arquitectura:

- La coherència es manté a granularitat de **línia de caché**.
- Cada caché guarda localment l'estat de les seves línies (**estat distribuït**).
- El bus serveix com a medi de *broadcast* que fa visibles totes les transaccions i en defineix l'**ordre global**.
- Les caches monitoritzen (*snoop*) el bus i actuen sobre els esdeveniments rellevants mitjançant controladors *snoopy* (SCC: *Snoopy Cache Controllers*).

4.4.1 Protocol MSI: estats d'una línia de caché

El protocol **MSI** és el protocol de snooping de referència. Cada línia present en una caché porta associats uns **bits d'estat** que indiquen la seva situació respecte a la resta del sistema:

- **M (*Modified*)**: la caché té l'única còpia vàlida i aquesta és **diferent** del valor a memòria principal (*dirty*). Cap altra caché pot tenir la línia en M o S simultàniament.
- **S (*Shared*)**: la còpia és **idèntica** a memòria principal (*clean*). Pot existir en diverses caches alhora.
- **I (*Invalid*)**: la línia no és vàlida (o directament no és present a la caché). Qualsevol accés a aquesta línia provocarà un *miss*.

4.4.2 Esdeveniments i transicions d'estat

Aquests estats canvien en resposta a dos tipus d'esdeveniments. D'una banda, els **CPU events** són les operacions que el processador local vol fer sobre la línia. De l'altra, quan el controlador de caché necessita comunicar-se amb la resta del sistema, genera **accions al bus (bus events)** que totes les altres caches poden observar.

La clau per entendre el protocol és que els **CPU events** provoquen **bus events**, i els bus events observats per les altres caches (*snooped*) provoquen canvis d'estat en les seves còpies.

CPU events (accions del processador local):

- **PrRd** (*Processor Read*): el processador vol llegir la línia.
- **PrWr** (*Processor Write*): el processador vol escriure a la línia.

Bus events (accions generades pel controlador de caché):

- **BusRd**: demana una còpia de la línia **sense** intenció de modificar-la (lectura).
- **BusRdX**: demana una còpia de la línia **amb** intenció de modificar-la (escriptura). **Al fer-ho invalida totes les altres còpies!**
- **BusUpgr**: Només s'usa si ja es té la línia en S (hit), i la converteix a M. **No es transfereixen dades**, perquè el sol·licitant ja té la línia, però si que **s'invaliden les còpies**.
- **Flush**: posa la línia al bus per servir dades a altres caches, o per fer *writeback* a memòria quan s'expulsa una línia *dirty*.

Transicions d'estat

Amb els estats i els esdeveniments definits, podem descriure exactament com reacciona cada línia de caché davant de cada situació. Les transicions es divideixen en dos grups: les provocades per **accions del processador local** (PrRd, PrWr) i les provocades per **observar transaccions d'altres processadors al bus** (*snooped events*).

Des de l'estat **I** (*Invalid*):

- PrRd → genera BusRd → passa a **S**
- PrWr → genera BusRdX → passa a **M**

Des de l'estat **S** (*Shared*):

- PrRd → hit local, es queda a **S**
- PrWr → genera BusUpgr → passa a **M**
- Snoop BusRd → no cal fer res, es queda a **S**
- Snoop BusRdX o BusUpgr → invalida → passa a **I**

Des de l'estat **M** (*Modified*):

- PrRd → hit local, es queda a **M**
- PrWr → hit local, es queda a **M**
- Snoop BusRd → Flush (serveix dades) → passa a **S**
- Snoop BusRdX → Flush (serveix dades) → passa a **I**

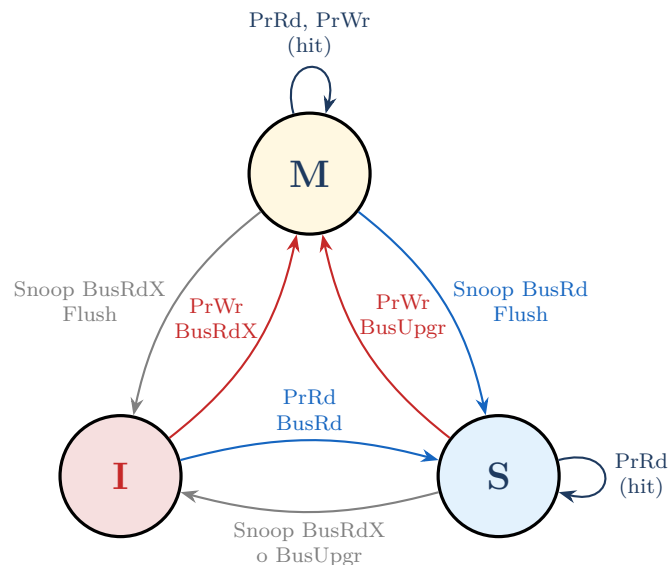


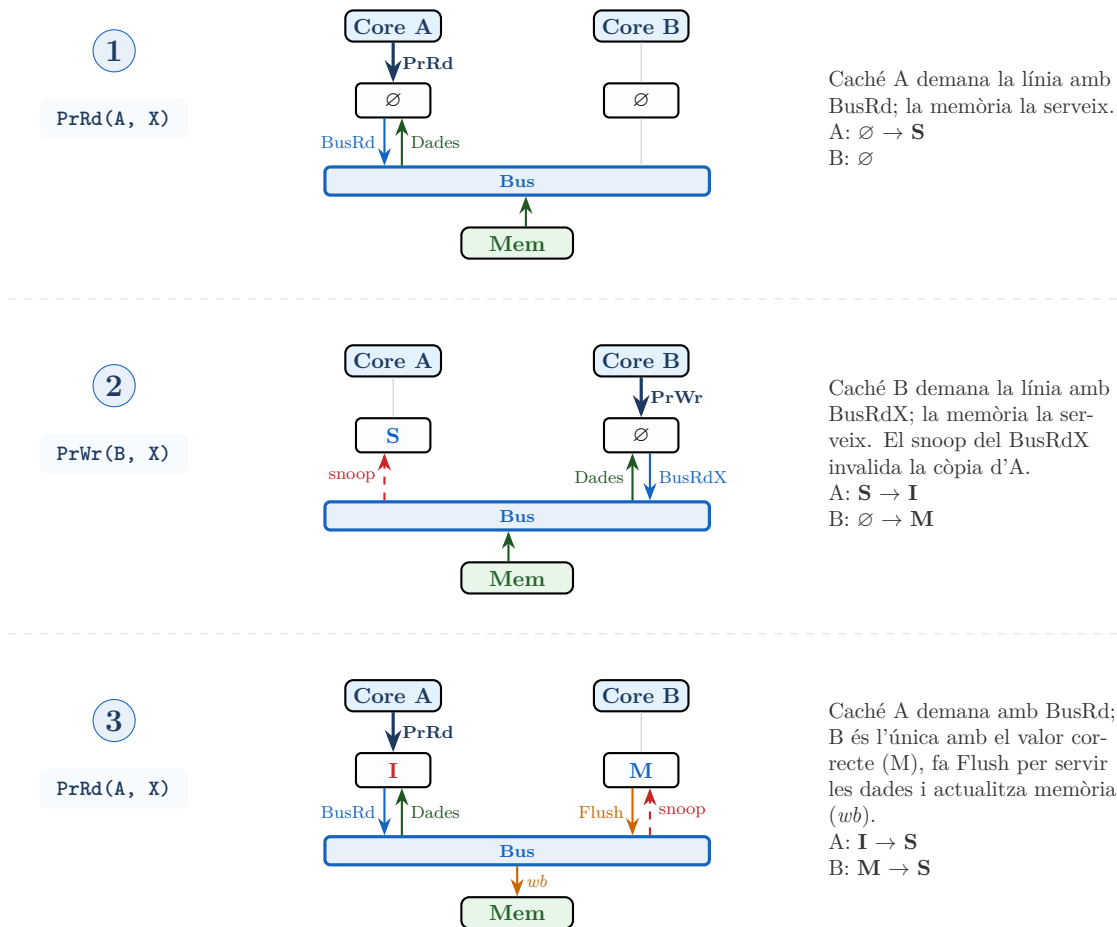
Diagrama de transicions del protocol MSI. Les fletxes blaves representen transicions de lectura, les vermelles d'escriptura, i les grises invalidacions provocades per altres cores. Sempre que és a un write es va cap a M, mentre que sempre que es fa un Rd es va cap a S M

Provisió de la línia

Un aspecte fonamental és **qui serveix les dades** quan una caché les demana. Si alguna altra caché té la línia en estat **M**, **ella** és l'única que té el valor correcte (la memòria principal està desactualitzada), de manera que ha de fer un **Flush** per posar la línia al bus.

En canvi, si cap caché té la línia en M (totes estan en S o totes en I), la **memòria principal** la proporciona directament.

L'exemple següent il·lustra com interactuen tots aquests elements en un escenari concret amb dos cores que accedeixen a la mateixa línia de caché:



Exemple MSI pas a pas sobre la línia X. Les caixes de caché mostren l'estat **inicial** de la línia (∅ = no present). Fletxes: ⇒ CPU event, → bus event, --> snoop, → dades, → flush/wb.

4.4.3 Optimitzacions del MSI: MESI, MOSI, MSIF

MSI és funcional però té casos comuns on fa massa trànsit. Per això apareixen extensions amb estats addicionals:

- **MESI (+E, Exclusive)**: una línia neta que **només existeix en una caché**. Com que ningú més la té, es pot passar de E → M **sense cap transacció de bus** (s'estalvia el BusUpgr). Si un altre core la demana, E passa a S.
- **MOSI (+O, Owned)**: una línia *dirty* que es pot compartir **sense actualitzar memòria**. La caché en estat O actua com a "propietària": serveix les dades quan altres les demanen. Memòria principal s'actualitza només quan la línia O s'expulsa.
- **MSIF (+F, Forward)**: quan hi ha diverses còpies netes (S), una es designa com a *forwarder* perquè sigui ella (i no la memòria) qui respongui peticions. Típicament, la darrera caché que va demanar la línia pren el rol F (explotant localitat temporal), mentre la resta queden en S.

Aquestes extensions es poden combinar: MESIF, MOESI, MOESIF, segons la implementació.

4.4.4 Cost de la coherència: bits d'estat per línia

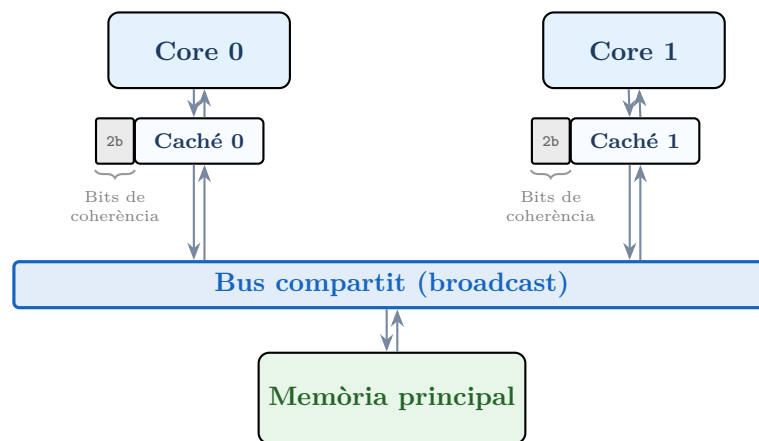
Mantenir la coherència no surt gratis: cada línia de caché necessita uns **bits d'estat addicionals** perquè el hardware pugui saber en quina situació es troba cada línia.

Aquests bits no contenen dades útils per al programa, però són imprescindibles perquè el protocol de coherència funcioni.

El nombre de bits que utilitzarem per línia depèn del protocol: amb **MSI** (3 estats) calen $\lceil \log_2 3 \rceil = 2$ bits per línia.

Amb MESI o MOSI passa igual. (4 estats, $\lceil \log_2 4 \rceil = 2$).

A partir de 5 estats (MOESI, MOESIF...) ja en calen 3.



En UMA amb snooping, la memòria no necessita bits de coherència: cada caché gestiona els seus propis bits (estat distribuït).

Sistema UMA: cada línia de caché porta 2-3 bits de coherència (en gris) que indiquen el seu estat (M/S/I o variants). El cost total és: $\frac{\text{mida caché}}{\text{mida línia}} \times \text{bits per línia}$.

El cost total de coherència d'una caché es calcula com:

$$\text{Nombre de línies} = \frac{\text{Mida de la caché}}{\text{Mida de la línia}}$$

$$\text{Bits de coherència} = \text{Nombre de línies} \times \text{num bits d'estat per línia}$$

Exemple de càlcul

Suposem una caché de **32 KB** amb línies de **64 Bytes** i protocol **MSI** (2 bits/línia):

$$\text{Nombre de línies} = \frac{32KB \times 1024}{64 \text{ Bytes/línia}} = 512 \text{ línies}$$

$$\text{Bits de coherència} = 512 \text{ línies} \times 2 \text{ bits/línia} = 1024 \text{ bits} = 128 \text{ bytes}$$

Això representa un *overhead* de $\frac{128}{32768} \approx 0.39\%$ respecte a la mida de la caché.

Un cost molt més petit per garantir la coherència, que no pas si dediquessim els 2 bits per cada element (cosa que implicaria un 25% d'espai addicional a la memòria caché per mantenir la coherència).

4.5 NUMA amb directoris

En una arquitectura **NUMA** (*Non-Uniform Memory Access*), també hi ha **un sol espai d'adreces compartit** i es programa igualment amb **load/store**, però la memòria física està repartida en diversos nodes. Cada node té CPUs/cachés i la seva memòria “local”.

- L'accés no és uniforme: llegir una dada de memòria local és **més ràpid** que llegir-la de memòria remota (d'un altre node).
- La **localitat** (on resideixen les dades i qui les usa) passa a ser **crítica per al rendiment**.

4.5.1 Limitacions del snooping en NUMA

Amb el mecanisme de snooping, el processador que inicia un accés envia un missatge a **tots** els altres processadors per saber l'estat d'una línia, independentment de si en tenen una còpia o no. Tot i que això pot ser viable en sistemes NUMA petits, en architectures més grans amb múltiples processadors i cachés, s'acaba generant un **tràfic de coherència excessiu** que satura el sistema.

Per garantir l'escalabilitat, s'utilitza la coherència basada en directoris (**directory-based**): en lloc de fer un *broadcast* massiu a totes les cachés, utilitzem un **directori** que registra exactament quins nodes tenen una còpia de cada línia, permetent una comunicació eficient i exclusivament punt a punt.

4.5.2 Protocol MSU - Coherència basada en directori

El protocol **MSU** s'utilitza en architectures NUMA per mantenir la coherència de les dades sense dependre del *snooping*. A diferència del MSI, on la responsabilitat de la coherència està distribuïda entre totes les cachés (que “escolten” el bus), en el MSU la responsabilitat recau en el **directori**.

Tot i que parlem de “directori” en singular, en realitat **no existeix un directori únic centralitzat**. El directori es divideix en porcions (*slices*), de manera que **cada node només emmagatzema l'estat de les línies de la seva pròpia memòria física** — és a dir, aquelles on ell actua com a *home node*. Per tant, el directori és una estructura **distribuïda** entre tots els nodes del sistema.

Els tres estats globals que pot tenir una línia al directori del seu *Home Node* són:

- **M (*Modified*)**: El directori indica que un únic node (l'*owner*) té l'exclusivitat sobre la dada. Aquest node en té una còpia modificada (*dirty*) i, per tant, la memòria principal està desactualitzada.
- **S (*Shared*)**: El directori indica que un o més nodes tenen còpies vàlides de només lectura. En aquest estat, la memòria principal està actualitzada.
- **U (*Uncached*)**: El directori indica que cap caché del sistema té la línia. La memòria principal és l'única que conté la dada vàlida.

Diferència clau: Uncached (U) vs. Invalid (I)

Mentre que l'estat **I** del protocol MSI és un estat local de la **caché** (indica que la meua còpia no és vàlida), l'estat **U** és un estat global del **directori**. Si una línia està en **U**, el *home node* té la certesa absoluta que no cal demanar la dada a cap altre node ni invalidar ningú; pot servir la dada directament des de la memòria principal.

4.5.3 Bits de presència (Sharers List)

En una arquitectura UMA, el *snooping* utilitza el *broadcast* perquè tothom vegi totes les peticions. En canvi, en NUMA necessitem els **bits de presència** (un vector de bits amb un bit per cada node del sistema) perquè el directori sàpiga exactament **quins nodes concrets** tenen una còpia de la dada.

Aquests es diferencien dels bits d'estat (MSU) per la naturalesa de la informació que guarden:

- Els **bits de coherència (MSU)** responen a la pregunta “*En quina situació es troba la dada?*” (el seu estat de validesa global).
- Els **bits de presència** responen a la pregunta “*On és geogràficament la dada?*” (el mapa de nodes afectats).

D'aquesta manera sabem a quins nodes hem d'enviar els missatges d'invalidació de forma exclusiva punt a punt. Sense els bits de presència, el directori hauria d'enviar missatges a tots els nodes del sistema cada vegada que hi hagués una escriptura, perdent així tot l'avantatge d'escalabilitat de les arquitectures NUMA.

4.5.4 Rols dels nodes

En una transacció de coherència basada en directori, cada node pot exercir un d'aquests rols segons la seva relació amb la línia de memòria:

- **Home node (Node origen):** node on resideix físicament la línia de memòria. Conté el *slice* del directori encarregat de coordinar la coherència d'aquella línia (emmagatzemant el seu estat MSU i els bits de presència).
- **Local node (Node sol·licitant):** node amb el processador que inicia la petició (*load* o *store*) que provoca la fallada (*miss*) a la caché.
- **Remote nodes (Nodes remots):** qualsevol altre node del sistema implicat en la transacció. Poden actuar com a:
 - **Owner:** Node remot que té la línia modificada (*dirty*) i ha de proveir la dada.
 - **Readers / Sharers:** Nodes remots que tenen còpies netes i han de rebre una invalidació.

4.5.5 Comandes del protocol de directori (simplificat)

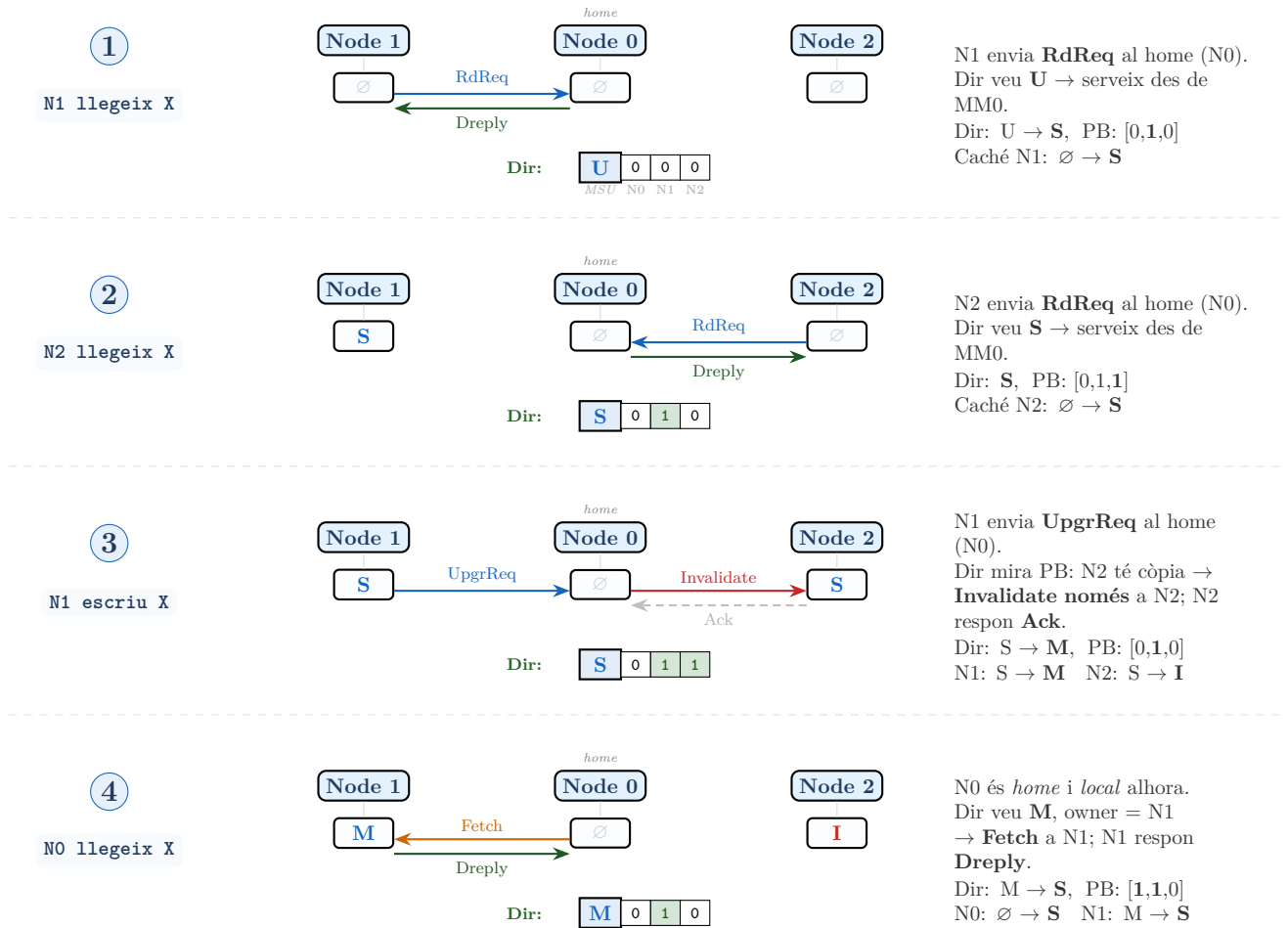
Comandes del node local cap al home node:

- **RdReq:** demana còpia de la línia sense intenció de modificar.
- **WrReq:** demana còpia de la línia amb intenció de modificar.
- **UpgrReq:** demana permís per modificar una línia ja existent a la seva caché, invalidant les altres còpies.

Comandes del home node cap a nodes remots (si cal):

- **Fetch:** demana al node remot (*owner*) una còpia de la línia (resposta: *Dreply*).
- **Invalidate:** demana al node remot (*reader*) que invalidi la seva còpia (resposta: *Ack*).

Vegem aquest escenari amb **3 nodes** (N0, N1, N2) i una línia X amb *home node* = N0.
Inicialment cap caché té la línia (directori en estat **U**).



Exemple MSU pas a pas sobre la línia X. Les caixes de caché mostren l'estat **inicial** de la línia (∅ = no present). Fletxes: → petició, → dades, → invalidació, → fetch, → ack.

Punts clau de l'exemple:

- **Cap missatge és broadcast.** Al pas 3, el directori consulta els PB i envia Invalidate **exclusivament** a N2 (l'únic node amb còpia). Amb snooping, el missatge hauria anat a **tots** els nodes. Aquesta és la raó per la qual els directoris escalen millor.
- **Els PB determinen a qui cal contactar.** Sense ells, el directori no sabria quins nodes tenen còpies i hauria de fer *broadcast*, perdent tot l'avantatge d'escalabilitat.
- **Qui serveix les dades depèn de l'estat.** Si la línia és S o U, la memòria principal (MM) serveix directament. Si és M, la memòria està desactualitzada i cal un **Fetch** a l'*owner* (pas 4).
- **UpgrReq vs WrReq.** Al pas 3, N1 ja té la línia en S: usa UpgrReq per demanar permís d'escriptura **sense transferir dades** (ja les té). WrReq s'usaria si la línia fos I.

4.5.6 Cost de la coherència: bits del directori

A diferència d'UMA (on cada caché manté els seus propis bits d'estat i la memòria principal no necessita informació de coherència), en NUMA **el directori ha d'emmagatzemar informació per cada línia de memòria**. Per cada entrada del directori cal:

- **Bits d'estat MSU:** $\lceil \log_2 3 \rceil = 2$ bits per indicar si la línia és M, S o U.
- **Bits de presència (*presence bits*):** **1 bit per node** (N bits en total), per indicar quins nodes tenen una còpia.

Per tant, cada entrada del directori ocupa $2 + N$ bits, on N és el nombre de nodes del sistema.

El cost total del directori d'un node es calcula com:

$$\text{Nombre d'entrades} = \frac{\text{Mida de la memòria local (MM)}}{\text{Mida de la línia de caché}}$$

$$\text{Bits del directori} = \text{Nombre d'entrades} \times \left(\underbrace{2}_{\text{MSU}} + \underbrace{N}_{\text{presence bits}} \right)$$

Exemple de càlcul

Suposem un sistema NUMA amb **4 nodes**. Cada node té una memòria local de **1 GB** i línies de caché de **64 Bytes**:

$$\text{Nombre d'entrades} = \frac{1 \text{ GB}}{64 \text{ Bytes/línia}} = \frac{2^{30}}{2^6} = 2^{24} \text{ entrades}$$

$$\text{Bits del directori} = 2^{24} \text{ entrades} \times (2 + 4) \text{ bits/entrada} = 3 \times 2^{25} \text{ bits} = 12 \text{ MB}$$

Això representa un *overhead* de $\frac{12 \text{ MB}}{1 \text{ GB}} \approx 1.17\%$ respecte a la mida de la memòria local.

Observem que el cost **creix linealment amb N** : afegir nodes augmenta els **presence bits de tots** les entrades de **tots** els directoris del sistema.

A més, cal recordar que cada caché dins del node també necessita els seus **bits de coherència locals** (2–3 bits per línia, com a UMA). Per tant, el cost total de coherència del sistema és la **suma** dels bits de directori (a la MM) i els bits d'estat locals (a cada caché).

4.6 Implicació pràctica: la política *first touch*

Ara que sabem que en NUMA l'accés a memòria remota és més costós i que el directori genera tràfic de coherència per cada accés, la pregunta natural és: **qui decideix a quin node resideix cada dada?**

El sistema operatiu decideix el *home node* a nivell de pàgina amb la política ***first touch***: el primer node que accedeix (toca) una pàgina fa que aquesta quedi assignada a la seva memòria. Això té una conseqüència directa per al programador:

Atenció: inicialització de dades

Inicialitzar en **sèrie** pot concentrar pàgines en un **sol node** i provocar molts accessos remots després. Inicialitzar en **paralel** pot repartir millor les dades entre nodes i millorar localitat.

Inicialització seqüencial (dolenta en NUMA):

```
for (int i = 0; i < N; i++) {  
    a[i] = random();  
    b[i] = random();  
}  
// Tots els vectors queden al·locats en un UNIC node (M0)
```

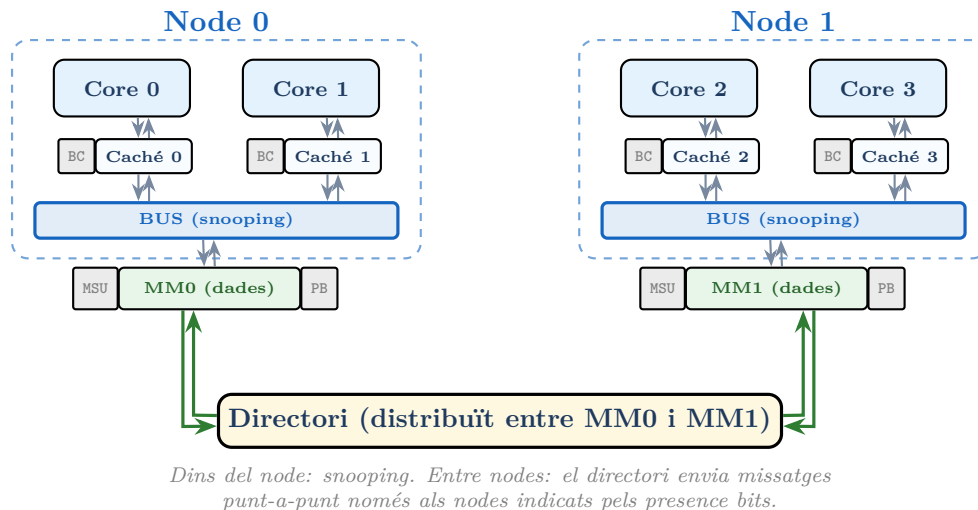
Inicialització paral·lela (bona en NUMA):

```
#pragma omp parallel num_threads(4)  
{  
    int myid = omp_get_thread_num();  
    int BS = N / omp_get_num_threads();  
    for (int i = myid*BS; i < (myid+1)*BS; i++) {  
        a[i] = random();  
        b[i] = random();  
    }  
}  
// Vectors DISTRIBUITS entre M0, M1, M2 i M3
```

Aquesta observació és el punt de partida del tema següent (descomposició de dades): no n'hi ha prou amb repartir la **feina** entre threads, cal repartir estratègicament les **dades** per maximitzar la localitat.

4.6.1 Snooping i directori alhora

És habitual una solució **híbrida**: **dins d'un node** es manté coherència amb **snooping** perquè el nombre d'agents és relativament petit i la interconnexió interna ho permet; **entre nodes**, la coherència es manté amb **directoris**. En aquest esquema, el hub del node actua com un agent que coordina el que passa dins del node i interactua amb els home nodes (directoris) quan cal coherència entre nodes.



Arquitectura NUMA amb dos nodes (2 cores/node). La memòria local de cada node (MM) emmagatzema, per a cada línia, els bits de directori: estat MSU (2 bits) i *presence bits* (1 bit/node). El directori és distribuït entre les MM.

Per altra banda, cada caché conté els seus propis bits de coherència (CB, 2 bits en MSI). Dins del node la coherència es manté per *snooping*; entre nodes, el directori envia missatges punt-a-punt només als nodes indicats pels *presence bits*.

4.7 Resum: tipus d'arquitectures multiprocessador

Arquitectura	Espai d'adreces	Comunicació	Noms
Memòria compartida centralitzada	Un sol espai d'adreces compartit; temps d'accés uniforme	Instruccions load/store	SMP, UMA
Memòria compartida distribuïda	Un sol espai d'adreces compartit; temps d'accés no uniforme	Instruccions load/store	DSM, NUMA
Memòria distribuïda	Múltiples espais d'adreces separats	Missatges explícits	<i>Message-passing</i> , Cluster

Nota: les arquitectures de memòria distribuïda (message-passing / clusters) utilitzen múltiples espais d'adreces separats amb comunicació explícita per missatges. No les treballem a PAR.

4.8 Sincronització de baix nivell

Per implementar primitives com *locks* i *barreres*, el software necessita **instruccions atòmiques** que el hardware garanteixi indivisibles.

4.8.1 Implementació de Locks: *Test-and-Set*

La instrucció ***Test-and-Set*** (***t&s***) llegeix el valor d'una adreça i, en un sol pas indivisible, hi escriu un 1. Un *spinlock* bàsic s'implementa:

Spinlock amb Test-and-Set

```
lock:    t&s    r1, flag    // r1 = flag; flag = 1 (atomicament)
        bnez   r1, lock    // si el lock estava pres, torna a intentar
        -ho
        // ... (Seccio critica) ...
unlock:  st     flag, #0    // allibera el lock
```

Problema: com que ***t&s*** inclou una escriptura, cada intent fallit genera una petició d'exclusivitat (BusRdX/WrReq) i invalida les còpies de la resta. Amb molts threads en espera, el bus o el directori queden **saturats** de tràfic inútil.

4.8.2 Optimització: *Test-and-Test-and-Set*

Test-and-Test-and-Set (TT&S)

```
lock:    ld     r1, flag    // 1r TEST: lectura normal
        bnez   r1, lock    // spin mentre el lock esta ocupat
        t&s    r1, flag    // 2n TEST: intenta agafar-lo
        bnez   r1, lock    // si algu se l'ha avançat, comença de nou
```

Mentre el lock és ocupat, els threads fan ***ld***: hit local a la caché (línia en estat **S**), **sense cap tràfic de bus**. Quan el propietari allibera el lock, les cachés s'invaliden i tots els threads intenten el ***t&s*** una vegada.

4.8.3 Load-Linked / Store-Conditional (LL/SC)

Alternativa preferida en arquitectures modernes (ARM, RISC-V) que evita bloquejar el bus:

- **LL (*Load-Linked*)**: llegeix una adreça i **marca** la línia com a vigilada.
- **SC (*Store-Conditional*)**: escriu a la mateixa adreça **només si** cap altre core l'ha modificada des del LL; retorna 1 si èxit, 0 si falla.

Si el SC falla, es torna a executar el LL. Delega el control de l'exclusió mútua al seguiment local de la caché, sense operacions pesades al bus.

4.8.4 Barreres de sincronització

Una **barrera** fa que *P* threads esperin fins que tots hi han arribat. S'implementa amb un *lock* que protegeix un comptador d'arribades i un *flag* d'alliberament: quan l'últim thread porta el comptador a *P*, activa el *flag* i desperta la resta. Mentre esperen, els threads fan spin sobre el *flag* en lectura (hits locals en **S**, sense tràfic de bus), aprofitant el mateix principi que el TT&S.

4.9 Consolidació T4 — El Hardware del parallelisme

UMA/SMP — Protocol MSI: taula de transicions

Estat	Esdeveniment	Acció al bus	Estat nou
I	PrRd	BusRd	S
I	PrWr	BusRdX	M
S	PrRd	—	S
S	PrWr	BusUpgr	M
S	Snoop BusRd	—	S
S	Snoop BusRdX / BusUpgr	—	I
M	PrRd	—	M
M	PrWr	—	M
M	Snoop BusRd	Flush	S
M	Snoop BusRdX	Flush	I

Si alguna caché té la línia en **M** → ella fa **Flush**; si és en S/I → la **memòria principal**.

NUMA — Protocol MSU, bits de presència i missatges

Estats globals al directori (*home node*):

Estat	Significat	Qui proveeix
U (<i>Uncached</i>)	Cap caché té la línia	Memòria principal
S (<i>Shared</i>)	Un o més nodes tenen còpies netes	Memòria principal
M (<i>Modified</i>)	Un node (<i>owner</i>) té còpia bruta	<i>Owner</i> (Fetch)

Bits de presència: 1 bit per node al directori. Registren exactament quins nodes tenen còpia d'una línia, permetent missatges **punt a punt** en lloc de *broadcast*.

Càlcul de bits de coherència

UMA — bits d'estat per caché:

$$\text{Bits coherència} = \frac{\text{Mida caché}}{\text{Mida línia}} \times \lceil \log_2(\text{núm. estats}) \rceil \quad (2 \text{ bits per MSI/MESI/MOSI; } 3 \text{ bits a partir de MOES})$$

NUMA — bits de directori per node (sobre la memòria local):

$$\text{Bits directori} = \frac{\text{Mida MM local}}{\text{Mida línia}} \times \left(\underbrace{2}_{\text{MSU}} + \underbrace{N}_{1 \text{ bit/node}} \right)$$

on N és el nombre total de nodes del sistema. El cost creix linealment amb N : afegir nodes augmenta els *presence bits* de totes les entrades de tots els directoris.

5 Descomposició de dades (*Data Decomposition*)

Com hem vist al tema anterior, en architectures amb memòria jeràrquica i NUMA cada bloc de dades pot estar a la memòria local d'un node, a la caché d'un core concret, o fins i tot en diverses caches allhora. Quan modifiquem aquestes dades podem provocar invalidacions que donen lloc a actualitzacions de línies de caché (coherència de dades) i, per tant, a missatges entre nodes NUMA causant *overheads* de comunicació.

Per això, quan definim les tasques no n'hi ha prou amb tenir en compte només la càrrega de treball: **cal repartir de forma estratègica les dades i qui les computa** per tal d'aprofitar al màxim:

- Aprofitar al màxim la localitat de la memòria caché
- Minimitzar el nombre de missatges de coherència i de comunicacions
- Mantenir un *load balance* raonable

5.1 Objectius de la descomposició de dades

1. Generar tasques amb càrrega similar (*load balance*):

Cada thread hauria de tenir aproximadament el mateix nombre d'iteracions o elements. Però no n'hi ha prou amb això: pot haver-hi iteracions que triguin molt més que les altres, i cal repartir bé sobretot els fragments més costosos.

2. Maximitzar la localitat de dades:

Volem que cada processador treballi sobre el seu tros de memòria. D'aquesta manera les dades tendeixen a estar ja carregades a la memòria caché (i, en NUMA, a memòria local) i reduïm fallades de caché i accessos remots.

3. Minimitzar dades compartides entre tasques:

Compartir dades entre threads implica invalidacions de caché, manteniment de coherència dins del NUMA node (bus snoopy) o entre NUMA nodes (directori MSU). Com menys dades es comparteixin de forma activa, menys tràfic de coherència generem.

4. Minimitzar volum i freqüència de transmissió de dades:

Com menys transmissions fem, millor. Si la transmissió és inevitable, és preferible fer-ho una sola vegada amb blocs grans i contigus que no pas moltes transferències petites i disperses (ens estalviem el temps d'*start-up*).

5. Solapar les transmissions de dades:

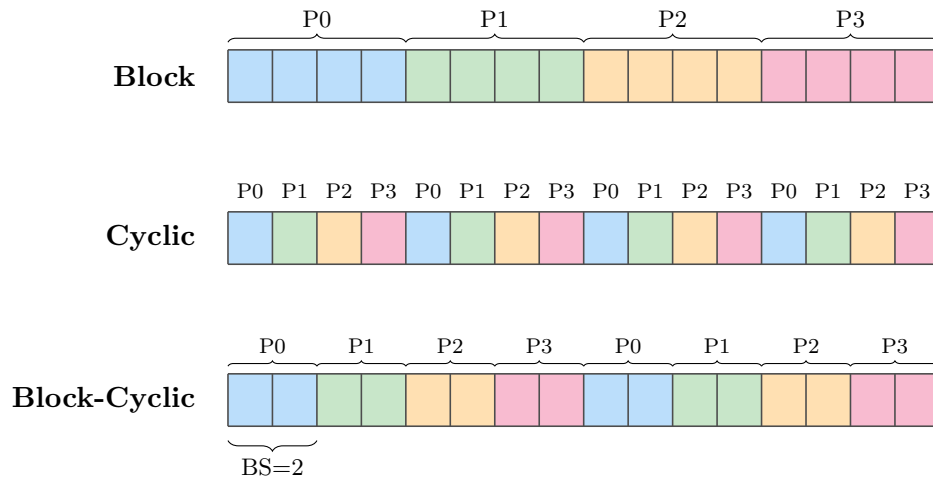
Si n'hi ha d'haver, intentarem fer-les de manera solapada per mitigar l'*overhead* de comunicació.

6. Minimitzar zones molt concurrents (*hot spots*):

Variables molt utilitzades per varis threads allhora (per exemple sumes globals) provoquen llarga espera per sincronització i molta activitat de coherència. Sempre que sigui possible, aplicarem estratègies de reducció (cada thread acumula localment i al final es combinen els resultats).

5.2 Estratègies geomètriques de descomposició

Les estratègies geomètriques particionen l'espai d'iteracions (o dades) de forma regular. Totes s'implementen amb **tasques implícites** (`omp parallel`) usant la `id` i `nt` (`omp_get_thread_num()` i `omp_get_num_threads()`) per decidir quin tros processa cada thread.



5.2.1 Block

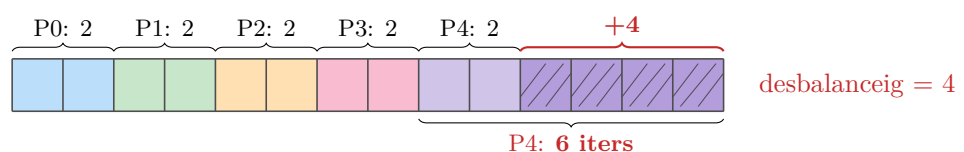
Dividim el rang d'iteracions en blocs contigus de mida similar (idealment un bloc per processador). Cada tasca executa totes les iteracions del seu bloc.

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    int BS = N / nt;
    int start = id * BS;
    int end = start + BS;
    if (id == nt-1) end = N; // ultim thread agafa el residu

    for (int i = start; i < end; i++) {
        ...
    }
}
```

Quan N no és divisible per nt , la divisió entera N/nt deixa un residu de $N \% nt$ iteracions. En el codi anterior, la condició `if (id == nt-1)end = N` fa que l'últim thread s'emporti *tot* el sobrant, provocant un desbalanceig que pot arribar a $nt-1$ iteracions extra:

$$N = 14, nt = 5 \Rightarrow BS = \lfloor 14/5 \rfloor = 2, res = 14 \bmod 5 = 4$$



L'últim thread acumula tot el residu $BS (2) + N \bmod nt(4)$ extres.

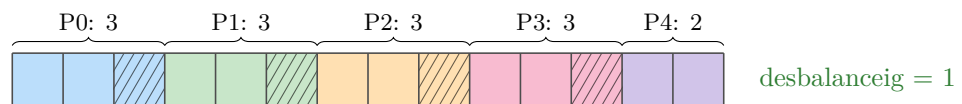
5.2.2 Block amb el residu repartit:

Per evitar-ho, podem repartir el residu entre els primers `res` threads, de manera que cadascun d'ells fa `BS+1` iteracions en lloc de `BS`. Així el desbalanceig màxim és sempre d'1 sola iteració:

```
int id = omp_get_thread_num();
int nt = omp_get_num_threads();
int BS = N / nt;
int res = N % nt;

// Els primers 'res' threads fan BS+1 iteracions, la resta fan BS
int start = BS * id + ((id < res) ? id : res);
int end = start + BS + ((id < res) ? 1 : 0);

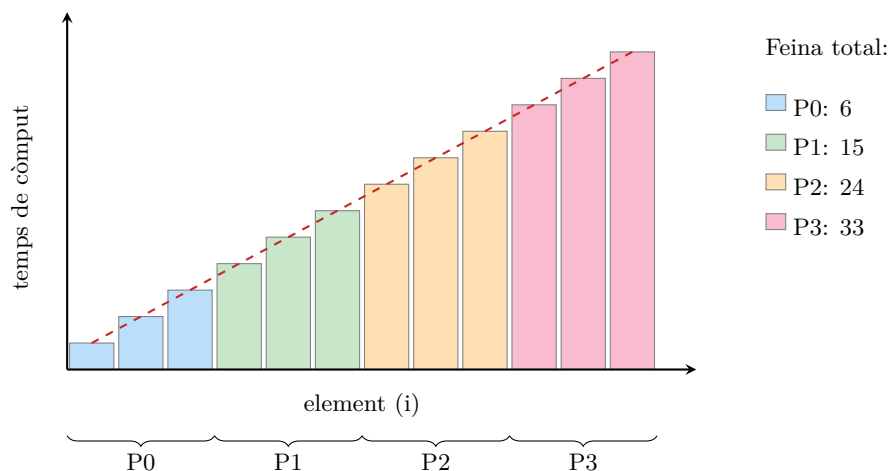
for (int i = start; i < end; i++) {
    ...
}
```



Els `res` (4) primers threads fan `BS+1` (3) iteracions i la resta fa `BS` (2), aconseguint `desbalanceig=1`

Propietats de l'estratègia Block:

- **Localitat:** cada thread recorre un rang contigu $\Rightarrow$ maximitza la localitat espacial de caché.
- **Coherència / false sharing:** baix en general, perquè els threads treballen sobre línies de caché diferents. Pot aparèixer *només* a les fronteres entre blocs si diversos threads escriuen a elements que cauen dins la mateixa línia de caché.
- **Load balance:**
 - **En elements:** amb el residu repartit, el desbalanceig és com a màxim d'1 iteració.
 - **En temps de còmput:** si el cost per iteració no és uniforme, els processadors amb les iteracions més costoses acumulen molta més feina que els altres, provocant que alguns treballin molt més que d'altres.



En casos com un espai triangular o el Mandelbrot, l'estratègia block pot deixar processadors inactius mentre d'altres acumulen la major part del treball.

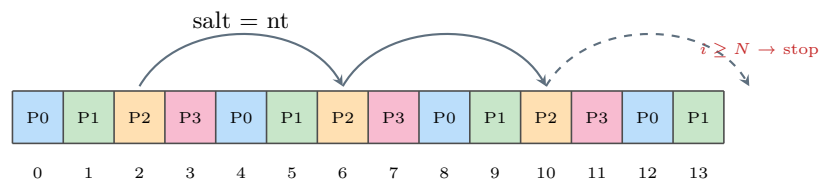
5.2.3 Cyclic

Dividim el rang d'iteracions assignant-les de manera saltejada: cada thread executa les iteracions id , $id+nt$, $id+2*nt$, ... fins a sobrepassar N .

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();

    for (int i = id; i < N; i += nt) {
        ...
    }
}
```

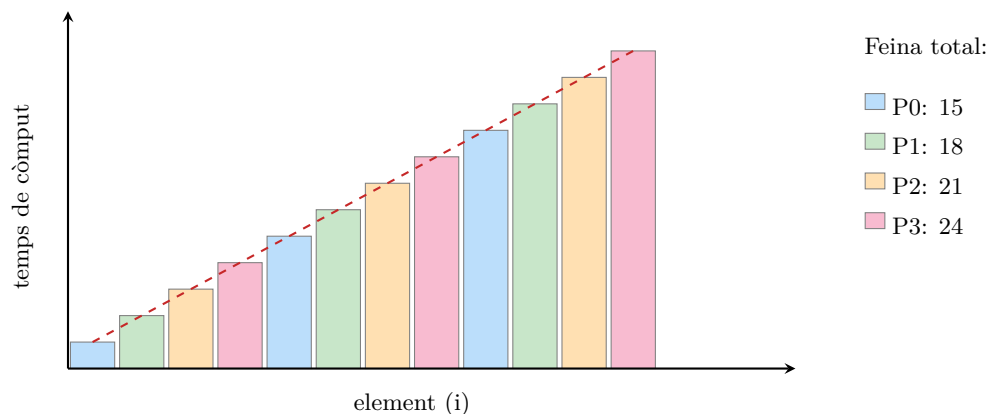
$nt = 4$: cada thread executa $i = id, id+4, id+2\cdot 4, \dots$



Cada thread salta de nt en nt elements, fins a sobrepassar N .

Propietats de l'estratègia Cyclic:

- **Localitat:** molt dolenta — cada thread salta de nt en nt elements, destruint la localitat espacial i gairebé mai reutilitzant una línia de caché ja carregada.
- **Coherència / false sharing:** molt alt. Els elements consecutius pertanyen a threads diferents, de manera que dins de cada línia de caché hi escriuen múltiples processadors, provocant invalidacions constants.
- **Load balance:**
 - **En elements:** diferència ≤ 1 iteració sense cap correcció de residu.
 - **En temps de còmput:** les iteracions es reparteixen intercalades, barrejant les cares amb les barates i equilibrant la càrrega total.



Amb cyclic, cada thread rep una barreja d'iteracions barates i cares, equilibrant la càrrega total tot i que el cost per iteració creix.

5.2.4 Block-Cyclic

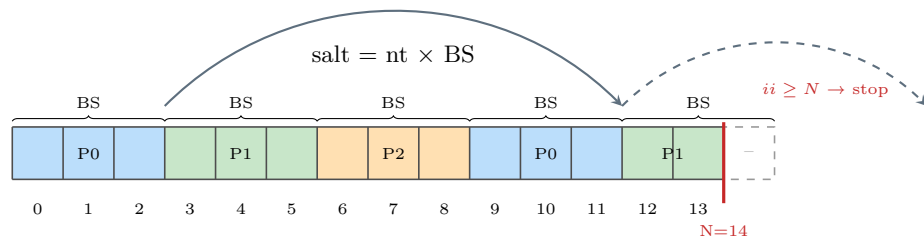
L'espai d'iteracions es parteix en blocs contigus de mida fixa BS i aquests blocs es reparteixen de manera cíclica entre els threads. És el **punt mig** entre block i cyclic, ja que:

- Si $BS = 1 \Rightarrow$ cyclic pur (màxim balanceig, mínima localitat).
- Si $BS = N/nt \Rightarrow$ block pur (màxima localitat, mínim balanceig temporal).

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int nt = omp_get_num_threads();
    const int BS = 3; // mida del bloc fixa (iteracions)

    // ii = inici del bloc actual
    // min(ii+BS, N) = fi del bloc (evita sobrepassar N)
    for (int ii = id * BS; ii < N; ii += nt * BS) {
        for (int i = ii; i < min(ii+BS, N); i++) {
            ...
        }
    }
}
```

$nt = 3$, $BS = 3$, cada thread comença el bloc a $ii = id \cdot BS$, $id \cdot BS + (nt \cdot BS)$, $id \cdot BS + 2(nt \cdot BS)$, ...



Cada thread processa un bloc contigu de BS elements i salta $nt \times BS$ posicions.

L'últim bloc és incomplet, ja que $\min(ii+BS, N)$ evita sobrepassar N.

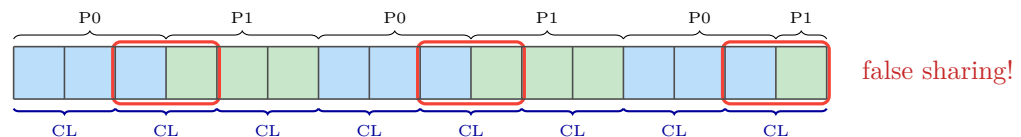
5.2.5 Propietats de l'estratègia Block-Cyclic:

- **Localitat:** millor que cyclic — dins de cada bloc, el thread recorre BS elements contigus, aprofitant la localitat espacial. Com més gran sigui BS, millor localitat (s'apropa a block).
- **Coherència / false sharing:** el risc depèn de BS. Si BS està alineat amb la línia de caché (per exemple, $BS = \text{múltiple d'elements per línia}$), cada bloc ocupa línies senceres i no hi ha false sharing a les fronteres. Un BS petit o no alineat pot provocar que el final d'un bloc i l'inici del següent (d'un altre thread) comparteixin línia.
- **Load balance:**
 - **En elements:** el desbalanceig pot arribar a BS iteracions (un processador executa un bloc més que els altres).
 - **En temps de còmput:** millora respecte block perquè els blocs es reparteixen cíclicament, barrejant zones de cost diferent entre els threads. Com més petit sigui BS, millor balanceig temporal (s'apropa a cyclic).

5.2.6 Block-Cyclic alineat a línia de caché

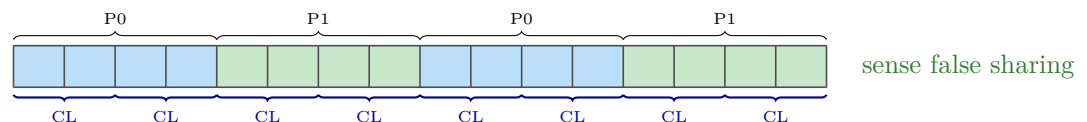
Un dels problemes del block-cyclic és que si BS no coincideix amb la mida de la línia de caché, la frontera entre blocs de threads diferents pot caure *dins* d'una mateixa línia, provocant false sharing.

BS = 3, CL = 2: les línies encerclades contenen elements de 2 processadors $\Rightarrow$ false sharing.



La solució és escollir BS com a múltiple del nombre d'elements que caben en una línia de caché:

$$BS = k \times \frac{\text{cache_line_size}}{\text{sizeof(element)}} \quad (k = 1, 2, \dots)$$



BS = 4 (múltiple de CL = 2): cada línia de caché pertany a un sol processador.

Amb BS alineat, cada bloc ocupa línies de caché senceres i cap línia és compartida entre threads, eliminant el false sharing.

5.2.7 Comparativa de les estratègies

Estratègia	Localitat	False sharing	Imbalance elements	Imbalance temps
Block	Bona	Baix (fronteres)	$\leq nt-1$	Alt
Block Repartit	Bona	Baix (fronteres)	≤ 1	Alt
Cyclic	Dolenta	Molt alt	≤ 1	Baix
Block-Cyclic	Bona	Depèn de BS	$\leq BS$	Mig
Block-Cyclic Alineat	Excel·lent	No n'hi ha!	$\leq BS$	Mig

- Si el cost per iteració és **uniforme** i volem **màxima localitat**: **Block (Repartit)** és la millor opció, ja que combina rang contigu amb desbalanceig mínim.
- Si el cost per iteració és **molt irregular** i el que més importa és **equilibrar la càrrega**: **Cyclic** reparteix millor el temps, però a costa de destruir la localitat i provocar false sharing.
- Si volem un **compromís** entre localitat i balanceig: **Block-Cyclic Alineat** és l'opció més versàtil. Ajustant BS controlem el balanç entre més localitat o més balanceig.

En general, **Block-Cyclic Alineat** amb un BS múltiple de la línia de caché sol ser l'estratègia per defecte més segura, ja que ofereix bona localitat, elimina el false sharing i manté un balanceig raonable.

5.3 Padding per solucionar false sharing

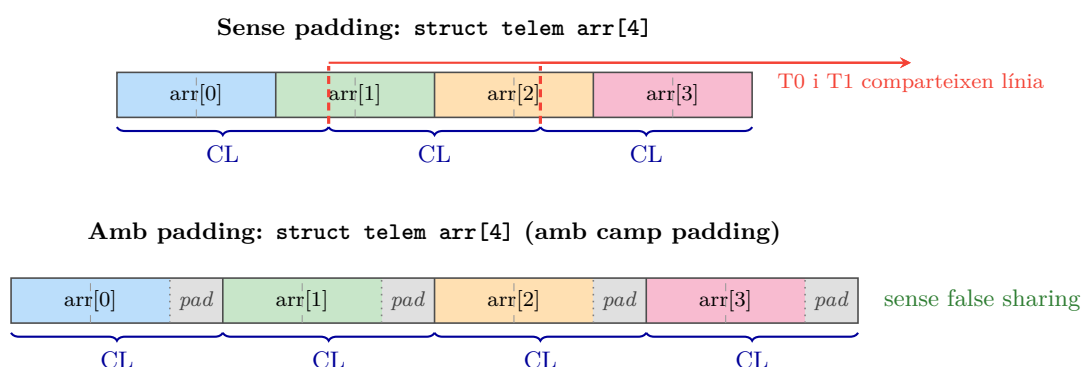
A més d'alinejar BS a la línia de caché, podem aplicar **padding** directament a les estructures de dades per garantir que elements de threads diferents ocupin línies separades.

Padding en structs

L'objectiu és que cada instància d'struct ocupi exactament un múltiple de la línia de caché. S'afegeix un camp padding dins l'struct per omplir fins al proper múltiple:

```
#define CL <cache_line_size_bytes>

struct telem {
    <camps_utils>
    char padding[CL - (sizeof(<camps_utils>)%CL)];
}; // Utilitzem char pq ocupa 1 byte, així controlem els bytes de padding
```



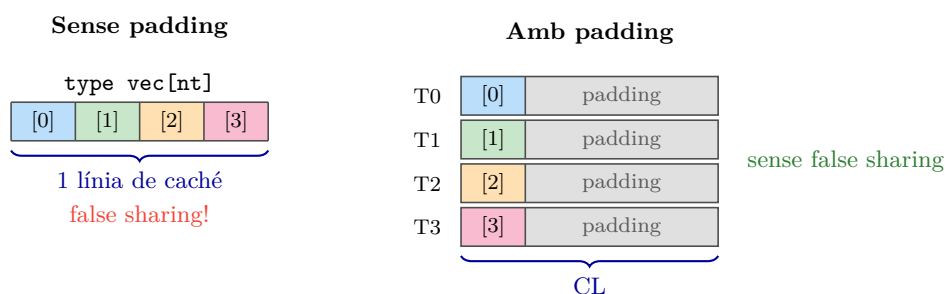
El padding intern alinea cada instància a una línia de caché, eliminant les zones de false sharing.

Padding en vectors

Quan threads escriuen a posicions contigües d'un vector compartit type vec[nt], diversos elements poden caure en la mateixa línia de caché, causant false sharing. Per evitar-ho, es separen els elements amb un factor PAD igual al nombre d'elements per línia de caché.

```
#define CL <cache_line_size_bytes>
#define PAD (CL / sizeof(type))
type vec[nt * PAD]; // cada thread a una linia diferent

// Accés: vec[id * PAD] en lloc de vec[id]
```

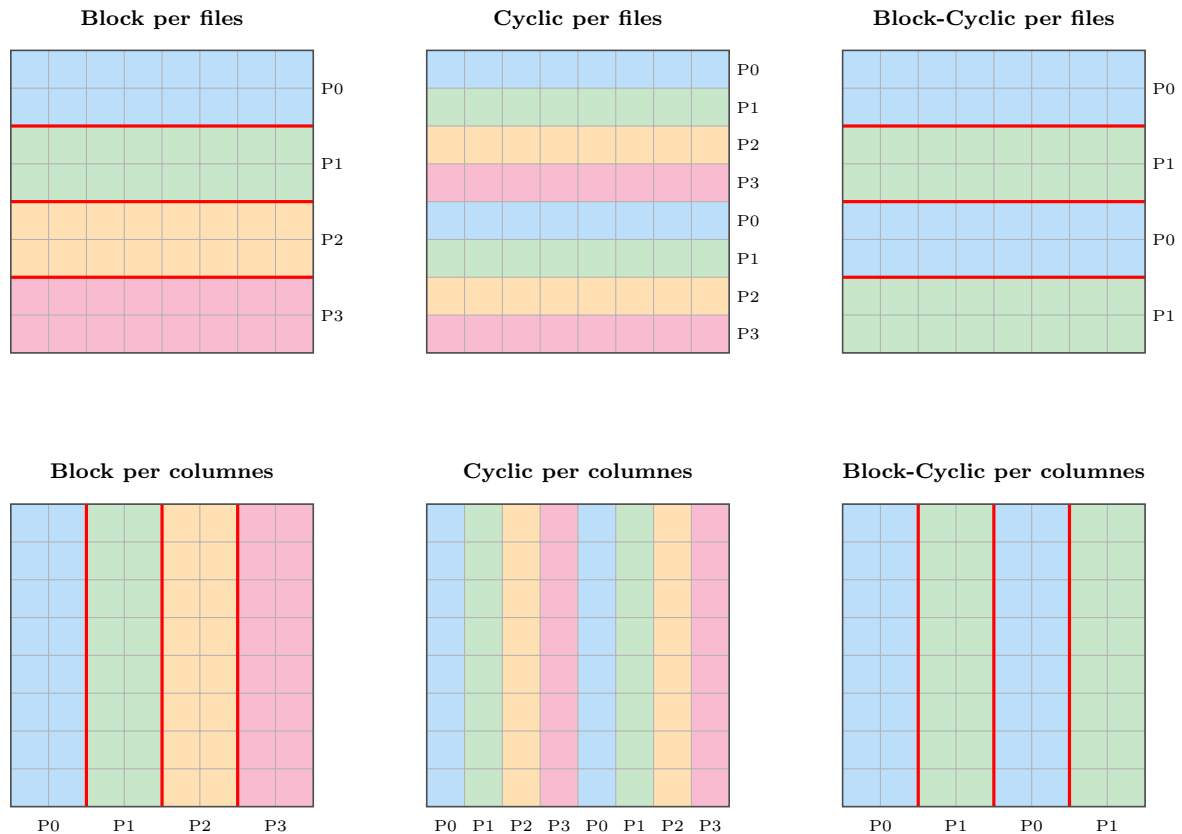


Cada element ocupa una línia de caché sencera, eliminant el false sharing.

El padding augmenta l'ús de memòria (la major part de l'array és espai buit), però en vectors petits de control com comptadors o sumes parcials, el cost és negligible comparat amb la millora de rendiment.

5.4 Descomposicions 2D (matrius)

Per a descompondre matrius, podem aplicar les mateixes estratègies per files, o bé per columnes:



Estratègia 2D	Localitat	False sharing	Imbalance elements	Imbalance temps
Block per files	Bona (files contigues)	Baix	≤ 1 fila	Alt
Cyclic per files	Acceptable (tot i anar saltant)	Baix	≤ 1 fila	Baix
Block-Cyclic per files	Bona (files contigues)	Baix	$\leq BS$ files	Mig
Block per columnes	Dolenta	Baix	≤ 1 col	Alt
Cyclic per columnes	Molt dolenta	Molt alt	≤ 1 col	Baix
Block-Cyclic per columnes	Dolenta	Depèn de BS	$\leq BS$ cols	Mig

La diferència clau entre files i columnes és la **disposició en memòria**. Les matrius es guarden per files (*row-major*), de manera que elements consecutius d'una fila són contigus en memòria. Per això qualsevol estratègia per files té millor localitat que la seva equivalent per columnes.

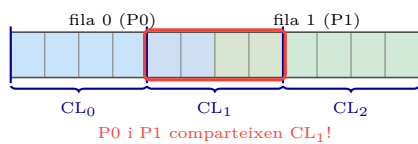
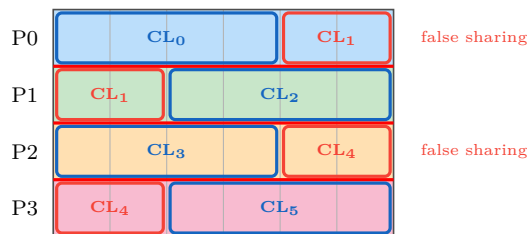
5.5 Padding en matrius

Quan el nombre de columnes no és múltiple de la línia de caché, el final d'una fila i l'inici de la següent comparteixen línia de caché en memòria (row-major), provocant false sharing entre threads que treballen sobre files adjacents. La solució és afegir columnes extra de padding perquè cada fila ocupi un nombre enter de línies de caché:

```
#define CL      <cache_line_size_bytes>
#define ELEM_CL (CL / sizeof(type)) // elements per linia de cache
#define PAD     ((COLS % ELEM_CL == 0) ? 0 : (ELEM_CL - COLS % ELEM_CL))

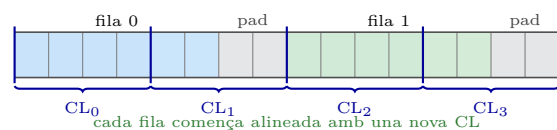
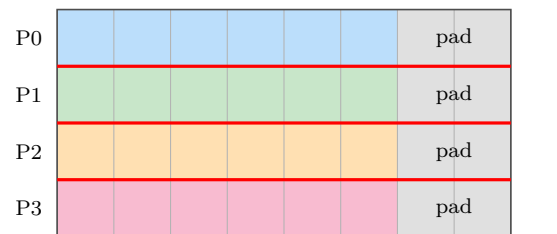
type M[ROWS][COLS + PAD]; // cada fila = multiple de CL bytes
// Accés: només columnes [0..COLS-1], ignorar el padding
```

Sense padding (6 cols, CL=4)



P0 i P1 comparteixen CL1!

Amb padding (6 cols + 2 pad)



cada fila comença alineada amb una nova CL

Matriu: si el nombre de columnes no és múltiple de CL, el final d'una fila i l'inici de la següent comparteixen línia (esquerra). Afegint columnes de padding, cada fila comença alineada (dreta).

El padding en matrius es combina amb qualsevol estratègia de descomposició. Per exemple, amb block-cyclic per columnes, BS pot servir alhora com a mida del bloc i com a unitat de padding, alineant cada bloc a una línia de caché sencera. El padding augmenta l'ús de memòria però pot millorar dràsticament el rendiment.

5.6 Consolidació T5 — Data decomposition

Objectius:

- Balanceig de càrrega: generar tasques amb temps de còmput similar (no només el mateix nombre d'iteracions).
- Maximitzar la localitat de dades (caché i NUMA).
- Minimitzar dades compartides entre tasques → menys tràfic de coherència.
- Minimitzar volum i freqüència de transmissions.
- Minimitzar zones molt concurrents (*hot spots*) → usar reduccions.
- Solapar transmissions per amagar-ne l'*overhead*.

Estratègies de descomposició geomètrica:

- **Block:** cada processador executa un rang contigu. Molt bona localitat, poc tràfic de coherència. Mal balanceig si el cost per iteració no és uniforme.
- **Cyclic:** iteracions saltejades. Excel·lent balanceig, però mala localitat i alt risc de false sharing.
- **Block-Cyclic:** compromís entre ambdues. Triar BS alineat amb la línia de caché per evitar false sharing i maximitzar localitat.

False sharing i solucions:

- Alinear BS a la línia de caché.
- Block-cyclic en lloc de cyclic.
- Padding (afegir espai entre variables de threads diferents).



ASSES NORD

González Tablas, 7
08034 Barcelona



93.125.70.02 / 605.411.687



@fiberases
@fib_primer_ases



www.asesacademia.com

