

Sobre el uniprocesador podemos "conseguir" paralelismo de varias formas:

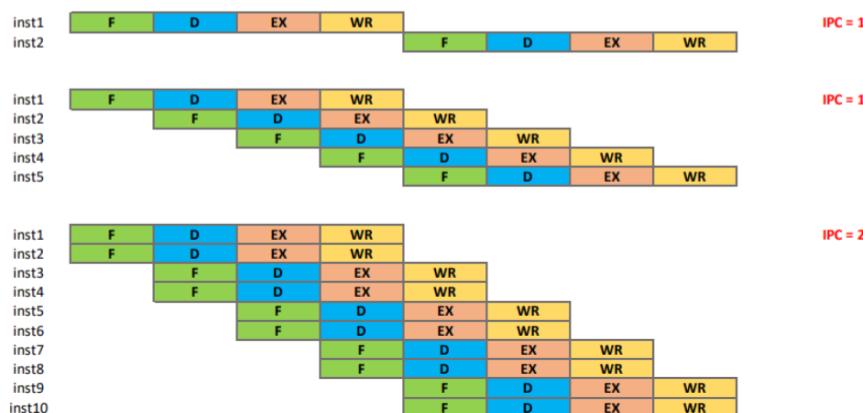
- Paralelismo a nivel de instrucción (procesadores superscalares): Añadimos hardware extra en el pipeline y podemos ejecutar más instrucciones a la vez (en el mismo ciclo) DEL MISMO CONTEXTO DE EJECUCIÓN².
- Paralelismo a nivel de thread: "Rellenar" los ciclos vacíos (o perdidos por fallo de cache, latencia de memoria) del pipeline con instrucciones de diferentes contextos de ejecución.
- Paralelismo a nivel de datos (instrucciones multimedia-SIMD³): Se trata de calcular, una única instrucción sobre un gran conjunto de datos (por ejemplo los datos de un bucle for se pueden hacer todos en paralelo con una única instrucción SIMD).

En resumen, podemos conseguir paralelismo en un programa con:

- Principios de localidad espacial y temporal.
- Vectorizando operaciones con instrucciones SIMD.
- Alineando a bloque cache / memoria para un acceso más rápido
- Reordenación de instrucciones para explotar el paralelismo a nivel de instrucción.
- Típicos de todos los programas, a través de la jerarquía de memoria.
- Programación consciente de la arquitectura (PCA).

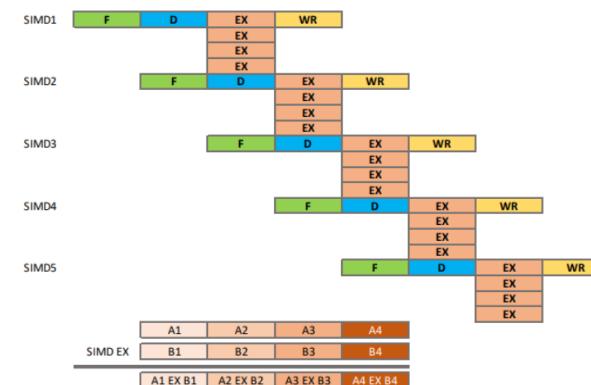
Pipelined and superscalar architecture

- ▶ Execution of single instruction divided in multiple stages
- ▶ Overlap the execution of different stages of consecutive instructions (pipelined) and consecutive instructions (superscalar)



SIMD architecture

- ▶ DLP (data-level parallelism): Single-Instruction executed on Multiple-Data (SIMD): vector functional unit



Memory hierarchy

- ▶ The principle of locality: if an item is referenced ...
 - ▶ Temporal locality: ... it will tend to be referenced again soon (e.g., loops, reuse)
 - ▶ Spatial locality: ... items whose addresses are close by tend to be referenced soon (e.g., straight line code, array access)
- ▶ Line (or block)
 - ▶ A number of consecutive words in memory (e.g. 32 bytes, equivalent to 4 words x 8 bytes)
 - ▶ Unit of information that is transferred between two levels in the hierarchy
- ▶ On an access to a level in the hierarchy
 - ▶ Hit: data appears in one of the lines in that level
 - ▶ Miss: data needs to be retrieved from a line in the next level

SIMD: Single Instruction Multiple Data: DLP (data-level parallelism)

Memory hierarchy:

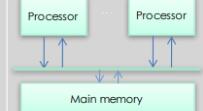
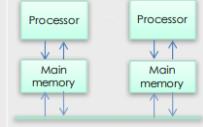
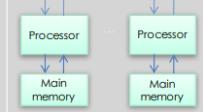
- LOCALITAT:
 - Temporal: la dada es tornarà a accedir aviat (loop)
 - Espaijal: accedir a les adreces del costat (vector, matriu)
- LÍNIA/BLOC: Paraules consecutives en memòria (32B = 4 words x 8B)
- ACCÉS:
 - Hit: la dada esa a una de les línies d'aquell nivell.
 - Miss: la dada s'ha de recuperar d'una línia del següent nivell.
- ALGORISMES DE REEMPLACAMENT: Iru, fifo, Ifu....
- WRITE POLICY: write-through, write-back (on hit), write-allocate, write-no-allocate (on miss)

Elements of cache design

- ▶ Organization
 - ▶ Single vs. multilevel cache, Unified vs. split (instruction/data)
 - ▶ Cache size and line size
 - ▶ Addressing: logical vs. physical
- ▶ Placement algorithm
 - ▶ Direct, associative, set associative
- ▶ Replacement algorithm
 - ▶ Random, Least Recently Used (LRU), First-in First-out (FIFO), Least Frequently Used (LFU)
- ▶ Write (on hit) Policy
 - ▶ Write-through, Write-back
- ▶ Write (on miss) Policy
 - ▶ Write-allocate, write-no-allocate

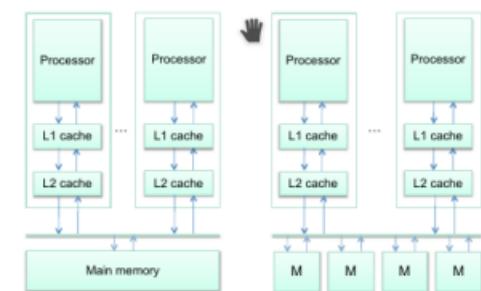
1. Organización de la jerarquía de memoria.
 - a. Una única caché.
 - b. Múltiples niveles de caché.
 - c. Caché de datos y de instrucciones.
2. Política de asignación en la caché : ¿Dónde colocamos una dirección de memoria?
 - a. Directa.
 - b. Asociativa.
 - c. Asociativa por conjuntos.
3. Política de reemplazo ¿Qué hacer si no hay sitio y tenemos que traer un bloque?
 - a. LRU
 - b. FIFO
 - c. etc...
4. Escritura en acierto : ¿Qué hacer si una escritura acierta en caché?
 - a. Write-through : Actualizar MEM + CACHE.
 - b. Write-back : Actualizar CACHE y memoria cuando la línea salga de la caché.
5. Escritura en fallo : ¿Qué hacer si escribimos y fallamos ?
 - a. Write-allocate: Nos traemos la línea de caché.
 - b. Write-no-allocate: No nos traemos la línea.

Classification of multi-processor architectures

Memory architecture	Address space(s)	Connection	Model for data sharing	Names
(Centralized) Shared-memory architecture	Single shared address space, uniform access time		Load/store instructions from processors	<ul style="list-style-type: none"> • SMP (Symmetric Multi-Processor) architecture • UMA (Uniform Memory Access) architecture
Distributed-memory architecture	Single shared address space, non-uniform access time		Load/store instructions from processors	<ul style="list-style-type: none"> • DSM (Distributed-Shared Memory architecture) • NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	<ul style="list-style-type: none"> • Message-passing multiprocessor • Cluster Architecture • Multicomputer

11/58

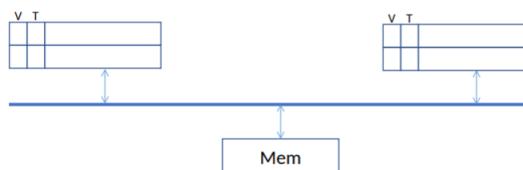
- Memoria centralizada (*shared-memory*): *Symmetric Multiprocessor Architecture SMP*
 - Un único espacio de direcciones y una única instancia de S.O.
 - El acceso a memoria de todos los procesadores es uniforme (U.M.A.).
 - El acceso a memoria se hace con instrucciones load/store.
 - **Existen problemas de coherencia de datos en memoria**



How to get coherence?

- 1) No caches (bad performance)
- 2) All cores share same L1 cache (bad performance)
- 3) Private write-through caches (Incoherent!)
- 4) Force read in one cache to see write made in another *on the writer side*:
 - a) Broadcast writes to update other caches (write update coherence)
 - b) Writes prevent hits to other copies (write-invalidate coherence)
 - c) Writes broadcasted on shared bus (snooping)
 - d) Each block assigned an ordering point (directory)

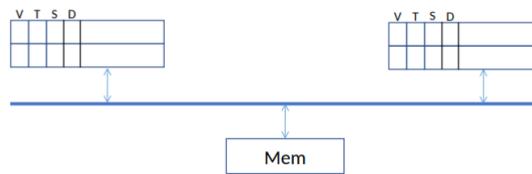
Write-update snooping coherence



Note 1: diagram showing the private caches of two processors and main memory. Each cache line has a valid bit (V), tag and data. Caches access the main memory through a bus.

Note 2: Listen to explanation in video for the specific example

Write-invalidate snooping coherence



Note 1: diagram showing the private caches of two processors and main memory. Each cache line has a valid bit (V), share bit (S) and dirty bit (D), tag and data. Caches access the main memory through a bus.

Note 2: Listen to explanation in video for the specific example

Coherence protocols:

- ▶ Write-update: writing processor broadcasts **the line** with the new value and forces all others to update their copies
- ▶ Write-invalidate: writing processor forces all others to invalidate their copies; **the line** with the new value is provided to others when requested or when flushed from cache

Coherence mechanisms:

- ▶ Broadcast-based (snooping): bus serves as broadcast mechanism to maintain coherency among copies of the same memory line in caches
- ▶ Directory-based: the sharing status of each line in memory is kept centralised in just one location (directory)

Write Update coherence: per cada **escriptura** s'ha d'enviar a totes les **altres caches** que poden tenir el **valor i actualizar-lo**

Write Invalidate coherence: després d'una **escriptura** a una de les cache, totes les **cache** que tenien una **copia** del valor s'han **d'invalitzar** i actuaran com si fos un miss el següent cop que vulguin accedir a la dada
Snooping: totes les **escriptures són broadcast** al bus compartit i l'ordre en el que apareixen en el bus és l'ordre en el que s'han de veure per tothom

Concepts in video lesson 5 (cont.)

The coherence problem:

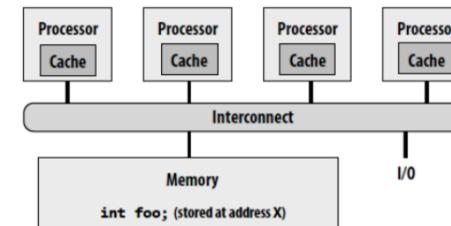


Chart shows value of **foo** (variable stored at address X) stored in main memory and in each processor's cache**

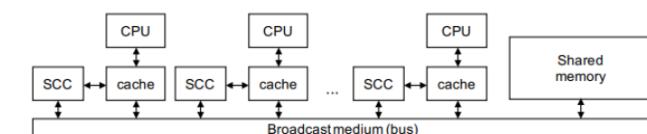
** Assumes write-back cache behavior

Action	P1 \$	P2 \$	P3 \$	P4 \$	mem[X]
P1 load X	0	miss	0	0	0
P2 load X	0	0	miss	0	0
P1 store X	1	0	0	0	0
P3 load X	1	0	0	miss	0
P3 store X	1	0	2	0	0
P2 load X	1	0	hit	2	0
P1 load Y (say this load causes eviction of foo)	0	2	0	0	1

(CMU 15-418, Spring 2012)

Broadcast-based (snooping) coherence mechanism

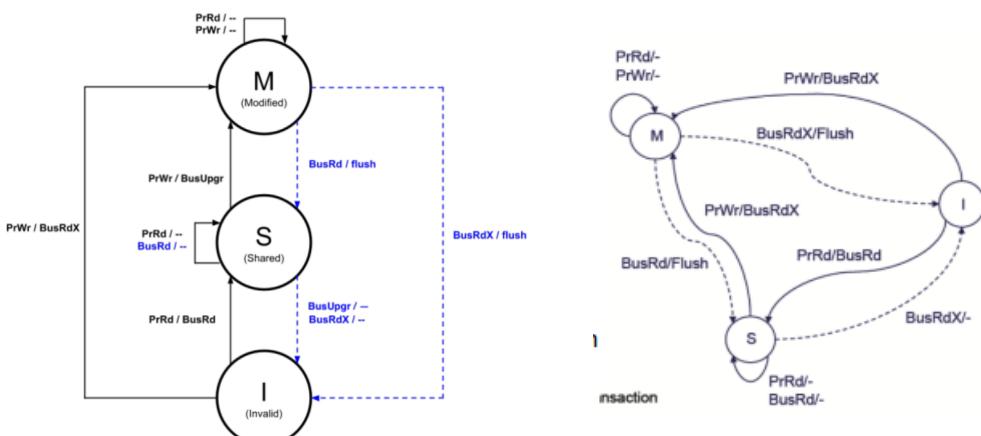
- ▶ Cache coherence is maintained at **cache line granularity**, NOT at the individual words inside the cache line
- ▶ Every cache that has a copy of a line from physical memory keeps its sharing status (status distributed)
- ▶ Broadcast medium (e.g. a bus) used to make all transactions visible to all caches and define **ordering**
- ▶ Caches monitor (snoop on) the medium and take action on relevant events (SCC: snoopy cache controllers)



Simple write-invalidate snooping protocol (MSI)

- ▶ A line in a cache memory can be in three different states:
 - ▶ Modified (M): dirty copy of the line
 - ▶ Shared (S): clean copy of the line
 - ▶ Invalid (I): invalidated copy of the line (not valid), or it does not exist in cache
- ▶ CPU events
 - ▶ PrRd (Processor read)
 - ▶ PrWr (Processor write)
- ▶ Bus events (caused by cache controllers)
 - ▶ BusRd: asks for copy with no intent to modify
 - ▶ BusRdX: asks for copy with intent to modify
 - ▶ BusUpgr: asks for permission to modify existing line, causes invalidation of other copies
 - ▶ Flush: puts line on bus, either because requested or voluntarily when dirty line in cache is replaced (WriteBack)

Simple write-invalidate snooping protocol (MSI)



- ▶ Who provides the line when requested via BusRd or BusRdX?
 - ▶ If line in S or I in other caches then main memory provides it
 - ▶ If line in M in another cache then this cache provides it (Flush)

Write-invalidate snooping protocol - MSI

En este protocolo, tendremos que cada CPU puede realizar lecturas y escrituras (eventos PrRd y PrWr) contra la jerarquía de memoria y el sistema de *broadcast* genera diferentes peticiones por el bus que ven todas las cachés:

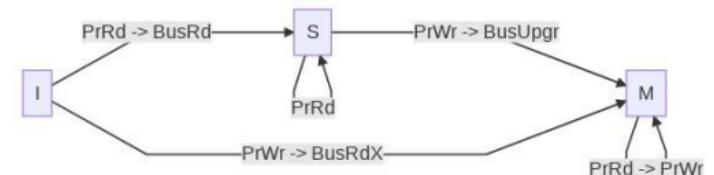
- BusRd: Petición de una línea para lectura.
- BusRdX: Petición de una línea con intención de modificarla.
- BusUpgr : Petición de escritura contra una línea de caché, causa la invalidación de las copias.
- Flush : Alguien nos pide el dato de caché o la política de reemplazo saca una linea en estado M.

Tenemos 3 posibles estados para una línea de caché:

1. Invalido: La línea de caché no es válida o no existe en la caché.
2. Modificado: Copia "sucia" de la línea de caché.
3. Shared: Copia "limpia" y compartida de una línea de cache.

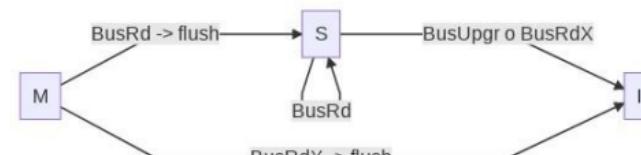
Tenemos dos diagramas de transición de estados:

1. En función de qué operación hace nuestra CPU y cómo afecta a nuestras líneas de caché.



En la imagen, \rightarrow se lee como "implica"

2. En función de qué operación realiza otra CPU y cómo afecta a nuestras líneas de caché. Cada caché observa los eventos del bus.



En la imagen, \rightarrow se lee como "implica". Si llega cualquier evento a I, nos quedamos y solo saldremos de él si necesitamos leer algo.

MSI optimizations. Thread-private lines

- ▶ MSI requires two bus transactions for the common case of read followed by write, both from the same processor (no sharing at all)
 - ▶ Transaction 1: BusRd to move from I to S state
 - ▶ Transaction 2: BusUpgr to move from S to M state
- ▶ MESI protocol adds E (Exclusive) clean state:
 - Cache line in E if only one clean copy of the line.
 - If write access by the same processor, the upgrade from E to M does not require a bus transaction (BusUpgr).
 - If line in E and another cache requests, it then cache line state changes from E to S.

Ejemplo de ejecución - MSI

Ejemplo de cómo varían los estados en una ejecución de instrucciones, con política de escritura retardada en acierto.

Evento	Cache P1	Cache P2	Cache P3	Status P1	Estado P2	Estado P3	MEM	Comentario
Init, T = 0	-	-	-	I	I	I	0	Si la línea no está en caché, estado I
P1 ld X	0 (miss)	-	-	S	I	I	0	Estado I → Estado S a causa de una lectura (PrRd)
P2 ld X	0	0(miss)	-	S	S	I	0	IDEML anterior
P1 st 1, X	1	0	-	M	I	I	0	Estado S → Estado M Una PrWr del procesador invalida todas las demás copias.
P3 ld X	1	0	1 (miss)	S	I	S	0	P3 hace una lectura (de I → S) y el dato se lo proporcionará la caché de P1 (flush). El dato de P1 pasa a estar compartido (estado S)
P3 st 2, X	1	0	2	I	I	M	0	P3 tiene una copia válida, estado S → estado M e invalida las demás.
P2 load X	1	2	2	I	S	S	0	P2 tiene una copia invalida, P3 hace flush y proporciona la línea de cache a P2.
P1 load Y	mem[Y]	2	2	I	S	S	0	Flush no upd.
P2 load Y	mem[Y]	mem[Y]	2	I	I	S	2	Flush+ update

Resumen y escalabilidad

- El protocolo de coherencia actúa sobre las líneas de caché.
- Si estamos en los estados **I** o **S**, el dato lo proporciona la memoria principal.
- Solamente puede haber una única copia sucia en toda la caché.
- Si hay una copia sucia (estado **M**) en alguna caché y hay una petición sobre esta línea, esta caché proporcionará el dato.
- Hay variaciones/optimizaciones de este protocolo añadiendo más estados para controlar ciertos casos.
- Este tipo de protocolos de *broadcast* no escalan, la red se puede colapsar en algún momento por la cantidad de mensajes necesarios; podemos evitar esto teniendo un sistema centralizado (directorio).

True vs. false sharing

- ▶ True sharing
 - ▶ Data sharing is unavoidable in parallel computing. Coherence mechanisms are there to allow this data sharing; synchronization allows to share appropriately.
- ▶ False sharing
 - ▶ Cache line may also introduce artefacts: more than 1 (distinct) data object, or also multiple elements of same object, may reside in the same cache line
 - ▶ False sharing occurs when different processors make references (read and write) to those different objects or elements within the same cache line, thereby inducing "unnecessary" coherence operations.

True sharing example

Assume each task is executing an instance of the following dot_product function:

```
int result = 0;
void dot_product(int *A, int *B, int n) {
    for (int i=0; i< n; i++)
        #pragma omp atomic
        result += A[i] * B[i];
}
```

The line containing variable `result` is subject to coherence actions at each iteration of `i`. It could be easily transformed into

```
void dot_product(int *A, int *B, int n) {
    int tmp = 0;
    for (int i=0; i< n; i++)
        tmp += A[i] * B[i];
    #pragma omp atomic
    result += tmp;
```

to reduce cache coherence traffic. **Note:** `atomic` is used to guarantee exclusive access to variable `result`

False sharing example

```
struct foo {
    int x, y; //x and y will reside in same cache line
} f; // aligned to cache line

void main() {
    int s=0;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(s)
        for (int i=0;i<1000000;i++)
            s+=f.x

        #pragma omp task
        for (int i=0;i<1000000;i++)
            f.y++
    }
}
```

How data is stored in memory?
cache line



Assumptions:
- Variable f aligned to cache line
- Cache line 16 bytes wide
- int occupies 4 bytes

False sharing of the line containing fields `x` and `y`. How could we force fields `x` and `y` to be in different cache lines?

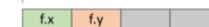
False sharing example (cont.)

```
struct foo {
    int x;
    int padding[3];
    int y; //x and y will NOT reside in same cache line
} f; // aligned to cache line

void main() {
    int s=0;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(s)
        for (int i=0;i<1000000;i++)
            s+=f.x

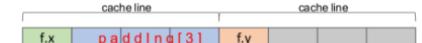
        #pragma omp task
        for (int i=0;i<1000000;i++)
            f.y++
    }
}
```

How data is stored in memory?
cache line



Assumptions:
- Variable f aligned to cache line
- Cache line 16 bytes wide
- int occupies 4 bytes

Padding to avoid false sharing



Add a dummy field in between to separate fields `x` and `y` in different cache lines: `int padding[PAD_SIZE]`; How much padding?

Bits per mantenir la coherència al sistema INSIDE numa node:

(#processadors *) [#línies * 2bits/línia, #línies = mida MC/ mida línia caché]

NUMA architectures (video lesson 5)

NUMA: Non-Uniform Memory Access

El temps d'accés és variable, depèn dels #salts que hagi de fer

Snooping: un sol bus on van totes les peticions dels cores. El bus es converteix en coll d'ampolla. No funciona bé amb més de 8-16 cores

Snoopy-invalidate: Quan escrius algo, invalides els següents perquè no es puguin llegir ni escriure.

Non-broadcast network: sobre la qual es van enviant les peticions.

- Distributed across cores: no centralitzat en un bus. No totes les peticions han d'anar a la mateixa part del Directory.

Bits per mantenir la coherència AMONG numa node:

(#nodes)* [(Mida MP/ Mida línia de MP) * (2+Presència bits/línia)]

Directory-based coherence

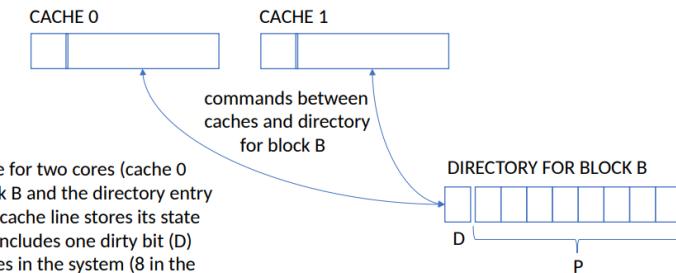
- Snooping:
 - Broadcast requests so others see them and to establish ordering
 - Bus becomes bottleneck!**
 - Snooping does not work well with more than 8-16 cores
- Non-broadcast network
 - How do we observe requests we need to see?**
 - How do we order requests to same block?**

Directory

- Distributed across cores
- Each “slice” serves a set of blocks
 - One entry for each block it serves
 - Entry tracks which caches have block (in non-I state)
 - Order of accesses determined by “home” slice
- Caches still have same states

Directory entry

- 1 dirty bit
- 1 bit/cache: present in that cache

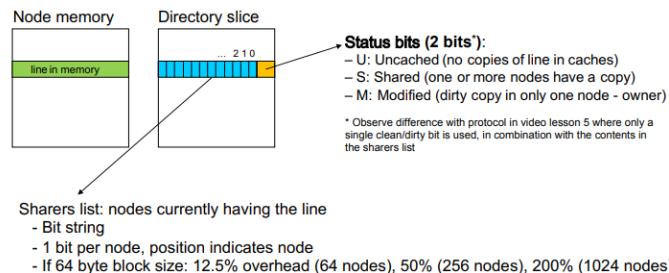


Scaling of the broadcast mechanism

- Snooping schemes broadcast coherence messages to determine the state of a line in the other caches
 - Processor initiating access sends command to ALL other processors (having or not copy of the line)
 - Could be extended to support coherence in small NUMA systems, but does not scale to large number of nodes (excessive coherence traffic)
- Alternative: avoid broadcast by storing information about the status of each line in main memory, in the so called directory divided in slices, one slice per node
 - Each slice of the directory tracks the location of copies in caches of its memory lines
 - Coherence is maintained by point-to-point messages between the nodes

MSU directory-based cache coherency

- One slice of the directory associated to each node memory: one entry per line of memory
 - **Status bits**: they track the state of cache lines in its memory
 - **Sharers list**: tracks the list of remote nodes having a copy of a line. For small-scale systems, implemented as a bit string



- Directory slice is the "centralised" structure that "orders" the accesses to the lines in the associated node

Simplified coherency protocol

Possible commands arriving to home node from local node:

- **RdReq**: asks for copy of line with no intent to modify
- **WrReq**: asks for copy of line with intent to modify
- **UpgrReq**: asks for permission to modify an existing line, invalidating all other copies

As a result of **RdReq** and **WrReq** the home node sends clean copy of line (**Dreply** command to local node). For **UpgrReq** it sends an acknowledgment (**Ack** command) to give permission.

If needed the home node may generate other commands to remote nodes:

- **Fetch**: asks remote (owner) node for a copy of line (**Dreply**)
- **Invalidate**: asks remote (reader) node to invalidate its copy, remote sends confirmation to home (**Ack**)

41/58

Directory-based cache coherency (cont.)

- Who is involved in maintaining coherence of a memory line?
 - **Home node**: node where the line is allocated. It has the directory slice with the information to maintain its coherence.
 - **Local node**: node with the processor accessing the line
 - **Remote nodes**: **Owner** node containing **dirty** copy or **Reader** nodes containing **clean** copies of the line
- But ... how the **home** node for a memory line is decided?
 - **OS managed**, for example using a policy named *first touch*
 - The node that first "touches" a **page** will be the **home** node for all the lines in that page
 - For example, if memory pages are $P = 4$ KBytes and memory lines are $L = 128$ Bytes, then a page will contain $P/L = 32$ consecutive memory lines
 - Unless indicated differently we will assume that the number of memory lines in a page is 1 (i.e. $P = L$)

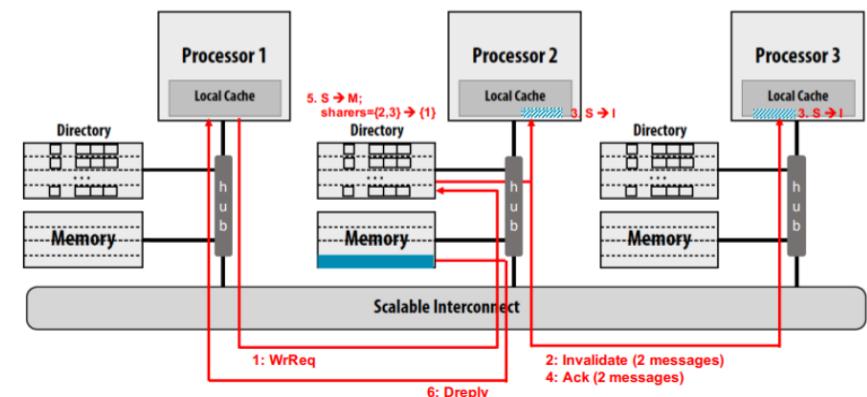
39/58

40/58

Directory-based cache coherency: example

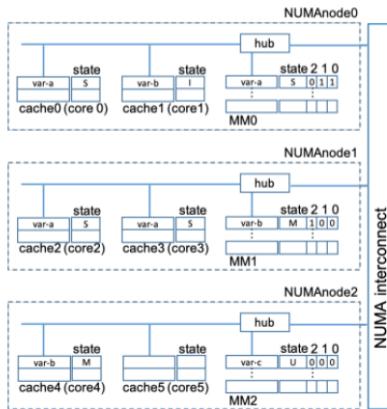
Write miss to clean line with two sharers

- Local node where the miss request originates: processor 1
- Home node where the memory line resides: processor 2
- Copies of line in caches of remote processors 2 and 3



Snooping- and directory-based protocols together!

If nodes have snoopy-based coherence, then the hub becomes an additional agent that interacts with the home (directory) nodes for the cache lines copied in the node



Coherence commands

- Core: PrRd_i and PrWr_i, being I the core number doing the action
- Snoopy: BusRd_j, BusRdX_j, BusUpgr, and Flush_j, being j the snoopy/cache number doing the action
- Hub/directory: RdReq_{i,j}, WrReq_{i,j}, UpgrReq_{i,j}, Dreply_{i,j}, Fetch_{i,j}, Invalidate_{i,j}, Ack_{i,j}, and WriteBack_{i,j}, from NUMA node i to NUMA node j

Line state in cache

- M (Modified), S (Shared), I (Invalid)

Line state in main memory

- M (Modified), S (Shared), U (Uncached)

Data sharing and initialization

```
#pragma omp parallel num_threads(4)
{
    int myid = omp_get_thread_num();
    int BS = 128 / omp_get_num_threads();
    for (int i=myid*BS; i<(myid+1)*BS; i++)
        b[i] = foo1(a[i]);
    for (int i=myid*BS; i<(myid+1)*BS; i++)
        a[i] = foo2(b[i]);
}
```

M ₀			
0.127			
P ₀ @ M ₀	P ₁ @ M ₁	P ₂ @ M ₂	P ₃ @ M ₃
for1	0.31	32..63	64..95
for2	0.31	32..63	64..95

x.y No coherence traffic
x.y Coherence traffic

M ₀	M ₁	M ₂	M ₃
P ₀ @ M ₀	P ₁ @ M ₁	P ₂ @ M ₂	P ₃ @ M ₃
for1	0..31	32..63	64..95
for2	0..31	32..63	64..95

x.y No coherence traffic
x.y Coherence traffic

How are synchronizations implemented?

Data sharing and initialization

- True and false sharing have now a much higher penalty
- What may be wrong with data initialization? Be aware of "first touch"

```
for (int i=0; i<128; i++) {
    a[i] = random();
    b[i] = random();
}
```

Vectors a and b are allocated in a single node of the NUMA system, as follows

M ₀
0..127

```
#pragma omp parallel num_threads(4)
{
    int myid = omp_get_thread_num();
    int BS = 128 / omp_get_num_threads();
    for (int i=myid*BS; i<(myid+1)*BS; i++) {
        a[i] = random();
        b[i] = random();
    }
}
```

Vectors a and b are distributed across the memories of the NUMA system, as follows

M ₀	M ₁	M ₂	M ₃
0..31	32..63	64..95	96..127

```
#pragma omp atomic
var += non_protected_func();
```

```
#pragma omp critical
{
    // exclusive access
}
```

```
omp_set_lock(&lock_var);
// exclusive access
omp_unset_lock(&lock_var);
```

- In fact, entry to and exit from **critical** is the same as **omp_set_lock** and **omp_unset_lock**, respectively, but using an implicit (hidden) **omp_lock_t** variable
 - atomic** could also be implemented with **omp_lock_t**, but usually there is much better support at the architecture level (see later ...)
 - test-and-set**: read value in location and set to 1
- Example: test-and-set based lock implementation

```
set_lock: t&s r2, flag
bnez r2, set_lock      // already locked?
...
unset_lock: st flag, #0          // free lock
```

INPUT BLOCK GEOMETRIC DATA DECOMPOSITION:

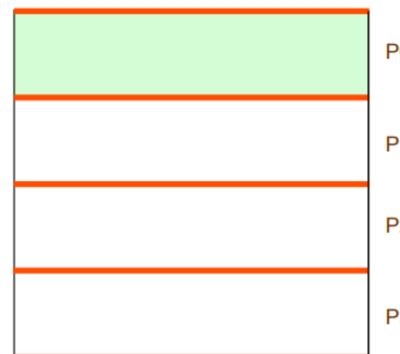
Requereix sincronització en l'update del output del vector del qual s'estigui fent la DD. Per reduir aquesta sync a 1 (només al final), s'ha d'usar una variable temporal tmp on anar acomulant els resultats dins dels inner loops: **scalar replacement**.

- Declarar com a private les variables que calgui // **#pragma omp parallel private(var)**
- Variables necessaries:
 - id = omp_get_thread_num();
 - nt = omp_get_num_threads();
 - bs = C/nt;
 - remainder = C%nt;
 - extra = id < remainder, extraprev = extra? id : remainder;
 - lowerb = id*bs + extraprev
 - upperb = lowerb + bs + extra
- for d'accés normal, i = 0, i < R ++i // j= lowerb, j < upperb, ++j i abans d'actualitzar vector **#pragma omp atomic**

Geometric Block data decomposition INPUT

```
omp_lock_t Locks[256];
for (i = 0; i < 256; i++)
    omp_init_lock(&Locks[i]);
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int BS = n / howmany;
    int rest = n % howmany;
    int start = myid * BS + ((rest > myid) ? myid : rest);
    int end = myid * BS + (rest > myid);

    for (i = start; i < min(end, n); i++) {
        index = ...;
        omp_set_lock(&Locks[i]);
        ...
        omp_set_lock(&Locks[i]);
    }
}
for (i = 0; i < 256; i++)
    omp_destroy_lock(&Locks[i]);
```



Geometric Block data decomposition OUTPUT

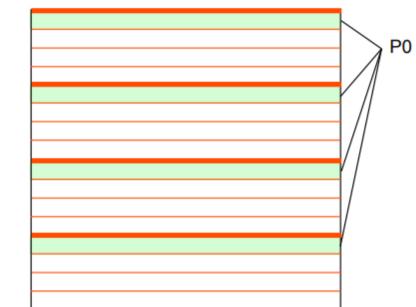
```
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int BS = n / howmany;
    int rest = n % howmany;
    int start = myid * BS + ((rest > myid) ? myid : rest);
    int end = myid * BS + (rest > myid);

    for (i = 0; i < n; i++) {
        index = ...;
        if (index >= start && index < end)
            ...
    }
}
```

Geometric Cyclic data decomposition INPUT

```
omp_lock_t Locks[256];
for (i = 0; i < 256; i++)
    omp_init_lock(&Locks[i]);
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i = myid; i < n; i += howmany) {
        index = ...;
        omp_set_lock(&Locks[i]);
        ...
        omp_set_lock(&Locks[i]);
    }
}
```



Geometric Cyclic data decomposition OUTPUT

```
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (i = 0; i < n; i++) {
        index = ...;
        if (index % howmany == myid)
            ...
    }
}
```

CYCLIC DATA DECOMPOSITION, by ROWS



```
#pragma omp parallel private (i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=my_id; i<N; i+= howmany)
        for (int j=0; j<N; j++)
            ... m[i][j] ... // Input and/or Output
    ...
}
```

FUNCIO MIN

```
#define min(a,b) ( (a) > (b) ? (b) : (a) )
```

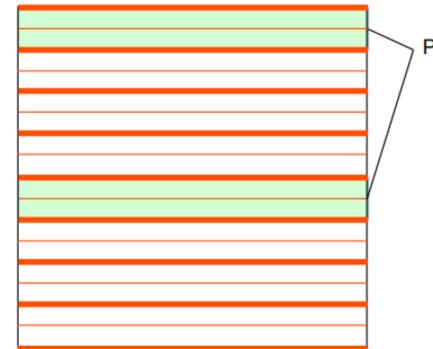
Geometric Block-Cyclic data decomposition INPUT

```
omp_lock_t Locks[256];

for (i = 0; i < 256; i++)
    omp_init_lock(&Locks[i]);

#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();
    int BS = n / howmany;
    int rest = n % howmany;
    int start = myid * BS;
    int step = howmany * BS;

    for (int ii = start; ii < n; ii += step) {
        for (i = ii; i < min(ii + BS, n); i++) {
            index = ...;
            omp_set_lock(&Locks[i]);
            ...
            omp_set_lock(&Locks[i]);
        }
    }
}
```



Geometric Block-Cyclic data decomposition OUTPUT

```
#pragma omp parallel private (index, i)
{
    int myid = omp_get_thread_num();
    int howmany = omp_get_num_threads();

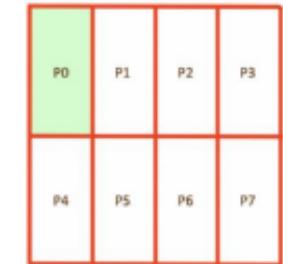
    for (i = 0; i < n; i++) {
        index = ...;
        if ((index/BS) % howmany == myid)
            ...
    }
}
```

OUTPUT BLOCK-CYCLIC GEOMETRIC DATA DECOMPOSITION:

- Definir número d'elements per línia de cache
- #pramgma omp parallel private(variables)
- Variables que necessitem: id, nt, BS = num_elements_per_cache_line, loweb = id*BS, step = nt*BS
- For per fer salts de blocs (ii = loweb, ii < R; ii+=step)
 - For per navegar per cada bloc (i = ii, i < min(R, ii+BS), i++)
 - codi del for normal amb scalar replacement

2D Block / Block data decomposition

```
#pragma omp parallel private (i, j)
{
    int my_i = omp_get_thread_num()/4;
    int my_j = omp_get_num_threads()%4;
    int BSi = N/2;
    int BSj = N/4;
    int i_start = my_i * BSi;
    int i_end = i_start + BSi;
    int j_start = my_j * BSj;
    int j_end = j_start + BSj;
    for (int i=i_start; i<i_end; i++)
        for (int j=j_start; j<j_end; j++)
    ...
}
```



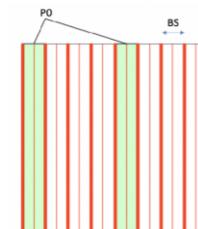
Example: matrix multiply using implicit tasks (3)

Let's reduce the load unbalance to 1 iteration at most ...

```
void matmul (double C[MATSIZE][MATSIZE],
             double A[MATSIZE][MATSIZE],
             double B[MATSIZE][MATSIZE]) {
    int i, j, k;

#pragma omp parallel
    {
        int myid = omp_get_thread_num();
        int howmany = omp_get_num_threads();
        int i_start = myid * (MATSIZE/howmany);
        int i_end = i_start + (MATSIZE/howmany);
        int rem = MATSIZE % howmany;
        if (rem != 0) {
            if (myid < rem) {
                i_start += myid;
                i_end += (myid+1);
            } else {
                i_start += rem;
                i_end += rem;
            }
        }
        ...
    }
}
```

BLOCK-CYCLIC DATA DECOMPOSITION, by COLUMNS



```
#pragma omp parallel private (i, j)
{
    int my_id = omp_get_thread_num();
    int howmany = omp_get_num_threads();

    for (int i=0; i<N; i++)
        for (int jj=my_id*BS; jj<N; jj+=howmany*BS)
            for (int j=jj; j<jj+BS; j++)
                ... m[i][j] ... // Input
                ... // and/or Output
    ...
}
```