


I multi processori

Prof. Alberto Borghese
Dipartimento di Scienze dell'Informazione
borghese@dsi.unimi.it

Università degli Studi di Milano

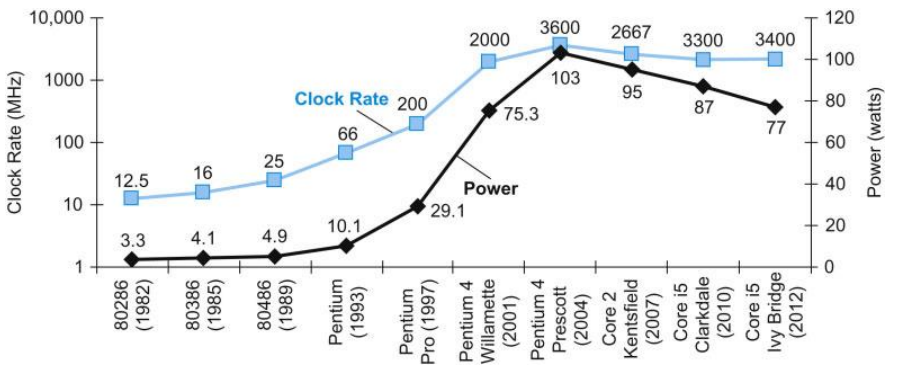
Patterson, sezione 1.5, 1.6, 2.17, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.9, 6.10


Sommario

Le architetture multi-processore

Le gerarchie di memoria

La barriera dell'energia



Power = Capacitive load × Voltage^2 × Frequency switched

E' aumentato di 30 volte

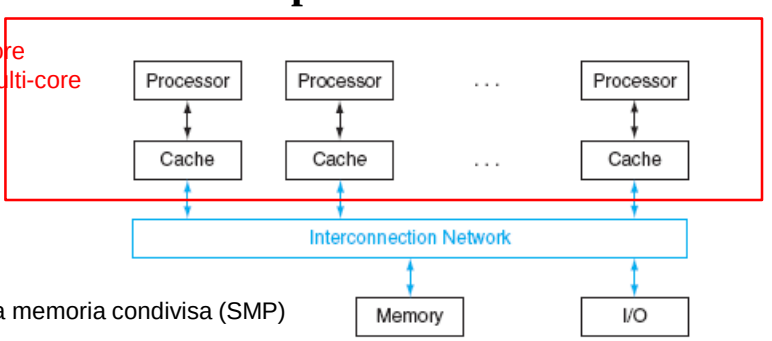
carico delle porte CMOS

È passata da 5V ad 1V

È aumentata di 200 volte in 20 anni

I multi-processori

Multi-processore
Processore multi-core



Processore a memoria condivisa (SMP)

Possibili utilizzo:
- Più task, ogni task su un processore diverso
- Stesso task, esecuzione parallela sui diversi processor (core)

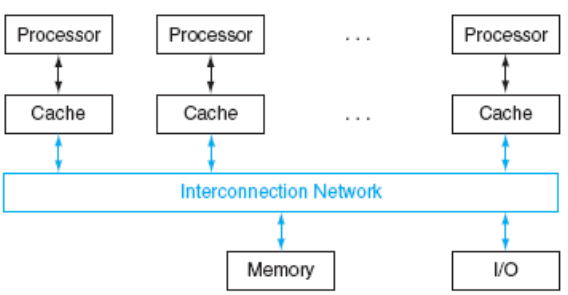
Esecuzione parallela il quadro generale

		Software	
		Sequential	Concurrent
Hardware	Serial (pipeline)	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel (pipeline)	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)

Alcuni task sono naturalmente paralleli (programmazione concorrente)

Altri task sono seriali e occorre capire come parallelizzarli in modo efficiente (moltiplicazione tra matrici)
Non è neppure facile parallelizzare task concorrenti in modo tale che le prestazioni aumentino con l'aumentare dei core

I multi-processori

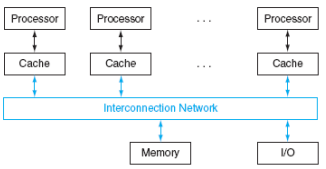


Chiama un parallelismo esplicito (la pipe-line multi-scalare è una forma di parallelismo implicito)

Un programma deve essere:
•Corretto
•Risolvere un problema importante (di grandi dimensioni)
•**Veloce** (altrimenti è inutile parallelizzare)

Esecuzione parallela

Esecuzione parallela richiede un Overhead:
- Scheduling.
- Coordinamento.



Scheduling:
•Analisi del carico di lavoro globale (scheduler).
•Partizionamento del carico sui diversi processori (scheduler).
•Coordinamento nella raccolta dei risultati (e.g. reorder station).

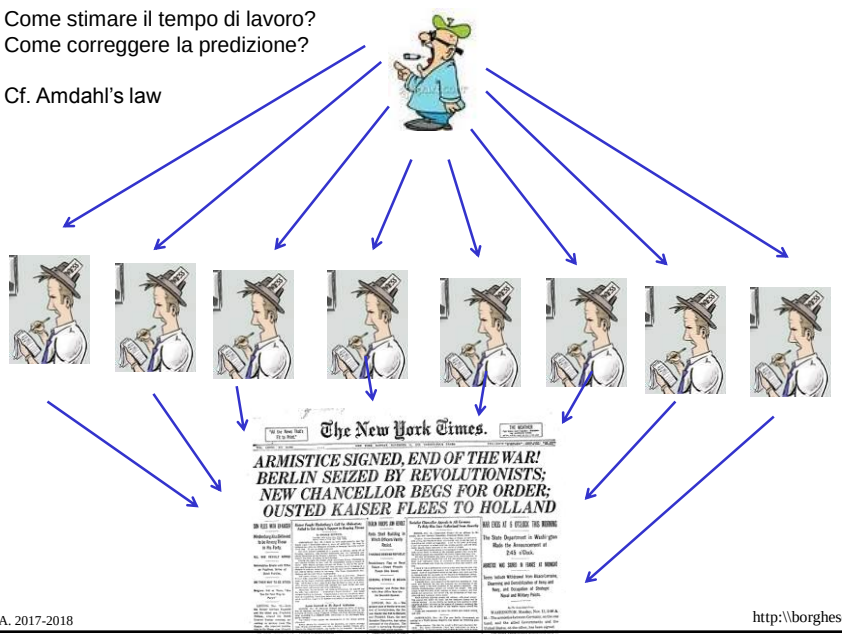
Lo scheduling può essere più (Pentium 4) o meno (CUDA) performante. Sempre più importante è il lavoro del programmatore / compilatore.

Infatti. Non si può "perdere" troppo tempo nello scheduling e/o nella compilazione.

Un esempio

Come stimare il tempo di lavoro?
Come correggere la predizione?

Cf. Amdahl's law



Parallelizzazione dell'esecuzione - I

- Somma di 128,000 elementi di un vettore (N=128,000) su un'architettura seriale

```
/* Execute sequentially - 128.000 steps */
sum = 0;
for (i = 0; i < 128000; i++)
        sum = sum + A[i]; /* sum the assigned areas*/
```

- Identifichiamo P lotti (batch) che possono essere elaborati in parallelo (non hanno dipendenze)

```
/* Execute sequentially - 128.000 steps = M * P = 1000 * 128  */
for (k=0; k < 127; k++)       // for each of the P batches
{   sum[k] = 0;
    for (i = k*1000;i<(k+1)*1000;i=i+1) // for each of the M values
                                        // inside one batch
    {   sum[k] = sum[k] + A[i];         // sum the assigned areas
    }
}
```

Il numero di passi di esecuzione non cambia, ma possiamo parallelizzare l'esecuzione

Parallelizzazione dell'esecuzione: divide - II

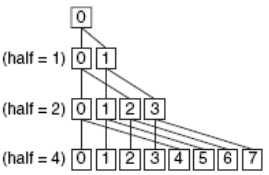
- Somma di 128,000 numeri (N=128,000) su un'architettura 128-core (P=128).
- Sommo N/P (=1,000) numeri su ciascun processore
  - Partizionamento dei dati in ingresso
  - Stessa memoria fisica. L'accesso dei diversi processori è su blocchi diversi di memoria fisica.

```
/* Execute in parallel on each Pn processor */
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
       sum[Pn] = sum[Pn] + A[i]; /* sum the assigned areas*/
```

- Posso eseguire le somme parziali in 1000 passi invece che in 1000 * 128 = 128,000 passi: il problema scala con il numero dei processori.
- Ottengo P = 128 somme parziali. Per ottenere la somma finale devo sommare tra loro le somme parziali (**riduzione**). Come?

Parallelizzazione dell'esecuzione: reduction - III

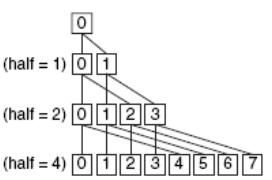
- Sommo i numeri a due a due in modo ricorsivo e gerarchico (**divide and conquer**)




```
half = 128; /* 128 processors, Pn, in multiprocessor*/
repeat
   synch(); /* wait for partial sum completion */
            /* Conditional sum needed when half is even */
   half = half/2; /* dividing line on who sums */
   if (Pn < half)
       sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in Sum[0] */
```


Osservazione

- **Quanto si guadagna?**



- La riduzione sequenziale costa M passi, dove M è il numero di somme parziali.
- La riduzione parallela costa log2(M) nel caso in cui ad ogni processore possa essere assegnato una somma parziale.

- Per M = 128 abbiamo:
  - 128 passi per la somma sequenziale
  - 7 passi per la riduzione parallela.

Inoltre, si guadagna dalla parallelizzazione delle somme: occorre un tempo pari a 1000 somme per sommare 128,000 numeri.

Osservazioni

La parallelizzazione è stata esplicitata.

`half = 128;` consente di scalare con il numero di processori

```
if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

Assegna ai diversi processori il ruolo (accumulatore o semplice memoria)

```
synch(); /* wait for partial sum completion */
```

Sincronizzazione esplicita alla fine di ogni livello di somme parziali

Distribuzione e sincronizzazione sono problematiche già viste nelle pipe-line superscalari dove venivano risolte dall'HW e/o dal compilatore. Qui distribuzione e sincronizzazione vengono eseguite a livello di codice. Potrebbero essere eseguite dai compilatori. Non sono ancora così "smart" per sfruttare appieno il parallelismo ...

OPEN MP

An API for shared memory multiprocessing in C, C++, or Fortran that runs on UNIX and Microsoft platforms. It includes compiler directives, a library, and runtime directives.

Its primary goal is to parallelize loops and to perform reductions.

Opzione utilizzata per il compilatore cc: `cc –fopenmp foo.c`

OpenMP extends C using *pragmas*, which are just commands to the C macro preprocessor like #define and #include.

To set the number of processors we want to use to be 128:

```
#define P 128
#pragma omp parallel num_threads(P)   // We will use P = 128 parallel threads
```

Application to a for cycle

For OpenMP

Fase di accumulazione

```
#pragma omp parallel for   // The loop will be the parallelized for:
for (Pn = 0; Pn < P; Pn += 1)
     for (i=1000*Pn; i < 1000*(Pn+1); i += 1)   // Each thread sums 1000 numbers
        sum[Pn] += A[i];                         //sums the assigned data of A */
```

Fase di riduzione

```
#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
    FinalSum += sum[i]; /* Reduce to a single number */
```

Open MP non ha un debugger potente. Altre API più potenti stanno nascendo.

Come rendere più veloci i calcolatori

Rendere veloce il caso più comune.

Si deve favorire il caso più frequente a discapito del più raro.

Il caso più frequente è spesso il più semplice e può essere quindi reso più veloce del caso infrequente.

Legge di Amdahl

Il miglioramento delle prestazioni globali ottenuto con un miglioramento particolare (e.g. un'istruzione), dipende dalla frazione di tempo in cui il miglioramento era eseguito.

Esempio: Pentium e PentiumPro: a fronte di un raddoppio della frequenza di clock che è passata da 100 a 200 Mhz, si è registrato un aumento delle prestazioni misurate tramite SpecInt di 1,7 volte e di 1,4 volte misurate in SpecFloat.

Corollario della legge di Amdhal

Se un miglioramento è utilizzabile solo per una frazione del tempo di esecuzione complessivo (F_m), allora non è possibile accelerare l'esecuzione più del reciproco di uno meno tale frazione: $Speedup_{globale} < 1/(1-F_m)$.

Definizioni:

1. **$Frazione_{migliorato}$** ($F_m \leq 1$), ovvero la frazione del tempo di calcolo della macchina originale che può essere modificato per avvantaggiarsi dei miglioramenti. Nell'esempio precedente la frazione è 0.90.

$T_m = F_m * T_{old}$
$T_{nm} = (1- F_m) * T_{old}$

2. **$Speedup_{migliorato}$** ($S_m \geq 1$), ovvero il miglioramento ottenuto dal modo di esecuzione più veloce.

$$Speedup_{globale} = T_{old} / T_{new} = T_{old} / (F_m / S_m + (1 - F_m)) * T_{old}$$

Nel precedente esempio questo valore viene fornito nella colonna chiamata Speedup_migliorato (pari a 2).

Esempio numerico

Somma di 10 variabili scalari e somma di una coppia di matrici bidimensionali 12x10

Supponiamo che solo la somma di matrici sia parallelizzabile e che abbiamo a disposizione 40 processori su cui parallelizzare. Ogni operazione di somma costa un tempo *t*.

Qual è lo speed-up?

Il tempo senza parallelizzazione sarà: $10t + (12x10)t = 130\ t$

Il tempo dopo la parallelizzazione sarà: $10t + ((12x10) / 40)t = 13\ t$

Lo speed-up sarà quindi: $130\ t\ /\ 13\ t\ = 10$

La velocità aumenta di 10 volte e non di 40 come ci si poteva aspettare.

Ci sono delle parti di codice non parallelizzate (somma di scalari) che limitano il guadagno.

Modalità di incremento delle prestazioni

Un'architettura scala in modo forte, quando le prestazioni aumentano linearmente con il numero di processori, senza variare le dimensioni del problema.

Un'architettura scala in modo debole, quando il tempo di esecuzione rimane lo stesso, ma la dimensione dei dati cresce linearmente con la dimensione del problema.

Qual è più facile da ottenere?

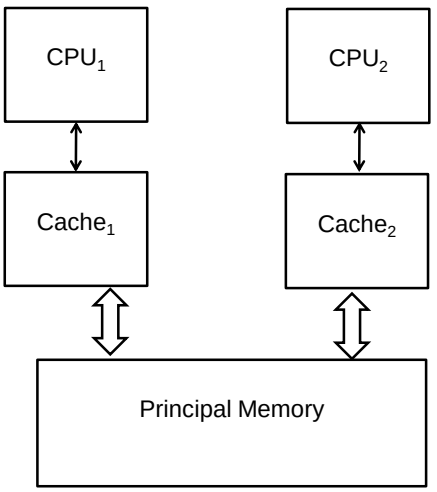
Che tipo di aumento abbiamo ottenuto nell'esempio precendete (somma degli elementi di un vettore)?

Sommario

Le architetture multi-processore

Le gerarchie di memorie

Cache in un'architettura dual-core



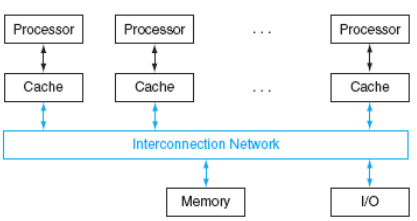
Le cache si parlano attraverso la memoria principale

Più cache ed un'unica memoria principale: **"the view of memory held by two different processors is through their individual caches"**

Esecuzione parallela con una memoria condivisa

La memoria viene suddivisa in memoria condivisibile e memoria non condivisibile o privata del singolo processo. In questo secondo caso la parte corrispondente della memoria principale non può essere utilizzata da altri processi.

- Competizione sui dati (data race).
- Occorre una coordinazione tra i diversi processi nell'accesso alla memoria (sincronizzazione) => **cache coherence**.
- Occorre coordinare la sequenza temporale degli accessi => **consistenza**.



Consistenza mediante lock

- Come garantire la piena proprietà di una linea di memoria?
- **Data race** sulla memoria principale ➔ **Lock** di una cella di memoria e unlock.
- Viene introdotta un'operazione atomica di scambio dei dati tra un registro ed una cella di memoria: nessun altro processsore o processo può inserirsi fino a quando l'operazione atomica non è terminata.
- Viene inserito un meccanismo hardware di blocco di una cella di memoria (lock o lucchetto).
- Viene gestito dal sotto-sistema di controllo della memoria dietro istruzioni della CPU.

Meccanismo di lock

Serve per proteggere la memoria da accessi di altre procedure.

L'accesso ad una cella di memoria viene riservato ad una certa procedura (e ad un certo processore)

```
try:    add $t0,$zero,$s4   # copy exchange value
        ll $t1,0($s1)       # load linked
        addi $t1, $t1, 4    # do something on $t1
        sc $t1,0($s1)       # store conditional the new value of $t1
        beq $t0,$zero,try   # repeat if store fails
```

Si controlla che la load abbia avuto successo controllando il contenuto di $t1 (sc). Se la procedura non è riuscita a leggere perchè c'era un blocco sulla cella di memoria, $t1 conterrà 0 e si riprova. La sc scambia un dato con il flag del risultato dell'operazione sulla memoria.

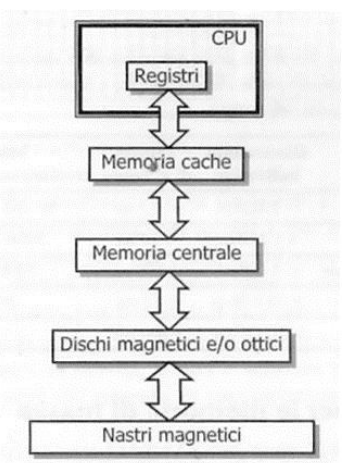
Protezione se la cella di memoria è riservata da un altro processo.

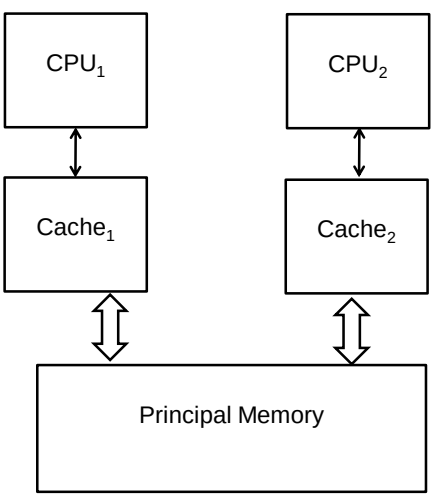
La condivisione della memoria

Istruzioni corrispondenti al meccanismo hardware del **lock**:

- *load linked* (load collegata, `ll`) + *store conditional* (store condizionata, `sc`) che vengono utilizzate in sequenza.
- Se il contenuto di una locazione di memoria letta dalla load linked venisse alterato prima che la store condizionata abbia salvato in quella cella di memoria il dato, l'istruzione di store condizionata fallisce.
- L'istruzione di store condizionata ha quindi due funzioni: salva il contenuto di un registro in memoria *ed* imposta il contenuto di quel registro a 1 se la scrittura ha avuto successo e a 0 se invece è fallita.
- L'istruzione di load linked restituisce il valore letto e l'istruzione di store condizionata restituisce 1 solamente se la scrittura ha avuto successo.
- Controllo del contenuto del registro target associato all'istruzione di store condizionata.

Coerenza in un'architettura





Più cache ed un'unica memoria principale: **"the view of memory held by two different processors is through their individual caches". Cache, Memoria prinicipale e le alter memorie dovrebbero contenere lo stesso dato allo stesso indirizzo.**

Write-through

Quando c'è register spilling il dato viene scritto in cache e si verifica un **disallineamento** della memoria principale e della cache.

Write-through. Scrittura in cache e contemporaneamente in RAM.

Write_buffer per liberare la CPU (DEC 3100)

Chi libera il write buffer?

Cosa succede se il write buffer è pieno?

Sincronizzazione tra contenuto della Memoria Principale (che può essere letto anche da I/O e da altri processori) e Cache.

Svantaggio: traffico intenso sul bus per trasferimenti di dati in memoria.

Ogni volta che si ha una miss, occorre ricaricare in cache l'intera linea, cioè tutte le parole del microblocco della memoria principale.

Write-back

Write-back. Scrittura ritardata. Scrivo quando devo scaricare il blocco di cache.

Utilizzo un bit di flag: UPDATE, che viene settato quando altero il contenuto del blocco. Questo flag si chiama anche "dirty bit".

Vantaggiosa con cache n-associative.

Alla Memoria Principale trasferisco il blocco quando devo scrivere da CPU a cache (è equivalente al register spilling).

Contenuto della memoria principale e della cache può non essere allineato.

E' vantaggiosa per le memorie virtuali: il costo di trasferimento di una linea dalla memoria principale è molto inferiore all'accesso alla memoria. Si cerca quindi di ottimizzare il trasferimento.

Coerenza e consistenza

Coerenza: determina se il dato letto dalla cache è lo stesso di quello contenuto nella memoria principale.
Consistenza: quando un dato può essere letto dopo una scrittura.

Esempio su una memoria Write-through

Time step	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A stores 1 into X	1	0	1

Cosa succederebbe se la cache fosse "Write-back"?

Consistenza: Need for a serialization of the writes (cf. Commit unit in the CPU)

Bus snooping

Mantenimento dell'informazione di cache coerente tra varie cache (sistemi multi-processori).

Elemento chiave è il protocollo per il **tracking dello stato** di ciascuna linea di ciascuna cache.

In cache, oltre al TAG e al bit di validità viene memorizzato lo stato della linea.

Ogni trasferimento dalla Memoria Principale viene monitorato da tutte le cache.

Il controller della cache monitora il bus indirizzi + segnale di controllo read della memoria e legge l'indirizzo della memoria principale delle richieste di tutte le altre cache.

Se l'indirizzo corrisponde all'indirizzo dei dati contenuti in una delle linee della cache, viene invalidato il contenuto della linea.

Quando funziona?

Write invalidate protocol

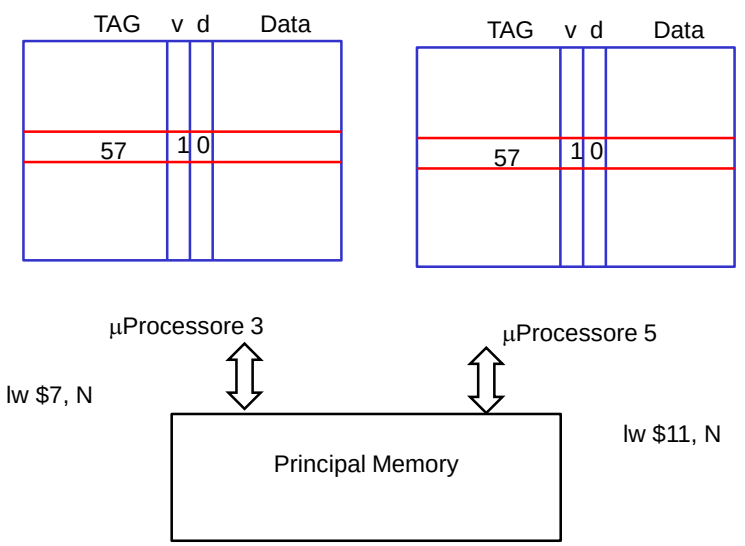
In questo caso si mira a garantire a ciascuna cache la piena proprietà di un dato. *"Exclusive access ensures that no other readable or writable copies of an item exist when the write occurs: all other cached copies of the item are invalidated".*

Il controller della memoria monitora il bus indirizzi (**snooping**). Quando si verifica una write, cerca se il dato nell'indirizzo di scrittura è presente nelle altre cache.
Invalida il blocco nelle cache in cui il dato sra stato copiato.
Non c'è un'informazione di "blocco aggiornato" centralizzata, ma è distribuita sulle varie cache.

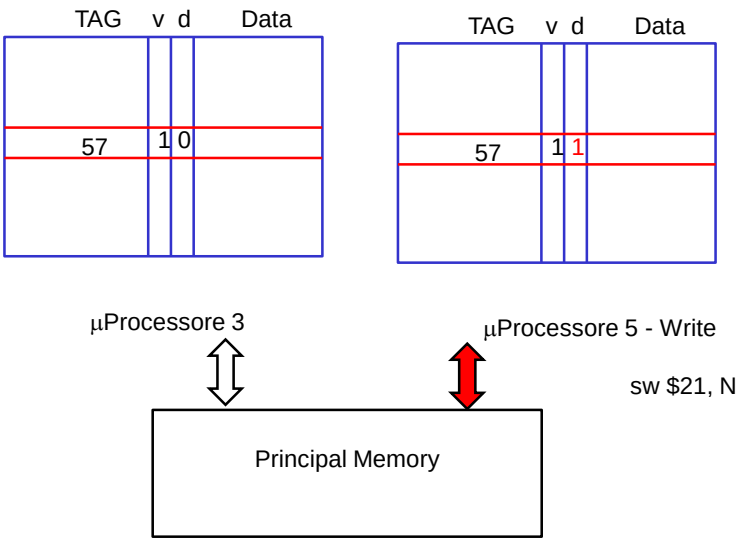
Invalidates cache of B

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

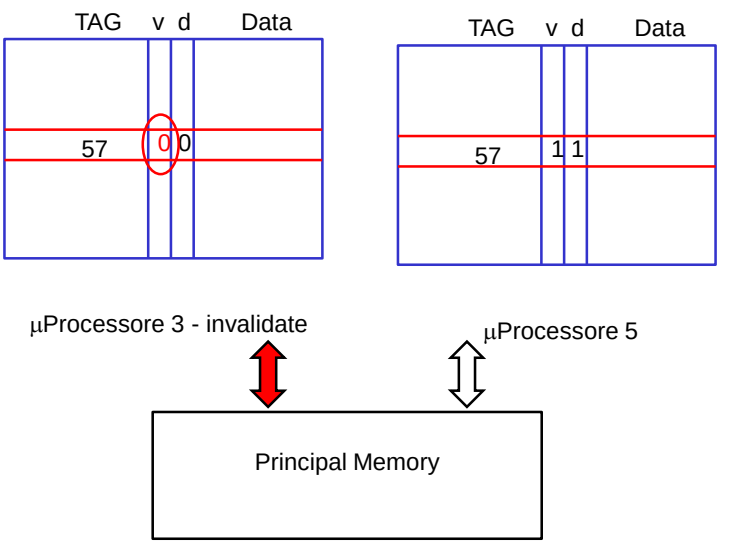
Come agisce il sottosistema di memoria - I



Come agisce il sottosistema di memoria - II



Come agisce il sottosistema di memoria - III



Altri meccanismi per la cache coherence

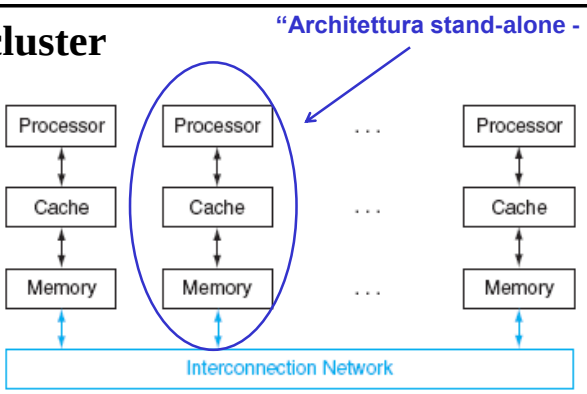
Hardware trasparency.
Circuito addizionale attivato ad ogni scrittura della Memoria Principale.
Copia la parola aggiornata in tutte le cache che contengono quella parola.

Noncachable memory.
Viene definita un'area di memoria condivisa, che non deve passare per la cache.

NB Blocchi di grandi dimensioni possono provocare il **false sharing**. Due programmi che stanno girando su due CPU diverse richiedono due variabili diverse ma che ricadono nello stesso blocco della cache (a mappatura diretta). Alla cache il blocco appare condiviso e si innesca il meccanismo di invalidazione.

Soluzione: allocare le due variabili in memoria principale in modo tale che cadano in due blocchi diversi. I compilatori (ed i programmatori) sono incaricati di risolvere questo problema.

I cluster



Synchronization through **message passing** multiprocessors (sender – receiver).

Modalità tipica delle architetture SW concorrenti (Robot: AIBO Sony, File servers)
Ogni architettura ha la sua memoria, il suo SO. La rete di interconnessione non può essere così veloce come quella dei multi-processori.

E' un'architettura molto più robusta ai guasti, facile da espandere.
I messaggi devono essere identificati in anticipo in modo esplicito.

Massive parallelism -> data center -> Grid computing (SETI@home, 257 TeraFLOPS-> Cloud computing.

Sommario

Le architetture multi-processore

La parallelizzazione del codice