I multi processori

Prof. Alberto Borghese
Dipartimento di Informatica
alberto.borghese@unimi.it

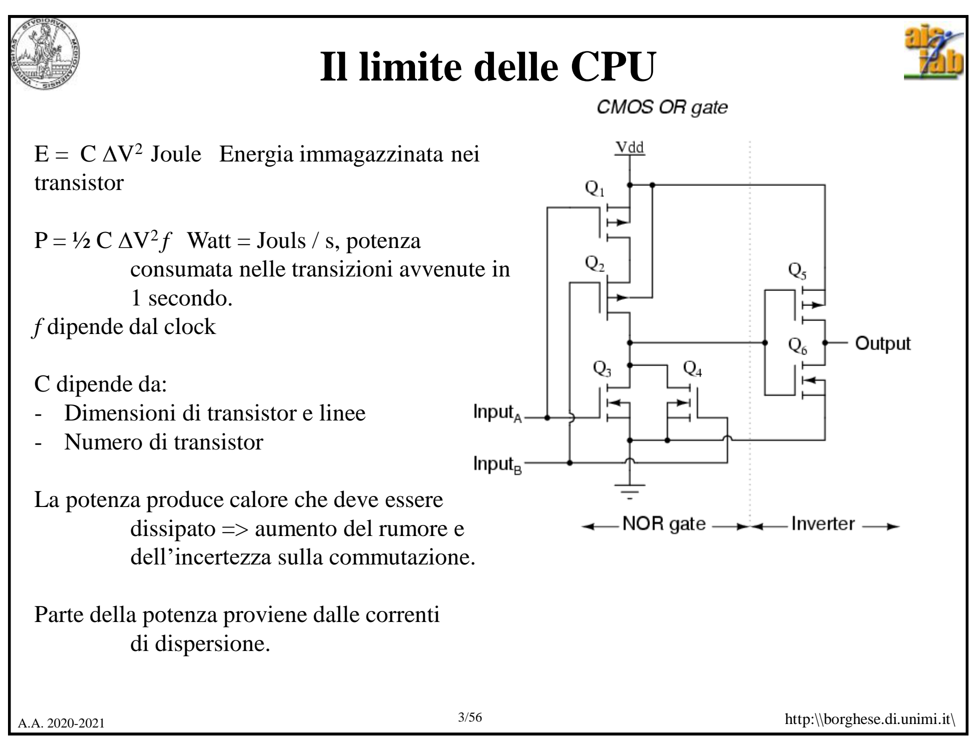
Università degli Studi di Milano

Patterson, sezioni 1,7 1.8, 6.1, 6.5, 2.11, 5.3, 5.10, 6.6
(approfondimenti in Appendice C)

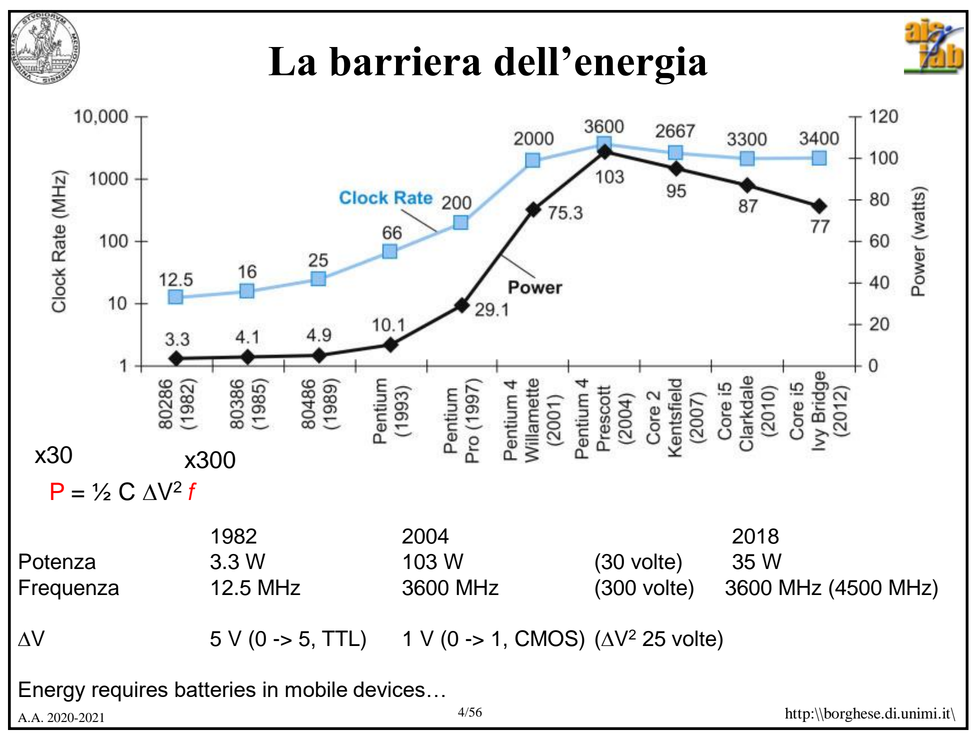
Sommario

Le architetture multi-processore

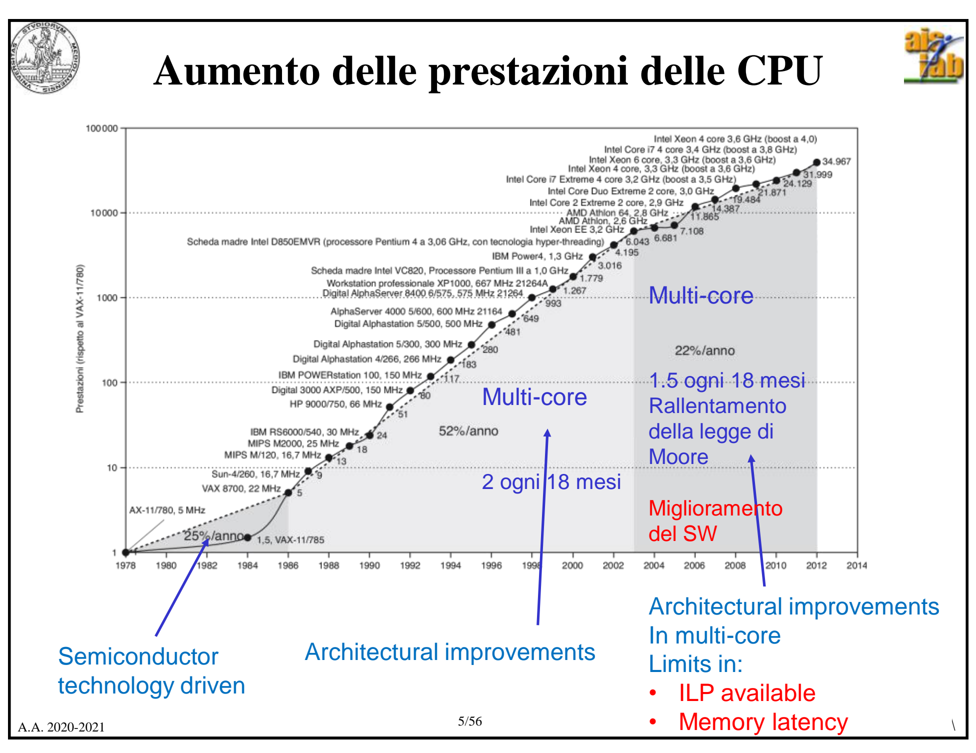
Gestione della memoria

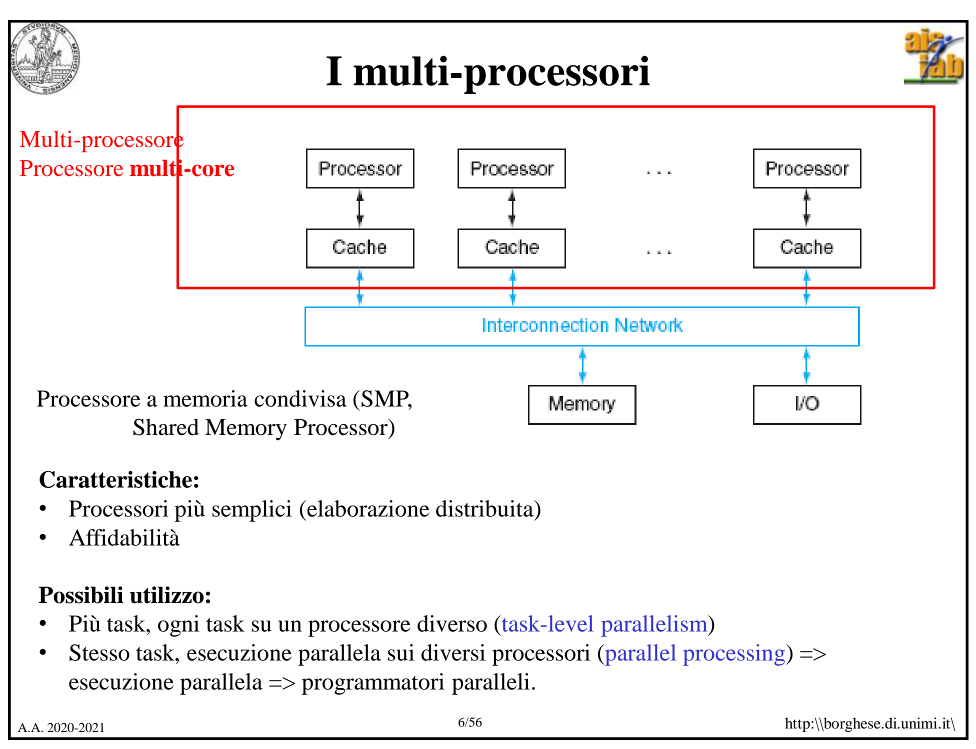


Il limite delle CPU
$E = C \Delta V^2$ Joule Energia immagazzinata nei transistor
$P = ½ C \Delta V^2 f$ Watt = Jouls / s, potenza consumata nelle transizioni avvenute in 1 secondo.
f dipende dal clock
C dipende da:
- Dimensioni di transistor e linee
- Numero di transistor
La potenza produce calore che deve essere dissipato => aumento del rumore e dell'incertezza sulla commutazione.
Parte della potenza proviene dalle correnti di dispersione.
CMOS OR gate
Vdd
Q1
Q2
Q3
Q4
Q5
Q6
Output
InputA
InputB
NOR gate
Inverter
A.A. 2020-2021
3/56
http:\\borghese.di.unimi.it\



La barriera dell'energia
Clock Rate (MHz)
Power (watts)
Clock Rate
Power
12.5
16
25
66
200
2000
3600
2667
3300
3400
3.3
4.1
4.9
10.1
29.1
75.3
103
95
87
77
80286 (1982)
80386 (1985)
80486 (1989)
Pentium (1993)
Pentium Pro (1997)
Pentium 4 Willamette (2001)
Pentium 4 Prescott (2004)
Core 2 Kentsfield (2007)
Core i5 Clarkdale (2010)
Core i5 Ivy Bridge (2012)
x30
x300
$P = ½ C \Delta V^2 f$
Potenza 1982 3.3 W; 2004 103 W (30 volte); 2018 35 W
Frequenza 1982 12.5 MHz; 2004 3600 MHz (300 volte); 2018 3600 MHz (4500 MHz)
ΔV 5 V (0 -> 5, TTL) 1 V (0 -> 1, CMOS) (ΔV² 25 volte)
Energy requires batteries in mobile devices…
A.A. 2020-2021
4/56
http:\\borghese.di.unimi.it\


Aumento delle prestazioni delle CPU
Multi-core
22%/anno
1.5 ogni 18 mesi
Rallentamento
della legge di
Moore
Multi-core
52%/anno
2 ogni 18 mesi
Miglioramento
del SW
25%/anno
Semiconductor
technology driven
Architectural improvements
Architectural improvements
In multi-core
Limits in:
• ILP available
• Memory latency
A.A. 2020-2021
5/56


I multi-processori
Multi-processore
Processore multi-core
Processor
Processor
Processor
Cache
Cache
Cache
Interconnection Network
Memory
I/O
Processore a memoria condivisa (SMP,
Shared Memory Processor)
Caratteristiche:
• Processori più semplici (elaborazione distribuita)
• Affidabilità
Possibili utilizzo:
• Più task, ogni task su un processore diverso (task-level parallelism)
• Stesso task, esecuzione parallela sui diversi processori (parallel processing) => esecuzione parallela => programmatori paralleli.
A.A. 2020-2021
6/56
http:\\borghese.di.unimi.it\

Esecuzione parallela il quadro generale

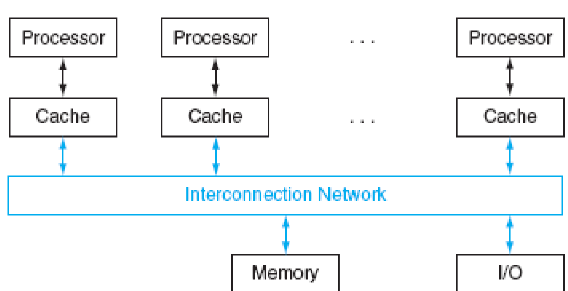
		Software	
		Sequential	**Concurrent**
Hardware	Serial (pipeline)	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel (pipeline)	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista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

Alcuni task sono naturalmente paralleli (programmazione concorrente) => possono essere eseguiti sequenzialmente o in time sharing su un'architettura monoprocessore.

Altri task sono seriali e occorre capire come parallelizzarli in modo efficiente (moltiplicazione tra matrici, e.g. blocking esteso ai multi-core).

Non è neppure facile parallelizzare task concorrenti in modo tale che le prestazioni aumentino con l'aumentare del numero di core.

I multi-processori

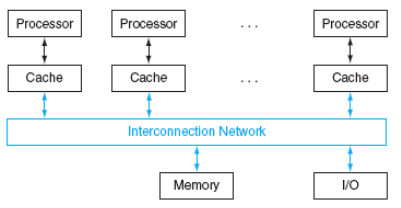


Chiama un parallelismo esplicito (la pipe-line multi-scalare è una forma di parallelismo implicito)

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 e sincronizzazione):
- Partizionamento e invio dei carichi di lavoro.
- Coordinamento.



Scheduling:
•Analisi del carico di lavoro globale (scheduler).
•Partizionamento del carico sui diversi processori (scheduler, **bilanciamento dei carichi**).

Sincronizzazione:
•Coordinamento nell'esecuzione e nella raccolta dei risultati (e.g. reorder station).

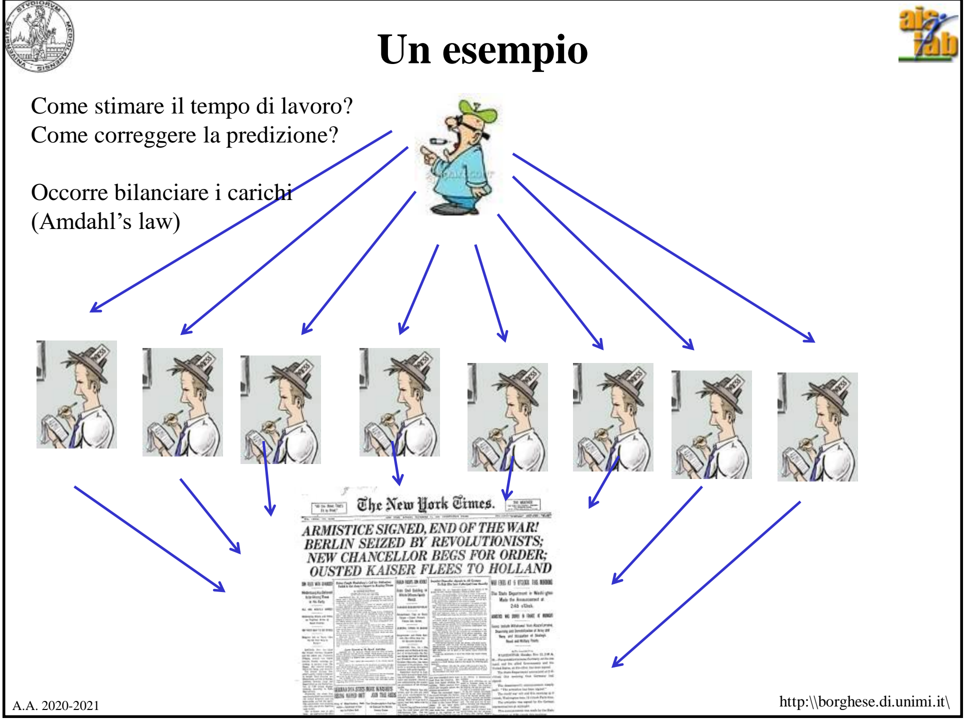
Lo scheduling può essere più (Pentium 4) o meno (CUDA) performante. Sempre più importante è il lavoro del programmatore / compilatore.

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?

Occorre bilanciare i carichi
(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 < 128.000; i++)
     sum = sum + A[i];      /* sum the assigned element */
```

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

```
/* Execute sequentially - 128.000 steps = M * P = 1.000 * 128  */
for (k=0; k < 128; k++)        // for each k batch of the P=128 batches
{   sum[k] = 0;
    for (i = k*1.000;i<(k+1)*1.000;i=i+1) // for each of the M=1.0000
 {      sum[k] = sum[k] + A[i];         // elements inside the k-th batch
 }                                    // sum the assigned element
}
for (k=1; k<128; k++) sum[0]+=sum[k];
```

Il numero di passi di esecuzione non cambia, ma abbiamo creato una gerarchia e 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 = 1.000*Pn; i < 1.000*(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 1.000 * 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**).

Indice del vettore da 0 a 127.999

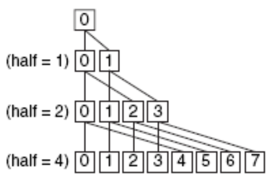
sum[0] = A[0] + A[1] + A[2] + ……… + A[999];
+
sum[N/2+0] = A[64*1.000+0] + A[64*1.000 + 1] + … A[64*1.000 + 999];
=>
sum[0]

sum[1] = A[1*1.000] + A[1*1.000+1] + A[1*1.000+2] + ... + A[1*1.000+999];
+
sum[N/2+1] = A[65*1.000+0] + A[65*1.000 + 1] + ….. A[65*1.000 + 999];
=>
sum[1]

sum[N/2-1] = A[63*1.000] + A[63*1.000+1] + A[63*1.000+2] + … + A[63*1.000+999];
+
sum[N-1] = A[127*1.000] + A[127*1.000+1] + A[127*1.000+2] + … + A[127*1.000+999];
=>
sum[N/2-1]

Parallelizzazione dell’esecuzione: reduction - III

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

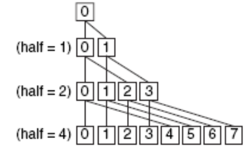



```
half = 128; /* 128 processors, Pn, in multiprocessor*/
repeat
   synch(); /* wait for partial sum completion */
            /* Conditional sum needed when half is even */
   half = half/2; /* dividing line on who sums */
   if (Pn < half)
       sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);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

Osservazione

```
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] */
```

- **Quanto si guadagna in tempo?**
- La riduzione sequenziale richiede M passi, dove M è il numero di somme parziali.
- La riduzione parallela richiede 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 lotti di 1,000 numeri.

- Per M = 1 processore abbiamo processori abbiamo 128,000 passi.
- Per M = 128 processori con riduzione sequenziale, abbiamo 1000 passi per la somma + 128 passi per la riduzione = 1.128 passi.
- Per M = 128 processori con riduzione parallela, abbiamo 1000 passi per la somma + 7 passi per la riduzione = 1007 passi (circa un fattore pari a 128, il numero di processori).

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)

Ma:

`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 (https://www.openmp.org/). It is managed by he nonprofit technology consortium *OpenMP Architecture Review Board* (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors.

The core elements of OpenMP are the constructs for **thread creation**, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables. **Each thread can be conveniently mapped to one processor or to a group of parallel execution lanes of super-scalar processors.** OMP uses the abstract view of threads.

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
```

For OpenMP to a for cycle

Fase di accumulazione

```
#pragma omp parallel for   // The loop will be the parallelized for:
for (Pn = 0; Pn < P; Pn += 1)
     for (i=1.000*Pn; i < 1.000*(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 */
```

Alcune criticalità del supporto alla parallelizzaione

Since OpenMP is a **shared memory** programming model, most variables in OpenMP code are visible to **all threads** by default (they are **shared** by threads). But sometimes private variables are necessary to avoid a **race condition** (competing for the same memory cell).

Moreover, there is a need to **pass values between the sequential part and the parallel region** (the code block executed inside one thread and that executed in other threads), so data environment management is introduced as ***data sharing attribute clauses*** by appending them to the OpenMP directive.

Clausole di coerenza della memoria

- **Shared:** the data declared outside a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, **all variables in the work sharing region are shared except the loop iteration counter**.
- **Private:** the data declared within a parallel region is **private to each thread**, which means each thread will have a local copy and use it as a temporary variable. Viene bloccata l'**A private variable is not initialized** (cf. Firstprivate) **and the value is not maintained for use outside the parallel region.** By default, the loop iteration counters in the OpenMP loop constructs are private (**the data are not modified by other threads, but are not modified by the same thread too except the iteration counters**).
- Firstprivate: like private except initialized to original value (that in MM).
- Lastprivate: like private except original value is updated after construct (**modification allowed at the end of execution**).
- Reduction: a safe way of joining work from all threads after construct.

Clausole di consistenza della memoria (sincronizzazione)

- **Critical:** the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race condition.
- **Atomic:** the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic (on that particular memory cell); only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.
- **Ordered:** the structured block is executed in the order in which iterations would be executed in a sequential loop (forces the sequentiality of reads / writes).
- **Barrier:** each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end (e.g. somma degli elementi di un vettore).
- **Nowait:** specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

Multi-threading processors

Each thread can be conveniently mapped to one processor or to a group of parallel execution lanes of super-scalar processors: **Simultanous multi-threading**.

Vantaggi, Se ci si limita all’esecuzione di un thread solo, si ha:
- Un parallelismo a livello di istruzioni limitato (non riesco a riempire gli issue slots)
- Stalli dovuti a miss che non si possono evitare (latenze lunghe dovute a miss penalty)

GPU

Sviluppo della grafica a partire dagli anni ’90. Workstation grafiche (Silicon Graphics). Calcolo intensive, parallelo (e.g. texture mapping, rendering, trasformazioni geometriche….). **Processori specializzati per la pipe-line grafica. Graphical Processing Units.**

CUDA (Compute Unified Device Architecture) parallel computing platform and programming model developed by NVIDIA. Consente di scrivere in modo relativamente facile programmi che possano essere eseguiti su GPU. **GPU come co-processore per calcolo vettoriale**.

Caratteristiche:

- Una GPU è un co-processore della CPU. Non deve fare tutto. E’ specializzata nel calcolo massivo parallelo. Gli altri task sono lasciati alla CPU.
- Trattamento di quantità massicce di dati (Terabyte). Velocità di calcolo elevate.
- Per nascondere la **latenza della memoria**, la GPU utilizza **multi-threading massiccio**, la CPU su una costellazione di tecniche tra cui la speculazione sulla gerarchia di memoria.
- La memoria della GPU è orientate alla larghezza di banda (i.e. interleaving, ampiezza del bus), la memoria della CPU è orientate alla riduzione della latenza.
- La GPU è basata su un gran numero di processori semplici paralleli (architettura MIMD).
- La GPU deve trasferire i dati da / per la memoria della CPU.

Struttura delle GPU

	Core i7 960	GTX 280 “Tesla”	GTX 480 “Fermi”	Rapporto 280 / i7	Rapporto 480 / i7
Numero di elementi di elaborazione (core o SM)	4	30	15	7,5	3,8
Frequenza di clock (GHz)	3,2	1,3	1,4	0,41	0,44
Dimensione del chip	263	576	520	2,2	2,0
Tecnologia	Intel 45 nm	TSMC 65 nm	TSMC 40 nm	1,6	1,0
Potenza (chip, non modulo)	130	130	167	1,0	1,3
Numero transistor	700 M	1400 M	3030 M	2,0	4,4
Larghezza di banda della memoria (GByte/s)	32	141	177	4,4	5,5
Larghezza SIMD in singola precisione	4	8	32	2,0	8,0
Larghezza SIMD in doppia precisione	2	1	16	0,5	8,0
FLOPS di picco scalari in singola precisione (GFLOP/s)	26	117	63	4,6	2,5
FLOPS di picco SIMD in singola precisione (GFLOP/s)	102	da 311 a 933	da 515 a 1344	3,0-9,1	6,6-13,1
(SP1 somma o moltiplicazione)	N.A.	(311)	(515)	(3,0)	(6,6)
(SP1 istruzioni moltiplicazione e addizione fuse	N.A.	(622)	(1344)	(6,1)	(13,1)
(SP rare, pacchetti di esecuzione doppi contenenti moltiplicazioni e addizioni fuse)	N.A.	(933)	N.A:	(9,1)	-
FLOPS di picco SIMD doppia precisione (GFLOP/s)	51	78	515	1,5	10,1

Benchmarks

Kernel	Unità di misura	Core i7 960	GTX 280	GTX 280 / i7 960
SGEMM	GFLOP/s	94	364	3,9
MC	Miliardi cammini/s	0,8	1,4	1,8
Conv	Milioni pixel/s	1,25	3,5	2,8
FFT	GFLOP/s	71,4	213	3,0
SAXPY	GByte/s	16,8	88,8	5,3
LBM	Milioni accessi/s	85	426	5,0
Solv	Frame/s	103	52	0,5
SpMV	GFLOP/s	4,9	9,1	1,9
GJK	Frame/s	67	1020	15,2
Sort	Milioni elementi/s	250	198	0,8
RC	Frame/s	5	8,1	1,6
Search	Milioni ricerche/s	50	90	1,8
Hist	Milioni pixel/s	1517	2583	1,7
Bilat	Milioni pixel/s	83	475	5,7

David A. Patterson, John L. Hennessy, STRUTTURA E PROGETTO DEI CALCOLATORI, Zanichelli editore S.p.A. Copyright © 2015

GPU è co-processore tutta la parte di interfacciamento con l'esterno, lancio … è demandata alla CPU.

Struttura di una GPU

La GPU lavora su thread

Scheduling gerarchico dell'esecuzione:

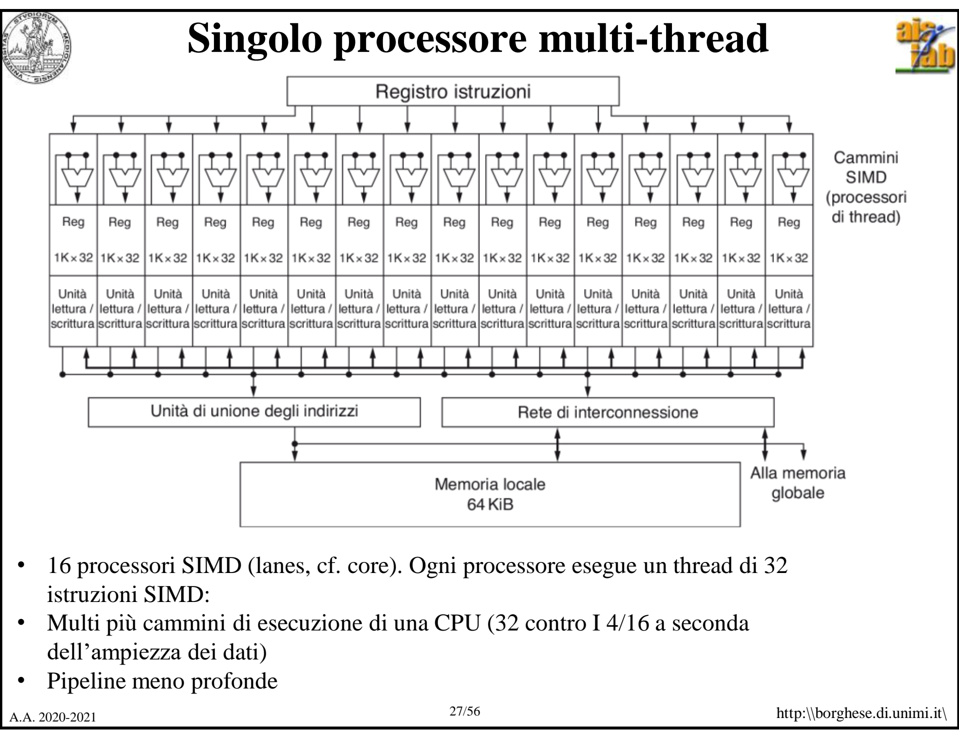
1) Thread block scheduler (Avvia ad esecuzione blocchi di thread).
   a) Blocchi di thread vengono assegnati ai diversi processori SIMD, ciascun processore è multi-thread.
   b) Nelle CPU i processi (thread multipli) vengono assegnati ai diversi Core, ciascun core è multiple-issue.

2) Nel processore SIMD abbiamo un thread scheduler.
   a) Lancia in esecuzione i diversi thread.
   b) Lancia in esecuzione le diverse istruzioni

NB Ciascun thread è costituito da un'istruzione SIMD (esecuzione in parallelo)

NB Ciascuna istruzione +(moderatamente) vettoriale.

Singolo processore multi-thread



- 16 processori SIMD (lanes, cf. core). Ogni processore esegue un thread di 32 istruzioni SIMD:
- Multi più cammini di esecuzione di una CPU (32 contro I 4/16 a seconda dell'ampiezza dei dati)
- Pipeline meno profonde

Visione generale del multi-threading

Le GPU vedono i processi come thread.

Supportano l'accesso sparso alla memoria gather-scatter).

L'esecuzione non è segregate per processi.

Esecuzione sequenziale nel tempo di thread provenienti da processi diversi.

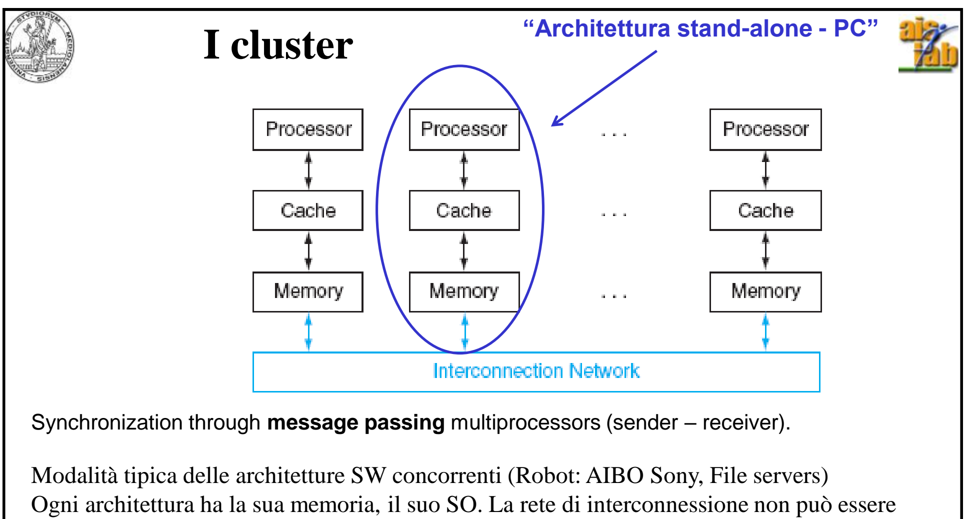
Stato dei thread deve essere leggero (coordinamento tramite la memoria).

Threading a grana fine.

Threading a granda grossa.

Switch di processo on il suo stato (centinaia di cicli di clock)

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.

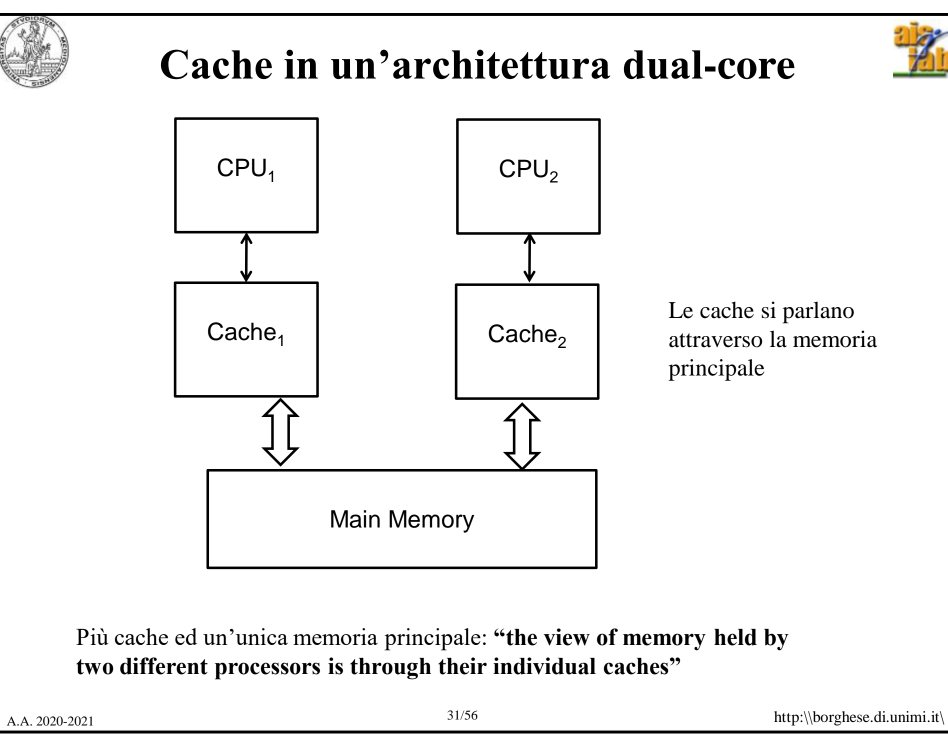
E' un'architettura molto più robusta ai guasti, facile da espandere.
I messaggi devono essere identificati in anticipo in modo esplicito.

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

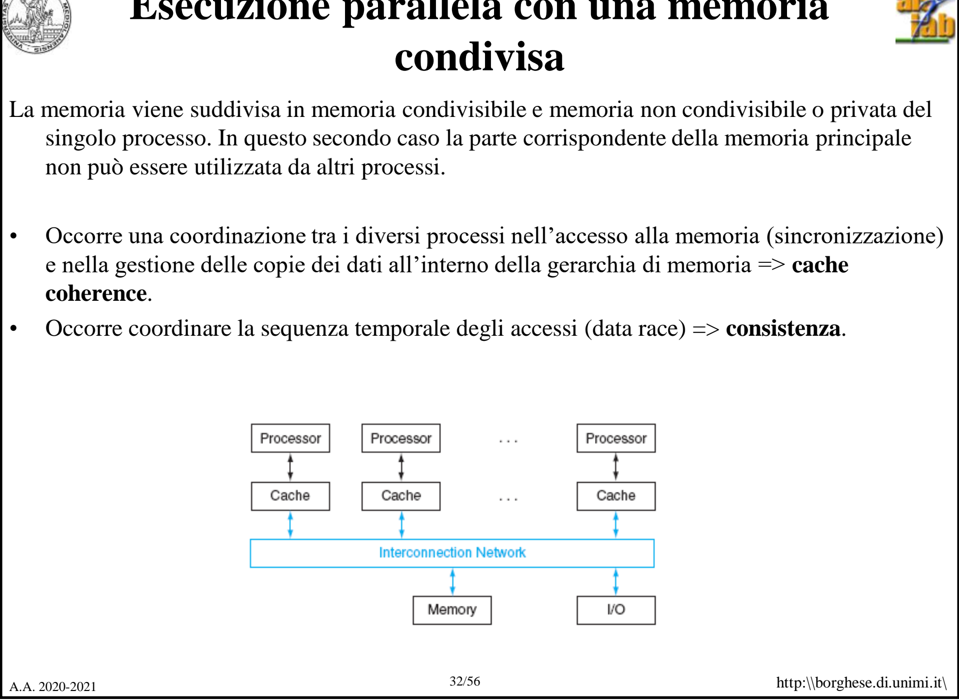
Sommario

Le architetture multi-processore

Gestione della memoria



Cache in un'architettura dual-core
CPU1
CPU2
Cache1
Cache2
Le cache si parlano attraverso la memoria principale
Main Memory
Più cache ed un'unica memoria principale: "the view of memory held by two different processors is through their individual caches"
A.A. 2020-2021
31/56
http:\\borghese.di.unimi.it\



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.
• Occorre una coordinazione tra i diversi processi nell'accesso alla memoria (sincronizzazione) e nella gestione delle copie dei dati all'interno della gerarchia di memoria => cache coherence.
• Occorre coordinare la sequenza temporale degli accessi (data race) => consistenza.
Processor
Processor
Processor
Cache
Cache
Cache
Interconnection Network
Memory
I/O
A.A. 2020-2021
32/56
http:\\borghese.di.unimi.it\

Consistenza della memoria mediante lock

Occorre che la lettura di una cella di memoria avvenga **dopo che** la scrittura della stessa cella da parte di un altro processo, logicamente eseguito prima, sia terminato. Questo non è garantito per:

- Esecuzione fuori ordine
- Esecuzione su più thread.

- Come garantire la piena proprietà di una linea di memoria?

- **Data race** sulla memoria principale ➔ **Lock** di una cella di memoria e unlock (**atomic operations**).
- Il lock è una proprietà definita via SW (librerie dal SO o compilatore (OpenMP)).

- Una tipica operzione di lock prevede lo 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 (**atomic swap**).
- Viene inserito un meccanismo hardware di blocco di una cella di memoria (lock o lucchetto).
- Viene gestito dalla parallelizzazione (e.g. OpoenMP) attraverso il sotto-sistema di controllo della memoria (libreria del SO).

Il lock è un meccanismo comune nella condivisione dei processi. Si utilizza ad esempio anche quando si scrivono documenti condivisi.

Meccanismo di lock – supporto ISA

Serve per proteggere la memoria da accessi di altre procedure. ***Non deve succedere nulla tra la read di una cella e la successiva scrittura della stessa cella.***

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

Scambio il contenuto di $s4 con il contenuto della memoria in 0($s1) incrementato di 4. Posso farlo solo se il lucchetto è aperto (e lo chiudi per gli altri thread), non posso farlo se il lucchetto è già chiuso da un altro thread.

```
try:    add $t0, $zero, $s4  # lucchetto tentativamente aperto, memorizzato in $t0
        ll $t1, 0($s1)       # load linked (lettura condizionata)
        addi $t1, $t1, 4     # do something on $t1
        sc $t0,0($t1)        # store conditional the new value of $t1
                             # il lucchetto viene testato in scrittura. Se la scrittura fallisce il
                             # lucchetto è chiuso da un altro thread e occorre aspettare
        beq $t0, $zero, try  # repeat if store fails (se ha successo $t0 = 1).
        addi $s4,$t1, 0
```

Se la procedura non è riuscita a leggere o non è riuscita a scrivere perchè c'era un blocco sulla cella di memoria, $t0 conterrà 0 e si riprova. Rischio di stallo. Utilizzata per costruire i semafori nei SO.

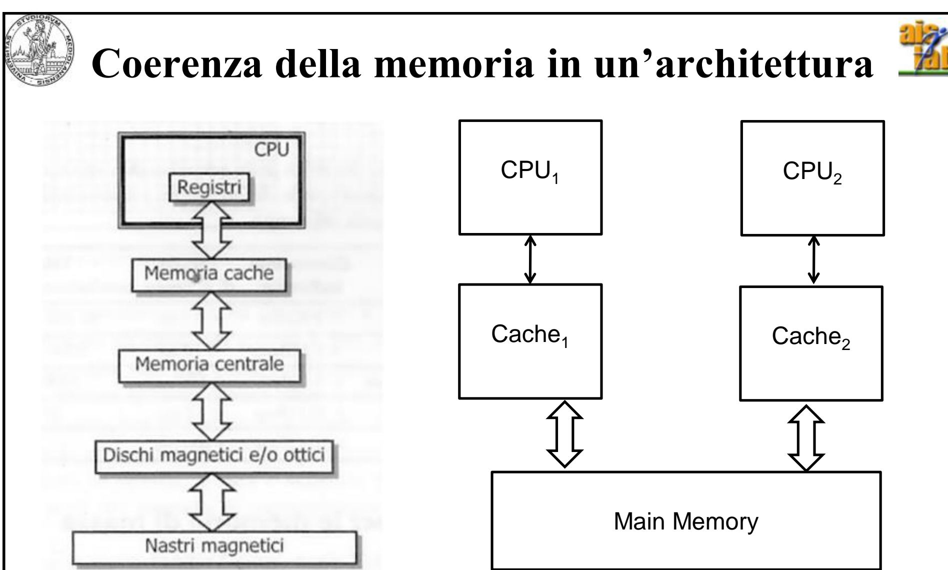
A \

Il meccanismo di lock - riassunto

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 della memoria 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.**

Singolo processore: scrittura in MM

Quando c'è register spilling o viene richiesta una scrittura in memoria (sw $t0, 24($t1)), il dato viene scritto in cache e si verifica un **disallineamento** della memoria principale e della cache.

La scrittura può provocare **disallineamento** (sfida la coerenza della gerarchia di memoria).

Ogni volta che si ha una richiesta di scrittura possono succedere due cose:
- La linea in cache individuata contiene già la parola con l'indirizzo della write.
- La linea in cache individuata non contiene la parola con l'indirizzo della write. In questo caso si ha una miss in cache e occorre prima caricare in cache l'intera linea, cioè tutte le parole del micro-blocco della memoria principale.

Si cerca di anticipare lo svuotamento di una linea di cache con i meccanismi di speculazione. Particolarmente vantaggioso per le memorie cache associative.

Write through

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

Write_buffer per liberare la CPU (DEC 3100) altrimenti stalli lunghi (100 cicli di clock)

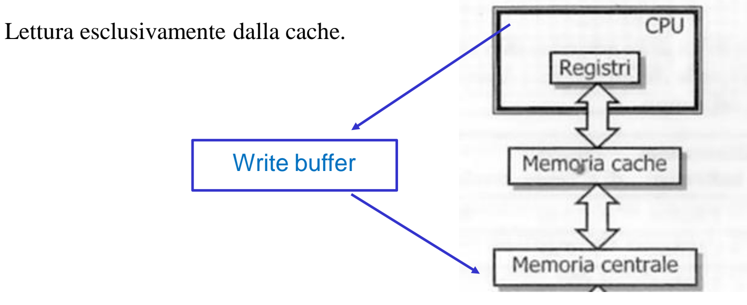
Chi libera il write buffer?

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.

Lettura esclusivamente dalla cache.



Write-back

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

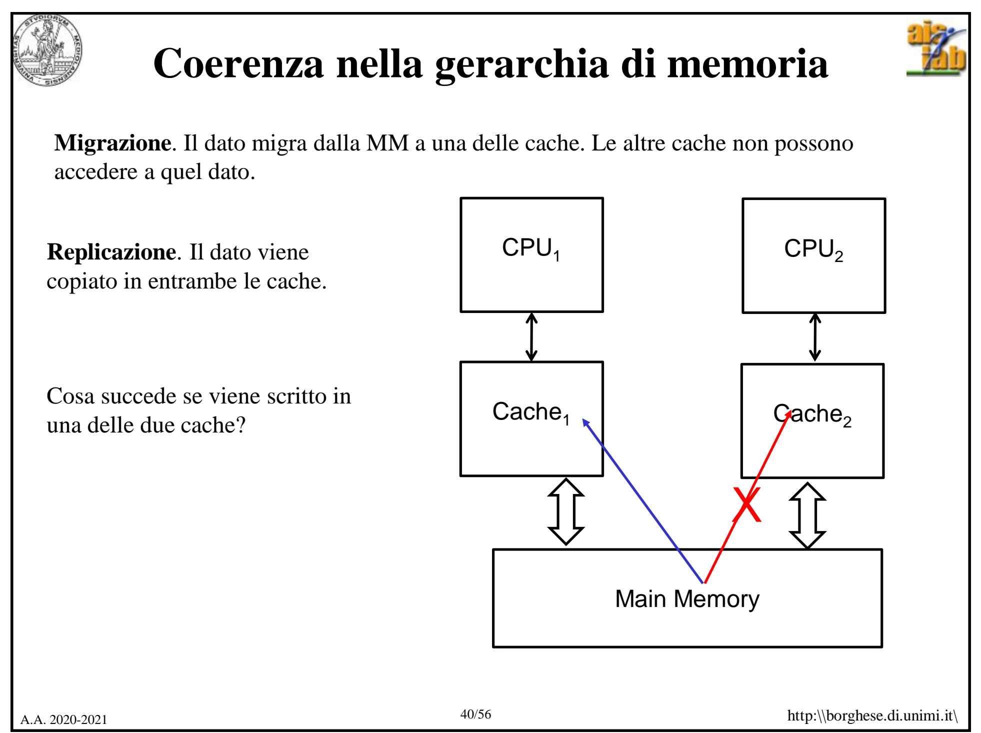
Utilizzo un bit di flag che viene settato quando altero il contenuto del blocco. Questo flag si chiama "dirty bit".

Vantaggiosa con cache n-associative.
Alla Memoria Principale trasferisco il blocco quando devo svuotare una linea di cache (è equivalente al register spilling).

Aumento dell'efficienza mediante **write buffer** per la gestione delle miss.

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.



Consistenza e coerenza: esempio - I

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.

Memoria write-through.

Passo	Evento	Contenuto cache processore A	Contenuto cache processore B	Contenuto MM
0	Nulla	X	X	0
1				
2				
3				

Consistenza e coerenza: esempio - II

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.

Memoria write-through.

Passo	Evento	Contenuto cache processore A	Contenuto cache processore B	Contenuto MM
0	Nulla	X	X	0
1	Processore A legge (lw)	**0**	X	0
2				
3				

Consistenza e coerenza: esempio - III

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.

Memoria write-through.

Passo	Evento	Contenuto cache processore A	Contenuto cache processore B	Contenuto MM
0	Nulla	X	X	0
1	Processore A legge (lw)	0	X	0
2	Processore B legge (lw)	0	**0**	0
3				

Consistenza e coerenza: esempio - IV

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.

Memoria write-through.

Passo	Evento	Contenuto cache processore A	Contenuto cache processore B	Contenuto MM
0	Nulla	X	X	0
1	Processore A legge (lw)	0	X	0
2	Processore B legge (lw)	0	0	0
3	Processore A scrive 1 (sw)	**1**	0	**1**

Disallineamento tra cache A e MM, e cache B.

Cosa succederebbe se la cache fosse “Write-back”? Disallineameno cache A, e cache B e MM che continuerà a contenere 0.

Deve esserci un meccanismo perché ci si accorga e si gestisca il disallineamento.

Coerenza delle memorie cache

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

Elemento chiave è il protocollo per il **tracking dello stato** (e.g. dirty bit, bit di validità) di ciascuna linea di ciascuna cache.

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

Protocollo di **bus snooping**.
- Distribuito (non esiste un controllore centrale delle cache cge mantenga l'informazione di stato – arbitraggio dei bus decentralizzato).
- Meccanismo di trasmissione a tutte le cache (broadcast) degli **eventi** (letture/scritture).
- Ogni cache monitora (snoops – curiosa/spia) questa trasmissione.
- In base a quello che monitora, decide "autonomamente" delle proprie linee.

Protocollo **directory-based**.
- Centralizzato (esiste un'unica locazione della MMU che mantiene lo stato di tutte le linee di cache – arbitraggio centralizzato).
- L'arbitro decide cosa fare delle linee di cache.
- Overhead maggiore.
- Traffico minore (niente broadcast).

Protocolli di snooping

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

Il controller della cache monitora indirizzo + segnale di controllo (**solo le write sono critiche**) per la MM prodotto da tutti i core.

Filosofia: il protocollo fa in modo che un processore abbia l'accesso esclusivo al dato (cella di MM) prima che possa scrivere.

Write invalidate protocol. Se il dato è posseduto anche da alter cache, viene invalidato su queste altre cache (viene invalidate l'intera linea).

La linea viene invalidate utilizzando il bit di validità.

Write invalidate protocol - I

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".*

Supponiamo questa situazione:

Attività Processore	Attività del bus	Contenuto cache processore A	Contenuto cache processore B	Contenuto della locazione X della MM
Nulla	Nulla	X	X	0

Write invalidate protocol - II

Lettura di un dato dalla Memoria Principale in posizione X sia da parte dalla CPU A che da parte della CPU B:

Attività Processore	Attività del bus	Contenuto cache processore A	Contenuto cache processore B	Contenuto della locazione X della MM
Nulla	Nulla	X	X	0
La CPU A legge X	La cache A ha una miss per X	**0**	X	0
La CPU B legge X	La cache B ha una miss per X	0	**0**	0

Write invalidate protocol - III

Scrittura di un dato nella Memoria Principale in posizione X da parte della CPU A.

Attività Processore	Attività del bus	Contenuto cache processore A	Contenuto cache processore B	Contenuto della locazione X della MM
Nulla	Nulla	X	X	0
La CPU A legge X	La cache A ha una miss per X	0	X	0
La CPU B legge X	La cache B ha una miss per X	0	0	0
La CPU A scrive 1 in X	Invalida la cache B	1	X	1

Potrebbe la CPU B scrivere nella stessa locazione di memoria X?

Si sta utilizzando una cache con scrittura in WriteBack o in WriteThrough?

Write invalidate protocol - IV

Scrittura di un dato nella Memoria Principale in posizione X da parte della CPU A.

Attività Processore	Attività del bus	Contenuto cache processore A	Contenuto cache processore B	Contenuto della locazione X della MM
Nulla	Nulla	X	X	0
La CPU A legge X	La cache A ha una miss per X	0	X	0
La CPU B legge X	La cache B ha una miss per X	0	0	0
La CPU A scrive 1 in X	La cache A ha una hit per X	1	0	1
Nulla	Invalida la cache B	1	X	1

Potrebbe la CPU B scrivere nella stessa locazione di memoria X?

Si sta utilizzando una cache con scrittura in WriteBack o in WriteThrough?

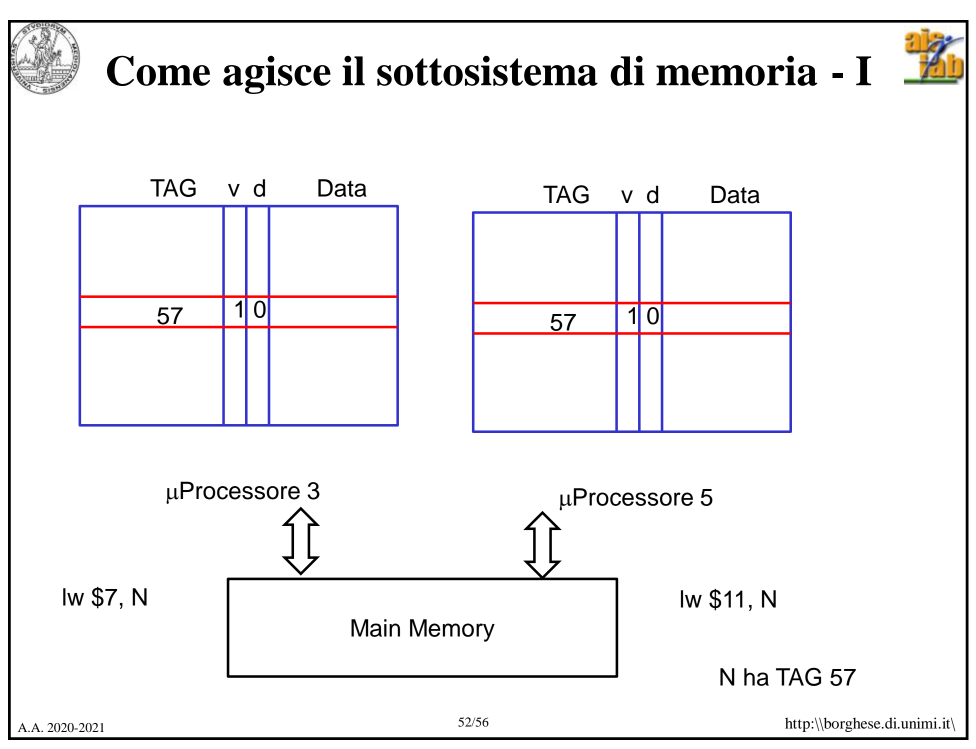
Write invalidate protocol - V

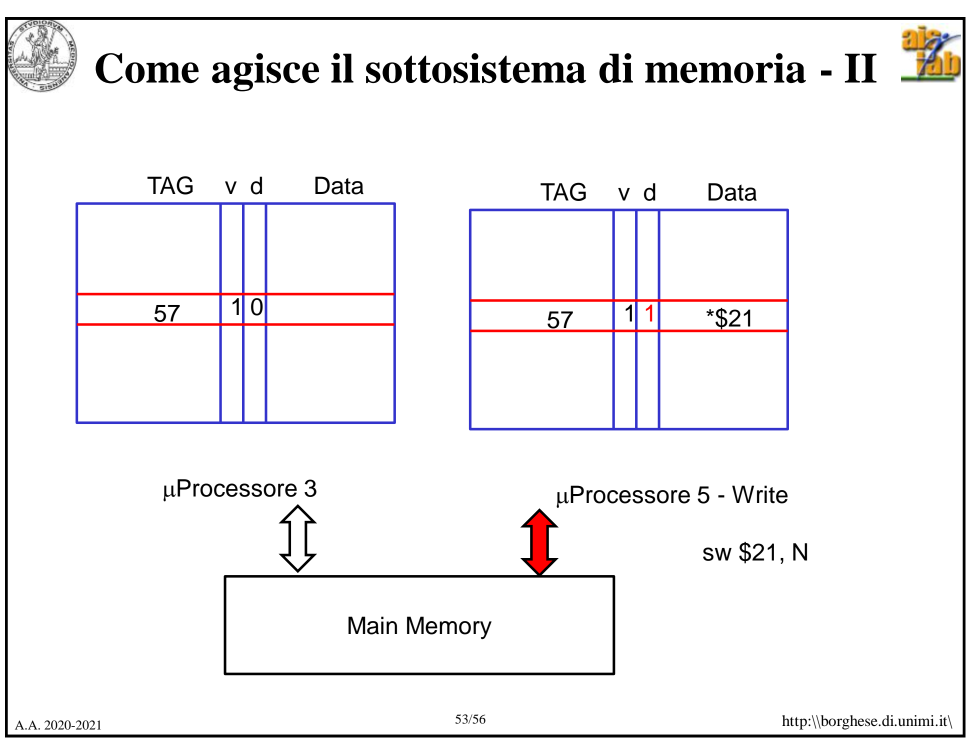
Attività Processore	Attività del bus	Contenuto cache processore A	Contenuto cache processore B	Contenuto della locazione X della MM
Nulla	Nulla	X	X	0
La CPU A legge X	La cache A ha una miss per X	0	X	0
La CPU B legge X	La cache B ha una miss per X	0	0	0
La CPU A scrive 1 in X	La cache A ha una hit per X	1	0	1
Nulla	Invalida la cache B	1	X	1
La CPU B legge X	La cache B ha una miss per X	1	1	1

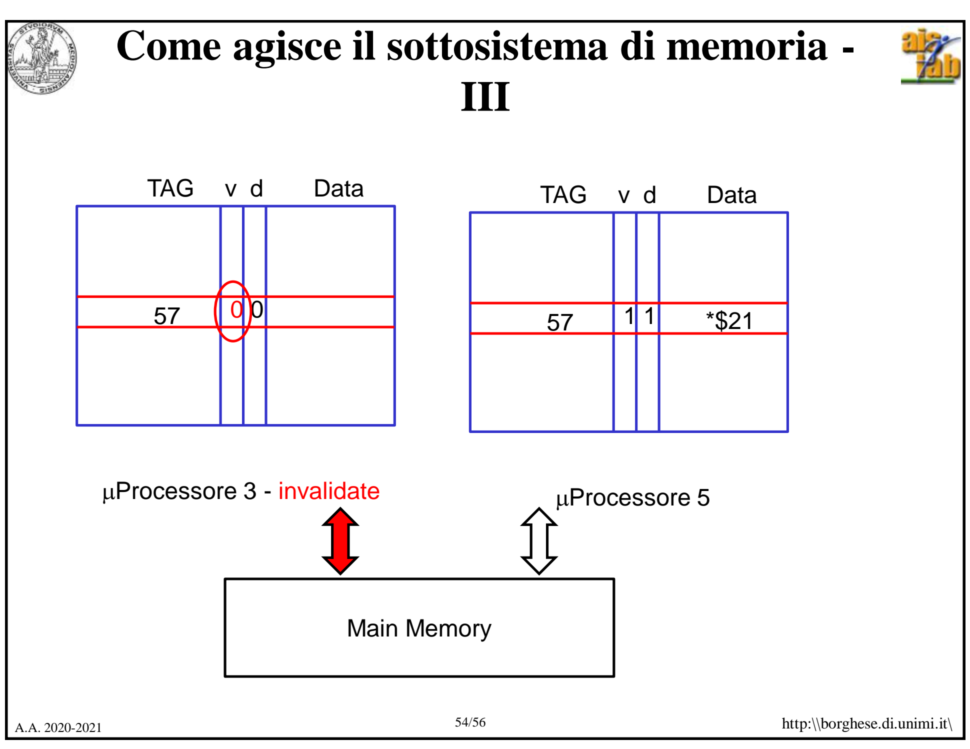
Supponiamo che la CPU B abbia bisogno ancora del dato in X. Questo dato non è ora più presente in cache (bit di validità = 0). Cosa deve fare?

E' stata ristabilita la coerenza.

Come agisce il sottosistema di memoria - I




Come agisce il sottosistema di memoria - II
TAG v d Data
57 1 0
TAG v d Data
57 1 1 *$21
μProcessore 3
μProcessore 5 - Write
sw $21, N
Main Memory
A.A. 2020-2021
53/56
http:\\borghese.di.unimi.it\


Come agisce il sottosistema di memoria - III
TAG v d Data
57 0 0
TAG v d Data
57 1 1 *$21
μProcessore 3 - invalidate
μProcessore 5
Main Memory
A.A. 2020-2021
54/56
http:\\borghese.di.unimi.it\

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 sono in esecuzione su due CPU diverse richiedono **due variabili diverse** ma che **ricadono nella stessa linea della cache**. Alla MMU la linea di cache appare condivisa e si innesca il meccanismo di invalidazione.

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

Sommario

Le architetture multi-processore

Gestione della memoria