# Introduzione

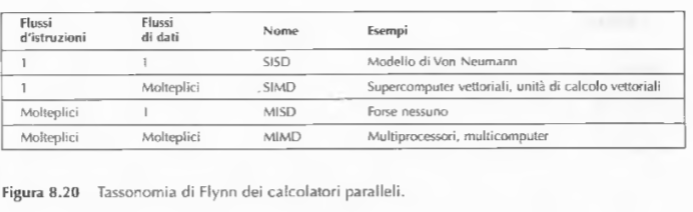
Benché i calcolatori diventino sempre più veloci, le aspettative dei loro utenti crescono almeno altrettanto rapidamente. Nonostante il costante incremento delle frequenze di clock, la velocità dei circuiti non può aumentare indefinitamente. La velocità della luce costituisce già oggi un limite pratico per i progettisti dei calcolatori.

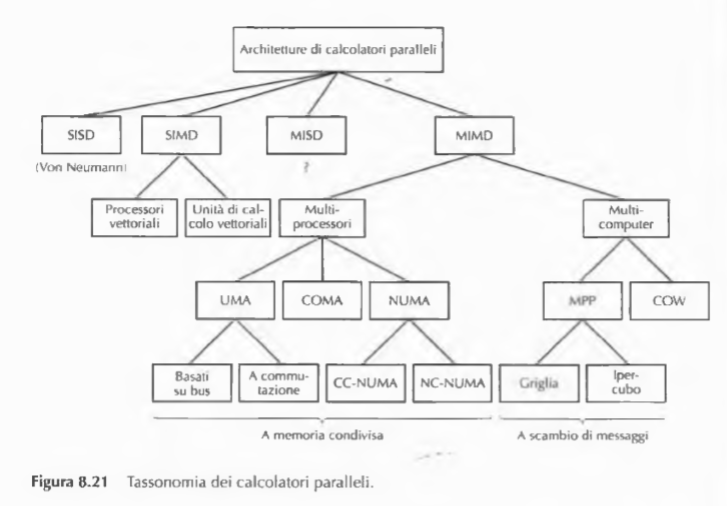
Un modo per ottenere una maggiore potenza di calcolo è attraverso l’utilizzo delle architetture parallele: molte CPU (con velocità normale) che collaborano per il conseguimento del medesimo obiettivo.

Il parallelismo nel chip della CPU (attraverso un progetto superscalare a pipeline) produce un fattore di miglioramento da 5 a 10.

Per incrementare dell’ordine delle centinaia occorre replicare le CPU, facendole cooperare in modo efficiente. In questo caso si può arrivare ad ottenere un incremento di 50, 100 o anche più.

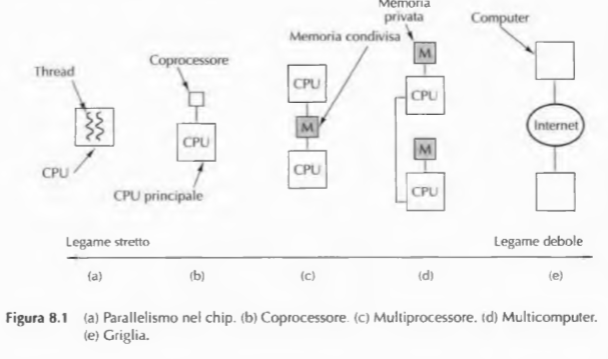
# Classificazione di Flynn





# Parallelismo nel chip

Obiettivo: far svolgere al chip più compiti alla volta.

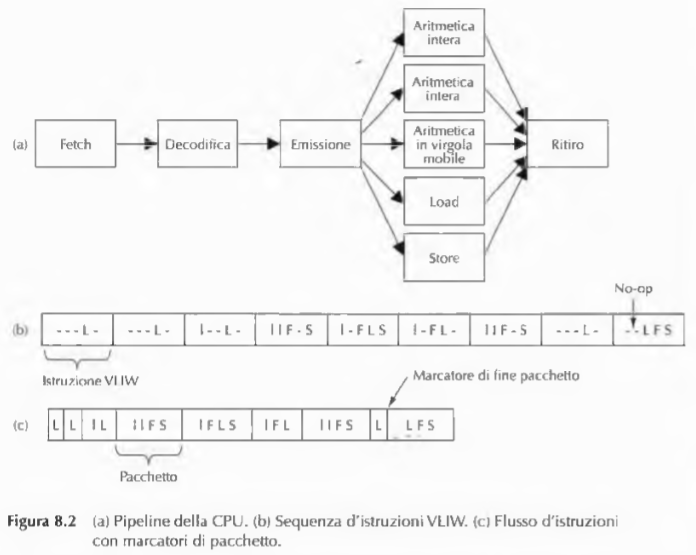


## Parallelismo a livello delle istruzioni

L’idea è di emettere più istruzioni per ciclo di clock. Ci sono due tipi di CPU a emissione multipla: processori superscalari e processori VLIW (“Very Long Instruction Word”, con parole di istruzione molto lunghe).

Le CPU superscalari sono composte da pipeline e più unità funzionali (Figura 8.2(a)).

I processori VLIW sono in grado di indirizzare le diverse unità funzionali con una sola linea di pipeline.



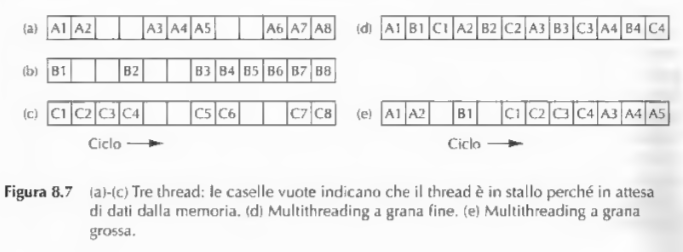
## Multithreading nel chip

Tutte le moderne CPU a pipeline presentano un problema: quando un riferimento in memoria fallisce nella cache (cache miss), bisogna aspettare molto tempo prima che la parola richiesta sia caricata nella cache, e così nel frattempo la pipeline è in stallo.

Il multithreading nel chip permette di mascherare queste situazioni in quanto la CPU gestisce contemporaneamente più thread di controllo. Così facendo, se il thread 1 è bloccato, la CPU può tenere impegnato l’hardware eseguendo il thread 2.

Esistono vari approcci.

### Multithreading a grana fine



La Figura 8.7 mostra una CPU capace di emettere un’istruzione per ciclo di clock. Nelle Figure 8.7(a)-(c) osserviamo l’attività dei thread A, B e C, lungo 12 cicli di macchina. Durante il primo ciclo, A esegue l’istruzione A1, che viene completata in un ciclo, quindi nel secondo comincia l’esecuzione di A2. Sfortunatamente questa istruzione provoca un fallimento nella cache di primo livello e così si perdono due cicli per recuperare l’istruzione dalla cache di secondo livello. Il thread riprende dal ciclo 5. Anche i thread B e C di tanto in tanto vanno in stallo, come illustrato nella figura. Secondo questo modello, quando un’istruzione va in stallo, le istruzioni successive non possono essere emesse.

Il multithreading a grana fine nasconde gli stalli grazie all’esecuzione a turno dei thread, con una commutazione a ogni ciclo, come mostra la Figura 8.7(d). Quando arriva il quarto ciclo l’operazione di memoria avviata da A1 è ormai stata completata e l’istruzione A2 può essere eseguita, anche se richiede il risultato di A1. Nell’esempio ogni stallo dura al massimo due cicli, perciò le operazioni vengono sempre completate in tempo visto che ci sono tre thread. Se ci fossero stalli di tre cicli, avremmo bisogno di quattro thread per assicurare la continuità dell’attività, e così via.

Dal momento che non vi è alcuna relazione tra i thread, ciascuno ha bisogno del proprio insieme di registri. All’emissione di un’istruzione è necessario accludere all’istruzione stessa un puntatore al suo insieme di registri così che, se viene referenziato un registro, l’hardware possa sapere quale registro usare. Per questa ragione, il numero massimo di thread che può essere eseguito in parallelo è stabilito a priori in fase di progettazione del chip.

Le operazioni di memoria non sono l’unica causa di stallo: alcune istruzioni condizionano l’esecuzione di altre.

Nella pipeline non ci sarà mai più di un’istruzione per thread e quindi il numero massimo di thread è pari al numero di stadi della pipeline.

### Multithreading a grana grossa

In questo caso un thread (A) si avvia e continua a emettere istruzioni finché non va in stallo, causando lo spreco di un ciclo. A quel punto l’esecuzione viene commutata su un altro

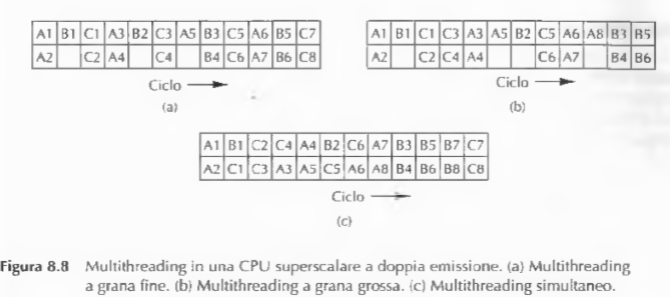
thread (B). Ma, poiché la prima istruzione di B va subito in stallo, si verifica un’altra commutazione di thread e al ciclo 6 va in esecuzione C1.

Visto che si perde un ciclo a ogni stallo, il multithreading a grana grossa è potenzialmente meno efficiente di quello a grana fine, ma presenta il vantaggio di richiedere meno thread per mantenere la CPU occupata.

Esiste una variante che permette di “guardare avanti” le istruzioni anticipando lo stallo e approssimando il multithreading a grana fine.

Indipendentemente dal tipo di multithreading usato, è necessario mantenere traccia dell’appartenenza delle operazioni ai thread. Nel caso del multithreading a grana fine l’unica possibilità ragionevole è l’inserimento di un identificatore di thread in ogni operazione, così da poter rintracciare la sua identità mentre attraversa i diversi stadi della pipeline. Nel caso del multithreading a grana grossa si può far di meglio: svuotare la pipeline a ogni commutazione di thread. In tal modo c’è sempre un solo thread nella pipeline e così la sua identità non è mai messa in dubbio. Questa soluzione ha senso solo se il tempo che intercorre tra due commutazioni è molto più lungo del tempo di svuotamento della pipeline.

Fin qui abbiamo presupposto che la CPU possa emettere una sola istruzione per ciclo, ma abbiamo già visto che le CPU moderne ne possono emettere di più. Nella Figura 8.8 supponiamo che la CPU possa emettere due istruzioni per ciclo, pur conservando la regola che, se un’istruzione va in stallo, le successive non possono essere emesse. La Figura 8.8(a) mostra il funzionamento di una CPU superscalare a doppia emissione e con multithreading a grana fine. Le prime due istruzioni del thread A possono essere emesse nel primo ciclo, ma nel caso di B incontriamo subito un problema nel ciclo successivo, e così può essere emessa una sola istruzione, e così via. La Figura 8.8(b) mostra il funzionamento di una CPU a doppia emissione con multithreading a grana fine, ma questa volta con uno schedulatore statico che non introduce un ciclo morto dopo lo stallo di un’istruzione. Praticamente, i thread si succedono a turno e la CPU emette due istruzioni per ogni thread finché non incontra uno stallo, nel qual caso commuta al thread successivo all’inizio del ciclo seguente.



### Multithreading simultaneo

C’è una terza possibilità di multithreading con le CPU superscalari, il multithreading simultaneo, illustrato nella Figura 8.8(c).

Ciascun thread emette due istruzioni per ciclo fintanto che può, altrimenti, non appena raggiunge uno stallo, viene emessa immediatamente un’istruzione del thread che segue affinché la CPU resti pienamente impegnata. Il multithreading simultaneo aiuta anche a mantenere occupate le unità funzionali. Quando un’istruzione non può essere avviata perché necessita di un’unità funzionale occupata, si può scegliere al suo posto un’istruzione di un altro thread. Nella figura supponiamo che B8 vada in stallo al ciclo 11, così C7 viene avviata al ciclo 12.

## 

## Multiprocessori in un solo chip

Si inseriscono più CPU (chiamate core) all’interno dello stesso chip (o meglio die).

I chip multicore sono come dei piccoli multiprocessori e per questa ragione vengono chiamati CMP (Chip-level MultiProcessor).

Dal punto di vista software essi non sono così differenti dai multiprocessori a bus o a reti di switch. Rispetto ai multiprocessori a bus che hanno una cache per ogni CPU, potrebbero avere delle prestazioni degradate sulla cache condivisa nelle situazioni in cui un core ingordo saturi la cache L2.

Altra differenza rispetto ai multiprocessori è la minore tolleranza ai malfunzionamenti causata dalla stretta connessione dei core (un errore su uno potrebbe propagarsi negli altri).

I multiprocessori possono essere realizzati con core identici (multiprocessori omogenei) oppure con core con specifiche funzionalità (multiprocessori eterogenei).

### Multiprocessori omogenei in un solo chip

Queste CPU si definiscono multiprocessori perché condividono le stesse cache, di primo e secondo livello, e la memoria principale.

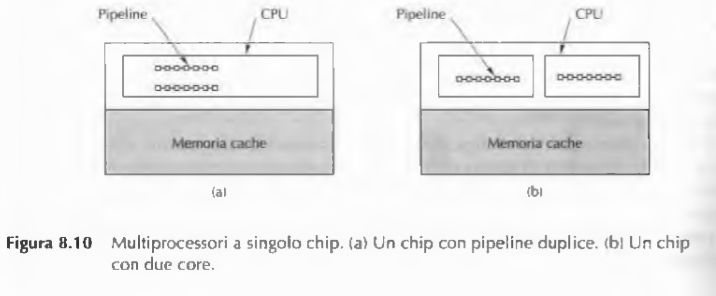
Esistono due tipologie predominanti per il progetto di multiprocessori in un solo chip di piccole dimensioni.

La prima è mostrata nella Figura 8.10(a): c’è davvero un solo chip, ma è dotato di una duplice pipeline che gli permette di raddoppiare potenzialmente il throughput.

Questo progetto permette la condivisione di alcune risorse, come le unità funzionali, così che una CPU possa sfruttare le risorse inutilizzate da parte dell’altra CPU.

La seconda è mostrata nella Figura 8.10(b): nel chip ci sono più core separati, ciascuno contenente un’intera CPU e con la propria pipeline. Un core (“nucleo, cuore”) è un grosso circuito, quale una CPU, un controllore di I/O o una cache, che può essere inserito su di un chip in maniera modulare, spesso uno di fianco all’altro.

In questo caso però l’interazione tra CPU non è semplice poiché risultano maggiormente disaccoppiate.



Mentre le CPU possono o meno condividere le cache, esse condividono sempre la memoria principale. Il processo di snooping (eseguito dall’hardware) garantisce che se una parola è presente in più cache e una CPU modifica il suo valore in memoria, essa è automaticamente rimossa in tutte le cache in modo da garantire la consistenza.

**Multiprocessori eterogenei in un solo chip**

In questo caso ogni core ha un compito specifico (come ad esempio decoder audio/video, crittoprocessore, interfacce di rete).

Poiché queste architetture realizzano un vero e proprio calcolatore completo in un singolo chip sono spesso dette system on a chip.

Come accaduto nel passato, l’hardware è molto più avanti del software: mentre sono attualmente disponibili dei chip multicore, non abbiamo applicazioni in grado di sfruttare queste nuove caratteristiche. Pochi programmatori sono in grado di scrivere algoritmi paralleli che gestiscano correttamente la competizione delle risorse condivise.

Come far comunicare tutti i core tra di loro? Nel caso di piccoli sistemi basterà un solo bus, ma nei sistemi grandi diverrebbe subito un collo di bottiglia. In genere il problema può essere risolto optando per più bus o per un anello che attraversa tutti i core. Nell’ultimo caso, l’arbitraggio avviene tramite il passaggio di un piccolo pacchetto chiamato token (gettone) lungo l’anello. Per poter trasmettere, un core deve per prima cosa catturare il token; quando ha finito di comunicare lo può reintrodurre sull’anello perché riprenda a circolare. Questo protocollo previene le collisioni sull’anello.

# Coprocessori

Per velocizzare un calcolatore si può aggiungere un secondo processore specializzato (coprocessore). Esistono molte varianti di coprocessori, di dimensioni molto diverse.

In alcuni casi la CPU assegna al coprocessore un’istruzione o un insieme d’istruzioni e gli ordina di eseguirle; in altri casi il coprocessore è più indipendente e lavora per conto proprio. In tutti i casi, ciò che li caratterizza è il fatto di assistere nell’esecuzione un altro processore, che resta il processore principale.

Esistono tre settori in cui è possibile velocizzare le prestazioni: elaborazione di rete(processori di rete), multimedia (processori grafici) e crittografia (crittoprocessori).

# Multiprocessori con memoria condivisa

Abbiamo visto come introdurre il parallelismo nel chip o all’interno di singoli sistemi tramite l’aggiunta di un coprocessore. Il passo successivo è la combinazione di più CPU per formare sistemi più grandi. Questi sistemi si dividono in due categorie: multiprocessori e multicomputer.

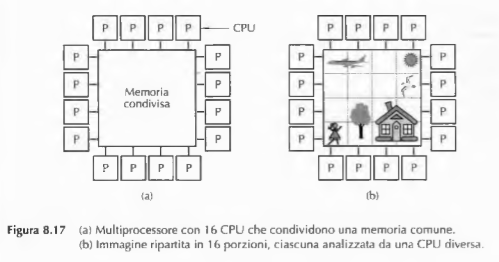
**Multiprocessori e multicomputer a confronto (MIMD)**

In un sistema di calcolo parallelo, le varie CPU che operano su parti diverse di uno stesso compito devono poter comunicare tra loro per scambiarsi informazioni.

Sono stati proposti e implementati due progetti diversi, i multiprocessori e i multicomputer, che si distinguono per la presenza o per l’assenza di memoria condivisa.

### Multiprocessori

Un multiprocessore è un calcolatore in cui tutte le CPU condividono una memoria comune.



Tutti i processi che cooperano in un multiprocessore possono condividere un solo spazio degli indirizzi virtuali mappato nella memoria comune. Ogni processo può leggere o scrivere una parola di memoria per mezzo di semplici istruzioni LOAD e STORE; non serve altro perché del resto si occupa l’hardware. Due processi possono comunicare in modo molto semplice: uno scrive dati in memoria, l’altro è in grado di leggerli. Quando un processo si blocca, la sua CPU salva il suo stato nelle tabelle del sistema operativo e cerca al loro interno un altro processo da eseguire.

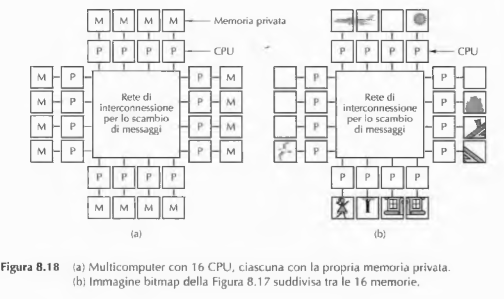
È proprio questa visione di un’unica memoria che distingue un multiprocessore da un multicomputer, in cui invece ogni calcolatore ha la propria copia del sistema operativo.

Un multiprocessore, come tutti i calcolatori, deve avere dispositivi di I/O, come i dischi o gli adattatori di rete, e altre periferiche. In alcuni sistemi multiprocessore solo una certa CPU ha accesso ai dispositivi di I/O ed è perciò dotata di funzioni speciali per l’I/O. Invece si parla di SMP (Symmetric MultiProcessor, “multiprocessore simmetrico”) quando tutte le CPU hanno uguale accesso a tutti i moduli di memoria e a tutti i dispositivi di I/O, e sono usate in modo intercambiabile dal sistema operativo.

Esistono due differenti tipologie di multiprocessori: UMA (Uniform Memory Access) in cui ogni parola di memoria può essere letta alla stessa velocità; NUMA (NonUniform Memory Access) in cui non tutte le parole di memoria possono essere lette alla medesima velocità.

### Multicomputer

Il secondo progetto di architettura parallela prevede per ogni CPU una memoria privata, cioè accessibile solo da essa e non dalle altre. Un progetto di questo tipo prende il nome di multicomputer o di sistema a memoria distribuita ed è illustrato nella Figura 8.18(a).



L’aspetto fondamentale di un multicomputer, che lo distingue da un multiprocessore, è il fatto che ogni CPU è dotata di una memoria privata e locale, cui può accedere tramite semplici istruzioni LOAD e STORE, ma cui nessun altra CPU può accedere.

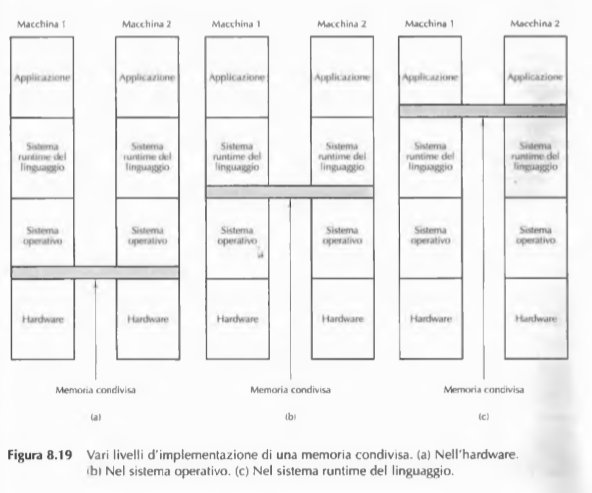
Visto che le CPU dei multicomputer non possono comunicare attraverso scritture e letture della memoria comune, c’è bisogno di un altro meccanismo di comunicazione: la soluzione adottata è lo scambio di messaggi lungo la rete di interconnessione.

Quando una CPU è interessata ai dati di un’altra CPU deve innanzitutto scoprire quale CPU possiede i dati che le interessano e spedirle quindi una richiesta di copia dei dati. Di norma questa operazione blocca la CPU finché la richiesta non viene soddisfatta. All’arrivo del messaggio alla CPU 1, il suo software deve analizzarlo e restituire i dati richiesti. Quando la

CPU 0 riceve il messaggio di risposta, il suo software si sblocca e continua l’esecuzione. La comunicazione tra i processi di un multicomputer avviene quindi tramite le primitive software *send* e *receive*.

Programmare un multicomputer è molto più difficile che programmare un multiprocessore. D’altra parte i multicomputer grandi sono molto più semplici ed economici da costruire rispetto a multiprocessori con lo stesso numero di CPU. Riuscire a implementare una memoria condivisa anche solo da qualche centinaia di CPU è praticamente un’impresa, mentre è semplice costruire un multicomputer con 10.000 CPU o più.

Sono stati compiuti sforzi ingenti per riuscire a costruire sistemi ibridi che fossero relativamente facili da costruire e programmare. Come risultato di questi sforzi si è giunti alla conclusione che la memoria condivisa può essere implementata in modi diversi, ciascuno con i propri vantaggi e svantaggi. La memoria condivisa può essere implementata a livelli diversi, come mostra la Figura 8.19.



Nella Figura 8.19(a) la memoria condivisa è implementata dall’hardware come in un vero multiprocessore. Secondo questo progetto, c’è una sola copia del sistema operativo.

Dal punto di vista del sistema operativo c’è una sola memoria ed esso si limita a mantenere traccia via software dell’attribuzione delle pagine ai processi.

Una seconda possibilità è usare l’hardware di un multicomputer e servirsi del sistema operativo per simulare la memoria condivisa come uno spazio paginato degli indirizzi virtuali condiviso in tutto il sistema. Questo approccio si chiama DSM (DistributedShared Memory, “memoria condivisa distribuita”) e prevede che ogni pagina sia localizzata in una delle memorie della Figura 8.18(a). Ogni macchina ha una propria memoria virtuale e una propria tabella delle pagine. Quando una CPU effettua una LOAD o una STORE all’interno di una pagina di cui non dispone, si verifica una trap rivolta al sistema operativo.

Una terza possibilità è implementare la memoria condivisa all’interno del sistema runtime nel livello utente. Secondo questo approccio, il linguaggio di programmazione fornisce una specie di astrazione della memoria condivisa che viene poi implementata dal compilatore e dal sistema runtime. I processi di tutte le macchine possono prelevare una tupla dallo spazio condiviso delle tuple o inserirvene una. Poiché il controllo degli accessi è effettuato tutto via software non c’è bisogno di hardware speciale o di supporto da parte del sistema operativo.

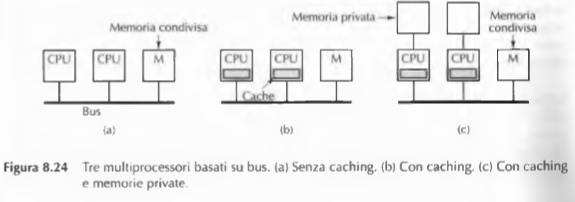
## 

## Architetture di multiprocessori simmetrici UMA

I multiprocessori più semplici si basano su un solo bus che interconnette tutte le CPU alla memoria condivisa, come illustrato nella Figura 8.24(a).

Quando una CPU vuole leggere una parola di memoria, per prima cosa verifica se il bus è occupato; se non lo è, può inserire nel bus l’indirizzo della parola voluta, attivare qualche segnale di controllo e aspettare finché la memoria non spedisce sul bus la parola desiderata. Se invece il bus è occupato, la CPU aspetta che si liberi; sta proprio qui il problema di questo progetto. Se le CPU sono due o tre, la contesa del bus è ancora gestibile; se sono molte di più allora diventa intrattabile perché il sistema sarà limitato completamente dalla larghezza di banda del bus, e la maggior parte delle CPU resterà inattiva per gran parte del tempo.

La soluzione del problema è l’aggiunta di una cache a ogni CPU, come rappresentato nella Figura 8.24(b). La cache locale consente di soddisfare molte più letture attingendo al proprio interno, perciò ci sarà molto meno traffico sul bus e il sistema potrà gestire più CPU.



### Cache snooping

Il caching non viene eseguito sulle singole parole di memoria ma su blocchi di 32 o 64 byte. Quando una parola è referenziata, l’intero blocco che la contiene, chiamato linea di cache, è caricato nella CPU che l’ha richiesta.

Ogni blocco di cache è contrassegnato come READ-ONLY oppure READ/WRITE.

Se una CPU scrive una parola che è contenuta anche in altri blocchi di cache remote, la CPU 2 si troverà a manipolare dati scaduti. Tutte le soluzioni richiedono controllori di cache progettati per essere in grado di “origliare” sul bus, cioè di monitorare tutte le richieste che transitano sul bus e che provengono da altre CPU o cache, al fine di intraprendere le azioni opportune. Questi dispositivi si dicono snooping cache o snoopy cache, cioè cache che letteralmente ficcano il naso nel bus. Le altre cache possono quindi avere: una copia “pulita” del blocco di memoria, allora la cache la sovrascrive con il valore aggiornato; una copia modificata del blocco di memoria (detta “copia sporca”), allora la cache lo deve prima trascrivere in memoria e poi applicare la modifica.

L’insieme di tali regole, utilizzate per la gestione della cache, è chiamato protocollo di coerenza della cache.

### 

### UMA con singolo bus e CPU dotate di RAM

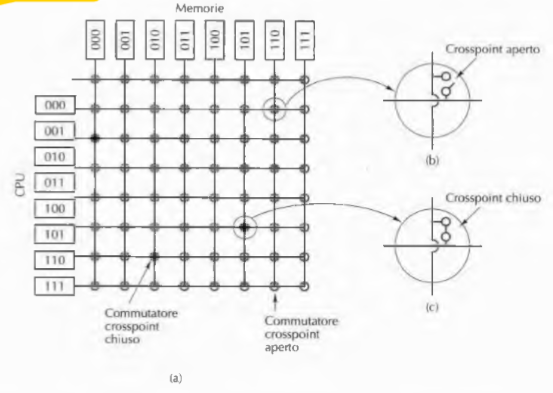
Una possibilità alternativa è il progetto della Figura 8.24(c), in cui ogni CPU dispone non solo di una cache, ma anche di una memoria locale privata, cui può accedere grazie a un bus dedicato (privato). A questo punto la memoria condivisa è utilizzata esclusivamente per scrivere variabili condivise (globali).

Questa soluzione riduce il traffico sul bus ma richiede una collaborazione attiva del compilatore che deve separare gli oggetti locali (il programma, lo stack, le variabili locali,...) da quelli globali.

### Multiprocessori UMA con commutatori crossbar

Per quanto se ne possa ottimizzare l’uso, la presenza di un solo bus limita il parallelismo dei multiprocessori UMA a circa 16 o 32 CPU. Per superare questo valore ci vuole un diverso tipo di rete di interconnessione. Il circuito più semplice che collega n CPU a k memorie è il commutatore crossbar (“a traversa”) mostrato nella Figura 8.27.

A ogni intersezione tra una linea orizzontale (CPU) e una verticale (RAM) c’è un crosspoint (“punto di incrocio”). Un crosspoint è esso stesso un piccolo commutatore che può essere aperto o chiuso elettronicamente, a seconda che si voglia collegare o meno le linee corrispondenti.



La Figura 8.27(a) mostra tre crosspoint chiusi contemporaneamente, consentendo così la comunicazione simultanea tra le seguenti coppie (CPU, memoria): (001, 000), (101, 101) e (110, 010).

Una delle proprietà più apprezzabili dei commutatori crossbar è che costituiscono reti non bloccanti, ovvero non succede mai che venga negata a una CPU la connessione di cui ha bisogno perché la linea è già occupata.

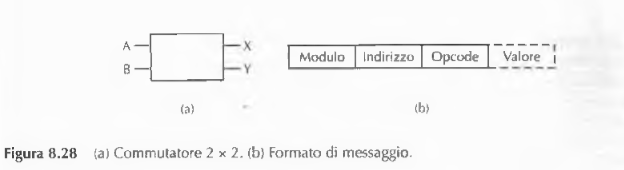
Inoltre non c’è bisogno di alcuna pianificazione: anche se sono già state stabilite sette connessioni qualsiasi, è sempre possibile collegare un’ottava CPU a una nuova memoria.

Rimane il problema della competizione per la memoria, qualora due, o più, CPU vogliono accedere allo stesso modulo nel medesimo istante, anche se, partizionando la memoria in n unità, la competizione si riduce di un fattore n rispetto al modello con bus singolo.

Una delle peggiori caratteristiche di questo schema è che il numero degli incroci cresce come n2. Con 1000 CPU e 1000 moduli di memoria occorrono un milione di crosspoints. Costruire un crossbar di queste dimensioni non è fattibile.

**Multiprocessori UMA con reti a commutazione multilivello**

Una progettazione di multiprocessori differente è basata su commutatore 2x2 della Figura 8.28(a).



Questo commutatore ha due ingressi e due uscite; i messaggi che provengono da ciascuno dei due input possono essere indirizzati verso ognuna delle due uscite.

Ogni messaggio contiene (come schematizzato nella Figura 8.28(b)):

- Modulo, che specifica la memoria da usare;

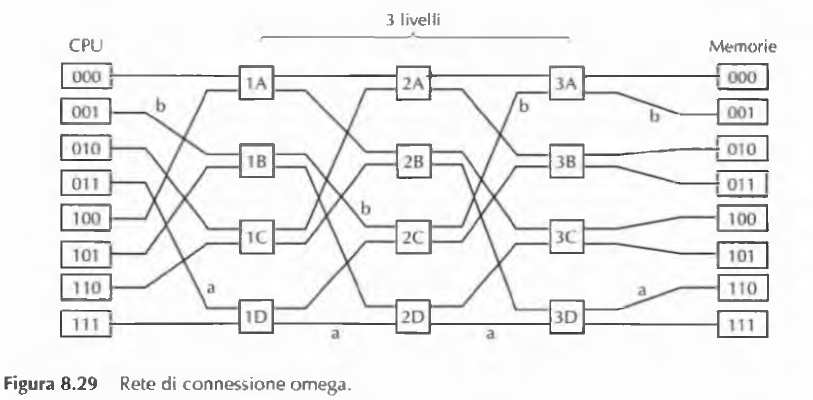
- Indirizzo, specifica un indirizzo all’interno di quel modulo;

- Opcode, che stabilisce il tipo di operazione, per esempio READ o WRITE;

- Valore, che potrebbe contenere un operando.

Il commutatore esamina il campo Modulo e lo usa per determinare la linea X o Y lungo cui inviare il massaggio. I nostri commutatori 2x2 possono essere composti in molti modi per dar vita a una più grande rete a commutazione multilivello.

Una possibilità economica e semplice è la rete omega (illustrata nella Figura 8.29).



In questo caso ci sono otto CPU connesse a otto memorie tramite 12 commutatori. In generale, per collegare n CPU a n memorie ci vogliono log2n livelli con n/2 commutatori ciascuno, per un totale di (n/2)log2n commutatori: è un numero molto inferiore a n2, specie per valori grandi di n.

Spesso ci si riferisce alla struttura dei collegamenti di queste reti con il termine di mescolamento perfetto (perfect shuffle) perché il mescolamento dei segnali a ogni livello somiglia a un mazzo di carte tagliato in due e rimescolato carta per carta.

Per capire il funzionamento di una rete omega, supponiamo che la CPU 011 voglia leggere una parola dal modulo di memoria 110. La CPU spedisce un messaggio *read* contenente 110 nel campo Modulo al commutatore 1D. Il commutatore preleva il primo bit (quello più a sinistra) da 110 e lo usa per l’instradamento. Lo 0 instrada verso l’uscita in alto, 1 verso la linea in basso. Poiché il bit vale 1 il messaggio viene instradato a 2D attraverso l’uscita in basso. Tutti i commutatori di secondo livello, compreso 2D, usano per l’instradamento il secondo bit. In questo caso è ancora un 1 e così il messaggio viene spedito a 3D attraverso l’uscita in basso. Qui viene testato il terzo bit che vale 0. Di conseguenza il messaggio fuoriesce dalla linea in alto e raggiunge la memoria 110, come richiesto. Il cammino percorso da questo messaggio è indicato con una lettera *a* nella Figura 8.29. Mentre i messaggi attraversano la rete di commutazione, i bit più a sinistra del campo Modulo diventano progressivamente inutili. Possono perciò essere riutilizzati registrando al loro interno i numeri delle linee in ingresso, in modo da specificare il percorso di ritorno del messaggio di risposta. Nel caso del cammino *a* le linee in ingresso sono 0 (ingresso in alto di 1D), 1 (ingresso in basso di 2D) e 1 (ingresso in basso di 3D). La risposta viene instradata usando 011, che questa volta va letto da destra a sinistra. Mentre si svolgono queste operazioni, la CPU 001 vuole scrivere una parola nel modulo di memoria 001. Il procedimento è analogo al precedente: il messaggio viene instradato rispettivamente attraverso le uscite in alto, in alto e poi in basso, come indicato dalla lettera *b*. Alla consegna del messaggio Modulo contiene 001, che rappresenta il cammino percorso. Le due richieste possono procedere parallelamente dal momento che usano commutatori, linee e memorie disgiunti.

Si consideri ora che cosa accadrebbe se la CPU 000 volesse accedere al modulo di memoria 000. La sua richiesta entrerebbe in conflitto con quella della CPU 001 in corrispondenza dello switch 3A. Una delle due dovrebbe aspettare.

Quindi, a differenza del commutatore crossbar la rete omega è una rete bloccante, cioè non è possibile elaborare simultaneamente qualsiasi insieme di richieste.

È quindi auspicabile riuscire a distribuire i riferimenti alla memoria in modo uniforme rispetto ai moduli. Una tecnica comune è quella di usare i bit meno significativi come numero di modulo.

Un sistema di memoria in cui le parole consecutive si trovano in moduli diversi si dice interlacciato. Le memorie interlacciate massimizzano il parallelismo perché la maggior parte dei riferimenti a memoria coinvolge indirizzi consecutivi.

È altresì possibile progettare reti di commutazione non bloccanti e che offrono cammini multipli da ogni CPU a ogni modulo di memoria, per meglio distribuire il traffico di rete.

## 

## Multiprocessori NUMA

Dovrebbe essere ora chiaro che i multiprocessori UMA a bus singolo sono spesso limitati a non più di qualche dozzina di CPU, e i multiprocessori con commutazione crossbar o multilivello necessitano di molto hardware (costoso) e perciò non possono essere molto più grandi. Per superare le 100 CPU bisogna rinunciare a qualcosa, spesso all’idea che tutti i moduli di memoria richiedano lo stesso tempo d’accesso. Questa rinuncia conduce ai multiprocessori NUMA (NonUniform Memory Access) che, come i cugini UMA, gestiscono un solo spazio degli indirizzi per tutte le CPU, ma diversamente da loro, garantiscono un accesso più veloce ai moduli vicini rispetto a quelli lontani.

Le macchine NUMA hanno tre caratteristiche chiavi:

- c’è un unico spazio di indirizzamento visibile a tutte le CPU;

- l’accesso alla memoria remota è attraverso istruzioni di LOAD e STORE;

- l’accesso alla memoria remota è più lento dell’accesso rispetto alla memoria locale.

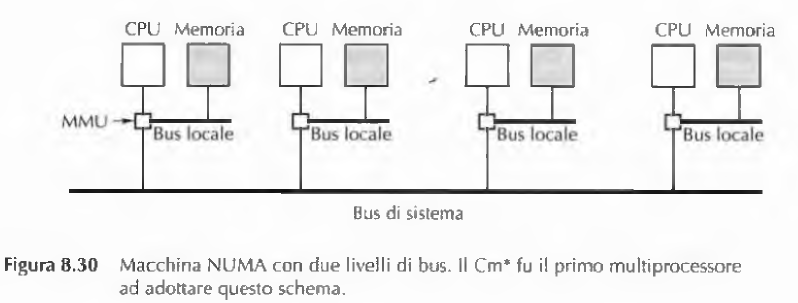
Ci sono due tipi di macchine NUMA:

- No Cache NonUniform Memory Access (NC-NUMA): quando il tempo di accesso alla memoria distante non è nascosto (perché non c’è caching);

- Cache Coherent NonUniform Memory Access (CC-NUMA): quando sono presenti cache coerenti.

**Multiprocessori NC-NUMA**

Una macchina NC-NUMA è illustrata in forma semplificata nella Figura 8.30. Essa contiene varie CPU, ciascuna dotata di una piccola memoria cui accedeva tramite un bus locale. Inoltre, le CPU erano collegate attraverso un bus di sistema.



Quando la MMU (opportunamente modificata) riceveva una richiesta di accesso a memoria, stabiliva per prima cosa se la parola si trovava in memoria locale. In tal caso la richiesta veniva inoltrata lungo il bus locale per ottenere la parola. In caso contrario, la richiesta veniva instradata lungo il bus di sistema verso il processore che conteneva la parola, che rispondeva di conseguenza. Naturalmente questa seconda evenienza impiegava molto più tempo della prima.

La coerenza della memoria in una macchina NC-NUMA è garantita dal fatto che non c’è caching. Ogni parola di memoria si può trovare in una sola locazione, perciò non c’è pericolo che ne esista una copia difforme: non ci sono copie.

D’altro canto, la collocazione di una pagina in memoria diventa molto importante perché il degrado delle prestazioni dovuto a un cattivo posizionamento dei dati è molto rilevante.

### Multiprocessori CC-NUMA

I progetti di multiprocessori come quello della Figura 8.30 risultano insoddisfacenti al crescere delle dimensioni dei sistemi, perché non prevedono il caching. Dover accedere alla memoria remota ogni volta che si fa riferimento a una parola di memoria non locale vuol dire pagare un alto prezzo in termini di prestazioni. Se però si aggiunge il caching, bisogna assicurare anche la coerenza delle cache. Un modo possibile per garantirla è permettere lo snooping del bus di sistema.

Attualmente il modo più diffuso per costruire grandi CC-NUMA è il sistema multiprocessore basato su directory (directory-based multiprocessor). L’idea è quella di mantenere un database per ricordare la collocazione e lo stato di ciascuna linea di cache. Dopo un riferimento a una linea di cache, si interroga il database per scoprire dove si trova la linea e se è intatta o è stata modificata. Poiché il database viene interrogato a ogni istruzione che accede alla memoria, è necessario che sia implementato con hardware specializzato molto veloce, capace di rispondere in una frazione di ciclo di bus.

L’unione di tutte le memorie dei nodi costituisce l’intero spazio di indirizzamento.

# 

# Tipi di sistema operativo multiprocessore

## Cenni di memoria virtuale

All’inizio dell’era informatica, le memorie dei computer erano costose e poco capienti. In quel periodo, i programmatori impiegavano molto tempo nel tentativo di “assottigliare” i programmi perché entrassero in memorie minuscole.

La soluzione tradizionale a questo problema era l’uso di una memoria secondaria, per esempio di un disco. Il programmatore divideva il programma in un certo numero di pezzi, detti overlay (“ricoprimento”), ciascuno dei quali poteva entrare in memoria. Al fine di eseguire il programma si recuperava il primo overlay e lo si eseguiva per un po’. Al termine della sua esecuzione il primo pezzo leggeva l’overlay successivo e lo mandava in esecuzione, e così via.

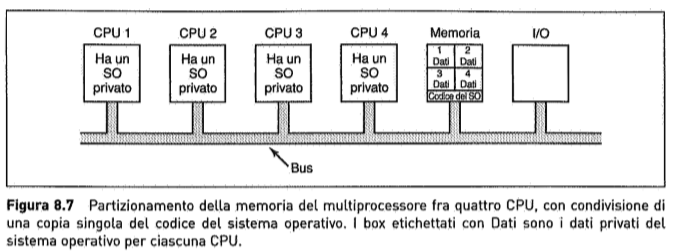
Nel 1961 si trovò un metodo per eseguire il processo di overlay automaticamente, in modo tale che il programmatore non avrebbe dovuto neanche accorgersi della sua presenza.

Questo metodo, oggi chiamato memoria virtuale, presentava il vantaggio evidente di liberare il programmatore da un grosso lavoro di noiosa preparazione.

## Ciascuna CPU ha il proprio sistema operativo

Il modo più semplice per organizzare un sistema operativo multiprocessore consiste nel dividere staticamente la memoria in tante partizioni quante sono le CPU. Ogni CPU ha un proprio sistema operativo e una propria memoria privata. Le CPU operano come computer indipendenti.

Una ovvia ottimizzazione è rendere possibile la condivisione del sistema operativo tra CPU e fare copie private delle sole strutture dati del sistema operativo, come illustrato nella Figura 8.7.



Questo schema è migliore rispetto a *n* computer separati, poiché permette alle CPU di condividere un insieme di dispositivi di I/O. Però, presenta diverse problematiche:

- quando un processo effettua una chiamata di sistema, essa è intercettata e gestita dalla propria CPU, utilizzando le strutture dati presenti nelle tabelle del proprio sistema operativo;

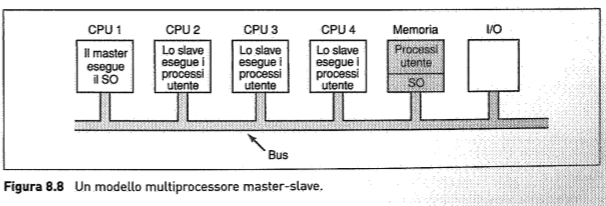
- non c’è condivisione di processi, ogni CPU gestisce indipendentemente i propri processi (quindi non è possibile il bilanciamento del carico);

- non c’è condivisione di pagine. Una CPU può eseguire molto swap a causa della carenza di spazio libero mentre un’altra può disporre di pagine libere;

- se un sistema operativo mantiene un buffer cache dei blocchi del disco usati recentemente indipendentemente dagli altri questo può condurre a letture inconsistenti (causate da letture sporche nei vari buffer).

**Multiprocessori Master-Slave**

In questo schema il sistema operativo e le sue tabelle sono presenti sulla CPU 1, e non sulle altre; quindi tutte le chiamate di sistema sono reindirizzate alla CPU 1 perché siano ivi elaborate.



Questo modello risolve la maggior parte dei problemi del modello precedente:

- c’è una singola struttura dati che tiene traccia dei processi pronti;

- la CPU master può bilanciare il carico sulle varie CPU;

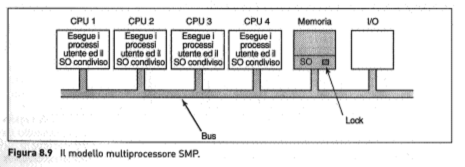
- le pagine sono allocate dinamicamente tra i processori e c’è una sola buffer cache che evita le inconsistenze.

Il problema è che, con molte CPU, il master diventa un collo di bottiglia, perché deve gestire tutte le chiamate di sistema.

In conclusione questo modello è utilizzabile per sistemi con poche CPU.

**Multiprocessori simmetrici**

L’SMP (MultiProcessore Simmetrico) elimina l’asimmetria del modello master-slave: in pratica, ogni CPU può diventare master.



Una copia del sistema operativo è in memoria e ciascuna CPU può eseguirla.

La CPU che effettua una chiamata di sistema, effettua una trap al kernel ed elabora la richiesta.

Questo schema bilancia processi e memoria dinamicamente (visto che c’è solo un insieme di tabelle del sistema operativo) ed elimina il collo di bottiglia.

D’altra parte introduce le seguenti problematiche:

- le CPU devono essere sincronizzate a causa della condivisione del sistema operativo;

- bisogna evitare i deadlock (potrebbe essere impossibile risolverli);

- la gestione è critica perché dipende dall’esperienza del programmatore nel contesto del sistema.

**Sincronizzazione dei multiprocessori**

Le CPU in un multiprocessore hanno bisogno di sincronizzarsi: le regioni critiche del kernel e le tabelle devono essere protette da mutex.

In primo luogo occorrono appropriate primitive per la sincronizzazione. Se un processo su un monoprocessore deve accedere ad una tabella critica del kernel, il codice del kernel può disabilitare le interruzioni e continuare il lavoro senza che un altro processo acceda alla tabella prima che abbia finito. Su un multiprocessore, la disabilitazione degli interrupt riguarda solo la CPU che la effettua, le altre CPU continuano a funzionare normalmente e possono accedere ad una regione critica. Di conseguenza, tutte le CPU devono rispettare un protocollo mutex appropriato.

Il cuore del protocollo mutex consiste in un’istruzione TSL (Test e Set Lock) che permette ad una parola di memoria di essere controllata e impostata in una sola operazione indivisibile.

Con più CPU, l’istruzione TLS deve prima bloccare il bus, evitando l’accesso di altre CPU, poi accedere alla memoria e quindi sbloccare il bus.

**Schedulazione dei multiprocessori**

Su un monoprocessore lo scheduling è mono-dimensionale. La sola domanda cui si deve rispondere è:“qual è il prossimo processo da eseguire?”.

In un multiprocessore, lo scheduling è bidimensionale: lo schedulatore deve decidere quale processo eseguire, e su quale CPU eseguirlo.

Un altro fattore di complicazione è che in alcuni sistemi non tutti i processi sono correlati o lo sono in gruppi.

Esistono vari algoritmi di scheduling per ciascuna situazione:

- timesharing;

- condivisione dello spazio;

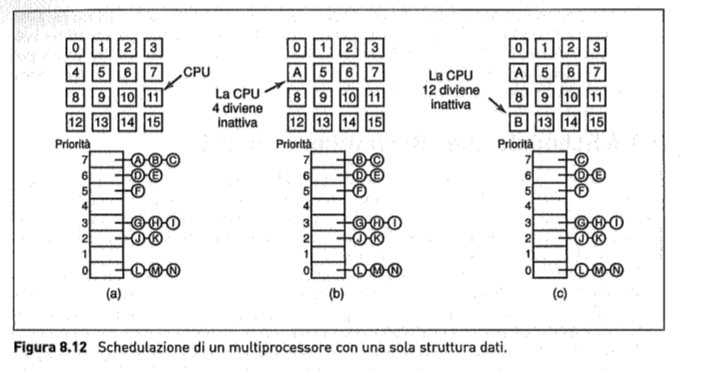
- schedulazione gang.

**Timesharing**

Il timesharing è utilizzato quando i processi sono indipendenti.

L’algoritmo utilizza un vettore di liste di processi (tutti nello stato “ready”) a diverse priorità di

esecuzione, come vediamo nella figura 8.12(a).



Il fatto di avere un’unica struttura dati utilizzata da tutte le CPU ripartisce il tempo tra le CPU come se fossero in un sistema monoprocessore, fornendo anche un sistema automatico di bilanciamento del carico poiché non accade mai che una CPU è inattiva e un’altra sovraccarica.

Questo algoritmo presenta due potenziali problemi:

- le dispute per l’accesso alla struttura dati di scheduling con il crescere del numero di CPU;

- il normale sovraccarico che si presenta per fare lo scambio di contesto quando un processo si blocca per l’I/O.

Supponiamo che un processo termini il suo quanto di tempo mentre mantiene uno spin lock, le altre CPU dovranno attendere il rilascio della risorsa che avverrà quando il processo tornerà attivo e rilascerà il lock. In alcuni sistemi, quando ciò accade, il processo che esegue un lock attiva un flag speciale di processo che permette allo scheduler di non bloccare il processo ma di concedergli un extra-quanto (smart scheduling).

Alcuni multiprocessori utilizzano lo scheduling per affinità (affinity scheduling). L’obiettivo è di far eseguire lo stesso processo alla medesima CPU per sfruttare, ad esempio, i blocchi del processo già presenti nella cache della CPU sulla quale è stato già eseguito

Per creare questa affinità si utilizza un algoritmo di schedulazione a due livelli:

- quando è creato un processo viene assegnato alla CPU più scarica (top-level scheduling);

- lo scheduling reale è fatto separatamente da ogni CPU (bottom-level scheduling) utilizzando l’affinità della cache e le priorità dei processi.

Lo scheduling a due livelli presenta tre vantaggi:

- distribuisce il carico in modo equo fra le CPU disponibili;

- l’utilizzo delle affinità della cache migliora le performance;

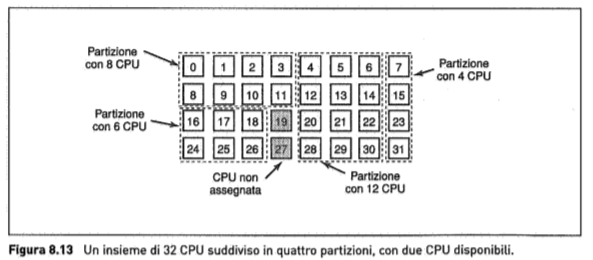
- si minimizzano le dispute sulle liste di processi poiché ogni CPU tenta di riusare i propri processi.

**Condivisione dello spazio**

Se esiste una qualche correlazione fra i processi si può utilizzare lo scheduling a condivisione di spazio (space sharing). Lo scheduling di molti processi nello stesso tempo su diverse CPU è chiamato space sharing scheduling.

Si supponga di creare contemporaneamente un insieme di thread correlati; lo scheduler controlla se ci sono tante CPU libere quanti sono i thread. Se esistono, ad ogni thread è assegnata una CPU dedicata ed è avviato; al contrario, nessun processo è avviato fintantoché non ci sono disponibili un numero adeguato di CPU.

Nella figura 8.13 sono illustrate partizioni di dimensioni 4, 6, 8 e 12 CPU.



Ad ogni istante di tempo, l’insieme delle CPU è partizionato staticamente, ciascun gruppo delle quali esegue il gruppo di processi correlati.

Un vantaggio evidente dello scheduling a condivisione di spazio è l’eliminazione della multiprogrammazione che elimina anche l’overhead dovuto agli scambi di contesto.

Uno svantaggio è il tempo sprecato quando si blocca una CPU perché non c’è nulla da fare.

**Schedulazione gang**

Conseguentemente alle ultime due affermazioni riguardo lo space sharing scheduling, si sono creati algoritmi che tentassero di schedulare spazio e tempo insieme.

Una soluzione è il gang scheduling che si compone di tre passaggi:

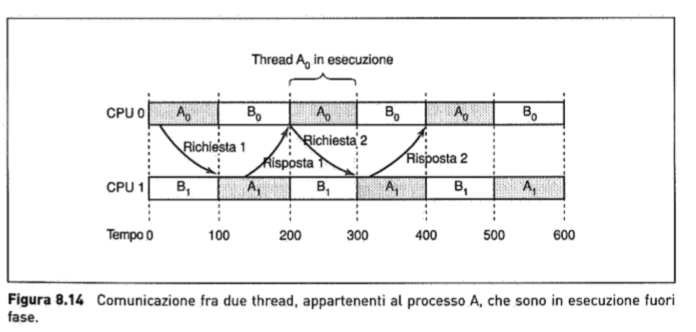
- i gruppi di thread correlati sono schedulati come un’unità (o gang);

- tutti i membri della gang sono in esecuzione simultaneamente, in differenti CPU in timesharing;

- tutti i thread della gang iniziano e terminano la loro porzione di tempo contemporaneamente.

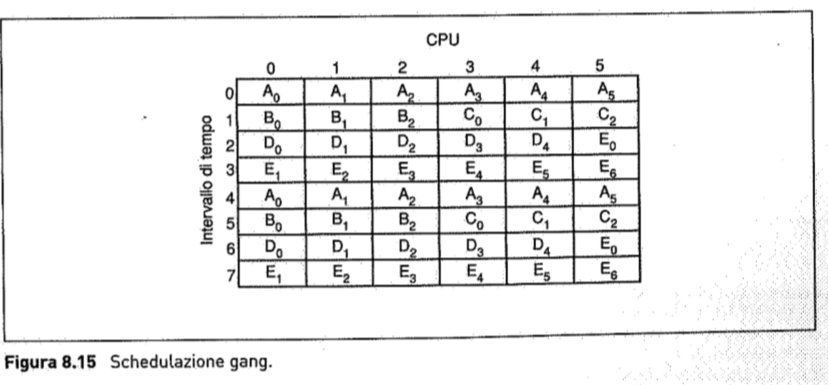
Questa schedulazione funziona perché tutte le CPU sono schedulate in sincronia: ciò significa che il tempo è suddiviso in quanti discreti, come nella Figura 8.14. All’inizio di ciascun nuovo quanto tutte le CPU sono di nuovo schedulate.

Se un thread si blocca, la sua CPU rimane inattiva fino allo fine del quanto.



Un esempio del funzionamento di questa schedulazione è presentato nella Figura 8.15. Abbiamo un multiprocessore con sei CPU, usate da cinque processi, da A ad E, per un totale di 24 thread pronti. Durante l’intervallo di tempo 0, sono schedulati ed eseguiti i thread da A0 ad A5, durante l’intervallo di tempo 1, i thread B0, B1, B2, C0, C1, C2. Durante l’intervallo 2 sono mandati in esecuzione i cinque thread di D e E0. I rimanenti sei thread del processo E sono eseguiti nell’intervallo di tempo 3. Quindi il ciclo si ripete, con l’intervallo 4, identico all’intervallo 0.

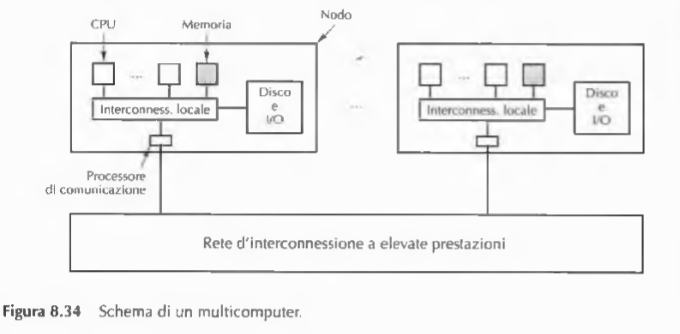
L’idea della schedulazione gang consiste nell’avere tutti i thread di un processo in esecuzione insieme, cosicché se uno di essi manda una richiesta ad un altro, quest’ultimo riceverà il messaggio e sarà in grado di rispondere quasi immediatamente. Nella Figura 8.15, poiché tutti i thread di A sono in esecuzione insieme durante un quanto, in tale intervallo di tempo possono mandare e ricevere un gran numero di messaggi, eliminando così il problema della Figura 8.14.



# Multicomputer a scambio di messaggi

Ogni CPU ha la propria memoria privata, non direttamente accessibile dalle altre. I programmi delle CPU di un multicomputer interagiscono per mezzo delle primitive *send* e *receive* e per lo scambio esplicito di messaggi, visto che non possono accedere alle rispettive memorie per mezzo di istruzioni di LOAD e STORE. Questa differenza cambia completamente il modello di programmazione. Ogni nodo di un multicomputer contiene una o più CPU, una certa dose di RAM (che si può pensare condivisa solo tra le CPU di quel nodo), un disco e/o altri dispositivi di I/O, più un processore di comunicazione. I processori di comunicazione sono collegati tramite una rete ad alta velocità.

Esistono diverse topologie, vari schemi di commutazione e algoritmi di instradamento, ma c’è un aspetto che tutti i multicomputer hanno in comune: quando un programma applicativo esegue una primitiva *send* il processore di comunicazione riceve una notifica e si incarica di trasmettere il blocco di dati dell’utente presso la macchina di destinazione (eventualmente dopo aver chiesto, e ricevuto, il permesso di farlo).



Questi sistemi sono anche chiamati Cluster di Computers o Cluster of Workstations (COW).

I multicomputer sono facili da costruire perché il componente base è un PC con una scheda di rete con alte performance.

Il segreto di ottenere alte performance in un multicomputer sta nel progetto della rete di interconnessione e delle schede di rete.

I messaggi sono spediti in un tempo nell’ordine dei μsec, mille volte in meno rispetto ad un accesso in memoria (ordine dei ns).

Il progetto è quindi più semplice da realizzare ed allo stesso tempo economico.

## 

## Topologia

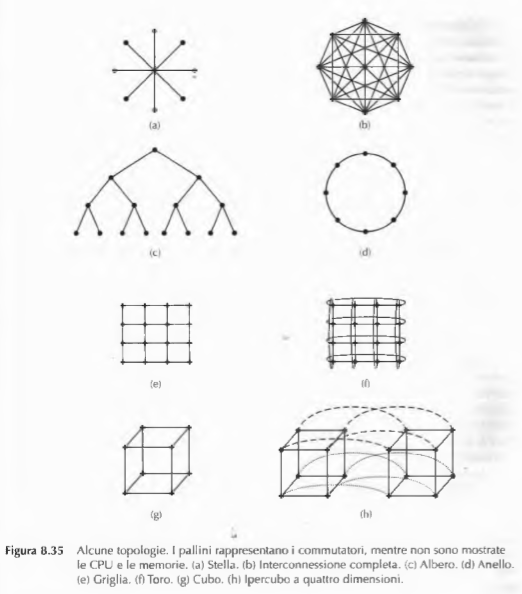
La topologia di una rete di interconnessione descrive il modo in cui sono disposti i collegamenti e i commutatori.

Il nodo base di un multicomputer consiste di una CPU, una memoria e una interfaccia di rete (talvolta anche un hard disk). Ogni nodo ha una interfaccia di rete con uno o due cavi (o fibra) che lo connette agli altri nodi oppure agli switch.

La Figura 8.35 mostra alcune topologie e prende in considerazione solo i collegamenti

(i segmenti) e i commutatori (i pallini).

Le memorie e le CPU non sono mostrate, ma si possono pensare collegate ai commutatori tramite interfacce.



Nella Figura 8.35(a) troviamo la configurazione a stella, in cui le CPU e le memorie sono collegate ai nodi esterni, mentre il nodo centrale è di pura commutazione. Sebbene sia un progetto semplice, la presenza di un nodo centrale commutatore può rivelarsi un collo di bottiglia per sistemi di grandi dimensioni. Inoltre il progetto è scadente anche dal punto di vista della tolleranza agli errori, visto che basta un difetto del commutatore centrale per distruggere l’intero sistema.

La Figura 8.35(b) mostra un interconnessione completa. In questo caso ogni nodo è collegato direttamente a ogni altro. Questo progetto massimizza la larghezza di banda ed è straordinariamente tollerante agli errori (potrebbero interrompersi sei collegamenti qualsiasi e la rete sarebbe ancora connessa). Sfortunatamente, al crescere dei nodi, richiede un numero di lati diventa presto intrattabile.

Un’altra topologia è l’albero, illustrato nella Figura 8.35(c). Il problema di questa topologia è che la larghezza di banda di bisezione è uguale alla capacità dei collegamenti.

La Figura 8.35(d) mostra la topologia ad anello in cui ogni nodo è connesso ad altri due nodi in ordine per formare un anello circolare (non sono necessari gli switch).

La griglia o mesh della Figura 8.35(e) è un progetto bidimensionale, utilizzato in molti sistemi complessi. È altamente regolare quindi altamente scalabile. Il percorso più lungo tra due nodi (diametro) aumenta come la radice quadrata del numero dei nodi.

Una variante della griglia è il toro della Figura 8.35(f), cioè una griglia con le estremità connesse. Non solo ha una miglior tolleranza agli errori rispetto alla griglia, ma presenta anche un diametro inferiore.

Un’altra topologia diffusa è una struttura tridimensionale regolare. Ogni nodo ha sei vicini, due lungo ciascun asse. I nodi sui bordi hanno collegamenti che li collegano al bordo opposto, proprio come in un toro.

Nella figura illustriamo un cubo 2 x 2 x 2, ma in generale si potrebbe avere un cubo k x k x k.

La Figura 8.35(h) mostra un ipercubo a quattro dimensioni costruito a partire da due cubi mediante il collegamento dei nodi corrispondenti.

Molti computer paralleli usano questa topologia perché il suo diametro cresce linearmente con la sua dimensione: diametro = log2 numero dei nodi.

## Cluster

Si tratta in genere di centinaia o migliaia di PC o di workstation (stazioni di lavoro) collegate per mezzo di schede di rete reperibili sul mercato.

Le tipologie dominanti sono due: i cluster centralizzati e quelli decentralizzati.

I primi sono cluster di workstation o di PC montati su grossi scaffali in un unico ambiente. Si tratta in genere di macchine omogenee.

I secondi sono formati da workstation e PC diffusi all’interno di un edificio o di un intero campus. Si tratta in genere di macchine eterogenee e dotate di un ricco equipaggiamento di periferiche.

**Schemi di switching**

Nei multicomputer sono usati due tipi di schemi di switching.

**Store-and-forward packet switching (connection-less)**

Ogni messaggio è suddiviso in pacchetti che sono poi inseriti nella rete. Il pacchetto raggiunge il nodo destinatario attraverso delle politiche di instradamento che dipendono da vari fattori (traffico dati, priorità, ecc…).

Questo sistema è flessibile ed efficiente ma ha il problema dell’incremento dei tempi di latenza lungo la rete di interconnessione.

**Circuit switching (connection-oriented)**

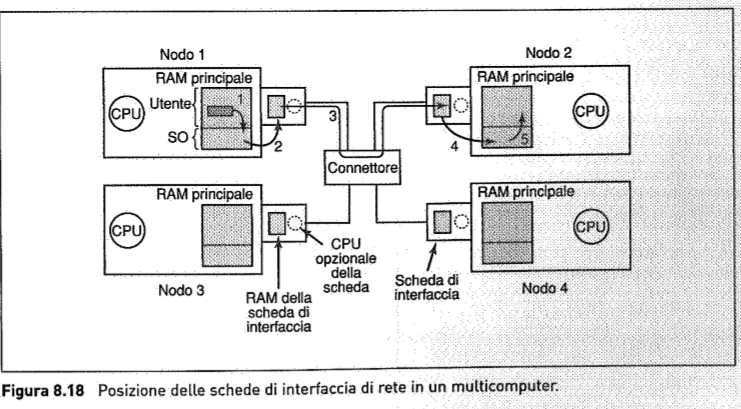
Nel secondo schema il primo switch stabilisce un collegamento fisico, attraverso tutti gli switch coinvolti, fino allo switch del nodo destinazione. Una volta che la connessione è creata i bit sono spediti alla massima velocità possibile (gli switch intermedi non hanno necessità di memorizzare i dati in transito).

Questo sistema richiede una fase di inizializzazione che prende tempo, ma una volta terminata il processo è velocissimo.

Una variante del circuit switching è il **wormhole routing**, spezza il pacchetto in sottopacchetti e permette a quest’ultimi di iniziare il tragitto prima che sia stato inizializzato il collegamento.

**Interfacce di rete**

In un multicomputer tutti i nodi hanno (almeno) una scheda di rete che permette di collegare il nodo alla rete di interconnessione. L’interfaccia contiene una RAM per memorizzare i pacchetti che entrano ed escono dal nodo.



La scheda d’interfaccia può avere uno o più canali DMA oppure una CPU completa (processori di rete).

I canali DMA possono copiare i pacchetti tra la scheda d’interfaccia e la RAM principale ad alta velocità, richiedendo i trasferimenti dei blocchi sul bus di sistema.

**Software di comunicazione di basso livello**

Il nemico delle prestazioni dei multicomputer è l’eccessiva copia di pacchetti (per esempio se sono necessarie troppe copie prima di spedire il pacchetto da un nodo ad un altro). Per evitare questo problema alcuni multicomputer mappano la scheda d’interfaccia nello spazio utente permettendo così al processo utente di inserire i dati direttamente nella scheda di rete (senza coinvolgere il kernel).

Questa soluzione pone due problematiche:

- la competizione dei processi concorrenti sullo stesso nodo che vogliono spedire pacchetti;

- la condivisione della scheda di rete tra il kernel, che magari vuole accedere ad un file system remoto, e il processo utente.

Riguardo la prima problematica, mappare la scheda di interfaccia nello spazio utente è realmente utile quando c’è un solo processo in esecuzione su ciascun nodo.

Riguardo la seconda è meglio utilizzare due schede di rete per ciascuna funzione.

**Software di comunicazione a livello utente**

I processi sulle diverse CPU di un multicomputer comunicano attraverso lo scambio di messaggi. Il sistema operativo fornisce le primitive che consentono ai processi utente di inviare (*send*) e ricevere messaggi (*receive*).

- *send(destination, &message\_pointer)*: spedisce il messaggio indirizzato da *message\_pointer* ad un processo identificato da *destination*, determinando il blocco del chiamante finché il messaggio non viene spedito;

- *receive(address, &message\_pointer)*: determina il blocco del chiamante finché non arriva un messaggio, quando arriva, è copiato nel buffer puntato da *message\_pointer* e il chiamante è sbloccato. Il parametro *address* specifica l’indirizzo su cui il ricevente è in ascolto.

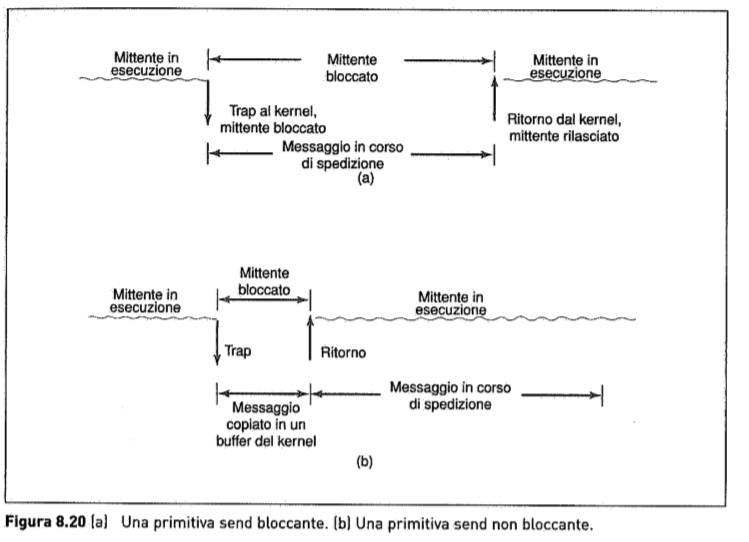
In un multicomputer statico il numero di CPU è noto a priori, quindi il campo *address* è formato dall’identificativo della CPU e dall’identificativo del processo o della porta sulla CPU selezionata.

**Primitive bloccanti e non bloccanti**

Le primitive *send* e *receive* possono essere bloccanti (sincrone) o non bloccanti (asincrone).

Se una *send* è bloccante, mentre il messaggio viene spedito il processo mittente si blocca; in altre parole l’istruzione che segue la chiamata della *send* non viene eseguita finché il messaggio non è stato completamente spedito. In modo analogo una chiamata ad una *receive* non termina fino a quando il messaggio non è stato ricevuto.

Se una primitiva è non bloccante, essa restituisce il controllo al chiamante immediatamente dopo la sua esecuzione e prima che il messaggio venga realmente spedito.



La scelta tra non bloccanti o bloccanti dipende dal progettista del sistema sebbene in pochi sistemi siano entrambe disponibili.

Il vantaggio in termini di performance offerto dalle chiamate non bloccanti viene controbilanciato da un serio svantaggio: il mittente non può modificare il buffer che contiene il messaggio fino a quando il messaggio non è spedito e, peggio ancora, non conosce quando la trasmissione sarà terminata e riusare il buffer senza fare danni.

Il processo mittente può eseguire quindi:

1) una spedizione bloccante e mantenere bloccata la CPU;

2) una spedizione non bloccante con copia (la CPU spreca tempo solo per eseguire una copia);

3) una spedizione non bloccante con interrupt (programmazione difficile);

4) una spedizione non bloccante con copia su scrittura (la CPU spreca tempo anche per la copia di fine processo oltre le scritture richieste).

In un sistema multi-thread la prima è la scelta migliore: mentre il thread che esegue la *send* è bloccato, gli altri continuano a lavorare.

Il processo destinatario può utilizzare una *receive* non bloccante: indica semplicemente al kernel dove è il buffer.

L’arrivo di un messaggio può essere gestito:

- tramite interrupt, ma sono difficili da programmare e molto lenti;

- richiamando una procedura *poll* che restituisce se ci sono messaggi in attesa di essere letti;

- attraverso la creazione automatica di un thread (chiamato thread pop-up) che finito il suo compito muore spontaneamente;

- attraverso un interrupt che attiva nella ISR il codice di gestione (questo schema è una versione ibrida dei precedenti che sfrutta l’idea di un thread pop-up senza creare alcun thread, migliorando così le performance, e si chiama messaggi attivi).

**Remote Procedure Call (RPC)**

Sebbene il modello a scambio di messaggi è un sistema conveniente per un sistema operativo multicomputer soffre di un grave difetto: tutte le comunicazioni, e quindi i programmi, utilizzano l’I/O.

Un approccio differente è quello che consente ai programmi di chiamare procedure che si trovano su le altre CPU del multicomputer in modo indipendente dall’I/O.

Quando un processo sulla macchina 1 chiama una procedura sulla macchina 2, il processo su 1 è sospeso, e l’esecuzione della procedura chiamante avviene su 2. L’informazione può essere trasportata dal chiamante al chiamato attraverso i parametri, e può tornare indietro attraverso i risultati della procedura.

Questa tecnica è detta RPC (Remote Procedure Call, chiamata di procedura remota) ed è alla base del software multicomputer.

La procedura chiamante è nota come client, quella chiamata come server. Per chiamare una procedura remota il programma client deve avere una piccola procedura chiamata client stub.

Analogamente, il programma server è legato ad una procedura chiamata server stub. Così facendo si nasconde il fatto che la chiamata di procedura dal client al server non è locale.

I passi effettivi per fare una RPC compaiono nella Figura 8.21:

- il passo 1 è la chiamata da parte del client del client stub, che è una chiamata di procedura locale, con i parametri messi sullo stack nel modo consueto;

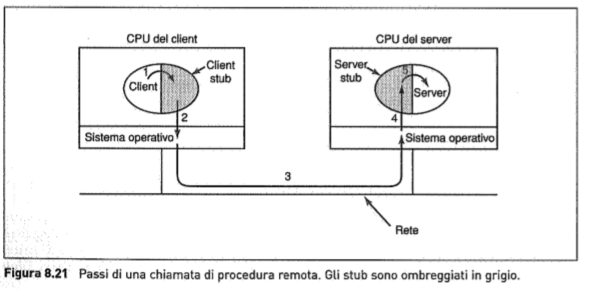
- il passo 2 consiste nell’impacchettamento dei parametri in un messaggio e nell’effettuare una chiamata di sistema per spedire il messaggio (impacchettare i parametri è detto marshaling);

- nel passo 3, il kernel spedisce il messaggio dalla macchina client a quella server;

- nel passo 4 il kernel passa il pacchetto in arrivo al server stub;

- nel passo 5 il server stub chiama la procedura del server.

La risposta effettua lo stesso cammino in direzione opposta.

****

Si osservi che la procedura client non effettua altro che una chiamata di procedura normale (cioè locale) al client stub, che ha lo stesso nome della procedura server; poiché la procedura del client e il client stub sono nello stesso spazio degli indirizzi, i parametri sono passati nel modo usuale. Allo stesso modo, la procedura del server è chiamata da una procedura nel suo spazio degli indirizzi, con i parametri che si aspetta; per la procedura del server, non c’è nulla di inusuale.

**Problemi implementativi**

- parametri puntatore: il passaggio dei puntatori è impossibile, perché il client ed il server hanno un diverso spazio degli indirizzi;

- array come parametri: questo problema capita con i linguaggio debolmente tipati, in cui è possibile scrivere una procedura che calcola il prodotto scalare di due array senza specificare la lunghezza, ad esempio decidendo che esiste un valore speciale, noto solo a chiamante e chiamato, che indica la fine dell’array. In questo caso, è impossibile che un client stub possa fare il marshaling dei parametri, se non ne conosce la dimensione.

- tipi di dato: non è sempre possibile dedurre i tipi dei parametri, nemmeno con una specifica normale, o dal codice stesso (funzioni polimorfe). Ad esempio la *printf*.

- variabili globali: la procedura chiamante e quella chiamata non possono comunicare per mezzo di variabili globali in quanto non esiste un contesto comune.

**Memoria condivisa distribuita**

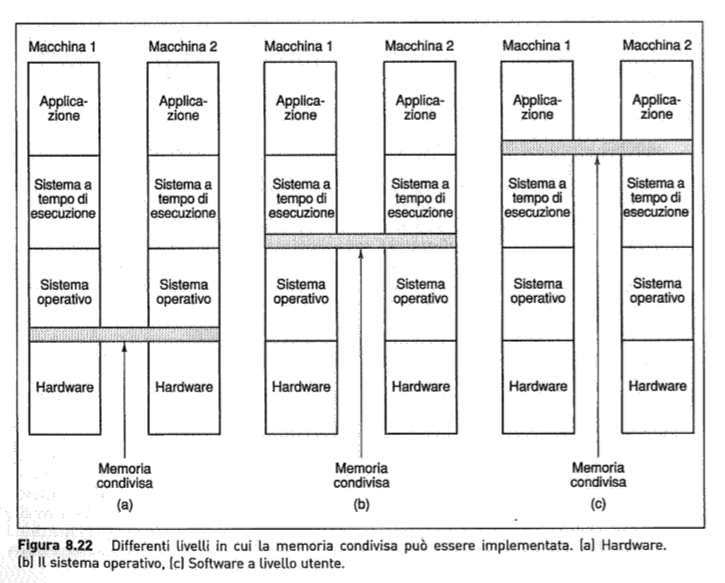
Benché RPC sia interessante, molti programmatori preferiscono un modello di memoria condivisa, e lo vorrebbero utilizzare anche su un multicomputer. Singolarmente, è possibile preservare l’illusione della memoria condivisa anche quando non esiste in realtà, utilizzando una tecnica chiamata DSM (Distributed Shared Memory).

Con DSM ciascuna pagina si trova in una delle memorie della Figura 8.21, ogni macchina ha la

propria memoria virtuale, e le proprie tabelle delle pagine.

Quando una CPU effettua una LOAD o una STORE su una pagina che non ha, avviene una trap al sistema operativo, che quindi, localizza la pagina, e chiede alla CPU che la possiede correntemente di invalidare la pagina, e spedirla sulla rete di interconnessione. Quando arriva, la pagina viene mappata e viene fatta ripartire l’istruzione che aveva provocato il fault; in effetti, il sistema operativo sta soddisfacendo i fault di pagina della RAM remota, invece che dal disco locale.

La differenza tra una vera memoria condivisa e DSM è illustrata nella Figura 8.22. Nella (a) si vede un vero multiprocessore con memoria fisica condivisa implementata tramite hardware; nella (b) si vede DSM implementata dal sistema operativo; nella (c) troviamo DSM implementata da livelli superiori del software.



**Scheduling**

In un multiprocessore tutti i processi risiedono nella stessa memoria. Quando una CPU finisce il suo task corrente, seleziona un processo e lo esegue: in linea di principio, tutti i processi sono potenzialmente candidati.

Su un multicomputer la situazione è piuttosto diversa: ciascun nodo, infatti, ha memoria e insieme di processi propri, e la CPU 1 non può improvvisamente decidere di eseguire un processo che si trova sul nodo 4 senza prima fare una certa quantità di lavoro per acquisirlo. Questa differenza significa che la schedulazione sui multicomputer è più facile, ma l’allocazione dei processi ai nodi è più importante.

La schedulazione dei multicomputer è in un certo senso simile a quella dei multiprocessori, ma non tutti gli algoritmi dei primi si applicano ai secondi. L’algoritmo più semplice usato per i multiprocessori, mantenere una sola lista centralizzata dei processi pronti, in ogni caso non funziona, poiché ciascun processo si può eseguire solo sulla CPU dove si trova correntemente, e quando si crea un nuovo processo, è possibile scegliere dove metterlo, ad esempio per bilanciare il carico.

Poiché ciascun nodo ha i suoi processi, si può usare qualsiasi algoritmo di schedulazione locale. È anche possibile, tuttavia, usare la schedulazione gang, come la si utilizza su un multiprocessore, perché richiede soltanto un accordo iniziale su quali processi eseguire nei diversi intervalli di tempo, e una coordinazione dell’inizio degli intervalli di tempo.

**Bilanciamento del carico**

Proprio perché esistono poche possibilità di controllare un processo assegnato ad un nodo, è importante decidere quale processo debba essere eseguito su un determinato nodo. Quindi, è opportuno studiare come sia concretamente possibile assegnare i processi ai nodi: gli algoritmi e l’euristica per effettuare tale assegnazione, sono noti come algoritmi di allocazione del processore, e nel corso degli anni, ne sono stati proposti molti, che differiscono in ciò che ipotizzano come noto e negli obiettivi.

Le proprietà di un processo che potrebbero essere note comprendono: il fabbisogno di CPU, l’utilizzo di memoria e la quantità di comunicazione con ogni altro processo.

I possibili obiettivi sono: la minimizzazione dei cicli di CPU sprecati per la carenza di lavoro locale, la minimizzazione della larghezza di banda di comunicazione totale, e condizioni eque per gli utenti e i processi.

**Virtualizzazione**

La virtualizzazione consente a un singolo computer “reale” di ospitare più computer “virtuali”, chiamati macchine virtuali. Ciascuna macchina virtuale è un computer “virtuale” in cui è possibile eseguire un proprio sistema operativo e dei propri servizi e applicazioni.

La virtualizzazione introduce importanti vantaggi:

- forte isolamento tra le macchine (isolamento dei malfunzionamenti);

- minore spazio occupato;

- minore consumo;

- minore calore da dissipare;

- maggiore manutenibilità;

- possibilità di creare dei checkpoint (punti di ripristino);

- possibilità di far girare applicazioni legacy su ambienti obsoleti;

- possibilità di effettuare test delle applicazioni su differenti sistemi operativi senza disporre dell’hardware fisico necessario.

Ogni CPU con una modalità kernel e una modalità utente ha un insieme di istruzioni che si comporta in modo diverso quando viene eseguito in modalità kernel rispetto a quando viene eseguito in modalità utente. Queste istruzioni sono dette istruzioni sensibili. Esistono poi un insieme di istruzioni (chiamate istruzioni privilegiate) che, quando vengono eseguite in modalità utente, causano un trap.

Una macchina è virtualizzabile solo se le istruzioni sensibili sono un sottoinsieme di quelle privilegiate.

A differenze del IBM/370, l’Intel 386 possiede delle istruzioni sensibili che sono ignorate se eseguite in modalità utente (come ad esempio la POPF).

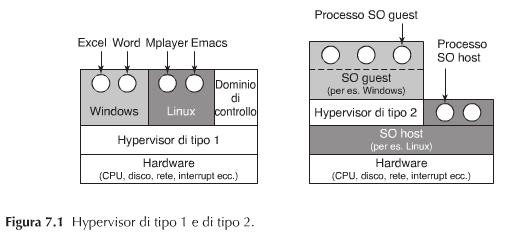
Quindi Intel 386 e i suoi successori non possono essere virtualizzati con un hypervisor di tipo 1.

Il problema viene risolto nel 2005, quando Intel e AMD introducono la virtualizzazione sulle loro CPU. Sulle CPU Intel la tecnologia è chiamata Virtualization Technology (VT) mentre sulle macchine AMD è denominata Secure Virtual Machine (SVM).

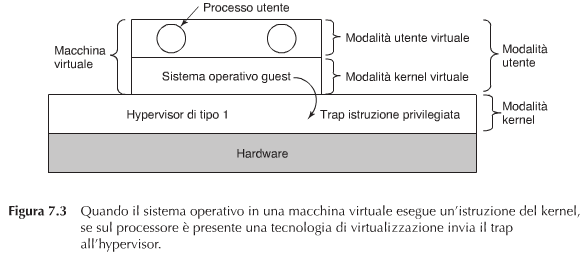
Sebbene entrambe si ispirano al funzionamento dell’IBM/370 hanno delle piccole differenze. L’idea fondamentale è creare dei contenitori all’interno dei quali eseguire le macchine virtuali. Quando in un computer si avvia un sistema operativo guest, questo continua a funzionare fino a che non solleva un’eccezione e passa un trap all’hypervisor.

**Hypervisor di tipo 1**

L’hypervisor di tipo 1 (illustrato nella Figura 7.1(a)) è nel sistema operativo reale (SO host) e gira in modalità kernel. Il suo compito è di supportare le macchine virtuali (SO guest).



La macchina virtuale è eseguita come un processo utente in modalità utente, non può eseguire istruzioni sensibili anche se pensa di essere in modalità kernel (modalità virtuale del kernel).



Quando il SO guest esegue una istruzione sensibile:

- se la CPU non ha la VT l’istruzione fallisce e il sistema operativo crasha;

- altrimenti avviene una trap nel kernel e l’hypervisor può vedere se l’istruzione è stata inviata da una VM del SO guest, in questo caso la esegue, o da un programma utente, in questo caso simula il comportamento dell’hardware reale.

**Hypervisor di tipo 2**

Nell’hypervisor type 2, illustrato nella Figura 7.1(b), è un programma utente che interpreta le istruzioni della VM e le traduce sul SO della macchina reale. È la soluzione che permette la virtualizzazione alle CPU Intel senza tecnologia VT (es. 386..).

VMware è un hypervisor di tipo 2 ed è eseguito come programma utente su un qualsiasi SO host.

Quando si avvia per la prima volta, si comporta come un computer senza SO e cerca di installare il SO guest nel suo disco virtuale.

Per eseguire un programma si utilizza una tecnica nota come traduzione binaria:

- esegue la scansione del codice alla ricerca dei blocchi base, cioè quei blocchi che terminano con istruzioni di cambio flusso (es. JMP, CALL, trap,...);

- ricerca nei blocchi le istruzioni sensibili e le traduce in una procedura VMware;

- il blocco è messo nella cache di VMware e può essere eseguito alla velocità della macchina fisica (istruzioni tradotte a parte).

**Confronto tra hypervisor**

Negli hypervisor tipo 2 tutte le istruzioni sensibili sono sostituite da chiamate a procedure che ne emulano il comportamento. Il SO guest non invierà mai nessuna istruzione sensibile alla macchina fisica.

L’approccio “trap-and-emulate”(tipo 1) adottato dagli hardware VT genera troppi trap ed un eccessivo overhead di gestione, mentre la traduzione delle istruzioni sensibili è più efficiente.

Alcuni hypervisor di tipo 1 effettuano la traduzione binaria (anche se non serve) comportandosi come quelli di tipo 2.

**Paravirtualizzazione**

Gli hypervisor tipo 1 e 2 funzionano senza modifiche al SO guest, ma con performance non eccellenti.

Un diverso approccio prevede la modifica del codice sorgente del SO guest: invece di eseguire istruzioni sensibili si effettuano chiamate di procedure definite dall’hypervisor. Quindi l’hypervisor definisce una interfaccia, cioè delle API (Application Program Interface), che i sistemi operativi guest possono attivare. Questo trasforma di fatto l’hypervisor in un microkernel e il SO guest modificato viene detto paravirtualizzato. Le performance ovviamente migliorano poiché le trap si trasformano in system call.