

12. Si calcoli il codice hash dei seguenti simboli sommando le lettere ( $A = 1, B = 2$  e così via) e calcolando il modulo tra il risultato e la dimensione della tabella hash. La tabella hash ha 19 posizioni, numerate da 0 a 18.  
els, jan, jelle, maaike  
Ciascuno di questi simboli genera un unico valore hash? In caso negativo, com'è possibile gestire il problema della collisione?
13. Il metodo della codifica hash descritto nel testo inserisce in una lista concatenata tutti gli elementi che hanno lo stesso codice hash. Un metodo alternativo è quello di avere un'unica tabella a  $n$  posizioni, in cui ciascuna ha lo spazio per memorizzare una chiave e il suo valore (o un puntatore a quest'ultimo). Se l'algoritmo di hash indica una posizione già occupata, si riprovi con un secondo algoritmo di hash. Se anche questa posizione è già occupata, se ne utilizzi un altro, continuando allo stesso modo finché non se ne trovi una vuota. Se la frazione di posizioni che sono occupate è  $R$ , quanti tentativi saranno necessari, in media, per poter inserire un nuovo simbolo?
14. Grazie ai progressi tecnologici un giorno potrebbe essere possibile collocare su un chip migliaia di CPU identiche, ciascuna delle quali con poche parole di memoria locale. Se tutte le CPU possono leggere e scrivere tre registri condivisi com'è possibile implementare una memoria associativa?
15. x86 ha un'architettura segmentata, composta da segmenti indipendenti. Un assemblatore per questa macchina potrebbe avere una pseudosistruzione SEG N, per indicare all'assemblatore di collocare il codice successivo nel segmento N. Questo schema avrebbe qualche influenza su ILC?
16. I programmi spesso si collegano a più DLL. Non sarebbe più efficiente inserire tutte le procedure in un'unica grande DLL e collegarla al programma?
17. È possibile mappare una DLL negli spazi degli indirizzi virtuali di due processi in due indirizzi virtuali diversi? Se sì, quali problemi sorgono? Possono essere risolti? Altrimenti, che cosa si può fare per eliminarli?
18. Un modo per effettuare il collegamento (statico) è il seguente. Prima di scandire la libreria il linker costruisce una lista delle procedure richieste, cioè una lista composta dai nomi definiti come EXTERN nei moduli che sta collegando. In seguito il linker analizza linearmente tutta la libreria, estraendo ogni procedura che si trova nella lista di quelle necessarie. Questo schema può funzionare? Altrimenti, perché no, e che rimedi si possono apporre?
19. È possibile utilizzare un registro come parametro attuale in una chiamata di macro? E una costante? Perché sì o perché no?
20. Si sta implementando un assemblatore con macro. Per ragioni estetiche il capo progetto ha deciso che le definizioni di macro non devono precedere le loro chiamate. Quali conseguenze ha questa decisione sull'implementazione?
21. Si pensi a un modo per far entrare un assemblatore con macro in un ciclo infinito.
22. Un linker legge cinque moduli, le cui lunghezze sono rispettivamente 200, 800, 600, 500 e 700. Se i moduli vengono caricati in quest'ordine quanto valgono le costanti di rilocazione?
23. Si scrivano due routine per gestire una tabella di simboli: `enter(symbol, value)` e `lookup(symbol, value)`. La prima inserisce i nuovi simboli nella tabella, mentre la seconda permette di cercare un simbolo al suo interno. Si utilizzi una codifica hash.
24. Si ripeta l'esercizio precedente senza utilizzare una tabella hash: dopo che l'ultimo simbolo viene inserito, si ordini la tabella e si utilizzi un algoritmo di ricerca dicotomica per trovare i simboli.
25. Si scriva un semplice assemblatore per il calcolatore Mic-I del Capitolo 4. Si fornisca la possibilità di assegnare durante la fase di assemblaggio valori costanti ai simboli e un modo per assemblare una costante in una parola della macchina (queste dovrebbero essere pseudoistruzioni).
26. Si aggiunga la funzionalità delle macro all'assemblatore dell'esercizio precedente.



## Architetture per il calcolo parallelo

Benché i calcolatori diventino sempre più veloci, le aspettative dei loro utenti crescono almeno altrettanto rapidamente. L'ambizione degli astronomi è la simulazione dell'intera storia dell'universo, dal big bang alla fine dei tempi. L'industria farmaceutica gradirebbe progettare sui propri computer medicine per la cura mirata di determinate malattie, invece di sacrificare intere legioni di ratti. I progettisti aeronautici sarebbero in grado di scoprire prodotti dotati di maggiore efficienza energetica se solo i calcolatori potessero farsi carico di tutto il lavoro, senza bisogno di costruire prototipi reali da valutare nella galleria del vento. È chiaro che, anche se è ormai disponibile molta potenza di calcolo, per un gran numero di utenti, e in special modo per gli scienziati e per l'industria, questa potenza non è mai abbastanza.

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 di fascia alta e le prospettive di riuscire a far muovere gli elettroni e i fotoni ancora più velocemente sono scarse. I problemi di dissipazione del calore fanno sì che i supercalcolatori diventino sempre più sofisticati condizionatori d'aria. Infine, le dimensioni dei transistor continueranno a diminuire e raggiungeranno presto il punto in cui conterranno così pochi atomi che gli effetti della meccanica quantistica (per esempio il principio di indeterminazione di Heisenberg) porranno seri problemi.

In conseguenza di tutto ciò, gli architetti degli elaboratori si rivolgono sempre più spesso al calcolo parallelo per trattare problemi di dimensioni sempre crescenti. Anche se non è possibile realizzare un calcolatore con una sola CPU e un ciclo di 0,001 ns, sarebbe sicuramente possibile costruirne uno con 1000 CPU, ciascuna con ciclo di clock di 1 ns. Sebbene il secondo progetto si avvalga di CPU più lente, in teoria la sua capacità totale di calcolo è la stessa del primo. Questa è l'idea cui sono affidate le speranze per il futuro.

Il parallelismo può essere introdotto a vari livelli. A quello più basso, può essere incorporato nel chip della CPU attraverso un progetto superscalare a pipeline con molteplici unità funzionali; oppure può essere introdotto con l'adozione d'istruzioni basate su parole molto lunghe e dotate di parallelismo隐式. La CPU può essere equipaggiata con caratteristiche speciali che le consentano di gestire il controllo di più thread alla volta, ed è anche possibile integrare più CPU sullo stesso chip. Tutti questi strati, se usati congiuntamente, possono produrre un incremento delle prestazioni per un fattore 10, al più, rispetto a un progetto puramente sequenziale.

A livello successivo, è possibile aggiungere al sistema delle schede dotate di CPU che arricchiscono le sue capacità di calcolo. In genere queste CPU *plug in* (cioè a innesto) svolgono funzioni specializzate, come l'elaborazione di pacchetti di rete, l'elaborazione multimediale o la crittografia. Consentono un fattore d'incremento delle prestazioni compreso tra 5 e 10, ma solo nel caso di applicazioni specializzate.

D'altra parte, l'unico modo per ottenere un incremento dell'ordine delle centinaia, delle migliaia o dei milioni, è di replicare il numero delle CPU, facendole cooperare in modo efficiente. Questa è l'idea alla base dei multiprocessori e dei multicomputer (*cluster di computer*).

È evidente che il montaggio di migliaia di processori in un unico grande sistema costituisce di per sé un problema che va risolto.

Va detto infine che oggi è possibile strutturare su Internet intere organizzazioni di computer come reti di calcolo (griglie) con connessioni lasche. Sono sistemi alle prime armi, ma manifestano un grosso potenziale per il futuro.

Due CPU o elementi di calcolo contigui che svolgono calcoli in modo molto interattivo e la cui connessione ha un'elevata larghezza di banda e un ritardo trascurabile, si dicono legati strettamente (*tightly coupled*). Viceversa si dicono legati debolmente (*loosely coupled*) se si trovano molto distanti, hanno una banda ridotta, un ritardo elevato e svolgono i loro calcoli in modo poco interattivo. In questo capitolo ci interessiamo ai principi progettuali delle varie forme di parallelismo e ne analizziamo una varietà di esempi. Cominciamo dai sistemi a legame più stretto, cioè quelli che usano il parallelismo nel chip, spostandoci gradualmente verso sistemi con legame più debole, per finire con lo spendere qualche parola sul *grid computing* ("calcolo su griglie di computer"). La Figura 8.1 mostra lo spettro delle possibilità (da legame più stretto a più debole).

La questione del parallelismo è oggetto di intensa ricerca a tutti i livelli, perciò nel corso del capitolo segnaliamo un gran numero di riferimenti bibliografici, rivolgendo particolare attenzione agli articoli più recenti. Diverse conferenze e riviste pubblicano articoli sull'argomento e la letteratura cresce rapidamente.

## 8.1 Parallelismo nel chip

Un modo per incrementare la produttività di un chip è di fargli svolgere più compiti alla volta: in altre parole, sfruttare il parallelismo. In questo paragrafo ci interessiamo ai modi in cui si può ottenere un incremento delle prestazioni tramite il parallelismo a livello del chip, compreso il parallelismo del livello delle istruzioni, il multithreading e la coabitazione di più CPU sullo stesso chip. Sono tecniche abbastanza diverse e ciascu-

na dà un proprio contributo. Il loro comune obiettivo è quello di svolgere più attività nello stesso tempo.

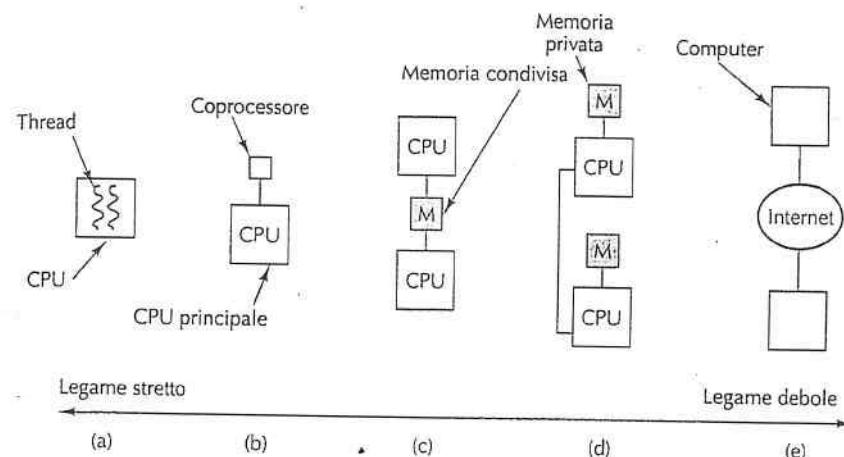


Figura 8.1 (a) Parallelismo nel chip. (b) Coprocessore. (c) Multiprocessore. (d) Multicomputer. (e) Griglia.

### 8.1.1 Parallelismo a livello delle istruzioni

A livello delle istruzioni, un modo per ottenere il parallelismo è di emettere più istruzioni per ciclo di clock. Ci sono due tipi di CPU a emissione multipla: processori superscalari e processori VLIW. Vi abbiamo già accennato precedentemente, ma può essere utile ritornare sull'argomento.

Abbiamo già visto le CPU superscalari (Figura 2.6). Nel caso più generale, c'è un momento lungo la pipeline in cui un'istruzione è pronta per essere eseguita. Le CPU superscalari sono in grado di emettere, verso le unità di esecuzione, più istruzioni in un solo ciclo. Il numero d'istruzioni emesse dipende dal progetto del processore e dalle circostanze contingenti. L'hardware determina il numero massimo d'istruzioni che può essere emesso, in genere da due a sei. In ogni caso un'istruzione non verrà emessa se attende un calcolo che non è stato ancora completato o se necessita di un'unità funzionale non disponibile.

L'altra forma di parallelismo a livello delle istruzioni è quello dei processori VLIW (*Very Long Instruction Word*, "con parola d'istruzione molto lunga"). Nella loro forma originaria, le macchine VLIW avevano davvero parole molto lunghe per contenere istruzioni che usavano diverse unità funzionali. Si consideri per esempio la Figura 8.2(a), in cui la macchina dispone di cinque unità funzionali e può eseguire simultaneamente due operazioni intere, una in virgola mobile, un caricamento e una memorizzazione. Un'istruzione VLIW per questa macchina conterebbe cinque codici operativi e cinque

coppie di operandi: un codice e una coppia di operandi per ogni unità funzionale. Se ci sono 6 bit per codice operativo, 5 bit per registro e 32 bit per gli indirizzi di memoria, le istruzioni possono raggiungere facilmente i 134 bit, il che fa di loro istruzioni abbastanza lunghe.

Questo progetto si rivelò troppo rigido perché non tutte le istruzioni riuscivano a sfruttare tutte le unità funzionali, il che conduceva all'uso di molte inutili NO-OP, come illustrato nella Figura 8.2(b). Perciò le macchine VLIW moderne dispongono di una modalità di costruzione di pacchetti (*bundle*) d'istruzioni, conclusi per esempio dal bit di "fine pacchetto", come mostrato nella Figura 8.2(c). Il processore è in grado di effettuare il fetch e l'emissione di un intero pacchetto per volta. Spetta al compilatore preparare pacchetti d'istruzioni compatibili.

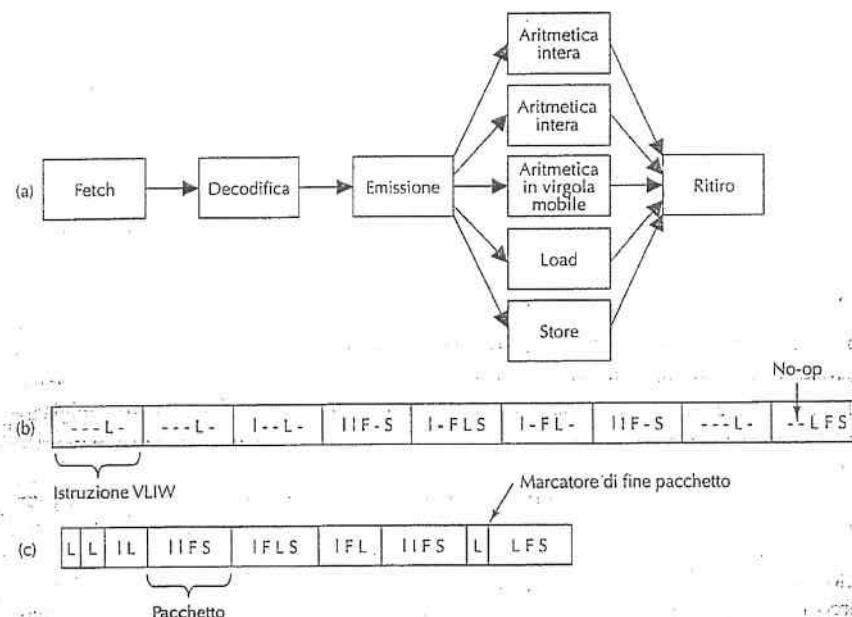


Figura 8.2 (a) Pipeline della CPU. (b) Sequenza d'istruzioni VLIW. (c) Flusso d'istruzioni con marcatori di pacchetto.

A tutti gli effetti, questa tecnica sposta dall'esecuzione alla compilazione il difficile compito di stabilire quali istruzioni possono essere avviate alla esecuzione contemporanea. Questa scelta consente di avere un hardware più semplice e veloce e, poiché un compilatore può far durare l'ottimizzazione per tutto il tempo necessario, permette di assemblare pacchetti d'istruzioni migliori di quanto potrebbe fare l'hardware durante l'esecuzione. Per contro, questo progetto comporta un cambio radicale nelle architetture

delle CPU che sarà difficile da far accettare, come dimostrato dalla scarsa accoglienza suscitata da Itanium, eccezione fatta per applicazioni di nicchia.

Val la pena notare qui che il parallelismo a livello delle istruzioni non è la sola forma di parallelismo di basso livello. C'è infatti il parallelismo a livello della memoria, secondo cui diverse operazioni di memoria vengono svolte contemporaneamente (Chou et al., 2004).

### CPU VLIW TriMedia

Nel Capitolo 5 abbiamo studiato l'Itanium-2, un esempio di CPU VLIW. Analizziamo ora TriMedia, un processore VLIW molto diverso e progettato da Philips, la compagnia olandese che ha anche inventato i CD audio e i CD-ROM. TriMedia è stato pensato come processore integrato per applicazioni intensive di tipo audio, video o su immagini, nei lettori CD, DVD o Mp3, nei registratori CD o DVD, nelle console TV interattive, nelle macchine fotografiche e videocamere digitali, e così via. Proprio perché destinato a questo tipo di applicazioni, non sorprende che TriMedia differisca molto da Itanium-2, che è invece una CPU per uso generale concepita per i server di fascia alta.

TriMedia è un vero processore VLIW in cui ogni istruzione può contenere cinque operazioni. In condizioni ideali, a ogni ciclo di clock viene avviata l'esecuzione di un'istruzione e l'emissione delle cinque operazioni. Il clock ha una frequenza di 266 o 300 MHz, che però equivale a una frequenza cinque volte maggiore, visto che può eseguire cinque operazioni per ciclo. In seguito ci concentreremo sull'implementazione TM3260 di TriMedia (esistono anche altre versioni, le cui differenze sono però trascurabili).

La Figura 8.3 illustra un'istruzione tipica. Si va dalle istruzioni intere standard di 8, 16 e 32 bit, alle istruzioni in virgola mobile IEEE 754, fino alle istruzioni multimediali parallele. Grazie all'emissione di cinque istruzioni multimediali per ciclo, TriMedia è sufficientemente veloce da decodificare un flusso di dati proveniente da una videocamera senza ridurre la dimensione delle immagini, né la frequenza di acquisizione.

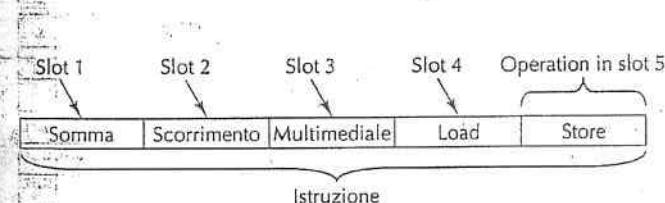


Figura 8.3 Tipica istruzione TriMedia, con cinque operazioni possibili.

TriMedia ha una memoria orientata al byte, con i registri di I/O mappati nello spazio di memoria. Le mezze parole (16 bit) e le parole intere (32 bit) devono essere allineate lungo le loro estremità (ovvero devono iniziare in byte di indirizzo pari o – per le parole intere – multiplo di 4). L'ordinamento dei byte può essere big-endian o little-endian, a seconda di un bit di PSW che può essere impostato dal sistema operativo. Questo bit ha

effetto solo sul modo in cui operano i trasferimenti di caricamento e memorizzazione tra i registri e la memoria. La CPU contiene una cache specializzata associativa a 8 vie, con linee di 64 byte sia per la cache delle istruzioni, sia per quella dei dati. La cache delle istruzioni è di 64 KB, quella dei dati è di 16 KB. Ci sono 128 registri di 32 bit per uso generale. Il registro R0 è cablato al valore 0, R1 a 1. I tentativi di modificare uno di questi registri hanno sulla CPU l'effetto di un infarto. I restanti 126 registri sono tutti equivalenti dal punto di vista funzionale e possono essere usati per qualsiasi scopo. Inoltre esistono quattro registri di 32 bit specializzati: il program counter, PSW e due registri legati agli interrupt. Infine, viene utilizzato un registro di 64 bit per contare i cicli dall'ultimo reset della CPU. A 300 MHz, ci vogliono circa 2000 anni perché il contatore vada in overflow. Il TriMedia TM3260 ha 11 diverse unità funzionali per svolgere le operazioni aritmetiche, logiche e di controllo del flusso, oltre a una per il controllo di cache che non prenderemo in considerazione. Le prime due colonne della Figura 8.4 indicano il nome dell'unità e ne danno una breve descrizione, la terza specifica il numero di copie hardware dell'unità e la quarta dà la latenza dell'unità, cioè il numero di cicli di clock che impiega per completare l'operazione. Notiamo qui che tutte le unità funzionali sono a pipeline, eccetto la unità VM (in virgola mobile) per la divisione/radice quadrata. La latenza indicata nella tabella stabilisce il tempo necessario alla disponibilità del risultato di un'operazione, ma a ogni ciclo è possibile iniziare una nuova operazione. Per esempio, tre istruzioni consecutive possono contenere ciascuna due carichi, così in un certo istante saranno in esecuzione sei operazioni di caricamento, ma a stadi differenti.

Infine, le ultime cinque colonne mostrano le possibili corrispondenze tra posizione delle operazioni nell'istruzione e le unità funzionali. Per esempio, l'operazione di confronto in virgola mobile deve occupare necessariamente il terzo posto di un'istruzione.

Unità	Descrizione	#	Latenza	1	2	3	4	5
Costante	Operazioni immediate	5	1	x	x	x	x	x
ALU intera	Aritmetica a 32 bit, operazioni booleane	5	1	x	x	x	x	x
Scorrimento	Scorrimento di più bit	2	1	x	x	x	x	x
Load/Store	Operazioni di memoria	2	3	x	x			
MUL Int/VM	Moltiplicazione intera e VM a 32 bit	2	3		x	x		
ALU VM	Aritmetica VM	2	3	x			x	
Confronto VM	Confronti VM	1	1			x		
sqr/div VM	Divisione e radice quadrata VM	1	17		x			
Salti	Controllo del flusso	3	3	x	x	x		
ALU DSP	Aritmetica multimediale a due o quattro operandi (16 o 8 bit)	2	3	x		x		x
MUL DSP	Moltiplicazioni multimediale a due o quattro operandi (16 o 8 bit)	2	3		x	x		

Figura 8.4 Unità funzionali di TM3260; numero, latenza e posizioni che possono avere nelle istruzioni.

L'unità costante si usa per le operazioni immediate, come il caricamento in un registro di un numero contenuto nell'operazione stessa. L'ALU intera svolge la somma, la sottrazione e le usuali operazioni booleane, e inoltre impacchetta/spacchetta le operazioni. L'unità di scorrimento può traslare il contenuto di un registro di un certo numero di bit in entrambe le direzioni.

L'unità load/store effettua il fetch delle parole di memoria verso i registri o viceversa. TriMedia è essenzialmente una CPU RISC potenziata, e le operazioni normali lavorano sui registri, mentre soltanto l'unità load/store è usata per gli accessi alla memoria, con trasferimenti di 8, 16 o 32 bit.

L'unità per la moltiplicazione gestisce sia i prodotti tra interi, sia tra numeri in virgola mobile. Le tre unità successive gestiscono la somma/sottrazione in virgola mobile, i confronti, la radice quadrata e la divisione.

Le operazioni di salto sono eseguite dall'unità di salto. Dopo un salto c'è un ritardo di 3 cicli, perciò le tre istruzioni successive a un salto (fino a 15 operazioni) vengono eseguite sempre, anche nel caso di salti incondizionati.

Veniamo infine alle unità multimediali per la gestione delle speciali operazioni multimediali. Il termine DSP sta per *Digital Signal Processor* ("processore dei segnali digitali"), le cui funzioni sono svolte in questo caso proprio dalle operazioni multimediali. Prima di entrare nei dettagli di queste operazioni, segnaliamo che usano tutte l'aritmetica satura (*saturated arithmetic*) invece dell'aritmetica in complemento a due usata dalle operazioni intere. Quando un'operazione produce un risultato che non può essere espresso a causa di un overflow, non viene sollevata un'eccezione, né viene restituito un risultato inservibile, bensì si ottiene il numero valido più vicino a quello corretto. Per esempio, se si usano i numeri di 8 bit senza segno, la somma di 130 + 130 dà 255. A causa del fatto che non tutte le operazioni possono occupare tutti i posti, accade frequentemente che un'istruzione non contenga tutte e cinque le operazioni che può ospitare.

Se un posto non è utilizzato, la sua dimensione viene ridotta per minimizzare lo spreco di spazio. Le operazioni presenti occupano 26, 34 o 42 bit. A seconda delle operazioni presenti, un'istruzione TriMedia può essere lunga dai 2 ai 28 byte, compresa l'intestazione di dimensione costante.

TriMedia non effettua controlli in fase d'esecuzione per verificare che le operazioni in un'istruzione siano compatibili. Se non lo sono, vengono eseguite comunque e producono un risultato errato. L'assenza del controllo è stata una scelta progettuale deliberata e intesa a risparmiare tempo e transistor. Il Core i7, durante l'esecuzione, svolge i controlli necessari per sincerarsi della compatibilità di tutte le operazioni superscalari, ma paga questa certezza in complessità, tempo e transistor. TriMedia si risparmia questa spesa affidando lo scheduling al compilatore, che dispone di tutto il tempo che vuole per ottimizzare con cura il posizionamento delle operazioni nelle istruzioni. Il rovescio della medaglia è che, se un'operazione necessita di un'unità funzionale non disponibile, l'intera istruzione va in stallo finché l'unità non torna disponibile.

Come nell'Itanium-2, le operazioni TriMedia sono preditative. Ogni operazione (a parte due eccezioni trascurabili) specifica un registro da esaminare prima dell'esecuzione dell'operazione; se il bit meno significativo del registro è asserito, l'istruzione viene

eseguita, viceversa viene saltata. A ognuna delle (al massimo) cinque operazioni viene attribuito un predicato individuale. Un esempio di operazione predicativa è

IF R2 IADD R4, R5 → R8

che effettua il test su R2 e, se il suo bit meno significativo vale 1, somma R4 a R5 e salva il risultato in R8. Un'operazione diventa incondizionata se usa R1 (che vale sempre 1) come registro predicativo. Se usa R0 (che vale sempre 0) diventa una no-op.

Le operazioni multimediali di TriMedia possono essere ripartite nei 15 gruppi della Figura 8.5. Molte operazioni coinvolgono tagli (*clipping*) mediante la specifica di un operando e di un intervallo; il taglio forza l'operando all'interno dell'intervallo, ovvero se il suo valore cade fuori dall'intervallo gli attribuisce il valore di uno degli estremi. Possono essere tagliati operandi di 6, 16 o 32 bit. Per fare un esempio, i tagli di 40 e di 340 in un intervallo da 0 a 255 restituiscono rispettivamente 40 e 255. Il gruppo dei tagli effettua appunto operazioni di taglio.

Gruppo	Descrizione
Tagli	Tagli di 4 byte o di 2 mezze parole
Valore assoluto DSP	Valore assoluto saturato con segno
Somma DSP	Somma saturata con segno
Sottrazione DSP	Sottrazione saturata' con segno
Moltiplicazione DSP	Moltiplicazione saturata con segno
Min, max	Minimo o massimo di quattro coppie di byte
Confronti	Confronti byte per byte di due registri
Scorrimento	Scorrimento di una coppia di operandi di 16 bit
Somma dei prodotti	Somma con segno di prodotti di 8 o 16 bit
Fusione, impacchettamento, scambio	Manipolazione di byte o mezze parole
Media di operandi di quattro byte	Media senza segno di operandi di quattro byte, byte per byte
Media di byte	Media senza segno di quattro elementi, byte per byte
Moltiplicazione di byte	Moltiplicazione senza segno a 8 bit
Stima del movimento	Somma senza segno di valori assoluti di differenze di 8 bit con segno
Miscellanea	Altre operazioni aritmetiche

**Figura 8.5** Principali gruppi di operazioni di TriMedia

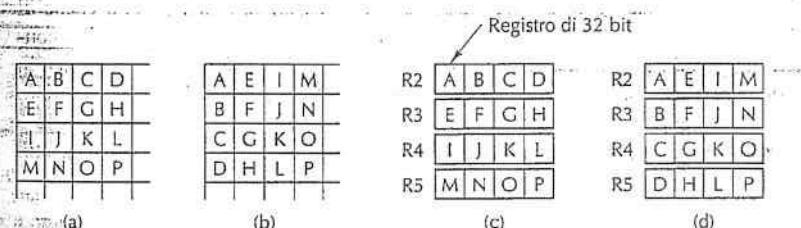
I quattro gruppi successivi della Figura 8.5 eseguono le operazioni indicate su operandi di varie dimensioni, tagliando il risultato in un certo intervallo. Il gruppo min, max esamina due registri e, per ogni byte, trova il valore minimo o massimo. Allo stesso modo, il gruppo dei confronti considera due registri come quattro coppie di byte ed effettua i confronti su ogni coppia.

Le operazioni multimediali non sono eseguite quasi mai su interi di 32 bit perché le immagini sono fatte di pixel RGB, definiti da 8 bit per ciascuno dei colori rosso, verde e blu. Durante l'elaborazione di un'immagine (per esempio la sua compressione) questa viene rappresentata in genere da tre componenti, uno per ogni colore (spazio RGB) o in una forma logicamente equivalente (spazio YUV, descritto nel seguito del capitolo). Comunque sia, gran parte dell'elaborazione si svolge su array rettangolari contenenti interi di 8 bit senza segno.

TriMedia è dotato di molte operazioni ideate specificatamente per l'elaborazione efficiente di interi senza segno di 8 bit. Consideriamo il semplice esempio del vertice superiore sinistro di un array di valori di 8 bit, collocati in memoria (*big-endian*) della Figura 8.6(a). I  $4 \times 4$  blocchi del vertice contengono 16 valori di 8 bit, etichettati da A a P. Se si vuole trasporre l'immagine, per produrre il risultato della Figura 8.6(b), come si può fare?

Una possibilità prevede l'utilizzo di 12 operazioni per caricare i byte nei registri, seguite da 12 operazioni per memorizzare i byte nella loro posizione corretta (i quattro byte lungo la diagonale non si spostano durante la trasposizione). Il problema è che questo approccio richiede 24 operazioni (lunghe e lente) che accedono alla memoria.

In alternativa si può cominciare con quattro operazioni di caricamento di una parola per volta nei quattro registri da R2 a R5, come mostrato nella Figura 8.6(c). Poi si ottengono le quattro parole di output della Figura 8.6(d) tramite operazioni di mascheratura e scorrimento. Infine, le parole sono pronte per essere salvate in memoria. Sebbene questo metodo riduca il numero di accessi in memoria da 24 a 8, le mascherature e gli scorrimenti sono costosi a causa del gran numero di operazioni richieste per estrarre e inserire ogni byte nella sua posizione corretta.



**Figura 8.6** (a) Un array di elementi di 8 bit. (b) L'array trasposto. (c) L'array originale memorizzato in quattro registri. (d) L'array trasposto nei quattro registri.

TriMedia consente di raggiungere una soluzione migliore delle precedenti. Si comincia con il fetch delle quattro parole nei registri, però, per costruire l'output, invece delle mascherature e degli scorrimenti, si usano operazioni speciali per l'estrazione e l'inserzione di byte nei registri. In definitiva, la trasposizione può essere ottenuta con 8 accessi alla memoria e 8 operazioni multimediali speciali. Il codice comincia con un'istruzione contenente due load in posizione 4 e 5, per il caricamento nei registri R2 e R3,

seguita da un'altra istruzione per caricare R4 e R5. Queste due istruzioni possono usare le altre posizioni per altri scopi. Al termine di tutti i caricamenti è possibile impacchettare le otto operazioni speciali per la costruzione dell'output in due istruzioni, seguite da altre due istruzioni per la memorizzazione. In totale servono solo sei istruzioni, in cui 14 delle 30 posizioni complessive sono ancora disponibili per ospitare altre operazioni. In effetti, il compito può essere svolto complessivamente con l'equivalente di circa tre istruzioni. Le altre operazioni multimediali sono altrettanto efficienti. Grazie alle sue potenti operazioni e alla capienza delle sue istruzioni, TriMedia è molto efficiente per la tipologia di calcolo richiesta dall'elaborazione multimediale.

### 8.1.2 Multithreading nel chip

Tutte le moderne CPU a pipeline presentano un problema: quando un riferimento in memoria fallisce nelle cache di primo e di secondo livello, bisogna aspettare molto tempo prima che la parola richiesta (e la corrispondente linea di cache) sia caricata nella cache, e così nel frattempo la pipeline è in stallo. Il **multithreading nel chip** costituisce un modo per trattare questa situazione perché, in una certa misura, si riescono a mascherare questi stalli consentendo alla CPU di gestire contemporaneamente più thread di controllo. Infatti, se il thread 1 è bloccato, la CPU ha ancora la possibilità di eseguire il thread 2 e di mantenere l'hardware impegnato.

Per quanto si tratti di un'idea molto semplice, ne esistono diverse varianti che ci accingiamo a esaminare. Il primo approccio si chiama **multithreading a grana fine** (*fine-grained multithreading*) ed è illustrato nella Figura 8.7, che 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. Naturalmente, alle volte è tuttavia possibile emettere altre istruzioni se si dispone di uno scoreboard più sofisticato, ma qui ignoriamo questa evenienza.

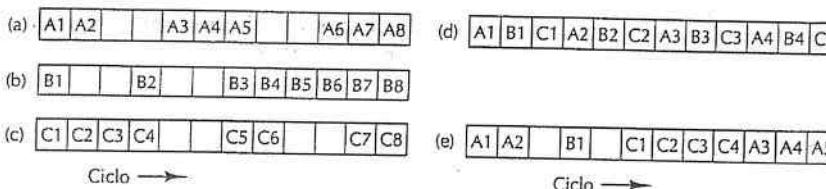


Figura 8.7 (a)-(c) Tre thread: le caselle vuote indicano che il thread è in stallo perché in attesa di dati dalla memoria. (d) Multithreading a grana fine. (e) Multithreading a grana grossa.

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. A volte un'istruzione aspetta il risultato di un'istruzione precedente non ancora completata, altre volte un'istruzione non può essere avviata perché segue un salto condizionato la cui destinazione non è ancora nota. Come regola generale, se la pipeline ha  $k$  stadi e ci sono almeno  $k$  thread in esecuzione a turno, nella pipeline non ci sarà mai più di un'istruzione per thread in esecuzione e perciò non si possono verificare conflitti. In queste condizioni la CPU può girare a pieno ritmo senza stalli.

Ovviamente potrebbero non esserci tanti thread quanti sono gli stadi della pipeline, perciò alcuni progettisti preferiscono un approccio differente, detto **multithreading a grana grossa** (*coarse-grained multithreading*), illustrato nella Figura 8.7(e). In questo caso 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 B1 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. Il multithreading a grana grossa è preferibile in tutte quelle situazioni in cui c'è un numero insufficiente di thread attivi, perché garantisce di trovarne almeno uno da mandare in esecuzione.

Anche se abbiamo descritto il multithreading a grana grossa come una commutazione sugli stalli, non è questa l'unica alternativa. Un'altra possibilità è di effettuare la commutazione immediatamente tutte le volte in cui un'istruzione potrebbe causare uno stallo: si pensi ai caricamenti, alle memorizzazioni e ai salti, ancor prima di scoprire se lo stallo si verificherà davvero. Secondo questa strategia la commutazione occorre in anticipo (non appena l'istruzione viene decodificata) e potrebbe così far evitare i cicli morti. È un po' come dire: "andiamo avanti finché potremmo avere un problema, al che commutiamo giusto in tempo". Così facendo, il multithreading a grana grossa, con le sue frequenti commutazioni, somiglia un po' di più a quello 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 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. È evidente che 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 grossa, ma questa volta con uno scheduler 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.

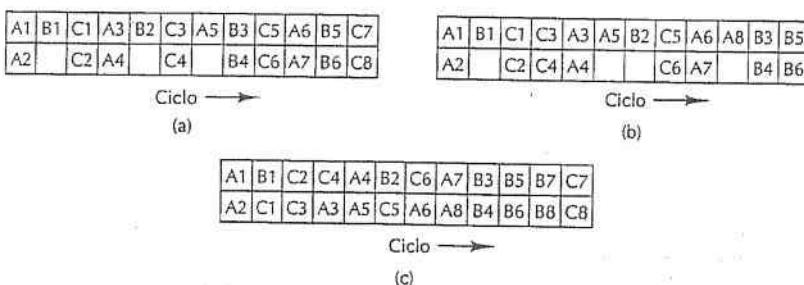


Figura 8.8 Multithreading in una CPU superscalare a doppia emissione. (a) Multithreading a grana fine. (b) Multithreading a grana grossa. (c) Multithreading simultaneo.

C'è una terza possibilità di multithreading con le CPU superscalari, il multithreading simultaneo, illustrato nella Figura 8.8(c). Lo si può considerare un raffinamento del multithreading a grana grossa, in cui ciascun thread emette due istruzioni per ciclo fin tanto 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.

Per maggiori informazioni sul multithreading si faccia riferimento a (Gebhart et al., 2011) e (Wing-kei et al., 2011).

### Hyperthreading nel Core i7

Lo studio astratto del multithreading ci consente ora di affrontare a un esempio pratico: quello del Core i7. Nei primi anni 2000 processori come il Pentium 4 non erano in grado di offrire gli incrementi di prestazioni di cui Intel aveva bisogno per mantenere il livello di vendite. Quando il Pentium 4 era già in produzione gli architetti di Intel cominciarono a riflettere sul modo di velocizzare il processore senza modificare l'interfaccia con il programmatore, la qual cosa sarebbe stata inaccettabile. Furono individuate subito cinque ipotesi.

1. Incrementare la velocità di clock.
2. Posizionare due CPU nel chip.
3. Aggiungere unità funzionali.
4. Allungare la pipeline.
5. Usare il multithreading.

Un modo ovvio per migliorare le prestazioni consiste nell'aumento della velocità del clock senza cambiare nient'altro. È una soluzione relativamente semplice e ben fondata, perciò ogni nuovo chip in genere supera di poco la velocità del suo predecessore. Sfortunatamente, l'incremento delle velocità di clock porta con sé due svantaggi che ne limitano la tollerabilità. Un clock più veloce assorbe più energia, un problema molto grande per i computer portatili e in genere per tutti quelli alimentati a batteria. In secondo luogo, più energia in ingresso vuol dire più calore da dissipare.

La sistemazione di due CPU in un chip è abbastanza semplice, ma equivale circa a raddoppiare l'area del chip se ciascun processore è dotato di cache propria; la riduzione del numero di chip per unità di superficie di un fattore due non fa che raddoppiare i costi unitari di produzione. Se i due chip condividono una cache comune grande quanto quella originale, la superficie del chip non viene raddoppiata, ma la quantità di cache per CPU è dimezzata e ciò riduce le prestazioni. Inoltre, se le applicazioni dei server di fascia alta sono spesso in grado di trarre il massimo giovamento dalla presenza di più CPU, le applicazioni dei desktop non hanno abbastanza parallelismo da giustificare

Anche l'aggiunta di unità funzionali è relativamente semplice, ma bisogna raggiungere il giusto equilibrio. È inutile disporre di 10 ALU se il chip non è in grado di passare loro le istruzioni abbastanza velocemente da tenerle occupate.

L'allungamento della pipeline tramite aggiunta di nuovi stadi, ciascuno destinato allo svolgimento di un compito più piccolo in un tempo più breve, può incrementare le prestazioni, ma aumenta anche gli effetti negativi delle predizioni errate sui salti, dei fallimenti di cache, degli interrupt e in genere di tutti quei fattori che interrompono il normale flusso esecutivo. Inoltre, per poter sfruttare pienamente la maggiore lunghezza della pipeline, è necessario un clock più veloce, il che implica un più elevato consumo di energia e un incremento di produzione di calore.

Resta infine l'introduzione del multithreading, il cui valore aggiunto sta nel permettere che un secondo thread utilizzi l'hardware che sarebbe rimasto altrimenti inattivo. A seguito di alcuni esperimenti, si è stabilito che il supporto del multithreading, a fronte

di un incremento del 5% della superficie del chip, produce un miglioramento delle prestazioni del 25% per molte applicazioni ed è perciò una buona scelta. La prima CPU Intel a usare il multithreading è stata Xeon nel 2002; il multithreading è stato aggiunto in seguito al Pentium 4, dalla versione a 3,06 GHz fino alle versioni più veloci di processori Pentium, tra cui il Core i7. Intel ha dato il nome di *hyperthreading* all'implementazione del multithreading nei suoi processori.

Alla sua base c'è l'idea di consentire a due thread (o anche a due processi, dal momento che la CPU non può distinguere tra thread e processi) di essere eseguiti contemporaneamente. Dal punto di vista del sistema operativo, un Core i7 con hyperthreading somiglia a un biprocessore in cui le CPU condividono cache e memoria principale. Il sistema operativo seleziona i thread per l'esecuzione in modo indipendente; se ci sono due applicazioni in esecuzione nello stesso istante, il sistema operativo può farle eseguire in parallelo. Per esempio, se un demone di posta riceve o spedisce email in background<sup>1</sup> mentre un utente interagisce con un altro programma, i due programmi possono essere eseguiti contemporaneamente proprio come se ci fossero due CPU disponibili.

Il software applicativo progettato per l'esecuzione con thread multipli può avvalersi di entrambe le CPU virtuali. Per esempio, i programmi di montaggio video in genere permettono all'utente di specificare alcuni filtri da applicare ai fotogrammi di un certo intervallo. Questi filtri possono agire su luminosità, contrasto, bilanciamento del bianco e altre proprietà di ciascun fotogramma. Il programma può decidere di affidare i fotogrammi pari a una CPU e quelli dispari all'altra, e le due CPU possono lavorare in parallelo.

Poiché i thread condividono tutte le risorse hardware, si rende necessaria una strategia per la gestione della condivisione. Intel ha identificato quattro strategie utili per la condivisione di risorse associate all'hyperthreading: duplicazione e ripartizione di risorse, condivisione a soglia oppure totale. Vediamole una per volta.

Cominciamo dalle risorse che vengono duplicate per i thread. Per esempio, poiché ogni thread ha il proprio controllo del flusso, si è resa necessaria l'aggiunta di un secondo program counter. È stata duplicata anche la tabella che mappa i registri architetturali (EAX, EBX e così via) nei registri fisici, nonché il controllore di interrupt, visto che i thread possono essere interrotti in modo indipendente.

Poi c'è la condivisione ripartita delle risorse (*partitioned resource sharing*), secondo cui le risorse hardware sono divise rigidamente tra i thread. Per esempio, se la CPU ha una coda tra due stadi funzionali della pipeline, metà delle posizioni della coda potrebbero essere dedicate al thread 1, l'altra metà al thread 2. La ripartizione delle risorse è facile da ottenere, non richiede informazioni addizionali e fa sì che i thread non vadano in conflitto. Se tutte le risorse sono ripartite è come avere due CPU separate. La ripartizione ha anche un aspetto negativo: può succedere facilmente che un thread non usi alcune delle sue risorse che farebbero invece comodo a un altro thread, che però non

ha diritto di usarle. Di conseguenza, alcune risorse che sarebbero potute servire a una qualche attività restano invece inoperose.

La condivisione totale delle risorse (*full resource sharing*) è il contrario della condivisione ripartita. Secondo questo schema, ogni thread può acquisire tutte le risorse di cui ha bisogno: il primo che arriva si serve da sé. Immaginiamo però due thread, uno veloce che fa solo somme e sottrazioni, uno più lento che svolge moltiplicazioni e divisioni. Se le istruzioni sono caricate dalla memoria con un ritmo maggiore rispetto alla velocità di esecuzione delle moltiplicazioni e delle divisioni, il numero d'istruzioni arretrate del thread lento (che si trova nella coda d'attesa per l'inserimento nella pipeline) crescerà di continuo. A un certo punto le istruzioni arretrate riempiranno completamente la coda d'attesa, costringendo il thread veloce ad arrestarsi per mancanza di spazio nella coda delle istruzioni. La condivisione totale delle risorse risolve il problema di quelle inattive richieste da un thread, ma crea un problema nuovo in cui un thread può impossessarsi di così tante risorse da rallentare l'altro o addirittura da farlo fermare.

Uno schema intermedio è la condivisione a soglia (*threshold sharing*), secondo cui un thread può acquisire le risorse dinamicamente (non c'è ripartizione fissa), ma solo fino a un certo valore massimo. Questo approccio permette una certa flessibilità ed esclude il pericolo che un thread vada in *starvation*<sup>2</sup> a causa della sua incapacità di procurarsi le risorse. Per esempio, se nessun thread può acquisire più dei 3/4 della coda delle istruzioni, il thread più veloce potrà sempre proseguire la propria esecuzione, qualsiasi cosa faccia il thread più lento.

L'hyperthreading del Core i7 usa strategie di condivisione differenti a seconda delle risorse, nel tentativo di far fronte ai problemi cui si è accennato. La duplicazione è utilizzata per le risorse di cui i thread hanno sempre bisogno, come il program counter, la mappa dei registri e il controllore di interrupt. La duplicazione di queste risorse richiede un incremento della superficie del chip del 5%, un prezzo accettabile per il multithreading. Le risorse che sono così abbondanti per cui non si corre il rischio di acquisizione da parte di un solo thread, come le linee di cache, sono totalmente condivise in maniera dinamica. Invece le risorse che controllano il funzionamento della pipeline, come le varie code presenti al suo interno, sono ripartite esattamente tra i due thread. La Figura 8.9 illustra la pipeline principale della microarchitettura Sandy Bridge usata nel Core i7, laddove le caselle grigie e quelle bianche indicano l'allocazione delle risorse tra il thread grigio e quello bianco.

Nella figura notiamo che tutte le code sono ripartite a metà tra i thread, perciò nessuno dei due thread può soffocare l'altro. Anche i registri di allocazione e di rinomina sono ripartiti. Lo scheduler è condiviso dinamicamente con una soglia, per evitare che un thread reclami per sé l'intera coda. Gli altri stadi della pipeline sono totalmente condivisi.

1 Per elaborazione in *background* ("sullo sfondo") si intende l'elaborazione a bassa priorità di programmi che non hanno bisogno di interazioni con l'utente, e che quindi sono spesso a lui invisibili (N.d.T.).

2 *To starve vuol dire "morire di fame"; in questo contesto indica l'inattività volontaria di un thread dovuta all'esaurimento delle risorse (N.d.T.).*

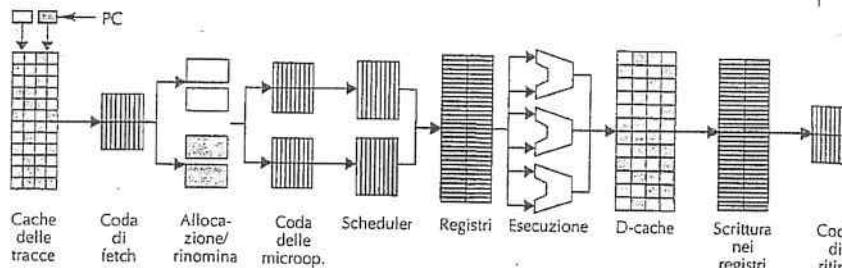


Figura 8.9 Condivisione delle risorse tra i thread della microarchitettura del Core i7.

C'è da dire che il multithreading non è la panacea, ma che esiste anche un suo aspetto negativo. Se la ripartizione delle risorse è facile da implementare, la loro condivisione dinamica, specie se basata su soglia, richiede lavoro di amministrazione per monitorarne l'uso in fase d'esecuzione. Oltre a ciò, esistono delle situazioni in cui i programmi funzionano molto peggio con il multithreading che non senza. Per esempio, si immaginino due thread che, per funzionare bene, hanno bisogno entrambi di 3/4 della cache. Se eseguiti separatamente, girano senza problemi con pochi (costosi) fallimenti di cache. Se eseguiti insieme, ciascuno di loro provoca molti fallimenti di cache e il risultato complessivo potrebbe essere di gran lunga peggiore.

Si possono trovare altre informazioni sul multithreading e sulle sue implementazioni nei processori Intel in (Gerber e Binstock, 2004) e (Gepner et al., 2011).

### 8.1.3 Multiprocessori in un solo chip

Anche se il multithreading consente un incremento significativo delle prestazioni a un costo contenuto, per certi tipi di applicazioni si richiedono prestazioni molto superiori di quelle che può assicurare il multithreading. A tal fine è diretto lo sviluppo dei multiprocessori (chip con due o più CPU), che suscita particolare interesse nel mercato dei server di fascia alta e dell'elettronica di consumo. Diamo una rapida scorsa a entrambi i settori.

#### Multiprocessori omogenei in un solo chip

Grazie ai progressi della tecnologia VLSI è oggi possibile inserire in un chip due o più CPU potenti. Queste CPU si definiscono multiprocessori perché condividono le stesse cache, di primo e secondo livello, e la memoria principale (come illustrato nel Capitolo 2). Comunemente trovano applicazione nelle server farm che raggruppano molti server web. La capacità di far coabitare due CPU in un unico sistema, condividendo non solo la memoria, ma anche il disco e le interfacce di rete, permette spesso di raddoppiare le prestazioni del server senza raddoppiarne i costi (anche se il costo della CPU raddoppia, il suo prezzo è solo una frazione di quello dell'intero sistema).

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. La Figura 8.10(b) mostra il secondo tipo di progetto: nel chip ci sono due core separati, ciascuno contenente un'intera CPU. 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.

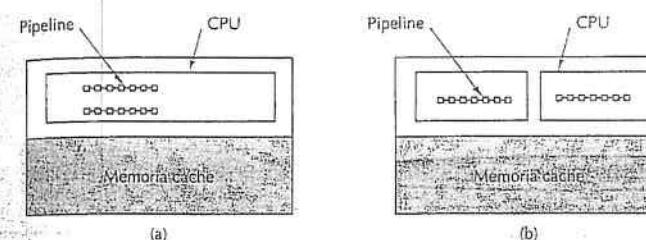


Figura 8.10 Multiprocessori a singolo chip. (a) Un chip con pipeline duplice. (b) Un chip con due core.

Il primo 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. Questo approccio però richiede la riprogettazione del chip e non si applica altrettanto bene al crescere del numero di CPU. Per contro, l'inserzione di due o più core CPU sullo stesso chip è relativamente facile.

L'argomento multiprocessori sarà trattato ancora in questo capitolo. Anche se l'analisi sarà focalizzata sui multiprocessori costruiti a partire da chip con CPU singola, si può estendere in larga misura anche ai chip con più CPU.

#### Il multiprocessore a singolo chip Core i7

La CPU Core i7 è un multiprocessore a singolo chip costruito con quattro (o più) core su un singolo stampo di silicio. L'organizzazione ad alto livello del Core i7 è mostrata nella Figura 8.11.

Ogni processore del Core i7 ha tre cache private. Due di primo livello per dati e istruzioni e una di secondo, unificata. I processori sono connessi alle cache private mediante connessioni dedicate punto a punto. Il livello successivo della gerarchia di memoria è la cache dati L3 unificata e condivisa.

Le cache L2 sono connesse alla cache L3 tramite una rete ad anello. Una richiesta di comunicazione che entra nella rete viene girata al nodo successivo della rete, dove si verifica se la richiesta ha raggiunto il suo nodo destinazione. Il processo si ripete fino a quando la richiesta arriva al nodo destinazione oppure ritorna al nodo origine (caso in cui il nodo destinazione non esiste). Il vantaggio della rete ad anello è che si tratta di un modo economico per avere alte larghezze di banda, al costo però di una maggior latenza per il passaggio della richiesta di nodo in nodo. La rete ad anello del Core i7 svolge due compiti principali. Prima di tutto offre un modo per spostare richieste di memoria e di

I/O tra le cache e il processore. Inoltre, implementa i controlli necessari ad assicurare che ogni processore abbia sempre una visione coerente della memoria. Impareremo di più su questi controlli di coerenza più avanti nel capitolo.

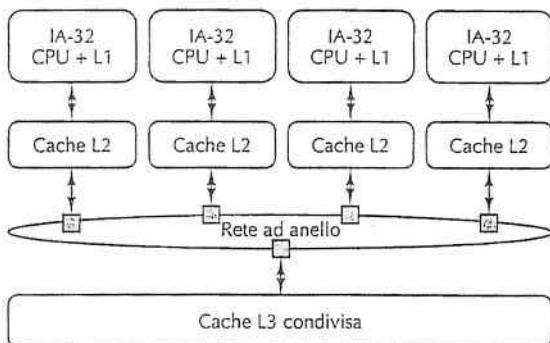


Figura 8.11 Architettura multiprocessore a singolo chip del Core i7.

### Multiprocessori eterogenei in un solo chip

Una gamma completamente diversa di multiprocessori in un solo chip è costituita dai sistemi integrati e in special modo dall'elettronica di consumo per contenuti audiovisivi, come apparecchiature TV, lettori DVD, videocamere, console per i videogiochi, telefoni cellulari e così via. Si tratta di sistemi che devono garantire prestazioni impegnative nel rispetto di forti vincoli. Anche se questi dispositivi sembrano a prima vista diversi, in realtà molti di loro non sono altro che piccoli calcolatori dotati di una o più CPU, di memorie, di controlleri di I/O e di dispositivi di I/O specifici. Un cellulare, per esempio, contiene una CPU, una memoria, una piccola tastiera, un microfono, un altoparlante e una connessione senza fili alla rete, il tutto in un piccolo involucro.

Si consideri un lettore DVD portatile. Il computer al suo interno deve gestire le funzioni seguenti:

1. Controllo di un servomeccanismo economico e poco affidabile per il puntamento della testina.
2. Conversione da analogico a digitale.
3. Correzione degli errori.
4. Decodifica crittografica e gestione dei diritti digitali.
5. Decompressione video MPEG-2.
6. Decompressione audio.
7. Codifica dell'output per i sistemi televisivi NTSC, PAL o SECAM.

Tutte queste azioni vanno svolte rigorosamente in tempo reale, con vincoli circa la qualità del servizio, l'assorbimento di energia, la dissipazione del calore, le dimensioni, il peso e il prezzo del dispositivo.

I CD, i DVD e i Blu-Ray contengono una lunga spirale d'informazioni (analogamente alla Figura 2.25 relativa ai CD). In questa sezione parleremo dei DVD, perché sono ancora più comuni dei Blu-Ray. Questi sono comunque molto simili ai DVD, eccetto per l'uso di MPEG-4 al posto di MPEG-2 per la codifica. In tutti i dispositivi ottici la testina di lettura deve puntare alla spirale con cura mentre il disco ruota. Si riesce a contenere il prezzo del dispositivo grazie alla progettazione di un meccanismo relativamente semplice e al controllo software della posizione della testina. Il segnale analogico proveniente dalla testina deve essere digitalizzato prima della sua elaborazione. Dopo la conversione è necessaria una pesante correzione degli errori perché i DVD ne contengono molti che devono essere corretti via software. La compressione video usa lo standard MPEG-2 che richiede un calcolo abbastanza complesso (simile alla trasformata di Fourier) per la decompressione. La compressione audio si avvale di un modello psicologico dell'udito e anch'esso richiede calcoli sofisticati in fase di decompressione. Infine, audio e video devono essere trasformati in un formato adatto al televisore e che varia da paese a paese: NTSC, PAL o SECAM. Non sorprende quindi come sia semplicemente impossibile svolgere tutte queste attività in tempo reale, via software e su CPU per uso generale. Ciò di cui c'è bisogno è un multiprocessore eterogeneo contenente più core, ciascuno specializzato per un certo compito. La Figura 8.12 presenta un esempio di lettore DVD.

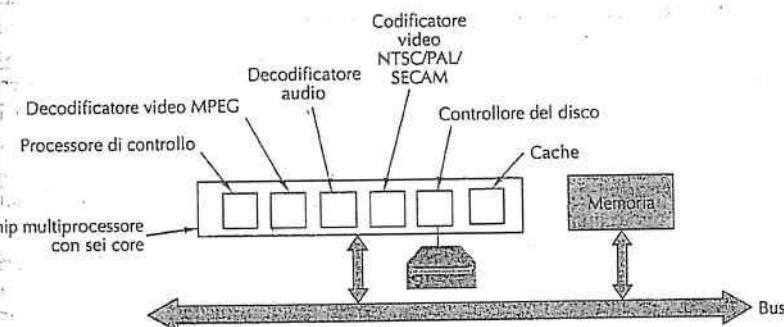


Figura 8.12 Struttura logica dei lettori DVD contenente un multiprocessore eterogeneo con vari core dedicati.

Le funzioni svolte dai core della Figura 8.12 sono tutte diverse; ciascun core è stato progettato con cura per svolgere il proprio compito al meglio e costare il meno possibile. Per esempio, il video dei DVD è compresso con MPEG-2 (da *Motion Picture Experts Group*, il gruppo di esperti che lo ha inventato). Si procede scomponendo ogni fotogramma (*frame*) in blocchi di pixel e applicando una complicata trasformazione

matematica a ogni blocco. Un fotogramma può essere fatto interamente di blocchi trasformati, oppure si può specificare che un determinato blocco sia uguale a un blocco del fotogramma precedente, localizzato a un offset ( $\Delta x, \Delta y$ ) rispetto alla sua posizione attuale, fatta eccezione per una manciata di pixel cambiati. Un calcolo di questo tipo sarebbe molto lento via software, ma è possibile costruire dei circuiti di decodifica MPEG-2 che possono svolgerlo in modo abbastanza rapido. Analogamente, anche la decodifica audio e la ricodifica composita audio-video del segnale (per rispettare gli standard televisivi mondiali) possono essere svolte con maggior successo da processori dedicati. Queste considerazioni giustificano largamente la progettazione di chip multiprocessori contenenti core specializzati per applicazioni audiovisive. Il processore di controllo è una CPU programmabile di uso generale, perciò il chip multiprocessore può essere usato anche per altre applicazioni simili.

Il telefono cellulare è un altro dispositivo che necessita di un multiprocessore eterogeneo per il suo funzionamento. I modelli attuali possono includere macchine fotografiche, videocamere, giochi, navigatori web, lettori di posta, ricevitori radio di segnali satellitari digitali; essi si basano sulla tecnologia cellulare (CDMA o GSM, a seconda del paese) o su Internet senza fili (IEEE 802.11, detta anche tecnologia WiFi). I modelli futuri saranno dotati di tutti questi dispositivi. Gli apparecchi acquistano sempre più funzionalità, gli orologi diventano mappe basate su GPS, gli occhiali diventano radio e la richiesta di multiprocessori eterogenei continuerà a crescere.

Verrà presto il giorno in cui i chip più grandi avranno decine di miliardi di transistor. Si tratterebbe di una dimensione troppo grande perché si possa progettare a partire dai circuiti elementari. Lo sforzo umano richiesto sarebbe tale da rendere un chip obsoleto ancor prima di essere ultimato. L'unico modo di procedere è usare i core (di fatto delle librerie), ciascuno contenente un sottoassemblato abbastanza grande, e collegarli tramite il montaggio su un chip. Ai progettisti resta solo da decidere quale core CPU usare per il processore di controllo e quali processori specializzati associargli per assistere. Spostare il carico di lavoro sul software che gira sul processore di controllo rallenta il sistema, ma consente chip più piccoli (e più economici). La presenza di processori diversi per l'elaborazione audio e video aumenta la superficie del chip e fa lievitare i costi, ma produce prestazioni migliori a parità di frequenza di clock, il che implica un minor consumo energetico e una minore dissipazione di calore. Di conseguenza, i progettisti si trovano sempre più spesso di fronte a questo tipo di compromessi macroscopici, piuttosto che alle prese con i dettagli realizzativi dei componenti hardware.

Le applicazioni audiovisive lavorano su grosse quantità di dati che devono essere elaborate velocemente, perciò la memoria (sotto varie forme) occupa dal 50% al 75% della superficie dei chip e questa percentuale è destinata a salire. Si pongono molti interrogativi di progetto: quanti livelli di cache prevedere? Usare una cache specializzata o una unificata? Quanto dovrebbe essere grande ciascuna cache? Quanto veloce? Vale la pena inserire parte della memoria nel chip? Meglio memoria SRAM o SDRAM? Le risposte a queste domande hanno forti implicazioni sulle prestazioni, sul consumo energetico e sulla dissipazione del calore del chip.

Al di là del progetto dei processori e dei sistemi di memoria, un altro tassello importante per le sue conseguenze è il sistema di comunicazione: 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.

La Figura 8.13 presenta CoreConnect di IBM, un esempio di interconnessione nel chip. È un'architettura per la connessione di core su multiprocessori eterogenei su un chip, usata soprattutto per la progettazione di chip contenenti interi sistemi. In un certo senso, CoreConnect sta ai multiprocessori nel singolo chip come il bus PCI sta al Pentium, ovvero è la colla che tiene uniti i vari pezzi. (Nei moderni sistemi Core i7, la colla è PCIe che è una rete punto a punto priva di un bus condiviso come il PCI). Però, a differenza del bus PCI, CoreConnect è stato progettato senza vincoli di retrocompatibilità dovuti a componenti o protocolli precedenti, e senza i vincoli propri dei bus che risiedono sulle schede, come il limite sul numero di piedini presenti sul connettore.

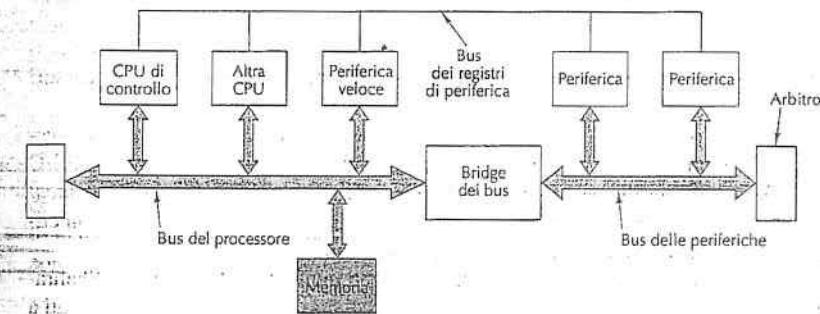


Figura 8.13 Esempio di architettura CoreConnect di IBM.

CoreConnect è composto da tre bus. Quello del processore è un bus veloce, sincrono e a pipeline con linee dati di 32, 64 o 128 bit, temporizzate a 66, 133, o 183 MHz. La velocità massima di trasferimento è 23,4 Gbps (contro 4,2 Gbps per il bus PCI). Il funzionamento a pipeline permette ai core di richiedere il bus mentre è già impegnato in un trasferimento e consente a core diversi di usare contemporaneamente linee dati distinte, come per il bus PCI. Il bus del processore è ottimizzato per il trasferimento di blocchi corti ed è concepito per connettere core veloci, come la CPU, il decodificatore MPEG-2, le reti ad alta velocità o componenti similari.

Se il bus del processore fosse esteso a tutto il chip le sue prestazioni degraderebbero, perciò esiste un secondo bus per i dispositivi di I/O più lenti, come gli UART, i timer, i controlleri USB e le porte seriali. Questo bus delle periferiche è stato progettato per avere un'interfaccia semplice con le periferiche a 8, 16 o 32 bit e per contenere non più

di qualche centinaio di porte logiche. Anch'esso è sincrono, ma la velocità massima di trasferimento è di 300 Mbps. I due bus sono collegati attraverso un *bridge* simile a quello usato qualche anno fa per collegare i bus PCI e ISA nei PC, prima che gli ISA diventassero obsoleti.

Il terzo bus dei registri di periferica è molto lento, asincrono e usa un *handshaking* per permettere a ogni processore di accedere ai registri di tutte le periferiche per il controllo dei rispettivi dispositivi. È concepito per trasferimenti poco frequenti e di pochi byte alla volta.

IBM CoreConnect ha definito uno standard per i bus nel chip, per la loro interfaccia e per l'intelaiatura generale del chip con l'ambizione di creare una versione in miniatura del mondo PCI, in cui ogni produttore possa costruire processori e controllori che si interconnettano con facilità. Esiste però una differenza: nel mondo PCI i produttori costruiscono e vendono schede acquistabili dai distributori e dagli utenti finali. Nel mondo CoreConnect i core sono progettati da terze parti che però non realizzano i prodotti, ma li brevettano come proprietà intellettuale e vendono le licenze alle aziende di elettronica di consumo, che progettano i chip multiprocessori eterogenei basandosi sui core propri e su quelli sottoposti a licenza di terzi. Poiché la produzione di chip così grandi e complessi richiede investimenti massicci in impianti produttivi, nella maggior parte dei casi le aziende di elettronica di consumo si limitano alla progettazione e affidano la realizzazione del chip a produttori di semiconduttori. Esistono core per numerose CPU (ARM, MIPS, PowerPC, e così via), per decodificatori MPEG, per processori di segnali digitali e per tutti i controllori di I/O standard.

Oltre a CoreConnect IBM, anche AMBA (*Advanced Microcontroller Bus Architecture*) è usato largamente per connettere CPU ARM ad altre CPU o a dispositivi di I/O (Flynn, 1997). Altri bus di questo tipo, però meno diffusi, sono VCI (*Virtual Component Interconnect*) e OCP-IP (*Open Core Protocol-International Partnership*), in competizione tra loro per una quota di mercato (Bhakthavatchalu et al., 2010).

E questo è soltanto l'inizio: c'è chi mette intere reti in un chip (Ahmadinia e Shahabi, 2011). A causa delle difficoltà legate ai problemi di dissipazione del calore (dovuti a loro volta a velocità di clock sempre maggiori) i multiprocessori in un singolo chip sono oggi un tema attuale; rimandiamo a (Gupta et al., 2010; Herrero et al., 2010; Mishra et al., 2011) per ulteriori informazioni.

## 8.2 Coprocessori

Dopo aver esaminato i modi del parallelismo nel chip, facciamo un passo in avanti e ci interessiamo al modo di velocizzare un calcolatore mediante l'aggiunta di un secondo processore specializzato. Esistono molte varianti di coprocessori, di dimensioni molto diverse. Sui mainframe IBM 360, e sui loro successori, esistono canali di I/O indipendenti per l'input e l'output. Allo stesso modo, il CDC 6600 disponeva di 10 processori indipendenti per le operazioni di I/O. Anche la grafica e l'aritmetica in virgola mobile si prestano all'uso di coprocessori, e lo stesso DMA può essere visto come un coprocessore. In alcuni casi la CPU assegna al coprocessore un'istruzione o un insieme d'istru-

zioni e gli ordina di eseguirle; in altri casi il coprocessore è più indipendente e lavora per conto proprio.

Dal punto di vista fisico, i coprocessori possono costituire apparati separati (i canali di I/O del 360), trovarsi su schede a innesto (i processori di rete) o occupare parte del chip principale (coprocessore in virgola mobile). In tutti i casi, ciò che li caratterizza è il fatto di assistere nell'esecuzione un altro processore, che resta il processore principale. Esaminiamo ora tre settori in cui è possibile velocizzare le prestazioni: elaborazione di rete, multimedia e crittografia.

### 8.2.1 Processori di rete

Oggi molti calcolatori sono connessi a una rete o a Internet. A seguito del progresso tecnologico dell'hardware di rete, le reti sono oggi così veloci che è sempre più difficile elaborare via software i dati in ingresso e in uscita. A questo riguardo sono stati sviluppati speciali processori di rete per gestire il traffico; molti calcolatori di fascia alta sono dotati di processori di questo tipo. In questo paragrafo cominciamo con l'introduzione alle reti, per poi illustrare il funzionamento dei processori di rete.

#### Introduzione alle reti

Le reti di calcolatori sono di due tipi: LAN (*Local-Area Network*, "rete locale"), che connettono più computer all'interno di un edificio o di un campus universitario, e WAN (*WideArea Network*, "rete geografica"), per la connessione di calcolatori che si trovano a grande distanza. Ethernet è la LAN più diffusa: inizialmente era costituita da cavi spessi contenenti un filo per ogni computer, collegato tramite una spina che veniva chiamata eufemisticamente *spina a vampiro* (*vampire tap*). L'odierna Ethernet prevede che i calcolatori siano collegati a un commutatore (*switch*) centrale, come schematizzato nella Figura 8.14. In origine, Ethernet si trascinava a 3 Mbps, ma la prima versione commerciale raggiungeva già i 10 Mbps.

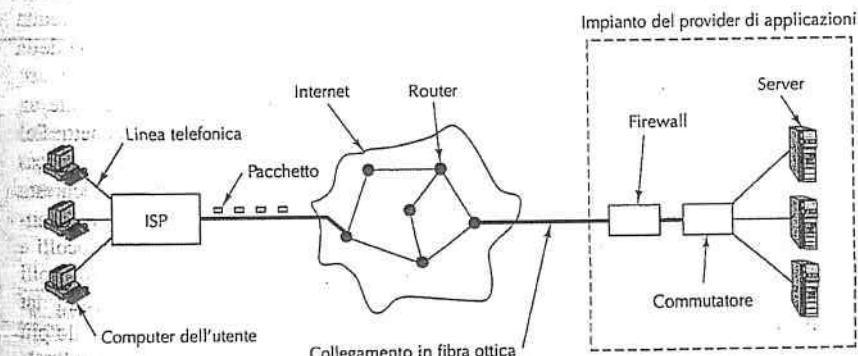


Figura 8.14 Connessione degli utenti ai server di Internet.

È stata poi sostituita da Ethernet veloce (*fast Ethernet*) a 100 Mbps e successivamente da *gigabit Ethernet* a 1 Gbps. Esiste già sul mercato una versione di Ethernet a 10 Gbps e la prossima sarà a 40 Gbps.

Le WAN sono organizzate in modo diverso. Sono costituite da computer per l'strumentazione, detti *router*, collegati tramite cavi o fibre ottiche, come illustrato al centro della Figura 8.14. I dati sono suddivisi in frammenti chiamati *pacchetti* (*packet*) che vanno in genere dai 64 ai 1500 byte e che vengono trasferiti dalla macchina mittente alla destinataria passando attraverso uno o più router. A ogni passaggio (*hop*, "salto"), il pacchetto è memorizzato in un router e quindi inoltrato al successivo lungo il cammino non appena la linea di trasmissione lo consente. Questa tecnica si chiama *commutazione di pacchetto store-and-forward*.

Anche se molte persone pensano a Internet nei termini di un'unica WAN, dal punto di vista tecnico è invece un insieme di molte WAN interconnesse. In ogni caso, questa distinzione non è rilevante ai nostri scopi. La Figura 8.14 fornisce una visione panoramica di Internet dal punto di vista dell'utente domestico. Il suo calcolatore è collegato a un server web tramite una linea telefonica, per mezzo di modem a 56 Kbps o ADSL, come illustrato nel Capitolo 2 (in alternativa è possibile una connessione che usa la linea della TV via cavo, nel qual caso la parte sinistra della figura cambia leggermente e il *provider* diventa la compagnia che fornisce la TV via cavo). Il calcolatore dell'utente spezzetta i dati da inviare al server in pacchetti e li spedisce all'*ISP* (*Internet Service Provider*, "fornitore di servizi Internet") dell'utente, una società che garantisce ai propri utenti l'accesso a Internet. L'*ISP* ha in genere una connessione ad alta velocità (di solito una fibra ottica) verso una delle reti regionali o una *backbone* (dorsale) di Internet. I pacchetti dell'utente sono inoltrati nella rete un salto alla volta finché non raggiungono il server web.

Molte società che forniscono servizi web dispongono di un *firewall* ("muro tagliafuoco"), un computer specializzato che filtra tutto il traffico in ingresso e cerca di rimuovere i pacchetti indesiderati (per esempio quelli che provengono da hacker che cercano di introdursi nel server). Il *firewall* è collegato alla LAN, in genere a un commutatore Ethernet, che instrada i pacchetti al server desiderato. Naturalmente, le cose nella realtà sono molto più complesse di come le abbiamo presentate, ma l'idea sostanziale della Figura 8.14 resta valida.

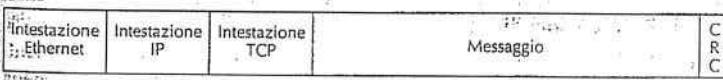
Il software di rete è organizzato per protocolli, ciascuno dei quali comprende un insieme di formati, di sequenze di scambio e di regole sul significato dei pacchetti. Per esempio, quando un utente vuole scaricare una pagina web da un server, il suo applicativo di navigazione (*browser*) spedisce al server un pacchetto contenente la richiesta *GET PAGE* e che usa *HTTP* (*HyperText Transfer Protocol*, "protocollo di trasferimento di ipertesti"); il server recepisce la richiesta e la elabora. Esistono molti protocolli e spesso vengono usati in combinazione. Nella maggior parte delle situazioni i protocolli sono strutturati come una serie di livelli (*layer*), in cui i pacchetti vengono passati dai livelli superiori a quelli inferiori perché li elaborino e, una volta raggiunto il livello più basso, perché siano trasmessi. Dal lato destinatario, i pacchetti si fanno strada tra i livelli in ordine inverso, dal basso verso l'alto.

Dal momento che l'elaborazione dei protocolli è l'attività principale dei processori di rete, dobbiamo dilungarci ancora un po' sull'argomento prima di passare ai processori veri e propri. Torniamo per un momento alla richiesta *GET PAGE*. Come avviene la sua spedizione al server web? In pratica il software di navigazione per prima cosa stabilisce una connessione con il server web basata sul protocollo TCP (*Transmission Control Protocol*, "protocollo di controllo della trasmissione"). Il software che implementa questo protocollo verifica che i pacchetti siano stati ricevuti tutti correttamente e nell'ordine giusto. Se un pacchetto viene perso, TCP garantisce che verrà ritrasmesso finché non raggiunga il destinatario.

Il navigatore web compila la richiesta *GET PAGE* come un messaggio HTTP e lo passa al software TCP per la trasmissione in rete. Il software TCP aggiunge all'inizio del messaggio un'intestazione contenente una serie di numeri e altre informazioni, che ovviamente prende il nome di **intestazione TCP (TCP header)**.

Quando ha finito la sua attività, il software TCP passa l'intestazione e la parte di messaggio vero e proprio (il *payload*, che in questo caso contiene la richiesta *GET PAGE*) a un altro componente software che implementa il protocollo IP (*Internet Protocol*). Questo software antepone al messaggio un'intestazione IP contenente l'indirizzo del mittente (la macchina da cui proviene il pacchetto) e quello del destinatario (la macchina cui il pacchetto è diretto), il massimo numero di passaggi cui può sopravvivere (per evitare che i pacchetti sopravvivano indefinitamente), una checksum (per rilevare errori di trasmissione o di memoria), e altri campi.

Il pacchetto risultante (dall'intestazione IP, intestazione TCP e richiesta *GET PAGE*) viene passato in basso a livello di trasmissione dei dati (*data link layer*), in cui gli viene allegata un'ulteriore intestazione che serve alla trasmissione vera e propria. Il livello di trasmissione dei dati allega in fondo al messaggio anche una checksum che si chiama CRC (*Cyclic Redundancy Check*, "controllo a ridondanza ciclica") e che serve per rilevare errori di trasmissione. Potrebbe sembrare ridondante la presenza di due checksum a livello di trasmissione dei dati e a livello IP, ma serve a migliorare l'affidabilità. A ogni salto lungo la rete viene controllato il valore di CRC, quindi viene rimosso e rigenerato (insieme all'intestazione) secondo un formato adatto al collegamento in uscita. La Figura 8.15 mostra le sembianze del pacchetto quando si trova in una Ethernet; il pacchetto in una linea telefonica (nel caso dell'ADSL) è simile, eccetto per il fatto che viene preceduto da una "intestazione di linea telefonica" invece che da un'intestazione Ethernet. La gestione delle intestazioni è importante ed è una delle mansioni dei processori di rete. È evidente che abbiamo appena sfiorato la superficie di una materia sconfidata quali sono le reti di computer. Per una trattazione più dettagliata si faccia riferimento a (Tanenbaum e Wetherall, 2011).



**Figura 8.15** Schema dei pacchetti Ethernet.

### Introduzione ai processori di rete

Le reti collegano molti tipi di dispositivi. Gli utenti domestici si collegano tramite PC (desktop o portatili), ma anche (e sempre più) tramite console di videogiochi, PDA (computer palmari) e smartphone; le aziende sono collegate tramite PC o server. D'altra parte, esistono anche molti dispositivi che svolgono nella rete una funzione d'intermediazione, tra cui router, commutatori, firewall, proxy web e bilanciatori di carico. È interessante notare che questi sistemi intermediari richiedono una quantità di risorse preponderante nella rete, dal momento che trasferiscono il maggior numero di pacchetti al secondo. Anche i server sono abbastanza esigenti, a differenza dei calcolatori degli utenti domestici.

Un pacchetto ricevuto può richiedere vari tipi di elaborazione, a seconda della rete o del pacchetto stesso, prima di essere inoltrato lungo la linea in uscita o prima di essere consegnato a un programma applicativo. Alcune elaborazioni possibili sono: decidere dove spedire il pacchetto, scomporlo o riassembrarlo i pezzi, gestire la sua qualità di servizio (specialmente per i flussi audio o video) o la sua sicurezza (per esempio cifrandolo o decifrandolo), comprimerlo o espanderlo, e così via.

In una LAN a 40 Gbps e con pacchetti di 1 KB, un computer di rete potrebbe dover elaborare quasi 5 milioni di pacchetti al secondo. Con pacchetti di 64 byte si arriva a circa 80 milioni di pacchetti al secondo. Lo svolgimento delle attività di cui sopra nel giro di 12-200 ns è semplicemente impossibile per il software (per non parlare del fatto che di ogni pacchetto servono più copie): è fondamentale l'assistenza dell'hardware.

Una possibile soluzione per elaborare velocemente i pacchetti prevede l'utilizzo di un **ASIC** (*Application-Specific Integrated Circuit*, "circuito integrato per applicazione specifica") progettato su misura. Un chip siffatto è una specie di programma cablato in grado di calcolare l'insieme di funzioni per cui è stato progettato. Attualmente molti router usano circuiti ASIC. Tuttavia i chip ASIC hanno numerosi problemi: innanzitutto richiedono molto tempo sia in fase di progettazione, sia in fase di produzione. Inoltre sono rigidi, perciò se gli si vuole aggiungere una funzionalità bisogna progettare e costruire un chip ex novo. La gestione dei bachi poi è un incubo, visto che l'unico modo per riparare un ASIC è progettare, costruire, spedire e installare un chip nuovo. Infine, si tratta di chip molto costosi, a meno che se ne produca un quantitativo così grande da ammortizzare i costi di sviluppo.

Un'altra possibile soluzione usa i circuiti **FPGA** (*Field Programmable Gate Array*, "circuito a matrice programmabile"), costituiti da un certo numero di porte che possono essere organizzate in un circuito mediante la modifica dei collegamenti tra di loro. Lo sviluppo di questi chip impiega molto meno tempo rispetto agli ASIC e in più gli FPGA possono essere ricablati sul campo: basta rimuoverli dal sistema e inserirli in un dispositivo speciale per la loro riprogrammazione. D'altra parte sono circuiti complessi, lenti e costosi, il che fa di loro circuiti poco richiesti, se non per certe applicazioni di nicchia.

Infine veniamo ai **processori di rete**, dispositivi programmabili che possono gestire i pacchetti in ingresso e in uscita alla stessa velocità con cui viaggiano sui collegamenti (cioè in tempo reale). Un progetto tipico vede il processore di rete posto su una scheda a innesto, insieme a una memoria e ai circuiti logici di supporto. La scheda è collegata a una o più linee di rete che sono dirette al processore di rete, che estrae i pacchetti, li

elabora e li spedisce su di una linea diversa (se è un router) o lungo il bus principale di sistema (per esempio lungo il bus PCI) nel caso si tratti di un dispositivo che si trova sul PC di un utente finale. La Figura 8.16 illustra la scheda e il chip di un comune processore di rete.

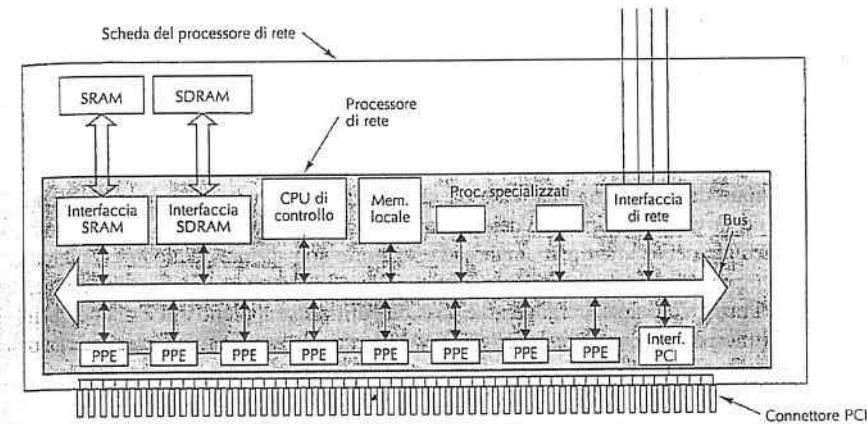


Figura 8.16 Scheda e chip di un comune processore di rete.

La scheda è provvista di memorie SRAM e SDRAM, usate in genere in modi diversi. La SRAM è più veloce e più costosa della SDRAM, perciò la sua disponibilità è molto limitata. La SRAM si usa per contenere le tabelle d'instradamento (*routing table*) e altre strutture dati importanti, mentre la SDRAM memorizza i pacchetti in elaborazione. Grazie alla separazione tra il chip e le memorie, i progettisti possono decidere liberamente le quantità di SRAM e di SDRAM da includere nella scheda. Così facendo, è possibile dotare di poca memoria le schede di fascia bassa con una sola linea in ingresso (destinate per esempio a un PC o a un server), mentre le schede di fascia alta, progettate per i grandi router, possono essere provviste di molta più memoria.

I chip dei processori di rete sono ottimizzati per elaborare velocemente un gran numero di pacchetti in ingresso e in uscita; in particolare, un router con una mezza dozzina di linee potrebbe trovarsi a elaborare milioni di pacchetti al secondo su ogni linea. L'unico modo per raggiungere queste velocità è di costruire processori di rete con molto parallelismo; infatti, ogni processore di rete contiene svariati PPE (acronimo di nomi diversi: *Protocol/Programmable/Packet Processing Engine*, "motore di elaborazione del protocollo/programmabile/di pacchetto"). Ogni PPE è un core RISC (eventualmente modificato) con un piccolo quantitativo di memoria per il programma e le variabili.

I motori PPE possono essere organizzati in due modi diversi. Il caso più semplice consiste nell'impiego di PPE identici: quando il processore di rete riceve un pacchetto (dalla rete o dal bus) lo smista a un PPE inattivo perché lo elabori. Se tutti i PPE sono

impegnati, il pacchetto entra in una coda contenuta nella SDRAM della scheda, in attesa che si liberi un PPE. In un'organizzazione di questo tipo non ci sono i collegamenti orizzontali mostrati nella Figura 8.16 tra un PPE e l'altro, perché questi non hanno alcun bisogno di comunicare.

L'altro tipo di organizzazione dei PPE è a pipeline, e ogni motore esegue un passo di elaborazione e inserisce nel pacchetto in uscita un puntatore al PPE successivo nella pipeline. In questo modo, la pipeline dei PPE somiglia molto alle pipeline delle CPU che abbiamo studiato nel Capitolo 2. In entrambe le organizzazioni, i motori PPE sono integralmente programmabili.

I progetti più avanzati prevedono PPE con multithreading, dotati perciò di diversi insiemi di registri e di un particolare registro che determina l'insieme correntemente in uso. Grazie a questa caratteristica è possibile eseguire più programmi alla volta e la commutazione da un programma (cioè da un thread) all'altro avviene semplicemente modificando la variabile “insieme dei registri corrente”. Comunemente, un PPE comuta immediatamente a favore di un thread eseguibile non appena quello in esecuzione va in stallo, per esempio a seguito di un accesso alla SDRAM (che richiede diversi cicli di clock). Grazie a questo accorgimento è possibile utilizzare appieno i PPE anche quando si bloccano per frequenti accessi alla SDRAM o quando eseguono un'altra operazione esterna piuttosto lenta.

Oltre ai PPE, i processori di rete contengono sempre un processore di controllo (in genere una CPU RISC) impiegato per svolgere tutto quel lavoro che non ha a che fare con l'elaborazione dei pacchetti, come l'aggiornamento delle tabelle d'instradamento. Il suo programma e i suoi dati si trovano in memoria locale del chip. Inoltre, molti chip di rete contengono uno o più processori specializzati per l'esecuzione di operazioni di corrispondenza tra forme (*pattern matching*) o di altre operazioni critiche. Si tratta per lo più di piccoli ASIC in grado di effettuare una sola, semplice operazione, come la ricerca di un indirizzo di destinazione all'interno della tabella d'instradamento. Tutti i componenti del processore di rete comunicano per mezzo di uno o più bus paralleli del chip, con velocità di svariati Gigabit al secondo.

### Elaborazione dei pacchetti

Alla sua ricezione, un pacchetto passa attraverso un certo numero di fasi di elaborazione, indipendentemente dal fatto che il processore di rete sia o meno organizzato a pipeline. Alcuni processori di rete dividono queste fasi in due tipi di operazioni, quelle da effettuare sui pacchetti in ingresso (dalla linea di rete o dal bus di sistema), dette perciò **elaborazione in entrata dei pacchetti di rete**, e quelle da effettuare sui pacchetti in uscita, che costituiscono l'**elaborazione in uscita dei pacchetti di rete**. Quando si utilizza questa distinzione, un pacchetto subisce prima l'elaborazione in entrata, poi quella in uscita. La demarcazione tra operazioni in entrata e in uscita è flessibile, perché alcune fasi possono essere svolte indifferentemente in uno dei due momenti (per esempio la raccolta delle statistiche sul traffico).

In seguito analizziamo un possibile ordinamento delle diverse fasi, ma si tenga presente che non tutti i pacchetti necessitano di tutte le fasi e che sono possibili altri ordinamenti ugualmente validi.

- Verifica della checksum.** Se il pacchetto in entrata proviene da Ethernet, viene ricalcolato il valore CRC e confrontato con quello nel pacchetto per verificare che non ci siano stati errori di trasmissione. Se il CRC di Ethernet è corretto o assente, viene ricalcolata la checksum IP e confrontata con quella del pacchetto IP per assicurarsi che non sia stato danneggiato da un bit difettoso della memoria del mittente e che avrebbe lasciato traccia nella checksum ivi calcolata. Se le checksum sono corrette, il pacchetto viene accettato per essere ulteriormente elaborato; in caso contrario, viene semplicemente scartato.
- Estrazione di campi.** Viene analizzata la parte significativa dell'intestazione e vengono estratti i campi più importanti. All'interno di un commutatore Ethernet viene esaminata solo l'intestazione Ethernet, mentre in un router IP viene ispezionata l'intestazione IP. I campi chiave sono memorizzati nei registri (organizzazione con PPE paralleli) o nella SRAM (organizzazione a pipeline).
- Classificazione dei pacchetti.** Il pacchetto è classificato in base a una serie di regole programmabili. La classificazione più semplice distingue tra pacchetti di dati e di controllo, ma di solito si effettuano distinzioni molto più accurate.
- Selezione del percorso.** Molti processori di rete predispongono un percorso veloce e ottimizzato per la gestione dei pacchetti più comuni, mentre gli altri pacchetti sono trattati diversamente, spesso a opera del processore di controllo. Si effettua quindi una selezione tra il percorso veloce e quello lento.
- Determinazione del destinatario di rete.** I pacchetti IP contengono l'indirizzo del destinatario di 32 bit. Non è possibile, né tanto meno auspicabile, effettuare la ricerca della destinazione di ogni pacchetto IP all'interno di una tabella di  $2^{32}$  elementi, perciò la parte più significativa dell'indirizzo IP viene usata per identificare il numero di rete, il resto per specificare una macchina in quella rete. I numeri di rete possono avere una lunghezza arbitraria, perciò la determinazione del numero della rete destinataria non è banale ed è complicata dal fatto che ci sono più corrispondenze possibili e che quella giusta è la più lunga. In genere questa fase è affidata a un circuito ASIC ad hoc.
- Instrandamento.** Una volta determinata la rete del destinatario, si sceglie la linea d'uscita lungo cui instradare il pacchetto cercandola in una tabella della SRAM. Anche per questa fase si può usare un circuito ASIC.
- Scomposizione e riassemblaggio.** I programmi cercano di passare a livello TCP ingenti quantitativi di dati per ridurre il numero di chiamate di sistema, ma TCP, IP e Ethernet definiscono tuttavia una dimensione massima per i pacchetti che possono gestire. In conseguenza di queste limitazioni, i dati utili e i pacchetti vengono scomposti dal mittente e i pezzi risultanti vengono riassemblati dal destinatario. È uno dei compiti che possono essere svolti dal processore di rete.
- Elaborazione.** Alle volte sono necessari calcoli molto pesanti sui dati che costituiscono il messaggio (come compressione/decompressione o cifratura/decifratura). È un altro compito affidabile al processore di rete.

9. Gestione dell'intestazione. Può essere necessario aggiungere, rimuovere o modificare alcuni campi dell'intestazione. Per esempio, l'intestazione IP ha un campo che conta il numero di salti che il pacchetto può effettuare prima di venire scartato. Questo campo va decrementato dopo ogni ritrasmissione: un altro compito adatto al processore di rete.
10. Gestione della coda. I pacchetti in entrata o in uscita vengono spesso messi in coda mentre attendono il loro turno di elaborazione. Le applicazioni multimediali potrebbero aver bisogno di un certo intervallo tra un pacchetto e l'altro per evitare il fenomeno del *jitter*. Un firewall o un router potrebbero distribuire il carico di pacchetti in entrata su diverse linee in uscita in base ad alcune regole. Tutte queste attività possono essere svolte dal processore di rete.
11. Generazione della checksum. I pacchetti in uscita devono essere provvisti di checksum. La checksum IP può essere generata dal processore di rete, ma il valore di CRC Ethernet di solito viene calcolato dall'hardware.
12. Contabilità. In alcuni casi si richiede un po' di contabilità sul traffico di pacchetti, soprattutto quando una rete instrada traffico per conto di altre reti come servizio commerciale. Il processore di rete può occuparsi della contabilità.
13. Raccolta di statistiche. Infine, molte organizzazioni intendono accumulare statistiche circa il proprio traffico, per esempio quanti pacchetti sono stati ricevuti e quanti pacchetti sono stati inviati e a che ora del giorno, e altro ancora. Il processore di rete si presta bene alla raccolta di questi dati.

#### Incremento delle prestazioni

Le prestazioni sono fondamentali per i processori di rete: che cosa si può fare per migliorarle? Prima di rispondere, dobbiamo chiarire cosa si intende per prestazioni. Una metrica possibile è il numero di pacchetti inoltrati ogni secondo, un'altra è il numero di byte inoltrati al secondo. Sono due misure differenti e non è detto che uno schema che funziona bene con pacchetti piccoli funzioni altrettanto bene con pacchetti più grandi. In particolare, nel primo caso potrebbe giovare una riduzione del numero di destinazioni ricercate in ogni secondo, mentre ciò potrebbe non essere vero nel caso di pacchetti grandi.

Il modo più diretto per migliorare le prestazioni è aumentare la velocità di clock del processore di rete. Ovviamente il miglioramento non è lineare, perché può essere influenzato dal tempo di ciclo della memoria e da altri fattori. E poi, un clock più veloce vuol dire più calore da dissipare.

Una soluzione che molto spesso ripaga è l'aggiunta di più PPE e l'incremento del parallelismo, soprattutto quando si usa l'organizzazione parallela dei PPE. Anche l'allungamento della pipeline può aiutare, ma solo se l'intera elaborazione di un pacchetto sia suddivisibile in fasi più corte.

Un'altra tecnica possibile prevede l'adozione di un processore specializzato o di un ASIC per gestire specificatamente le operazioni che richiedono molto tempo, che vengono eseguite ripetutamente e la cui l'esecuzione via hardware risulta molto più veloce

di quella via software. Tra i candidati ci sono sicuramente le ricerche nelle tabelle, il calcolo della checksum e la cifratura.

L'aggiunta di un maggior numero di bus interni e l'ampliamento dei bus esistenti può aiutare ad aumentare la velocità contribuendo ad accelerare lo spostamento dei pacchetti nel sistema. Infine, tra i miglioramenti si può considerare anche la sostituzione della SDRAM con memoria SRAM, ma che comporta costi aggiuntivi.

Naturalmente resta molto da dire sui processori di rete. Alcuni riferimenti in tal senso sono (Freitas et al., 2009; Lin et al., 2010; Yamamoto e Nakao, 2011).

#### 8.2.2 Processori grafici

Un altro settore in cui si rendono utili i coprocessori è quello della gestione di elaborazioni grafiche ad alta risoluzione, come il rendering 3D. Le normali CPU non sono molto dotate per elaborare volumi di dati così ingenti come quelli richiesti da queste applicazioni. Perciò la maggior parte dei PC e molti dei processori futuri saranno dotati di GPU (Graphics Processing Units) a cui affidare gran parte del lavoro di elaborazione.

##### La GPU NVIDIA Fermi

Ci occupiamo del settore dei processori grafici, la cui importanza è in continua ascesa, con un esempio concreto: la GPU NVIDIA Fermi, un'architettura utilizzata in una famiglia di chip per l'elaborazione grafica disponibili con diverse velocità e dimensioni. L'architettura della GPU Fermi, mostrata nella Figura 8.17, è organizzata in 16 SM (*Streaming Multiprocessor*) ognuno con una sua cache L1 privata a elevata larghezza di banda. Ogni SM contiene 32 core CUDA, per un totale di 512 core CUDA in ogni GPU Fermi. Un core CUDA (*Compute Unified Device Architecture*) è un semplice processore che supporta calcolo intero a singola precisione e in virgola mobile. Un SM con 32 core CUDA è mostrato nella Figura 2.7. I 16 SM condividono l'accesso a un'unica cache unificata di secondo livello di 768 KB, che è connessa a un'interfaccia DRAM dotata di più porte. L'interfaccia del processore host offre un percorso di comunicazione tra il sistema host e la GPU per mezzo di un'interfaccia bus DRAM condivisa, di solito utilizzando un'interfaccia PCI-Express.

L'architettura Fermi è progettata per eseguire in maniera efficiente codice per l'elaborazione grafica, video e di immagini, che di solito contiene computazioni ridondanti distribuite su un gran numero di pixel. A causa di questa ridondanza gli SM, in grado di eseguire 16 operazioni alla volta, eseguono in un singolo ciclo operazioni identiche. Questo stile di elaborazione viene chiamato elaborazione SIMD (*Single-Instruction Multiple Data*) e ha l'importante vantaggio che ogni SM preleva e decodifica una sola istruzione per ciclo. Per la condivisione dell'elaborazione delle istruzioni tra tutti i core di un SM, NVIDIA dispone di 512 core su un unico stampo di silicio. Se i programmati fossero in grado di sfruttare tutte le risorse di calcolo (si tratta di un "se" grande e incerto), il sistema offrirebbe notevoli vantaggi computazionali rispetto alle architetture scalari tradizionali, come il Core i7 o l'OMAP4430.

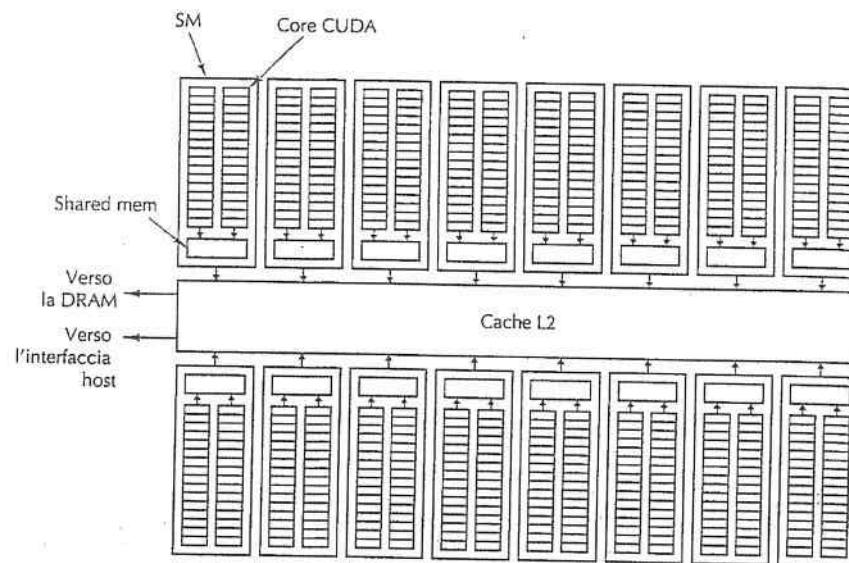


Figura 8.17 L'architettura della GPU Fermi.

I requisiti dell'elaborazione SIMD all'interno degli SM vincolano la tipologia di codice eseguibile su queste unità. Ogni core CUDA, per poter raggiungere le 16 operazioni simultanee, deve infatti eseguire lo stesso codice. Per alleggerire questo onere a carico del programmatore, NVIDIA ha sviluppato il linguaggio di programmazione CUDA. Questo linguaggio specifica il parallelismo del programma utilizzando i thread che vengono poi raggruppati in blocchi, e assegnati agli SM. Se ogni thread in un blocco esegue esattamente la stessa sequenza di codice (cioè, tutte le diramazioni prendono la stessa decisione), verranno eseguite fino a 16 operazioni simultanee (supponendo che vi siano 16 thread pronti per l'esecuzione). Quando i thread su un SM prendono diramazioni diverse, si verifica un effetto dannoso per le prestazioni chiamato divergenza di diramazione. Tale effetto forza i thread con sequenze di codice differenti a essere eseguiti in serie sullo SM. La divergenza di diramazione riduce il parallelismo e rallenta l'elaborazione della GPU. Fortunatamente, vi è una vasta gamma di attività nell'elaborazione grafica e di immagini in grado di evitare divergenze di diramazione e ottenere un buon incremento di velocità. È stato dimostrato che molti altri contesti possono beneficiare dell'architettura SIMD dei processori grafici, come per esempio l'imaging in ambito medico, la dimostrazione automatica, la previsione finanziaria e l'analisi di grafi. Questo allargamento delle potenziali applicazioni delle GPU ha valso loro il nuovo soprannome di **GPGPU** (*General-Purpose Graphics Processing Unit*).

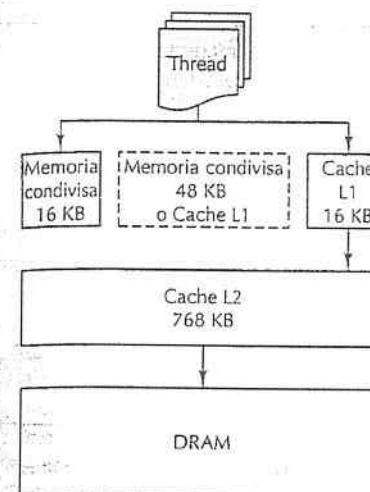


Figura 8.18 La gerarchia di memoria della GPU Fermi.

Senza un'adeguata larghezza di banda la GPU Fermi, con i suoi 512 core CUDA, arriverebbe ad arrestarsi. Per fornire questa larghezza di banda, la GPU Fermi implementa una moderna gerarchia di memoria, come illustrato nella Figura 8.18. Ogni SM ha sia una memoria dedicata condivisa che una cache dati-privata di primo livello. La memoria dedicata condivisa viene indirizzata direttamente dai core CUDA e fornisce una condivisione veloce dei dati tra i thread all'interno di un singolo SM. La cache di primo livello velocizza l'accesso ai dati sulla DRAM. Per far spazio alla vasta gamma di dati gli SM possono essere configurati con 16 KB di memoria condivisa e 48 KB di cache di primo livello oppure con 48 KB di memoria condivisa e 16 KB di cache L1. Tutti gli SM condividono una cache unificata di secondo livello di 768 KB. La cache L2 offre un accesso veloce ai dati sulla DRAM che non trovano spazio sulla cache di primo livello e fornisce un meccanismo di condivisione tra gli SM, anche se questa modalità di condivisione è più lenta della condivisione intra-SM che si ha all'interno della memoria condivisa degli SM. Dopo la cache di secondo livello vi è la DRAM, che mantiene altri dati, immagini e texture utilizzati dai programmi in esecuzione sulla GPU Fermi. I programmi efficienti proveranno in tutti i modi a evitare accessi alla DRAM, perché un solo accesso richiede centinaia di cicli per essere completato.

Per un programmatore esperto, la GPU Fermi rappresenta una delle piattaforme computazionalmente più capaci mai create. Una sola GPU GTX 580 basata su Fermi in esecuzione a 772 MHz con 512 core CUDA può sostenere velocità di calcolo di 1,5 teraflop consumando 250 watt di potenza. Questi valori sono ancora più impressionanti se si considera che il prezzo al dettaglio di una GPU GTX 580 GPU è inferiore ai 600 dollari. A titolo di confronto, nel 1990 il computer più veloce al mondo, il Cray-2, aveva una velocità di 0,002 teraflop e un prezzo (calcolato tenendo conto dell'inflazione) di 30

milioni di dollari. Riempiva inoltre una piccola stanza ed era dotato di un proprio sistema di raffreddamento per dissipare i 150 KW di potenza consumata. La GTX 580 ha una potenza di calcolo 750 volte superiore, costa 1/50000 del prezzo e consuma 1/600 di energia. Non è certo un cattivo affare.

### 8.2.3 Crittoprocessori

La sicurezza è il terzo settore in cui sono molto diffusi i coprocessori, soprattutto per quanto riguarda le reti. Quando si stabilisce una connessione tra un client e un server, spesso la prima cosa che viene richiesta a entrambi è l'autenticazione reciproca. A tal fine stabiliscono una connessione cifrata così da poter trasferire i dati in modo sicuro, vanificando ogni tentativo di intromissione nella linea.

Il problema legato alla sicurezza è che, per garantirla, bisogna ricorrere alla crittografia, che richiede però molte risorse di calcolo. Esistono due tipologie di metodi crittografici: a chiave simmetrica e a chiave pubblica. La prima si basa sull'idea di confondere tutti i bit del messaggio, producendo più o meno ciò che si otterrebbe inserendolo in un frullatore elettronico. La seconda si basa sulla moltiplicazione e sull'elevamento a potenza di numeri molto grandi (per esempio di 1024 bit) e impegna molte risorse di tempo.

Molte aziende hanno prodotto *critto-coprocessori*, spesso sotto forma di schede a innesto nel bus PCI, per gestire i calcoli richiesti dalla cifratura sicura dei dati (per la loro trasmissione o memorizzazione) e dalla loro successiva decifratura. Questi coprocessori sono provvisti di hardware speciale che consente loro di svolgere tutte le necessarie operazioni crittografiche molto più velocemente di quanto possa fare una CPU. Sfortunatamente, la descrizione dettagliata del funzionamento dei crittoprocessori richiederebbe innanzitutto una spiegazione abbastanza accurata della stessa crittografia, il che va oltre gli scopi di questo libro. Si possono trovare maggiori informazioni sui critto-coprocessori in (Gaspar et al., 2010; Haghaghizadeh et al., 2010; Shoufan et al., 2011).

## 8.3 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 multicompiler. Cominciamo con il definire il significato di questi termini, poi ci inoltreremo nell'analisi dei multiprocessori e dei multicompiler.

### 8.3.1 Multiprocessori e multicompiler a confronto

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. Le modalità della comunicazione sono oggetto di grande dibattito nella comunità degli architetti degli elaboratori. Sono stati proposti e implementati due progetti diversi, i multiprocessori e i multicompiler, che si distinguono per la presenza o per l'assenza di memoria

condivisa. Questa differenza influisce sul loro progetto, sulla loro costruzione e programmazione, nonché sui loro costi e dimensioni.

### Multiprocessori

Un multiprocessore è un calcolatore in cui tutte le CPU condividono una memoria comune, come schematizzato nella Figura 8.19. 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.

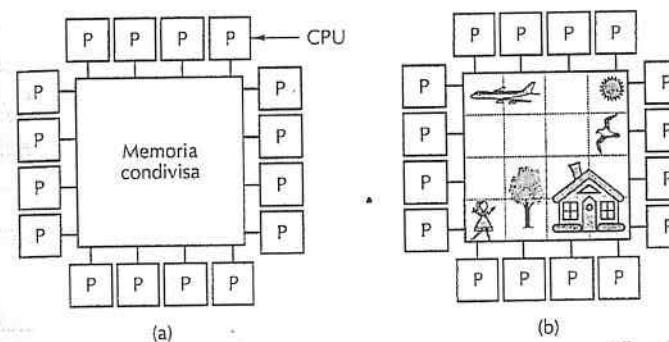


Figura 8.19 (a) Multiprocessore con 16 CPU che condividono una memoria comune. (b) Immagine ripartita in 16 porzioni, ciascuna analizzata da una CPU diversa.

La facilità con cui due (o più) processi riescono a comunicare tramite operazioni sulla memoria è la ragione della diffusione dei multiprocessori. È un modello semplice da capire per i programmati ed è applicabile con successo a un gran numero di problemi. Si consideri, per esempio, un programma che ispeziona un'immagine e che elenca tutti gli oggetti in essa contenuti. Una copia dell'immagine è contenuta in memoria, come mostrato dalla Figura 8.19(b). Ciascuna delle 16 CPU elabora un processo singolo cui è stata assegnata una delle 16 porzioni dell'immagine da analizzare. Ciononostante, ogni processo ha accesso all'intera immagine, il che è essenziale poiché alcuni oggetti possono occupare più sezioni contigue. Se un processo scopre che un oggetto si estende all'interno di un'altra porzione, può seguire l'oggetto semplicemente leggendo le parole di quella porzione. In questo esempio alcuni oggetti verranno scoperti da più processi, perciò sarà necessario attribuire delle coordinate che permettano di contare a posteriori quante case, alberi e aeroplani ci sono nell'immagine.

Dal momento che tutte le CPU di un multiprocessore vedono la stessa immagine della memoria, c'è una sola copia del sistema operativo e, di conseguenza, c'è una sola

mappa delle pagine e una sola tabella dei processi. 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.

### 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 8.20(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. Dunque i multiprocessori hanno un solo spazio degli indirizzi fisici comune a tutte le CPU, mentre i multicomputer hanno uno spazio degli indirizzi fisici per ogni CPU.

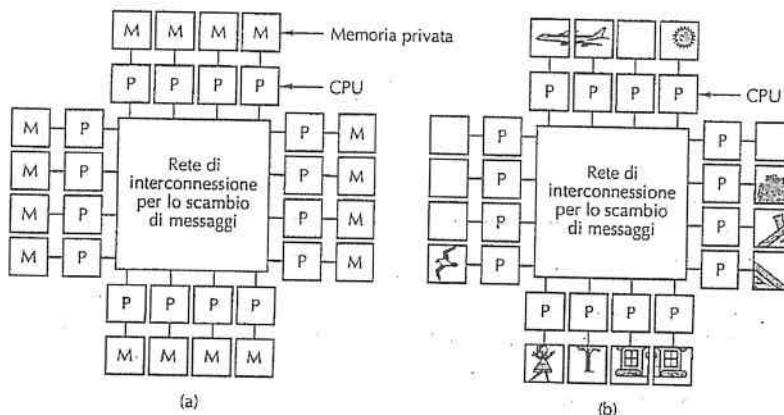


Figura 8.20 (a) Multicomputer con 16 CPU, ciascuna con la propria memoria privata. (b) Immagine bitmap della Figura 8.19 suddivisa tra le 16 memorie.

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. Tra i multicomputer annoveriamo BlueGene/L di IBM, Red Storm e i cluster di Google.

L'assenza di una memoria condivisa via hardware ha implicazioni importanti sulla struttura del software di un multicomputer. Non è più possibile avere un solo spazio virtuale degli indirizzi in cui tutti i processi sono in grado di leggere e scrivere per mezzo dell'esecuzione di semplici istruzioni LOAD e STORE. Per esempio, se la CPU 0 della Figura 8.19(b) (quella nell'angolo in alto a sinistra) scopre che parte del proprio oggetto si estende all'interno della porzione assegnata alla CPU 1, può comunque accedere alla coda dell'aeroplano continuando a leggere dati dalla memoria. Viceversa, quando la CPU 0 della Figura 8.20(b) fa la stessa scoperta, non può leggere in alcun modo la memoria della CPU 1. Per accedere a quei dati deve avvalersi di un meccanismo sostanzialmente diverso.

Infatti deve innanzitutto scoprire (in qualche modo) quale CPU possiede i dati che le interessano e spedire 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. Ciò rende la struttura del software molto diversa e di gran lunga più complessa rispetto a un multiprocessore. Inoltre è chiaro come diventi particolarmente problematico in un multicomputer riuscire a suddividere i dati correttamente e assegnarli alle locazioni in modo ottimale; la questione non costituiva una grossa problema nei multiprocessori perché in quel caso la collocazione dei dati non influiva sulla correttezza o sulla programmabilità del codice, tutt'al più sulle sue prestazioni. In poche parole, programmare un multicomputer è molto più difficile che programmare un multiprocessore.

Se le cose stanno così, perché mai qualcuno dovrebbe voler costruire multicomputer, quando i multiprocessori sono più facili da programmare? La risposta è semplice: 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. Nel seguito del capitolo studieremo un multicomputer con più di 50.000 CPU.

Ci troviamo davanti a un dilemma: i multiprocessori sono difficili da costruire, ma facili da programmare, i multicomputer sono facili da costruire, ma difficili da programmare. Per queste ragioni 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. Così una fetta considerevole della ricerca attuale nelle architetture parallele si concentra sulla convergenza di architetture multiprocessore e multicomputer che conservi i punti di forza di ciascuna. Il "Santo Graal" di questa disciplina è riuscire a ideare progetti scalabili, ovvero che continuino a comportarsi bene anche al crescere delle dimensioni per l'aggiunta di altre CPU.

Un approccio per la costruzione di sistemi ibridi si basa sul fatto che i sistemi di calcolo moderni non sono monolitici, bensì costruiti come una serie di livelli (il tema di questo libro). Questa intuizione apre la strada all'implementazione della memoria condivisa a livelli diversi, come mostra la Figura 8.21. Nella Figura 8.21(a) la memoria condivisa è implementata dall'hardware come in un vero multiprocessore. Secondo questo progetto, c'è una sola copia del sistema operativo con un solo insieme di tabelle e, in particolare, con una sola tabella di allocazione della memoria. Quando un processo ha bisogno di più memoria, invia una trap al sistema operativo e questo cerca una pagina libera nella propria tabella e la mappa nello spazio degli indirizzi del chiamante. 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. Come vedremo, sono possibili molte implementazioni hardware di una memoria condivisa.

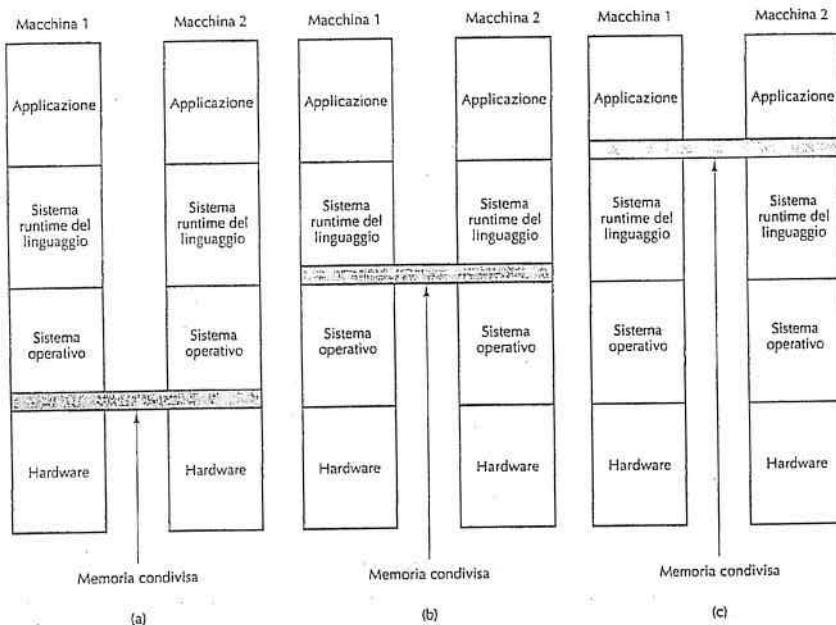


Figura 8.21 Vari livelli d'implementazione di una memoria condivisa. (a) Nell'hardware. (b) Nel sistema operativo. (c) Nel sistema runtime del linguaggio.

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 (*Distributed Shared Memory*, "memoria condivisa distribuita"; Li e Hudak, 1989) e prevede che ogni

pagina sia localizzata in una delle memorie della Figura 8.20(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; questo localizza la pagina e richiede alla CPU che la possiede correntemente di estrometterla dalla memoria e di spedirla lungo la rete di interconnessione. Una volta recapitata, la pagina viene mappata in memoria della prima CPU e l'istruzione può riprendere la propria esecuzione. A tutti gli effetti, il sistema operativo non sta facendo altro che gestire gli errori di pagina attingendo a una memoria distante invece che al disco. Dal punto di vista dell'utente, il sistema sembra un sistema a memoria virtuale. Analizzeremo DSM nel prosieguo del capitolo.

Una terza possibilità è implementare la memoria condivisa all'interno del sistema runtime (eventualmente specifico di un linguaggio) 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. Per esempio, il modello Linda si basa sull'astrazione di uno spazio condiviso di tuple (record di dati contenenti un insieme di campi). 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 (dal sistema runtime di Linda) non c'è bisogno di hardware speciale o di supporto da parte del sistema operativo.

Un altro esempio di memoria condivisa implementata dal sistema runtime di un linguaggio è il modello Orcà di condivisione degli oggetti di dati. In Orcà, i processi condividono oggetti generici e non tuple, e possono eseguire su di loro alcuni metodi specifici degli oggetti stessi. Quando un metodo invoca una modifica allo stato interno di un oggetto, spetta al sistema runtime assicurarsi che tutte le copie dell'oggetto residenti sulle diverse macchine vengano aggiornate simultaneamente. Sottolineiamo che non c'è bisogno di alcuna assistenza da parte dell'hardware o del sistema operativo perché gli oggetti sono un concetto puramente software. In questo capitolo ci occuperemo ancora di Linda e di Orcà.

### Tassonomia dei calcolatori paralleli

Torniamo ora al nostro argomento principale: l'architettura dei calcolatori paralleli. Nel corso degli anni sono stati proposti e costruiti molti tipi di calcolatori paralleli, dunque viene naturale chiedersi se è possibile organizzarli in una tassonomia di categorie. Molti ricercatori si sono dedicati a questa attività, con risultati diversi (Flynn, 1972; Treleaven, 1985). Purtroppo non è ancora nato il Linneo<sup>3</sup> del calcolo parallelo. L'unico schema abbastanza usato è quello di Flynn (Figura 8.22), che resta comunque molto approssimativo. La classificazione di Flynn si basa su due concetti: i flussi d'istruzioni e i flussi di dati.

Un flusso d'istruzioni corrisponde a un program counter. Un sistema con  $n$  CPU ha  $n$  program counter, quindi  $n$  flussi d'istruzioni.

<sup>3</sup> Carlo Linneo (1707-1778): biologo svedese che inventò il sistema di classificazione delle piante e degli animali ancora in uso e che cataloga gli individui in base alle categorie di regno, phylum (tipo o divisione), classe, ordine, famiglia, genere e specie.

Flussi d'istruzioni	Flussi di dati	Nome	Esempi
1	1	SISD	Modello di Von Neumann
1	Molteplici	SIMD	Supercomputer vettoriali, unità di calcolo vettoriali
Molteplici	1	MISD	Forse nessuno
Molteplici	Molteplici	MIMD	Multiprocessori, multicompiler

Figura 8.22 Tassonomia di Flynn dei calcolatori paralleli.

Un flusso di dati è un insieme di operandi. Per esempio, in un sistema per le previsioni metereologiche dotato di un gran numero di sensori ogni sensore potrebbe emettere un flusso di temperature a intervalli regolari.

I flussi d'istruzioni e di dati sono in buona parte indipendenti, perciò si danno le quattro possibili combinazioni della Figura 8.22. SISD è la macchina sequenziale classica di Von Neumann, caratterizzata da un solo flusso d'istruzioni, un solo flusso di dati e svolge una sola attività alla volta. Le macchine SIMD hanno una sola unità di controllo che emette un'istruzione alla volta, ma dispongono di varie ALU per operare simultaneamente su diversi insiemi di dati. L'ILLIAC IV (Figura 2.7) è il prototipo di macchina SIMD. I mainframe SIMD sono sempre più rari, sebbene i calcolatori convenzionali possano avere alcune istruzioni SIMD per l'elaborazione di materiale audiovisivo (per esempio le istruzioni SSE del Core i7). Comunque sia, c'è un nuovo settore in cui sembrano giocare un ruolo importante le idee del paradigma SIMD: gli *stream processor*. Si tratta di macchine progettate specificatamente per la gestione del rendering multimediale e potrebbero acquisire in futuro una certa importanza (Kapasi et al., 2003).

Le macchine MISD fanno parte di una categoria un po' strana, in cui ci sono molteplici istruzioni che operano sugli stessi dati. Non è ben chiaro se esista una qualche macchina di questo tipo, anche se alcuni ritengono che le macchine a pipeline siano macchine MISD.

Infine vengono le macchine MIMD, semplicemente un insieme di CPU indipendenti che operano come componenti di un sistema più grande. Molti dei calcolatori paralleli fanno parte di questa categoria. I multiprocessori e i multicompiler sono macchine MIMD. La tassonomia di Flynn si ferma qui, ma l'abbiamo arricchita come mostrato nella Figura 8.23. Abbiamo suddiviso le macchine SIMD in due sottogruppi: il primo comprende i supercomputer numerici e le macchine che operano su vettori, effettuando la stessa operazione su ciascun elemento del vettore; il secondo contiene le macchine di tipo parallelo, come l'ILLIAC IV, in cui un'unità principale di controllo invia le istruzioni a molte ALU indipendenti.

Nella nostra tassonomia abbiamo diviso la categoria MIMD in multiprocessori (macchine a condivisione di memoria) e multicompiler (macchine a scambio di messaggi). Esistono tre tipi di multiprocessori, caratterizzati dal modo in cui implementano la memoria condivisa: sono chiamati UMA (*Uniform Memory Access*, "accesso uniforme alla memoria"), NUMA (*NonUniform Memory Access*) e COMA (*Cache Only Memory Access*). Queste sottocategorie esistono perché in genere la memoria dei multiprocessori molto grandi è divisa in più moduli. Le macchine UMA hanno la proprietà di garan-

tire lo stesso tempo d'accesso a ogni modulo di memoria da parte di ogni CPU, ovvero ogni parola di memoria può essere letta tanto velocemente quanto qualsiasi altra. Se questa proprietà è tecnicamente impossibile da garantire, gli accessi più veloci sono rallentati alla velocità dei più lenti così che i programmati non si accorgano della differenza. È questa l'accezione di "uniforme" che si intende in questo contesto. L'uniformità permette di fare previsioni sulle prestazioni, il che è importante per la scrittura di codice efficiente.

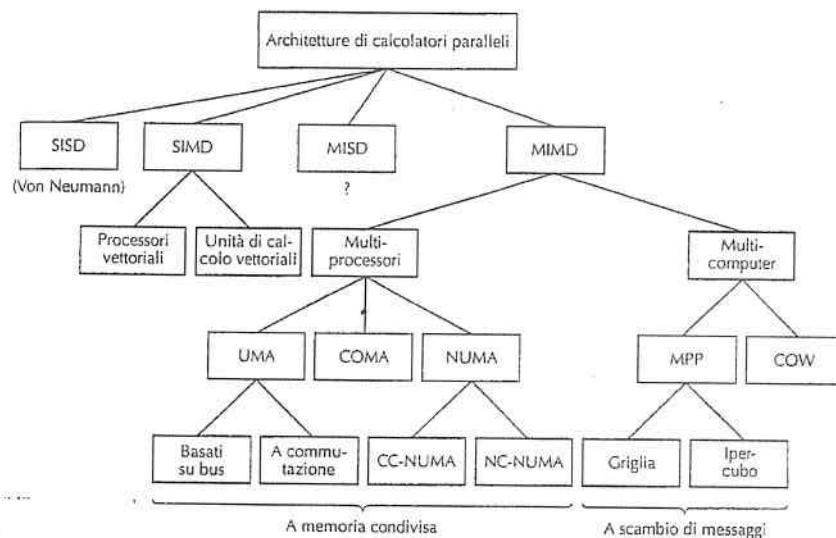


Figura 8.23 Tassonomia dei calcolatori paralleli.

Viceversa, nei multiprocessori NUMA questa proprietà non vale. Spesso c'è un modulo di memoria vicino a ogni CPU e il suo accesso è molto più veloce rispetto ai moduli distanti. Perciò, ai fini delle prestazioni, conta molto la collocazione dei dati e dei programmi. Anche le macchine COMA non sono uniformi, ma in un modo diverso. Ci occuperemo più tardi di queste categorie di macchine.

L'altra categoria principale delle macchine MIMD è costituita dai multicompiler che, a differenza dei multiprocessori, non hanno una memoria principale condivisa a livello di architettura. In altre parole, il sistema operativo di una CPU di un multicompiler non può accedere alla memoria di un'altra CPU tramite la sola esecuzione di un'istruzione LOAD. Deve invece spedire un messaggio e aspettare la relativa risposta. La capacità del sistema operativo di leggere parole a distanza effettuando una semplice LOAD è ciò che distingue i multiprocessori dai multicompiler. Abbiamo già detto che anche i programmi utente di un multicompiler possono essere resi in grado di accedere alle memorie distanti tramite istruzioni LOAD e STORE, però si tratta in questo caso di

Flussi d'istruzioni	Flussi di dati	Nome	Esempi
1	1	SISD	Modello di Von Neumann
1	Molteplici	SIMD	Supercomputer vettoriali, unità di calcolo vettoriali
Molteplici	1	MISD	Forse nessuno
Molteplici	Molteplici	MIMD	Multiprocessori, multicompiler

Figura 8.22 Tassonomia di Flynn dei calcolatori paralleli.

Un flusso di dati è un insieme di operandi. Per esempio, in un sistema per le previsioni metereologiche dotato di un gran numero di sensori ogni sensore potrebbe emettere un flusso di temperature a intervalli regolari.

I flussi d'istruzioni e di dati sono in buona parte indipendenti, perciò si danno le quattro possibili combinazioni della Figura 8.22. SISD è la macchina sequenziale classica di Von Neumann, caratterizzata da un solo flusso d'istruzioni, un solo flusso di dati e svolge una sola attività alla volta. Le macchine SIMD hanno una sola unità di controllo che emette un'istruzione alla volta, ma dispongono di varie ALU per operare simultaneamente su diversi insiemi di dati. L'ILLIAC IV (Figura 2.7) è il prototipo di macchina SIMD. I mainframe SIMD sono sempre più rari, sebbene i calcolatori convenzionali possano avere alcune istruzioni SIMD per l'elaborazione di materiale audiovisivo (per esempio le istruzioni SSE del Core i7). Comunque sia, c'è un nuovo settore in cui sembrano giocare un ruolo importante le idee del paradigma SIMD: gli *stream processor*. Si tratta di macchine progettate specificatamente per la gestione del rendering multimediale e potrebbero acquisire in futuro una certa importanza (Kapasi et al., 2003).

Le macchine MISD fanno parte di una categoria un po' strana, in cui ci sono molteplici istruzioni che operano sugli stessi dati. Non è ben chiaro se esista una qualche macchina di questo tipo, anche se alcuni ritengono che le macchine a pipeline siano macchine MISD.

Infine vengono le macchine MIMD, semplicemente un insieme di CPU indipendenti che operano come componenti di un sistema più grande. Molti dei calcolatori paralleli fanno parte di questa categoria. I multiprocessori e i multicompiler sono macchine MIMD. La tassonomia di Flynn si ferma qui, ma l'abbiamo arricchita come mostrato nella Figura 8.23. Abbiamo suddiviso le macchine SIMD in due sottogruppi: il primo comprende i supercomputer numerici e le macchine che operano su vettori, effettuando la stessa operazione su ciascun elemento del vettore; il secondo contiene le macchine di tipo parallelo, come l'ILLIAC IV, in cui un'unità principale di controllo invia le istruzioni a molte ALU indipendenti.

Nella nostra tassonomia abbiamo diviso la categoria MIMD in multiprocessori (macchine a condivisione di memoria) e multicompiler (macchine a scambio di messaggi). Esistono tre tipi di multiprocessori, caratterizzati dal modo in cui implementano la memoria condivisa: sono chiamati UMA (*Uniform Memory Access*, "accesso uniforme alla memoria"), NUMA (*NonUniform Memory Access*) e COMA (*Cache Only Memory Access*). Queste sottocategorie esistono perché in genere la memoria dei multiprocessori molto grandi è divisa in più moduli. Le macchine UMA hanno la proprietà di garan-

tire lo stesso tempo d'accesso a ogni modulo di memoria da parte di ogni CPU, ovvero ogni parola di memoria può essere letta tanto velocemente quanto qualsiasi altra. Se questa proprietà è tecnicamente impossibile da garantire, gli accessi più veloci sono rallentati alla velocità dei più lenti così che i programmati non si accorgano della differenza. È questa l'accezione di "uniforme" che si intende in questo contesto. L'uniformità permette di fare previsioni sulle prestazioni, il che è importante per la scrittura di codice efficiente.

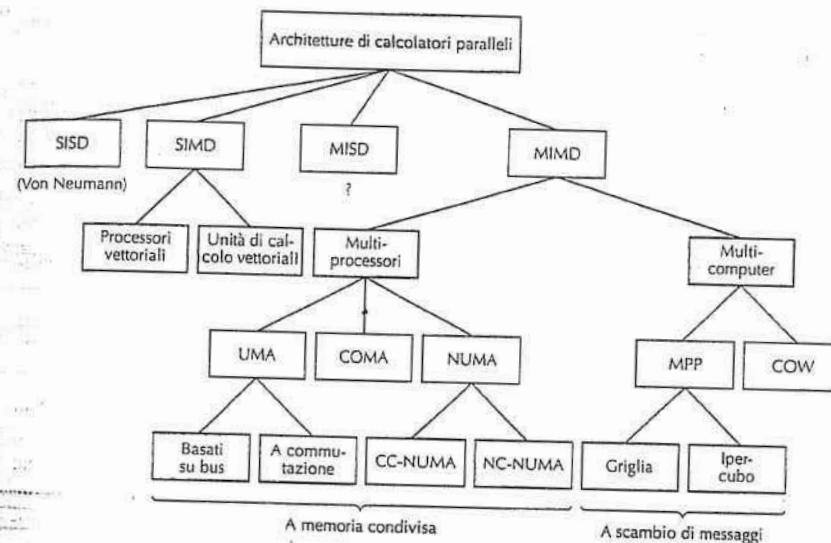


Figura 8.23 Tassonomia dei calcolatori paralleli.

Viceversa, nei multiprocessori NUMA questa proprietà non vale. Spesso c'è un modulo di memoria vicino a ogni CPU e il suo accesso è molto più veloce rispetto ai moduli distanti. Perciò, ai fini delle prestazioni, conta molto la collocazione dei dati e dei programmi. Anche le macchine COMA non sono uniformi, ma in un modo diverso. Ci occuperemo più tardi di queste categorie di macchine.

L'altra categoria principale delle macchine MIMD è costituita dai multicompiler che, a differenza dei multiprocessori, non hanno una memoria principale condivisa a livello di architettura. In altre parole, il sistema operativo di una CPU di un multicompiler non può accedere alla memoria di un'altra CPU tramite la sola esecuzione di un'istruzione LOAD. Deve invece spedire un messaggio e aspettare la relativa risposta. La capacità del sistema operativo di leggere parole a distanza effettuando una semplice LOAD è ciò che distingue i multiprocessori dai multicompiler. Abbiamo già detto che anche i programmi utente di un multicompiler possono essere resi in grado di accedere alle memorie distanti tramite istruzioni LOAD e STORE, però si tratta in questo caso di

un'illusione di cui si fa carico il sistema operativo e che non è sorretta dall'hardware. Si tratta di una differenza impercettibile, eppure molto importante. Per queste ragioni i multicompiler sono detti anche NORMA (*NO Remote Memory Access*).

È possibile individuare due categorie di multicompiler. La prima contiene processori di tipo MPP (*Massively Parallel Processor*), supercomputer molto costosi costituiti da molte CPU strettamente legate da una rete d'interconnessione ad alta velocità brevettata dal costruttore. L'IBM SP/3 ne è un esempio commerciale molto noto.

L'altra categoria raggruppa i PC consueti o le workstation, eventualmente montati in scaffali (*rack*), collegati tramite una tecnologia commerciale d'interconnessione pronta per l'uso. Dal punto di vista teorico non c'è molta differenza, ma i supercomputer giganti sono usati per scopi diversi rispetto alle reti di PC, assemblate dagli utenti a un costo che rappresenta una frazione molto piccola rispetto ai milioni di euro necessari all'acquisto di un MPP. Queste macchine fatte in casa vengono denominate in modi diversi, tra cui NOW (*Network of Workstations*), COW (*Cluster Of Workstations*) o semplicemente cluster.

### 8.3.2 Semantica della memoria

Anche se i multiprocessori mostrano la memoria alle CPU come un unico spazio condiviso degli indirizzi, spesso ci sono diversi moduli di memoria, ciascuno contenente una parte della memoria fisica. CPU e memorie sono collegate tramite una complessa rete d'interconnessione, come già illustrato nel Paragrafo 8.1.2. È possibile che diverse CPU tentino di leggere contemporaneamente la stessa parola di memoria, mentre altre CPU cercano di sovrascriverne il contenuto; ed è anche possibile che i messaggi di richiesta si sorpassino l'un l'altro durante il tragitto e che vengano quindi consegnati in un ordine diverso rispetto alla loro emissione. Si aggiunga poi il problema dell'esistenza di copie multiple di alcuni blocchi di memoria (per esempio nelle cache) e si capirà facilmente che è necessario adottare delle regole precise per prevenire il caos totale. In questo paragrafo vedremo che cosa voglia dire davvero condividere la memoria e come sia possibile agire nelle circostanze suddette.

Un modo per concepire la semantica della memoria è come contratto tra l'hardware della memoria e il software (Adve e Hill, 1990): se il software accetta di stare ad alcune regole, la memoria prende l'impegno di fornire certi risultati. L'analisi si sposta quindi sulla determinazione di queste regole, chiamate **modelli di consistenza**, per cui esistono molte proposte e altrettante implementazioni (Sorin et al., 2011).

Per chiarire la natura del problema, ipotizziamo che la CPU 0 scriva il valore 1 in una certa parola di memoria e che poco dopo la CPU 1 scriva il valore 2 nella stessa locazione. A questo punto la CPU 2 legge la parola e ottiene il valore 1. Che cosa dovrebbe fare il proprietario di quel calcolatore, riportarlo in negozio per una riparazione?

#### Consistenza stretta

Il modello più semplice è la **consistenza stretta** (*strict consistency*), secondo cui ogni lettura di una locazione  $x$  deve sempre restituire il valore della scrittura più recente. I programmati adorano questo modello, che però è pressoché impossibile da implementare se non con un solo modulo di memoria che serva tutte le richieste nell'ordine in cui

sono recapitate, senza effettuare caching né duplicazione di dati. Sfortunatamente questa implementazione renderebbe la memoria un collo di bottiglia per l'intero sistema e perciò non va presa in seria considerazione.

#### Consistenza sequenziale

Il secondo modello più promettente è la **consistenza sequenziale** (*sequential consistency*; Lamport, 1979). L'idea è la seguente: se ci sono più richieste di lettura e scrittura per una locazione, l'hardware effettua un rimescolamento (non deterministico) dell'ordine delle richieste e tutte le CPU vedono lo stesso ordine.

Esaminiamone il significato con un esempio. Si ipotizzi che la CPU 1 scriva il valore 100 nella parola  $x$  e 1 ns dopo la CPU 2 scriva il valore 200 nella stessa parola  $x$ . Quando è passato 1 ns dall'emissione della seconda scrittura (ma non necessariamente dal suo completamento) altre due CPU, la 3 e la 4, effettuano entrambe due letture di  $x$  in rapida successione, come mostrato dalla Figura 8.24(a). Le Figure 8.24(b)-(d) mostrano tre possibili ordinamenti dei sei eventi (due scritture e quattro letture): nella prima figura la CPU 3 ottiene (200, 200) e la CPU 4 ottiene (200, 200); nella Figura 8.24(c) ottengono rispettivamente (100, 200) e (200, 200); nella Figura 8.24(d) ottengono rispettivamente (100, 100) e (200, 100). Sono tutte combinazioni lecite, come lo sono le altre che non abbiamo elencato. Si noti che non esiste un singolo valore "corretto".

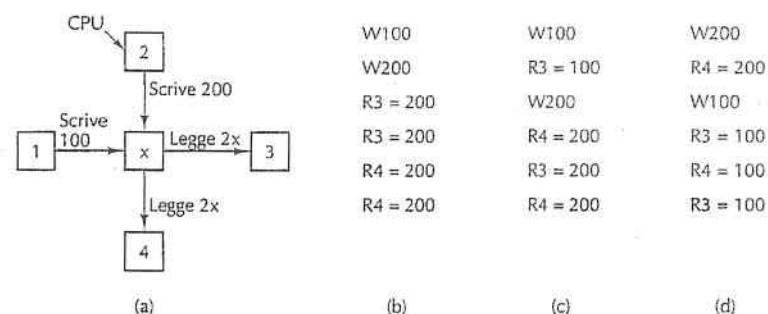


Figura 8.24 (a) Due CPU che scrivono una parola di memoria letta da altre due CPU. (b)-(d) Tre permutazioni dell'ordine temporale delle due scritture e delle quattro letture.

Comunque vada, è impossibile che la CPU 3 ottenga (100, 200) mentre la CPU 4 ottiene (200, 100), e questa è l'essenza stessa della consistenza sequenziale. Se ciò potesse capitare, vorrebbe dire che, secondo la CPU 3, la scrittura di 100 da parte della CPU 1 è stata completata prima della scrittura di 200 da parte della CPU 2, e questo va bene. Però vorrebbe dire anche che, secondo la CPU 4, la scrittura di 200 da parte della CPU 2 è stata completata prima della scrittura di 100 da parte della CPU 1. Anche questo risultato è possibile in sé, ma il fatto è che la consistenza sequenziale garantisce l'esistenza di un unico ordinamento globale delle scritture visibile a tutte le CPU. Se la CPU 3 osserva per prima la scrittura di 100, così deve essere anche per la CPU 4.

Nonostante la consistenza sequenziale sia una regola meno potente della consistenza stretta, si rivela comunque molto utile. Infatti stabilisce che, quando più eventi si verificano in modo concorrente, esiste un ordine secondo cui avvengono, determinato forse dalla temporizzazione o dal caso, ma si tratta pur sempre di un ordine che deve essere osservato da tutti i processori. Benché questa osservazione appaia ovvia, passiamo ora a modelli di consistenza che non garantiscono neanche questo.

#### Consistenza di processore

La consistenza di processore (*processor consistency*; Goodman, 1989) è un modello di consistenza meno rigido, ma più facile da implementare su multiprocessori grandi. Manifesta due proprietà:

1. le scritture effettuate da qualsiasi CPU sono percepite da tutte le altre CPU nello stesso ordine in cui sono state emesse;
2. per ogni parola di memoria, tutte le CPU vedono lo stesso ordine di scritture al suo interno.

La prima di queste due importanti proprietà afferma che, se la CPU 1 emette in sequenza le scritture di valori 1A, 1B e 1C in una locazione di memoria, tutti gli altri processori le vedono in quello stesso ordine. Ciò equivale a dire che, ogni processore che osservi il valore assunto da quella locazione di memoria mediante un ciclo di letture, non potrà mai ritenere il valore 1B scritto prima del valore 1A, e così via. La seconda proprietà serve affinché ogni parola di memoria contenga un valore non ambiguo al termine di numerose scritture da parte di più CPU. Tutti devono concordare su chi ha scritto per ultimo.

Pur con questi vincoli, al progettista resta ampio margine di scelta. Si consideri che cosa succede se la CPU 2 emette le scritture 2A, 2B e 2C concorrentemente alle tre scritture della CPU 1. Le altre CPU che stanno osservando la memoria mediante letture ripetute, vedranno un rimescolamento delle sei scritture, per esempio 1A, 1B, 2A, 2B, 1C, 2C oppure 2A, 1A, 2B, 2C, 1B, 1C o altri ancora. La consistenza di processore *non* garantisce che tutte le CPU vedano lo stesso ordinamento (a differenza della consistenza sequenziale). Perciò è del tutto legittimo per l'hardware comportarsi in questo modo e mostrare a ogni CPU un ordinamento potenzialmente diverso. Ciò che è garantito invece è che nessuna CPU veda una sequenza in cui 1B viene prima di 1A, e così via. L'ordine relativo delle scritture emesse da ogni singola CPU viene mantenuto comunque.

Va detto che alcuni autori definiscono la consistenza di processore in modo diverso e che non richiedono la seconda proprietà.

#### Consistenza debole

Il modello successivo, la **consistenza debole** (*weak consistency*), non garantisce nemmeno l'ordinamento delle scritture effettuate da una sola CPU (Dubois et al., 1986). In una memoria debolmente consistente una CPU potrebbe vedere 1A prima di 1B e un'altra potrebbe vedere l'esatto contrario. Tuttavia, per riportare un po' di ordine nel caos, le memorie debolmente consistenti sono dotate di variabili e di operazioni di sincronizzazione. Quando viene eseguita una sincronizzazione, vengono completate tutte le scritture pendenti e non è possibile effettuarne di nuove finché non termina la sincroniz-

zazione. La sincronizzazione effettua un “flush della pipeline” (scarica le operazioni in coda) e conduce la memoria a uno stato stabile, senza operazioni pendenti. Le operazioni di sincronizzazione sono esse stesse sequenzialmente consistenti, cioè anche se vengono emesse da più CPU, tutti i processori percepiscono lo stesso ordine globale della loro esecuzione.

In questo modello di consistenza, il tempo è diviso in epochhe ben definite e delimitate dalle sincronizzazioni (sequenzialmente consistenti), come illustrato dalla Figura 8.25. Non è garantito l'ordine relativo tra 1A e 1B, cioè CPU diverse potrebbero osservare le due scritture in ordine diverso (una CPU potrebbe vedere 1A, 1B, un'altra potrebbe vedere 1B, 1A). Però tutte le CPU devono vedere 1B prima di 1C, perché l'operazione di sincronizzazione forza il completamento di 1A, 1B e 2A prima che 1C, 2B, 3A o 3B possano cominciare. La sincronizzazione è un modo con cui il software può imporre un certo ordinamento nella sequenza di eventi, ma al prezzo di un flush della pipeline di memoria, che richiede un certo tempo e rallenta dunque, in una certa misura, la macchina. Effettuare spesso questa operazione può diventare un problema.

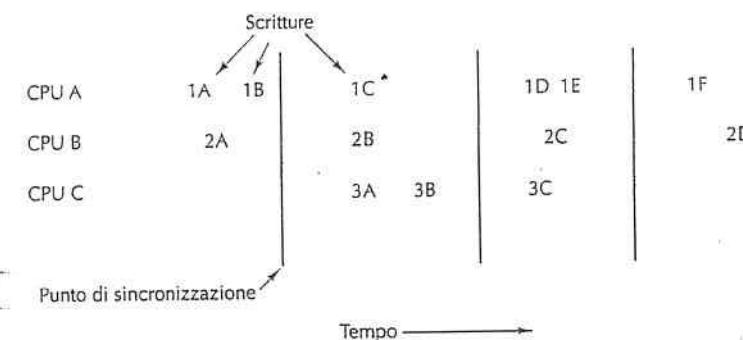


Figura 8.25 La memoria a consistenza debole utilizza la sincronizzazione per suddividere il tempo in una successione di epochhe.

#### Consistenza dopo rilascio

La consistenza debole è abbastanza inefficiente perché comporta il completamento di tutte le operazioni di memoria pendenti e il trattamento di tutte le operazioni nuove finché non vengano terminate le precedenti. La **consistenza dopo rilascio** (*release consistency*) costituisce un miglioramento e adotta un modello simile alle sezioni critiche (Gharachorloo et al., 1990). L'idea alla base di questo modello è la seguente: quando un processo esce da una regione critica, non è necessario richiedere il completamento immediato di tutte le scritture. Basta solo assicurarsi che vengano effettuate prima del reingresso di un altro processo nella regione critica.

Secondo questo modello, l'operazione di sincronizzazione fornita dalla consistenza debole viene suddivisa in due operazioni differenti. Per leggere o scrivere una variabile condivisa, la CPU (cioè il suo software) deve prima eseguire un'operazione *acquire*

sulla variabile di sincronizzazione per acquisire l'accesso esclusivo ai dati condivisi. Quindi la CPU può usarli come vuole, sia in lettura sia in scrittura. Al termine della sua attività, la CPU invoca l'operazione *release* sulla variabile di sincronizzazione per indicare il suo rilascio. La *release* non forza il completamento di tutte le scritture pendenti, bensì attende per il proprio completamento che siano avvenute tutte le scritture emesse precedentemente. Inoltre non ritarda l'avvio di nuove operazioni di memoria.

All'emissione di una *acquire*, per prima cosa si verifica se tutte le *release* precedenti sono state completate. Se così non è, la *acquire* resta sospesa fino al loro termine (dunque finché non sono state completate tutte le scritture precedenti). Così facendo, se una *acquire* è abbastanza lontana dalla *release* precedente, non deve aspettare alcunché prima di entrare nella regione critica. Se invece avviene troppo presto, dovrà essere ritardata fino al completamento di tutte le *release* pendenti, per garantire il corretto aggiornamento delle variabili della sezione critica. Si tratta di uno schema leggermente più complesso della consistenza debole, ma presenta il vantaggio significativo di non ritardare le istruzioni così spesso, pur mantenendo la consistenza.

La consistenza di memoria non è un capitolo chiuso, la ricerca di nuovi modelli va avanti (Naeem et al., 2011; Sorin et al., 2011; Tu et al., 2010).

### 8.3.3 Architetture di multiprocessori simmetrici UMA

I multiprocessori più semplici si basano su un solo bus, come illustrato nella Figura 8.26(a). Ci sono due o più CPU, uno o più moduli di memoria, e tutti usano lo stesso bus per la comunicazione. 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.

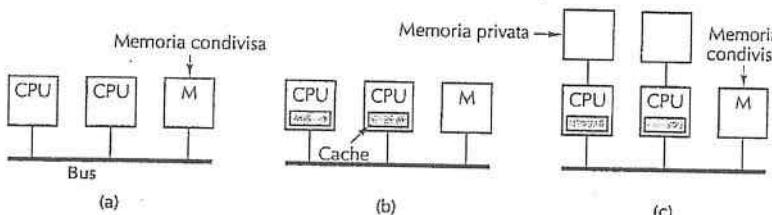


Figura 8.26 Tre multiprocessori basati su bus. (a) Senza caching. (b) Con caching. (c) Con caching e memorie private.

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 32 o 64 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 8.26(b). La cache può trovarsi all'interno del chip della CPU, vicino alla CPU, sulla scheda del processore o in una posizione intermedia. 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. Il caching risulta qui una scelta vincente. Tuttavia, come vedremo tra poco, non è semplice mantenere la coerenza delle cache.

Una possibilità alternativa è il progetto della Figura 8.26(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). Per sfruttare questa configurazione in modo ottimale, il compilatore dovrebbe usare la memoria privata per contenere i sorgenti del programma, le stringhe, le costanti e gli altri dati in sola lettura, gli stack e le variabili locali. La memoria condivisa verrebbe usata solo per la scrittura delle variabili condivise. Questa collocazione riduce in molti casi il traffico sul bus, ma richiede la cooperazione attiva da parte del compilatore.

#### Cache snooping

Il ragionamento fatto circa le prestazioni è corretto, ma abbiamo liquidato troppo velocemente una questione fondamentale. Se la memoria ha consistenza sequenziale, che cosa succede nel caso in cui la CPU 1 ha nella propria cache una linea e la CPU 2 cerca di leggere una parola dalla stessa linea di cache? In assenza di regole speciali, quest'ultima non farebbe altro che procurarsi una copia della linea da inserire nella propria cache. In teoria è ammissibile che una linea sia presente in due cache diverse. Se poi la CPU 1 modifica la linea e, immediatamente dopo, la CPU 2 legge la propria copia, ma si troverà a manipolare dati scaduti (*stale data*), violando così il contratto tra software e memoria. Il programma in esecuzione sulla CPU 2 viene danneggiato da questo comportamento.

A tale proposito la letteratura specializzata parla del **problema della coerenza o consistenza della cache**. Si tratta di un problema molto grave che, se non risolto, preclude l'uso del caching e i multiprocessori orientati al bus sarebbero così limitati a due o tre CPU. In ragione dell'importanza della questione, sono state proposte negli anni molte soluzioni (per esempio Goodman, 1983; Papamarcos e Patel, 1984) che definiscono un **protocollo di coerenza delle cache** e, sebbene differiscano nei dettagli, impediscono la presenza simultanea di versioni discordanti di una stessa linea all'interno di due o più cache. L'unità di trasferimento e di memorizzazione di una cache si dice *linea di cache* ed è solitamente di 32 o 64 byte.

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 *ficciano il naso* nel bus. Il protocollo di coerenza delle cache è composto dall'insieme di regole osservate dalle cache, dalla CPU e dalla memoria per prevenire la presenza di versioni diverse dei dati all'interno di varie cache.

Il più semplice protocollo di coerenza delle cache si chiama *write through* e può essere capito facilmente attraverso le quattro situazioni elencate nella Figura 8.27. Quando una CPU tenta di leggere una parola che non si trova nella sua cache, evento

detto *read miss*, il controllore carica nella cache la linea contenente quella parola. La linea proviene dalla memoria che, secondo questo protocollo, viene mantenuta costantemente aggiornata. Le successive richieste di lettura possono essere soddisfatte dalla cache (*read hit*).

In seguito a un fallimento in scrittura, evento detto *write miss*, la parola modificata viene scritta in memoria principale, mentre la linea contenente la parola *non* viene caricata nella cache. In caso di successo in scrittura (*write hit*) viene aggiornata la cache e inoltre la parola viene scritta direttamente in memoria principale. La caratteristica più importante di questo protocollo è che tutte le operazioni di scrittura agiscono sulla memoria, che risulta perciò sempre aggiornata.

Riconsideriamo tali azioni, questa volta dal punto di vista di una *snooping cache* (colonna più a destra nella Figura 8.27). Chiamiamo cache 1 la cache che effettua le azioni e cache 2 la cache “ficciaso”. Dopo un *read miss*, la cache 1 effettua una richiesta di caricamento di una linea dalla memoria; la cache 2 osserva questa richiesta senza intervenire. Dopo un *read hit* la richiesta viene soddisfatta localmente, perciò non viene inviata alcuna richiesta sul bus, quindi la cache 2 non sa della lettura della cache 1.

Azione	Richiesta locale	Richiesta a distanza
Read miss	Preleva dati in memoria	Nessuna
Read hit	Usa dati dalla cache locale	"
Write miss	Aggiorna i dati in memoria	"
Write hit	Aggiorna la cache e la memoria	Invalida l'elemento di cache

Figura 8.27 Protocollo write-through di coerenza delle cache.

Le situazioni in scrittura sono più interessanti. Se la CPU 1 effettua una scrittura, la cache 1 invia una richiesta di scrittura sul bus, sia in caso di miss sia di hit. A ogni scrittura, la cache 2 verifica se possiede la parola che deve essere scritta. In caso negativo, dal suo punto di vista si tratta di una semplice richiesta a distanza che provoca un *write miss*, perciò non intraprende alcuna azione (un fallimento a distanza nella Figura 8.27 significa che la parola non è presente nella snooping cache, indipendentemente dalla sua eventuale presenza nella cache 1; una richiesta può causare localmente un hit nella prima cache e un miss nella snooping cache, o viceversa).

Se invece la cache 1 scrive una parola presente nella cache 2 (richiesta a distanza che ha successo), quest'ultima deve intraprendere qualche azione per evitare di avere dati scaduti, perciò contrassegna come non valido l'elemento di cache contenente la parola modificata. Ciò equivale a rimuovere l'elemento dalla cache. Poiché tutte le cache monitorano tutte le richieste di bus, ogni scrittura di una parola provoca l'aggiornamento della cache del richiedente e della memoria, la rimozione dei dati scaduti dalle altre cache, e così si previene l'insorgenza di versioni incoerenti.

Naturalmente è possibile che la CPU della cache 2 voglia leggere quella stessa parola al ciclo successivo. In tal caso, la cache 2 leggerebbe la parola dalla memoria, che è perfettamente aggiornata. Dopo la lettura la cache 1, la cache 2 e la memoria avrebbero

tutte una copia fedele del dato. Se una delle due CPU effettuasse ora una scrittura, l'elemento di cache dell'altra CPU verrebbe rimosso e la memoria aggiornata.

Sono possibili molte variazioni di questo semplice protocollo. Per esempio, una snooping cache potrebbe evitare di invalidare un proprio elemento a seguito di un *write hit*, ma anzi, potrebbe aggiornarlo con il nuovo valore. Dal punto di vista concettuale questo comportamento è equivalente a un'invalidazione seguita da un caricamento da memoria principale. In tutti i protocolli di cache bisogna scegliere se adottare una strategia di aggiornamento o una d'invalidazione.

La scelta produce protocolli che esibiscono prestazioni differenti a seconda dei carichi di lavoro. I messaggi di aggiornamento contengono le informazioni da adeguare e sono perciò più grandi di quelli d'invalidazione, ma prevengono miss futuri della cache.

Un'altra variante prevede di effettuare un caricamento nella snooping cache anche in caso di *write miss*. La correttezza dell'algoritmo non è compromessa dal caricamento, ma le prestazioni lo sono. La domanda è: “Qual è la probabilità che una parola appena scritta venga riscritta presto?” Se è alta, allora ci sono argomenti a favore del caricamento nella cache dopo un *write miss*, e si parla di una politica *write-allocate*. Se invece la probabilità è bassa, è preferibile non aggiornare dopo un *write miss*: se la parola viene letta poco dopo, verrà caricata comunque a seguito di un *read miss*, perciò si perde poco a non caricarla dopo un *write miss*.

Così come gran parte delle soluzioni semplici, anche questa è inefficiente. Ogni operazione di scrittura raggiunge la memoria tramite il bus, che quindi risulterà un collo di bottiglia anche con un numero modesto di CPU. Per contenere l'aumento del traffico sul bus sono stati ideati altri protocolli di cache, che hanno in comune la proprietà che non tutte le scritture raggiungono direttamente la memoria. Quando viene modificata una linea di cache, viene asserito al suo interno un bit che annota la sua validità in contrapposizione alla memoria. La linea così modificata dovrà essere scritta prima o poi in memoria, ma ciò si potrebbe verificare dopo molte scritture al suo interno. Questi tipi di protocollo si chiamano protocolli *write-back*.

#### Protocollo MESI di coerenza delle cache

MESI è un protocollo di coerenza delle cache molto diffuso e di tipo *write-back* (Papamarcos e Patel, 1984). Si chiama così perché si avvale degli stati *Modified*, *Exclusive*, *Shared* e *Invalid*, ed è basato su un precedente protocollo detto *write-once* (Goodman, 1983). Il protocollo MESI è utilizzato dal Pentium 4 e da molte altre CPU per lo snooping sul bus. I quattro stati in cui si possono trovare gli elementi della cache sono i seguenti.

1. Non valido: l'elemento di cache non contiene dati validi.
2. Condiviso: la linea potrebbe essere contenuta in più di una cache; la memoria è aggiornata.
3. Esclusivo: nessun'altra cache contiene la linea; la memoria è aggiornata.
4. Modificato: l'elemento è valido; la memoria non lo è; non ci sono altre copie della linea.

Quando una CPU viene avviata, tutti gli elementi di cache sono contrassegnati come non validi. Alla prima lettura la linea referenziata viene prelevata dalla memoria e inserita

nella cache della CPU; il suo stato è esclusivo, perché è la sola copia presente in una cache, come illustrato nella Figura 8.28(a) con riferimento alla CPU 1 dopo la lettura della linea A.

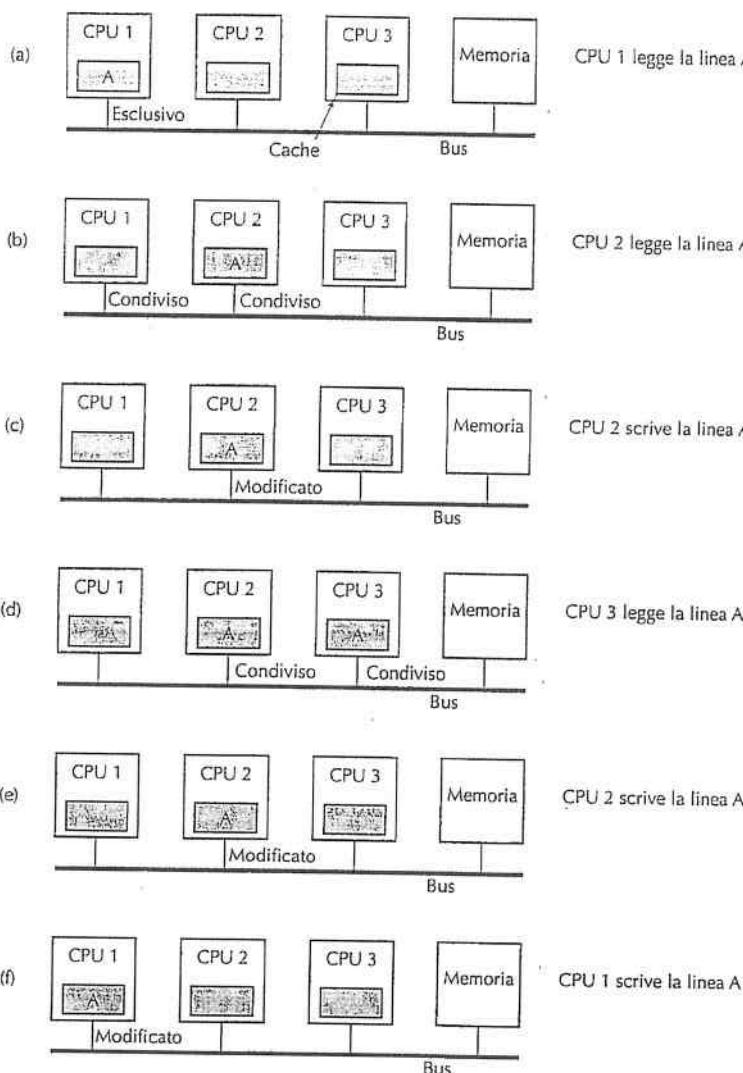


Figura 8.28 Protocollo MESI di coerenza delle cache.

Le letture successive a opera di quella CPU usano l'elemento della cache e non transitano sul bus. È possibile che un'altra CPU prelevi la stessa linea di cache, ma grazie allo snooping la CPU 1 sa di non essere più la sola detentrice e annuncia sul bus di possedere una copia della linea. Entrambe le copie vengono contrassegnate come S (*Shared*), come mostrato nella Figura 8.28(b). Detto altrimenti, lo stato S significa che la linea si trova in una o più cache per la lettura e che la memoria è aggiornata. Le letture di una CPU all'interno di una linea nello stato S non si avvalgono del bus e non ne modificano lo stato.

Si consideri ora che cosa succede se la CPU 2 scrive in una sua linea di cache che si trova nello stato S. La CPU emette un segnale d'invalidazione sul bus, indicando alle altre CPU di eliminare le proprie copie. La copia passa ora allo stato M, come mostrato nella Figura 8.28(c). La linea non viene scritta in memoria. Val la pena notare che se una linea si trova nello stato E al momento di una scrittura, non ci vuole alcun segnale d'invalidazione a beneficio delle altre cache, perché è chiaro che non esistono altre copie.

Si consideri poi che cosa succede quando la CPU 3 legge la linea. La CPU 2 detiene ora la linea e sa che la copia in memoria non è valida, perciò invia sul bus un segnale che indica alla CPU 3 di attendere mentre la linea viene scritta in memoria. Al completamento dell'operazione, la CPU 3 preleva la sua copia e la linea viene contrassegnata come condivisa da entrambe le cache, come mostrato dalla Figura 8.28(d). Dopo di ciò, la CPU 2 scrive ancora nella linea, il che invalida la copia della CPU 3 (Figura 8.28(e)).

Infine anche la CPU 1 scrive una parola nella linea. La CPU 2 osserva il tentativo di scrittura e invia un segnale sul bus che richiede alla CPU 1 di attendere la scrittura della linea in memoria. Al completamento dell'operazione contrassegna la propria linea come non valida, perché sa che un'altra CPU sta per modificarla. A questo punto ci possiamo trovare nella situazione in cui una CPU sta scrivendo all'interno di una linea fuori dalla cache. Se si usa la politica write-allocate, la linea viene caricata nella cache e contrassegnata come modificata, come succede nella Figura 8.28(f). Altrimenti, la scrittura avviene direttamente in memoria e la linea non viene caricata in alcuna cache.

#### 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 d'interconnessione. Il circuito più semplice che collega  $n$  CPU a  $k$  memorie è il **commutatore crossbar** ("a traversa") mostrato nella Figura 8.28. I commutatori crossbar sono stati usati per decenni all'interno delle cabine di commutazione telefoniche per poter collegare in qualsiasi modo un certo numero di linee in ingresso a un insieme di linee in uscita. A ogni intersezione tra una linea orizzontale (in ingresso) e una verticale (in uscita) 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 8.29(a) mostra tre crosspoint chiusi contemporaneamente, consentendo così la comunicazione simultanea tra le seguenti coppie (CPU, memoria): (001, 000), (101, 101) e (110, 010). Sono possibili molte altre combinazioni; infatti il numero di combinazioni è uguale ai diversi modi in cui si possono sistemare otto torri su di una scacchiera senza che si attaccino<sup>4</sup>.

4 Si tratta del famoso problema delle Non-Attacking Rooks (N.d.R.).

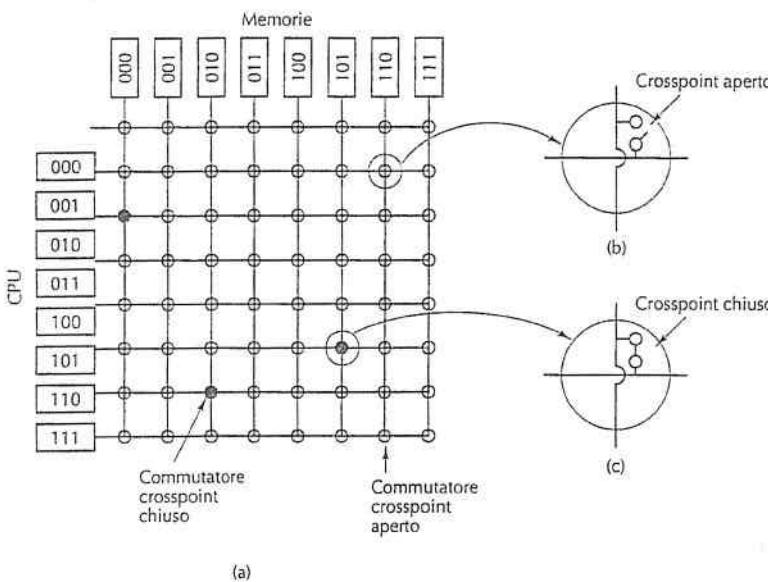


Figura 8.29 (a) Commutatore crossbar  $8 \times 8$ . (b) Crosspoint aperto. (c) Crosspoint chiuso.

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 (ipotizzando che il modulo di memoria desiderato sia disponibile). 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. Vedremo più avanti alcuni schemi d'interconnessione che non hanno queste proprietà.

Uno dei difetti di questi crossbar è che il numero di crosspoint cresce come  $n^2$ . Un progetto può pensare di adottare un commutatore crossbar se il sistema in questione è di medie dimensioni. Più avanti nel capitolo incontreremo un progetto di questo tipo: il Sun Fire E25K. D'altra parte, se ci sono 1000 CPU e 1000 moduli di memoria ci vorrebbero un milione di crosspoint, il che non è fattibile in pratica. In una situazione del genere c'è bisogno di qualcosa di molto diverso.

#### Multiprocessori UMA con reti a commutazione multilivello

Quel "qualcosa di molto diverso" può essere l'umile commutatore  $2 \times 2$  della Figura 8.30(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. Ipotizziamo per comodità che i messaggi siano composti di quattro parti, come schematizzato nella Figura 8.30(b). *Modulo* specifica la memoria da usare. *Indirizzo* specifica un indirizzo all'interno di quel modulo. *Opcode* stabilisce il tipo di operazione, per esempio

*READ* o *WRITE*. Infine il campo opzionale *Valore* potrebbe contenere un operando, come una parola di 32 bit da scrivere con una *WRITE*. Il commutatore esamina il campo *Modulo* e lo usa per determinare la linea X o Y lungo cui inviare il massaggio.

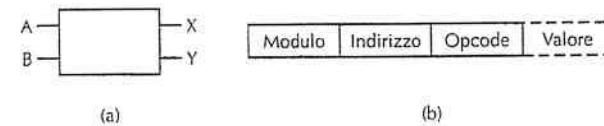


Figura 8.30 (a) Commutatore  $2 \times 2$ . (b) Formato di messaggio.

I nostri commutatori  $2 \times 2$  possono essere composti in molti modi per dar vita a una più grande rete a commutazione multilivello. Una possibilità economica e senza fronzoli è la rete omega, illustrata nella Figura 8.31. 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  $\log_2 n$  livelli di  $n/2$  commutatori ciascuno, per un totale di  $(n/2) \log_2 n$  commutatori: è un numero molto inferiore a  $n^2$ , specie per valori grandi di  $n$ .

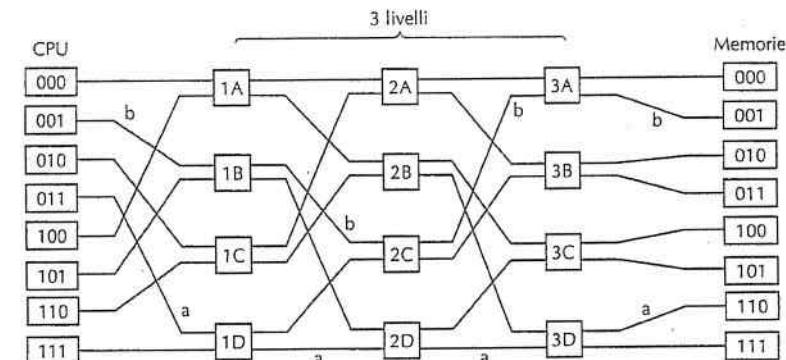


Figura 8.31 Rete di connessione omega.

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 8.31.

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; a differenza del commutatore crossbar la rete omega è una rete bloccante. Non è possibile elaborare simultaneamente qualsiasi insieme di richieste e i conflitti possono verificarsi per l'uso di un collegamento o di un commutatore sia tra richieste che *vanno* alla memoria, sia tra quelle che *provengono* da essa.

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. Si consideri per esempio uno spazio degli indirizzi orientato al byte in un calcolatore che accede prevalentemente a parole di 32 bit. I due bit meno significativi si troveranno spesso a 00, ma i 3 bit successivi saranno distribuiti uniformemente. Se si usano questi 3 bit come numero di modulo, le parole indirizzate in modo consecutivo si troveranno in moduli consecutivi. 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.

### 8.3.4 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. Dunque tutti i programmi UMA continuano a funzionare sulle macchine NUMA, ma le prestazioni degradano molto rispetto alle macchine UMA che hanno la stessa frequenza di clock.

Tutte le macchine NUMA hanno sempre tre caratteristiche che insieme le distinguono dagli altri multiprocessori.

1. C'è un solo spazio degli indirizzi visibile a tutte le CPU.
2. L'accesso alla memoria distante si effettua tramite istruzioni LOAD e STORE.
3. L'accesso alla memoria distante è più lento di quello alla memoria locale.

Quando il tempo di accesso alla memoria distante non è nascosto (perché non c'è caching) si parla di sistemi NC-NUMA. Viceversa, quando sono presenti cache coerenti si parla di CC-NUMA (almeno tra gli esperti di hardware). Gli esperti di software spesso usano il termine DSM hardware perché si tratta fondamentalmente di una memoria condivisa distribuita, ma questa volta implementata in hardware con una dimensione di pagina piccola.

Una macchina NC-NUMA ante litteram fu il Carnegie-Mellon Cm\*, illustrato in forma semplificata nella Figura 8.32 (Swan et al., 1977). Cm\* conteneva varie CPU LSI-11, ciascuna dotata di una piccola memoria cui accedeva tramite un bus locale (il processore LSI-11 era una versione a singolo chip del DEC PDP-11, un minicomputer molto diffuso negli anni '70). Inoltre, i chip LSI-11 erano collegati 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. Anche se un programma poteva girare senza problemi usando solo la memoria distante, impiegava un tempo circa 10 volte maggiore rispetto alla sua esecuzione basata sulla sola memoria locale.

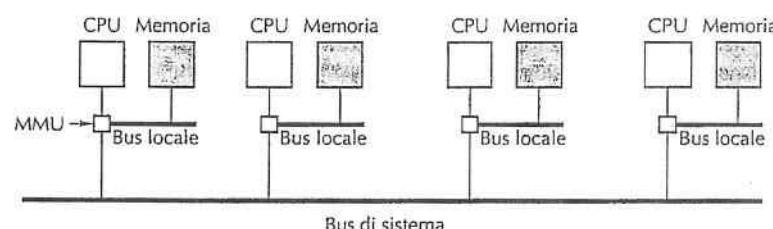


Figura 8.32 Macchina NUMA con due livelli di bus. Il Cm\* fu il primo multiprocessore ad adottare questo schema.

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. In ragione di ciò, le macchine NC-NUMA usano un software molto elaborato per spostare le pagine all'interno del sistema, in modo da garantire le massime prestazioni.

Generalmente esiste un processo demone chiamato **scanner delle pagine**, eseguito a intervalli di pochi secondi. Il suo compito è esaminare le statistiche di utilizzo e spostare le pagine nel tentativo d'incrementare le prestazioni. Se una pagina sembra posizionata male, lo scanner delle pagine la rimuove dalla mappa delle pagine e così l'accesso successivo a essa causerà un errore di pagina. L'errore di pagina è l'occasione per decidere dove "collocare" la pagina, probabilmente in una memoria diversa da quella in cui si trovava prima. Per evitare il *thrashing* si stabilisce la regola secondo cui, una volta caricata, una pagina viene lasciata al proprio posto per almeno un dato numero di secondi. Sono stati studiati molti algoritmi per la paginazione nelle macchine NC-NUMA, ma nessuno di loro si comporta al meglio in ogni circostanza (LaRowe ed Ellis, 1991). Le prestazioni dipendono dall'applicazione.

#### Multiprocessori NUMA con cache coerente

I progetti di multiprocessori come quello della Figura 8.32 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. Si tratta di una soluzione facile da implementare dal punto di vista tecnico, ma che ben presto si rivela praticamente inattuabile al crescere del numero di CPU. La costruzione di multiprocessori veramente grandi richiede un approccio completamente diverso.

Attualmente il modo più diffuso per costruire grandi CC-NUMA (*Cache Coherent NUMA Multiprocessor*, "multiprocessori NUMA con coerenza della cache") è il sistema **multiprocessore basato su directory**. 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.

Per meglio circostanziare l'idea alla base di questi multiprocessori, consideriamo un semplice esempio di un sistema (ipotetico) di 256 nodi, ciascuno costituito da una CPU e da una RAM di 16 MB, collegate tramite un bus locale. La memoria totale è di  $2^{32}$  byte, divisi in  $2^{26}$  linee di cache di 64 byte ciascuna. La memoria è allocata staticamente ai diversi nodi, laddove i primi 16 MB sono attribuiti al nodo 0, le locazioni da 16 a 32 MB al nodo 1, e così via. Si dice che il nodo 0 è il nodo di appartenenza (*home node*) dei blocchi di memoria dei primi 16 MB e delle corrispondenti linee di cache. I nodi

sono collegati per mezzo della rete d'interconnessione rappresentata nella Figura 8.33(a). La rete d'interconnessione potrebbe essere una griglia, un ipercubo o avere un'altra topologia. Ogni nodo contiene anche gli elementi di directory per le  $2^{18}$  linee di cache da 64 byte che costituiscono i suoi  $2^{24}$  byte di memoria. Ipotizziamo per il momento che ogni linea si possa trovare al massimo in una cache.

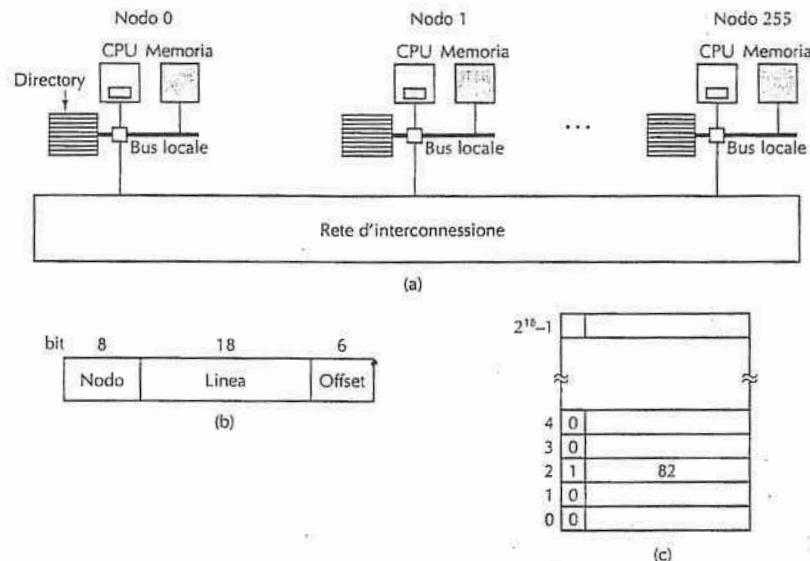


Figura 8.33 (a) Multiprocessore di 256 nodi basato su directory. (b) Divisione in campi di un indirizzo di memoria di 32 bit. (c) Directory in corrispondenza del nodo 36.

Per comprendere il funzionamento della directory seguiamo un'istruzione LOAD della CPU 20, che fa riferimento a una certa linea di cache. All'inizio la CPU emette l'istruzione e la passa alla propria MMU, che la traduce in un indirizzo fisico, diciamo 0x24000108. La MMU divide l'indirizzo nelle tre parti mostrate nella Figura 8.33(b). In decimale, le tre parti valgono: nodo 36, linea 4, offset 8. La MMU si accorge che il riferimento è a una parola di memoria del nodo 36, non del nodo 20, perciò invia un messaggio di richiesta attraverso la rete d'interconnessione verso il nodo che ospita la linea, cioè il 36, e gli richiede se abbia in cache la linea 4 e, in caso affermativo, dove si trovi.

La richiesta, pervenuta al nodo 36 dalla rete d'interconnessione, viene instradata all'hardware della directory, che la usa come indice per la sua tabella di  $2^{18}$  elementi (uno per ogni linea di cache) per estrarre l'elemento 4. Nella Figura 8.33(c) osserviamo che la linea non è presente in cache, perciò l'hardware preleva la linea 4 dalla RAM

valgono: nodo 36, linea 4, offset 8. La MMU si accorge che il riferimento è a una parola di memoria del nodo 36, non del nodo 20, perciò invia un messaggio di richiesta attraverso la rete d'interconnessione verso il nodo che ospita la linea, cioè il 36, e gli richiede se abbia in cache la linea 4 e, in caso affermativo, dove si trovi.

La richiesta, pervenuta al nodo 36 dalla rete d'interconnessione, viene instradata all'hardware della directory, che la usa come indice per la sua tabella di  $2^{18}$  elementi (uno per ogni linea di cache) per estrarre l'elemento 4. Nella Figura 8.31(c) osserviamo che la linea non è presente in cache, perciò l'hardware preleva la linea 4 dalla RAM locale, la spedisce al nodo 20 e aggiorna l'elemento 4 della directory a indicare che la linea è adesso presente nella cache del nodo 20.

Prendiamo ora in considerazione una seconda richiesta, questa volta riguardante la linea 2 del nodo 36. La Figura 8.31(c) evidenzia che la linea è presente nella cache presso il nodo 82. A questo punto l'hardware potrebbe aggiornare l'elemento 2 della directory per specificare che la linea si trova ora nel nodo 20 e inviare quindi un messaggio al nodo 82 per ordinargli di consegnare la linea al nodo 20 e di invalidare la propria cache. Si noti che anche un cosiddetto "multiprocessore a memoria condivisa" nasconde una certa mole di scambi di messaggi.

Prendiamoci un attimo per calcolare lo spazio di memoria richiesto dalle directory. Ogni nodo ha 16 MB di RAM e  $2^{18}$  elementi di 9 bit per mantenere traccia della RAM. Perciò l'informazione accessoria richiesta dalla directory è di circa  $9 \times 2^{18}$  bit diviso 16 MB, cioè circa l'1,76%; è una quantità abbastanza accettabile (anche se si tratta di memoria ad alta velocità, il che ne incrementa i costi). Se le linee di cache fossero di 32 byte, la percentuale salirebbe soltanto al 4%. Con linee di cache di 128 byte sarebbe invece al di sotto dell'1%.

La restrizione evidente imposta da questo progetto è che una linea si trovi per lo più nella cache di un solo nodo. Per permettere il caching delle linee in più nodi si dovrebbe trovare un modo per localizzarle tutte, in modo da poterle invalidare o aggiornare dopo una scrittura. Per permettere il caching simultaneo all'interno di più nodi sono possibili diverse opzioni.

Una possibilità è dotare ogni elemento di directory di  $k$  campi atti a individuare altri nodi, consentendo così il caching di una linea in un massimo di  $k$  nodi. Una seconda possibilità è la sostituzione del numero di nodo del nostro semplice progetto con una bit map, in cui a ogni bit corrisponde un nodo. Secondo questa opzione non c'è alcun limite al numero di copie di una linea, ma c'è un sostanziale incremento del volume d'informazioni accessorie. Una directory con 256 bit per ogni linea di cache di 64 byte (512 bit) comporta una percentuale d'informazioni accessorie superiore al 50%. Una terza possibilità è di gestire un campo da 8 bit in ogni elemento di directory e usarlo come puntatore a una lista concatenata contenente tutte le copie della linea di cache corrispondente. Questa strategia richiede sia spazio accessorio per i puntatori alle liste, sia tempo di esecuzione per scorrere le liste quando è necessario individuare tutte le copie di una linea. Ciascuna possibilità presenta vantaggi e svantaggi, e tutte e tre sono state usate su sistemi reali.

Un altro miglioramento che si può apportare al progetto di directory è la capacità di ricordare lo stato di un linea: "pulita" o "sporca", a seconda che la memoria di appartenenza sia o meno aggiornata. Se giunge una richiesta per una linea di cache pulita, il nodo di appartenenza può soddisfare la richiesta direttamente dalla memoria, senza dover accedere alla cache. Tuttavia, una richiesta di lettura di una linea di cache sporca deve essere inoltrata al nodo contenente la linea di cache, perché è il solo ad averne una copia valida. Se è consentita una sola copia per ogni linea di cache, come nella Figura 8.31, non si ottiene alcun mi-

glioramento reale a mantenere traccia dello stato delle linee, perché ogni nuova richiesta porta l'invio di un messaggio alla copia esistente per invalidarla.

Naturalmente se si tiene traccia dello stato di una linea bisogna anche informare il suo nodo di appartenenza ogni volta che viene modificata, anche se ne esiste una sola copia. Se invece esistono molteplici copie, la modifica di una di loro richiede l'invalidazione di tutte le altre, perciò si rende necessario un protocollo per evitare le corse critiche. Per esempio, per modificare una linea di cache condivisa, uno dei suoi detentori potrebbe dover richiedere l'accesso esclusivo *prima* della modifica. Una richiesta di questo tipo causerebbe l'invalidazione di tutte le altre copie prima di permettere l'accesso. Altre ottimizzazioni per le prestazioni delle macchine CC-NUMA sono trattate in (Stenstrom et al., 1997).

### Il multiprocessore NUMA Sun Fire E25K

Studiamo ora la famiglia di processori Sun Fire di Sun Microsystems come esempio di multiprocessori NUMA a memoria condivisa. Benché questa famiglia raggruppi vari modelli, focalizzeremo la nostra attenzione sul modello E25K, equipaggiato con 72 CPU UltraSPARC IV. Un UltraSPARC IV è dotato di due processori UltraSPARC III che condividono una cache e una memoria. Il modello E15K è del tutto analogo all'E25K, se non che è costruito a partire da processori singoli invece che da chip a due processori. Esistono anche modelli più piccoli di questi, ma dal nostro punto di vista ci interessiamo proprio al funzionamento di quelli contenenti il maggior numero di CPU.

Il sistema E25K è formato da 18 insiemi di schede: ogni insieme comprende una scheda di CPU e memoria, una scheda di I/O con quattro alloggiamenti PCI e una scheda di espansione che serve a collegare le due schede precedenti al piano centrale, che sorregge tutte le schede e contiene i circuiti per la commutazione. Ogni scheda di CPU e memoria contiene quattro chip di CPU e quattro moduli di RAM da 8 GB. Dunque ogni scheda di CPU e memoria dell'E25K contiene otto CPU (quattro coppie) e 32 GB di RAM (quattro CPU e 32 GB di RAM nell'E15K). Un E25K completo (illustrato nella Figura 8.32) contiene perciò 144 CPU, 576 GB di RAM, 72 alloggiamenti PCI. È interessante sapere che il numero 18 è frutto di vincoli di imballaggio: il sistema con 18 insiemi di schede si è rivelato il più grande sistema che potesse attraversare i vani delle porte senza essere smontato. Laddove i programmati pensano solo in funzione di 0 e 1, gli ingegneri devono preoccuparsi anche di altro, per esempio del fatto che i clienti riescano a trasportare il loro prodotto all'interno degli edifici passando attraverso le porte.

Il piano centrale si compone di un insieme di tre commutatori crossbar  $18 \times 18$  per la connessione dei 18 gruppi di schede. Un crossbar serve per le linee di indirizzi, uno per le risposte e uno per il trasferimento dati. Oltre alle 18 schede di espansione, il piano centrale è collegato anche a un insieme di schede di controllo del sistema contenente una sola CPU, ma che ha interfacce per CD-ROM, periferiche a nastro, linee seriali e altri dispositivi di periferica necessari al riavvio, alla manutenzione e al controllo del sistema.

Il cuore di ogni multiprocessore è il sottosistema di memoria. Come si possono connettere 144 CPU alla memoria distribuita? I modi diretti, un grosso snooping bus condiviso o un commutatore crossbar  $144 \times 72$ , non funzionano bene. Il primo non conviene perché il bus diventa un collo di bottiglia, il secondo perché il commutatore risulta troppo difficile e costoso da costruire. Per queste ragioni i multiprocessori come l'E25K sono costretti a usare un sottosistema di memoria più complesso.

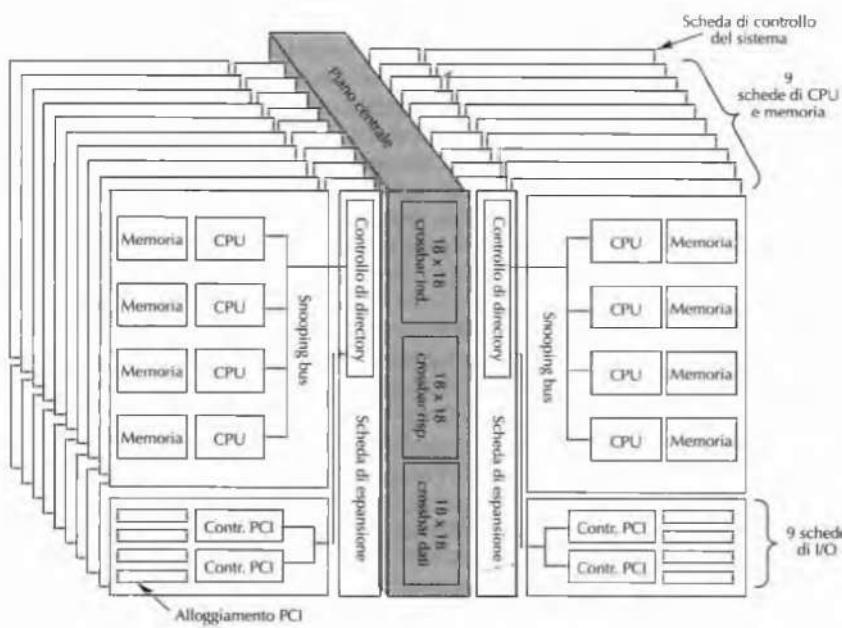


Figura 8.32 Multiprocessore E25K di Sun Microsystems.

A livello degli insiemi di schede si utilizza lo snooping di modo che tutte le CPU locali possono esaminare tutte le richieste di memoria che provengono dallo stesso insieme di schede e che referenziano i blocchi che si trovano nelle loro cache. Perciò quando una CPU necessita di una parola di memoria, per prima cosa converte l'indirizzo virtuale in un indirizzo fisico e controlla la propria cache (gli indirizzi fisici sono di 43 bit, ma per limitare il volume del multiprocessore la memoria è di soli 576 GB). Se la linea di cache richiesta si trova nella sua cache, la parola viene prelevata direttamente. Altrimenti, i circuiti di snooping verificano se esiste una copia di quella parola in qualche altra locazione dell'insieme di schede. Se così fosse, la richiesta verrebbe soddisfatta. Se invece la ricerca è infruttuosa, la richiesta viene trasferita sul commutatore crossbar  $18 \times 18$  degli indirizzi, come descritto in seguito. La logica di snooping può effettuare un controllo per ogni ciclo di clock. Visto che il clock oscilla a 150 MHz, è possibile effettuare 150 milioni di operazioni di snooping al secondo all'interno di ciascun insieme di schede, ovvero 2,7 miliardi di snooping/s nell'intero sistema.

Anche se da un punto di vista logico la circuiteria di snooping è un bus (Figura 8.32) dal punto di vista fisico si tratta di un dispositivo ad albero, in cui i comandi attraversano l'albero dall'alto verso il basso o viceversa. Quando una CPU o una scheda PCI emettono un indirizzo, questo giunge a un ripetitore di indirizzi tramite una connessione diretta (Figura 8.33). I due ripetitori di indirizzi convergono sulla scheda di espansione, da cui gli indirizzi vengono rispediti indietro verso la parte bassa dell'albero, affinché ogni di-

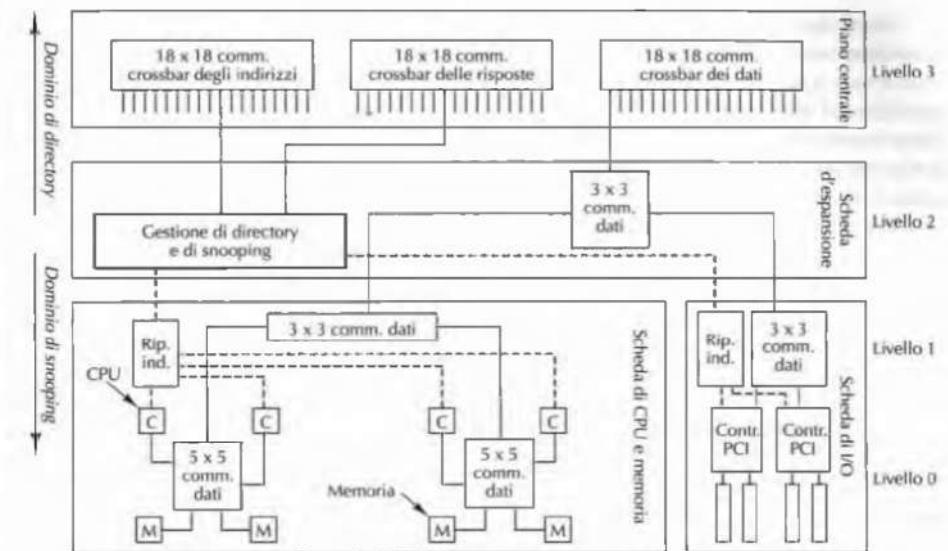


Figura 8.33 Sun Fire E25K usa un'interconnessione a quattro livelli. Le linee tratteggiate rappresentano i percorsi degli indirizzi, quelle continue i percorsi dei dati.

spositivo possa verificare la presenza della parola richiesta. Questo stratagemma serve a evitare l'implementazione di un bus che collega tre schede diverse.

I trasferimenti di dati usano un'interconnessione a quattro livelli (Figura 8.33) allo scopo di raggiungere prestazioni elevate. A livello 0 i chip con le copie di CPU e le memorie sono collegati tramite un piccolo commutatore crossbar da cui si diparte anche un collegamento verso il livello 1. I due gruppi di memorie e copie di CPU sono collegati da un secondo commutatore crossbar del livello 1. I commutatori crossbar sono circuiti progettati su misura e ciascuno di loro può prelevare gli input sia dalle righe sia dalle colonne, anche se non tutte le combinazioni vengono utilizzate (e non tutte sono sensate). Tutta la circuiteria di commutazione che si trova sulle schede è costruita a partire da commutatori crossbar  $3 \times 3$ .

Ogni insieme di schede ne contiene tre: la scheda di CPU e memoria, la scheda di I/O e la scheda di espansione, che collega le prime due. Il livello 2 d'interconnessione, che si trova sulla scheda di espansione, è un altro commutatore crossbar  $3 \times 3$  che collega la memoria alle porte di I/O (che sono mappate in memoria come in tutti gli UltraSPARC). Tutti i trasferimenti di dati, in ingresso o in uscita dagli insiemi di schede, che siano diretti alla memoria o a una porta di I/O, passano attraverso il commutatore di livello 2. Infine, i dati che devono essere trasferiti verso una scheda distante o che provengono da una di loro, passano attraverso un commutatore crossbar  $18 \times 18$  che si trova a livello 3.

I trasferimenti di dati sono effettuati a gruppi di 32 byte alla volta, perciò l'unità di trasferimento usuale, 64 byte, impiega due cicli di clock.

Dopo aver dato uno sguardo alla disposizione dei componenti, indirizziamo ora la nostra attenzione sulla modalità operativa della memoria condivisa. A livello più basso, i 576 GB di memoria sono suddivisi in  $2^{29}$  blocchi di 64-byte ciascuno. Questi blocchi sono le unità atomiche del sistema di memoria e a ciascuno di loro è associata una scheda di appartenenza (*home board*) che li ospita preferibilmente quando non sono usati da un'altra parte. La maggior parte dei blocchi si trova per gran parte del tempo nelle rispettive schede di appartenenza. Tuttavia, quando una CPU ha bisogno di un certo blocco di memoria, che si trovi sulla sua scheda o su di una delle rimanenti 17, ne richiede una copia per la propria cache e quindi vi accede all'interno della cache. Anche se ogni chip dell'E25K contiene due CPU, queste condividono una sola cache fisica e perciò ne condividono tutti i blocchi.

I blocchi di memoria e le linee di cache di ogni chip di CPU si possono trovare in uno dei tre stati seguenti:

1. accesso esclusivo (in scrittura)
2. accesso condiviso (in lettura)
3. non valido (cioè vuoto).

Quando una CPU vuole leggere o scrivere una parola di memoria, per prima cosa controlla nella propria cache. Se non vi trova la parola, emette una richiesta locale per l'indirizzo fisico corrispondente che viene propagata solo all'interno dello stesso insieme di schede. Se una delle cache nell'insieme di schede contiene la linea richiesta, la logica di snooping rileva il successo della ricerca e risponde alla richiesta. Se la linea si trova nello stato di accesso esclusivo, viene trasferita al richiedente e la copia originale viene contrassegnata come non valida. Se si trova nello stato di accesso condiviso, la cache non risponde perché se la linea è "pulita" la memoria risponde sempre.

Se la logica di snooping non riesce a trovare la linea di cache oppure se questa è presente e condivisa, una richiesta lungo il piano centrale viene spedita verso la scheda di appartenenza della linea per rintracciare il corrispondente blocco di memoria. Lo stato di ciascun blocco di memoria è conservato nei bit ECC del blocco, perciò la scheda di appartenenza può determinare immediatamente il suo stato. Se il blocco non è condiviso oppure è condiviso da una o più schede remote, la memoria di appartenenza è aggiornata e la richiesta può essere soddisfatta dalla memoria della scheda di appartenenza. In tal caso, la copia della linea di cache viene trasmessa attraverso il commutatore crossbar dei dati nell'arco di due cicli di clock, e nel volgere di qualche tempo giunge alla CPU richiedente.

Se la richiesta era di lettura, viene creato un elemento nella directory della scheda di appartenenza a specificare che un nuovo cliente condivide la linea di cache e che la transazione è stata ultimata. Se invece la richiesta era di scrittura, bisogna spedire un messaggio d'invalidazione a tutte le (eventuali) schede che ne posseggono una copia. Così facendo, la scheda richiedente si trova ad avere l'unica copia in circolazione.

Si consideri ora il caso in cui il blocco richiesto si trovi nello stato di accesso esclusivo presso una scheda differente da quella di appartenenza. Quando la scheda di appartenenza riceve la richiesta, cerca nella directory l'identità della scheda remota e spedisce al richiedente un messaggio in cui specifica la locazione della linea di cache. Il richiedente spedisce quindi una richiesta all'insieme di schede corretto e questo, una volta ricevuto il messaggio, risponde inviando la linea di cache. Se si tratta di una richiesta di lettura, la linea viene con-

trassegnata come condivisa e viene spedita anche una copia alla scheda di appartenenza. Se è invece di scrittura, il destinatario del messaggio invalida la propria copia, di modo che il richiedente possa avere la sua copia esclusiva.

Ogni scheda ha  $2^{29}$  blocchi di memoria, quindi nel caso peggiore ci vorrebbe una directory di  $2^{29}$  elementi per tener traccia di tutti i blocchi. In realtà la directory è molto più piccola, perciò può capitare che non ci sia abbastanza spazio per alcuni elementi (la ricerca all'interno della directory è svolta in maniera associativa). In tale evenienza, la directory di appartenenza è costretta a emettere una richiesta broadcast a tutte le altre 17 schede per localizzare il blocco. Il commutatore crossbar delle risposte è coinvolto nel protocollo di coerenza e aggiornamento delle directory, perché gestisce gran parte del traffico di risposta al richiedente. La divisione del traffico di protocollo su due bus (indirizzi e risposte), e del traffico dati su di un terzo bus, incrementa il volume di dati trasferiti nel sistema.

Grazie alla distribuzione del carico di lavoro tra numerosi dispositivi che risiedono su schede diverse, il Sun Fire E25K raggiunge delle prestazioni molto elevate. Oltre ai 2,7 miliardi di snooping/s già menzionati, il piano centrale può gestire fino a nove trasferimenti simultanei, tra nove schede mittenti e nove destinatarie. Il crossbar dei dati è largo 32 byte, perciò a ogni ciclo di clock è possibile trasferire 288 byte attraverso il piano centrale. Con una frequenza di clock di 150 MHz, ciò equivale a un picco di larghezza di banda cumulativa di 40 GB/s quando tutti gli accessi avvengono a distanza. Se poi il software è in grado di collocare le pagine in modo tale da garantire una predominanza di accessi locali, la larghezza di banda del sistema può superare notevolmente i 40 GB/s.

Rimandiamo a (Charlesworth, 2002; Charlesworth, 2001) per maggiori informazioni tecniche su Sun Fire.

### 8.3.5 Multiprocessori COMA

Le macchine NUMA e CC-NUMA presentano lo svantaggio per cui i riferimenti alla memoria distante sono molto più lenti degli accessi alla memoria locale. Le macchine CC-NUMA nascondono in una certa misura questa differenza di prestazioni grazie al caching. Malgrado ciò, se la quantità di dati distanti richiesti eccede di molto la capacità della cache, si verificheranno continui fallimenti di cache e le prestazioni degraderanno sensibilmente.

Dunque ci troviamo da una parte con le macchine UMA che hanno prestazioni eccellenti, ma che sono limitate per dimensioni e abbastanza costose; dall'altra ci sono le macchine NC-NUMA che raggiungono dimensioni maggiori, ma richiedono una collocazione delle pagine manuale o semiautomatica, spesso con risultati altalenanti. Il problema è che è difficile predire quali pagine saranno richieste e dove; inoltre, le pagine costituiscono spesso un'unità troppo grande perché le si possa trasferire da una parte all'altra del sistema. Le macchine CC-NUMA, come il Sun Fire E25K, possono manifestare scarse prestazioni quando molte CPU richiedono dati distanti. Tutto considerato, ciascuno di questi progetti presenta delle gravi limitazioni.

Esiste un altro tipo di multiprocessori che cerca di aggirare questi problemi usando ogni memoria principale di CPU come una cache. Questo progetto si chiama **COMA** (*Cache Only Memory Access*, "accesso alla memoria di tipo cache") e non richiede che ogni pagina abbia una macchina prefissata di appartenenza, come succede nelle macchine NUMA e CC-NUMA. Infatti il concetto di pagina perde di significato: lo spazio degli indirizzi fisici è invece diviso in linee di cache che vengono trasferite all'interno del sistema su richiesta.

Le linee non hanno una macchina che le ospita di preferenza, sono un po' come i nomadi di alcuni paesi del Terzo Mondo: la loro casa è il luogo in cui si trovano in quel momento. Una memoria che si limita ad attrarre le linee quando sono richieste si dice **memoria d'attrazione**, in inglese *attraction memory*, in contrapposizione a *home memory*. L'utilizzo della RAM principale come una grande cache aumenta notevolmente il tasso di hit e dunque le prestazioni.

Sfortunatamente, come al solito, niente si ottiene per niente. I sistemi COMA introducono due nuovi problemi:

1. come localizzare le linee di cache
2. che cosa succede quando viene estromessa dalla memoria l'ultima copia di una linea.

Il primo problema riguarda il fatto che, quando la MMU ha tradotto l'indirizzo virtuale in un indirizzo fisico, se la linea non si trova nell'hardware della vera cache è difficile stabilire se sia veramente in memoria. L'hardware di paginazione non dà alcun aiuto, perché ogni pagina è fatta di molte linee di cache che vagabondano liberamente. Inoltre, se anche si sapesse che la linea non si trova in memoria, ...dove si trova allora? Non c'è modo di chiederlo alla sua macchina di appartenenza, perché non c'è alcuna macchina di appartenenza.

Esistono delle proposte per risolvere il problema della localizzazione. Al fine di stabilire se una linea di cache si trova in memoria, è possibile aggiungere nuovo hardware che tenga traccia di un'etichetta per ogni linea presente nella cache. La MMU potrebbe poi confrontare l'etichetta della linea richiesta con le etichette delle linee di cache in memoria in cerca di una corrispondenza.

In alternativa, si possono mappare le pagine intere senza richiedere però la presenza di tutte le linee di cache. Secondo questa soluzione, l'hardware dovrebbe tenere una bit map per ogni pagina; ciascun bit corrisponderebbe a una linea di cache e ne indicherebbe la presenza o l'assenza. Questo progetto si chiama **COMA semplice** (S-COMA) e stabilisce che, se una linea è presente, deve trovarsi nella posizione corretta all'interno della sua pagina, ma se non è presente, qualsiasi tentativo di utilizzarla causa una trap che consente al software di cercarla e di caricarla.

A questo punto si rende necessario saper reperire linee veramente distanti. Una soluzione è quella di attribuire a ogni pagina una macchina di appartenenza in termini della collocazione del suo elemento di directory, non dei suoi dati. Così è possibile spedire un messaggio alla macchina di appartenenza quanto meno per localizzare la linea di cache. Un altro schema richiede di organizzare la memoria come un albero e percorrerlo verso l'alto finché non viene trovata la linea ricercata.

Il secondo problema elencato precedentemente equivale a non cancellare l'ultima copia di una linea. Come nelle macchine CC-NUMA, anche qui una linea di cache si può trovare simultaneamente presso più nodi. Al verificarsi di un fallimento di cache si rende necessario il caricamento di una linea, il che spesso richiede l'estromissione di un'altra linea. Che cosa succede se si sceglie una linea che è una copia unica? In tal caso bisogna evitare di cancellarla.

Una soluzione è interrogare la directory e verificare se ci sono altre copie. In caso affermativo si può cancellare la linea senza problemi, altrimenti bisogna trasferirla da qualche altra parte. Un'altra soluzione consiste nell'etichettare una copia di ogni linea come copia originale che non può essere mai cancellata. In questo modo si può evitare il controllo nella directory. Tirando le somme, le macchine COMA promettono prestazioni migliori di quelle

CC-NUMA, ma ne esistono ancora pochi esemplari e ci vuole maggiore esperienza per poter giudicare. Le prime due macchine COMA sono state il KSR-1 (Burkhardt et al., 1992) e la Data Diffusion Machine (Hagersten et al., 1992). Una macchina più recente è SDAARC (Eschmann et al., 2002).

## 8.4 Multicomputer a scambio di messaggi

Come indicato nella Figura 8.21, ci sono due tipi di processori paralleli MIMD: i multiprocessori e i multicomputer. Nel paragrafo precedente abbiamo studiato i primi e abbiamo visto che il loro sistema operativo li percepisce come provvisti di una memoria condivisa cui è possibile accedere tramite le ordinarie istruzioni di LOAD e STORE. Abbiamo illustrato le possibili implementazioni di questa memoria condivisa: snooping bus, commutazione crossbar, reti a commutazione multilivello e altri schemi basati su directory. Al di là di ciò, i programmi scritti per un multiprocessore possono accedere a ogni locazione di memoria senza sapere nulla della topologia interna o dello schema implementativo. Questa illusione è ciò che rende i multiprocessori così attraenti e la ragione per cui costituiscono un modello di programmazione così gradito agli sviluppatori di software.

A ogni modo, anche i multiprocessori presentano alcune limitazioni ed è per questo che i multicomputer sono altrettanto importanti. In primo luogo, le prestazioni dei multiprocessori peggiorano con il crescere delle loro dimensioni. Abbiamo già visto l'enorme quantità di hardware impiegata da Sun nell'E25K per raggiungere i 72 chip con doppia CPU, mentre in seguito studieremo un multicomputer che ha ben 65.536 CPU. Ci vorranno anni prima che qualcuno riesca a costruire un multiprocessore commerciale da 65.536 nodi, mentre per quel tempo saranno già in uso multicomputer da milioni di nodi.

Oltre a ciò, un altro fattore che può influenzare le prestazioni di un multiprocessore è la contesa di memoria: se 100 CPU cercano di leggere e scrivere sempre le stesse variabili, la contesa delle varie memorie, dei bus e delle directory può comportare un grosso degrado delle prestazioni.

La conseguenza di queste considerazioni è che si pone oggi molto interesse nei multicomputer, cioè nei computer paralleli in cui 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 per lo scambio esplicito di messaggi, visto che non possono accedere alle rispettive memorie per mezzo d'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 di uno dei tipi esaminati nel Paragrafo 8.1.2. Esistono diverse topologie, vari schemi di commutazione e algoritmi d'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).

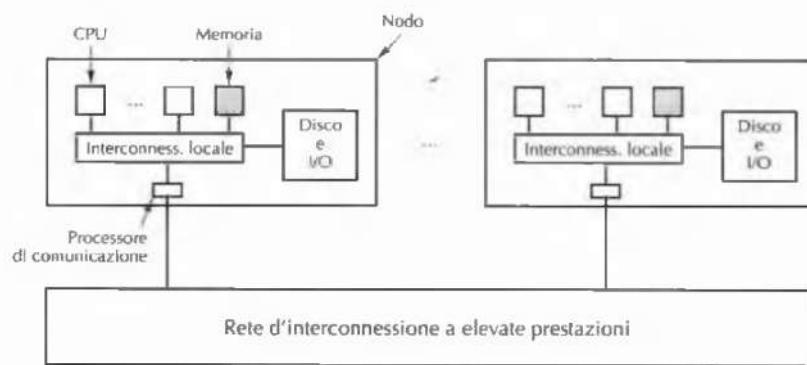


Figura 8.34 Schema di un multicomputer.

#### 8.4.1 Reti d'interconnessione

La Figura 8.34 (che mostra un multicomputer idealizzato) evidenzia il fatto che i multicomputer sono tenuti insieme dalla rete d'interconnessione. Da questo punto di vista i multiprocessori e i multicomputer sono sorprendentemente simili, poiché i multiprocessori sono spesso dotati di moduli di memoria che devono essere interconnessi tra loro e con le CPU. Di conseguenza, quanto trattato in questo paragrafo si applica spesso a entrambi i tipi di sistemi.

La ragione fondamentale della somiglianza tra le reti d'interconnessione dei multiprocessori e quelle dei multicomputer è che in fondo si tratta in entrambi i casi di reti a scambio di messaggi.

Anche nei sistemi monoprocessoressi, quando la CPU vuole leggere o scrivere una parola, in genere attiva certe linee sul bus e aspetta una risposta. Questa azione è sostanzialmente uno scambio di messaggi: il primo agente invia una richiesta e aspetta una risposta. Nei grandi multiprocessori, la comunicazione tra le CPU e la memoria distante avviene quasi sempre tramite l'invio esplicito di un messaggio di richiesta dati, il cosiddetto **pacchetto**, da parte di una CPU verso la memoria, che replica con un pacchetto di risposta.

#### Topologia

La topologia di una rete d'interconnessione descrive il modo in cui sono disposti i collegamenti e i commutatori, per esempio lungo un anello o su una griglia. I progetti topologici possono essere modellati con grafi i cui lati sono i collegamenti e i cui nodi sono i commutatori, come nella Figura 8.35. Ogni nodo di una rete d'interconnessione (o del suo grafo) dispone di un certo numero di collegamenti in uscita da esso. I matematici chiamano questo numero il **grado (uscente)** del nodo, mentre gli ingegneri lo chiamano il **fanout** (da *fan out*, "ventaglio di uscite"). In generale, maggiore è il fanout, più numerose sono le scelte d'instradamento e quindi più alta è la tolleranza agli errori: anche se un collegamento è difettoso, la rete può continuare a funzionare se riesce ad aggirarlo. Se ogni nodo ha  $k$  lati e se i collegamenti sono effettuati correttamente, è possibile progettare una rete che resti pienamente connessa anche se si interrompono  $k - 1$  collegamenti.

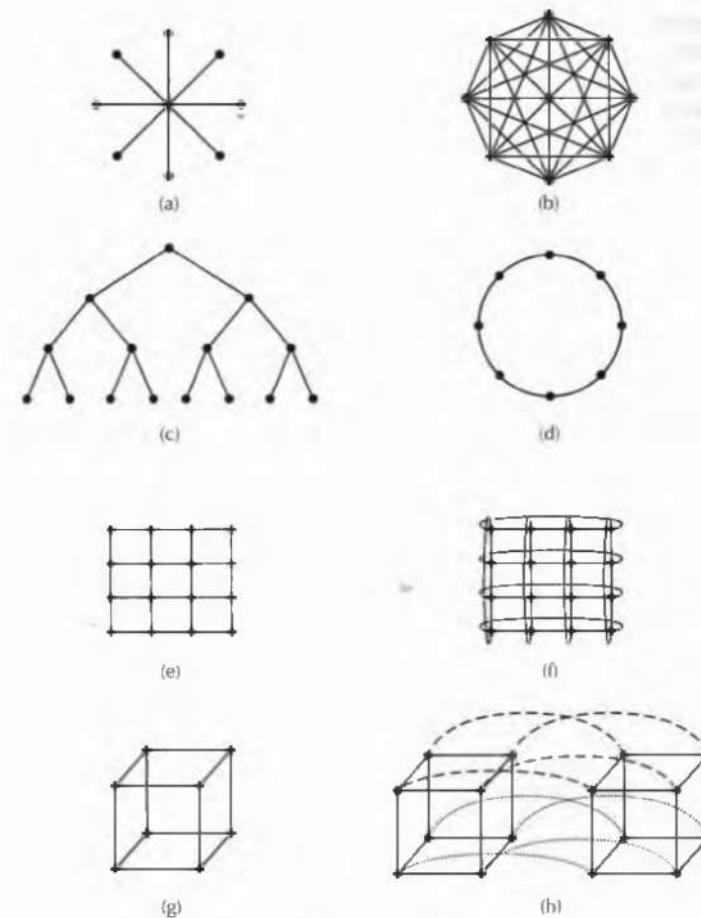


Figura 8.35 Alcune topologie. I pallini rappresentano i commutatori, mentre non sono mostrate le CPU e le memorie. (a) Stella. (b) Interconnessione completa. (c) Albero. (d) Anello. (e) Griglia. (f) Toro. (g) Cubo. (h) Ipercubo a quattro dimensioni.

Un'altra proprietà della rete d'interconnessione (o del suo grafo) è il suo **diametro**. Se misuriamo la distanza tra due nodi con il numero minimo di lati che devono essere attraversati per giungere da uno all'altro, allora il diametro del grafo è la distanza tra due nodi che si trovano alla massima distanza. Il diametro di una rete d'interconnessione è legato al ritardo che si può verificare nel caso peggiore quando si spedisce un pacchetto da una CPU a un'altra o da una CPU a una memoria, dal momento che ogni passaggio attraverso un collegamento impiega una quantità finita di tempo. Più piccolo è il diametro, migliori sono le prestazioni nel caso peggiore. È importante anche la distanza media tra i nodi, poiché influenza il tempo medio di consegna di un pacchetto.

Un'altra proprietà significativa di una rete d'interconnessione è la sua capacità di trasmissione, ovvero la quantità di dati che può trasferire al secondo. Un modo conveniente per misurare questa capacità è usare la **larghezza di banda di bisezione**. Per calcolarla, prima bisogna ripartire (concettualmente) la rete in due parti uguali (per numero di nodi) e scollegarle rimuovendo i lati che collegano nodi di parti diverse. Quindi si calcola la larghezza di banda totale dei lati che sono stati rimossi. Ci possono essere molti modi diversi di ripartire la rete in due parti uguali e la larghezza di banda di bisezione è il valore minimo che si ottiene per tutte le possibili partizioni. Il significato di questo numero è che, se per esempio la larghezza di banda di bisezione è di 800 bit/s, allora, se c'è molta comunicazione in corso tra le due metà, nel caso peggiore la quantità di dati in trasferimento non può superare gli 800 bit/s. Molti progettisti credono che la larghezza di banda di bisezione sia la proprietà più importante per misurare le prestazioni di una rete d'interconnessione, perciò molte reti vengono progettate appositamente per massimizzarla.

Le reti d'interconnessione possono essere caratterizzate dalla loro **dimensionalità**. Per i nostri scopi, la dimensionalità è il numero di modi diversi in cui è possibile raggiungere la destinazione a partire dalla sorgente. Se non c'è mai possibilità di scelta (cioè se c'è solo un cammino possibile da ogni sorgente a ogni destinazione) allora la rete è zero-dimensionale. Se c'è una dimensione in cui è possibile effettuare una scelta, per esempio andare a est o a ovest, allora la rete è uno-dimensionale. Se sono due assi lungo cui si può scegliere (per esempio se un pacchetto può scegliere di proseguire tra est e ovest, oppure tra nord e sud) allora la rete è due-dimensionale, e così via.

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 zero-dimensionale 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 altro progetto zero-dimensionale che però si trova all'altro estremo dello spettro: un'**interconnessione completa** (e completo è detto il grafo a lei associato). In questo caso ogni nodo è collegato direttamente a ogni altro. Questo progetto massimizza la larghezza di banda di bisezione, minimizza il diametro ed è straordinariamente tollerante agli errori (potrebbero interrompersi sei collegamenti qualsiasi e la rete sarebbe ancora connessa). Sfortunatamente richiede  $k(k - 1)/2$  lati per collegare  $k$  nodi, un numero che diventa presto intrattabile al crescere di  $k$ .

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. Dal momento che ci sarà molto traffico in corrispondenza dei nodi vicini alla cima dell'albero, questi pochi nodi diventeranno facilmente un collo di bottiglia per l'intera rete. Una possibile soluzione è aumentare la larghezza di banda di bisezione attribuendo maggiore larghezza di banda ai collegamenti che si trovano più in alto. Per esempio il livello più in basso potrebbe avere capacità  $b$ , quello successivo capacità  $2b$  e il livello più alto capacità  $4b$ . Questo tipo di progetto si chiama **fat tree** ("albero grasso") ed è stato utilizzato in alcuni multicomputer commerciali, come l'ormai scomparso CM-5 di Thinking Machines.

L'**anello** della Figura 8.35(d) è, secondo la nostra definizione, una topologia monodimensionale, perché ogni pacchetto può scegliere soltanto se andare a destra o a sinistra. La **griglia** o **mesh** (maglia, reticolato) della Figura 8.35(e) è un progetto bidimensionale, utilizzato in molti sistemi commerciali. È altamente regolare, facile da implementare al crescere delle dimensioni del sistema e il suo diametro cresce come la radice quadrata del numero di 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, perché i vertici opposti possono ora comunicare in soli due passaggi.

Un'altra topologia diffusa è una struttura 3D con nodi nei punti le cui coordinate sono gli interi nell'intervallo che va da  $(1, 1, 1)$  a  $(l, m, n)$ . 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.

Il **cubo** della Figura 8.35(g) è una normale topologia tridimensionale. Nella figura illustriamo un cubo  $2 \times 2 \times 2$ , ma in generale si potrebbe avere un cubo  $k \times k \times k$ . La Figura 8.35(h) mostra un ipercubo a quattro dimensioni costruito a partire da due cubi mediante il collegamento dei nodi corrispondenti. Potremmo costruire un cubo a cinque dimensioni clonando la struttura della Figura 8.35(h) e collegando i nodi corrispondenti in modo da formare un blocco di quattro cubi. Per raggiungere le sei dimensioni possiamo replicare il blocco di quattro cubi e interconnettere i nodi corrispondenti, e così via. Un cubo  $n$ -dimensionale così formato si chiama un **iper cubo**. Molti computer paralleli usano questa topologia perché il suo diametro cresce linearmente con  $n$ . Detto altrimenti, il diametro è il logaritmo in base 2 del numero di nodi, perciò un ipercubo a dieci dimensioni di 1024 nodi ha un diametro di 10, il che garantisce proprietà di ritardo eccellenti. Si noti, come termine di paragone, il caso di 1024 nodi sistemati in una griglia  $32 \times 32$ : il loro diametro è 62, più di sei volte quello dell'iper cubo. Il prezzo pagato dall'iper cubo per avere un diametro inferiore è un maggiore fanout e quindi un maggiore numero di collegamenti (che accrescono il costo della rete). Malgrado ciò, l'iper cubo è una scelta comune a molti sistemi dalle prestazioni elevate.

Esistono multicomputer di tutte le forme e dimensioni, perciò è difficile darne una tassonomia chiara. Si possono identificare comunque due "stili" generali: gli MPP e i cluster. Studiamoli uno per volta.

#### 8.4.2 MPP: processori massicciamente paralleli

La prima categoria che individuiamo è formata dagli **MPP** (*Massively Parallel Processor*, "processore massicciamente parallelo"): supercomputer enormi da svariati milioni di euro, usati per svolgere ingenti quantità di calcoli scientifici, ingegneristici o industriali, per la gestione di grandi numeri di transazioni al secondo o per l'immagazzinamento di dati (memorizzazione e gestione di database immensi). Inizialmente gli MPP venivano usati come supercomputer scientifici, ma adesso gran parte di loro è impiegata in ambienti commerciali. Queste macchine sono in un certo senso i successori dei poderosi mainframe degli anni '60 (ma la parentela è molto lontana, come se un paleontologo sostenesse che uno stormo di passeri discende dal *Tyrannosaurus Rex*). Si può dire che gli MPP hanno soppiantato al vertice della catena alimentare digitale le macchine SIMD, i supercomputer vettoriali e le unità di calcolo vettoriale.

Molte macchine MPP usano come processori CPU standard. È frequente l'uso di Pentium (Intel), di UltraSPARC (Sun) e di PowerPC (IBM). Ciò che distingue gli MPP è l'uso di una rete d'interconnessione brevettata, ad altissime prestazioni, progettata per trasferire i messaggi con poca latenza ed elevata larghezza di banda. Sono due proprietà importanti, perché nella stragrande maggioranza dei casi i messaggi sono piccoli (ben sotto i 256 byte), ma gran parte del traffico totale è dovuto ai messaggi grandi (più di 8 KB). Gli MPP vengono forniti con grandi quantità di software e librerie brevettati.

Un'altra caratteristica degli MPP è la loro enorme capacità di I/O. I problemi abbastanza grandi da meritare l'impiego di un MPP coinvolgono sempre enormi quantità di dati, spesso dell'ordine dei terabyte. I dati devono essere distribuiti tra molti dischi e devono essere trasferiti all'interno della macchina a grande velocità.

Infine, un'altra questione d'interesse circa gli MPP è la loro tolleranza agli errori. La presenza di migliaia di CPU rende inevitabili un certo numero di anomalie nel corso del tempo. Interrrompere una computazione di 18 ore per l'insuccesso di una CPU è inaccettabile, specialmente se è prevedibile osservare una tale anomalia nel corso di una settimana. Perciò gli MPP più grandi sono sempre dotati di hardware e software speciali per il monitoraggio del sistema, per il rilevamento delle anomalie e il loro trattamento.

Sembra interessante studiare i principi generali alla base dei progetti MPP, in realtà va detto che non ci sono molti principi di base. In fondo gli MPP sono una raccolta di nodi di calcolo più o meno standard, collegati tramite un'interconnessione molto veloce scelta tra uno dei tipi già illustrati. In ragione di ciò, passiamo piuttosto allo studio di due esempi di MPP: BlueGene/L e Red Storm.

### BlueGene

Come primo esempio di processore massicciamente parallelo prendiamo in esame il sistema BlueGene di IBM, un progetto concepito nel 1999 per la risoluzione di problemi che esigono molte risorse di calcolo, come quelli che sorgono nelle scienze biologiche. Per esempio, i biologi credono che la struttura tridimensionale di una proteina determini la sua funzionalità, eppure il calcolo della struttura spaziale di una piccola proteina a partire dalle leggi fisiche impiegava anni sui supercomputer del tempo. Ogni essere umano contiene più di mezzo milione di proteine, molte delle quali sono estremamente grandi, ed è noto che certi loro ripiegamenti errati (*misfolding*) sono responsabili di alcune malattie (come la fibrosi cistica). È chiaro che la determinazione della stereostruttura di tutte le proteine umane richiederebbe un incremento della capacità di calcolo mondiale di svariati ordini di grandezza, e la modellazione delle proteine è solo uno dei problemi per cui BlueGene è stato progettato. Esistono sfide ugualmente complesse nella dinamica molecolare, dei modelli climatici, in astronomia o anche nella modellazione finanziaria, che richiederebbero tutte un potenziamento dei supercomputer di diversi ordini di grandezza.

IBM crede così tanto nel mercato del supercalcolo massiccio che ha investito 100 milioni di dollari nel progetto e nella costruzione di BlueGene. Nel novembre 2001, il Livermore National Laboratory, gestito dal dipartimento statunitense per l'energia, ha firmato un contratto come primo cliente di un esemplare della famiglia BlueGene, chiamato **BlueGene/L**.

L'obiettivo del progetto non era solo quello di produrre il supercomputer MPP più veloce del mondo, ma anche di produrre l'esemplare più efficiente in termini di teraflop/dollaro, teraflop/watt e teraflop/m<sup>3</sup>, cioè di costi, consumo energetico e dimensione. Per que-

sta ragione IBM rifiutò la filosofia che ispirava gli MPP precedenti, cioè di usare i componenti più veloci che si potessero comprare. Al contrario, si decise di produrre un componente su misura con un intero sistema nel chip, che girasse a velocità modesta e che consumasse poca energia, in modo da produrre una macchina davvero grande e con un'alta densità di assemblaggio. Il primo chip venne alla luce nel giugno 2003. Il primo quarto del sistema BlueGene/L, costituito da 16.384 nodi, divenne pienamente operativo a partire dal novembre 2004 e, grazie ai suoi 71 teraflop/s, fu insignito del premio di supercomputer più veloce al mondo. Il consumo di soli 0,4 MWatt gli garantì anche il premio di supercomputer più efficiente della sua categoria, con 177,5 megaflop/watt. La consegna della parte restante del sistema, che lo porterebbe a una dimensione di 65.536 nodi di calcolo, era prevista per l'estate del 2005.

Il punto forte del sistema BlueGene/L è rappresentato dai suoi nodi, ciascuno costituito dal chip su misura illustrato nella Figura 8.36, formato da due core PowerPC 440 a 700 MHz. Il PowerPC 440 è un processore superscalare a pipeline e a doppia emissione, molto diffuso nei sistemi integrati. Ogni core ha un paio di unità in virgola mobile a doppia emissione, che possono emettere congiuntamente quattro istruzioni in virgola mobile per ogni ciclo. Le unità in virgola mobile sono state potenziate con l'aggiunta d'istruzioni di tipo SIMD che si rivelano spesso utili al calcolo scientifico vettoriale. Benché non sia proprio inefficiente, sicuramente non è un multiprocessore al vertice della gamma.

I due core di CPU sono identici per struttura, ma sono programmati diversamente: uno viene usato per il calcolo, l'altro per gestire la comunicazione tra i 65.536 nodi.

Nel chip sono presenti tre livelli di cache. Le cache L1 (primo livello) hanno 32 KB per le istruzioni e 32 per i dati. Non c'è alcuna coerenza tra le cache L1 delle due CPU perché i core PowerPC 440 non la supportavano e si è deciso di non modificarli. Le cache L2 (secondo livello) sono unificate e grandi 2 KB; non vengono usate come vere cache, ma come buffer per il prefetch. Ognuna fa snooping sull'altra cache e sono reciprocamente consistenti. La cache L3 (terzo livello) ha 4 MB, è condivisa, unificata e consistente, e rifornisce le

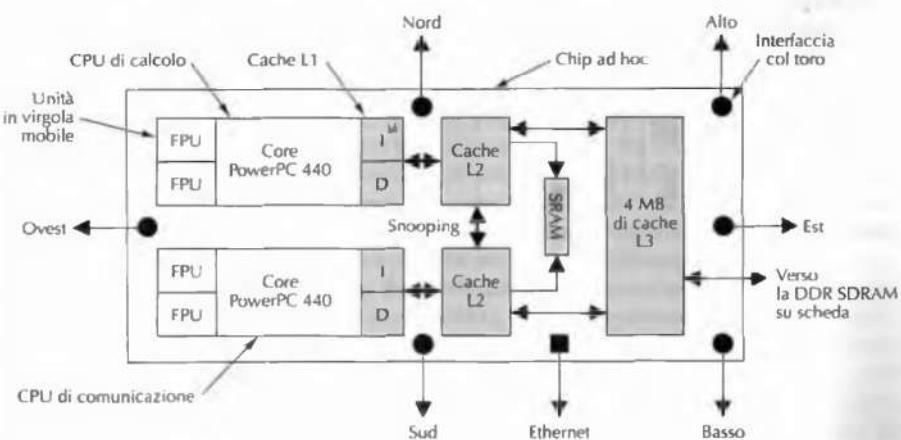


Figura 8.36 Il chip costruito per BlueGene/L.

cache L2. Un riferimento alla memoria che fallisce nella cache L1, ma che ha successo nella L2, impiega circa 11 cicli di clock. Un fallimento a livello 2 che ha successo a livello 3 impiega 28 cicli. Infine, un riferimento che non ha successo neanche nella cache L3 deve raggiungere la SDRAM principale e impiega 75 cicli.

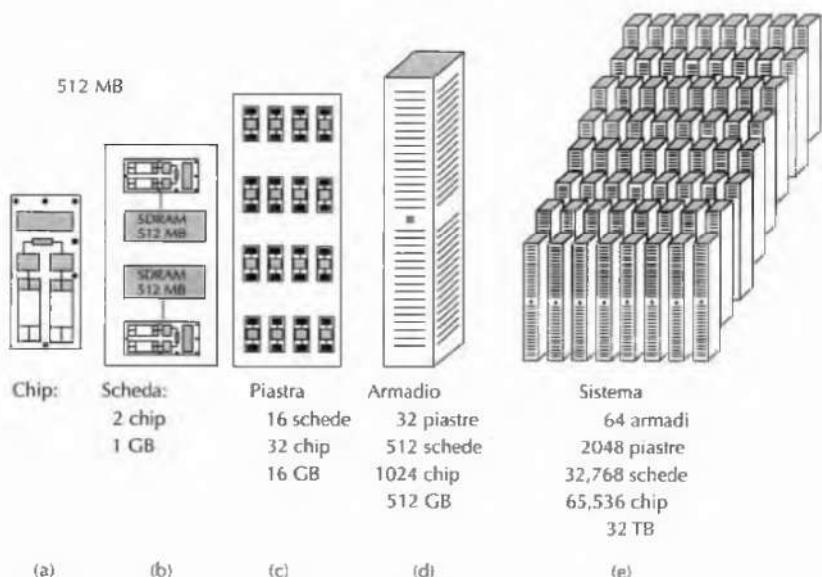
Le cache L2 sono collegate a una piccola SRAM, a sua volta connessa all'interfaccia JTAG per il riavvio, il debugging e la comunicazione con l'host principale che gestisce lo stack di sistema e fornisce i semafori, le barriere e altre operazioni di sincronizzazione.

Al livello gerarchico successivo, IBM ha progettato su misura una scheda con due chip (Figura 8.36) e una RAM da 1 GB (le versioni future potranno contenere 4 GB). Il chip e la scheda sono mostrati nelle Figure 8.37(a)-(b).

Le schede sono montate a innesto su una piastra che può ospitarne 16 per un totale di 32 chip (e quindi di 32 CPU di calcolo). Poiché ogni scheda contiene 1 GB di SDRAM, ogni piastra contiene 16 GB di memoria. La Figura 8.37(c) illustra una piastra.

A livello successivo, le piastre vengono inserite in un armadio di base 60 × 90 cm. La metà superiore ospita 16 piastre e altre 16 trovano spazio nella metà inferiore, per un totale di 1024 CPU, una densità molto elevata. I due gruppi di 16 piastre sono separati da un commutatore che può estromettere un gruppo dal sistema per ragioni di manutenzione. La Figura 8.37(d) mostra un armadio.

Il sistema completo consiste in 64 armadi con 65.536 CPU di calcolo e altre 65.536 per la comunicazione, come rappresentato nella Figura 8.37(e). Grazie alle 131.072 CPU intere a doppia emissione e alle 262.144 CPU in virgola mobile a doppia emissione, il si-



**Figura 8.37** Componenti gerarchiche di BlueGene/L: (a) Chip (b) Scheda (c) Piastra (d) Armadio (e) Sistema.

stema potrebbe emettere fino a 786.432 istruzioni in un ciclo di clock. Tuttavia, una delle unità intere serve a rifornire le unità in virgola mobile e ciò riduce il numero d'istruzioni a 655.360 in ogni ciclo di clock, cioè a  $4,6 \times 10^{14}$  istruzioni/s, il che lo rende comunque e di gran lunga il più potente sistema di calcolo mai costruito.

Il sistema è un multicomputer nel senso che nessuna CPU ha accesso diretto alla memoria, fatta eccezione per i 512 MB residenti sulla sua scheda. Nessuna coppia di CPU condivide memoria comune. Inoltre, non si effettua paginazione a richiesta perché non ci sono dischi locali da cui paginare. Il sistema dispone invece di 1024 nodi di I/O collegati a dischi e ad altri dispositivi periferici.

Tutto considerato, a dispetto delle sue dimensioni esagerate, il sistema è abbastanza semplice e introduce poca tecnologia, a eccezione della capacità di assemblaggio ad alta densità. La scelta della semplicità non è stata accidentale, ma è frutto dell'aver perseguito come obiettivi primari l'elevata affidabilità e reperibilità dei componenti. Di conseguenza, è stata dedicata molta cura ingegneristica all'alimentazione elettrica, alle ventole, al raffreddamento e alla cablatura, allo scopo di raggiungere un tempo medio di 10 giorni tra l'occorrenza di due anomalie successive.

La connessione di tutti i chip richiede una rete d'interconnessione scalabile e dalle elevate prestazioni. Il progetto impiegato è un toro  $64 \times 32 \times 32$ . Perciò ogni CPU ha bisogno di sei collegamenti: due verso le CPU che si trovano logicamente in alto o in basso rispetto a essa, due verso est e ovest, due verso nord e sud. I sei collegamenti sono etichettati quindi come alto, basso, est, ovest, nord e sud, come evidenziato nella Figura 8.36. Ogni armadio è un toro  $8 \times 8 \times 16$ . Quattro paia di armadi della stessa riga formano un toro  $8 \times 32 \times 32$ . Infine, le otto righe di armadi formano il toro  $64 \times 32 \times 32$ .

Dunque tutti i collegamenti sono diretti da nodo a nodo e operano a 1,4 Gbps. Visto che ciascuno dei 65.536 nodi ha tre collegamenti verso nodi di numero "maggiore", uno lungo ciascuna dimensione, la larghezza di banda totale del sistema è di 275 terabit/s. Il contenuto informativo di questo libro è pari a circa 300 milioni di bit, inclusa la grafica in formato incapsulato PostScript, perciò BlueGene sarebbe in grado di trasferire 900.000 copie di questo libro in un secondo. Dove sarebbero distribuite e chi le potrebbe richiedere è una domanda che lasciamo al lettore come esercizio.

La comunicazione nel toro si svolge sotto forma d'instradamento **cut through virtuale**. Si tratta di una tecnica abbastanza simile alla commutazione di pacchetto store-and-forward, con la differenza che i pacchetti non vengono memorizzati prima di essere inoltrati. Non appena un byte raggiunge un nodo, può essere inoltrato immediatamente al nodo successivo lungo il cammino, addirittura prima che giunga l'intero pacchetto cui appartiene. Sono possibili sia l'instradamento dinamico (adattivo), sia deterministico (fisso). Una piccola parte dell'hardware del chip è destinato all'implementazione del cut through virtuale.

Oltre al toro principale usato per il trasferimento dati, sono presenti altre quattro reti di comunicazione. La seconda è una rete ad albero, *combining network*, che riunisce tutti i nodi. Infatti molte delle operazioni eseguite nei sistemi altamente paralleli come BlueGene/L hanno bisogno della partecipazione di tutti i nodi. Si consideri, per esempio, la ricerca del minimo in un insieme di 65.536 valori, ciascuno contenuto in un nodo. La rete in questione riunisce tutti i nodi in un albero e può essere usata per il calcolo del minimo: quando un nodo riceve i valori provenienti dai nodi a lui inferiori, sceglie il più piccolo tra questi e il proprio valore e lo inoltra lungo l'albero verso l'alto. Così facendo, il nodo radice dell'albero

viene raggiunto da un numero di pacchetti molto inferiore a quanti ne riceverebbe se tutti i 65.536 nodi gli spedissero un messaggio direttamente.

La terza rete è usata per le barriere globali e gli interrupt. Alcuni algoritmi lavorano in più passi e richiedono che ogni nodo attenda finché tutti gli altri hanno completato la fase in esecuzione prima di entrare nella successiva. La rete di barriere consente la definizione di queste fasi via software e fornisce uno strumento per la sospensione di tutte le CPU che hanno terminato una fase, fino al completamento della fase da parte di tutte le altre CPU; solo a questo punto vengono liberate dall'attesa. Anche gli interrupt viaggiano lungo questa rete.

La quarta e la quinta rete usano entrambe una Gigabit Ethernet. La prima collega i nodi di I/O ai server di file, che sono esterni a BlueGene/L, e attraverso questi a Internet. L'altra serve al debugging del sistema.

Ogni nodo di calcolo e comunicazione esegue un piccolo kernel progettato su misura, che supporta un solo utente e un solo processo. Questo processo contiene due thread, uno per ogni CPU del nodo. La semplicità della struttura è intesa a garantire prestazioni e affidabilità elevate.

Per incrementare l'affidabilità, il software applicativo può richiamare una procedura di libreria per inserire un punto di controllo (*checkpoint*). Una volta spediti in rete tutti i messaggi in uscita, si può stabilire e memorizzare un punto di controllo globale di modo che, a seguito di un'anomalia del sistema, sia possibile far ripartire l'esecuzione dal punto di controllo invece che dall'inizio. I nodi di I/O utilizzano il tradizionale sistema operativo Linux e supportano processi multipli. Si possono trovare maggiori informazioni su BlueGene/L in (Adiga et al., 2002; Almasi et al., 2003a, 2003b; Blumrich et al., 2005).

### Red Storm

Come secondo esempio di MPP, prendiamo in considerazione la macchina Red Storm (detta anche il "Martello di Thor") che si trova presso Sandia National Laboratory. Questo istituto è gestito dalla Lockheed Martin e svolge attività per conto del dipartimento statunitense dell'energia, sia di natura riservata (segreta) sia pubblica. Parte del lavoro riservato concerne il progetto e la simulazione di armi nucleari, che richiedono molta potenza di calcolo.

Sandia è un soggetto attivo da molti anni e nel passato ha avuto a disposizione numerosi supercomputer di prima caratura. Per alcuni decenni ha prediletto i supercomputer vettoriali, ma alla lunga gli MPP sono risultati più convenienti sia dal punto di vista tecnologico, sia economico. Nel 2002 ASCI Red, l'allora supercomputer MPP di Sandia, cominciava a scricchiolare. Era costituito da 9460 nodi, ma in totale disponeva di soltanto 1,2 TB di RAM e 12,5 TB di spazio su disco, e il sistema poteva macinare appena 3 teraflop/s. Perciò nell'estate del 2002 Sandia ha scelto Cray Research, un costruttore di supercomputer di vecchia data, per far costruire il sostituto dell'ASCI Red.

Il rimpiazzo venne consegnato nell'agosto 2004, dopo un lasso di tempo eccezionalmente contenuto per il progetto e l'implementazione di una macchina così grande. La ragione della celerità della sua progettazione e consegna è che Red Storm è composto quasi interamente da parti già pronte all'uso, fatta eccezione per un chip per l'instradamento progettato su misura.

La CPU scelta per Red Storm è Opteron di AMD. Opteron presenta numerose caratteristiche chiave che ne fanno la scelta migliore. La prima è che può funzionare in tre modalità operative. In modalità *legacy* Opteron esegue programmi standard per il Pentium senza biso-

gno di modifiche; in modalità di *compatibilità*, il sistema operativo funziona a 64 bit e può indirizzare  $2^{64}$  parole di memoria (ma i programmi applicativi girano a 32 bit); in modalità a 64 bit, infine, l'intera macchina funziona a 64 bit e tutti i programmi possono utilizzare lo spazio degli indirizzi di 64 bit. In quest'ultimo caso si possono affiancare nell'esecuzione programmi a 32 e a 64 bit, il che predisponde il sistema ad aggiornamenti futuri.

La seconda caratteristica chiave di Opteron è la sua attenzione al problema della larghezza di banda della memoria. Negli ultimi anni la velocità delle CPU è cresciuta molto più rapidamente rispetto alla larghezza di banda della memoria, rendendo i fallimenti di cache di secondo livello un grosso fattore di penalizzazione delle prestazioni. AMD ha integrato il controllore della memoria all'interno di Opteron di modo che possa girare alla stessa velocità del clock del processore e non più alla velocità del bus della memoria, migliorando le prestazioni di memoria. Il controllore può gestire otto DIMM di 4 GB ciascuna, per un totale di 32 GB di memoria per ogni Opteron. Nei sistemi Red Storm ogni Opteron ha solo 2-4 GB di memoria, però si può star sicuri che in futuro ne verrà aggiunta altra, visto che la memoria diventa sempre più economica. Un altro possibile miglioramento è l'adozione dei chip Opteron con due core, che raddoppierebbe la potenza grezza di calcolo.

Ogni Opteron ha un proprio processore di rete dedicato, Seastar, progettato su misura e prodotto da IBM. Si tratta di un componente critico perché quasi tutto il traffico di dati tra processori passa lungo la rete Seastar. Se non ci fossero questi chip progettati appositamente per garantire un'interconnessione ad altissima velocità, il sistema si troverebbe spesso impantanato nella grande quantità di dati.

I processori Opteron sono prodotti commerciali facilmente reperibili e pronti all'uso, ma le parti di assemblaggio di Red Storm sono costruite su misura. Ogni piastra di Red Storm (Figura 8.38) contiene quattro Opteron, 4 GB di RAM, quattro Seastar, un pro-

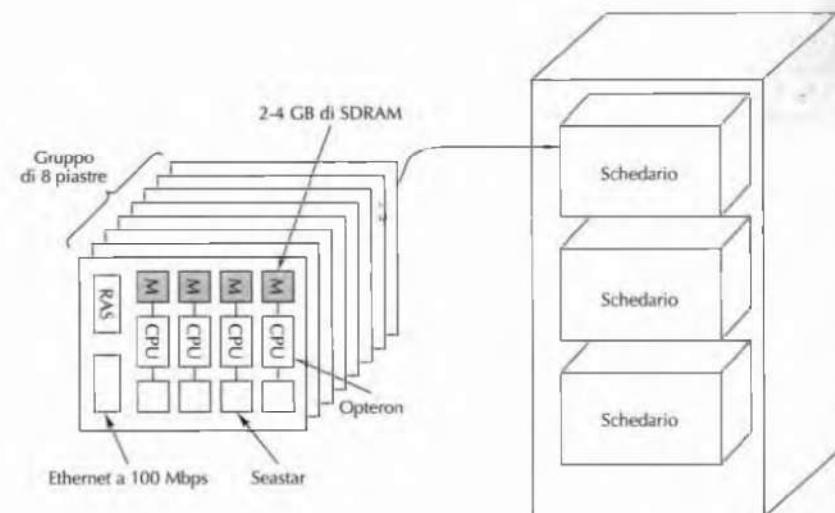


Figura 8.38 Assemblaggio dei componenti di un Red Storm.

cessore RAS (*Reliability-Availability-Service*, "affidabilità-reperibilità-servizio") e una scheda Ethernet a 100 Mbps.

Le piastre sono raggruppate a otto a otto e collegate a una scheda madre inserita in uno schedario. Ogni armadio contiene tre sportelli di schede per un totale di 96 Opteron, oltre ai necessari dispositivi di alimentazione e ventilazione. L'intero sistema è formato da 108 armadi per i nodi di calcolo, per un totale di 10.368 Opteron con 10 TB di SDRAM. Ogni CPU ha accesso esclusivamente alla propria SDRAM: non c'è alcuna memoria condivisa. La potenza di calcolo teorica del sistema è 41 teraflop/s.

L'interconnessione tra le CPU Opteron è realizzata tramite i router Seastar (uno per ogni CPU), collegati per mezzo di un toro di dimensioni  $27 \times 16 \times 24$  (a ogni intersezione della rete c'è un chip Seastar). Ogni Seastar ha sette collegamenti bidirezionali da 24 Gbps che sono diretti verso nord, est, sud, ovest, alto, basso e verso il processore Opteron. Il tempo di transito tra due punti adiacenti nella rete è 2  $\mu$ s, e bascano 5  $\mu$ s per attraversare tutto l'insieme dei nodi di calcolo. C'è anche una seconda rete Ethernet a 100 Mbps per servizio e manutenzione.

Oltre ai 108 armadi di calcolo, il sistema comprende 16 armadi per i processori di I/O e di servizio. Ciascuno di loro contiene 32 Opteron le cui CPU sono così suddivise: 256 per l'I/O, 256 di servizio. Lo spazio rimanente è occupato dai dischi, impostati come RAID 3 e RAID 5, e a ciascuno di loro è associato un disco di parità e uno di ricambio. Lo spazio totale su disco è di 240 TB, mentre la larghezza di banda complessiva dei dischi è di 50 GB/s.

Il sistema è suddiviso in due sezioni, una riservata e una no, collegate tramite commutatori e che possono quindi essere agganciate o sganciate meccanicamente. Entrambe le parti contengono sempre almeno 2688 CPU di calcolo. Le rimanenti 4992 CPU di calcolo possono essere commutate su una qualsiasi delle due sezioni, come illustrato nella Figura 8.39. Le 2688 CPU riservate sono Opteron con 4 GB di RAM, mentre le rimanenti hanno 2 GB. I processori di I/O e di servizio sono ripartiti nelle due sezioni.

Il tutto è ospitato all'interno di un edificio di  $2000 \text{ m}^3$  costruito per l'occasione. L'edificio è stato concepito in modo da poter ospitare un sistema Red Storm che possa essere potenziato in futuro fino a 30.000 CPU. I nodi di calcolo assorbono 1,6 MW di potenza, e un altro MW è assorbito dai dischi. Se si conteggiano anche i consumi dei sistemi di aerazione e raffreddamento, le apparecchiature assorbono complessivamente 3,5 MW.

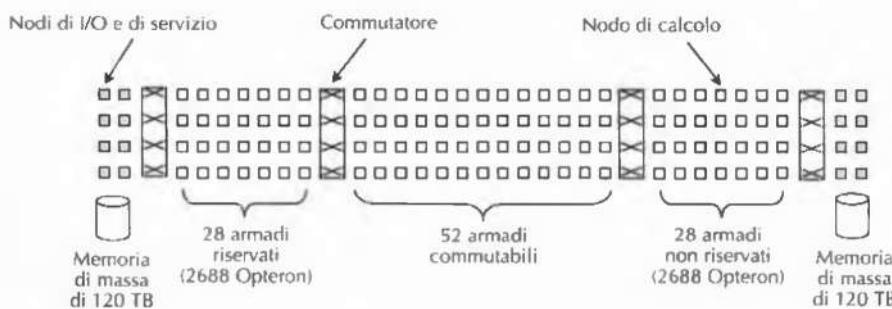


Figura 8.39 Visione dall'alto di Red Storm.

Il costo di hardware e software è stato di 90 milioni di dollari, l'edificio e il raffreddamento sono costati altri 9 milioni di dollari. Ovviamente parte dei costi è da considerarsi una tantum, in quanto dovuti alla progettazione e alla ingegnerizzazione. Se voleste ordinare una copia di Red Storm dovreste pensare a una cifra dell'ordine dei 60 milioni di dollari. Cray ha intenzione di produrre una versione più contenuta del sistema, che si chiamerà X3T.

I nodi di calcolo eseguono un kernel leggero che si chiama **cataamount**. I nodi di I/O e quelli di servizio eseguono la versione vanilla di Linux con l'aggiunta del supporto di MPI (ritorneremo sull'argomento nel corso del capitolo). I nodi RAS eseguono una versione di Linux ridotta al minimo. Red Storm può eseguire quasi tutto il software scritto per ASCI Red, compresi i programmi di allocazione per le CPU, gli scheduler, le librerie MPI, le librerie matematiche e anche i programmi applicativi.

Per un sistema così grande l'affidabilità è essenziale. Ogni piastra ha un processore RAS per la manutenzione, più altre funzionalità hardware speciali. L'obiettivo è quello di raggiungere un MTBF (*Mean Time Between Failures*), ovvero un tempo medio tra due guasti di 50 ore. ASCI Red poteva vantare un MTBF di 900 ore, ma subiva il crollo del sistema operativo circa ogni 40 ore. Benché il nuovo hardware sia molto più affidabile del vecchio, il software resta il punto debole del sistema.

Per ulteriori informazioni su Red Storm si faccia riferimento a (Brightwell et al., 2005).

### Confronto tra BlueGene/L e Red Storm

Red Storm e BlueGene/L sono per certi versi paragonabili, ma sotto altri aspetti restano molto differenti, perciò è interessante elencare i loro parametri fondamentali (Figura 8.40).

Le due macchine sono state costruite quasi contemporaneamente, perciò le loro differenze non sono attribuibili a livello tecnologico, ma alle diverse visioni dei progettisti e, in una certa misura, anche alle differenze tra i due costruttori, IBM e Cray. BlueGene/L è stato progettato sin dall'inizio come una macchina commerciale che IBM spera di vendere in gran numero alle aziende biotecnologiche, farmaceutiche, e così via. Red Storm è nato da un contratto speciale con Sandia, anche se Cray pensa di venderne comunque una versione ridotta.

La visione di IBM è chiara: combinare core già esistenti in chip ideati su misura che possono essere prodotti con bassi costi unitari. I chip girano a bassa velocità e sono facilmente componibili in gran numero mediante una rete di comunicazione di velocità modesta. La visione di Sandia è altrettanto chiara, ma diversa: usare una potente CPU a 64 bit già esistente, progettare su misura un chip per l'instradamento estremamente veloce e collegare al tutto tanta memoria, ottenendo così un nodo di gran lunga più potente di quello di BlueGene/L; dunque basterà un numero di nodi inferiore e la comunicazione tra di loro potrà essere più veloce.

Queste scelte influenzano direttamente l'assemblaggio dei componenti. IBM ha raggiunto una maggiore densità grazie all'inserimento di un processore e di un router all'interno di un solo chip: 1024 CPU/armadio. Viceversa, Sandia ha scelto di usare senza modifiche un chip di CPU già esistente e di equipaggiare ogni nodo con 2-4 GB di RAM, perciò è riuscita ad alloggiare solo 96 CPU in ogni armadio. In ragione di ciò, Red Storm occupa più superficie e assorbe più potenza di BlueGene/L.

Nell'esotico mondo dei laboratori di ricerca, il traguardo si misura in termini di prestazioni. Da questo punto di vista BlueGene/L vince per 71 a 41 Tflop/s, ma Red Storm è

Elemento	BlueGene/L	Red Storm
CPU	PowerPC a 32 bit	Opteron a 64 bit
Clock	700 MHz	2GHz
CPU di calcolo	65.536	10.368
CPU per piastra	32	4
CPU per armadio	1024	96
Armadi di calcolo	64	108
Teraflop/s	71	41
Memoria per CPU	512 MB	2-4 GB
Memoria complessiva	32 TB	10 TB
Router	PowerPC	Seastar
Numero di router	65.536	10.368
Interconnessione	Toro 3D $64 \times 32 \times 32$	Toro 3D $27 \times 16 \times 24$
Altre reti	Gigabit Ethernet	Fast Ethernet
Ripartibile	No	Sì
SO di calcolo	Su misura	Su misura
SO di I/O	Linux	Linux
Costruttore	IBM	Cray Research
Costo elevato	Sì	Sì

Figura 8.40 Confronto tra BlueGene/L e Red Storm.

espandibile e grazie all'aggiunta di altri 10.368 Opteron (per esempio passando ai chip con due core) potrebbe raggiungere gli 82 Tflop/s. IBM potrebbe rispondere spingendo un po' sul pedale del clock (una frequenza di 700 MHz non è proprio il massimo della tecnologia più avanzata). In breve, i supercomputer MPP non hanno ancora avvicinato i limiti fisici imposti dalla loro architettura e continueranno a crescere negli anni a venire.

#### 8.4.3 Cluster

La seconda categoria di multicomputer è costituita dai **cluster di computer** (Anderson et al., 1995; Martin et al., 1997). 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. La differenza tra un MPP e un cluster è analoga a quella tra un mainframe e un PC: entrambi hanno una CPU, un po' di RAM, dischi, un sistema operativo, ecc., solo che nei mainframe tutto è più veloce (a parte forse il sistema operativo). Ciononostante sembrano diversi e sono infatti usati e gestiti in modi differenti. Lo stesso accade agli MPP e ai cluster.

Storicamente, la caratteristica che ha sempre distinto gli MPP è l'interconnessione ad alta velocità, ma l'arrivo sul mercato di collegamenti veloci e pronti all'uso ha cominciato a colmare il divario. È probabile che i cluster spingano gli MPP verso nicchie sempre più ristrette, proprio come i PC hanno fatto con i mainframe, trasformandoli in oggetti esoterici

per specialisti. La nicchia principale per gli MPP è quella dei supercomputer molto costosi cui si richiedono le massime prestazioni, ma il cui prezzo è praticamente irraggiungibile.

Anche se esistono molti tipi di cluster, 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. Alle volte sono assemblati in modo ancora più compatto per ridurre le dimensioni e la lunghezza dei cavi. In genere le macchine sono omogenee e non hanno alcuna periferica, a parte la scheda di rete ed eventualmente alcuni dischi. Gordon Bell, il progettista del PDP-11 e di VAX, ha definito queste macchine **headless workstation** ("stazioni di lavoro senza capo", nel senso che non hanno un proprietario). Saremmo stati tentati di definirle COW senza capo, ma ci siamo trattenuti per il timore di sacrificare troppi bovini<sup>5</sup>.

I cluster decentralizzati sono formati da workstation e PC diffusi all'interno di un edificio o di un intero campus. Molte macchine restano inoperose per diverse ore al giorno, specie di notte, e sono spesso collegate tramite una LAN. Si tratta in genere di macchine eterogenee e dotate di un ricco equipaggiamento di periferiche, anche se un cluster con 1024 mouse non è per nulla migliore di uno che ne sia privo. Quel che più conta è che molti proprietari di computer sono assai gelosi delle loro macchine e guardano con sospetto gli astronomi che vogliono usarle per simulare il big bang. L'uso di workstation inoperose per formare un cluster richiede necessariamente la capacità di far migrare un compito da una macchina a un'altra quando il proprietario ne reclama l'uso. La migrazione dei compiti è possibile, ma complica il software.

I cluster sono spesso oggetti di piccole dimensioni, che vanno da una dozzina a 500 PC. È però possibile costruirne di molto grandi a partire da comuni PC. È proprio ciò che ha fatto Google e in un modo molto interessante, che ora esaminiamo.

#### Google

Google è un motore di ricerca molto in voga per la selezione d'informazioni in Internet. La sua popolarità è dovuta almeno in parte all'interfaccia semplice e al tempo di risposta rapido, ma il suo funzionamento è tutt'altro che semplice. Dal punto di vista di Google, il problema è trovare, indicizzare e memorizzare l'intero World Wide Web (più di 8 miliardi di pagine e 1 miliardo di immagini), riuscire a effettuare ricerche al suo interno in meno di mezzo secondo e gestire migliaia di interrogazioni al secondo che provengono da tutto il mondo, ventiquattrore al giorno. Per dirl'più deve restare sempre attivo, anche in caso di terremoti, black out elettrici, interruzioni delle linee di comunicazione, anomalie hardware e bachi software. Tutto ciò deve essere realizzato naturalmente al minor costo possibile. La costruzione di un clone di Google sicuramente non è un esercizio per il lettore.

Come funziona Google? Per cominciare, Google gestisce numerosi centri dati in diversi paesi del mondo. Questa soluzione serve non solo a fornire un backup dei dati nel caso un centro venisse inghiottito da un terremoto, ma anche a smistare i collegamenti: quando un utente digita [www.google.com](http://www.google.com) il suo IP viene esaminato e la stringa viene tradotta nell'indirizzo del centro più vicino, e a quell'indirizzo il software di navigazione specifica la richiesta.

<sup>5</sup> L'acronimo COW (*Cluster Of Workstations*) è anche la parola inglese che designa l'animale "mucca" (N.d.T.).

Ogni centro dati dispone di almeno una connessione in fibra ottica OC-48 (2,488 Gbps) con Internet, tramite cui riceve le interrogazioni e spedisce le risposte, oltre a una connessione di backup OC-12 (622 Mbps) verso un secondo fornitore di telecomunicazioni, nel caso il primo smettesse di funzionare. Ogni centro dati è equipaggiato con un gruppo di continuità e con generatori diesel di emergenza, per garantire la continuazione del servizio in caso di black out. Perciò Google è in grado di funzionare anche in concomitanza di una calamità naturale di grosse proporzioni, sebbene a prestazioni ridotte.

Per capire la scelta architettonica di Google, è utile descrivere brevemente l'elaborazione di un'interrogazione (*query*) che perviene al centro dati di pertinenza. Dopo l'arrivo (passo 1 nella Figura 8.41), il bilanciatore di carico instrada l'interrogazione verso uno dei tanti gestori d'interrogazioni (2) e da lì prosegue in parallelo al correttore ortografico (3) e al server di pubblicità (4). Quindi le diverse parole di ricerca sono cercate in parallelo nei server degli indici (5) che contengono un elemento per ogni parola presente nel Web. Ogni elemento contiene una lista di tutti i documenti (pagine web, file PDF, presentazioni PowerPoint, e così via) che contengono la parola, ordinati secondo il loro *page rank* (la posizione che ogni pagina detiene nella classifica globale del Web). Il *page rank* è calcolato con una formula complicata (e segreta), in cui giocano un ruolo predominante il numero di collegamenti entranti nella pagina e il rank delle pagine da cui provengono.

Al fine di incrementare le prestazioni, gli indici sono divisi in **shard** ("frammenti, schegge") sui quali è possibile effettuare la ricerca in parallelo. Almeno dal punto di vista concettuale, lo shard 1 contiene tutte le parole indicizzate, ciascuna seguita dagli ID dei pri-

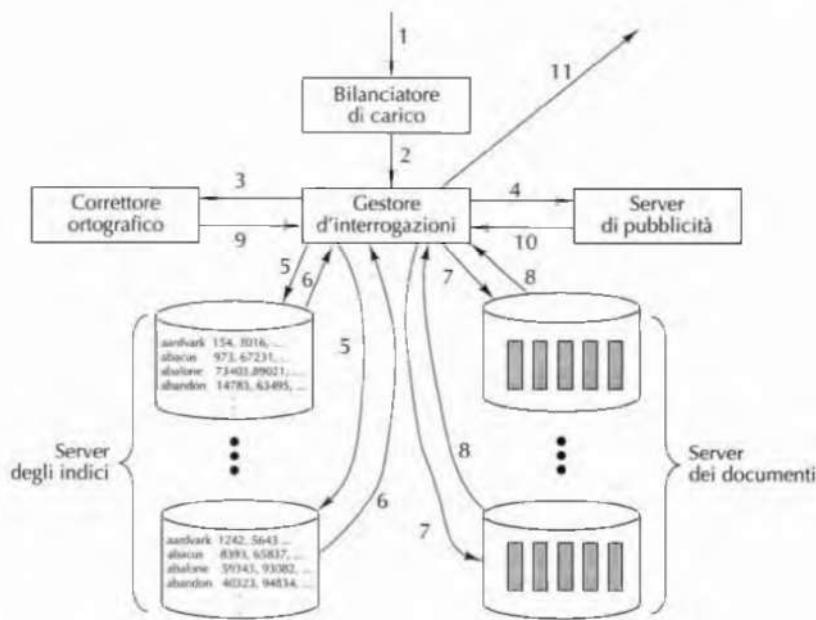


Figura 8.41 Gestione di un'interrogazione Google.

mi  $n$  documenti (nella classifica delle pagine) contenenti quella parola. Lo shard 2 contiene tutte le parole e gli ID degli  $n$  documenti successivi (nella classifica delle pagine) e così via. Quando il Web sarà cresciuto ulteriormente, si potrà dividere ogni shard in un insieme di shard di  $k$  parole ciascuno, per ottenere un maggiore parallelismo di ricerca.

I server degli indici restituiscono un insieme di identificativi di documento (6) che sono poi combinati secondo le relazioni booleane specificate dall'interrogazione. Per esempio, se si è ricercato + digitale + bradipo + danza, solo i documenti che appaiono in tutti e tre gli insiemi sono usati al passo successivo (7), in cui si accede ai documenti stessi per ricavarne i titoli, gli URL e gli estratti di testo che circondano i termini ricercati. I server dei documenti contengono molte copie dell'intero Web in ciascun centro dati (al momento si parla di centinaia di terabyte). I documenti sono suddivisi a loro volta in shard per aumentare il parallelismo di ricerca. Anche se un'interrogazione non richiede la lettura dell'intero Web (e neanche delle decine di terabyte dei server degli indici), normalmente comporta comunque un'elaborazione dell'ordine dei 100 MB.

I risultati vengono quindi rispediti al gestore delle interrogazioni (8) che riordina le pagine secondo il loro *page rank*. Se sono stati rilevati potenziali errori di digitazione (9) questi vengono resi noti e infine vengono aggiunti annunci pubblicitari attinenti alla ricerca (10). La vendita di specifici termini di ricerca (per esempio "hotel" o "videocamera") cui associare annunci pubblicitari è la sorgente dei proventi di Google. Infine, i risultati sono formattati in HTML (*HyperText Markup Language*) e spediti all'utente richiedente sotto forma di pagina web.

A questo punto possiamo esaminare l'architettura di Google. La maggior parte delle aziende, di fronte alla necessità di gestire un database enorme, comprerebbe l'attrezzatura più grande, veloce e affidabile che si trova sul mercato. Google ha fatto l'esatto opposto: ha comprato un gran numero di PC economici dalle prestazioni modeste e li ha usati per costruire il più grande cluster del mondo, fatto solo di componenti già disponibili sul mercato. Il principio guida alla base di questa decisione è semplice: ottimizzare il rapporto costi/prestazioni.

La logica della decisione è di natura economica: i PC sono a buon mercato, a differenza dei server di fascia alta e, soprattutto, dei multiprocessori di grandi dimensioni. Un server di fascia alta potrebbe offrire prestazioni due o tre volte superiori a quelle di un desktop PC di fascia media, a un prezzo che è in genere dalle cinque alle dieci volte maggiore, perciò la sua scelta non è conveniente.

Ovviamente i PC economici sono soggetti a più anomalie dei server che si trovano ai vertici della gamma, ma anche questi ultimi incappano alle volte in qualche malfunzionamento. Sin dall'inizio Google è stato progettato per funzionare su hardware fallibile, indipendentemente dal tipo di attrezzatura utilizzata. Una volta scritto software tollerante agli errori, non importa molto se il tasso di errore è dello 0,5% o del 2% all'anno, le anomalie vanno trattate in ogni caso. Nell'esperienza di Google, il 2% dei PC mostra almeno un'anomalia nel corso di un anno; più della metà dei malfunzionamenti sono dovuti a dischi difettosi, mentre i restanti sono attribuibili prevalentemente agli alimentatori elettrici e ai chip di RAM. Le CPU già collaudate non incappano mai in errori. In realtà la maggior parte dei crolli di sistema non è dovuta per nulla all'hardware, ma al software. La prima reazione a un crollo è riavviare il sistema, il che spesso risolve il problema (l'equivalente elettronico del consiglio del dottore: "prenda un'aspirina e vada a letto").

Un tipico PC moderno di Google dispone di un Pentium a 2 GHz, 512 MB di RAM e un disco di 80 GB circa, il tipo di calcolatore che comprerebbe una nonna per controllare la posta elettronica, di tanto in tanto. L'unica componente aggiuntiva è un chip Ethernet molto economico e per niente sofisticato. I PC sono impilati a gruppi di 40 in scaffali alti poco più di mezzo metro. Ogni scaffale ospita due gruppi, uno sul davanti, uno sul retro, per un totale di 80 PC. I PC dello stesso scaffale sono collegati tramite una Ethernet, il cui commutatore è contenuto nello scaffale. Anche gli scaffali di un centro dati sono collegati tramite Ethernet, con due commutatori ridondanti per far fronte alle anomalie di commutazione.

La Figura 8.42 riporta lo schema di un tipico centro dati di Google. La fibra OC-48 ad alta larghezza di banda, per le interrogazioni in entrata, viene instradata verso i commutatori Ethernet a 128 porte, come anche la fibra OC-12 di backup. Le fibre in entrata usano schede speciali e non impegnano alcuna delle 128 porte Ethernet. Ogni scaffale ha quattro collegamenti Ethernet, due verso il commutatore di sinistra, due verso quello di destra. Mediante questo schema il sistema può sopravvivere alla rottura di uno dei due commutatori. Poiché ogni scaffale ha quattro connessioni con i commutatori (due provenienti dai 40 PC sul davanti, due dai 40 sul retro), ci vogliono quattro interruzioni di collegamenti,

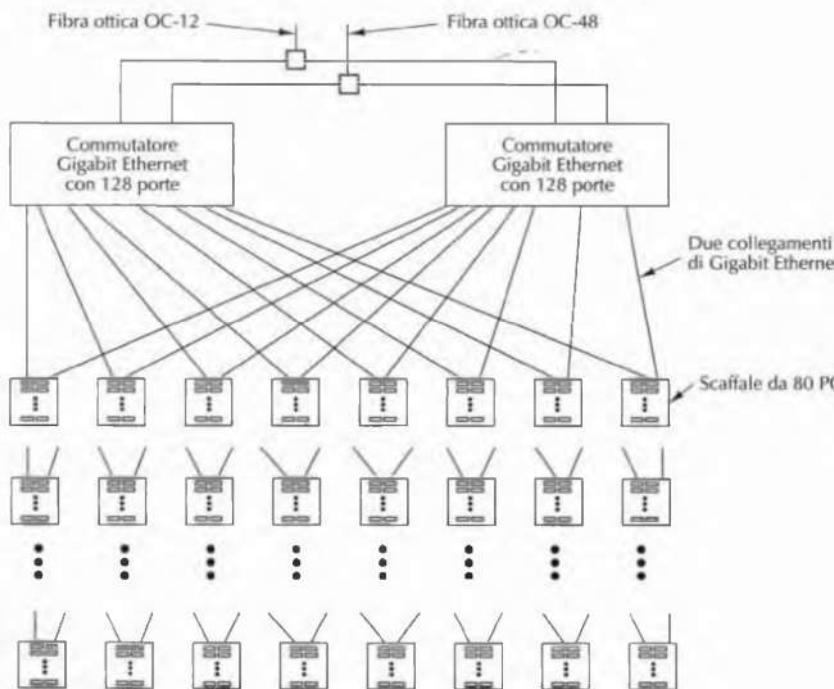


Figura 8.42 Un tipico cluster di Google.

oppure due interruzioni e una rottura di un commutatore, per disconnettere uno scaffale. I due commutatori a 128 porte possono connettere al massimo 64 scaffali con quattro collegamenti ciascuno. Dato che ogni scaffale ospita 80 PC, un centro dati può contenere fino a 5120 PC. Tuttavia gli scaffali possono contenere un numero di PC diverso da 80 e i commutatori possono avere più o meno di 128 porte. I numeri che abbiamo fornito sono solo quelli di un tipico cluster di Google.

Anche la densità di potenza è un fattore critico. Un PC comune assorbe 120 watt, uno scaffale 10 kilowatt. Uno scaffale ha bisogno di 3 m<sup>2</sup> perché ci sia spazio sufficiente alla manutenzione (installazione e rimozione di PC) e al sistema di condizionamento d'aria. Questi parametri inducono una densità di potenza maggiore di 3000 watt/m<sup>2</sup>, mentre gran parte dei centri dati sono progettati per 600-1200 watt/m<sup>2</sup>, dunque si rendono necessarie speciali misure per il raffreddamento degli scaffali.

Google ha acquisito tre conoscenze a proposito della gestione di server web di dimensioni enormi, che vale la pena ripetere.

1. I componenti sono soggetti ad anomalie, perciò bisogna prenderle in considerazione a priori.
2. Duplicare tutti i componenti aumenta la produttività e la disponibilità di risorse.
3. Conviene ottimizzare il rapporto prezzo/prestazioni.

Il primo punto afferma che bisogna progettare software tollerante agli errori. Anche se si dispone dell'attrezzatura migliore in assoluto, la presenza di un numero enorme di componenti implica necessariamente qualche malfunzionamento e il software deve essere pronto a gestire la situazione. Non importa se si verificano 1 o 2 anomalie alla settimana, il software deve essere in grado di fronteggiare i malfunzionamenti di sistemi di queste dimensioni.

Il secondo punto specifica che l'hardware e il software devono essere molto ridondanti. Così facendo non solo si migliorano le proprietà di tolleranza agli errori, ma anche la produttività (il *throughput*, cioè il volume di dati elaborati per unità di tempo). Nel caso di Google, PC, dischi, cavi e commutatori sono tutti duplicati più volte. Inoltre, anche all'interno dello stesso centro dati esistono molteplici copie di indici e documenti.

Il terzo punto è una conseguenza dei primi due. Se un sistema è stato progettato correttamente per trattare le anomalie, è un errore comprare componenti costosi come le unità RAID con dischi SCSI. Anche queste esibirebbero anomalie, e spendere 10 volte di più per dimezzare il tasso di errore è una cattiva idea. È meglio comprare hardware in quantità 10 volte maggiore e trattare gli errori quando si presentano. Inoltre, la presenza di hardware duplicato garantisce prestazioni migliori quando tutto funziona correttamente.

Facciamo riferimento a (Barroso et al., 2003; Ghemawat et al., 2003) per maggiori informazioni su Google.

#### 8.4.4 Software di comunicazione per multicomputer

La programmazione di un multicomputer richiede software speciale, spesso di libreria, per la gestione della comunicazione e della sincronizzazione tra processi. In questo paragrafo speniamo qualche parola in proposito. Gli MPP e i cluster possono eseguire per buona parte gli stessi pacchetti software, perciò le applicazioni possono essere portate facilmente da una piattaforma all'altra.

I sistemi a scambio di messaggi eseguono due o più processi in modo indipendente l'uno dall'altro. Per esempio, un processo potrebbe produrre alcuni dati e uno o più altri processi potrebbero consumarli. Non c'è alcuna garanzia che il destinatario (o i destinatari) sia pronto a trattare i dati quando questi sono disponibili presso il mittente, dal momento che ciascuno di loro esegue il proprio programma.

Molti dei sistemi a scambio di messaggi forniscono le primitive `send` e `receive` (spesso sotto forma di chiamate di libreria), ma ci sono possibili ulteriori possibilità. Le tre varianti principali sono:

1. scambio sincrono di messaggi
2. scambio di messaggi bufferizzato
3. scambio di messaggi non bloccante.

Lo **scambio sincrono di messaggi** prevede che, se il mittente esegue una `send` e il destinatario non ha ancora eseguito una `receive`, il mittente si blocca e resta sospeso finché il destinatario non esegue la `receive`, e solo a questo punto il messaggio viene copiato. Alla restituzione del controllo al mittente dopo la chiamata, questi è sicuro che il messaggio è stato spedito e ricevuto correttamente. Questo metodo ha una semantica semplice e non richiede l'uso di buffer. Presenta però la forte limitazione di sospendere il mittente fino alla consegna del messaggio al destinatario e della rispettiva ricevuta al mittente.

Nello **scambio di messaggi bufferizzato**, se un messaggio viene spedito prima che il destinatario sia pronto, il messaggio viene posto in un buffer da qualche parte, per esempio in una mailslot, finché il destinatario non lo riceve. In tal modo il mittente può proseguire la sua esecuzione dopo una `send`, anche se il destinatario è occupato in un'altra attività. Dal momento che il messaggio è stato effettivamente spedito, il mittente è libero di riutilizzare il buffer di messaggi immediatamente. Questo schema riduce il tempo di attesa per il mittente; infatti, il mittente è libero di proseguire l'esecuzione non appena il sistema abbia spedito il messaggio. Tuttavia, il mittente non ha alcuna garanzia che il messaggio sia stato ricevuto correttamente. Anche in presenza di comunicazione affidabile, potrebbe sempre accadere che l'esecuzione del destinatario si sia interrotta prima di aver ricevuto il messaggio.

Lo **scambio di messaggi non bloccante** consente al mittente di continuare la propria esecuzione dopo la chiamata. La libreria si limita a comunicare al sistema operativo di effettuare la chiamata quando ha tempo di farlo, di conseguenza il mittente non viene bloccato per nulla. Lo svantaggio di questo metodo è che, dopo una `send`, il mittente non può riutilizzare il buffer di messaggi, perché il messaggio precedente potrebbe non essere ancora stato spedito. Deve cercare di sapere in qualche modo se può già riutilizzare il buffer: un'idea è fare *polling* sul sistema, un'altra è prevedere l'invio di un interrupt quando il buffer torna disponibile. Nessuna delle due possibilità semplifica la scrittura del software.

Esaminiamo ora un sistema a scambio di messaggi molto diffuso e disponibile su diversi multicomputer: MPI.

### **MPI – Interfaccia a scambio di messaggi**

Per un certo numero di anni, PVM (*Parallel Virtual Machine*) è stato il software di comunicazione più diffuso tra i multicomputer (Geist et al., 1994; Sunderram, 1990), ma di recente è stato rimpiazzato del tutto da MPI (*Message-Passing Interface*), molto più articolato e com-

plesso, che fornisce un maggior numero di chiamate di libreria, più opzioni di scelta e molti più parametri per ogni chiamata. La versione originale di MPI, nota come MPI-1, è stata ampliata nel 1997 nella versione MPI-2. Di seguito daremo un'introduzione molto rapida a MPI-1 (che contiene tutti gli elementi basilari), quindi diremo qualcosa circa le aggiunte di MPI-2. Maggiori informazioni sono reperibili su MPI in (Gropp et al., 1994; Snir et al., 1996).

A differenza di PVM, MPI-1 non si occupa della creazione o gestione dei processi e spetta quindi all'utente creare i processi con le chiamate di sistema locali. Una volta creati, i processi sono organizzati in gruppi statici che non possono essere modificati. MPI lavora a livello di questi gruppi.

MPI si basa su quattro concetti principali: comunicatori, tipi di dati dei messaggi, operazioni di comunicazione e topologie virtuali. Un **comunicatore** è un gruppo di processi unito a un contesto. Un contesto è un'etichetta usata per identificare qualcosa, per esempio una fase dell'esecuzione. All'atto della spedizione o della ricezione di messaggi, il contesto può essere usato per evitare che messaggi correlati interferiscano gli uni con gli altri.

I messaggi hanno un tipo: ne sono supportati molti, tra cui i caratteri, gli interi corti, regolari o lunghi, i numeri in virgola mobile a precisione singola o doppia, e così via. È anche possibile costruire tipi di dati derivati da questi.

MPI supporta un insieme di operazioni di comunicazione molto vasto. L'operazione fondamentale per spedire un messaggio è la seguente:

```
MPI_Send(buffer, numero, tipo_dati, destinazione, etichetta,
          comunicatore)
```

Questa chiamata invia alla destinazione un buffer con `numero` oggetti del tipo di dati specificato. Il campo `etichetta` serve al destinatario perché questi potrebbe specificare di voler ricevere solo messaggi con una certa etichetta. L'ultimo campo specifica il gruppo di processi cui appartiene il destinatario (il campo `destinazione` non è altro che un indice nella lista dei processi del gruppo specificato). La chiamata per la ricezione del messaggio è:

```
MPI_Recv(&buffer, numero, tipo_dati, sorgente, etichetta,
         comunicatore, &stato)
```

che specifica il tipo di messaggio atteso dal destinatario, la sorgente di provenienza e l'etichetta.

MPI supporta quattro modalità basilari di comunicazione. La modalità 1 è sincrona: il mittente non può cominciare a spedire finché il destinatario non abbia invocato `MPI_Recv`. La modalità 2 è bufferizzata, per cui la limitazione non esiste più. La modalità 3 è quella standard, cioè dipende dall'implementazione e può essere sincrona o bufferizzata. La modalità 4 è la modalità *pronto*, secondo cui il mittente suppone che il destinatario sia pronto alla comunicazione (come nel caso sincrono) ma non viene effettuato alcun reale controllo. Ciascuna di queste primitive è disponibile in versione bloccante e non bloccante, per un totale di otto primitive. La ricezione ha solo due varianti: bloccante e non bloccante.

MPI fornisce il supporto per la comunicazione collettiva (tra gruppi di processi), comprese le comunicazioni broadcast, *scatter/gather*, a scambio totale, ad aggregazione e a barriera<sup>6</sup>.

In tutte le forme di comunicazione collettiva, i processi del gruppo devono effettuare la chiamata e usare argomenti compatibili, altrimenti si verifica un errore. Una forma co-

<sup>6</sup> *Scatter* e *gather* sono forme di I/O in cui dati contigui vengono trasferiti in locazioni non contigue (N.d.R.).

mune di comunicazione collettiva si ha quando i processi sono organizzati ad albero e alcuni valori vengono fatti propagare dalle foglie verso la radice; a ogni passo subiscono un'elaborazione, per esempio la somma di un certo valore o la selezione del valore minimo.

Il quarto concetto alla base di MPI è la **topologia virtuale**, che permette di specificare se i processi sono organizzati ad albero, ad anello, a griglia, in un toro o con un'altra topologia. Questo concetto mette a disposizione un modo per denominare i percorsi di comunicazione e dunque la facilità.

MPI-2 fornisce inoltre i processi dinamici, l'accesso alla memoria distante, la comunicazione collettiva non bloccante, il supporto di I/O scalabile, l'elaborazione in tempo reale e molte altre caratteristiche che vanno oltre lo scopo di questa trattazione. Nella comunità scientifica si è svolta per svariati anni una guerra per la supremazia tra MPI e PVM. I fautori di PVM sostenevano che fosse più semplice da imparare e più facile da usare. I fautori di MPI sostenevano fosse più ricco in dotazione e sottolineavano il fatto che costituiva uno standard formale regolamentato da un documento di definizione ufficiale, a sua volta redatto da parte di una commissione per la standardizzazione. Su questo punto concordavano anche i sostenitori di PVM, ma rispondevano che l'assenza di un apparato burocratico per la piena standardizzazione non costituiva necessariamente un aspetto negativo. Dopo tutto questi batti e ribatti, MPI ha sopravanzato definitivamente PVM nella disputa.

#### 8.4.5 Scheduling

MPI fornisce ai programmati uno strumento semplice per la creazione di compiti (*job*) il cui svolgimento richiede molteplici CPU e la cui esecuzione impiega un tempo consistente. In presenza di numerose richieste indipendenti da parte di utenti diversi, ciascuna riguardante un certo numero di CPU e lunga un certo lasso di tempo, si rende necessaria la programmazione delle attività del cluster (*scheduling*) per stabilire quale compito deve trovarsi in esecuzione in un determinato momento.

Secondo il modello più semplice, lo scheduler dei compiti richiede che ciascun compito specifichi il numero di CPU di cui ha bisogno. Dopodiché i compiti vengono eseguiti in ordine FIFO, come illustrato nella Figura 8.43(a). Questo modello prevede un controllo, dopo la partenza di un compito, per verificare se sono disponibili abbastanza CPU per avviare il compito successivo nella coda. In caso di esito affermativo il compito può partire, altrimenti il sistema deve aspettare che si liberino altre CPU. Come nota a margine, nella figura abbiamo suggerito la presenza di otto CPU nel cluster, ma potrebbe anche trattarsi di un cluster con 128 CPU allocate in unità di 16 (risultando in otto gruppi di CPU) o di qualche altra combinazione.

Un algoritmo migliore evita il bloccaggio in testa alla coda, perché salta i compiti che sono incompatibili e, scandendo la coda in ordine FIFO, sceglie il primo che è compatibile con le CPU disponibili. Questo algoritmo produce il risultato della Figura 8.43(b).

Un algoritmo di scheduling ancora più sofisticato richiede che ogni compito accodato dichiari la sua forma, ossia il numero di CPU che impegnă e il numero di minuti che impiega per l'esecuzione. Grazie a questa informazione, lo scheduler dei compiti può cercare una configurazione che riempia al meglio il rettangolo numero di CPU-tempo. Questa operazione si dice *tiling* ("rivestimento con piastrelle") ed è molto efficace in quelle situazioni in cui le richieste dei compiti pervengono durante il giorno e la loro esecuzione è prevista durante la notte;

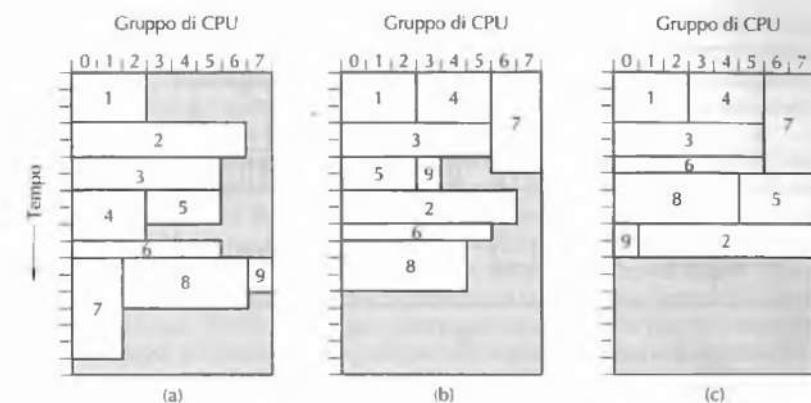


Figura 8.43 Scheduling di un cluster. (a) FIFO. (b) Senza bloccaggio in testa alla coda. (c) Tiling. Le parti ombreggiate indicano le CPU inattive.

in questo modo lo scheduler dei compiti ottiene tutte le informazioni in anticipo e può far eseguire i compiti secondo l'ordine ottimale, come mostrato nella Figura 8.43(c).

#### 8.4.6 Memoria condivisa a livello applicativo

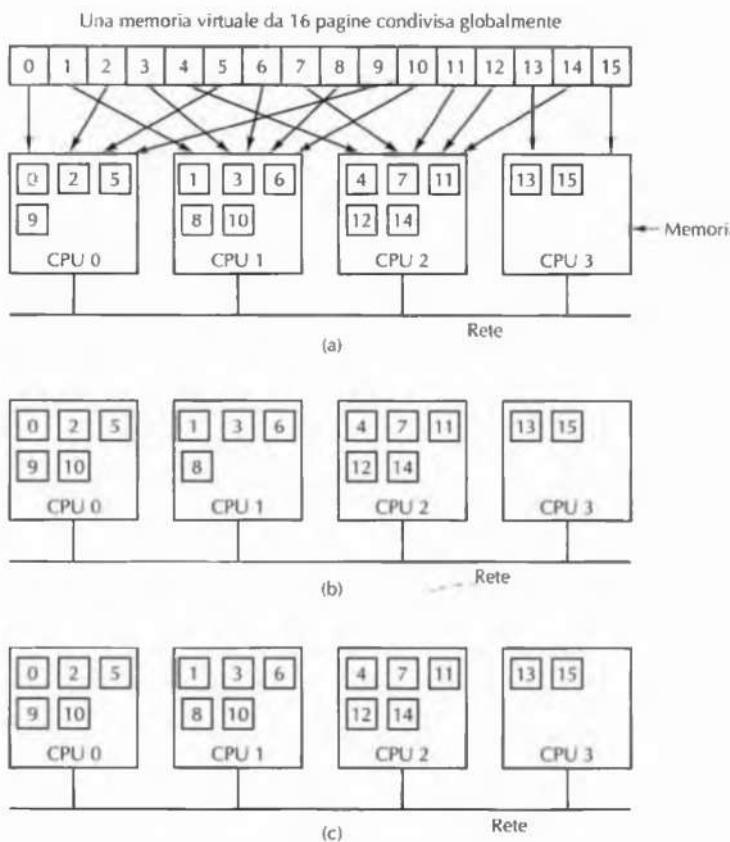
Gli esempi proposti dovrebbero aver dimostrato come i multicomputer raggiungano dimensioni molto maggiori rispetto ai multiprocessori. Questo fatto ha portato allo sviluppo di sistemi a scambio di messaggi come MPI, eppure molti programmati non amano questo modello e vorrebbero poter lavorare con l'illusione di una memoria condivisa, anche se non esiste davvero. Riuscire a soddisfare questa richiesta vorrebbe dire mettere insieme il meglio di entrambi i paradigmi: hardware economico in quantità enormi, e programmazione semplice. Questo obiettivo è il Santo Graal del calcolo parallelo.

Molti ricercatori sono giunti alla conclusione che la condivisione di memoria non funziona bene su sistemi molto grandi se implementata a livello di architettura, ma che sono possibili altri modi per raggiungere lo stesso scopo. Già dalla Figura 8.19 sappiamo che la memoria condivisa può essere introdotta ad altri livelli, nei paragrafi seguenti vedremo com'è possibile introdurre la memoria condivisa nel modello di programmazione di un multicomputer, benché non esista a livello hardware.

#### Memoria condivisa distribuita

Un tipo di sistema a memoria condivisa a livello applicativo è quello dei sistemi basati su pagine, chiamati spesso DSM (*Distributed Shared Memory*). L'idea è semplice: un insieme di CPU di un multicomputer condivide uno spazio di indirizzi virtuali paginato. Nella versione più semplice, ogni pagina è contenuta nella RAM di una sola CPU. La Figura 8.44(a) mostra uno spazio di indirizzi virtuali condiviso e formato da 16 pagine, disseminate tra quattro CPU.

Quando una CPU fa riferimento a una pagina che si trova nella sua RAM locale, la lettura o scrittura si svolge senza alcun ritardo, ma il riferimento a una pagina in una memoria



**Figura 8.44** Spazio degli indirizzi virtuali con 16 pagine distribuite su quattro nodi di un multicomputer. (a) Situazione iniziale. (b) Dopo un riferimento alla pagina 10 da parte della CPU 0. (c) Dopo un riferimento alla pagina 10 da parte della CPU 1 (nell'ipotesi di una pagina di sola lettura).

distante provoca un errore di pagina. La differenza qui è che la pagina mancante non viene caricata dal disco, bensì il sistema *runtime* o il sistema operativo spediscono un messaggio al nodo che la detiene affinché se ne privi e la spedisca al richiedente. Una volta giunta a destinazione, la pagina viene mappata nella RAM locale e l'istruzione che ha provocato l'errore può essere eseguita, proprio come succede dopo un normale errore di pagina. La Figura 8.44(b) illustra la situazione dopo che la CPU 0 ha causato un errore sulla pagina 10, che viene trasferita conseguentemente dalla CPU 1 alla CPU 0.

Questa semplice idea, implementata per la prima volta in IVY (Li e Hudak, 1986, 1989), consente l'utilizzo in un multicomputer di una memoria pienamente condivisa a consistenza sequenziale, anche se sono possibili ulteriori ottimizzazioni per migliorarne le prestazioni.

La prima variante di IVY ammette che le pagine contrassegnate in sola lettura siano presenti simultaneamente in più nodi. Perciò quando si verifica un errore di pagina, viene spedita una copia della pagina alla macchina richiedente, ma la versione originale resta al suo posto perché non c'è pericolo di conflitto. La Figura 8.44(c) mostra la situazione in cui due CPU condividono una pagina in sola lettura (la pagina 10).

Nonostante questa ottimizzazione, le prestazioni sono spesso inaccettabili, specie nei casi in cui un processo scrive qualche parola all'inizio di una pagina mentre un altro processo sta scrivendo alcune parole in fondo alla stessa pagina. L'esistenza di una sola copia della pagina costringe la sua spedizione ripetuta avanti e indietro, una condizione nota come **falsa condivisione**. Questo problema può essere affrontato in modi diversi. Nel sistema Treadmarks, per esempio, si abbandona la consistenza sequenziale a favore della consistenza dopo rilascio (Amza, 1996). Una pagina potenzialmente scrivibile può trovarsi contemporaneamente presso più nodi, ma un processo che voglia effettuare una scrittura al suo interno deve prima segnalare le proprie intenzioni eseguendo un'operazione di *acquire*. A questo punto vengono invalidate tutte le copie della pagina tranne la più recente e non è possibile effettuare nuove copie finché non viene eseguita la *release* corrispondente; solo allora la pagina può tornare a essere condivisa.

Una seconda ottimizzazione implementata in Treadmarks è la dichiarazione iniziale in modalità di sola lettura delle pagine scrivibili. All'atto della prima scrittura in una pagina, viene sollevato l'errore di protezione e il sistema crea una copia della pagina chiamata **pagina gemella**. Quindi la pagina originale viene classificata con modalità di lettura/scrittura e le scritture successive possono procedere appieno ritmo. Se più tardi si verifica un errore di pagina e bisogna spedire la pagina al richiedente, viene effettuato un confronto parola per parola tra la pagina corrente e la sua gemella e vengono spedite solo le parole che sono effettivamente cambiate, riducendo così la dimensione del messaggio.

Il verificarsi di un errore di pagina pone il problema di localizzare la pagina mancante. Sono possibili svariate soluzioni, incluse quelle adottate nelle macchine NUMA e COMA, come le directory (basate sui nodi di appartenenza). Infatti, molte soluzioni usate nel contesto DSM sono applicabili anche ai contesti NUMA e COMA, perché DSM non è altro che un'implementazione software di NUMA o COMA, in cui ogni pagina è trattata come una linea di cache.

I sistemi DSM sono oggetto di intensa ricerca. Tra i sistemi più interessanti annoveriamo CASHMERE (Kontothanassis et al., 1997; Stets et al., 1997), CRL (Johnson et al., 1995), Shasta (Scales et al., 1996) e Treadmarks (Amza, 1996; Lu et al., 1997).

### Linda

I sistemi DSM basati su pagine come IVY e Treadmarks usano l'hardware della MMU per intercettare gli accessi alle pagine mancanti. Benché la spedizione delle differenze tra pagine migliori le prestazioni rispetto all'invio delle pagine intere, rimane il fatto che le pagine sono unità inadatte alla condivisione. Per questo motivo sono stati tentati approcci alternativi al problema.

Linda, uno di questi tentativi, fornisce la gestione di processi su macchine diverse con il supporto di una memoria condivisa distribuita altamente strutturata (Carriero e Gelernter, 1989). L'accesso alla memoria avviene tramite un piccolo insieme di operazioni primitive che possono essere aggiunte ai linguaggi esistenti (C o FORTRAN) per costituire linguaggi paralleli, in questo caso C-Linda e FORTRAN-Linda.

```
("abc"; 2; 5)
("matrice"; i; 6; 3,14)
("famiglia"; "è sorella"; Carolina; Eleonora)
```

Figura 8.45 Tre tuple di Linda.

Il concetto unificatore di Linda è l'esistenza di uno **spazio di tuple** astratto, globale al sistema e accessibile da ogni suo processo. Lo spazio di tuple è simile a una memoria globale condivisa, dotata in più di una struttura intrinseca. Lo spazio delle tuple contiene un certo numero di **tuple**, ciascuna costituita a sua volta da uno o più campi. Nel caso di C-Linda, i tipi dei campi comprendono gli interi, gli interi lunghi e i numeri in virgola mobile, oltre ai tipi composti quali gli array (compresa le stringhe) e le strutture (ma non altre tuple). La Figura 8.45 mostra tre esempi di tuple.

Sono disponibili quattro operazioni sulle tuple. La prima, **out**, inserisce una tupla nello spazio delle tuple. Per esempio

```
out("abc"; 2; 5);
```

inserisce la tupla ("abc", 2, 5) nello spazio delle tuple. I campi di **out** sono in genere costanti, variabili o espressioni, come in

```
out("matrice"; i; j; 3,14);
```

che produce una tupla con quattro campi, dove il secondo e il terzo sono specificati dal valore corrente delle variabili *i* e *j*.

Le tuple vengono prelevate dallo spazio delle tuple tramite la primitiva **in**; il loro indirizzamento è per contenuto e non per nome o per indirizzo. I campi di **in** possono essere espressioni o parametri formali. Si consideri, per esempio

```
in("abc"; 2; ? i);
```

Questa operazione "cerca" nello spazio delle tuple una tupla formata dalla stringa "abc", dall'intero 2 e da un terzo campo che contiene un intero qualsiasi (ipotizziamo *i* sia di tipo intero). Se viene trovata, la tupla è rimossa dallo spazio delle tuple e viene assegnato alla variabile *i* il valore del terzo campo della tupla. Le operazioni di ritrovamento e rimozione sono atomiche, perciò se due processi eseguono simultaneamente la stessa operazione **in**, solo uno di loro avrà successo, a meno che non esistano due o più tuple corrispondenti ai criteri di ricerca. Lo spazio delle tuple potrebbe contenere addirittura copie multiple della stessa tupla.

L'algoritmo di ricerca usato da **in** è semplice. Concettualmente si procede così: i campi della primitiva **in**, che ne costituiscono il **template** (cioè lo schema, il modello), vengono confrontati con i campi corrispondenti di tutte le tuple dello spazio. Si verifica una corrispondenza se risultano soddisfatte le tre condizioni:

1. il template e le tuple hanno lo stesso numero di campi
2. i tipi dei campi corrispondenti sono uguali
3. ogni costante o variabile del template si accorda al rispettivo campo della tupla.

I parametri formali, indicati dal punto interrogativo seguito da un nome di variabile o da un tipo, non partecipano al confronto (fatta eccezione per quello di tipo); cionononostante, quelli che contengono una variabile vengono assegnati al termine di una ricerca fruttuosa.

Se non si trova alcuna tupla corrispondente ai criteri, il processo chiamante resta sospeso finché un altro processo non inserisce la tupla richiesta, al che il primo viene automaticamente risvegliato e gli viene consegnata la nuova tupla. Il bloccaggio e lo sbloccaggio automatico dei processi garantisce che, se un processo sta per produrre una tupla e un altro sta per prelevarla, non importa chi dei due comincia per primo.

Oltre a **in** e **out**, Linda fornisce anche la primitiva **read** che opera analogamente a **in**, ma non rimuove la tupla dallo spazio. Esiste anche la primitiva **eval** che richiede la valutazione parallela dei suoi parametri e l'inserzione della tupla risultante nello spazio delle tuple. Questo meccanismo permette di svolgere qualsiasi tipo di calcolo ed è il modo in cui sono creati i processi paralleli in Linda.

Un paradigma di programmazione tipico di Linda è il **replicated worker model** ("modello del lavoratore replicato"). Questo modello si basa sull'idea di una **task bag** ("borsa dei lavori") piena di compiti da svolgere. Il processo principale inizia la sua esecuzione con un ciclo che contiene

```
out("task-bag", compito);
```

che a ogni iterazione inserisce nello spazio delle tuple la descrizione di un compito diverso. Ogni lavoratore comincia con il prelevare una tupla di descrizione di un compito con

```
in("task-bag", ?compito);
```

e poi inizia a svolgerlo; al suo compimento ne preleva un altro e così via. È anche possibile che vengano inseriti nella task bag nuovi lavori durante l'esecuzione. Così facendo, il lavoro viene suddiviso dinamicamente tra i lavoratori e ciascuno di loro resta costantemente impegnato; il tutto funziona con una produzione e uno scambio d'informazioni accessorie relativamente contenuti.

Esistono varie implementazioni di Linda per sistemi multicomputer e tutte si trovano ad affrontare la questione chiave della distribuzione delle tuple tra le macchine e del modo di localizzarle quando richieste. Tra le diverse possibilità ci sono la tecnica broadcast e le directory. Anche la duplicazione è una questione critica da affrontare; per un'analisi di tutti questi aspetti si veda (Bjornson, 1993).

### Orca

Un approccio un po' diverso per la condivisione di memoria a livello applicativo nei multicomputer è l'uso di oggetti veri e propri come unità della condivisione invece di semplici tuple. Gli oggetti sono costituiti da uno stato interno (nascosto) unito ai metodi per agire su questo stato. La scelta di non permettere ai programmati di accedere direttamente allo stato degli oggetti offre molte modalità di condivisione tra macchine che non hanno una memoria fisica condivisa.

Un sistema basato su oggetti che fornisce l'illusione di una memoria condivisa nei multicomputer è Orca (Bal, 1991; Bal et al., 1992; Bal e Tanenbaum, 1988), un linguaggio di programmazione tradizionale (basato su Modula 2) con l'aggiunta di due nuove caratteristiche: gli oggetti e la capacità di creare nuovi processi. Un oggetto di Orca è un tipo di dati astratto, analogo a un oggetto di Java o a un package di Ada, che incapsula le strutture dati interne e i metodi scritti dall'utente, chiamati **operazioni**. Gli oggetti sono passivi,

```

Object implementation stack:
top:integer;
stack: array [integer 0..N-1] of integer;

operation push(item: integer);
begin
  guard top < N - 1 do
    stack[top] := item;
    top := top + 1;
  od;
end;

operation pop(): integer;
begin
  guard top > 0 do
    top := top - 1;
    return stack[top];
  od;
end;

begin
  top := 0;
end;

```

# allocazione dello stack e puntatore alla cima  
# funzione che non restituisce valori  
# push di un elemento sullo stack  
# incremento del puntatore allo stack  
# funzione che restituisce un intero  
# sospensione in caso di stack vuoto  
# decremento del puntatore allo stack  
# restituzione del valore in cima allo stack  
# inizializzazione

Figura 8.46 Versione semplificata di un oggetto stack di Orca comprendente i dati interni e due operazioni.

cioè non contengono thread cui inviare messaggi, mentre i processi sono attivi e accedono ai dati interni di un oggetto invocando i suoi metodi.

Ogni metodo di Orca consiste in una lista di coppie (sentinella, blocco d'istruzioni). La sentinella è un'espressione booleana la cui valutazione non ha effetti collaterali, oppure una sentinella vuota che assume sempre il valore *true*. All'invocazione di un'operazione vengono valutate tutte le sue sentinelle in un ordine qualsiasi: se valgono tutte *false*, allora il processo chiamante viene sospeso finché almeno una non diventi *true*. Quando si trova almeno una sentinella con valore *true*, il blocco d'istruzioni corrispondente viene eseguito. La Figura 8.46 rappresenta un oggetto *stack* con due operazioni, *push* e *pop*.

Una volta definito uno *stack*, è possibile dichiarare variabili di questo tipo con

```
s, t: stack;
```

che crea due oggetti *stack* e inizializza la variabile *top* di ciascuna al valore 0. La variabile *intra* *k* può essere impilata sullo *stack* *s* tramite l'istruzione

```
$$push(k);
```

e così via. L'operazione *pop* ha una sentinella, perciò un tentativo di estrarre una variabile dalla cima di uno *stack* vuoto causerà la sospensione del chiamante finché un altro processo non impilerà qualcosa sullo *stack*.

Orca definisce un'istruzione **fork** per creare nuovi processi all'interno di un processore specificato dall'utente. Il nuovo processo esegue la procedura indicata nell'istruzione **fork**. È possibile passare al nuovo processo alcuni parametri, inclusi gli oggetti, ed è questo il modo in cui gli oggetti diventano distribuiti tra macchine. Per esempio l'istruzione

```
for i in 1 .. n do fork foobar(s) on i; od;
```

genera un nuovo processo su tutte le macchine dalla 1 alla *n*, e ciascuno di loro esegue il programma *foobar* all'interno della macchina corrispondente. Poiché gli *n* nuovi processi (e il loro genitore) girano in parallelo, possono tutti eseguire *push* e *pop* di elementi sullo *stack* condiviso *s* come se si trovassero in un multiprocessore a memoria condivisa. Il sistema *runtime* crea l'illusione di una memoria condivisa, che in realtà non esiste.

Le operazioni sugli oggetti condivisi sono atomiche e sequenzialmente consistenti. Il sistema garantisce che, se diversi processi eseguono operazioni sullo stesso oggetto condiviso quasi in simultanea, allora sceglierà di eseguirle in un certo ordine e tutti i processi osserveranno lo stesso ordine di eventi.

Orca integra la gestione dei dati condivisi e della sincronizzazione in un modo non presente nei sistemi DSM basati su pagine. I programmi paralleli hanno bisogno di due tipi di sincronizzazione: il primo è la sincronizzazione con mutua esclusione, che impedisce a due processi di trovarsi contemporaneamente all'interno di una regione critica. In Orca, ogni operazione su di un oggetto condiviso è in tutto simile a una regione critica, perché il sistema garantisce che il risultato finale di un insieme di operazioni parallele è lo stesso risultato che si otterrebbe dopo una loro esecuzione sequenziale. Da questo punto di vista, un oggetto di Orca è simile a una forma distribuita di monitor (Hoare, 1975).

L'altro tipo di sincronizzazione è quella su condizione, in cui un processo si blocca in attesa del soddisfacimento di una certa condizione. In Orca, la sincronizzazione su condizione si implementa con le sentinelle. Nell'esempio della Figura 8.46, se un processo cerca di eseguire una *pop* di un elemento da uno *stack* vuoto, resterà sospeso fin quando lo *stack* non sarà più vuoto.

Il sistema runtime di Orca gestisce la duplicazione, migrazione e coerenza di oggetti, e l'invocazione delle loro operazioni. Ogni oggetto può trovarsi in uno di due stati: essere una copia unica o un duplicato. Un oggetto nello stato di copia unica esiste in una sola macchina, perciò tutte le richieste che lo riguardano sono spedite lì. Un oggetto replica è presente su tutte le macchine che contengono un processo che lo usa; ciò semplifica le operazioni di lettura (che possono essere svolte localmente), ma accresce il costo delle operazioni di aggiornamento. Quando si vuol eseguire un'operazione che modifica un oggetto duplicato, per prima cosa l'operazione deve procurarsi un numero di sequenza presso un apposito processo centralizzato. Dopodiché viene spedito un messaggio a tutte le macchine che detengono una copia dell'oggetto, ordinando loro di eseguire l'operazione. Poiché tutte le operazioni di aggiornamento sono associate a un numero sequenziale, alle macchine non resta altro che eseguire le operazioni in quell'ordine, garantendo così la consistenza sequenziale.

### Globe

Molti DSM, così come anche Linda e Orca, girano su sistemi localizzati, ovvero compresi all'interno di un unico edificio o campus. D'altra parte, è anche possibile costruire sistemi multicomputer, con memoria condivisa a livello applicativo, che lavorino su scala globale. Il sistema Globe consente di localizzare gli oggetti in uno spazio degli indirizzi condiviso da svariati processi che possono trovarsi in esecuzione su macchine residenti in continenti diversi (Kermarrec et al., 1998; Popescu et al., 2002; Van Steen et al., 1999). Per accedere ai dati di un oggetto condiviso, un processo utente deve usare i propri metodi, il che permette di differenziare le strategie implementative a seconda del tipo di oggetto. Per esempio, si può mantenere una copia unica dei dati da far circolare dinamicamente su richiesta (una buona

soluzione se i dati sono aggiornati frequentemente da un unico processo proprietario), oppure replicare i dati in tutte le copie dell'oggetto e inviare gli aggiornamenti a ogni copia tramite un protocollo di trasmissione uno-a-molti affidabile.

Globe ha l'ambizione di raggiungere un miliardo di utenti e mille miliardi di oggetti che possano migrare nel sistema. La localizzazione degli oggetti e la loro gestione diventano cruciali al crescere delle dimensioni del sistema stesso. La soluzione di Globe consiste nel fornire uno schema generale in cui ciascun oggetto può specificare le proprie strategie di duplicazione, di sicurezza, e così via. Con ciò si superano i problemi presenti in altri sistemi e dovuti alla pretesa di adottare una strategia unica per tutte le situazioni, e si preserva al contempo la facilità della programmazione offerta dalla condivisione di memoria.

Tra gli altri sistemi distribuiti su aree di vaste dimensioni ricordiamo Globus (Foster e Kesselman, 1998a; Foster e Kesselman, 1998b) e Legion (Grimshaw e Wulf, 1996; Grimshaw e Wulf, 1997), che però non supportano lo stesso grado di condivisione di memoria garantito da Globe.

#### 8.4.7 Prestazioni

Lo scopo della costruzione di un calcolatore parallelo è di renderlo più veloce di una macchina monoprocesso; se non raggiunge questo semplice obiettivo, tanto vale farne a meno. Inoltre, l'obiettivo dovrebbe essere soddisfatto in modo conveniente dal punto di vista dei costi: un sistema due volte più veloce di una macchina uniprocesso, ma che costa 50 volte tanto, non è un grande affare. In questo paragrafo ci occupiamo delle questioni legate alle prestazioni che riguardano le architetture per il calcolo parallelo.

##### Parametri di valutazione hardware

In una prospettiva hardware, i parametri significativi per la misura delle prestazioni sono la velocità di CPU e I/O, e le prestazioni della rete d'interconnessione. La velocità di CPU e I/O si valuta come nel caso uniprocesso, mentre il parametro chiave che discrimina il caso parallelo è associato all'interconnessione. Esistono due concetti chiave, la latenza e la larghezza di banda, che presentiamo in successione.

La latenza di andata e ritorno (*roundtrip latency*) è il tempo che trascorre da quando una CPU spedisce un pacchetto a quando ne riceve una risposta. Se il pacchetto viene spedito alla memoria, la latenza misura il tempo di lettura o scrittura di un blocco di parole, se invece è destinato a un'altra CPU, misura il tempo di comunicazione tra processori per pacchetti di quella dimensione. In genere si è interessati alla latenza dei pacchetti di dimensione minima, spesso una parola o una piccola linea di cache.

La latenza è influenzata da svariati fattori e cambia in base al tipo d'instradamento: a commutazione di circuito, store-and-forward, a cut through virtuale (o *wormhole*). Nel primo caso, la latenza è la somma del tempo di configurazione e del tempo di trasmissione. La configurazione di un circuito avviene tramite la spedizione di un pacchetto sonda che serve a riservare le risorse e a riferire gli esiti dell'operazione al suo ritorno. Dopo di ciò, bisogna assemblare il pacchetto dati e, una volta pronto, è possibile spedire i bit a piena velocità. Perciò, se il tempo totale di configurazione è  $T_s$ , se il pacchetto contiene  $p$  bit e la larghezza di banda è di  $b$  bit/s, la latenza di sola andata è di  $T_s + p/b$  secondi. Se il circuito è

full duplex, il ritorno non comporta alcun tempo di configurazione, e così la latenza minima per la spedizione di un pacchetto di  $p$  bit, e per la ricezione di  $p$  bit di risposta, è di  $T_s + 2p/b$  secondi.

Nel caso della commutazione di pacchetto, non è necessario spedire in anticipo un pacchetto sonda alla destinazione, ma ci vuole comunque un certo tempo di configurazione  $T_s$  per assemblare il pacchetto. Il tempo di trasmissione di sola andata è  $T_d + p/b$ , ma si tratta solo del tempo impiegato per raggiungere il primo commutatore. All'interno del commutatore la trasmissione subisce un ritardo finito, chiamiamolo  $T_d$ , e il procedimento si ripete nei commutatori successivi. Il ritardo  $T_d$  comprende sia il tempo di elaborazione, sia il ritardo dovuto all'attesa nella coda perché si liberi una porta disponibile alla spedizione. Se ci sono  $n$  commutatori, allora la latenza totale di sola andata è data dalla formula  $T_s + n(p/b + T_d) + p/b$ , dove l'ultimo termine è richiesto dalla copia tra l'ultimo commutatore e la destinazione.

Nel caso dell'instradamento a cut through virtuale, la latenza di sola andata è nel migliore dei casi  $T_d + p/b$  secondi, poiché non ci sono pacchetti sonda da inviare per configurare il circuito, né ritardi dovuti all'operazione di store-and-forward. In sostanza è richiesto solo il tempo di configurazione iniziale per assemblare il pacchetto, più il tempo necessario alla spedizione dei bit. In entrambi i casi va poi aggiunto il ritardo di propagazione, che spesso è trascurabile.

L'altra metrica hardware è la larghezza di banda. Molti programmi paralleli spostano grandi quantità di dati, perciò il numero di byte trasferibili in ogni secondo diventa un parametro cruciale per le loro prestazioni. Esistono molte metriche per la misurazione della larghezza di banda e ne abbiamo già incontrata una: la larghezza di banda di bisezione. Un'altra è la **larghezza di banda complessiva** (*aggregate bandwidth*), definita come la somma delle capacità di tutti i collegamenti. Questa quantità indica il numero massimo di bit che può transitare simultaneamente sulla rete. Un altro parametro importante è la larghezza di banda media in uscita da ogni CPU. Se ciascuna CPU è in grado di spedire dati a 1 MB/s, il sistema non trae alcun giovamento dall'avere una larghezza di banda di bisezione di 100 GB/s. La comunicazione sarà comunque limitata dal quantitativo di dati che può essere emesso da ciascuna CPU.

Nella pratica è pressoché impossibile avvicinarsi ai limiti teorici della larghezza di banda perché bisogna svolgere molte attività accessorie che riducono la capacità. Per esempio, ogni pacchetto richiede sempre lo svolgimento di alcune attività preparatorie come l'assemblaggio, la costruzione della sua intestazione e la spedizione. Spedire 4 pacchetti da 1024 byte non potrà mai raggiungere la stessa larghezza di banda della spedizione di un pacchetto da 4096 byte. Sfortunatamente però, la spedizione di pacchetti piccoli aiuta a diminuire la latenza, dal momento che i pacchetti grandi occupano linee e commutatori per troppo tempo. C'è dunque un'incompatibilità insita nel cercare di raggiungere latenze medie contenute congiuntamente a larghezze di banda elevate. Le prime sono importanti per alcune applicazioni, in altri casi è vero l'esatto contrario. Vale la pena notare che, comunque, è sempre possibile procurarsi maggiore larghezza di banda (aggiungendo cavi su cavi), ma che non è possibile comprare latenze più basse. Perciò è buona regola eccedere nel tentativo di ridurre la latenza il più possibile, per poi preoccuparsi in un secondo momento della larghezza di banda.

mento si ripete nei commutatori successivi. Il ritardo  $T_d$  comprende sia il tempo di elaborazione, sia il ritardo dovuto all'attesa nella coda perché si liberi una porta disponibile alla spedizione. Se ci sono  $n$  commutatori, allora la latenza totale di sola andata è data dalla formula  $T_a + n(p/b + T_d) + p/b$ , dove l'ultimo termine è richiesto dalla copia tra l'ultimo commutatore e la destinazione.

Nel caso dell'instradamento a cut through virtuale, la latenza di sola andata è nel migliore dei casi  $T_a + p/b$  secondi, poiché non ci sono pacchetti sonda da inviare per configurare il circuito, né ritardi dovuti all'operazione di store-and-forward. In sostanza è richiesto solo il tempo di configurazione iniziale per assemblare il pacchetto, più il tempo necessario alla spedizione dei bit. In entrambi i casi va poi aggiunto il ritardo di propagazione, che spesso è trascurabile.

L'altra metrica hardware è la larghezza di banda. Molti programmi paralleli spostano grandi quantità di dati, perciò il numero di byte trasferibili in ogni secondo diventa un parametro cruciale per le loro prestazioni. Esistono molte metriche per la misurazione della larghezza di banda e ne abbiamo già incontrata una: la larghezza di banda di bisezione. Un'altra è la **larghezza di banda complessiva** (*aggregate bandwidth*), definita come la somma delle capacità di tutti i collegamenti. Questa quantità indica il numero massimo di bit che può transitare simultaneamente sulla rete. Un altro parametro importante è la larghezza di banda media in uscita da ogni CPU. Se ciascuna CPU è in grado di spedire dati a 1 MB/s, il sistema non trae alcun giovamento dall'avere una larghezza di banda di bisezione di 100 GB/s. La comunicazione sarà comunque limitata dal quantitativo di dati che può essere emesso da ciascuna CPU.

Nella pratica è pressoché impossibile avvicinarsi ai limiti teorici della larghezza di banda perché bisogna svolgere molte attività accessorie che riducono la capacità. Per esempio, ogni pacchetto richiede sempre lo svolgimento di alcune attività preparatorie come l'assemblaggio, la costruzione della sua intestazione e la spedizione. Spedire 4 pacchetti da 1024 byte non potrà mai raggiungere la stessa larghezza di banda della spedizione di un pacchetto da 4096 byte. Sfortunatamente però, la spedizione di pacchetti piccoli aiuta a diminuire la latenza, dal momento che i pacchetti grandi occupano linee e commutatori per troppo tempo. C'è dunque un'incompatibilità insita nel cercare di raggiungere latenze medie contenute congiuntamente a larghezze di banda elevate. Le prime sono importanti per alcune applicazioni, in altri casi è vero l'esatto contrario. Vale la pena notare che, comunque, è sempre possibile procurarsi maggiore larghezza di banda (aggiungendo cavi su cavi), ma che non è possibile comprare latenze più basse. Perciò è buona regola eccedere nel tentativo di ridurre la latenza il più possibile, per poi preoccuparsi in un secondo momento della larghezza di banda.

#### Parametri di valutazione software

Le metriche appena illustrate, quali la latenza e la larghezza di banda, si riferiscono alle capacità dell'hardware, ma l'interesse degli utenti è rivolto verso una prospettiva diversa: vogliono sapere quanto si ridurranno i tempi di esecuzione passando da un sistema monoprocesso a uno parallelo. Per quanto li riguarda, la metrica fondamentale è il fattore d'incremento della velocità, cioè di quante volte risulta più veloce l'esecuzione di un programma su un sistema a  $n$  processori rispetto a un sistema a monoprocesso.

L'andamento di questo rapporto viene presentato spesso mediante grafici simili a quello della Figura 8.49, in cui sono raffigurati diversi programmi paralleli eseguiti su di un multicomputer costituito da 64 CPU Pentium Pro. Ogni curva mostra l'incremento di velocità di un programma eseguito su  $k$  CPU, in funzione di  $k$ . La linea pura<sup>7</sup> indica l'incremento massimo possibile che corrisponde alla situazione in cui l'uso di  $k$  CPU accelera  $k$  volte il programma, per qualsiasi valore di  $k$ . Non sono molti i programmi che raggiungono l'incremento ideale, ma alcuni ci vanno vicini. La versione parallela della soluzione al problema degli N-corpi funziona molto bene<sup>7</sup>. La simulazione di awari (un gioco da tavolo di origine africana che si giocava in buchette scavate per terra) si comporta ragionevolmente bene. Al contrario l'inversione di una matrice a banda variabile (*skyline matrix*) non supera mai più di cinque volte la velocità di esecuzione su macchine monoprocesso, indipendentemente dal numero di processori disponibili. Per un excursus su questi programmi e sui relativi risultati si veda (Bal et al., 1998).

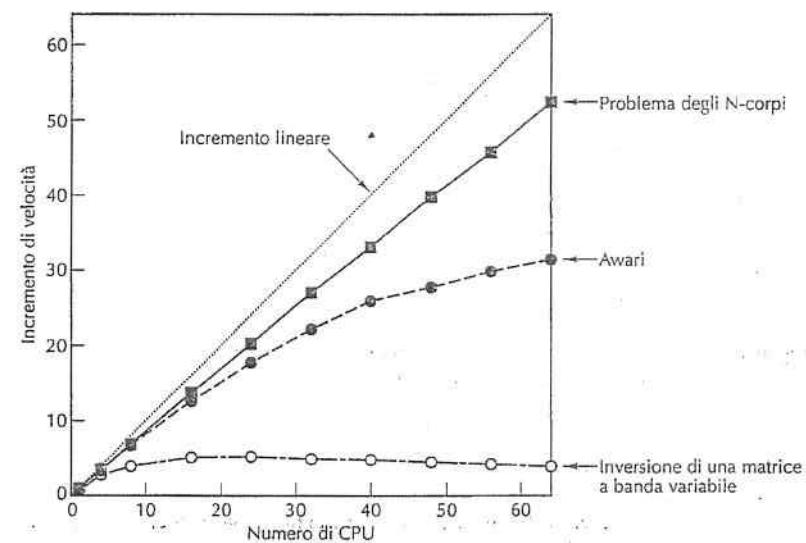


Figura 8.49 I programmi reali hanno incrementi di velocità minori dell'incremento lineare.

L'impossibilità pratica di raggiungere l'incremento ideale è dovuta in parte al fatto che quasi tutti i programmi hanno qualche componente sequenziale, di solito la fase di inizializzazione, la lettura dei dati o la raccolta dei risultati. In queste situazioni la disponibilità di più CPU non aiuta affatto. Nella Figura 8.50(a) mostriamo un programma che gira in  $T$  secondi su un monoprocesso, laddove una frazione  $f$  di questo tempo è codi-

<sup>7</sup> Si tratta del classico problema dell'evoluzione dinamica di un sistema di  $N$  masse gravitazionali (N.d.R.).

ce sequenziale e la rimanente frazione  $(1 - f)$  è potenzialmente parallelizzabile. Se questa seconda parte di codice può essere distribuita tra  $n$  CPU senza subire rallentamenti, allora il suo tempo di esecuzione può ridursi al meglio da  $(1 - f)T$  a  $(1 - f)T/n$  secondi, come indicato nella Figura 8.50(b). In ragione di ciò, il tempo totale di esecuzione risulta di  $fT + (1 - f)T/n$  secondi. Il fattore d'incremento è esattamente il rapporto tra il tempo di esecuzione del programma originale,  $T$ , e il nuovo tempo di esecuzione:

$$\text{Incremento di velocità} = \frac{n}{1 + (n - 1)f}$$

Quando  $f = 0$  otteniamo l'incremento lineare, ma per  $f > 0$  l'incremento perfetto diventa impossibile a causa della componente sequenziale. Questa formula è nota come legge di Amdahl.

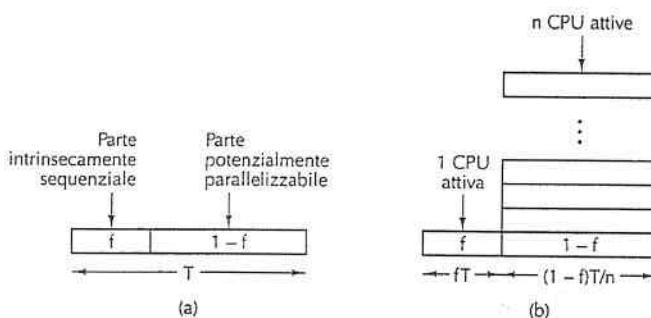


Figura 8.50 (a) I programmi hanno una parte sequenziale e una parallelizzabile. (b) Effetto dell'esecuzione parallela di una parte del programma.

Questa legge non è la sola ragione per l'impossibilità del raggiungimento dell'incremento ideale. Un ruolo in tal senso è giocato anche dalle latenze di comunicazione, dalle limitazioni poste dalla larghezza di banda e dalle inefficienze algoritmiche. Per di più, anche disponendo di 1000 CPU, non tutti i programmi possono essere scritti per utilizzare un numero di processori così elevato, e il tempo necessario per avviarli tutti potrebbe risultare significativo. Va aggiunto che, alle volte, l'algoritmo migliore per un determinato problema non è parallelizzabile efficacemente e quindi bisogna accontentarsi di un algoritmo parallelo subottimale. Al di là di queste osservazioni, è evidente che per molte applicazioni può far comodo che un programma giri  $n$  volte più velocemente, anche se ciò comporta l'uso di  $2n$  CPU. Dopo tutto le CPU non sono tanto costose e molte aziende proliferano pur raggiungendo percentuali di efficienza considerevolmente inferiori al 100%.

### Miglioramento delle prestazioni

Il modo più diretto per migliorare le prestazioni è aggiungere nuove CPU al sistema, benché questa aggiunta vada fatta in modo da non creare colli di bottiglia. I sistemi in cui la potenza di calcolo aumenta proporzionalmente al numero di CPU aggiunte si dicono scalabili.

Per meglio comprendere questo concetto, consideriamo il sistema della Figura 8.51(a) composto da quattro CPU collegate da un bus. Scalando il sistema alla dimensione 16, aggiungendo cioè 12 nuove CPU, ci riportiamo alla situazione della Figura 8.51(b). Se il bus ha larghezza di banda pari a  $b$  MB/s, la quadruplicazione del numero di CPU riduce la larghezza di banda disponibile per ogni CPU a  $b/16$  MB/s. Un sistema siffatto non è scalabile.

Ripetiamo ora la stessa operazione con il sistema a griglia delle Figure 8.51(c)-(d). In questa topologia, l'aggiunta di nuove CPU comporta la creazione di nuovi collegamenti, perciò la crescita delle dimensioni del sistema non provoca la diminuzione della larghezza di banda per CPU che abbiamo osservato nel sistema basato su bus. Infatti, il rapporto tra numero di collegamenti e numero di CPU aumenta da 1 (4 CPU, 4 collegamenti) a 1,5 (16 CPU, 24 collegamenti), dunque l'aggiunta di nuove CPU giova alla larghezza di banda per CPU.

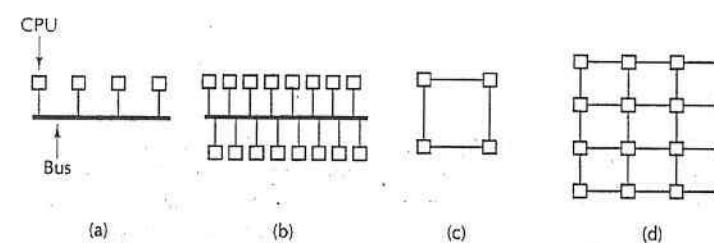


Figura 8.51 (a) Un sistema a bus con 4 CPU. (b) Un sistema a bus con 16 CPU. (c) Un sistema a griglia con 4 CPU. (d) Un sistema a griglia con 16 CPU.

Naturalmente la larghezza di banda non è il solo parametro significativo. Il collegamento al bus di nuove CPU non aumenta il diametro della rete d'interconnessione, né la latenza in assenza di traffico, ma non è così per la griglia. Il diametro di una griglia  $n \times n$  è lungo  $2(n - 1)$ , perciò la latenza nel caso medio è peggiora aumenta grosso modo come la radice quadrata del numero di CPU. Un sistema con 400 CPU ha diametro 38, uno con 1600 CPU ha diametro 78, dunque quadruplicare il numero di CPU equivale approssimativamente a raddoppiare il diametro e quindi la latenza media.

Un sistema scalabile dovrebbe mantenere idealmente la stessa larghezza di banda media per CPU e la stessa latenza media al crescere delle sue dimensioni. Tuttavia nella pratica della progettazione, se si riesce spesso a garantire sufficiente larghezza di banda per CPU, in genere si osserva però un aumento della latenza al crescere delle dimensio-

ni del progetto. Una crescita logaritmica, come quella garantita da un ipercubo, è quanto di meglio si possa fare.

La latenza è un parametro cruciale per le prestazioni di molte applicazioni a granularità fine e media (la granularità di un'applicazione si definisce come il rapporto tra il tempo di calcolo e il tempo di comunicazione). Se un programma ha bisogno di dati che non si trovano nella sua memoria locale, allora sperimenterà un certo ritardo nel procurarseli che sarà tanto maggiore quanto più grande è il sistema, come abbiamo appena argomentato. Il problema si presenta ugualmente nei multiprocessori e nei multicellulari, perché in entrambi i casi la memoria fisica si trova divisa in moduli molto distanti gli uni dagli altri.

In conseguenza di ciò, molti progettisti di sistemi cercano di intervenire su queste distanze per ridurre la latenza o almeno per nasconderla, avvalendosi di diverse tecniche che ci apprestiamo a menzionare. La prima tecnica per mascherare la latenza è la duplicazione dei dati: se presso diverse locazioni esistono delle copie di un blocco di dati, gli accessi provenienti da queste locazioni possono trarne giovamento. Una tecnica di questo tipo è il **caching**, secondo cui una o più copie dei dati sono conservate in prossimità delle locazioni che li utilizzano o cui "appartengono". D'altro canto, un'altra strategia prevede di mantenere copie di pari grado, cioè che hanno lo stesso stato, invece di utilizzare la relazione gerarchica e asimmetrica del caching che distingue tra copia primaria e secondaria. Quando si gestiscono le duplicazioni, implementate sotto qualsiasi forma, bisogna affrontare i problemi relativi alla localizzazione dei blocchi di dati, alla tempistica dei trasferimenti e all'identificazione dei loro detentori. Le risposte a questi problemi vanno dall'allocazione dinamica su richiesta a carico dell'hardware, all'allocazione intenzionale durante il normale caricamento delle variabili mediante le direttive del compilatore. In ogni caso, si pone il problema di garantire la coerenza.

Una seconda tecnica per celare la latenza è il **prefetching**. Se è possibile caricare un dato prima di doverlo utilizzare, la sua lettura può essere sovrapposta alla normale esecuzione e così il dato sarà già disponibile al momento della richiesta. Il prefetching può essere automatico o sotto il controllo del programma. Il caricamento nella cache di un'intera linea, e non solo della parola referenziata, punta sul fatto che probabilmente anche le parole successive verranno richieste molto presto.

Vediamo com'è possibile controllare il prefetching in modo esplicito: quando il compilatore si rende conto che avrà bisogno di certi dati, può inserire un'istruzione esplicita per il loro caricamento e anticiparla in modo tale che i dati siano già disponibili quando necessario. Ciò comporta da parte del compilatore una conoscenza totale della macchina sottostante e della sua temporizzazione, come anche il pieno controllo della localizzazione dei dati. Una siffatta istruzione speculativa di LOAD funziona bene solo se si è certi che i dati saranno richiesti; provocare un errore di pagina a seguito di una LOAD che risulta poi inutile si rivela molto penalizzante.

Una terza tecnica utile per nascondere la latenza è il **multithreading**, come abbiamo già visto. Se la commutazione tra processi può essere resa abbastanza veloce, per esempio dotando ciascun processo di una propria mappa di memoria e di propri registri, allora l'hardware può commutare velocemente da un thread bloccato in attesa di dati distanti verso un thread in grado di proseguire l'esecuzione. Nel caso peggiore la CPU

esegue un'istruzione del primo thread, una del secondo, e così via. Così facendo, si è sicuri di mantenere la CPU occupata anche in presenza di latenze di memoria molto lunghe cui andrebbero soggetti i singoli thread.

Una quarta tecnica che aiuta a nascondere la latenza è quella delle scritture non bloccanti. In genere l'esecuzione di un'istruzione STORE causa il blocco della CPU fino al completamento dell'operazione. L'avvio di una scrittura non bloccante consente invece la prosecuzione del programma anche se l'operazione di memoria non è stata ancora completata. La prosecuzione dell'esecuzione durante un'istruzione di LOAD è più difficile da realizzare, ma è possibile in caso di esecuzione fuori sequenza.

## 8.5 Grid computing

Molte delle sfide attuali nelle scienze, nell'ingegneria, nell'industria, nell'ambiente e in altri settori, sono di natura interdisciplinare e di vaste dimensioni. La loro risoluzione richiede esperienza, abilità, conoscenza, disponibilità di mezzi, software e dati provenienti da numerose organizzazioni, spesso ubicate in paesi diversi. Vediamo alcuni esempi.

1. Il progetto di una missione su Marte.
2. Un consorzio di aziende che costruisce un prodotto molto complesso (per esempio, una diga o un aeroplano).
3. Una squadra di soccorso internazionale che coordina gli aiuti dopo una catastrofe naturale.

Alcune di queste cooperazioni sono a lungo termine, altre a breve termine, ma tutte presentano come caratteristica comune il coinvolgimento di organizzazioni separate cui viene richiesto di partecipare con le proprie risorse e procedure di lavoro per raggiungere insieme lo scopo condiviso.

Fino a qualche tempo fa, la condivisione di dati e risorse tra organizzazioni che utilizzavano sistemi operativi, basi di dati e protocolli differenti era molto difficile. Ciononostante, il bisogno crescente di cooperazione su larga scala tra organizzazioni internazionali ha portato allo sviluppo di sistemi per la connessione di calcolatori molto distanti e altrimenti separati, che prendono il nome di tecnologia grid, o **grid computing**. In un certo senso tale tecnologia è lo stadio successivo della Figura 8.1 e può essere pensata come un cluster internazionale molto grande, lasciamente connesso ed eterogeneo.

Lo scopo della tecnologia grid è fornire un'infrastruttura tecnica che permetta la condivisione di un obiettivo comune tra un gruppo di organizzazioni che formano così un'organizzazione virtuale. Tale organizzazione deve essere flessibile, deve poter comprendere un numero di partecipanti molto grande e gestire le loro variazioni, deve consentire ai propri membri di lavorare in accordo quando lo ritengono opportuno, mentre deve garantire loro di conservare il controllo delle proprie risorse con il grado di esclusività desiderato. I ricercatori del settore grid stanno lavorando allo sviluppo di servizi, strumenti e protocolli per rendere queste organizzazioni virtuali in grado di funzionare.

La tecnologia grid è intrinsecamente multilaterale, perché ha molti partecipanti di pari grado e può essere utile metterla in contrapposizione con gli altri modelli di calcolo esistenti. Nel modello client-server una transazione coinvolge due agenti: il server, che offre un certo servizio, e il client, che desidera avvalersene. Un esempio tipico di modello client-server è il Web, in cui gli utenti si rivolgono ai server web per trovare le informazioni. La tecnologia grid differisce anche dalle applicazioni *peer-to-peer* ("da pari a pari"), le quali consentono lo scambio diretto di file tra coppie di utenti (l'e-mail ne è un esempio comune). Poiché la tecnologia grid differisce da tutti questi modelli, richiede nuovi protocolli e tecnologie.

La tecnologia grid ha bisogno di poter accedere a un gran numero di risorse. Ogni risorsa risiede su di un sistema specifico e appartiene a un'organizzazione che decide con quali modalità renderla disponibile, quando e a chi. In astratto si può dire che la tecnologia grid si occupa dell'accesso alle risorse e della loro gestione.

La Figura 8.52 mostra un modo possibile di modellare una tecnologia grid come una gerarchia di livelli. Il livello struttura (*fabric layer*) alla base della gerarchia comprende l'insieme dei componenti costitutivi dell'infrastruttura, ovvero CPU, dischi, reti e sensori (per quanto riguarda l'hardware), e i dati e i programmi (per il lato software). Queste sono le risorse messe a disposizione in maniera controllata dalla tecnologia grid.

Livello	Funzione
Applicativo	Applicazioni che condividono le risorse gestite in modo controllato
Di raccolta	- Scoperta, mediazione, monitoraggio e controllo di gruppi di risorse
Risorse	Accesso guidato e sicuro alle risorse individuali
Struttura	Risorse fisiche: calcolatori, dispositivi di memorizzazione, reti, sensori, dati e programmi

Figura 8.52 Livelli della tecnologia grid.

Il livello risorse (*resource layer*) gestisce le risorse individuali. In molti casi, una risorsa che partecipa a un sistema grid viene gestita da un processo locale che consente agli utenti distanti di accedervi in modo controllato. Questo livello fornisce ai livelli superiori un'interfaccia uniforme per ispezionare le caratteristiche e lo stato delle risorse individuali, per monitorarle e per usarle in modo sicuro.

Il livello successivo è quello di raccolta (*collective layer*) che gestisce gruppi di risorse. Una delle sue funzioni è la scoperta di risorse, tramite cui un utente può localizzare cicli di CPU disponibili, spazio su disco o dati specifici. Il livello di raccolta può servirsi di directory o di altri database per gestire queste informazioni, e può anche fornire un servizio di mediazione per trovare le corrispondenze migliori tra fornitori di

servizi e loro utenti, eventualmente allocando le risorse poco reperibili tra gli utenti in competizione per accaparrarsene. Il livello di raccolta è responsabile anche della duplicazione dei dati, dell'ammissione di nuovi membri e risorse e del mantenimento del database delle politiche che specifica quali utenti possono usare quali risorse.

Al di sopra dei precedenti si trova il livello applicativo (*application layer*), che si avvale dei livelli inferiori per procurarsi le credenziali che provano il suo diritto all'uso di certe risorse, a richiederne l'impiego, a monitorarne i progressi, a trattarne gli errori e a notificare i loro risultati agli utenti.

La caratteristica fondamentale di una buona tecnologia grid è la sicurezza. I proprietari delle risorse sono molto insistenti nel richiedere un controllo ravvicinato delle loro risorse e vogliono sapere chi le usa, per quanto tempo e con quale intensità. Nessuna organizzazione renderebbe disponibili le proprie risorse con una tecnologia grid senza un buon livello di sicurezza. D'altro canto, se un utente dovesse avere un account protetto da password su ogni singolo computer che intende usare, l'uso della tecnologia grid diventerebbe insopportabilmente scomodo. Dunque è necessario sviluppare un modello di sicurezza che risponda a queste richieste adeguatamente.

Una delle caratteristiche fondamentali del modello di sicurezza è richiedere una sola autenticazione per l'ingresso nel sistema (*single sign on*). Il primo passo per l'uso di un tecnologia grid è venire autenticati e acquisire delle credenziali, cioè un documento con firma digitale che individua l'utente per conto del quale va svolto il lavoro. Le credenziali possono essere usate come delega, così, se un'elaborazione ha bisogno di creare sotto-elaborazioni, anche i processi figli possono essere identificati. Quando si presenta una credenziale a una macchina distante, questa deve verificarla con il proprio meccanismo di sicurezza locale. Per esempio, nei sistemi UNIX gli utenti sono identificati tramite un identificatore di 16 bit, ma altri sistemi usano schemi diversi. Infine la tecnologia grid deve prevedere alcuni meccanismi per la specifica, il mantenimento e l'aggiornamento delle politiche di accesso.

L'interoperatività tra organizzazioni e macchine diverse richiede l'adozione di standard, sia in termini di servizi offerti, sia di protocolli usati per il loro accesso. La comunità grid ha creato un'organizzazione, il Global Grid Forum, proprio per gestire il processo di standardizzazione. Un esito del forum è stato la produzione di uno schema chiamato OGSA (*Open Grid Services Architecture*) per la sistemazione dei diversi standard in via di sviluppo. Laddove possibile, si utilizzano standard già esistenti: per esempio è stato usato WSDL (*Web Services Definition Language*) per la descrizione dei servizi OGSA. I servizi attualmente in corso di standardizzazione appartengono alle seguenti otto vaste categorie, ma ne verranno certamente create di nuove.

1. Servizi di infrastruttura (per la comunicazione tra risorse).
2. Servizi di gestione delle risorse (prenotazione e messa a disposizione delle risorse).
3. Servizi per i dati (trasferimento e duplicazione dei dati laddove richiesto).
4. Servizi di contesto (descrizione delle risorse richieste e delle politiche di utilizzo).
5. Servizi informativi (ottenimento delle informazioni circa la disponibilità di risorse).
6. Servizi di amministrazione interna (garanzia della qualità di servizio specificata).

7. Servizi di sicurezza (rispetto delle politiche di sicurezza).
  8. Esecuzione di servizi amministrativi (gestione dello scheduling dei compiti).
- Resterebbe ancora molto da dire su questo argomento, ma per ragioni di spazio non possiamo dilungarci ulteriormente. Per una trattazione approfondita delle tecnologie grid rimandiamo ai riferimenti (Abramson, 2011; Balasangameshwara e Raju, 2012; Celaya e Arronategui, 2011; Foster e Kesselman, 2003; Lee et al., 2011).

## 8.6 Riepilogo

È sempre più difficile rendere più veloci i computer limitandosi a incrementare la loro frequenza di clock, a causa dei problemi della dissipazione del calore e di altri fattori. I progettisti cercano invece di sfruttare il parallelismo per incrementare le prestazioni. Il parallelismo può essere introdotto a molti livelli, a partire dai più bassi, dove gli elementi dell'elaborazione sono legati molto strettamente, fino a quelli più alti, dove sono legati molto debolmente.

A livello più basso c'è il parallelismo nel chip. In primo luogo esiste il parallelismo a livello delle istruzioni, in cui un'istruzione o una sequenza d'istruzioni emettono diverse operazioni che possono essere svolte in parallelo da unità funzionali distinte. Una seconda forma di parallelismo nel chip è il multithreading, in cui la CPU può commutare continuamente tra un thread e l'altro in base al tipo d'istruzione, creando così un multiprocessore virtuale. Una terza forma di parallelismo nel chip è il multiprocessore a chip singolo, in cui due o più core disposti all'interno dello stesso chip sono in grado di girare in parallelo.

A livello superiore troviamo i coprocessori, che risiedono in genere su schede a innesto e mettono a disposizione potenza di calcolo ausiliaria per certi calcoli specializzati, come l'elaborazione di protocolli di rete o l'elaborazione multimediale. Questi processori aggiuntivi sollevano la CPU da una porzione del lavoro e le permettono così di svolgere altri calcoli mentre si occupano dei loro compiti specializzati.

A livello successivo troviamo i multiprocessori con memoria condivisa. Questi sistemi contengono due o più CPU vere e proprie che condividono una memoria. I multiprocessori UMA comunicano attraverso un bus condiviso (che effettua snooping), un commutatore crossbar o una rete a commutazione multilivello. Sono caratterizzati dal fatto che garantiscono un tempo di accesso uniforme alle diverse locazioni di memoria. Anche i multiprocessori NUMA mostrano a tutti i processi uno spazio degli indirizzi condiviso, ma in questo caso gli accessi a distanza impiegano un tempo considerevolmente maggiore rispetto a quelli locali. Infine, i multiprocessori COMA costituiscono un'ulteriore variante di multiprocessori in cui le linee di cache vengono trasferite all'interno del sistema su richiesta, ma, diversamente da altri tipi di progetto, non appartengono ad alcuna locazione in particolare...

I multicompiler sono sistemi dotati di molte CPU che non condividono memoria. Ciascuna CPU ha la sua memoria privata e comunica con le altre tramite scambio di messaggi. Gli MPP, tra cui ricordiamo BlueGene/L di IBM, sono multicompiler molto grandi dotati di reti di comunicazione specializzate. I cluster sono sistemi più semplici

costruiti a partire da componenti comunemente disponibili sul mercato, come le macchine che fanno funzionare Google.

I multicompiler sono spesso programmati avvalendosi di pacchetti software per lo scambio di messaggi quali MPI. Un approccio alternativo consiste nell'usare la condivisione di memoria a livello applicativo; ne sono alcuni esempi un sistema DSM basato su pagine, lo spazio delle tuple di Linda o gli oggetti di Orca o di Globe. DSM simula la memoria condivisa a livello della pagine, rendendo il sistema simile a una macchina NUMA, pur penalizzando molto di più gli accessi a distanza.

Infine, a livello più alto e legato più debolmente, c'è il grid computing. Si tratta di sistemi che raggruppano intere organizzazioni di utenti Internet, disposti a condividere potenza di calcolo, dati e altre risorse.

### PROBLEMI

1. Le istruzioni Intel x86 possono raggiungere i 17 byte di lunghezza. Una CPU x86 è VLIW?
2. Non appena le tecnologie hanno permesso agli ingegneri di mettere ancora più transistor in un singolo chip, Intel e AMD hanno scelto di aumentare il numero di core su ogni chip. Sarebbero state possibili scelte differenti?
3. Quali sono i valori saturati di 96, -9, 300 e 256 se l'intervallo di saturazione è 0–255?
4. Si stabilisca se le seguenti istruzioni di TriMedia sono lecite e, in caso negativo, si indichi perché non lo sono.
  - a. Somma intera, sottrazione intera, caricamento, somma in virgola mobile, caricamento immediato.
  - b. Sottrazione intera, moltiplicazione intera, caricamento immediato, scorrimento, scorrimento.
  - c. Caricamento immediato, somma in virgola mobile, moltiplicazione in virgola mobile, salto, caricamento immediato.
5. Le Figure 8.7(d) e 8.7(e) mostrano 12 cicli d'istruzioni. Per ciascuna delle due sequenze, si indichi che cosa succede nei tre cicli successivi.
6. In una certa CPU, un'istruzione che causa un fallimento nella cache di primo livello, ma che ha successo nella cache di secondo livello, richiede in tutto  $k$  cicli. Se si usa il multithreading per nascondere i fallimenti di cache di primo livello, quanti thread bisogna eseguire contemporaneamente per evitare cicli di inattività con il multithreading a grana fine?
7. La GPU NVIDIA Fermi è simile, come spirito, a una delle architetture studiate nel Capitolo 2. Quale?
8. Una mattina l'ape regina di un certo alveare convoca le api operale e affida loro, come compito per la giornata, la raccolta di nettare di calendula. Quindi le operaie volano in direzioni diverse in cerca di calendule. Si tratta di un sistema SIMD o MIMD?
9. Quando abbiamo esaminato i modelli di consistenza della memoria, abbiamo detto che un modello di consistenza è una specie di contratto tra il software e la memoria. Perché è necessario un tale contratto?
10. Si consideri un multiprocessore con bus condiviso. Che cosa succede se due processori cercano di accedere alla memoria globale esattamente nello stesso istante?
11. Si consideri un multiprocessore con bus condiviso. Che cosa succede se tre processori cercano di accedere alla memoria globale esattamente nello stesso istante?
12. Si supponga che, per qualche ragione tecnica, una cache possa fare lo snooping solo delle linee degli indirizzi, e non delle linee dati. Questa limitazione avrebbe ripercussioni sul protocollo write through?
13. Si prenda in considerazione un semplice modello di sistema multiprocessore senza caching basato sul bus, in cui ogni quattro istruzioni accede alla memoria e l'accesso alla memoria occupa il bus per un tempo pari all'e-

- secuzione di un'istruzione. Quando il bus è occupato, le CPU richiedenti vengono poste in una coda d'attesa FIFO. Quanto un sistema di 64 CPU risulterà più veloce rispetto a un sistema con una sola CPU?
14. Il protocollo MESI di coerenza delle cache ha quattro stati, mentre altri protocolli write-back di coerenza delle cache ne hanno solo tre. Quale dei quattro stati di MESI può essere sacrificato e con quali conseguenze? Se si dovessero scegliere solo tre stati, quali sarebbero?
  15. Il protocollo MESI di coerenza delle cache dà luogo a situazioni in cui una linea di cache richiede una transazione sul bus pur essendo presente nella cache locale? Se sì, spieghi perché.
  16. Ci sono  $n$  CPU collegate a un bus comune. La probabilità che una CPU cerchi di usare il bus in un determinato ciclo è  $p$ . Qual è la probabilità che durante un ciclo
    - a. il bus resti inoperoso (0 richieste);
    - b. venga fatta una sola richiesta;
    - c. vengano fatte più richieste?
  17. Si elenchino i maggiori vantaggi e svantaggi dei commutatori crossbar.
  18. Quanti commutatori crossbar ci sono in un E25K di Sun Fire completo?
  19. Se si interrompe il collegamento tra i commutatori 2A e 3B della rete omega della Figura 8.31, quali elementi ne risultano scollegati?
  20. Gli hot spot ("punti caldi") sono le locazioni di memoria cui si accede frequentemente e costituiscono un problema grave nelle reti a commutazione multilivello. Sono un problema anche nei sistemi basati su bus?
  21. Una rete di commutazione omega collega 4096 CPU RISC, ciascuna con ciclo di 60 ns, a 4096 moduli di memoria infinitamente veloci. Gli elementi che effettuano la commutazione introducono un ritardo di 5 ns. Quant'intervalli di ritardo sono necessari a un'istruzione LOAD?
  22. Si consideri una macchina che usa una rete di commutazione omega come quella mostrata nella Figura 8.29. Se il modulo di memoria  $i$  contiene il programma e lo stack del processore  $i$ , si proponga una piccola modifica della topologia che produca un grosso cambiamento delle prestazioni (RP3 di IBM e Butterfly di BBN usano questa topologia modificata). Quale svantaggio comporta la nuova topologia rispetto a quella originale?
  23. In un multiprocessore NUMA, un riferimento alla memoria locale impiega 20 ns, mentre un riferimento a distanza impiega 120 ns. Un certo programma effettua  $N$  riferimenti a memoria durante la sua esecuzione, l'1% dei quali avviene verso la pagina  $P$ . La pagina  $P$  si trova inizialmente a distanza e ci vogliono  $C$  secondi per copiarla localmente. In quali condizioni conviene copiare la pagina localmente, in assenza di un suo utilizzo significativo da parte di altri processori?
  24. Si consideri un multiprocessore CC-NUMA simile a quello della Figura 8.33, ma con 512 nodi dotati di 8 MB ciascuno. Se le linee di cache sono di 64 byte, qual è la percentuale d'informazioni accessorie necessarie alle directory? L'incremento del numero di nodi aumenta, diminuisce o lascia invariata questa percentuale?
  25. Che differenza c'è tra NC-NUMA e CC-NUMA?
  26. Si calcoli il diametro di rete delle topologie mostrate nella Figura 8.37.
  27. Si calcoli il grado di tolleranza agli errori delle topologie mostrate nella Figura 8.37. Il grado di tolleranza è definito come il massimo numero di collegamenti che è possibile interrompere senza separare la rete in due componenti sconnesse.
  28. Si consideri la topologia a doppio toro della Figura 8.37(f) espansa alle dimensioni  $k \times k$ . Qual è il diametro della rete risultante? Suggerimento: distinguere i casi con  $k$  pari e  $k$  dispari.
  29. Una rete d'interconnessione ha la forma di un cubo  $8 \times 8 \times 8$ . Ogni collegamento ha una larghezza di banda full duplex di 1 GB/s. Qual è la larghezza di banda di bisezione della rete?

30. La legge di Amdhal limita l'incremento potenziale raggiungibile da un calcolatore parallelo. Si calcoli, in funzione di  $f$ , l'incremento massimo raggiungibile per un numero di CPU che tende a infinito. Quali conseguenze ha questo limite per  $f = 0,1$ ?
31. La Figura 8.51 mostra la buona scalabilità di una griglia in contrapposizione alla cattiva scalabilità di un bus. Se ogni bus o collegamento ha una larghezza di banda  $b$ , si calcoli la larghezza di banda media per CPU in ciascuno dei quattro casi della figura. Si scalino quindi tutti i sistemi alla dimensione di 64 CPU e si ripeta il calcolo. Qual è il limite se il numero di CPU tende a infinito?
32. Abbiamo trattato nel capitolo tre varianti di send: sincrona, bloccante e non bloccante. Si fornisca un quarto metodo che sia simile alla variante bloccante, ma che abbia proprietà leggermente diverse. Si elenchino vantaggi e svantaggi della versione proposta valutati rispetto alla send bloccante.
33. Si consideri un multicomputer che usa una rete che fa broadcast hardware come Ethernet. Perché è importante il rapporto tra il numero di operazioni di lettura (che non modificano lo stato interno delle variabili) e il numero di operazioni di scrittura (che modificano lo stato interno delle variabili)?