UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in Ingegneria Informatica

TESI DI LAUREA

Sistemi multiprocessore e multicore

Relatore
Prof. Sergio Congiu

Candidato
Lorenzo Baesso

Anno Accademico 2010 – 2011

Abstract

In questa tesi viene trattato l'argomento dei sistemi multiprocessore e multicore sotto diversi aspetti. Una prima introduzione darà un'idea generale circa l'importanza di queste recenti tecnologie definendo inoltre gli elementi chiave di questo argomento, i motivi che hanno spinto i ricercatori e le case costruttrici ad investire tempo e denaro in questo mercato ed i vantaggi ottenibili da queste macchine. A seguire, verrà fornita una classificazione di tutte le possibili architetture parallele. Poi verranno descritti i vari tipi di sistemi multiprocessore e multicore secondo principi di classificazione come il tipo di gestione della memoria o il tipo di multithreading usato. Infine verranno illustrati i principali strumenti di valutazione delle prestazioni e ripresi i punti chiave trattati nei precedenti capitoli, che più di tutti caratterizzano questi computer.

Indice

Introduzione

Si immagini di dover ordinare un mazzo di carte: una prima soluzione sarebbe quella di procedere ordinando per seme e poi riunendo i vari mazzetti. Se ci fosse un'altra persona ad aiutarvi, potreste dividervi il compito e utilizzando lo stesso procedimento impieghereste la metà del tempo utilizzato prima, lavorando contemporaneamente. Con l'aiuto di una terza persona impieghereste un terzo del tempo iniziale e così via fino a raggiungere il punto in cui dividere il compito tra più persone non è più conveniente in termini di tempo. Quello appena fornito è un esempio di applicazione del calcolo parallelo ad un particolare compito. Ora si immagini che il compito da svolgere sia un programma o un processo e che le persone siano i processori di un sistema multiprocessore.

Tra gli obiettivi principali che progettisti e costruttori si sono sempre posti troviamo i seguenti: ridurre al massimo il tempo impiegato dal sistema per risolvere problemi di dimensioni considerevoli, seguire la strategia più economica possibile e contemporaneamente garantire una certa affidabilità del sistema. Infatti è stato accertato che, in linea generale, in termini di tempo è più conveniente far lavorare contemporaneamente più processori e poi combinare le soluzioni che utilizzare un unico processore superveloce, il cui costo crescerebbe esponenzialmente in accordo con la potenza. Anche dal punto di vista economico, l'utilizzo di più processori di media/alta velocità è più vantaggioso perché permette una riutilizzazione dei componenti che non sono di ultima generazione. Più processori inoltre garantiscono più affidabilità da parte del sistema in quanto nel caso in cui un processore si guastasse gli altri sarebbero pronti a sostituirlo portando avanti il lavoro assegnato, cosa non possibile in una macchina monoprocessore.

Nonostante i progressi fatti nel campo tecnologico si è arrivati a dover affrontare limiti insormontabili come la velocità di trasmissione tra computer (la quale costituisce un collo di bottiglia rispetto alla velocità di lavoro interna del singolo computer), l'impossibilità di miniaturizzare i componenti al di sotto di una certa soglia per motivi fisici e l'inconvenienza per le case costruttrici nel costruire processori sempre più veloci ed esponenzialmente più costosi, in quanto non sarebbero più prodotti competitivi e non rientrerebbero quindi nella fascia di mercato dominata dal consumatore medio.

Per i suddetti motivi, ad un certo punto è stato indispensabile introdurre i sistemi multiprocessore e multicore, definiti come:

- S. multiprocessore (Fig. 1): *"Sistema (computer, workstation, server o reti di computer) equipaggiato con 2 processori o più, operanti in parallelo."* [1]

- S. multicore (Fig. 2): *"Sistema le cui Central Processing Unit sono composte da due o più core, ovvero da più nuclei di processori fisici montati sullo stesso chip package."* [2]

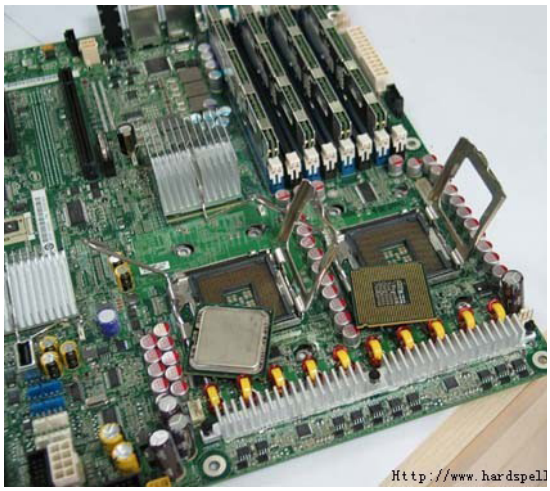

Fig.2 Chip package dotat o di quattro core

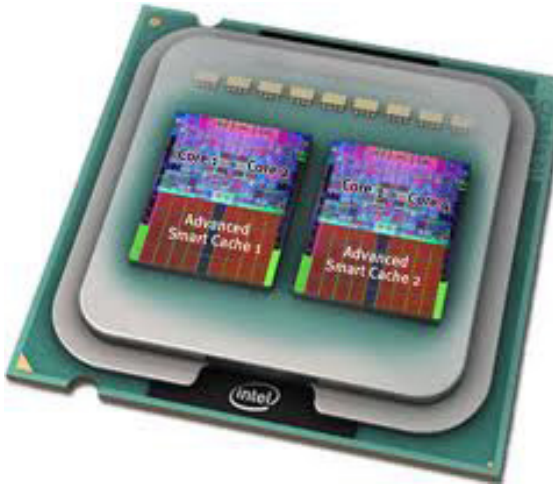

Fig.1 Scheda madre dotata di due processori

Ai fini concettuali di questa tesi si considererà i sistemi multiprocessore e multicore come equivalenti anche se nella pratica sono implementazioni differenti dello stesso concetto, il che potrebbe portare a certe differenze. Ad esempio, tra i vantaggi dei multicore si ha una possibile utilizzazione del clock a frequenze maggiori, un uso più efficiente delle memorie cache, dei tempi di risposta migliori con carichi di lavoro intensi, una notevole riduzione del circuito stampato e la possibilità di lavorare a correnti più basse, quindi una dispersione di calore minore. Ciononostante nei sistemi multicore sono necessarie delle modifiche al sistema operativo e ai programmi preesistenti; in generale sono più difficili da gestire rispetto ai sistemi monoprocessori.

1. Classificazione delle architetture parallele

Qualsiasi architettura parallela è stata definita nel 1989 dai professori George S. Almasi e Allan Gottlieb come:

"Insieme di elementi di elaborazione che cooperano e comunicano per risolvere velocemente problemi di dimensioni considerevoli, talvolta intrattabili su macchine sequenziali."

Essa è caratterizzata da tre livelli di parallelismo [3]:

- Data Level Parallelism; ottenuto quando i dati, su cui il programma lavora, vengono distribuiti ed elaborati contemporaneamente da più processori.
- Instruction Level Parallelism; ottenuto quando le istruzioni, che compongono il programma, vengono distribuite ed eseguite contemporaneamente da più processori.
- Thread Level Parallelism; ottenuto quando le applicazioni utilizzano thread/processi concorrenti [4], cioè thread/processi che vengono eseguiti in parallelo da più processori. Si noti che non necessariamente i processori operano in modo sincrono.

Sulla base di questi concetti, l'ingegnere e informatico Michael J. Flynn nel 1966 ha definito quella che tutt'oggi è la tassonomia più completa e popolare. La classificazione si basa appunto sulla nozione di flusso di informazioni. In un processore sono presenti due tipi di flusso di informazioni: istruzioni e dati. Concettualmente questi possono essere pensati come due flussi indipendenti, a seconda che essi scorrano su due cablature differenti o meno. La tassonomia di Flynn [5] classifica le macchine a seconda che esse abbiano un flusso o più per ciascun tipo (Fig. 3).

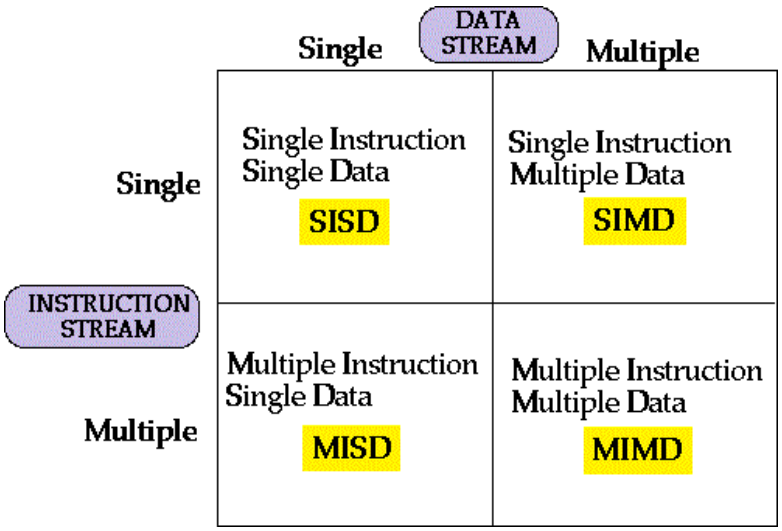

Fig. 3 Tassonomia di Flynn

1.1 Architettura Single Instruction Single Data

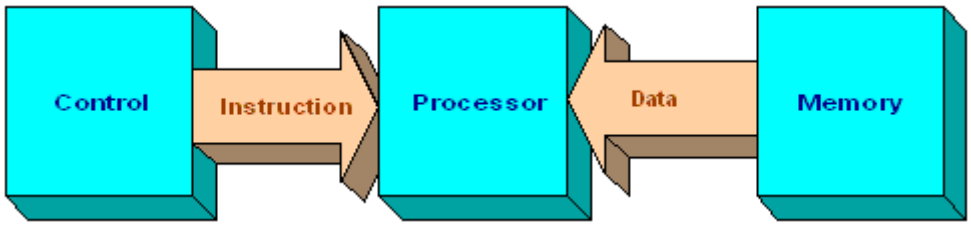

Fig. 4 Architettura SISD

Un computer SISD (Fig. 4) è costituito da una singola unità di elaborazione, la quale riceve un unico flusso di istruzioni che opera su un unico flusso di dati. Ad ogni ciclo di clock, l'unità centrale emette un'istruzione che opera su un dato ottenuto dalla memoria centrale, il tutto secondo le seguenti tre fasi:

1. Fetch: Prelevamento dalla memoria centrale dell'istruzione da eseguire utilizzando l'indirizzo contenuto nel Program Counter, incremento del PC in maniera tale che possa contenere l'indirizzo della prossima istruzione da eseguire.
2. Decode: La Control Unit interpreta l'istruzione e determina le operazioni da eseguire.
3. Execute: L'Arithmetic Logic Unit esegue le operazioni e la CU controlla l'andamento dell'esecuzione delle operazioni.

In Fig. 5 sono evidenziate nello specifico quali aree di un comune chip vengono utilizzate nelle fasi di fetch, decode ed execute.

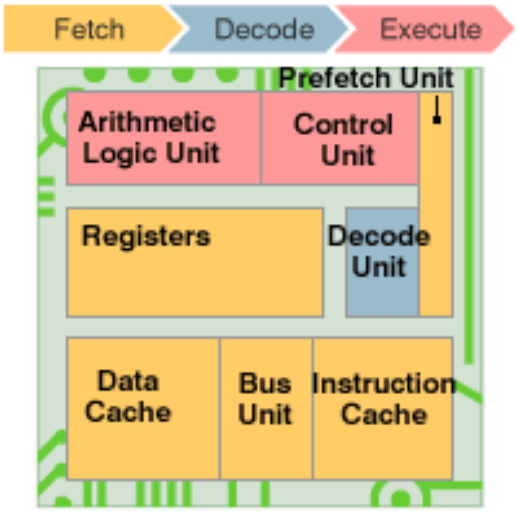

Fig. 5 Componenti del chip utilizzate nelle fasi di Fetch-Decode-Execute

Quasi tutti i computer oggi in uso aderiscono al modello inventato dal matematico e informatico John von Neumann [6] negli anni '40, schematizzato in Fig. 6. Gli algoritmi che vengono eseguiti su questa tipologia di computer sono detti sequenziali (o seriali), in quanto non contengono alcun parallelismo. Esempi di computer SISD sono i Personal Computer, le workstation e i mainframe dotati di una singola CPU.

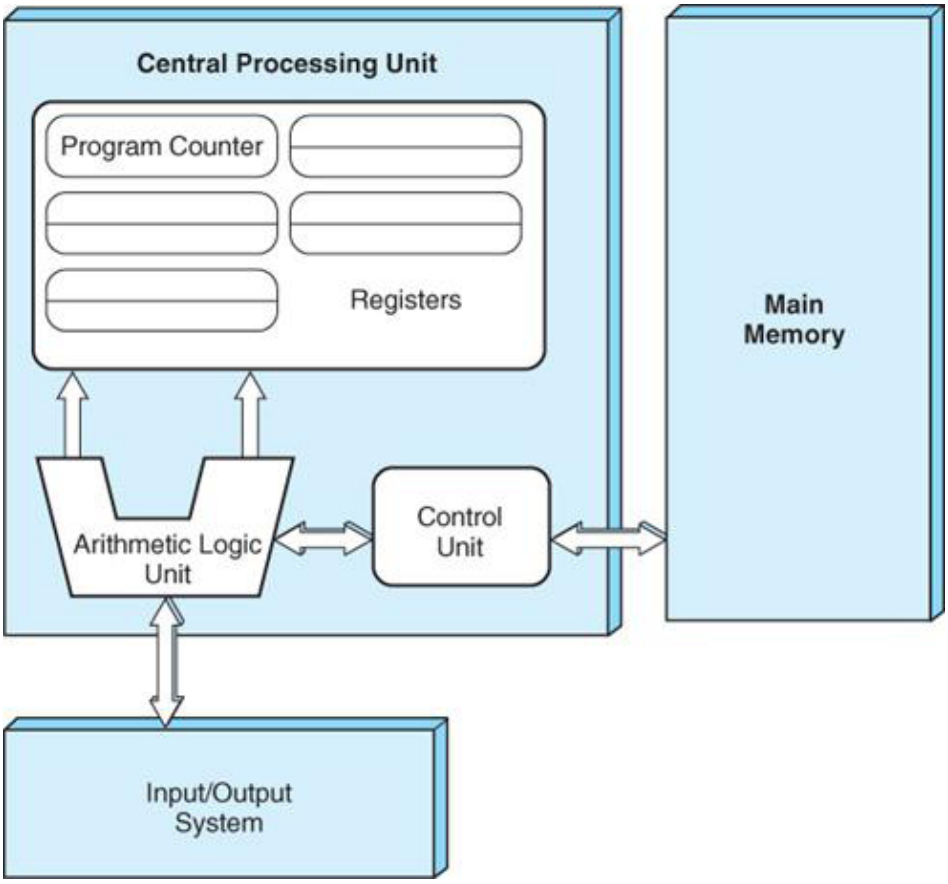

Fig. 6 Architettura di Von Neumann

Elementi chiave:

- Memoria centrale: usata per memorizzare sia le istruzioni che i dati del programma.
- CPU: ottiene le istruzioni e/o i dati dalla memoria, decodifica le istruzioni e sequenzialmente le implementa.
- Sistema I/O: insieme dei dati in ingresso ed in uscita al programma.

1.2 Architettura Multiple Instruction Single Data

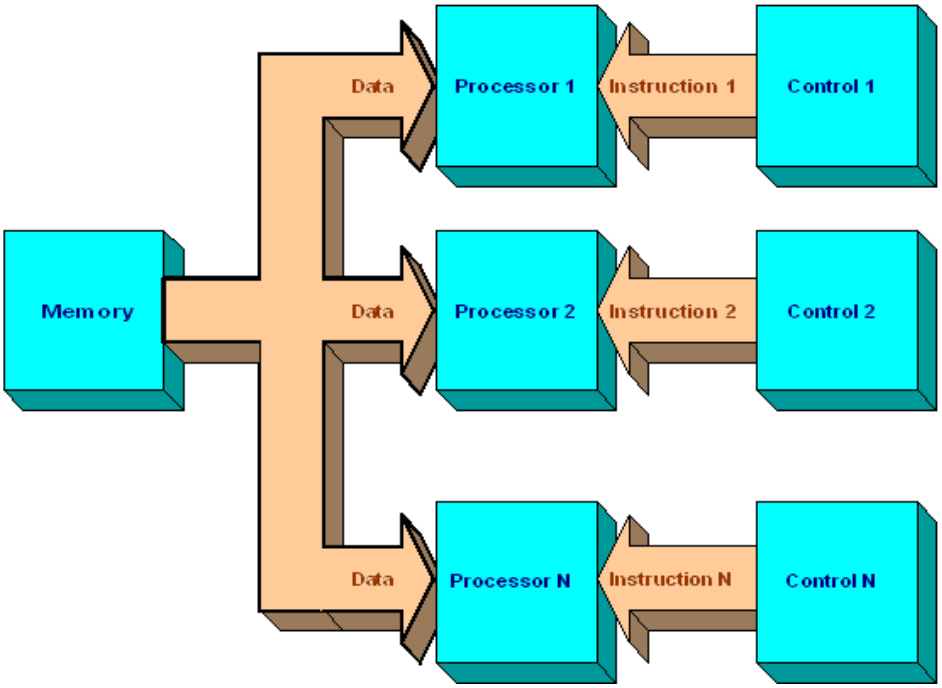

Fig. 7 Architettura MISD

In questo modello *n* processori, ciascuno con la propria unità di controllo, condividono un'unica unità di memoria. Ad ogni ciclo di clock, il dato ricevuto dalla memoria viene elaborato da tutti i processori simultaneamente, ciascuno secondo le istruzioni ricevute dalla propria unità di controllo. Il parallelismo, a livello di istruzioni, si ottiene lasciando fare ai processori diverse operazioni sugli stessi dati.

Questa classe di computer si presta naturalmente a quei compiti che devono eseguire operazioni diverse su input identici, come nel caso della classificazione dei problemi computazionali. I tipi di problemi che possono essere risolti in modo efficiente su questi computer sono piuttosto particolari, come quelli riguardanti la crittografia, e proprio per questo i computer MISD (Fig. 7) non hanno trovato spazio nel settore commerciale.

1.3 Architettura Single Instruction Multiple Data

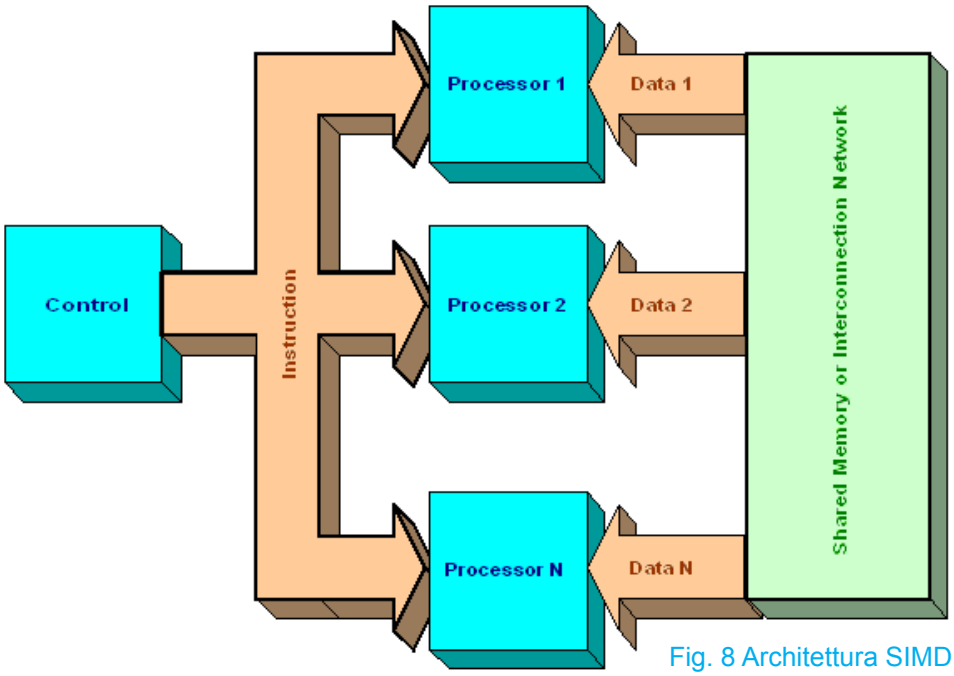

Fig. 8 Architettura SIMD

Un computer SIMD (Fig. 8) è composto da *n* processori identici, ognuno dotato di una propria memoria locale dove gli è possibile archiviare i dati. Tutti i processori lavorano sotto il controllo di un flusso di istruzioni unico e rilasciato da un'unità di controllo centrale. Inoltre sono presenti *n* flussi di dati, uno per ogni processore. I processori funzionano contemporaneamente: ad ogni passo, tutti i processori eseguono la stessa istruzione ma su elementi di dati diversi. Questo è un esempio di parallelismo a livello dati.

I computer SIMD sono molto più versatili dei computer MISD. Numerosi problemi che coprono una vasta gamma di applicazioni possono essere risolti da algoritmi paralleli su computer SIMD. Un'altra caratteristica interessante è che gli algoritmi per questi computer sono relativamente facili da progettare, analizzare e implementare. Come limitazione però, si ha che solo i problemi che possono essere suddivisi in una serie di sottoproblemi tutti identici, ognuno dei quali verrà poi risolto contemporaneamente tramite lo stesso insieme di istruzioni, possono essere affrontati con i computer SIMD. I problemi che non rientrano in questa casistica sono quelli tipicamente suddivisi in sottoproblemi, non necessariamente identici, e risolvibili utilizzando computer MIMD. Le architetture SIMD negli ultimi anni hanno preso piede nel campo dell'elaborazione grafica e delle applicazioni multimediali. Ciononostante i processori vettoriali [7] compongono la classe più ampia di questi calcolatori. I processori vettoriali sono comuni nelle applicazioni scientifiche e si trovano spesso alla base dei supercomputer, fin dagli anni ottanta.

1.4 Architettura Multiple Instruction Multiple Data

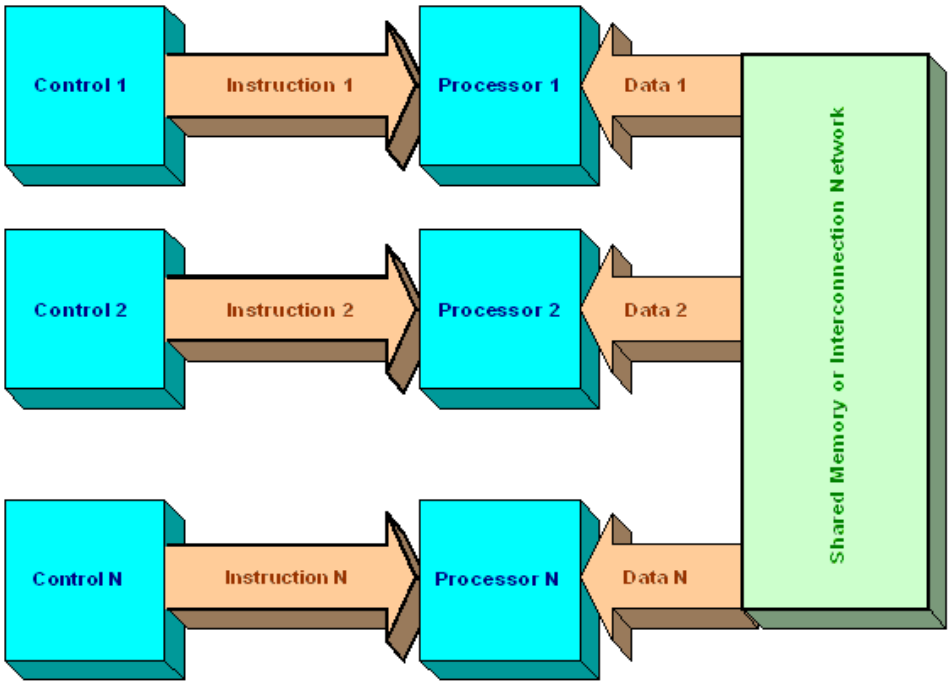

Fig. 9 Architettura MIMD

Questa classe di computer paralleli è la più generale e più potente nella classificazione di Flynn. Qui ci sono *n* processori, *n* flussi di istruzioni e *n* flussi di dati. Ogni processore possiede una propria unità di controllo e una propria memoria locale, rendendoli più potenti di quelli utilizzati nei computer SIMD. Ogni processore opera sotto il controllo di un flusso di istruzioni rilasciato dalla propria unità di controllo: pertanto, i processori possono potenzialmente eseguire programmi diversi su dati diversi, risolvendo sottoproblemi differenti ma facenti parte di un unico problema più grande. È così raggiunto il parallelismo a livello di thread/processi. Questo significa inoltre che i processori operano di solito in modo asincrono.

Il modello di calcolo parallelo MIMD (Fig. 9) è il più generale e potente: i computer in questa classe vengono utilizzati per risolvere in parallelo quei problemi che non hanno la struttura regolare richiesti dal modello SIMD. Al giorno d'oggi questa architettura è applicata a molti PC, supercomputer e reti di computer. C'è però un contro da tenere molto in considerazione: gli algoritmi asincroni sono difficili da progettare, analizzare e implementare.

2. Organizzazione della memoria

Un aspetto molto importante a cui si deve porre particolare attenzione nel progettare un computer multiprocessore, se si vuole ottenere da esso il massimo delle prestazioni, è l'organizzazione della memoria o, per meglio dire, il modo in cui vi si accede. Non importa quanto veloce si renda l'unità di elaborazione, se la memoria non può mantenere e fornire istruzioni e dati ad una velocità sufficiente non ci sarà alcun miglioramento nelle prestazioni. Il problema principale che deve essere superato per rendere compatibili il tempo di risposta della memoria e la velocità del processore è il tempo di ciclo di memoria, definito come il tempo che intercorre tra due operazioni successive. I tempi di ciclo del processore sono in genere molto più brevi dei tempi di ciclo di memoria. Quando un processore avvia un trasferimento da o verso la memoria, questa rimarrà occupata per tutto il tempo di ciclo di memoria. Durante questo periodo nessun altro dispositivo (controller I / O, processori, o anche il processore stesso che ha fatto la richiesta) può utilizzare la memoria in quanto sarà impegnata a rispondere alla richiesta in corso.

Soluzioni al problema di accesso alla memoria hanno portato ad una dicotomia delle architetture MIMD (Fig. 10). In un tipo di sistema, noto come sistema a memoria condivisa, c'è un'elevata memoria virtuale, e tutti i processori hanno pari accesso ai dati e alle istruzioni presenti in questa memoria. L'altro tipo di sistema è quello a memoria distribuita, in cui ogni processore ha una memoria locale che non è accessibile agli altri processori.

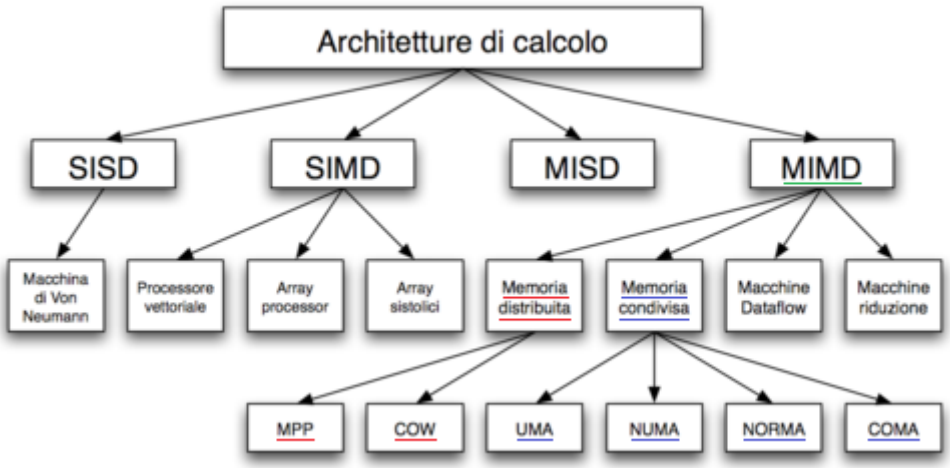

Fig. 10 Tassonomia di Flynn in dettaglio

La differenza tra memoria condivisa e distribuita sta nella struttura della memoria virtuale, ossia la memoria vista dalla prospettiva del processore. Fisicamente, quasi ogni sistema di memoria è suddiviso in componenti distinte che sono accessibili in modo indipendente. Ciò che distingue una memoria condivisa da una memoria distribuita è come il sottosistema di memoria interpreta un indirizzo generato da un processore. Come esempio, supponiamo che un processore esegua l'istruzione *load R0,i* il che significa "carica nel registro *R0* il contenuto della locazione di memoria *i*". La domanda è: cosa vale *i*? In un sistema a memoria condivisa, *i* è un indirizzo globale e quindi la locazione *i* è la stessa per ogni processore. Se due processori eseguissero allo stesso tempo questa istruzione, caricherebbero la stessa informazione nei loro registri *R0*. In un sistema a memoria distribuita, *i* è un indirizzo locale. Se due processori eseguissero l'istruzione *load R0,i* contemporaneamente, valori diversi potrebbero finire nei rispettivi registri *R0* in quanto, in questo caso, *i* designa celle di memoria diverse, una per ogni memoria locale.

La distinzione tra memoria condivisa e memoria distribuita è molto importante per i programmatori, perché determina il modo in cui diverse parti di un programma parallelo devono comunicare. In un sistema a memoria condivisa è sufficiente costruire una struttura dati in memoria e passare alle subroutine parallele le variabili di riferimento di tale struttura dati. Una macchina a memoria distribuita, d'altronde, deve creare copie dei dati condivisi in ciascuna memoria locale. Queste copie vengono create inviando, da un processore all'altro, un messaggio contenente i dati da condividere. Un inconveniente di questa organizzazione di memoria è che a volte questi messaggi possono essere molto grandi e richiedere tempi di trasferimento relativamente lunghi.

2.1 Memoria Condivisa

Un modo semplice per collegare più processori insieme per costruire un multiprocessore a memoria condivisa è mostrato in Fig. 11.

Le connessioni fisiche sono abbastanza semplici. La maggior parte delle strutture bus permettono un arbitrario (ma non troppo grande) numero di dispositivi che condividono lo stesso canale. I protocolli di bus sono stati inizialmente concepiti per consentire ad un singolo processore e ad uno o più dischi o controller a nastro di comunicare con la memoria condivisa. Se ai controller I/O si aggiungono i processori, si ottiene un multiprocessore a bus unico. Si noti che ad ogni processore è stata associata una memoria cache in quanto si presume che la probabilità che un processore necessiti di un dato o di un'istruzione presente nella memoria locale sia molto alta ($p>0.9$).

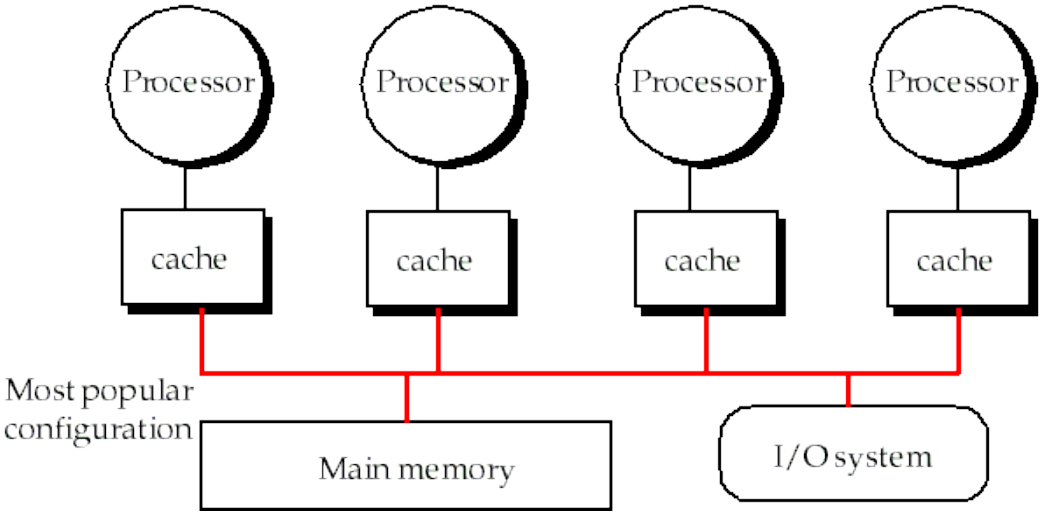

Fig. 11 Schema di un'architettura a memoria condivisa

Il problema nasce quando un processore modifica un dato della memoria principale simultaneamente utilizzato da altri processori. Il nuovo valore passerà dalla cache del processore che l'ha modificato alla memoria condivisa; in seguito, però, esso dovrà esser passato anche a tutti gli altri processori in modo che essi non lavorino con un valore obsoleto. Questo problema è noto come problema di coerenza della cache (Fig. 12) [8], caso particolare del problema di coerenza della memoria, che richiede delle implementazioni hardware in grado di gestire problemi di concorrenza e sincronizzazione, similarmente a quelli che si ha con i thread a livello di programmazione.

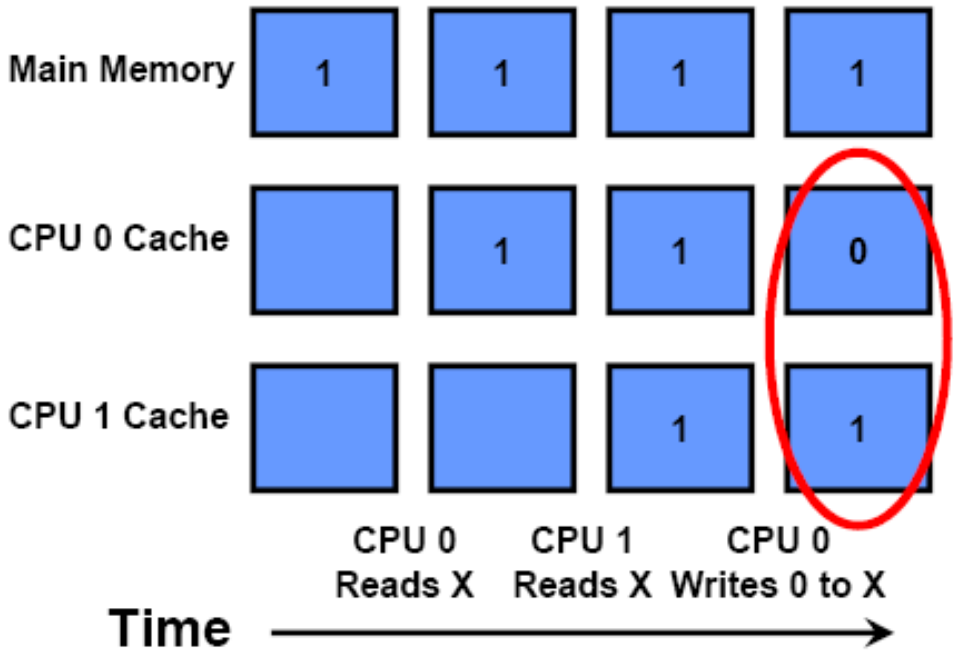

Fig. 12 Problema di coerenza della cache

Caratteristiche dei sistemi multiprocessore a memoria condivisa:

- La memoria logica è la stessa per tutti i processori; ad esempio, tutti i processori associati alla stessa struttura dati lavoreranno con gli stessi indirizzi logici, in quanto globali, accedendo così alle stesse locazioni di memoria.
- La sincronizzazione è ottenuta leggendo i compiti dei vari processori e concedendo a turno la memoria condivisa; infatti i processori possono solo accedervi uno alla volta.
- Una locazione di memoria condivisa non deve essere modificata da un compito mentre un altro compito concorrente vi accede.
- La condivisione dei dati tra i vari compiti è veloce; infatti il tempo necessario alle attività per comunicare tra loro è il tempo che una di esse impiega per leggere una singola locazione (ciò dipende dalla velocità di accesso alla memoria).
- La scalabilità è limitata dal numero di vie d'accesso alla memoria; questo limite si presenta soprattutto quando ci sono più compiti che connessioni alla memoria. In queste situazioni si avranno dei processori in stato d'attesa e quindi tempi di latenza maggiori.
- Il programmatore è responsabile della gestione della sincronizzazione, inserendo opportuni controlli, semafori, lock ecc nel programma che gestisce le risorse.

Classificazione dei sistemi multiprocessore a memoria condivisa:

- Uniform Memory Access (Fig. 13): la caratteristica fondamentale di questo sistema è il tempo di accesso alla memoria che è costante per ogni processore e per qualsiasi zona di memoria. Proprio per questa relazione simmetrica dei processori con la memoria, questi sistemi sono anche detti Simmetric Multiprocessor (SMP). Essi sono relativamente semplici da implementare ma non molto scalabili [9]; possono arrivare a gestire una dozzina di processori, al massimo.

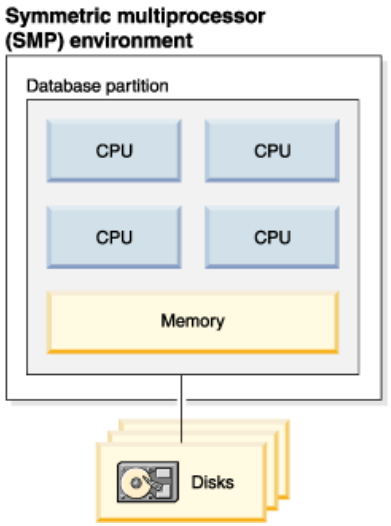

Fig. 13 Schema dell'architettura SMP

- NonUniform Memory Access (Fig. 14): queste architetture suddividono la memoria in una zona ad alta velocità assegnata singolarmente ad ogni processore ed una eventuale zona comune per lo scambio dei dati, ad accesso più lento. Per questa caratteristica, tali sistemi sono anche detti Distributed Shared Memory Systems (DSM). Essi sono molto scalabili ma complessi da sviluppare.

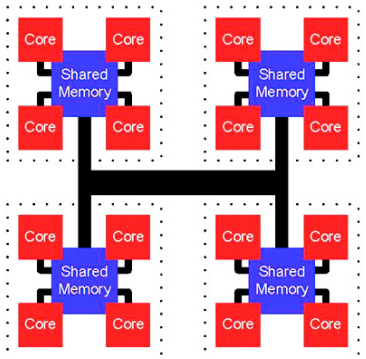

Fig. 14 Schema dell'architettura DSM

- NO-Remote Memory Access (Fig. 15): la memoria è distribuita fisicamente tra i processori (local memory). Tutte le memorie locali sono private e può accedervi solo il processore locale. La comunicazione tra i processori avviene tramite un protocollo di comunicazione per scambio di messaggi (Message Passing, vedi Fig. 18).

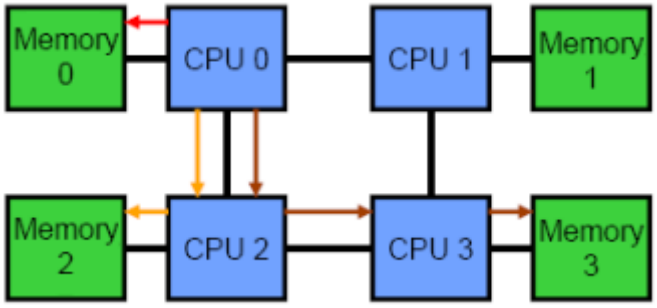

Fig. 15 Schema dell'architettura NORMA

- Cache Only Memory Access (Fig. 16): questa tipologia di elaboratori sono dotati solamente di memorie cache. Analizzando le architetture NUMA si è notato che queste mantenevano delle copie locali dei dati nelle cache e che questi dati erano memorizzati come doppioni anche nella memoria principale. Questa architettura elimina i doppioni mantenendo solo le memorie cache.

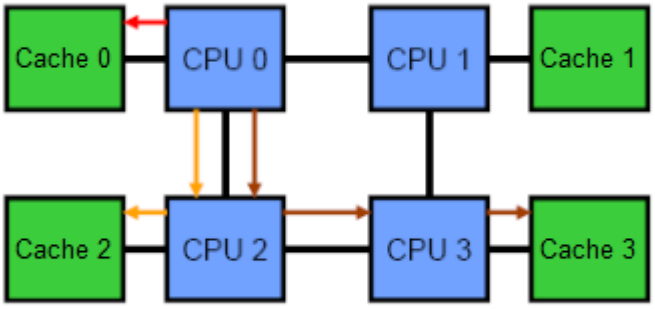

Fig. 16 Schema dell'architettura COMA

2.2 Memoria Distribuita

In un sistema a memoria distribuita la memoria è associata ai singoli processori e un processore è solamente in grado di indirizzare la propria memoria. Alcuni autori fanno riferimento a questo tipo di sistema come "multicomputer" [10], riflettendo il fatto che i blocchi del sistema sono a loro volta piccoli sistemi completi di processore e memoria, come si può vedere in Fig. 17.

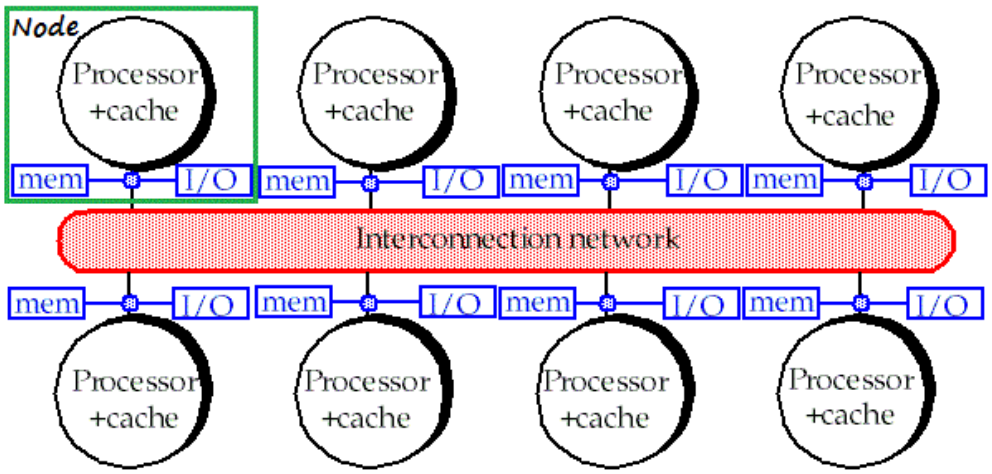

Fig. 17 Schema di un'architettura a memoria distribuita

Questa organizzazione presenta diversi vantaggi. In primo luogo, non vi sono conflitti a livello di bus o switch. Ogni processore può utilizzare l'intera larghezza di banda della propria memoria locale, senza subire interferenze da parte di altri processori. In secondo luogo, la mancanza di un bus comune significa che non c'è limite intrinseco al numero di processori, ora la dimensione del sistema è limitata soltanto dalla rete utilizzata per collegare i processori. In terzo luogo, non ci sono problemi di coerenza della cache. Ogni processore è responsabile dei propri dati, e non deve preoccuparsi di aggiornare eventuali copie.

Il principale svantaggio nel disegno a memoria distribuita è che la comunicazione interprocessore è più difficile da implementare. Se un processore richiedesse dei dati presenti nella memoria di un altro processore, i due processori dovrebbero necessariamente scambiarsi dei messaggi tramite il Message Passing (Fig. 18) [11]. Ciò introduce due fonti di rallentamento: per costruire e inviare un messaggio da un processore all'altro ci vuole tempo, e inoltre un qualsiasi processore deve essere interrotto al fine di gestire i messaggi ricevuti da altri processori.

Un programma, creato per funzionare su una macchina a memoria distribuita, deve essere organizzato come un insieme di attività indipendenti che comunicano tra loro tramite messaggi.

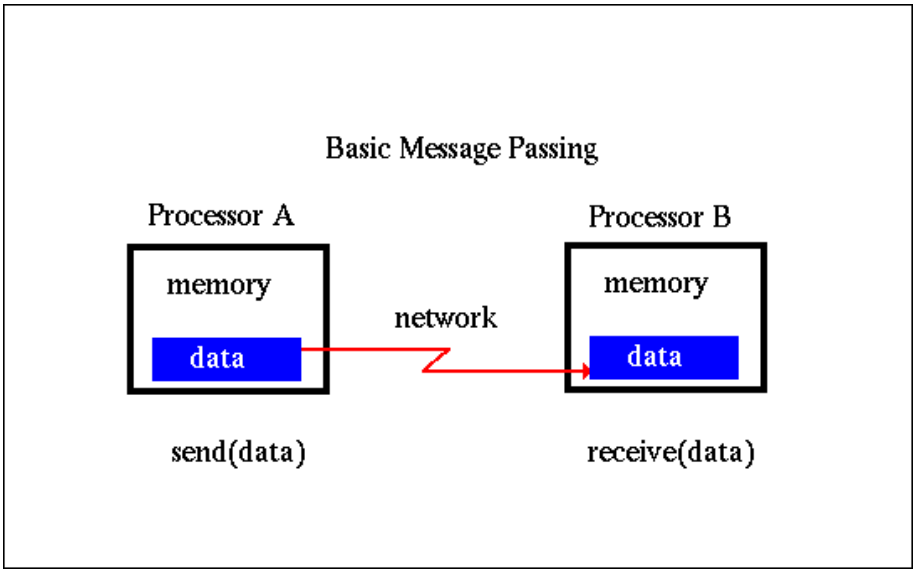

Fig. 18 Rappresentazione generica del Message Passing

Caratteristiche dei sistemi multiprocessore a memoria distribuita:

- La memoria è fisicamente distribuita tra i vari processori; ogni memoria locale è accessibile direttamente solo dal suo processore.

- La sincronizzazione è ottenuta mediante lo spostamento di dati (anche se è solo il messaggio stesso) tra i processori (comunicazione).

- La suddivisione dei dati nelle memorie locali incide molto sulle prestazioni della macchina: è fondamentale fare una suddivisione accurata in modo da ridurre al minimo le comunicazioni tra le CPU. Inoltre, il processore che coordina queste operazioni di decomposizione e composizione deve comunicare efficacemente con i processori che operano sulle singole parti delle strutture dati.

- Il Message Passing consiste nel far comunicare le CPU tra di loro tramite scambi di pacchetti dati. I messaggi trasmessi sono unità discrete di informazione; nel senso che hanno una identità ben definita, perciò deve essere sempre possibile poterli distinguere gli uni dagli altri.

Classificazione dei sistemi multiprocessore a memoria distribuita:

- Massively Parallel Processing (Fig. 19): le macchine MPP sono composte da centinaia di processori (che possono diventare anche centinaia di migliaia in alcune macchine) collegati da una rete di comunicazione. Le macchine più veloci del pianeta sono basate su queste architetture [12].

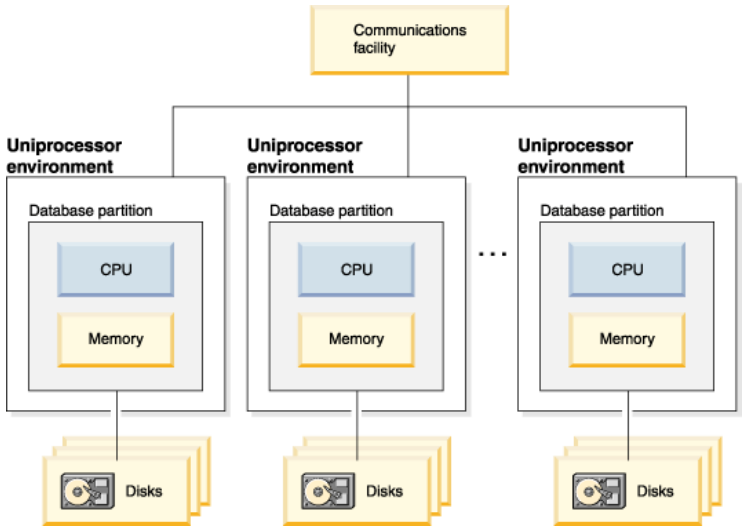

Fig. 19 Schema dell'architettura MPP

- Cluster Of Workstations (Fig. 20): le architetture COW sono sistemi di elaborazione basati su classici computer collegati da reti di comunicazione. I cluster di calcolo [13] ricadono in questa classificazione.

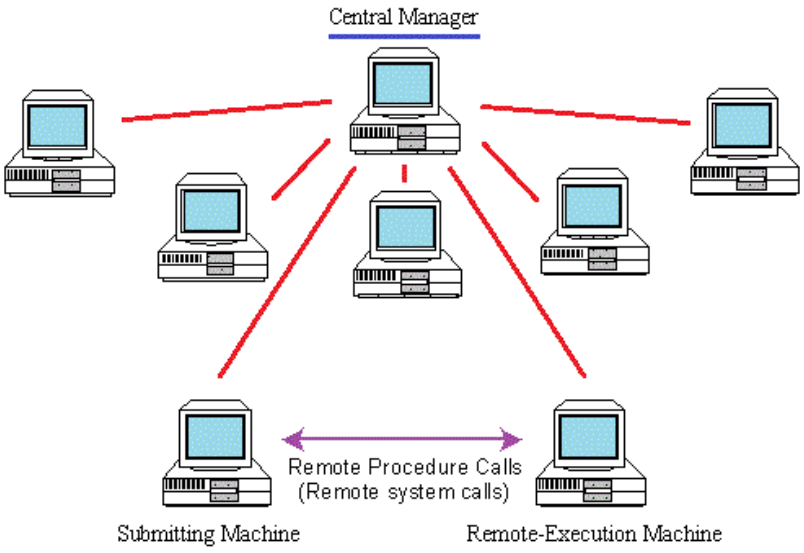

Fig. 20 Schema dell'architettura COW

3. Multithreading

Top500.org è il sito web ufficiale che si occupa di tenere aggiornata, ogni sei mesi, la lista dei 500 computer più veloci al mondo. Nel glossario dei termini, alla voce multithreading (Fig. 21) troviamo la seguente definizione:

"Capacità di un processore di passare da un thread all'altro. [...] Questa capacità è usata quando uno dei thread è in uno stato di stallo, ad esempio perché i dati necessari non sono ancora disponibili. Passare ad un altro thread, le cui istruzioni possono essere eseguite, comporterà un'elaborazione dei dati migliore."

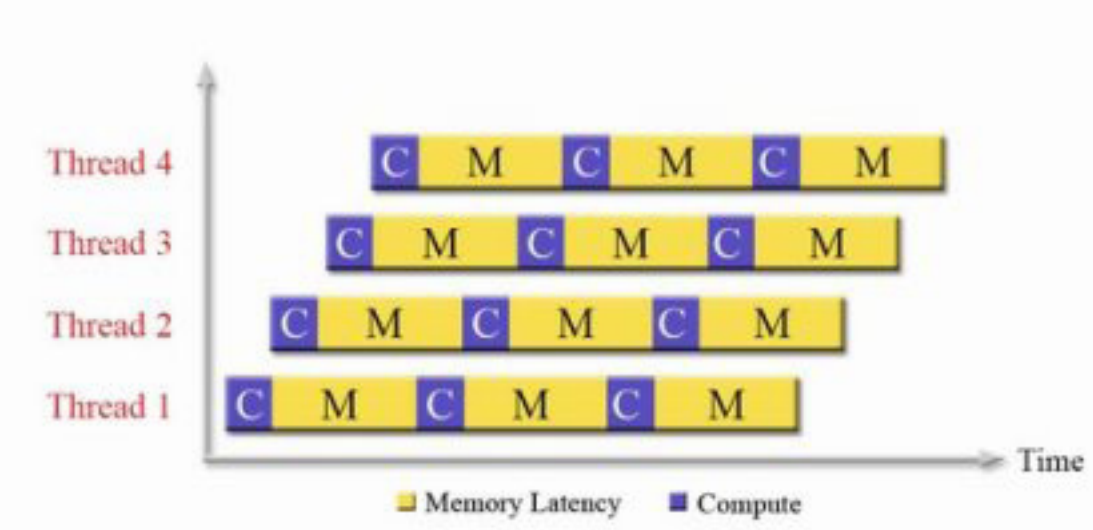

Fig. 21 Esempio di multithreading (Sun UltraSparc T1 - Niagara)

In altre parole il multithreading permette a diversi thread di condividere le risorse hardware di un unico processore in modo tale che esso possa eseguirli in parallelo. Affinché questa tecnica sia efficace, è necessario che il sistema operativo e le applicazioni la supportino. Quindi il software non dovrà più essere di tipo sequenziale ma bensì programmato secondo una logica parallela. Inoltre, anche l'hardware dovrà essere progettato in modo da poter passare velocemente da un thread all'altro.

Il multithreading soddisfa il TLP ed è stato introdotto quando ci si è accorti che per aumentare le prestazioni dei calcolatori l'ILP non era più sufficiente. Con il solo utilizzo dell'ILP si arrivò a situazioni conflittuali che comportavano di conseguenza inutili sprechi di risorse. Paradigmi di processi sono stati inventati per migliorare l'ILP ed aumentare il volume di istruzioni eseguibili simultaneamente, come l'Out of Order Processing [14], ma si è visto che le prestazioni ottenute non potevano giustificare le costose migliorie da fare a livello hardware. Perciò si è optato per il multithreading come soluzione finale.

I due tipi principali di hardware multithreading sono il Fine-Grained Multithreading e il Coarse-Grained Multithreading. Tuttavia tratterò anche una variante, molto importante per le ottime prestazioni computazionali che offre, e cioè il Simultaneous Multithreading.

3.1 Coarse-Grained Multithreading

Il multithreading a grana grossa rende più efficiente l'utilizzo delle unità funzionali mediante l'esecuzione di un solo thread alla volta, per un certo numero di cicli di clock. Questa tecnica è efficiente laddove interi cicli di clock sono inattivi in quanto il thread che si stava eseguendo è in fase di stallo. L'efficienza è dovuta appunto all'eliminazione di questi cicli di clock inattivi; il processore, appena rileva che un thread è in stato di stallo, passa immediatamente all'esecuzione di un altro thread. Quindi non resta più inattivo aspettando che si compiano tutti i cicli di clock assegnati ad un thread, bensì continua a lavorare per un altro.

Prima di passare al thread successivo, il processore salva lo stato di quello attuale facendo una copia delle istruzioni presenti nella pipeline [15] e le unità funzionali in uso. Il tutto utilizzando un certo numero di set di registri. Più rari sono errori del tipo richiedere un dato mancante nella cache, di secondo livello generalmente, o esaurire tutte le istruzioni indipendenti, più questo tipo di multithreading è efficiente. Inoltre difficilmente il CMT rallenta l'esecuzione dei singoli thread. D'altronde questa tecnica risulta totalmente inefficace nel caso in cui gli stalli siano molto piccoli. in tal caso il tempo necessario per salvare lo stato della pipeline e riempirla con le istruzioni del thread successivo sarebbe di gran lunga maggiore rispetto a quello che si dovrebbe aspettare per far passare i cicli di clock inattivi.

3.2 Fine-Grained Multithreading

Un approccio più aggressivo al multithreading è chiamato multithreading a grana fine. Come il CMT, l'obiettivo del FMT è di passare rapidamente tra i vari thread. A differenza del CMT, però, i passaggi avvengono ad ogni ciclo di clock. Grazie a questa strategia l'alternanza di tutti i thread è garantita; il che permette di ottenere un rendimento complessivo più alto. Questa alternanza è spesso creata in stile round-robin [16], saltando qualsiasi thread si trovi in uno stato di stallo nel momento in cui dovrebbe essere eseguito.

Anche l'hardware deve essere implementato in modo da poter passare da un thread all'altro ad ogni ciclo di clock. Il vantaggio principale del FMT è che, sia per stalli brevi che per stalli lunghi, viene mantenuto un certo rendimento, in quanto in ogni caso il processore deve procedere eseguendo delle istruzioni. Lo svantaggio più grande di questa tecnica è che essa rallenta l'esecuzione dei singoli thread, in quanto un thread che non presenta stalli verrà purtroppo ritardato dall'esecuzione delle istruzioni degli altri thread.

3.3 Simultaneous Multithreading

In questo multithreading il processore è in grado di gestire un parallelismo sia a livello di thread sia a livello di istruzioni. Va detto però che di solito le architetture che utilizzano l'SMT dispongono di un numero di unità funzionali maggiore di quello di cui un singolo thread potrebbe necessitare in realtà. Inoltre le indipendenze tra le istruzioni vengono risolte grazie alla grande versatilità di questi processori per la programmazione dinamica [17].

Lo scopo dell'SMT è quello di eseguire istruzioni provenienti da diversi thread, in qualsiasi momento, in una qualsiasi unità funzionale. Nell'alternare i thread, un chip funziona come un processore FMT ma allo stesso tempo agisce come un processore CMT, eseguendo simultaneamente istruzioni proprie di thread diversi. Per questo motivo, l'SMT permette agli architetti di hardware di progettare core sempre più grandi senza preoccuparsi che questi ne risentano in termini di rendimento. Come ci si aspetta, più thread sono in uso più alto sarà il rendimento complessivo; però, per ottenere questo vantaggio i costruttori devono porre particolare attenzione al momento in cui dimensionano le cache nei vari livelli. Infatti incrementando o decrementando le dimensioni delle cache si possono ottenere diversi vantaggi o svantaggi [18].

3.4 CMT, FMT e SMT a confronto

In Fig. 22 sono rappresentati quattro thread differenti, i quali sono eseguiti su processori superscalari [19]. In questi diagrammi, dove la dimensione orizzontale rappresenta le istruzioni per ciclo di clock e quella verticale la sequenza dei cicli di clock, è quindi totalmente escluso alcun tipo di multithreading. Ogni quadrante colorato indica l'esecuzione di un'istruzione, mentre ogni quadrante bianco rappresenta uno stato di inattività del processore.

Si può facilmente notare l'enorme spreco di risorse. Tale spreco è dovuto principalmente a due motivi: il primo è che i cicli di clock in cui tutti gli slot a disposizione vengono utilizzati sono molto rari, il secondo è che gli stalli possono paralizzare interi thread.

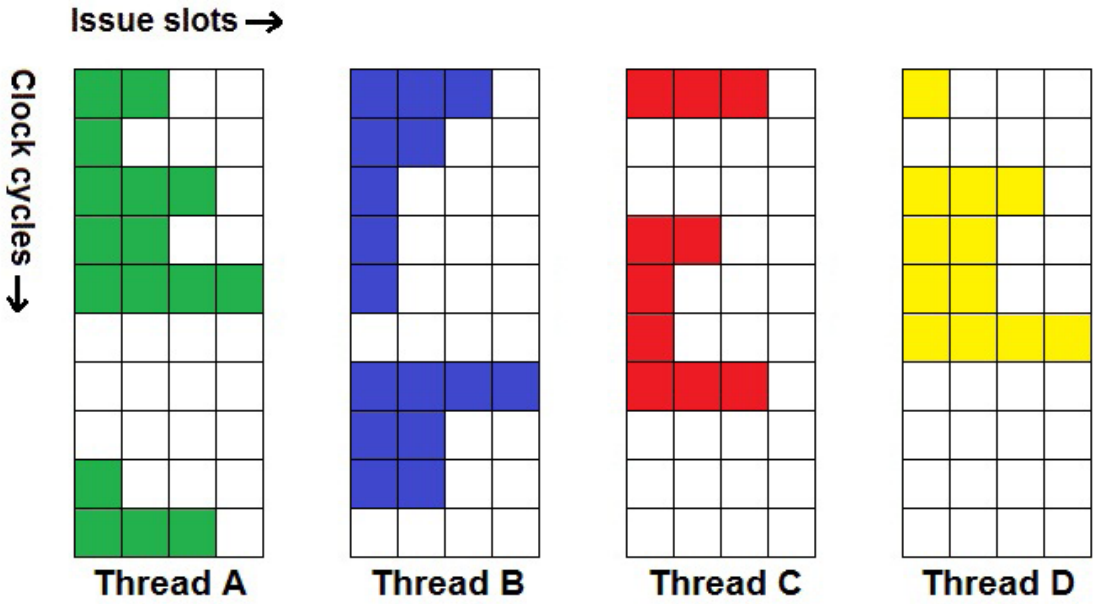

Fig. 22 Esecuzione di quattro thread senza multithreading

Con l'intervento del multithreading le cose vanno sicuramente meglio, come si può notare in Fig. 23. Nel caso del CMT, gli stalli più lunghi vengono parzialmente risolti mediante il passaggio da un thread inattivo ad uno attivo. Nonostante questo riduca il numero di cicli di clock completamente inattivi, il caricamento nella pipeline delle istruzioni degli altri thread porta ad avere ancora cicli di clock inattivi. Inoltre, i limiti dell'ILP fanno si che non tutti gli slot vengano utilizzati ad ogni ciclo di clock.

Con un processore superscalare di tipo FMT è possibile eliminare i cicli di clock totalmente inattivi ed ogni ciclo è riservato ad un unico thread. Restano comunque le limitazioni imposte dal parallelismo a livello di istruzioni.

Tramite l'uso dell'SMT si ottiene la massima efficienza, infatti gli slot inattivi sono veramente pochi. Questo grazie alla combinazione dell'ILP e del TLP che, ad ogni ciclo di clock, permette a thread multipli di utilizzare quasi tutte le risorse messe a disposizione dal processore. Teoricamente l'utilizzo degli slot è esclusivamente riservato ai vari thread ma nella pratica altri fattori possono incidere e limitare l'utilizzo di alcuni slot.

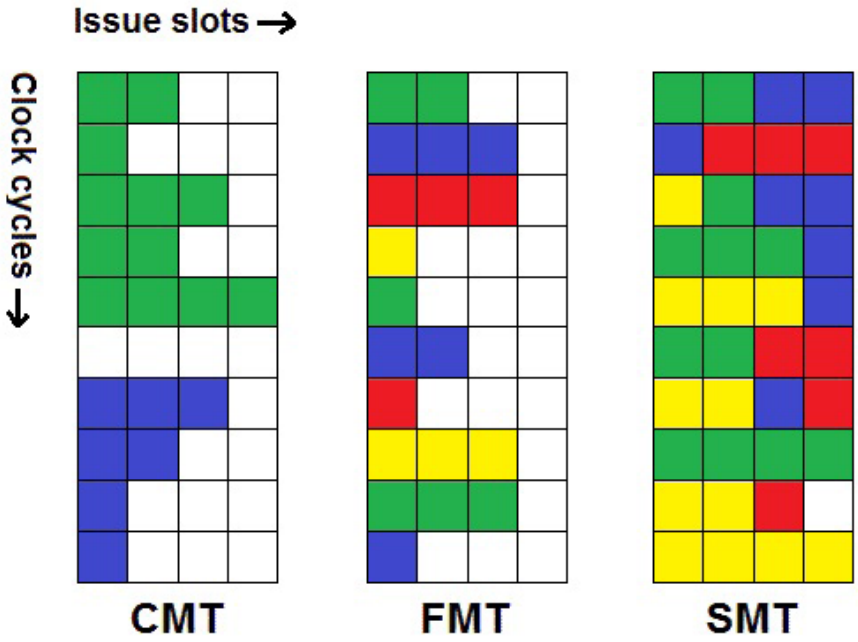

Fig. 23 Esecuzione dei quattro thread in CMT, FMT e SMT multithreading

Dai test fatti su tecniche di multithreading più avanzate, progettisti e costruttori sono giunti alla conclusione comune che i vantaggi ottenuti non giustificano i costi di produzione troppo elevati. Proprio per questo è stato deciso di puntare di più sulle architetture multiprocessore che su tecniche di multithreading troppo sofisticate.

4. Indici e modelli di valutazione delle prestazioni

Fin dalla nascita del concetto di parallelismo è stato importante dare un senso matematico e analitico a tutti quei cambiamenti che ne sono derivati. In altre parole, si è cercato di definire degli strumenti di analisi che fossero in grado innanzitutto di prevedere i limiti dei programmi paralleli ed eventualmente valutarne le prestazioni. Ecco quindi come si è cominciato a parlare, ad esempio, di speedup dei programmi paralleli rispetto a quelli sequenziali. Definiti i concetti sono state poi elaborate le leggi che tutt'oggi hanno tanta importanza quanta validità; come la legge di Amdahl, che definisce i limiti del parallelismo, o la legge di Gustafson, che suggerisce invece in che casi una parallelizzazione del codice può risultare efficiente. Dalle leggi sono stati derivati altri parametri, come quello di Karp-Flatt che cerca di fornire un modello più preciso di quelli forniti da Amdahl e Gustafson. Infine, sulla base di queste conoscenze teoriche, sono stati costruiti i modelli su cui tutt'oggi i progettisti fanno affidamento per analizzare le prestazioni dei sistemi multiprocessore e multicore, come il noto modello Roofline.

Si noti che i paragrafi seguenti tratteranno solo alcuni dei principali strumenti di analisi dei sistemi multiprocessore e multicore; nella realtà dei fatti essi non sarebbero comunque sufficienti a fornire un quadro completo dell'analisi, andrebbero a loro volta integrati con modelli che considerano anche altri aspetti di questi sistemi [20].

4.1 Speedup

Con il termine speedup ci si riferisce a quanto un algoritmo parallelo è più veloce del corrispondente algoritmo sequenziale. Questo coefficiente è definito dalla seguente formula:

$$S_p = \frac{T_1}{T_p}$$

Dove:

- p è il numero dei processori;
- T_1 è il tempo di esecuzione della versione sequenziale dell'algoritmo;
- T_p è il tempo di esecuzione della versione parallela dell'algoritmo con p processori.

Lo speedup ideale, anche detto speedup lineare, si ottiene quando $Sp=p$. Se si eseguisse un algoritmo con speedup lineare e si raddoppiasse il numero di processori allora raddoppierebbe anche la velocità. Poiché questo corrisponderebbe al caso ideale, si ritiene che un sistema così avrebbe la massima scalabilità possibile, e cioè la massima versatilità a crescere o decrescere in funzione delle necessità o disponibilità del problema.

Lo speedup è detto assoluto quando T_1 è il tempo di esecuzione del miglior algoritmo sequenziale e relativo quando T_1 è il tempo di esecuzione del corrispondente algoritmo parallelo su un unico processore. Lo speedup relativo è generalmente quello sottointeso se il tipo di speedup non è specificato, in quanto non richiede l'implementazione dell'algoritmo sequenziale.

L'efficienza è un parametro delle prestazioni, definito come:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

Si tratta di un valore, compreso tra zero e uno, che stima quanto vengono sfruttati i processori nella risoluzione del problema rispetto ai loro sforzi compiuti nelle comunicazioni e nelle sincronizzazioni. Gli algoritmi con speedup lineare e gli algoritmi in esecuzione su un singolo processore hanno efficienza pari a 1, mentre molti algoritmi, difficili da parallelizzare, hanno efficienza pari a $\frac{1}{\log p}$. Tale parametro tende a zero all'aumentare del numero di processori.

4.2 Legge di Amdahl

La legge di Amdahl prende il nome dall'ingegnere e informatico Gene Amdahl, ed è usata per trovare lo speedup massimo di un sistema in cui è possibile parallelizzare solo una parte del programma in esecuzione e le dimensioni del problema non cambiano. Lo speedup di un programma parallelizzato solo in parte è limitato dal tempo necessario ad eseguire la parte sequenziale. Infatti, indipendentemente dal numero di processori che dedichiamo all'esecuzione in parallelo di questo programma, il tempo di esecuzione minimo non può essere inferiore a quello della parte sequenziale. Da questo si è dedotto che il massimo speedup ottenibile da un programma non totalmente parallelizzato è:

$$S = \frac{1}{1-P}$$

Dove *1-P* è la parte di programma che deve rimanere sequenziale. Più precisamente, la legge di Amdahl calcola lo speedup complessivo secondo la seguente formula:

$$SpeedupComplessivo = \frac{1}{(1-P) + \frac{P}{S}}$$

Dove:

- *1* rappresenta il tempo di esecuzione dell'algoritmo sequenziale;
- *(1-P)*, come prima, è la porzione di tempo di esecuzione non migliorabile (es. se il tempo di esecuzione parallelizzabile è il 90%, *P* sarà uguale a 0.9);
- $\frac{P}{S}$ è la porzione di tempo di esecuzione migliorabile divisa per il suo speedup *S*, ottenuto da tale miglioramento.

Ricordando che, nell'ambito del parallelismo, il caso ideale si ha quando *S* è uguale a *N* numero di processori utilizzati, si ottiene il seguente speedup complessivo ideale:

$$SpeedupComplessivoIdeale = \frac{1}{(1-P) + \frac{P}{N}}$$

Teoricamente, al tendere all'infinito di *N*, lo speedup complessivo tende a $\frac{1}{1-P}$ (ecco dimostrata la deduzione citata all'inizio). In realtà, il rapporto qualità prezzo decresce rapidamente appena *N* aumenta nel caso in cui ci sia anche solo una piccola componente *(1-P)*. Infatti questa legge consiglia appunto di parallelizzare il codice il più possibile, anziché aumentare a dismisura il numero di processori utilizzati (vedi Fig. 24).

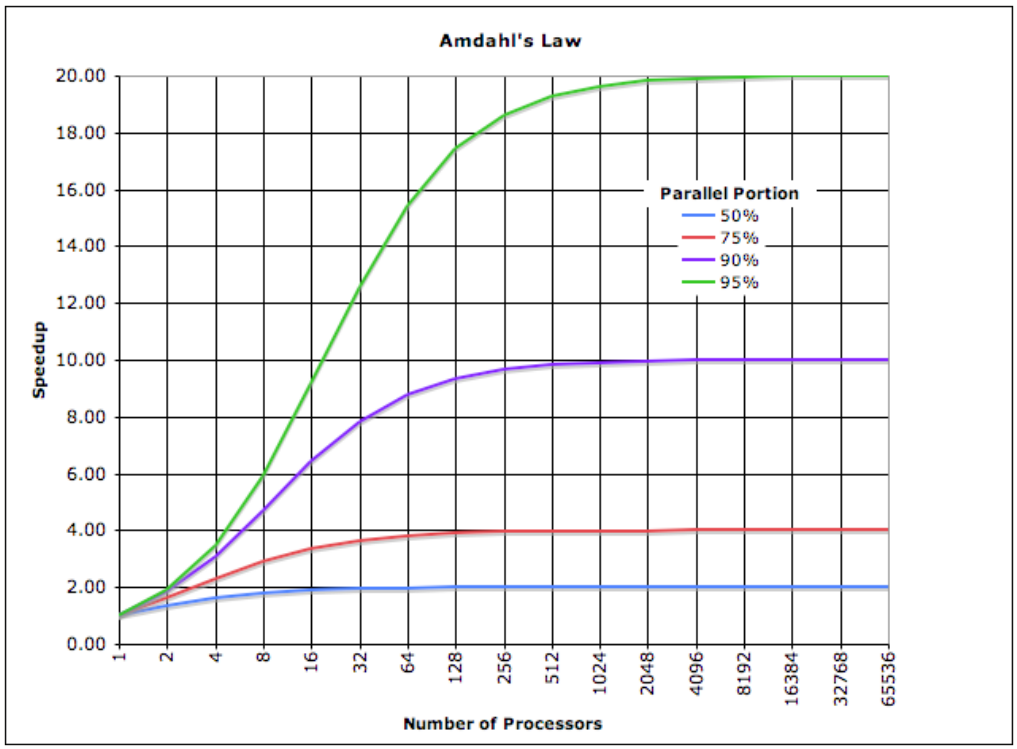

Fig. 24 Legge di Amdahl al variare del livello di parallelizzazione del programma e del numero di processori

Se nel grafico si considera un certo numero fisso di processori, ci si accorge che, a seconda di quanto il codice sia stato parallelizzato, lo speedup può variare di molto. Ad esempio, considerando 16 processori; col 50% di parallelizzazione si ha uno speedup pari a 2, col 75% circa 3, col 90% circa 6 ed infine col 95% circa 9 (quasi 5 volte quello iniziale!).

In generale, escludendo i tratti iniziali delle curve, si può affermare che aumentare il livello di parallelizzazione rende sicuramente di più rispetto a raddoppiare il numero di processori. Basti considerare la curva col 90 % di parallelizzazione dove in corrispondenza a 32 processori si ha uno speedup pari a 8. Se si raddoppiasse il numero di processori si otterrebbe uno speedup pari a 9, mentre se si parallelizzasse solo il 5% di codice in più si otterrebbe uno speedup maggiore e cioè quasi pari a 13 (curva verde).

L'ultima considerazione che emerge dal grafico è quella che si è dedotta dall'analisi analitica sopracitata e cioè che dopo un certo numero di processori lo speedup non aumenterà più ma tenderà al valore $\frac{1}{1-P}$. Infatti dal grafico si vede che le curve del 50%, 75%, 90% e 95% non crescono più una volta raggiunti rispettivamente i 16, 128, 1024 e 8192 processori.

4.3 Legge di Gustafson

La legge di Gustafson (Fig. 25) afferma che i problemi con grandi e ripetitivi gruppi di dati possono essere efficacemente parallelizzati. Questa legge è per questo motivo in contraddizione con quella di Amdahl. La legge di Gustafson, definita per la prima volta dall'informatico John L. Gustafson con l'aiuto del collega Edward H. Barsis, è la seguente:

$$S(P) = P - \alpha \cdot (P - 1)$$

Dove:

- P è il numero di processori;
- S è il fattore di speedup;
- α è la parte non parallelizzabile del programma.

La legge di Gustafson affronta i deficit della legge di Amdahl, la quale non considera che all'aumentare del numero di macchine aumenta anche la potenza di calcolo disponibile. La legge di Gustafson suggerisce ai programmatori di impostare innanzitutto il tempo concesso per la risoluzione in parallelo del problema, considerando tutti i calcolatori a disposizione, e sulla base di quello dimensionare il problema. Pertanto, più il sistema parallelo a disposizione è veloce, più grandi sono i problemi che possono essere risolti nello stesso arco di tempo.

L'effetto della legge di Gustafson è stato quello di orientare gli obiettivi della ricerca informatica verso la selezione o riformulazione dei problemi in modo tale che la soluzione di un problema più ampio sarebbe comunque possibile nella stessa quantità di tempo. Inoltre, la legge ridefinisce il concetto di efficienza come necessità di ridurre al minimo la parte sequenziale di un programma, nonostante l'aumentare del carico di lavoro.

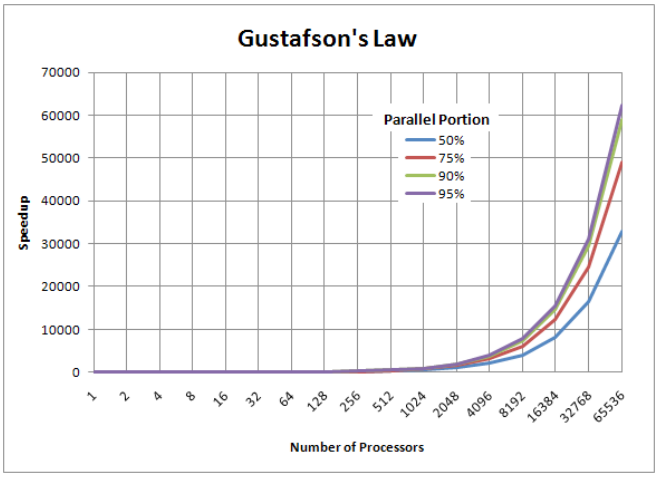

Fig. 25 Legge di Gustafson al variare del livello di parallelizzazione del programma e del numero di processori

4.4 Parametro di Karp-Flatt

Questo modello, proposto da Alan H. Karp e Orazio P. Flatt nel 1990, integra le leggi di Amdahl e Gustafson fornendo una misura precisa circa il grado di parallelizzazione del codice.

Dato un problema di calcolo parallelo eseguito ad un certo speedup σ con l'ausilio di p processori, dove $p>1$, il parametro di Karp-Flatt si calcola secondo la seguente formula:

$$e = \frac{\frac{1}{\sigma} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Dove più piccolo è il valore di e, migliore è il livello di parallelizzazione.

Questo parametro si occupa delle carenze proprie delle altre leggi e parametri utilizzati per misurare la parallelizzazione del codice informatico. In particolare, la legge di Amdahl non tiene conto della possibilità di bilanciare il carico di lavoro sui processori, né prende in considerazione possibili casi di problemi di gestione. Utilizzando questo indice si ottengono numerosi vantaggi rispetto ai predenti modelli, anche al crescere del numero di processori.

Per un problema di dimensione fissa, l'efficienza di un calcolo parallelo tipicamente diminuisce all'aumentare del numero di processori. A questo proposito è stato introdotto un ulteriore indice, ottenuto sperimentalmente usando il parametro di Karp-Flatt e definito come:

$$T(p) = T_s + \frac{T_p}{p}$$

Dove:

- $T(p)$ è il tempo totale necessario per l'esecuzione del codice in un sistema a p processori;
- Ts è il tempo necessario per eseguire la parte seriale;
- Tp è il tempo necessario per eseguire la parte parallela utilizzando un unico processore;
- P è il numero di processori.

Grazie a questo indice è possibile determinare se il calo di efficienza è dovuto ad eventuali limiti di parallelizzazione oppure al crescere dei problemi di gestione dell'algoritmo o dell'architettura.

4.5 Modello Roofline

Siccome le prestazioni e la scalabilità delle architetture multicore e multiprocessore possono essere estremamente difficili da intuire per i programmatori meno esperti ed è importante che in genere tutti i programmatori capiscano a fondo questi concetti, è stato creato il modello Roofline. Questo è un tipo di modello grafico e intuitivo che fornisce previsioni realistiche sulle prestazioni e la produttività dei computer, evidenzia i limiti hardware relativi ad un particolare kernel ed infine mostra i potenziali benefici e i gradi di priorità delle ottimizzazioni che si vogliono apportare. Inoltre tale modello tiene conto anche degli aspetti più importanti da considerare nell'ambito delle prestazioni e della scalabilità delle macchine parallele:

- Potenza di calcolo: nonostante ci siano diversi tipi di dati, quello a virgola mobile è quello preso in considerazione dal modello Roofline ed in particolare quello a doppia precisione. Per il test sulle prestazioni si devono quindi analizzare calcoli fatti con numeri in virgola mobile e tale misurazione si esprime in Floating Point Operations Per Second (numero di operazioni in virgola mobile eseguite in un secondo dalla CPU). Il picco di prestazioni in virgola mobile per chip multicore è la somma dei picchi dei singoli core.

- Intensità delle comunicazioni: il parametro preso in considerazione qui è la larghezza di banda della memoria, che si misura in Giga Byte Per Second (miliardi di byte trasferiti al secondo). Infatti la memoria incide molto sul tempo di risoluzione del problema, in quanto le unità aritmetiche devono in ogni caso operare sui dati memorizzati e leggerli e scriverli sono operazioni relativamente lente. Tuttavia è possibile ottimizzare i tempi di latenza [21] tramite una giusta scelta del tipo di memoria e l'uso di tecniche di accelerazione come il prefetching sulle istruzioni [22]. Inoltre anche le ottimizzazioni dal punto di vista software possono aiutare ad evitare dei cache miss [23] e quindi a minimizzare le comunicazioni inutili da e verso le memorie.

- Locazione delle memorie: la logica di questo aspetto è che più memorie ci sono e più sono distribuite, meno frequenti o comunque più veloci saranno le comunicazioni. Opportuni cambiamenti nell'hardware possono facilmente limitare casi di cache miss come i capacity miss o i conflict miss, ma soprattutto minimizzare il livello di traffico obbligatorio.

Il modello Roofline consiste appunto in un grafico che considera tutte e tre le caratteristiche sopracitate ma analiticamente cosa relaziona i GFlop/s ai GB/s? La risposta a questa domanda è l'indice chiamato Intensità Aritmetica (Fig. 26) [24], calcolato secondo la seguente equazione:

$$ArithmeticIntensity = \frac{ComputationPerformance}{MemoryBandwidth} = \frac{FloatingPoint\ Operations/Sec}{FloatingPoint\ Operations/Byte} = Byte/Sec$$

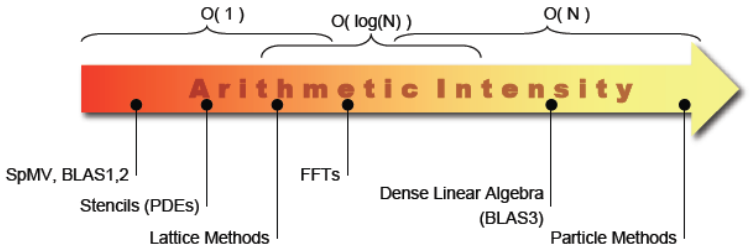

Fig. 26 Intensità Aritmetica

Tale indice è la stima del numero di richieste compiute da un programma alla memoria del sistema, più tecnicamente è il numero di operazioni in virgola mobile per byte di memoria acceduta. Si noti che utilizzando nell'equazione i byte totali di memoria, complessivi di tutte le cache, si terrà in considerazione anche la locazione delle memorie.

In Fig. 27 è finalmente presentato il grafico, relativo a due generici kernel:

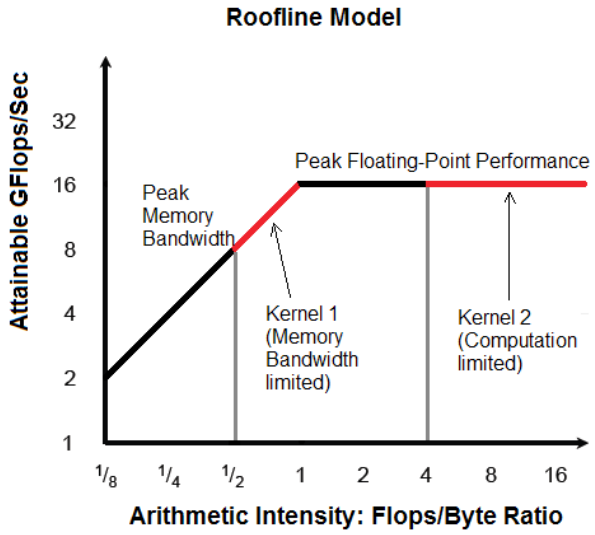

Fig. 27 Modello Roofline con casi di limiti, di larghezza di banda di memoria e potenza di calcolo, presenti

Come si può osservare, a seconda del valore assunto dall'indice di Intensità Aritmetica, un kernel può avere un picco di prestazioni più o meno elevato. La determinazione analitica del picco massimo è dato dalla seguente formula:

$$\text{GFlops/Sec\_effettivi} = \text{Min}\begin{cases}\text{Larghezza di banda della memoria} \cdot \text{Intensità Aritmetica} \\ \text{Picco delle prestazioni di calcolo}\end{cases}$$

Si noti l'esistenza di due limiti; il primo, dato dal segmento obliquo e incidente sul kernel numero 1, è la larghezza di banda e il secondo, dato dal segmento orizzontale e incidente sul kernel numero 2, è la potenza di calcolo. Un'altra cosa da notare è che più il punto di spezzamento dei due segmenti tende a destra nel grafico, più alta sarà l'Intensità Aritmetica richiesta ad un kernel per raggiungere il picco massimo di prestazioni. Invece più è tendente alla parte sinistra, minore sarà l'Intensità Aritmetica richiesta e quindi potenzialmente qualsiasi kernel potrà raggiungere il picco massimo.

Inoltre, il modello suggerisce anche che ottimizzazioni si possono fare, e in che ordine, al fine di aumentare le prestazioni della macchina:

1. Far sì che ci sia un numero equo di operazioni di addizione e moltiplicazione eseguite simultaneamente, in modo da sfruttare al massimo l'hardware.

2. Modificare il codice favorendo il parallelismo a livello di istruzioni ed applicare la tecnica SIMD che permette di lavorare con coppie di operandi a precisione doppia alla volta.

3. Caricare in anticipo le istruzioni in memoria senza aspettare che il flusso dati sia pronto, risparmiando così sul tempo di estrazione delle istruzioni.

4. Incrementare l'affinità di memoria e cioè far in modo che i dati e i relativi thread si trovino entrambi nella memoria dello stesso chip, senza dover andare in quella di altri e provocando così tempi di latenza maggiori.

In Fig. 28 sono evidenziate le quattro ottimizzazioni sopracitate; si noti che in certe zone del grafico esse possono essere singole o combinazioni delle tali.

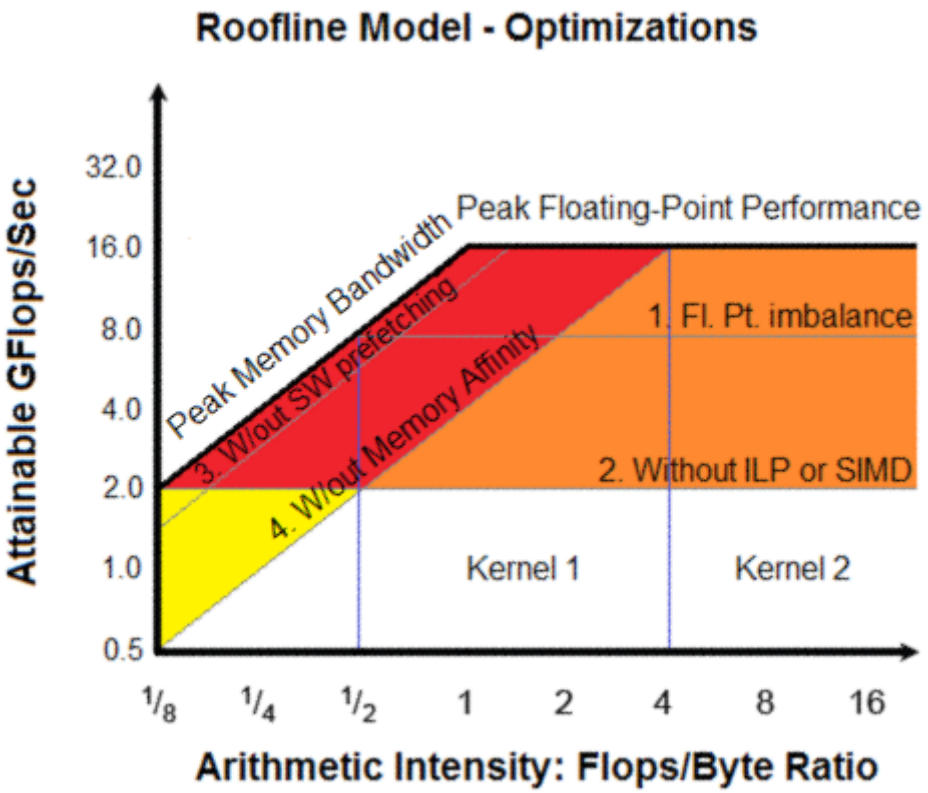

Fig. 28 Ottimizzazioni suggerite dal modello Roofline

Per un'esemplificazione sull'utilizzo pratico di tale modello si consiglia di consultare *Computer Organization and Design, 4th Ed, D. A. Patterson and J. L. Hennessy* al paragrafo 7.11.

5. Conclusioni

Attraverso i precedenti capitoli, si è visto come sia complicato ottenere grandi prestazioni facendo uso di calcolatori paralleli e volendo garantire un certo grado di efficienza. Infatti la riuscita o meno della creazione di un computer parallelo molto potente dipende da diverse scelte: il tipo di architettura, il modo di gestire la memoria, il tipo di multithreading, ecc. Molte ottimizzazioni si possono ottenere, sia via hardware che via software, utilizzando strumenti di analisi come quelli introdotti nell'ultimo capito. In generale i punti, riguardanti i sistemi multiprocessore e multicore, da tenere in considerazione sono i seguenti:

- Indispensabilità: queste macchine sono indispensabili se si vuole risolvere grandi problemi di calcolo in tempi relativamente brevi.

- Produttività: riducendo il tempo necessario alle macchine per svolgere un determinato compito è possibile aumentare il carico di lavoro a loro affidato, pur rimanendo dentro i limiti di tempo prestabiliti.

- Affidabilità: questi sistemi si rivelano molto affidabili anche nel caso in cui uno dei processori smetta di funzionare perché gli altri possono sempre continuare ad elaborare il thread/processo in esecuzione.

- Affinità: le prestazioni ottenibili variano a seconda del tipo di problema di calcolo, infatti la cosa migliore sarebbe costruire una macchina ad-hoc per il problema che si vuole risolvere.

- Parallelizzazione: per ottenere il massimo livello di parallelizzazione possibile, opportune modifiche vanno eseguite sia a livello hardware che a livello software. Deve essere garantita la presenza di dispositivi capaci di fare calcoli aritmetici simultaneamente e allo stesso tempo di programmi progettati secondo una logica parallela (come conferma la Legge di Amdahl).

- Innovazione: il fatto che il software possa fare la differenza in termini di prestazioni, una volta che è stato fatto tutto il possibile dal punto di vista hardware, spinge i programmatori di tutto il mondo a creare software e linguaggi di programmazione sempre più moderni ed efficienti.

Elenco degli acronimi

ALU	Arithmetic Logic Unit
CMT	Coarse-Grained Multithreading
COMA	Cache Only Memory Access
COW	Cluster Of Workstations
CPU	Central Processing Unit
CU	Control Unit
DLP	Data Level Parallelism
DSM	Distributed Shared Memory Systems
FLOPS	Floating Point Operations Per Second
FMT	Fine-Grained Multithreading
GBPS	Giga Byte Per Second
GFLOPS	Giga Floating Point Operations Per Second
ILP	Instruction Level Parallelism
MIMD	Single Instruction Multiple Data
MISD	Multiple Instruction Single Data
MP	Message Passing
MPP	Massively Parallel Processing
NORMA	No-Remote Memory Access
NUMA	Non Uniform Memory Access
OOP	Out of Order Processing
PC	Personal Computer
PC	Program Counter
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMP	Simmetric Multiprocessor
SMT	Simultaneous Multithreading
TLP	Thread Level Parallelism
UMA	Uniform Memory Access

Elenco delle figure

Bibliografia

1 S. Congiu (2007), *Architettura degli elaboratori, Organizzazione dell' hardware programmazione in linguaggio assembly,* Quinta Edizione, Pàtron Editore

2 D. A. Patterson e J. L. Hennessy (2008), *Computer Organization and Design, The Hardware/Software Interface,* Quarta Edizione, Morgan Kaufmann

3 A. S. Tanenbaum (2006), *Architettura dei calcolatori, Un Approccio Strutturale,* Quinta Edizione, Prentice Hall

4 W. Stallings (2004), *Architettura e Organizzazione dei Calcolatori, Progetto e Prestazioni,* Sesta Edizione, Addison Wesley

5 A. Murli (2006), *Lezioni di Calcolo Parallelo*, Prima Edizione, Liguori

6 A. D. Pimentel, S. Vassiliadis (2004), *Computer Systems: Architectures, Modeling and Simulation,* Quarta Edizione, Springer

7 K. Popovici, F. Rousseau, A. A. Jerraya e M. Wolf (2010), *Embedded Software Design and Programming of Multiprocessor System-on-Chip,* Prima Edizione, Springer

8 Congiu S. (2007), *Architettura degli elaboratori, Organizzazione dell'hardware programmazione in linguaggio assembly,* Quinta Edizione, Pàtron Editore

9 W. Stallings (2004), *Architettura e Organizzazione dei Calcolatori, Progetto e Prestazioni, Sesta Edizione,* Addison Wesley

10 A. S. Tanenbaum (2006), *Architettura dei calcolatori, Un Approccio Strutturale,* Quinta Edizione, Prentice Hall

11 A. S. Tanenbaum e M. Van Steen (2007), *Sistemi Distribuiti,* Seconda Edizione, Prentice Hall

12 K. Popovici, F. Rousseau, A. A. Jerraya e M. Wolf (2010), *Embedded Software Design and Programming of Multiprocessor System-on-Chip,* Prima Edizione, Springer

13 A. Murli (2006), *Lezioni di Calcolo Parallelo,* Prima Edizione, Liguori Editore

14 I. A. Dhotre (2009), *Operating System,* Prima Edizione, Technical Publications Pune

15 Congiu S. (2007), *Architettura degli elaboratori, Organizzazione dell'hardware programmazione in linguaggio assembly,* Quinta Edizione, Pàtron Editore

16 T. J. Fountain (2006), *Parallel Computing, Principles and Practice,* Prima Edizione, Cambridge University Press

17 A. S. Tanenbaum e M. Van Steen (2007), *Sistemi Distribuiti,* Seconda Edizione, Prentice Hall

18 D. A. Patterson e J. L. Hennessy (2008), *Computer Organization and Design, The Hardware/Software Interface,* Quarta Edizione, Morgan Kaufmann

19 T. J. Fountain (2006), *Parallel Computing, Principles and Practice,* Prima Edizione, Cambridge University Press

20 W. Stallings (2004), *Architettura e Organizzazione dei Calcolatori, Progetto e Prestazioni,* Sesta Edizione, Addison Wesley

21 A. D. Pimentel, S. Vassiliadis (2004), *Computer Systems: Architectures, Modeling and Simulation,* Quarta Edizione, Springer

22 D. A. Patterson e J. L. Hennessy (2008), *Computer Organization and Design, The Hardware/Software Interface,* Quarta Edizione, Morgan Kaufmann

23 K. Popovici, F. Rousseau, A. A. Jerraya e M. Wolf (2010), *Embedded Software Design and Programming of Multiprocessor System-on-Chip,* Prima Edizione, Springer

24 I. A. Dhotre (2009), *Operating System,* Prima Edizione, Technical Publications Pune