1 Lezione del 21-10-25

1.1 Tassonomia di Flynn

Prima di venire alla sincronizzazione dei processi, vediamo brevemente la **classificazione delle architetture** attraverso la *tassonomia di Flynn*.

Questa è una classificazione che vede un sistema di elaborazione da 2 punti di vista ortogonali:

- La capacità di avere più flussi di **esecuzione**: si possono distinguere **SI** (*Single Instruction Stream*) e **MI** (*Multiple Instruction Stream*). Questo concetto è vicino a quello di *thread* visto nella scorsa sezione;
- La capacità di avere più flussi di **dati**: si possono distinguere **SD** (*Single Data Stream*) e **MD** (*Multiple Data Stream*);

Abbiamo quindi la prima distinzione:

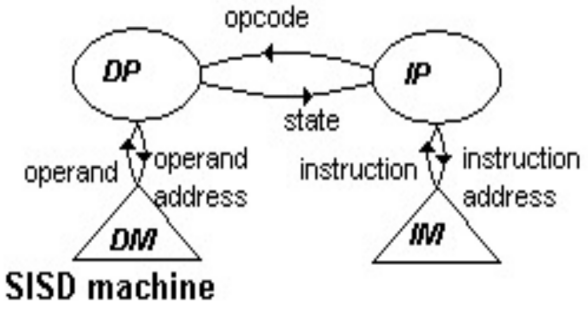
	SI (*Single Instruction Stream*)	**MI (*Multiple Instruction Stream*)**
SD (*Single Data Stream*)	Macchine SISD	Macchine MISD
MD (*Multiple Data Stream*)	Macchine SIMD	Macchine MIMD

Iniziamo a vedere dove si collocano le macchine che conosciamo.

1.1.1 Macchine SISD

Le macchine SISD rappresentano le tradizionali macchine *sequenziali* e *monoprocessore* definite dall'architettura di Von Neumann. In questo caso si ha un solo flusso di istruzioni, ciascuna agente su al più un flusso dati, e ad ogni istante temporale si esegue una singola istruzione.

Vediamo una schematizzazione di questa architettura coerente con Flynn:



Da questa schematizzazione notiamo:

- Un'unita di elaborazione **dati**, detta **DP** (*Data Processor*), che interagisce ottenendo operandi e fornendo indirizzi di operandi (e sperabilmente dati) con uno stream **dati**, detto **DM** (*Data Memory*);
- Un'unità di elaborazione **istruzioni**, detta **IP** (*Instruction Processor*), che interagisce ottenendo istruzioni e fornendo indirizzi di operazioni con uno stream **istruzioni**, detto **IM** (*Instruction Memory*).

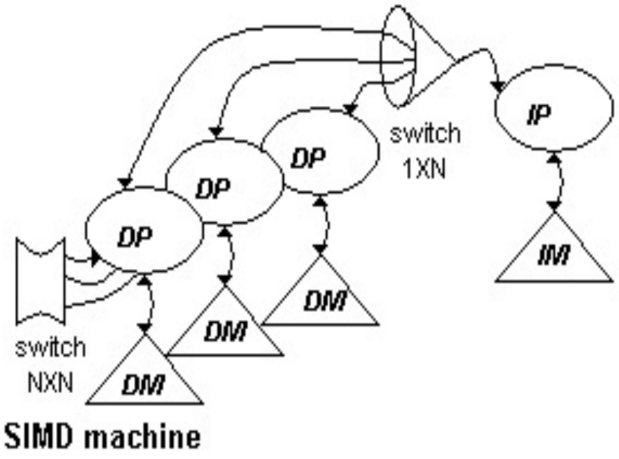
IP e DP interagiscono scambiandosi codifiche di istruzioni (IP $\rightarrow$ DP) e variazioni di stato (della memoria dati, DP $\rightarrow$ IP).

Possiamo notare che la cosiddetta *architettura Harvard* è un architettura che prevede, come dalla schematizzazione sopra, una forte separazione fra stream istruzioni e dati (di contro alla Von Neumann, che prevede un unica fonte di memoria per dati e istruzioni). Entrambe le architetture sono classificate come SISD, la Harvard è più usata in sistemi real-time mentre la Von Neumann è ancora oggi più usata nei sistemi general purpose.

1.1.2 Macchine SIMD

Le macchine SIMD sono formate da unità di elaborazione multiple, che eseguono le stesse istruzioni *contemporaneamente*, ma su flussi di dati differenti.

Una schematizzazione simile a quella sopra riportata è la seguente:



Abbiamo che questa moltiplicazione dei flussi dati su cui si elabora si ha moltiplicando le unità di elaborazione dati (i **DP**), facendole obbedire ad una singola unità di elaborazione istruzion (l'**IP**). Vediamo innanzitutto come si realizza la sincronizzazione fra questi DP: si prevede uno *switch* $1 \times N$ che porta le codifiche di istruzioni dall'IP a tutti i DP.

Per l'interazione fra le DP prevediamo poi uno switch $N \times N$ che le collega. Chiaramente questo switch sarà inefficiente, e vorremo usarlo il meno possibile.

Architetture di questo tipo possono essere *regolari* o create *ad hoc* sulla base della struttura del problema: nel caso di architetture regolari (cioè che rispettano la topologia fisica) non si hanno conflitti, e questo le rende efficienti e poco costose.

Le applicazioni di un'architettura di questo tipo sono nel caso di operazioni fortemente vettorizzate, come ad esempio nelle applicazioni grafiche e multimediali. Inoltre, questo è il tipo di architettura che incontriamo spesso per macchine che devono portare avanti moli massiccie di computazione come i *supercomputer*.

Riassumendo, possiamo dire che l'architettura SIMD prevede 2 tipi di parallelismo:

- **Parallelismo temporale**: c'è un meccanismo di *pipeline*, cioè fasi diverse di un'unica istruzione sono eseguite in parallelo in differenti moduli connessi in cascata.
- **Parallelismo spaziale**: i medesimi passi sono eseguiti contemporaneamente su un array di processori perfettamente uguali, sincronizzati da un solo controllore.

Lato programmatore possiamo prevedere due paradigmi per la compilazione di programmi pensati per l'esecuzione su macchine SIMD:

- Il primo modo è non riscrivere il codice, sapendo che un programma pensato come scalare su macchina sequenziale (SISD), in esecuzione su macchina SIMD sarà vettoriale.

  Avremo quindi che, ad esempio:

```
// somma scalari
c = a + b
```

  su macchina SIMD diventerà:

```
// soma vettori (!)
C = A + B
```

- Nel caso si voglia essere più espliciti nel tipo di operazioni che facciamo, possiamo implementare un **compilatore vettoriale**: l'idea è che questo riconosca automaticamente quando le istruzioni SIMD potrebbero tornare utili per parallelizzare delle operazioni vettoriale, e inserisca quindi le operazioni necessarie.

  Ad esempio, potremmo volere:

```
for(int i = 0; i < 100; i++) {
    c[i] = a[i] + b[i];
}
// qui riconosciamo che il ciclo e' vettorizzabile, e quindi lo
    vettorizziamo
```

1.1.3 Macchine MISD

In una macchina MISD vogliamo avere più flussi di istruzioni che lavorano contemporaneamente su un unico flusso dati.

Abbiamo che questa categoria è sostanzialmente vuota: il parallelismo fra più istruzioni in esecuzione sullo stesso flusso dati si ha effettivamente nei processori moderni solo attraverso il meccanismo della **pipeline**.

Se prevediamo che il processore debba svolgere per ogni istruzione più fasi, fra cui ad esempio:

1. Prelievo istruzione;
2. Decodifica;
3. Prelievo operandi;
4. Esecuzione;
5. Scrittura.

Avremo che questo potrà, disponendo di più unità di elaborazione atte a completare ognuna di queste fasi, parallelizzare come segue:

	Prelievo istruzione	Decodifica	Prelievo operandi	Esecuzione	Scrittura
t_0	i				
t_1	$i+1$	i			
t_2	$i+2$	$i+1$	i		
t_3	$i+3$	$i+2$	$i+1$	i	
t_4	$i+4$	$i+3$	$i+2$	$i+1$	i

In questo, a regime (nella tabella tempo t_4) si avranno più di un unità di elaborazione a lavoro contemporaneamente, e potremmo dire di aver realizzato in qualche modo il paradigma MISD.

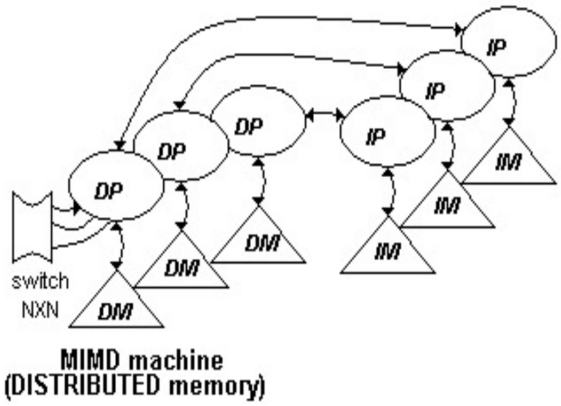
Facciamo tra l'altro una nota sull'efficienza: se l'esecuzione di un'istruzione senza pipeline richiedeva un tempo Δt, e il troughput era $\frac{1}{T}$, prevedendo una pipeline ad n stadi si riesce ad arrivare a $\frac{1}{T} \times n$ (assunto che ogni stadio richieda lo stesso tempo e si riesca a ridurre l'overhead dato da n troppo grandi).

1.1.4 Macchine MIMD

Le macchine MIMD rappresentano per noi la categoria più interessante più flussi di istruzioni sono in esecuzione contemporaneamente su più processori, elaborando insiemi di dati distinti, privati o condivisi.

Ne prevediamo due tipologie principali:

- **DM-MIMD** (*Distributed Memory MIMD*), cioè a *memoria distribuita*. Queste si schematizzano come segue:



In questo caso abbiamo più coppie IP-DP (con relative memorie IM e DM), che rappresentano sostanzialmente più macchine SISD. Uno switch $N \times N$ permette quindi la comunicazione fra le unità.

Abbiamo quindi che tra i nodi non esiste memoria condivisa e ogni nodo esegue indipendentemente un flusso di istruzioni su un differente insieme di dati, memorizzati su spazi differenti. La comunicazione è realizzata mediante una sottorete dedicata (appunto, lo switch).

Applicazioni di questa architettura sono ad esempio una qualsiasi *rete di calcolatori* (come sia sviluppato lo switch non è specificato dal modello, e per noi potrebbe essere anche un router internet). Questo è il caso dei *cluster* di workstation, realizzati solitamente attraverso *Ethernet*. In questo richiediamo dalle macchine che compongono la rete 2 caratteristiche principali:

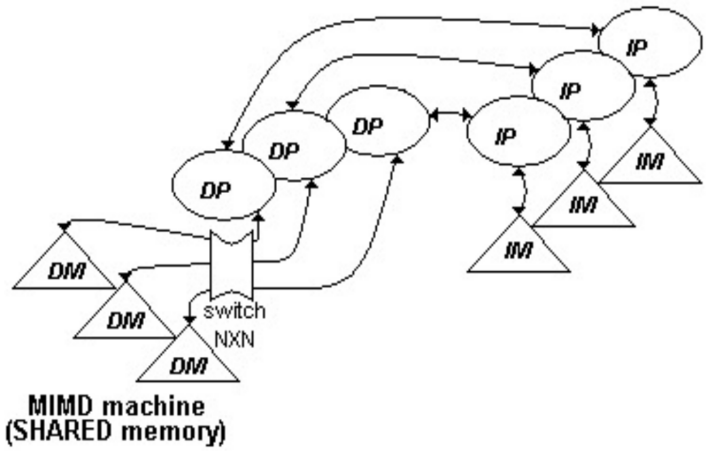
1. **High-availability**: in caso di guasti, la computazione può migrare da un nodo all'altro;
2. **Load-balancing**: i task da eseguire sono allocati nei nodi che hanno il minor carico.

Architetture MIMD più stabili sono poi le rete di interconnessione regolari e dirette (ipercubi, mesh, torus), attraverso cui i nodi si scambiano informazioni secondo il paradigma del *message passing* (*"scambio di messaggi"*). Queste macchine sono molto scalabili e si prestano bene ad algoritmi ad elevata località: sono stati costruiti cluster composti anche da milioni di unità sequenziali.

Una variante del DM-MIMD è il **DM-MIMD MPP** (*Massively Parallel Processing*). Questo è un paradigma utile in applicazioni scientifiche e particolari contesti di calcolo commerciale-finanziario. In un sistema MPP si ha:

  - Migliaia di nodi (CPU standard, ognuna con la propria memoria e la propria copia del SO)
  - Una rete di interconnessione custom molto potente (larga banda e bassa latenza). Affinché l'elaborazione MPP dia effettivi vantaggi occorre disporre di software capace di partizionare il lavoro e i dati su cui opera tra i vari processori.

- **SM-MIMD** (*Single Memory MIMD*), cioè a *memoria condivisa*. Queste si schematizzano come segue:



  Sono quindi macchine sempre *multiprocessore*, ma dove le varie unità di elaborazione dati comunicano attraverso uno switch $N \times N$ con un *pool* unico di memoria (formato anche da più flussi di memoria, ma visti allo stesso modo da ogni unità di elaborazione dati).

  Questo approccio è meno scalabile, realizza la comunicazione fra processori condividendo aree di memoria, e richiede una rete di interconnessione (lo switch $N \times N$ estremamente efficiente). Vogliamo che il numero N di processori sia piccolo ($N < 100$), affiché si possa avere stretto accompiamento fra i nodi. Si incorre chiaramente in problemi di competizione fra unità di elaborazione (mutua esclusione e sincronizzazione) che potrebbero impattare le prestazioni.

1.1.5 Confronto fra SIMD e MIMD

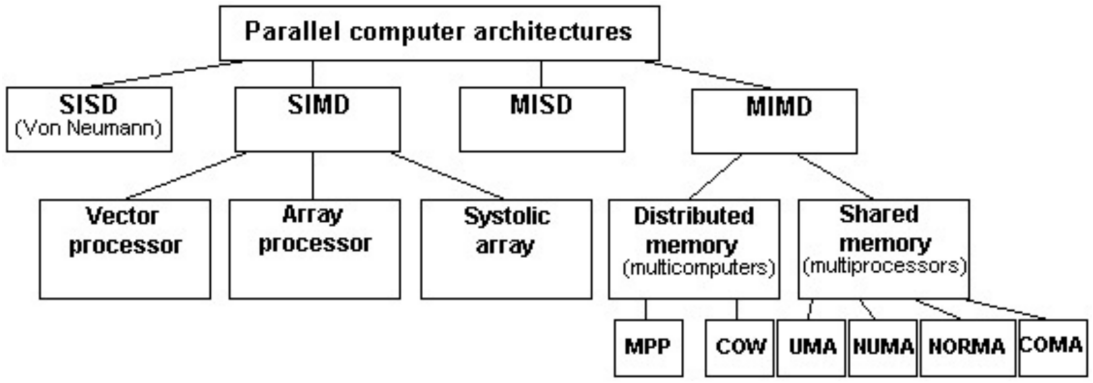
Possiamo fare quindi un confronto fra le architetture apperentemente simili, SIMD e MIMD:

- Le SIMD richiedono meno hardware delle MIMD (c'è un unica unità di controllo, o *elaborazione istruzioni*);
- Le MIMD usano spesso processori general-purpose, quindi costano meno delle SIMD (che richiedono processori o ISA particolari, quindi meno hardware ma più specializzato);
- Le SIMD usano meno memoria delle MIMD (una sola copia del programma in memoria);

- Le MIMD godono di una grande flessibilità in termini di modelli computazionali supportati (si pensi *client-server*, *P2P*, ecc...). Di contro, è piuttosto semplice modificare un programma seqeunziale perché esegua su architettura SIMD in maniera vettorizzata.

1.1.6 Sintesi della tassonomia di Flynn

Possiamo quindi vedere un grafico riassuntivo delle tassonomie viste:

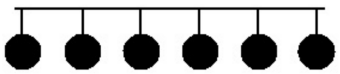


Dove si notano le 4 categorie principali (SISD, SIMD, MISD e MIMD) ed alcune sottocategorie (non abbiamo parlato di tutte le sottocategorie, per una trattazione più completa si rimanda alla letteratura).

1.2 Tipologie di interconnessione

Dopo aver discusso le architetture descritte dalla tassonomia di Flynn, vediamo i tipi principali di **interconnessione** che si possono avere fra unità di elaborazione (o in generale nodi).

1.2.1 Bus



Il **bus** è la più semplice rete di interconnessione che abbiamo visto. Rappresenta una configurazione semplice ed affidabile (a meno che non si rompa il bus tutto funziona), dove ogni nodo a *grado* 1 (tutti i nodi sono direttamente connessi al bus e a nient'altro), e il *diametro* è 1 (la distanza massima è data dal solo bus). Il numero totale di *link* di cui abbiamo bisogno è sempre 1: il bus stesso è l'unico link di cui necessitiamo (i nodi devono solo collegarsi a tale link).

Il problema è chiaramente la *competizione* sull'accesso al mezzo, che è massima: si hanno spesso problemi di mutua esclusione sulle stesse risorse, ad esempio quando più nodi vogliono accedere alla stessa risorsa contemporaneamente e devono farlo attraverso un unico bus.

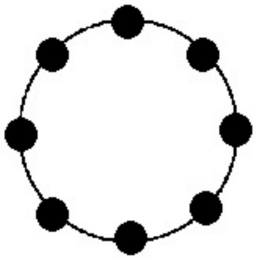
1.2.2 Array lineare

L'**array lineare** espande in qualche modo l'idea del bus: in un certo senso vogliamo partizionare il bus in tanti link nodo a nodo. In questo caso il grado del primo e dell'ultimo nodo sarà 1, mentre quello dei restanti nodi sarà 2. Il diametro sara $n - 1$ per n nodi, e il numero totale di link $n - 1$ (quelli necessari a legare ogni nodo con i nodi adiacenti).

Il vantaggio di questo approccio è la competizione, che viene ridotta al minimo (ogni coppia adiacente di nodi può comunicare indipendentemente dagli altri). Più nello specifico, nel caso ideale possiamo avere fino a $\frac{N}{2}$ competizioni contemporanee e parallele (appunto, una per ogni coppia).

I nodi dovranno quindi fornire servizi di *routing*, cioè permettere a nodi da un capo dell'array di comunicare con nodi dall'altro capo, prendendosi a carico in qualche modo il messaggio da comunicare. Questo è chiaramente a scapito della robustezza: se un nodo si rompe due parti dell'array lineare rimangono separate.

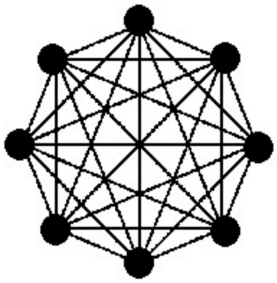
1.2.3 Ring



Il **ring** è un array lineare chiuso su stesso (dove si sono collegati i nodi estremi, cioè quelli con grado 1). Il grado diventa quindi 2 per tutti i nodi. Il diametro subisce la prima caratteristica fondamentale del ring: per raggiungere un dato nodo si hanno a disposizione due direzioni anziché una, per cui il diametro complessivo è $\frac{N}{2}$.

Anche la tolleranza ai guasti migliora, in quanto un nodo guasto non pregiudica necesssariamente l'integrità del sistema (ne servono 2 per isolare una parte della rete).

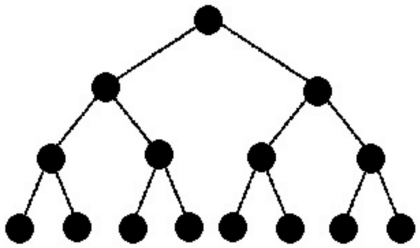
1.2.4 Connessione completa



La **connessione completa** (o *tutti-a-tutti*) è la rete di interconnessione più, appunto, *connessa*, che vediamo. In questo caso ogni nodo comunica direttamente con ogni altro nodo: il grado è per tutti i nodi $N - 1$ e il diametro è 1 (si arriva direttamente al nodo desiderato). Svantaggioso è chiaramente il numero totale di link, che cresce come $N\frac{N-1}{2}$: l'approccio chiaramente non è scalabile!

Dobbiamo quindi trovare modelli per reti di interconnessione che presentino parte dei vantaggi della connessione completa (alta tolleranza ai guasti, bassissimo diametro), riducendo però il numero di link e quindi aumentando la scalabilità.

1.2.5 Albero binario



Si può pensare di ordinare i nodi secondo un **albero binario**. In questo caso vorremo definire l'*altezza* dell'albero come $h = \log_2(N)$ per N nodi e i gradi come:

- 2 per la radice;
- 1 per le foglie;
- 3 per tutti gli altri nodi.

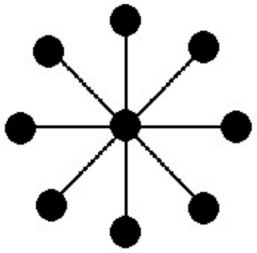
Il diametro sarà (senza dimostrazione) $2 \times (h - 1)$ e il numero totale di link $N - 1$ ($O(N)$ anziché $O(N^2)$, già migliore).

Questo tipo di rete permette una facile comunicazione fra nodi sugli stessi sottoalberi, mentre per comunicazioni fra sottoalberi distinti porta alla *congestione dei rami alti*: questo la rende poco scalabile. In particolare, più ci avviciniamo alla radice minori saranno i link (e quindi maggiore il carico sul singolo link). Inoltre, i nodi dovranno fare da router, e quindi più ci avviciniamo alla radice più i nodi hanno responsabilità di router sempre maggiori. Questo culmina sulla radice stessa, che chiaramente è sottoposta ad un carico non indifferente e rappresenta il punto più debole dell'architettura ad albero binario.

Per quanto riguarda la tolleranza ai guasti, vale lo stesso discorso: più in alto (verso la radice) avviene il guasto, maggiori sono le conseguenze per il sistema. Come caso limite, se si guasta la radice si incorre in un partizionamento in due dell'intero sistema.

Soluzioni alternative si possono avere sfruttando alberi, anziche binari, *n-ari*, cioè con n figli per nodo.

1.2.6 Stella



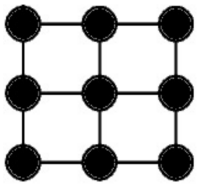
Le reti di interconnessione a **stella** prevedono un singolo nodo centrale che connette $n-1$ nodi periferici. Il grado di tale nodo centrale sarà quindi $n - 1$, mentre i nodi periferici

avranno 1. Il diametro sarà 2 (dobbiamo passare sempre dal nodo centrale, a meno che non si voglia parlare col nodo centrale stesso). Il numero di link è ridotto ($N - 1$), e quindi da questo punto di vista il sistema è vantaggioso.

Il difetto più grande è chiaramente la presenza di un singolo nodo centralizzato soggetto a guasti o sovraccarichi. Questo è il classico problema del *single point of failure* delle architetture client-server: possiamo infatti intendere il nodo centrale come un *server* e i nodi periferici come *client* di tale server.

Abbiamo quindi che per quanto si possa rendere potente il nodo server, questo dovrà portare tutto il carico della rete (bassa scalabilità), e un guasto del server rappresenterà una mancanza di servizio per tutti i nodi client (bassa robustezza).

1.2.7 Mesh bidimensionale



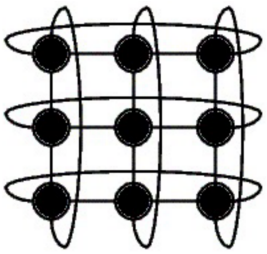
La struttura a **mesh bidimensionale** prevede di disporre i nodi su una griglia bidimensionale. In questo caso il lato della griglia sarà $r = \sqrt{N}$ e il grado dei nodi sarà:

- 2 per i nodi ai vertici,
- 3 per in nodi *"centrali"* ai lati (diciamo nodi agli *spigoli*);
- 4 per tutti gli altri nodi.

Il diametro sarà $2 \times (r - 1)$, e il numero totale di link ($2 \times (n - 2) \times r$).

La resistenza ai guasti di queste configurazioni è buona, ma può essere milgiorata. Vediamo come.

1.2.8 Toro bidimensionale



Collegando i nodi ai lati opposti di una struttura a mesh bidimensionale si ottiene una struttura a **toro bidimensionale**. In questo caso il grado è 4 per tutti i nodi, ma il diametro migliora: va sostanzialemente come r (dove r è il lato calcolato come $r = \sqrt{N}$, uguale alla rete bidimensionale). Il numero totale di link è invece stabile a $2N$.

Questa topologia risulta ben scalabile e notevolmente resistente ai guasti: sostanzialmente rappresenta per la mesh bidimensionale quello che il ring rappresenta per l'array lineare.

Architetture come le ultime viste (mesh, tori, ecc...) possono essere estese a più dimensioni, portando a *ipercubi*, *ipertori*, ecc...

1.3 Metriche di prestazione

Finiamo questa sezione del programma parlando di alcune **metriche** per le **prestazioni** di architetture descritte secondo la tassonomia di Flynn.

- Chiamiamo **speed-up** (S) il rapporto fra il tempo di esecuzione sequenziale sul tempo di esecuzione parallelo (SIMD o MIMD, cioè):

  $$S = \frac{T_1}{T_N}$$

  Questa metrica rappresenta quindi il *guadagno* in velocità che si ha passando ad un sistema multiprocessore. Idealmente vorremmo $S = N$, cioè *lineare* col numero di unità di elaborazione, ma come vedremo a causa dell'overhead introdotto da più processori dobbiamo accontentarci di $S < N$, con $S \approx N$.

  l valore dello speed-up dipende dalle applicazioni, ma anche dall'architettura: nelle SIMD spesso $S \approx N$, mentre nelle MIMD è difficile far crescere S (non è facile far lavorare pienamente tutte le CPU, o trasferire efficientemente fra di esse).

- L'**efficienza** (E) è invece definita come lo speed-up sul numero di unità di elaborazione, cioè:

  $$E = \frac{S}{N}$$

  Come prima, vorremmo $E = 1$, ma dobbiamo accontentarci di $E < 1$, con $E \approx 1$.

Per giustificare come mai non si può mai avere $S = N$ (o equivalentemente, $E = 1$), ci viene incontro la *legge di Amdhal*. Questa dice che un parallelismo *"perfetto"* (nelle varie attività compiute da un calcolatore) non è mai raggiungibile in quanto saranno *sempre* presenti sequenza di programmi *intrinsecamente seriali*. Semplificando, si può dire che ci saranno sempre degli intervalli di tempo sequenziale (T_{seq}), impiegati ad eseguire istruzioni non parallelizzabili come operazioni di I/O, costrutti condizionali, algoritmi intrinsecamente sequenziali, ecc....

Più nello specifico, la legge di Amdahl ridefinisce lo speed-up come:

$$S = \frac{T_1}{T_{\text{seq}} + \frac{T_1 - T_{\text{seq}}}{N}}$$

cioè il guadagno di speed-up è dato solo dalle parti parallelizzabili del programma ($T_1 - T_{\text{seq}}$), mentre le parti seriali (T_{seq}) non hanno grandi guadagni.

Il problema è che, prendendo il limite per $N \to \infty$, si ha:

$$\lim_{N\to\infty} S = \lim_{N\to\infty} \frac{T_1}{T_{\text{seq}} + \frac{T_1 - T_{\text{seq}}}{N}} = \frac{T_1}{T_{\text{seq}}}$$

cioè a dominare lo speed-up sono le parti sequenziali (le parti non sequenziali vengono abbattute velocemente, quelle sequenziali mai e rimangono come overhead complessivo dell'intero sistema).

1.3.1 Multitasking

In conclusione, vogliamo dire che il *multitasking* è quasi sempre vantaggioso (anche intuitivamente), ed è di notevole importanza anche nelle macchine per mantere alto lo sfruttamento delle CPU (ciò che avevamo chiamato *efficienza CPU*).

Il vincolo che vogliamo rispettare sarà, quindi, di base:

$$P >> N$$

cioè avere più processi che unità di elaborazione.

1.4 Sincronizzazione processi

Iniziamo quindi a vedere la **sincronizzazione** fra processi, in particolare con riferimento ai *tipi* di interazione che ci possono essere fra processi, e i problemi di **mutua esclusione** e **sincronizzazione**.

1.4.1 Tipi di interazione

Ricordiamo che i processi possono interagire fra di loro secondo 2 modalità:

- **Cooperazione**: quindi per sincronizzazione diretta o esplicita, cioè definita dai programmi;
- **Competizione**: quindi per sincronizzazione indiretta o implicita, non definita dal codice dei programmi ma causata da tentativi di accesso *simultaneo* a risorse limitati.

Nominiamo poi l'**interferenza**, rappresentata da errori dipendenti dal tempo.
Per l'interazione fra processi faremo riferimento a 2 modelli principali:

- A **memoria comune**: in questo caso prevediamo n processi con risorse private (cioè spazi di indirizzamento privati per ognuno), ma che possono accedere a risorse *condivise* in un terzo spazio di indirizzamento comune per tutti.

  Se vogliamo ricondurci alla tassonomia di Flynn, questo è un esempio di macchina multiprocessore (o monoprocessore in *timesharing*...) di tipo MIMD (se vogliamo SM-MIMD): più unità di elaborazione sugli stessi dati (tralasciando il fatto che ogni unità ha poi i suoi dati privati);

- A **scambio di messaggi**: in questo caso prevediamo n processi in esecuzione su unità di elaborazione distinte (ancora, multiprocessore o monoprocessore in *timesharing*) con le loro risorse (memorie) locali, che possono comunicare fra di loro attraverso un meccanismo di *scambio di messaggi*.

  Notiamo di nuovo che non è necessario che le unità siano necessariamente distinte, e ricordiamo che non sono i processi a comunicare in sé per sé, ma le unità di elaborazione a supportare un meccanismo che permette tale comunicazione.

  Questo è sempre un esempio di architettura MIMD (se vogliamo DM-MIMD), simile ad esempio a quella che ci permettono i sistemi multicalcolatore connessi in rete (più macchine distinte, con le loro risorse, che comunicano fra di loro).