SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" Università di Roma
Ingegneria dell'Informazione,
Informatica e Statistica
Dipartimento di Informatica

Programmazione di Sistemi Embedded e Multicore

Appunti integrati con il libro "An Introduction to Parallel Programming", Peter Pacheco

Author
Simone Bianco

15 novembre 2023

Indice

Informazioni e Contatti

Appunti e riassunti personali raccolti in ambito del corso di *Programmazione di Sistemi Embedded e Multicore* offerto dal corso di laurea in Informatica dell'Università degli Studi di Roma "La Sapienza".

Ulteriori informazioni ed appunti possono essere trovati al seguente link: **https://github.com/Exyss/university-notes**. Chiunque si senta libero di segnalare incorrettezze, migliorie o richieste tramite il sistema di Issues fornito da GitHub stesso o contattando in privato l'autore :

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

Gli appunti sono in continuo aggiornamento, pertanto, previa segnalazione, si prega di controllare se le modifiche siano già state apportate nella versione più recente.

Prerequisiti consigliati per lo studio:

Apprendimento del materiale relativo al corso *Sistemi Operativi II*.

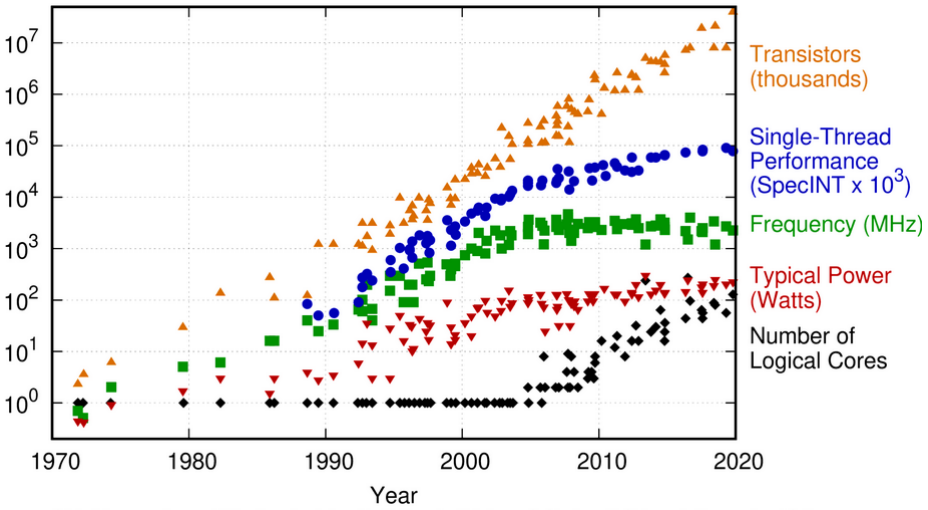
Licence:

1

Introduzione al Parallelismo

Dal 1986 al 2002, la velocità dei microprocessori è salita notevolmente, aumentando le prestazioni per una media del 50% annuo. Tuttavia, da tale periodo l'aumento è sceso a circa il 20% annuo.

Nonostante la famosa **legge di Moore**, la quale stabilisce che il numero di transistor nei circuiti integrati raddoppi circa ogni due anni, si riveli tuttora vera, le altre statistiche legate ai microprocessori non seguono tale trend:



Affinché il numero di transistor possa incrementare senza aumentare la superficie su cui posizionarli, nel corso degli anni sono state notevolmente ridotte le loro dimensioni. Procedendo in tal modo, tuttavia, risulta evidente il **limite fisico** di tale procedura: transistor più piccoli implicando processori più veloci, aumentando la quantità di energia consumata e di conseguenza anche il calore generato, portando i processori stessi ad essere inconsistenti.

Per tanto, non potendo più diminuire le dimensioni dei transistor e non volendo aumentare le dimensioni dei circuiti, l'unica soluzione risulta essere l'uso di **sistemi con più processori** in grado di comunicare tra di loro.

L'uso di architetture basate su più processori, tuttavia, richiedono la **conversione di programmi seriali in programmi paralleli**. Poiché alcuni costrutti di codice possono essere automaticamente riconosciuti e parallelizzati,inizialmente si provò ad effettuare la transizione tramite programmi di conversione automatica, i quali tuttavia generavano programmi inefficienti e poco ottimizzati. La soluzione migliore, dunque, risulta essere la **riscrittura** completa dei programmi basandosi su un **approccio parallelo**.

Supponiamo di voler calcolare n valori tramite una particolare funzione, per poi sommare tra di loro i valori ottenuti.

Un **approccio seriale** prevede la seguente soluzione banale:

```
sum = 0;
for(i = 0; i < n; i++){
    x = compute_next_value(...);
    sum += x;
}
```

richiedendo quindi l'uso di un singolo core per essere eseguito

Un **approccio parallelo**, invece, prevede l'uso di p core (dove $p < n$), assegnando ad ogni core un insieme di $\frac{n}{p}$ valori da calcolare. Per tanto, ogni core eseguirà il seguente frammento di codice:

```
my_sum = 0;
my_first_i = ...;
my_last_i = ...;
for(my_i = my_first_i; i < my_last_i; i++){
    my_x = compute_next_value(...);
    my_sum += my_x;
}
```

Osservazione 1: Variabili locali

Ogni variabile è **locale per ogni core**, ossia ogni core possiede una propria copia indipendente dalle altre su cui poter lavorare

Una volta che ogni core avrà terminato la computazione, le p somme parziali verranno sommate dal **master core**, ossia il core designato come principale:

```
if(I'm teh master core){
    sum = my_x;
    for each core other than myself{
        sum += value received from that core;
    }
}
else{
    send my_x to the master core;
}
```

In tal modo, ogni core secondario svolgerà $\frac{n}{p}$ somme, mentre il master core svolgera $\frac{n}{p} + p$ somme poiché deve sommare le p somme parziali al proprio risultato.

Notiamo quindi che tale soluzione presenta ancora alcune **inefficienze**, poiché solo il master core lavora durante la somma finale, mentre tutti gli altri core rimangono inutilizzati. La soluzione risulta ovvia: **distribuire** nuovamente il carico di lavoro.

Metodo 1: Computazione ad albero

Dato un problema di calcolo con p valori da eseguire su p core, la computazione parallela può essere sviluppata tramite il seguente algoritmo:

1. Ad ogni core viene assegnato un indice, dove 0 è il master core
2. Viene posto $k = 1$
3. Ogni processore con indice i multiplo di k somma il proprio valore calcolato x_i al valore x_j calcolato dal successivo core di indice j multiplo di k (dunque $x_i + x_j$)
4. Viene incrementato k, tornando al passo precedente finché la computazione non verrà svolta da un singolo core

In tal modo, ogni core svolgerà un massimo di $\frac{n}{p} + \log_2(p)$ somme parallele

Esempio:

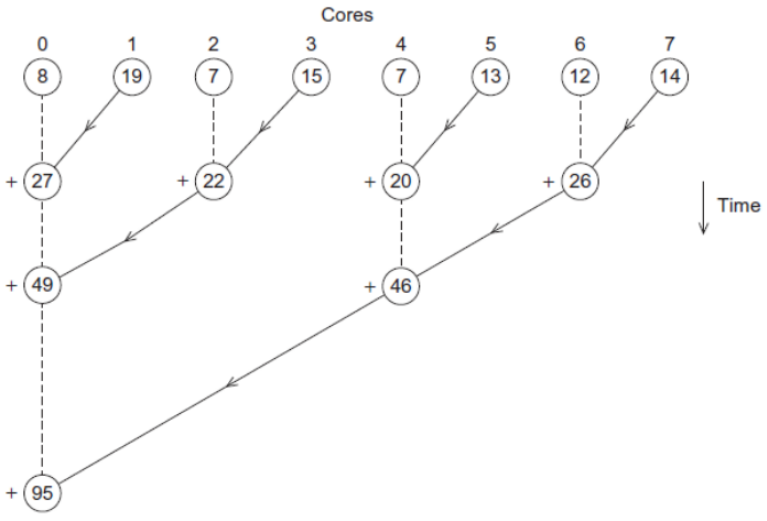
- Consideriamo il seguente insieme di $n = 24$ valori da sommare tra di loro:

$$1, 4, 3, 9, 2, 8, 5, 1, 1, 5, 2, 7, 2, 5, 0, 4, 1, 8, 6, 5, 1, 2, 3, 9$$

- Avendo $p = 8$ core, distribuiamo $\frac{n}{p} = \frac{24}{8} = 3$ valori a ciascun core:

$$1, 4, 3 \quad 9, 2, 8 \quad 5, 1, 1 \quad 5, 2, 7 \quad 2, 5, 0 \quad 4, 1, 8 \quad 6, 5, 1 \quad 2, 3, 9$$

- Una volta calcolate le somme parziali da parte di ogni core, la computazione verrà sviluppata ad albero:



Definizione 1: Task parallelism e Data parallelism

Dato un carico di lavoro, definiamo come:

- **Task parallelism** una distribuzione del carico basata sulla suddivisione dei compiti da svolgere
- **Data parallelism** una distribuzione del carico basata sulla suddivisione dei dati su cui lavorare

Esempio:

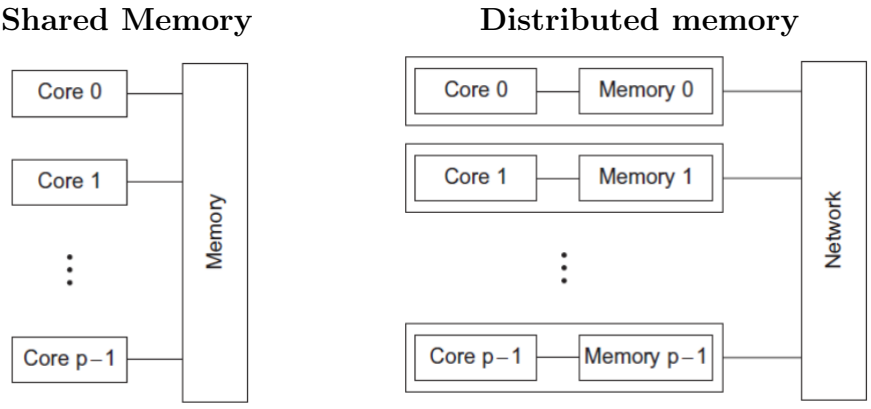
- Tre professori vogliono suddividersi la valutazione di 300 consegne di un esame composto da 15 domande.
- Una suddivisione basata sul task parallelism prevede che ogni professore corregga tutte le 300 consegne, ma che il primo professore corregga solo le domande dalla 1 alla 5, il secondo dalla 6 alla 10 e il terzo dalla 11 alla 15
- Una suddivisione basata sul task parallelism prevede che ogni professore corregga tutte le 15 domande, ma che ogni professore debba correggere solo 100 consegne

Definizione 2: Shared Memory System

Definiamo come **shared memory system** un sistema in cui i core possono condividere l'accesso alla memoria del computer, richiedendo coordinazione per svolgere operazioni di lettura e scrittura sul contenuto condiviso

Definizione 3: Distributed Memory System

Definiamo come **distributed memory system** un sistema in cui ogni core possiede la propria memoria, richiedendo uno scambio di messaggi tramite una rete di interconnessione per effettuare il passaggio di dati



1.1 Hardware delle architetture moderne

1.1.1 Cache, memoria virtuale e TLB

Le moderne architetture prevedono componenti hardware aggiuntivi rispetto all'architettura minimale basata sul **modello di Von Neumann**, il quale ricordiamo essere basato sulla sola presenza di una CPU costituita da una Control Unit (CU) ed una Arithmetic Logic Unit (ALU), una memoria principale e periferiche di I/O.

In particolare, le moderne CPU sono dotate di piccole **cache**, tipicamente situate sullo stesso chip, le quali sono accessibili più velocemente rispetto alla memoria principale.

Definizione 4: Caching

Definiamo come **caching** l'uso di una collezione di locazioni di memoria accessibili più velocemente rispetto ad altre locazioni, le quali possono essere all'interno della stessa o di una diversa memoria

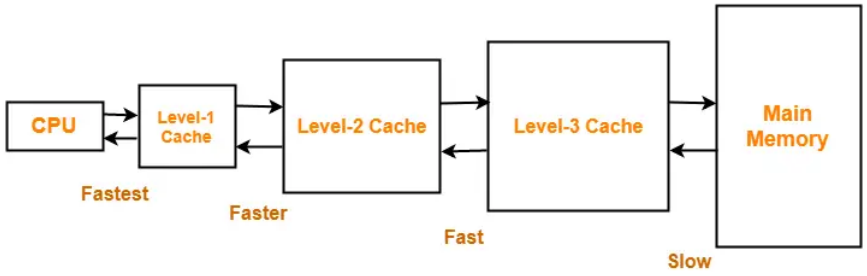
L'uso del caching genera un alto impatto sulle performance delle moderne architetture per via del **principio di località**.

Principio 1: Principio di località

Il **principio di località** afferma che durante l'esecuzione di un qualsiasi programma prevalgono le due seguenti proprietà:

- **Località spaziale**: gli accessi ad una locazione di memoria sono per lo più seguiti da accessi a locazioni ad essa adiacenti
- **Località temporale**: una locazione di memoria viene solitamente acceduta ripetute volte solo nel breve periodo di tempo, per poi non venir più acceduta durante l'esecuzione del programma

Per alleviare il **bottleneck** (*collo di bottiglia*) tra cache e memoria principale, viene utilizzata una **gerarchia di cache**, dove ogni sottolivello è costituito da una memoria più lenta ma anche più capiente rispetto alla precedente, fino a raggiungere la memoria principale stessa. Solitamente, l'ultima cache prima della memoria principale viene detta **Last Level Cache (LLC)**.



Essendo più piccole della memoria principale, ogni cache può gestire solo una **limitata quantità di dati**: quando un dato richiesto da parte di una CPU viene trovato all'interno della sua cache, esso viene immediatamente restituito (**cache hit**). In caso contrario, tale dato verrà cercato nella memoria di livello direttamente inferiore (**cache miss**), finché esso non verrà eventualmente trovato (nel caso peggiore verrà prelevato direttamente dalla memoria principale).

Per mantenere i propri dati, ogni cache è dotata di m **linee**, ognuna composta da n **set** all'interno dei quali vengono conservati byte di dati. Le mappature blocco-set vengono gestite tramite una delle seguenti modalità:

- **Direct-mapped cache**: la cache è dotata di m linee ciascuna con un singolo set
- n**-way set associative cache**: la cache è dotata di m linee ciascuna con m set
- **Fully-associative cache**: la cache è dotata di una sola linea contenente n set

In base alla mappatura utilizzata, ogni indice della memoria (corrispondete ad un byte) viene mappato ad una determinata linea. Tuttavia, essendo le linee limitate, più indici verranno **mappati sulla stessa linea**.

	Cache Location		
Memory Index	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

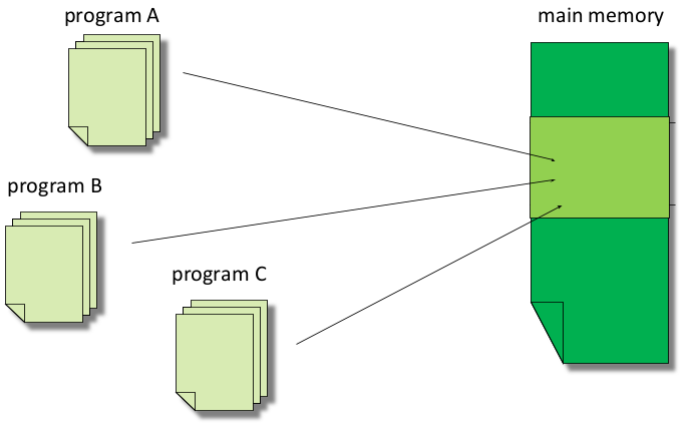
Per tanto, quando una linea non avrà più posti disponibili, sarà necessario **rimpiazzare** il dato contenuto in uno dei suoi set. Solitamente, viene rimpiazzato quello utilizzato meno recentemente (**Least Recently Used Policy - LRU**) o quello utilizzato di meno in generale (**Least Frequently Used Policy - LFU**).

Inoltre, quando vengono scritti dati su una cache, tali dati potrebbero essere **inconsistenti** con quelli della memoria principale. Per gestire tale dinamica, le due strategie più comuni sono:

- **Write-through**: il dato viene immediatamente scritto sulla memoria principale
- **Write-back**: il dato viene marcato come **dirty** tramite un bit speciale. Quando il set della cache contenente tale dato verrà rimpiazzato, il dato verrà prima scritto sulla memoria

Eseguendo un programma di grandi dimensioni o che accede a grandi insiemi di dati, tutte le istruzioni e i dati potrebbero non entrare all'interno della memoria principale.

Con **memoria virtuale** intendiamo una modalità di gestione della memoria basata sulla tecnica del **paging**, ossia la suddivisione della memoria principale in **frame** e della memoria di ogni programma in **pagine**, dove frame e pagine sono blocchi di byte della stessa dimensione. Le pagine di un programma possono essere **inattive**, ossia conservate sullo **swap space** della memoria secondaria, o **attive**, ossia attualmente caricate in memoria ed associate ad un frame. In particolare, ogni frame può conservare una sola pagina per volta.



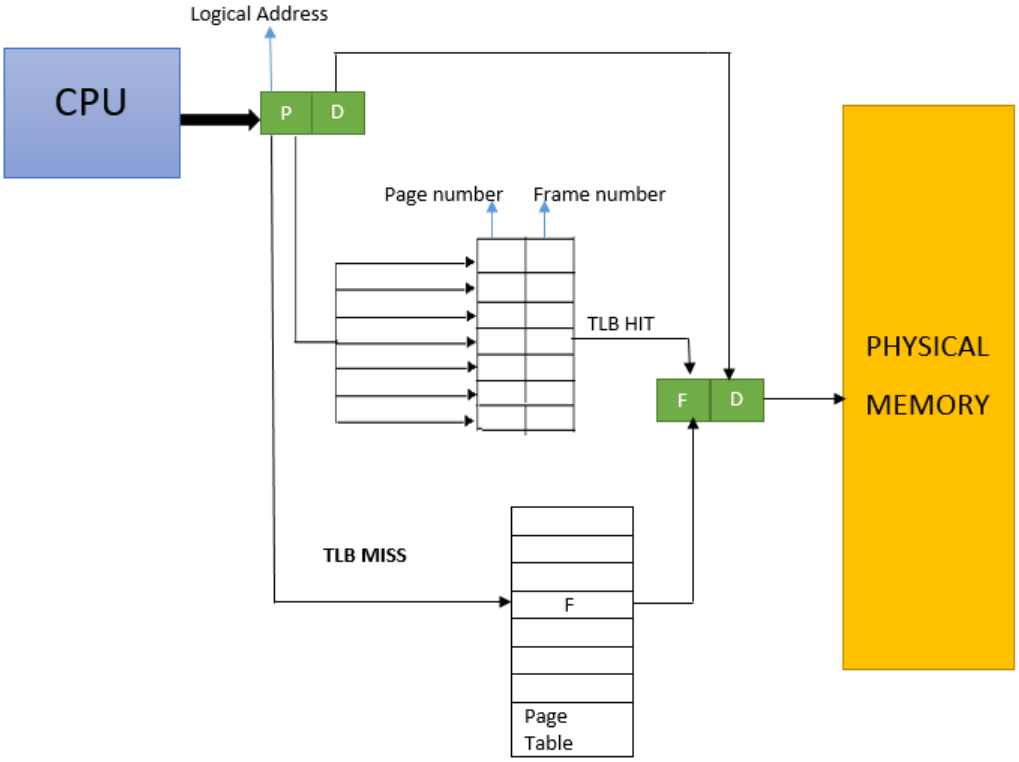
Quando un programma viene compilato, alle sue pagine vengono assegnati dei **virtual page number**, tramite cui vengono generati i **virtual address** di istruzioni e dati. Una volta che tale programma verrà eseguito verrà creata una sua **page table** all'interno della quale vengono conservate le **traduzioni** tra indirizzi virtuali e fisici. Nel caso i cui si tenti di accedere all'indirizzo fisico di una pagina attualmente non presente sulla memoria principale (**page fault**), essa verrà prelevata dal disco e associata ad un frame libero (rimpiazzandone uno se necessario).

Nonostante l'uso della memoria virtuale migliori notevolmente le prestazioni, le page table vengono comunque conservate all'interno della memoria principale, rendendo lento il processo di traduzione. Pertanto, viene utilizzata una **Translation Lookaside Buffer (TLB)**, ossia una cache speciale addetta solo al salvataggio di poche entrate delle varie page table.

Per gestire i **TLB miss**, vengono utilizzate due strategie principali:

- **Hardware managed**: a seguito del miss, la CPU scorre automaticamente la page table cercando la pagina richiesta
- **Software manager**: a seguito del miss, viene generata una TLB miss exception e il sistema operativo si occupa di scorrere la page table non appena possibile

Se la pagina richiesta non viene trovata neanche in page table, essa verrà prelevata dallo swap space per poi venire salvata in TLB, generando anche una page fault exception.



1.1.2 Parallelismo a livello hardware

Definizione 5: Instruction level parallelism (ILP)

L'**instruction level parallelism (ILP)** prevede l'utilizzo di CPU composte da più unità funzionali in grado di eseguire simultaneamente istruzioni diverse, permettendo di ottenere parallelismo all'interno delle CPU stesse.

La prima modalità di implementazione dell'ILP prevede l'uso del **pipelining** (*catena di montaggio*), ossia la suddivisione dell'esecuzione di una singola istruzione in più **fasi**, dove ogni fase è eseguita da un componente a se stante, permettendo quindi l'esecuzione simultanea di più istruzioni.

Esempio:

- Supponiamo di voler eseguire il seguente frammento di codice:

```
...
float x[1000], y[1000], z[1000];
...

for(i = 0; i < 1000; ++i){
    z[i] = x[i] + y[i];
}
```

- La somma tra due numeri float è suddivisibile nelle seguenti 7 fasi. Ad esempio, la somma dei numeri $9.87 \cdot 10^4$ e $6.54 \cdot 10^3$ viene svolta nel seguente modo:

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

- Assumendo che ognuna delle 7 fasi impieghi 1 ns, il tempo totale impiegato dal loop corrisponderebbe a 7000 ns
- Suddividendo il floating point adder (ossia l'adder della CPU addetto a tale compito) in 7 unità funzionali a cui vengono associate le 7 fasi della somma, tali somme verrebbero svolte simultaneamente:

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

  richiedendo quindi solo 1005 ns per eseguire l'intero loop

La seconda modalità di implementazione, invece, prevede l'uso del **multiple issue**, ossia l'esecuzione sincrona di istruzioni dello stesso tipo su copie diverse della stessa unità funzionale (es: con due floating point adder è possibile eseguire due somme float contemporaneamente). Per utilizzare tale modalità è richiesto uno **scheduling** delle unità funzionali da far utilizzare alle istruzioni. In particolare, lo **static multiple issue** prevede che le unità funzionali vengano schedulate al tempo di compilazione, mentre il **dynamic multiple issue** prevede che ciò avvenga in fase di esecuzione stessa.

Per poter schedulare correttamente le istruzioni, il compilatore (o la CPU nel caso del dynamic multiple issue) deve essere in grado di **speculare** su quali istruzioni possano essere avviate simultaneamente. In caso di speculazione errata, l'esecuzione viene terminata e rieseguita correttamente.

Per quanto riguarda il multithreading, invece, alcune volte risulta impossibile eseguire simultaneamente thread diversi. Per tale motivo, viene implementato del **multithreading a livello hardware** in grado di permettere ad un sistema di continuare a svolgere lavoro quando una task attualmente in esecuzione rimane in attesa (**stalled**):

- **Fine-grained hardware multithreading** (*a grana fine*): il processore alterna i thread ad ogni istruzione, saltando quelli in attesa. In tal modo viene generalmente ridotto il tempo sprecato a seguito degli stalli, ma un thread che deve eseguire molte istruzioni richiederà più tempo per terminare il lavoro
- **Coarse-grained hardware multithreading** (*a grana grossa*): il processore alterna solo i thread che sono in attesa, permettendo agli altri di continuare l'esecuzione. In tal modo viene generalmente ridotta la quantità di switch tra i thread, ma il processore potrebbe rimanere inattivo durante stalli brevi per via del continuo switch tra thread inattivi
- **Simultaneous hardware multithreading**: variante del fine-grained dove viene permesso ai thread di utilizzare molteplici unità funzionali

1.1.3 Tassonomia di Flynn

La **tassonomia di Flynn** è un sistema di classificazione delle architetture degli elaboratori a seconda della **molteplicità** del flusso di **istruzioni** e del flusso dei **dati** che possono gestire:

- **Single Instruction stream with Single Data stream (SISD)**: vi è un solo flusso di istruzioni che opera su un solo flusso di dati. Esempio tipico di tale architettura è il modello di Von Neumann.
- **Single Instruction stream with Multiple Data stream (SIMD)**: vi è un solo flusso di istruzioni che opera su più flussi di dati in parallelo. Tale architettura è molto utilizzata nei sistemi moderni richiedenti la manipolazione di grandi quantità di dati.
- **Multiple Instruction stream with Single Data stream (MISD)**: vi sono molteplici flussi di istruzioni che operano in parallelo sullo stesso flusso di dati. Tale architettura viene utilizzata molto di rado in quanto estremamente situazionale.
- **Multiple Instruction stream with Multiple Data stream (MIMD)**: vi sono molteplici flussi di istruzioni che operano in parallelo sullo molteplici flussi di dati. I moderni processori multicore rientrano in tale categoria.

1.2 Architetture SIMD

All'interno delle architetture SIMD, il **parallelismo** viene ottenuto suddividendo i flussi di dati tra i vari processori, dove ognuno di essi applicherà la stessa istruzione (**data parallelism**). Visualmente, è possibile immaginare l'esecuzione di istruzioni sulle architetture SIMD come un macchinario industriale di una fabbrica: lo stesso compito viene eseguito su n oggetti contemporaneamente.

Esempio:

- Consideriamo il seguente codice

```
...
for(i = 0; i < n; ++i){
    x[i] += y[i];
}
...
```

- Supponendo di avere m core, abbiamo che:
  - Se $m \geq n$ allora ogni somma di tale loop viene assegnata ad un core, per poi venir eseguite tutte parallelamente
  - Se $m < n$ allora i dati su cui svolgere le somme vengono assegnate iterativamente, eseguendo quindi m somme parallele per poi ripetere la fase di assegnamento iterativo

Nonostante la loro semplicità, le architetture SIMD presentano alcuni **svantaggi**:

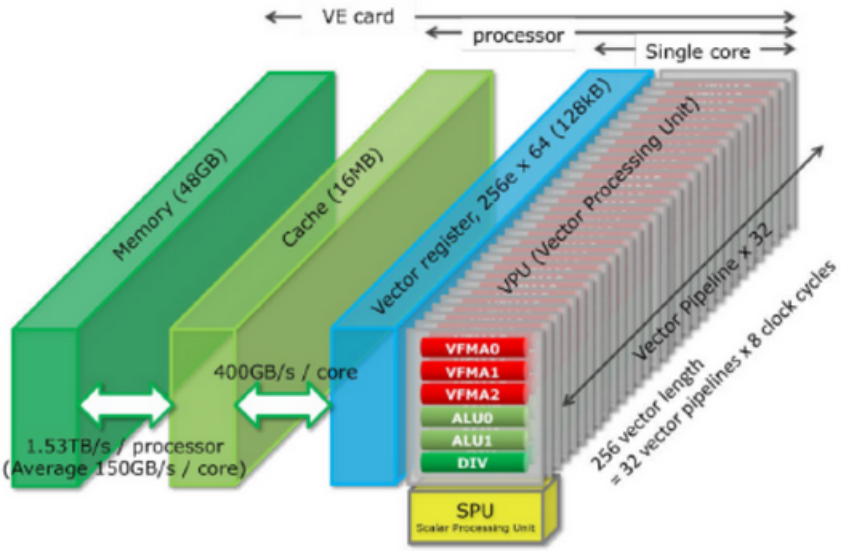
- Poiché i core devono eseguire la stessa istruzione degli altri core, ogni ALU deve scegliere se eseguire l'istruzione attuale o attendere la prossima
- Nei design classici è richiesto che le ALU siano sincronizzate tra loro
- Molto efficiente solo per programmi paralleli con molti dati ed istruzioni semplici

Esempi tipici di sistemi basati su architettura SIMD sono i sistemi dotati di **processori vettoriali**, i quali operano su **array** (detti *vettori*) **di dati** invece che su singoli elementi (detti *scalari*). Sono dotati di:

- **Registri vettoriali** capaci di contenere vettori di scalari ed operare simultaneamente su di essi
- **Unità funzionali vettoriali** capaci di applicare la stessa operazione ad ogni scalare dei vettori su cui si sta lavorando
- **Memoria interlacciata** composta da molteplici banchi di memoria, i quali possono essere acceduti più o meno indipendentemente. Gli elementi dei vettori vengono distribuiti tra i banchi, riducendo o persino eliminando i ritardi dovuti a load/store di elementi successivi
- **Accesso progressivo alla memoria (hardware scatter/gather)**, ossia la possibilità di accedere agli elementi di un vettore solo ad intervalli fissi

Grazie ai **compilatori vettoriali**, i quali ottimizzano il codice e forniscono informazioni sulle parti di codice non vettorializzabili, l'uso dei processori vettoriali risulta veloce e semplice. Inoltre, la loro natura stessa favorisce notevolmente l'uso delle **cache**, in quanto i vettori sono contigui.

Tuttavia, quest'ultimi non sono in grado di gestire **strutture dati irregolari**, ossia non contigue (es: liste, alberi, ...), e possiedono **scarsa scalabilità** per problemi di dimensioni più grosse.



Schematizzazione del processore vettoriale Aurora NEC

Basate sull'architettura SIMD sono anche molte **moderne GPU**. Difatti, rispetto quest'ultime sono improntate **Throughput Oriente Design**:

- Molteplici piccoli core e dotati di piccole cache
- Assenza di branch prediction e di data forwarding
- Una grande quantità di ALU ma energeticamente efficienti (dunque più deboli) e richiedenti molteplici e lunghe latenze, compensate tramite un uso estensivo del pipelining
- Richiedono un enorme numero di thread per gestire le latenze

Le **normali CPU** (dunque del modello SISD), invece, sono improntate ad un **Latency Oriented Design**:

- Una sola CPU dotata di cache di grandi dimensioni
- Presenza di branch prediction e data forwarding
- ALU potenti e richiedenti meno latenze