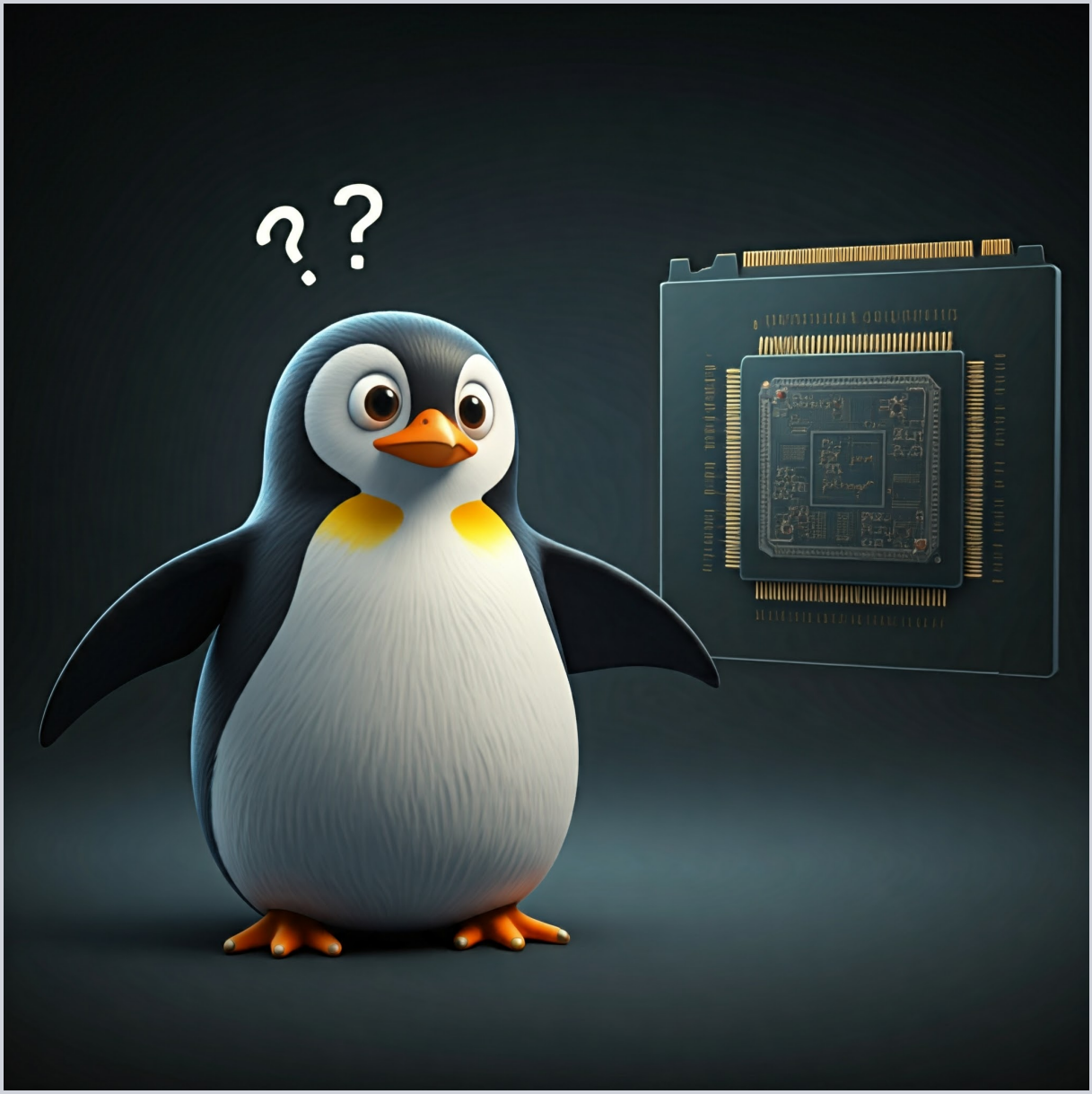
Marco Casu

Programmazione di Sistemi Multicore



Sapienza
Università di Roma

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Dipartimento di Informatica

Questo documento è distribuito sotto la licenza GNU, è un resoconto degli appunti (eventualmente integrati con libri di testo) tratti dalle lezioni del corso di Programmazione di Sistemi Multicore per la laurea triennale in Informatica. Se dovessi notare errori, ti prego di segnalarmeli.

INDICE

CAPITOLO

1

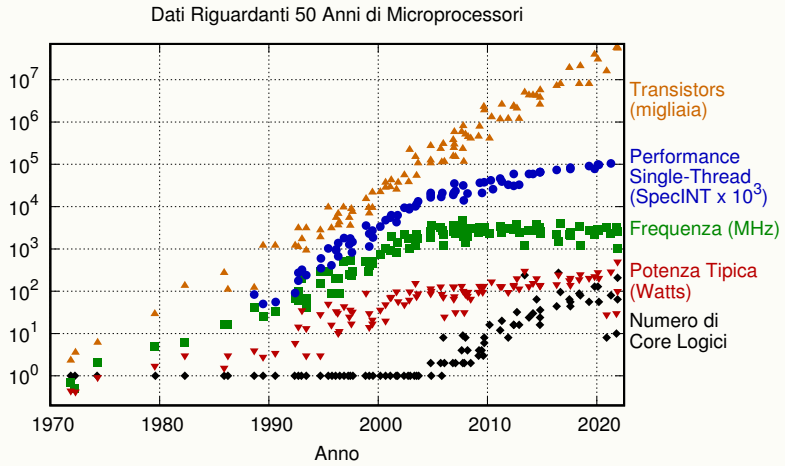
PARALLELISMO : MOTIVAZIONI

1.1 Introduzione

In una *GPU* (Graphics Processing Unit), nota anche come scheda video, ci sono circa 80 miliardi di transistor, e vengono utilizzate per allenare i grossi modelli di intelligenza artificiale, i quali necessitano migliaia di GPU, non è un caso se *Nvidia* ad oggi, con il boom dell'IA, è una delle aziende più quotate al mondo. Le GPU, e la loro programmazione, sono uno fra i principali argomenti di questo corso.

L'evoluzione dell'hardware, ha portato i grossi sistemi di computazione, ad essere formati da svariate unità di calcolo piuttosto che una singola unità molto potente, i processori stessi di uso comune, ad oggi sono composti da più *core*.

La legge di Moore riguarda una stima empirica che mette in correlazione lo scorrere del tempo con l'aumentare della potenza di calcolo dei processori, se inizialmente, a partire dagli anni 70, tale potenza raddoppiava ogni due anni, ad oggi tale andamento è andato rallentando, raggiungendo un incremento 1.5 in 10 anni.



L'obbiettivo di costruire calcolatori sempre più potenti è dipeso dalla necessità dell'Uomo di risolvere problemi sempre più complessi, come ad esempio, la risoluzione del genoma umano.

Il motivo per il quale non è possibile costruire processori monolitici sempre più potenti, risiede in un *limite fisico* riguardante la densità massima possibile dei transistor in un chip.

1. transistor più piccoli $\longrightarrow$ processori più veloci
2. processori più veloci $\longrightarrow$ aumento del consumo energetico
3. aumento del consumo energetico $\longrightarrow$ aumento del calore
4. aumento del calore $\longrightarrow$ problemi di inaffidabilità dei transistor

1.2 Modelli di Parallelismo

L'informatico che intende scrivere del codice per un sistema multicore, deve esplicitamente sfruttare i diversi core, limitandosi a scrivere un codice sequenziale, non starebbe sfruttando a pieno l'hardware a disposizione, rendendo il processo meno efficiente di quanto potrebbe essere.

La maggior parte delle volte, un algoritmo sequenziale, non può essere direttamente tradotto in un algoritmo parallelo, per questo bisogna scrivere il codice facendo riferimento all'hardware di destinazione. Si consideri adesso il seguente codice sequenziale, che ha lo scopo di sommare n numeri dati in input.

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

Si vuole rendere tale algoritmo parallelo, sapendo di essere a disposizione di p core.

```
local_sum = 0;
first_index = ...;
last_index = ...;
for(local_i=first_index; first_index<last_index; local_i++){
    local_x = compute_next_value(...);
    local_sum += local_x;
}
```

In tale esempio, ogni core possiede le sue variabili private non condivise con gli altri core, ed esegue indipendentemente il blocco di codice. Ogni core conterrà la somma parziale di n/p valori.

Esempio (24 numeri, 8 core) :

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

core	0	1	2	3	4	5	6	7
`local_sum`	8	19	7	15	7	13	12	14

A questo punto, per ottenere la somma totale, vi sarà un core *master* che riceverà le somme parziali da tutti gli altri core, per poi eseguire la somma finale.

```
if(master){
    sum = local_sum;
    for c : core{
        if(c!=self){
            sum += c.local_sum;
        }
    }
}else{
    send local_sum to master;
}
```

Dividere i dati per poi far eseguire la stessa computazione ai diversi nodi è la forma più semplice di parallelismo. La soluzione adottata non è ideale, in quanto, in seguito al calcolo delle somme parziali, tutti i core escluso il master non staranno eseguendo calcoli. Una possibile idea alternativa è di far si che a coppie i nodi si condividano le somme parziali per poi calcolarne una somma comune, sviluppando uno scambio di dati ad albero, come mostrato in figura 1.1.

Possiamo identificare due tipi di parallelismo :

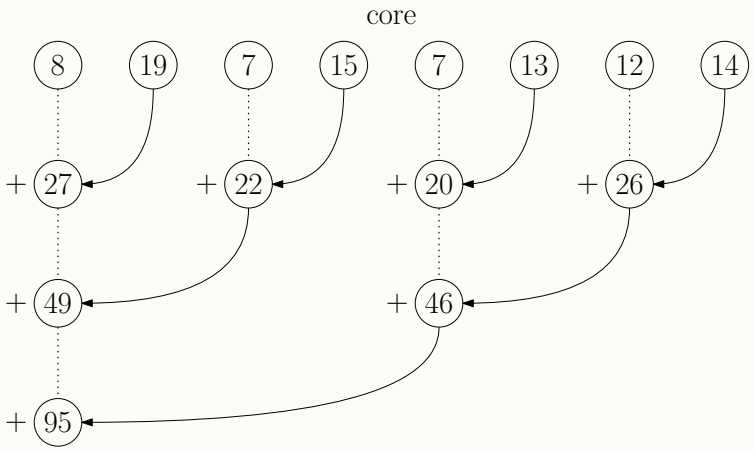


Figura 1.1: calcolo somme a coppie

- **parallelismo dei task** : fra i core vengono divise diverse attività che vengono svolte autonomamente.
- **parallelismo dei dati** : i dati da elaborare vengono divisi, ogni core eseguirà la stessa computazione ma su una porzione diversa dei dati.

Quando si scrive un programma parallelo bisogna prestare attenzione alla *sincronizzazione* dei processi, in quanto potrebbero dover accedere ad una stessa area di memoria. Risulta cruciale saper mettere in *comunicazione* i vari core, e suddividere equamente il *carico di lavoro* fra di essi. Verranno considerate 4 diverse tecnologie per la programmazione multicore :

- *MPI* (Message Passing Interface) [libreria]
- *Posix* Threads [libreria]
- *OpenMP* [libreria e compilatore]
- *CUDA* [libreria e compilatore]

La programmazione delle GPU richiederà un diverso compilatore, e non il solito `gcc`, in quanto l'architettura della scheda video differisce da quella del processore, e con essa le istruzioni.

I sistemi paralleli possono essere categorizzati sotto vari aspetti.

- **shared memory** : Tutti i core accedono ad un'area di memoria comune. L'accesso e la sincronizzazione vanno gestiti con cautela.
- **distributed memory** : Ogni core ha un area di memoria privata, e la comunicazione avviene attraverso un apposito canale per lo scambio dei messaggi.

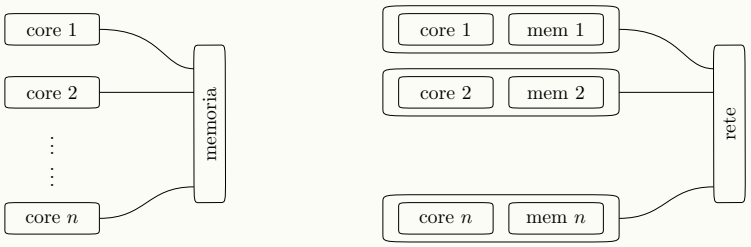


Figura 1.2: modelli di parallelismo

Vi è un altra suddivisione nei sistemi paralleli :

- **MIMD** : Ogni core ha una control unit indipendente, diversi core possono eseguire diverse istruzioni nello stesso momento.

- **SIMD** : Vi è un singolo program counter per tutti i core, che eseguono in maniera parallela le stesse istruzioni. Due core non possono eseguire operazioni diverse nello stesso momento.

Le GPU hanno una struttura *SIMD*.

	shared memory	distributed memory
SIMD	CUDA	
MIMD	Pthreads/OpenMP/CUDA	MPI

Fin'ora sono stati utilizzati 3 termini chiave riguardante i tipi di programmazione, sebbene non vi sia una definizione comunemente accettata, la seguente verrà adottata in tale contesto :

- *concorrente* : più processi sono attivi in uno stesso momento
- *parallela* : diverse entità cooperative che operano in maniera ravvicinata per un obbiettivo comune.
- *distribuita* : diverse entità cooperative.

La programmazione parallela o distribuita implica che sia anche concorrente, non è vero il contrario.