



**SAPIENZA**  
UNIVERSITÀ DI ROMA

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

# **Programmazione di Sistemi Embedded e Multicore**

**Autore:**  
Simone Lidonnici

24 settembre 2024

# Indice

<b>1</b>	<b>Programmazione parallela</b>	<b>1</b>
1.1	Perchè usare la programmazione parallela . . . . .	1
1.2	Scrivere programmi paralleli . . . . .	1
1.2.1	Tipi di parallelismo . . . . .	2
1.2.2	Come scrivere programmi paralleli . . . . .	2
1.3	Tipi di sistemi paralleli . . . . .	3
1.3.1	Concorrenti vs Paralleli vs Distribuiti . . . . .	3
1.4	Architettura di Von Neumann . . . . .	4

# 1

## Programmazione parallela

### 1.1 Perchè usare la programmazione parallela

Dal 1986 al 2003 le performance dei microprocessori aumentavano di circa il 50% l'anno, dal 2003 questo aumento è diminuito fino ad arrivare al 4% l'anno. Questa diminuzione è causata dal fatto che l'aumento delle performance dipende dalla densità dei transistor, che diminuendo di grandezza generano più calore e questo li fa diventare inaffidabili. Per questo conviene avere più processori nello stesso circuito rispetto ad averne uno singolo più potente.

### 1.2 Scrivere programmi paralleli

Per utilizzare al meglio i diversi processori, bisogna volutamente scrivere il programma in modo da usare il parallelismo, in alcuni casi è possibile convertire un programma sequenziale in uno parallelo ma solitamente bisogna scrivere un nuovo algoritmo.

#### Esempio:

Computare  $n$  valori e sommarli tra loro.

#### Soluzione sequenziale:

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

#### Soluzione parallela:

Avendo  $p$  core ognuno eseguirà  $\frac{n}{p}$  somme

```
sum=0;
start=...; #definiti in base al core
end=...;   #che esegue il programma
for(i=start;i<end;i++){
    x=ComputeValue(...);
    sum+=x;
}
```

Dopo che ogni core avrà finito la sua somma, per ottenere la somma totale si può designare un core come **master core**, a cui tutti gli altri core invieranno la propria somma e che eseguirà la somma totale.

Questa soluzione però non è ottimale perchè il master core esegue una ricezione e una somma per ogni altro core ( $p - 1$  volte), mentre gli altri sono fermi. Per migliorare l'efficienza si sommano a coppie i risultati di ogni core:

- il core 0 somma il risultato con il core 1, il core 2 con il core 3 ecc...
- si ripete con solo i core 0, 2 ecc...
- si continua creando uno schema ad albero binario

Questo secondo metodo è molto più efficiente perchè il numero di ricezioni e somme che esegue il core che ottiene il risultato finale sono  $\log_2(p)$ .

### 1.2.1 Tipi di parallelismo

Ci sono due principali tipi di parallelismo:

- **Task parallelism:** Diversi task vengono divisi tra i vari core e ogni core esegue operazioni diverse su tutti i dati (il parallelismo temporale nei circuiti è un tipo di task parallelism)
- **Data parallelism:** Vengono divisi i dati tra i core e ogni core esegue operazioni simili su porzioni di dati diverse (il parallelismo spaziale nei circuiti è un tipo di task parallelism)

#### Esempio:

Bisogna correggere 300 esami ognuno con 15 domande e ci sono 3 professori disponibili.

In questo caso:

- Data parallelism:
  - Ogni assistente corregge 100 esami
- Task parallelism:
  - Il primo professore corregge le domande 1-5 di tutti gli esami
  - Il secondo professore corregge le domande 6-10 di tutti gli esami
  - Il terzo professore corregge le domande 11-15 di tutti gli esami

### 1.2.2 Come scrivere programmi paralleli

Se i core possono lavorare in modo indipendente scrivere un programma parallelo è molto simile a scriverne uno sequenziale, in pratica però i core devono comunicare, per diversi motivi:

- **Comunicazione:** un core deve inviare dei dati ad un altro
- **Bilanciamento del lavoro:** bisogna dividere il lavoro in modo equo per far sì che un core non sia sovraccaricato rispetto agli altri, sennò tutti i core aspetterebbero il più lento
- **Sincronizzazione:** ogni core lavora a velocità diversa e bisogna controllare che uno non vada troppo avanti rispetto agli altri

Per creare programmi paralleli useremo 4 diverse estensioni di C:

1. **Message-Passing Interface (MPI):** Libreria
2. **Posix Threads (Pthreads):** Libreria
3. **OpenMP:** Libreria e Compilatore
4. **CUDA:** Libreria e Compilatore

## 1.3 Tipi di sistemi paralleli

I sistemi paralleli possono essere divisi in base alla divisione della memoria:

- **Memoria Condivisa:** Tutti i core hanno accesso alla memoria del computer e si coordinano modificando locazioni di memoria condivisa
- **Memoria Distribuita:** Ogni core possiede una propria memoria e per coordinarsi devono mandarsi dei messaggi

Possono essere anche divisi in base al numero di **Control Unit**:

- **Multiple-Instruction Multiple-Data (MIMD):** Ogni core ha la sua CU e può lavorare indipendentemente dagli altri
- **Single-Instruction Multiple-Data (SIMD):** I core condividono una sola CU e devono eseguire tutti le stesse operazioni o stare fermi

Le librerie nominate precedentemente si collocano in una tabella:

	SIMD	MIMD
Memoria Condivisa	CUDA	Pthreads OpenMP CUDA
Memoria Distribuita		MPI

### 1.3.1 Concorrenti vs Paralleli vs Distribuiti

I sistemi possono essere:

- **Concorrenti:** più task possono essere eseguite in ogni momento, possono essere anche sequenziali
- **Paralleli:** più task cooperano per risolvere un problema comune, i core sono condividono la memoria o sono connessi tramite un network veloce
- **Distribuiti:** un programma che coopera con altri programmi per risolvere un problema comune, i core sono connessi in modo più lento

I sistemi paralleli e distribuiti sono anch'essi concorrenti.

## 1.4 Architettura di Von Neumann

Per poter scrivere codice efficiente bisogna conoscere l'architettura su cui si sta eseguendo il codice ed ottimizzarlo per essa.

L'architettura di Von Neumann è composta da:

- **Memoria principale:** Insieme di locazioni, ognuna con un indirizzo e del contenuto (dati o istruzioni)
- **CPU/Processore/Core:** Control Unit, che decide le istruzioni da eseguire, e **datapath**, che eseguono le istruzioni. Lo stato di un programma in esecuzione viene salvato nei registri, un registro molto importante è il **Program Counter (PC)** dove viene salvato l'indirizzo della prossima istruzione da eseguire
- **Interconnessioni:** Usate per trasferire dati tra CPU e memoria, tradizionalmente con un bus

Una macchina di Von Neumann esegue un'istruzione alla volta, operando su piccole porzioni di dati contenuti nei registri. La CPU può leggere (fetch) dati dalla memoria o scriverci (store), la separazione tra CPU e memoria è conosciuta come **Bottleneck di Von Neumann**, cioè le interconnessioni determinano la velocità con cui i dati vengono trasferiti.