Programmazione Multicore

Alessio Marini, 2122855

Appunti presi durante il corso di **Programmazione Multicore** nell'anno **2025/2026** del professore Daniele De sensi.
Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏 .

Contatti:
alem1105
marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1. Parallel Computing

Dal 1986 al 2003 le velocità dei microprocessori aumentava di un 50% all'anno, ovvero un 60x in 10 anni. Dal 2003 in poi però questo incremento ha iniziato a rallentare, ad esempio dal 2015 al 2017 c'è stato soltanto un 4% all'anno ovvero un 1.5x in 10 anni.

Ci sono motivi fisici dietro a questo fenomeno e per questo, invece di continuare a costruire processori più potenti abbiamo iniziato ad inserire più processori all'interno dello stesso circuito.

I programmi seriali ovviamente non sfruttano questi benefici e continuano a venir eseguiti in un singolo processore, anche se ovviamente significa che possiamo eseguirli in più istanze contemporaneamente. Sono quindi i programmatori che devono sapere come utilizzare queste tecnologie per scrivere programmi che le sfruttano.

Ci serve tutta questa efficienza?

Per la maggior parte dei programmi no ma esistono dei campi di ricerca che dove c'è bisogno di eseguire tantissime operazioni in poco tempo o comunque su tantissimi dati, ad esempio:

- LLMs
- Decoding the human genome
- Climate modeling
- Protein folding
- Drug discovery
- Energy research
- Fundamental physics

Il motivo fisico che abbiamo accennato prima del perché costruiamo questi sistemi paralleli è dovuto al fatto che le performance di un processore aumentano con l'aumentare della densità di transistor che ha, questo comporta alcune cose:

- Transistor più piccoli -> Processori più veloci
- Processori più veloci -> Aumenta il loro consumo energetico
- Consumo più alto -> Aumenta la temperatura del processore
- Temperatura alta -> Comportamenti del processore inaspettati

Quindi anche se alcuni programmi possiamo eseguirli in più istanze e aumentare la loro efficienza spesso non è la strada migliore e dobbiamo imparare a scrivere del codice che usa la parallelizzazione. Con il temo sono stati scoperti dei «pattern» facilmente convertibili in codice parallelo ma spesso la strada migliore è quella di fare un passo indietro e ripensare un nuovo algoritmo. Non sempre saremo in grado di parallelizzare completamente il codice.

Esempio

Codice Seriale - *Compute n values and add them together*

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

Codice Parallelo - abbiamo p cores dove $p \ll n$ e ogni core calcola la somma di $\frac{n}{p}$ valori

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

Quindi ogni core usa delle variabili per memorizzare il suo primo e ultimo valore da sommare, in questo modo ogni core può eseguire del codice indipendentemente dagli altri core.

Ad esempio se abbiamo 8 core e 24 valori, ogni core somma 3 valori:
- 1, 4, 3 - 9, 2, 8 - 5, 1, 1 - 6, 2, 7 - 2, 5, 0 - 4, 1, 8 - 6, 5, 1 - 2, 3, 9

E quindi avremo come somme:
- 8, 19, 7, 15, 7, 13, 12, 14

Ci basta quindi sommare la somma di tutti i core e ottenere il risultato finale.

Però usando questa soluzione, nello step finale abbiamo che un core solo (il principale) effettua la somma dei risultati degli altri, mentre loro appunto non stanno facendo nulla.

Per risolvere questo possiamo ad esempio accoppiare il core 0 con il core 1 e fare in modo che il core 0 sommi al suo risultato quello del core 1, poi possiamo fare lo stesso lavoro con il 2 e il 3 ed avere quindi nel core 2 la somma di 2 e 3 ecc...

Ripetiamo tutto questo accoppiando 0-2, 4-6... e poi continuiamo a ripetere accoppiando 0-4...

Vediamo graficamente:

TODO: image

Se confrontiamo i due metodi:

TODO: image

Notiamo che con il primo metodo, quello a sinistra, se abbiamo 8 cores facciamo 7 somme aggiuntive, in generale $p - 1$ somme.

Con il secondo metodo se abbiamo 8 cores facciamo 3 somme aggiuntive (sono sempre 7 ma sono parallele e avvengono nello stesso momento), in generale abbiamo $\log_2(p)$ somme.

Quindi se ad esempio avessimo avuto $p = 1000$ con il primo metodo avremmo avuto 999 somme mentre con il secondo soltanto 10.

Come scriviamo codice parallelo?

- **Task Parallelism**: Dividere alcune task fra i cores. L'idea è quella di eseguire compiti diversi in parallelo.
- **Data Parallelism**: Partizionare i dati fra i cores, fargli risolvere operazioni simili e risolvere il problema raccogliendo i dati. In generale quando i cores svolgono la stessa operazione ma su pezzi di dati diversi.

Esempio

Devo valutare 300 esami da 15 domande ciascuno e ho 3 assistenti:

- Data Parallelism: Ogni assistente valuta 100 esami
- Task Parallelism:
- L'assistente 1 valuta tutti gli esami ma soltanto le domande 1-5
- L'assistente 2 valuta tutti gli esami ma soltanto le domande 6-10
- L'assistente 3 valuta tutti gli esami ma soltanto le domande 11-15

L'esempio che abbiamo fatto precedentemente con le somme dei vari cores, è stato parallelizzato on Data o Task Parallelism?

TODO: image

Se ogni core può lavorare in modo indipendente dagli altri allora la scrittura del codice sarà molto simile a quella di un programma seriale. In generale dobbiamo coordinare i cores, questo perché:

- Communication: Ad esempio perché ogni core manda una somma parziale ad un altro
- Load Balancing: Nessun core deve svolgere troppo lavoro in più rispetto ad altri perché altrimenti qualche core dovrà aspettare che alcuni finiscano e questo significa perdita di risorse e potenza.
- Synchronization: Ogni core lavora al suo ritmo ma dobbiamo assicurarci che nessuno vada troppo avanti. Ad esempio se un core compila una lista dei file da comprimere ed i cores che comprimono partono troppo presto potrebbero perdersi qualche files.

Noi scriveremo codice esplicitamente parallelo usando 4 diverse estensioni delle API di C:

- Message-Passing Interface (MPI) [Library]
- Posix Threads (Pthreads) [Library]
- OpenMP [Library + Compiler]
- CUDA [Library + Compiler]

Useremo anche librerie ad alto livello già esistenti che però hanno un compresso per quanto riguarda facilità di utilizzo e performance.

1.1. Type of Parallel Systems

- **Shared Memory**: I core lavorano sulla stessa memoria e vanno coordinati.

- **Distributed Memory**: Ogni core ha la sua memoria dedicata e si scambiano messaggi fra loro per coordinarsi.

In generale quindi devono comunque avere un modo per coordinarsi

- **Multiple-Instruction Multiple-Data (MIMD)**: Ogni core ha la sua unità di controllo e può lavorare indipendentemente dagli altri. Come ad esempio la CPU classica dei PC, ogni core può fare qualcosa di diverso.
- **Single-Instruction Multiple-Data (SIMD)**: Ogni core può lavorare su un pezzo di dato diverso, ma tutti devono lavorare per la stessa istruzione. Ad esempio possono lavorare tutti su un vettore ma ognuno su una parte di vettore diversa.

Concurrent

Parallel

Distributed

- **Von Neumann architecture**