Corso di Programmazione Concorrente

Programmazione Concorrente: Introduzione

Valter Crescenzi
crescenz@dia.uniroma3.it
http://crescenzi.inf.uniroma3.it

Sommario

- Concetti fondamentali e terminologia
  - Programma
  - Flusso di esecuzione
  - Risorsa
  - Programmazione
    - Sequenziale vs Concorrente
  - Esecuzioni sequenziali e concorrenti
  - Diagramma delle Precedenze
  - Sequenze di esecuzioni ammissibili
  - Sequenze di interleaving
- Legge di Amdahl

Programmi e Flussi di Esecuzione

- *Programma*: descrizione statica, ovvero che non cambia nel tempo, di un flusso di esecuzione
- *Flusso di Esecuzione*: concetto dinamico, ovvero che evolve nel tempo
  - un flusso di esecuzione scaturisce con l'esecuzione di un programma da parte di un *esecutore*
  - ad un medesimo programma possono corrispondere molteplici flussi di esecuzione

- Un flusso di esecuzione per poter avanzare necessita di *risorse*, prima tra tutte, l'esecutore

Risorse

- Servono per l'avanzamento dei flussi di esecuzione
- L'*esecutore* è una particolare tipologia di queste risorse, indispensabile, come la *memoria*, per l'esecuzione, ma ne esistono di molti altri tipi
- Molte risorse *software* sono ottenute virtualizzando risorse *hardware* sottostanti
  - Esecutore fisico → esecutore virtuale
  - Memoria fisica → memoria virtuale

  sono sostanzialmente *servizi* offerti dalla piattaforma sottostante

Prerilasciabilità delle Risorse

- Alcune proprietà delle risorse sono di particolare interesse per la programmazione concorrente...
- Risorse *Prerilasciabili*
  - si possono sottrarre al flusso di esecuzione che le sta usando senza causare il fallimento dell'esecuzione in atto
  - risorsa non seriale: molteplicità della risorsa > 1
    - es. memoria in lettura
- Risorse *Non Prerilasciabili*
  - se sottratte al flusso di esecuzione che le sta usando, l'esecuzione fallisce
  - risorse seriali; molteplicità = 1
    - es. stampante, masterizzatore
- *Molteplicità* di una risorsa: numero massimo di flussi di esecuzione che la possono usare concorrentemente senza comprometterne l'utilizzo

Risorse di Molteplicità Finita

- La *virtualizzazione* delle fondamentali risorse fisiche seriali (memoria, esecutori) si basa su meccanismi per ottenere la loro pre-rilasciabilità (*context-switching*)
  - Rende disponibili risorse virtuali di pari molteplicità
  - Ma disponibili in numero arbitrario e quindi allocabili a diversi *f.d.e.*

- Per disciplinare gli accessi alle risorse di molteplicità finita, è necessario prevedere:

  - un *Gestore della Risorsa*

  - un *protocollo* di accesso alla Risorsa
  - richiesta ed ottenimento della risorsa
  - utilizzo della risorsa
  - rilascio della risorsa

Programmazione Sequenziale

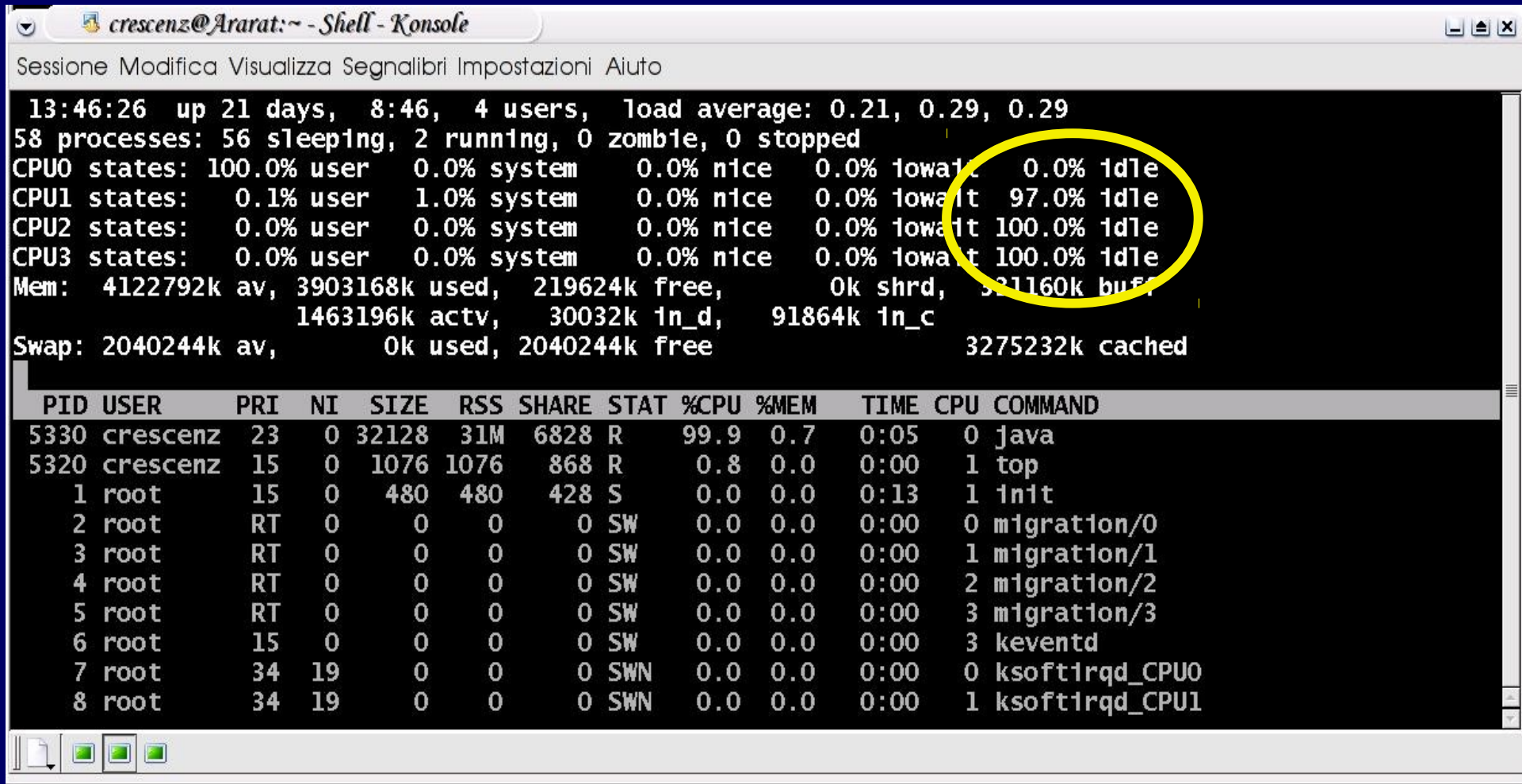
- La programmazione è di solito insegnata con riferimento ad un esecutore *sequenziale*
- Un esecutore sequenziale svolge una sola azione alla volta sulla base di un *programma sequenziale*
- L'esecuzione di un programma sequenziale origina un *flusso di esecuzione sequenziale* che conferisce un ordinamento totale alle azioni eseguite

- La programmazione di un esecutore *concorrente*, ovvero in grado di eseguire più istruzioni contemporaneamente, sebbene più difficile di quella tradizionale, ha forti motivazioni didattiche e pratiche

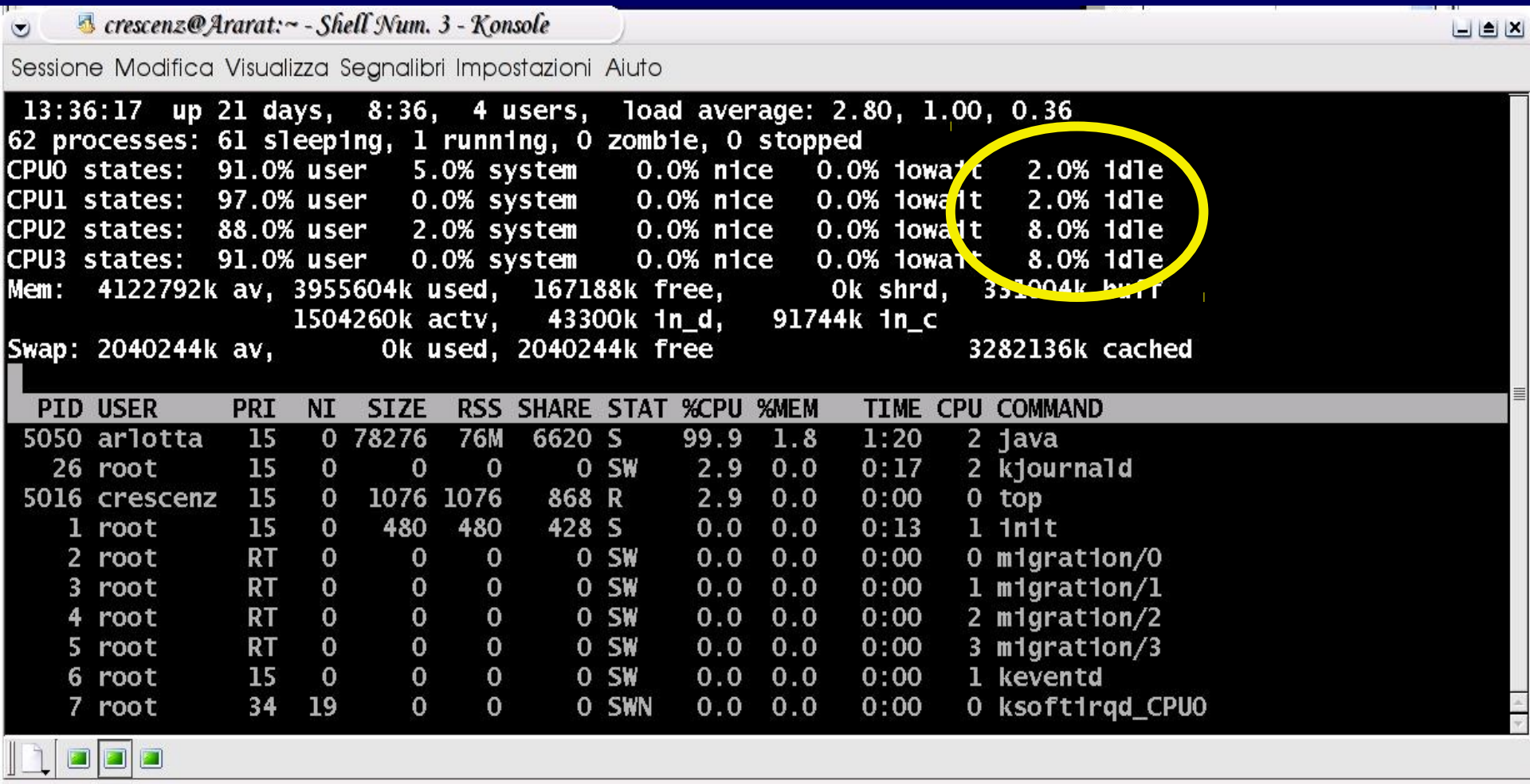
Programmazione Concorrente: Motivazioni

- Migliorare la comprensione di un SO che regola diverse attività parallele
- Sfruttare le prestazioni ottenibili da architetture multi-processor
  - un programma sequenziale non beneficia di una architettura parallela >>
- Migliorare la reattività delle applicazioni all'input dell'utente durante lunghe operazioni di I/O o di elaborazione
- La maggiore naturalezza con la quale si possono scrivere alcune tipologie di applicazioni (server, robotica, giochi, simulazioni di attività concorrenti)
- Scalabilità… un problema sempre più attuale

Utilizzo dei Processori:
Applicazione mono-thread *CPU-intensive*



Utilizzo dei Processori: Stessa applicazione *CPU-intensive* riscritta in versione multi-thread



Istruzione ed Area Memoria

- Per ragionare a vari livelli di granularità, consideriamo astrattamente i due concetti di istruzione ed area di memoria
  - Istruzione; alcune possibili esemplificazioni:
    - istruzione macchina
    - istruzione firmware
    - uno statement java
    - un metodo di una classe java
    - un intero programma
    - una stored-procedure di un DBMS
    - la scrittura di un blocco del gestore della concorrenza di un DBMS
  - Area di Memoria; alcune possibili esemplificazioni:
    - un bit, un byte, una parola macchina
    - un campo di una struttura dati, una struttura dati intera
    - un attributo, una tupla, una tabella, un intero db
    - un blocco di un dispositivo di memoria secondaria

Flussi di Esecuzione Concorrenti

- Un *f.d.e. sequenziale* definisce un ordinamento totale sull'esecuzione delle istruzioni
- Un *f.d.e. concorrente* definisce un ordinamento parziale sull'esecuzione delle istruzioni
  - su alcune istruzioni l'esecutore è libero di scegliere quali iniziare prima e/o di eseguirle contemporaneamente

- Ideiamo un formalismo per esprimere semplici algoritmi concorrenti:

  Diagramma delle Precedenze

Diagramma delle Precedenze

- N.B. Consideriamo al momento solo algoritmi privi di istruzioni di iterazione
- Esprimiamo l'ordinamento parziale delle istruzioni con un grafo orientato aciclico i cui nodi sono in corrispondenza con le istruzioni da eseguire concorrentemente

- Esempio: programma che legge un valore *X*, ne calcola e stampa le prime quattro potenze. Si supponga di disporre di tre sole tipologie di istruzione:
  - *leggi <variabile>*
  - *scrivi <variabile>*
  - *<variabile>* ← *<variabile>* * *<variabile>*

Diagramma delle Precedenze: Esempio

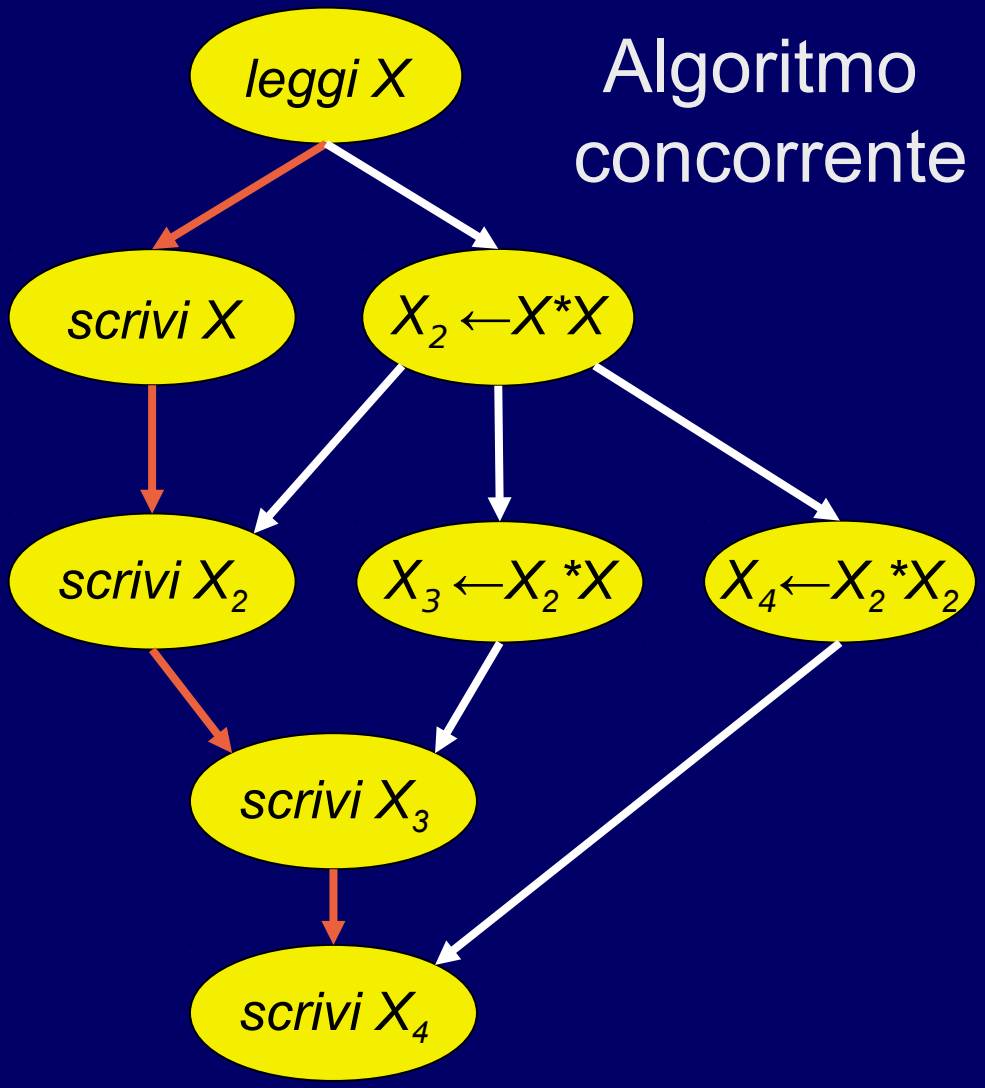
Algoritmo sequenziale

begin

1. *leggi X;*
2. *scrivi X;*
3. *X2 ← X * X;*
4. *scrivi X2;*
5. *X3 ← X2 * X;*
6. *scrivi X3;*
7. *X4 ← X2 * X2;*
8. *scrivi X4;*

end

Algoritmo concorrente



Esercizi

Esercizio: costruire un diagramma delle precedenze che esprima il massimo grado di parallelismo nel calcolo delle seguenti espressioni sullo stile dell'esempio appena visto:

a) (A+B)*(C+D)

b) A*X^4+B*X^3+C*X^2+D*X+E

c) (-B-SQRT(B^2-4*A*C))/(2*A)

Esecuzioni Sequenziali e Concorrenti

- Risulta utile modellare alcuni aspetti delle esecuzioni concorrenti
- Sono possibili molti modelli di diverse complessità
- Quelli scelti si segnalano per la semplicità e l'essenzialità
  - *sequenze di esecuzioni ammissibili*
  - *sequenze di interleaving*
- Ad esempio, consentono di calcolare il numero totale di possibili esecuzioni concorrenti che possono scaturire dal medesimo algoritmo concorrente

- Il modello delle *sequenze di esecuzioni ammissibili*, si limita a modellare due eventi per ciascuna istruzione eseguita concorrentemente

Esecuzioni Sequenziali e Concorrenti

- Sia *i* una generica istruzione (in generale può essere divisibile in istruzioni più elementari)
- Siano I_i e F_i gli eventi di inizio e fine esecuzione
- Date due istruzioni a e b consideriamo i 6 possibili ordinamenti in cui occorrono i quattro eventi I_a, F_a, I_b, F_b

$I_a\ F_a\ I_b\ F_b$

$I_b\ F_b\ I_a\ F_a$

esecuzioni sequenziali

$I_a\ I_b\ F_a\ F_b$

$I_a\ I_b\ F_b\ F_a$

$I_b\ I_a\ F_a\ F_b$

$I_b\ I_a\ F_b\ F_a$

esecuzioni parallele

Sequenze di Esecuzione Ammissibili

- Una *sequenza di esecuzione ammissibile* è una sequenza di questi eventi che rispetta i vincoli espressi dal diagramma delle precedenze
- Ad un certo diagramma delle precedenze corrispondono molteplici sequenze di esecuzione ammissibili
- Es. di *s. e. a.*, con riferimento al diagramma delle precedenze visto prima:

$$I_{i_1} F_{i_1} I_{i_2} I_{i_3} F_{i_2} F_{i_3} I_{i_4} I_{i_7} I_{i_5} F_{i_5} F_{i_7} F_{i_4} I_{i_6} F_{i_6} I_{i_8} F_{i_8}$$

Sequenze di Interleaving

- Un caso speciale ma rilevante di sequenza di esecuzione ammissibile; consideriamo:
  - *un solo esecutore fisico*
  - *istruzioni indivisibili*

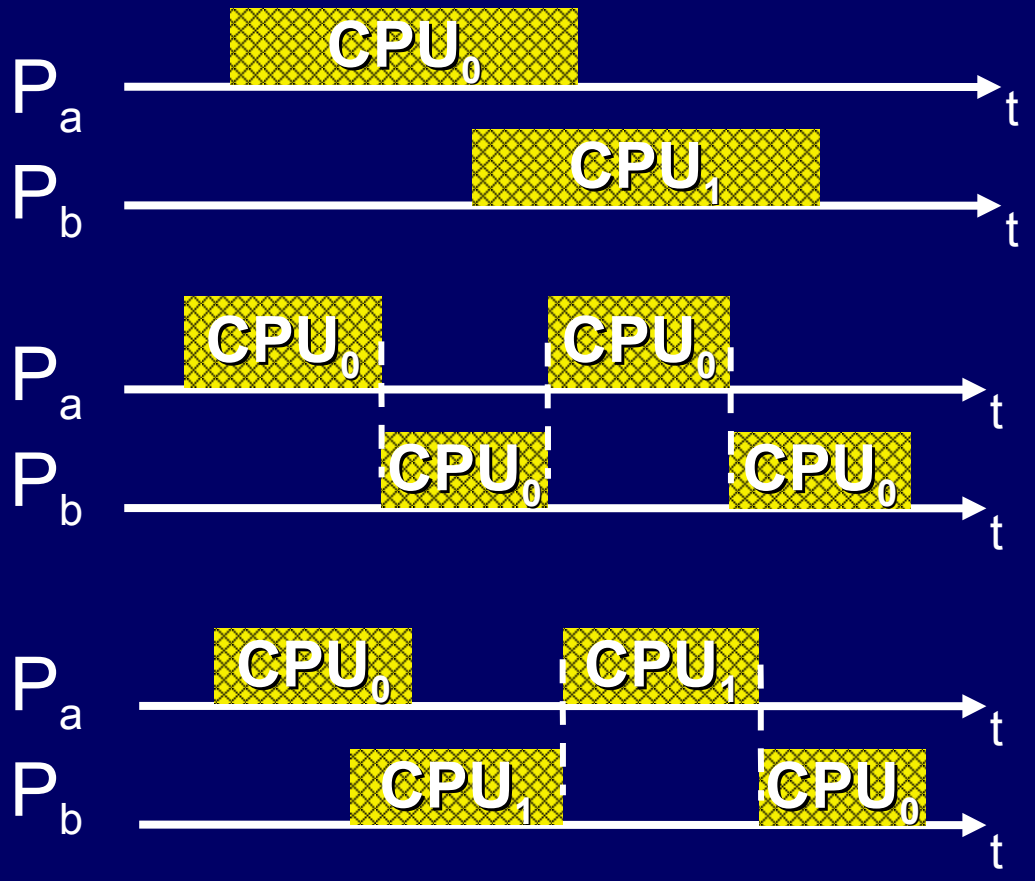
  - Esempio: due flussi di esecuzione sequenziali A e B con istruzioni
    - $a_1a_2a_3a_4...$
    - $b_1b_2b_3b_4...$
- Diciamo *sequenza di interleaving* la sequenza scelta dall'esecutore, ad esempio:
  - $a_1b_1b_2a_2b_3a_3a_4b_4...$
- Analogamente per tre o più flussi di esecuzione
- E' il modello preferito, quando applicabile, per la sua semplicità

Processori Virtuali

- Astrazione dei servizi offerti dai sistemi operativi moderni
- Molteplici esecutori virtuali possono essere implementati con uno o più processori fisici attraverso tecniche di context-switching
- In base al numero di processori fisici disponibili e al numero di f.d.e. esistenti, risultano possibili varie situazioni per farli avanzare concorrentemente
  - interleaving
  - overlapping
  - una combinazione di queste due

Overlapping ed Interleaving

- L'esecutore può eseguire più istruzioni concorrentemente mediante
  - overlapping
  - interleaving
  - combinazione



Speed-Up

- Il miglioramento delle prestazioni ottenuto grazie alla parallelizzazione del codice e l'esecuzione su di una architettura multi-processore con *N* CPU fisiche, è tipicamente espresso in termini di *speed-up,*
- Misurabile sperimentalmente come rapporto tra

  - il tempo impiegato da una esecuzione seriale

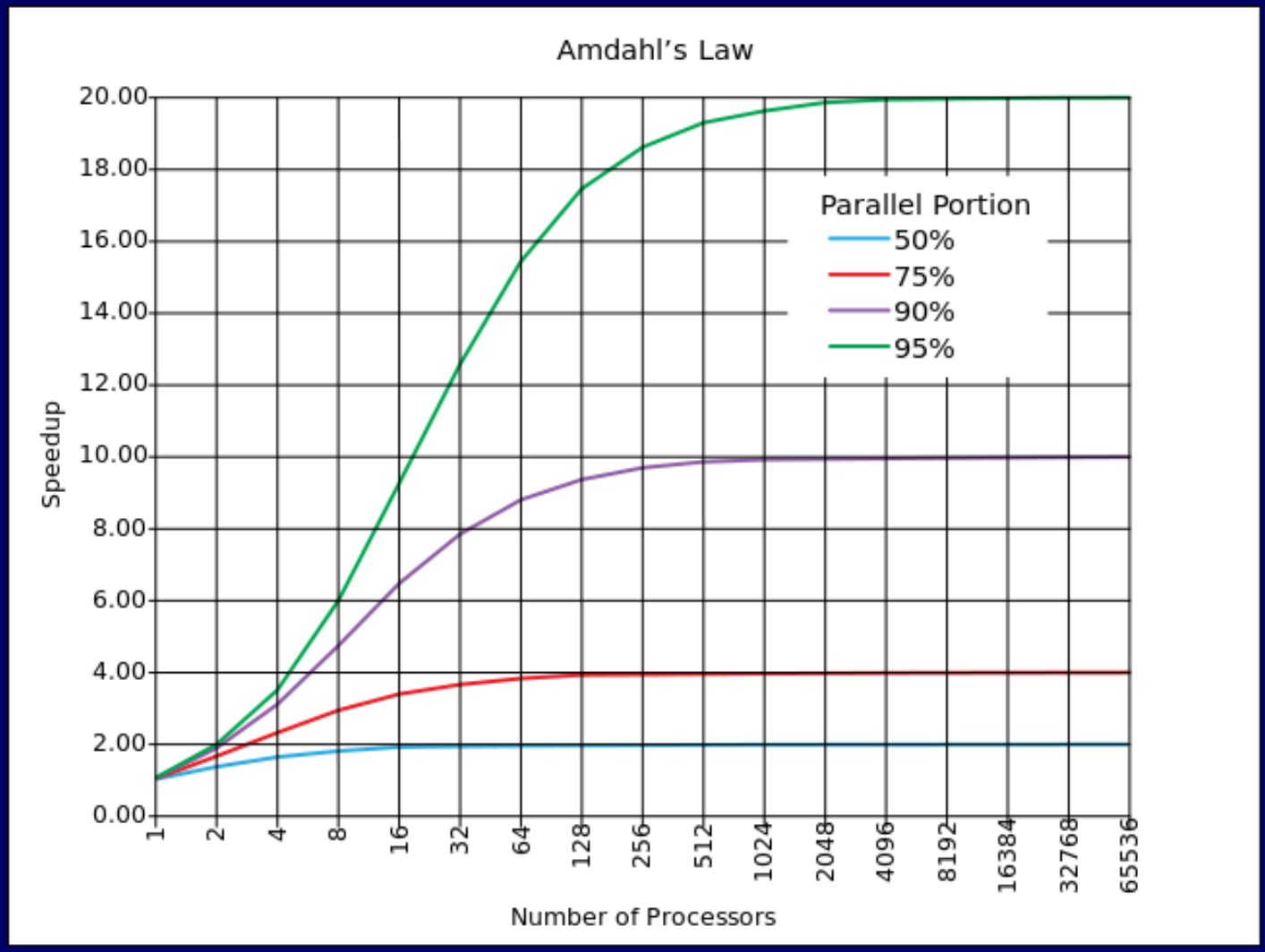
  - Il tempo impiegato da una esecuzione concorrente

- E' utopico sperare che possa raggiungere *N*
- Ogni programma concorrente contiene sempre una percentuale di lavoro *F* intrinsecamente non parallelizzabile

Legge di Amdahl

$$Speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

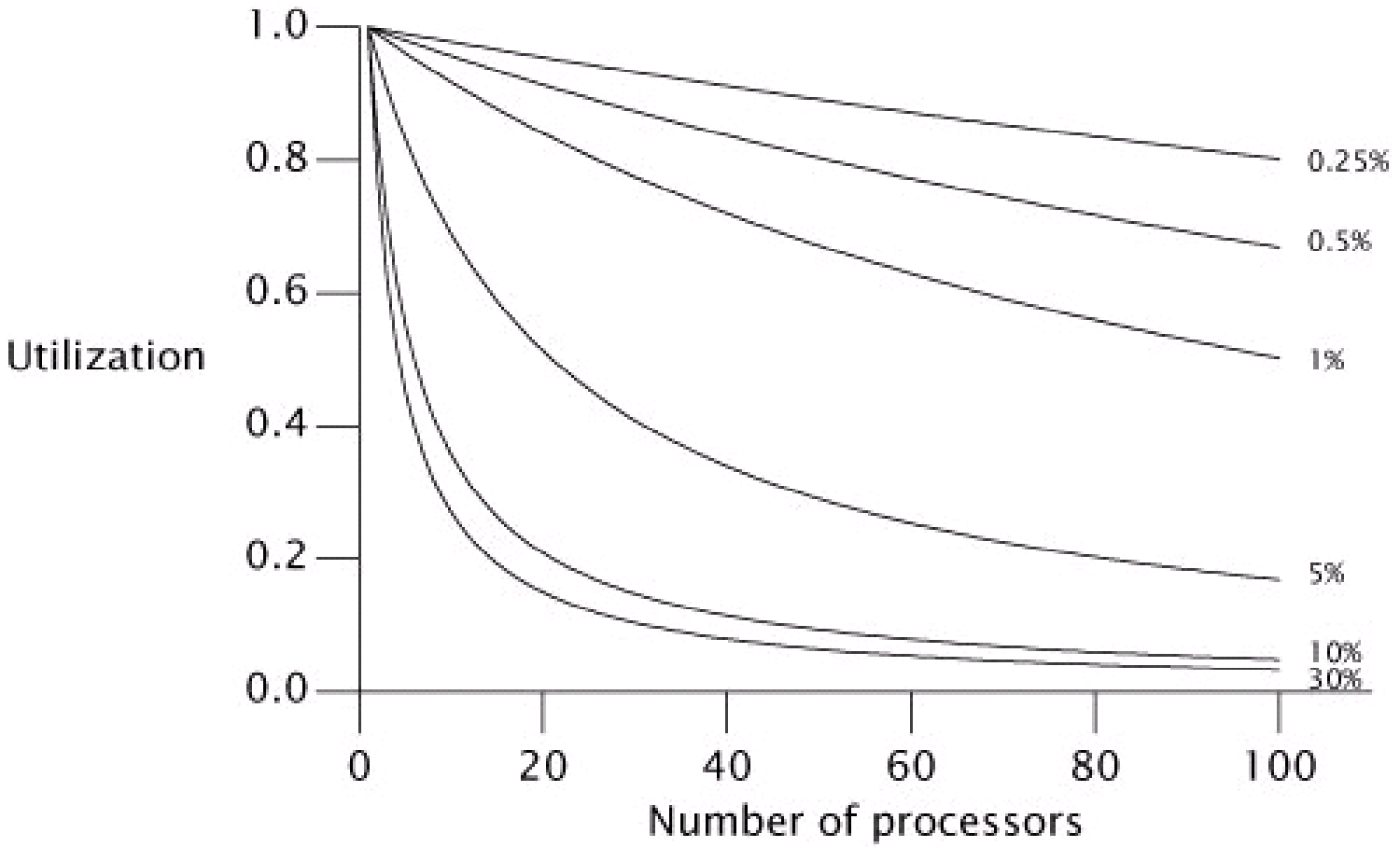
F: frazione di lavoro seriale
N: numero di processori



Legge di Amdahl

$$Speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

F: frazione di lavoro seriale
N: numero di processori



Commenti alla legge di Amdahl

- E' un modello estremamente semplice ma efficace per stimare il miglioramento ottenibile parallelizzando un algoritmo sequenziale
  - trascura numerosissimi elementi (es. *transitori*, costi del *contex-switching*)
  - utilizzabile solo quando il problema è di dimensioni talmente grandi da rendere questi elementi trascurabili
- Ogni programma concorrente contiene sempre una percentuale di lavoro intrinsecamente non parallelizzabile
  - Ad esempio la componente che si occupa di sincronizzare i flussi tra cui è stato ripartito il lavoro
  - … o quella che si occupa di stampare i risultati!
- Questa componente seriale limita il massimo speed-up ottenibile e rende antieconomico aumentare il numero di processori fisici oltre un certo numero

Alla Ricerca dello Speed-Up

- Dato un problema, si cerca di trovare una soluzione alla componente parallelizzabile che sia agevolmente decomponibile in sottoproblemi
- Si affidano i sottoproblemi ad esecutori distinti
- Si ricompongono (serialmente :(?) le soluzioni parziali

- Mai perdere di vista la legge di Amdahl
- Utilizzare tante CPU ha senso solo per certe tipologie di problemi (basso *F*) e solo per problemi di dimensioni relativamente grandi (grande *N*).
- E' meglio:
  - tenere occupati tutti gli esecutori fisici disponibili
  - *non* dissipare troppe risorse solo per la decomposizione
  - *non* creare più flussi di quelli che possono avanzare
  - *non* creare sottoproblemi troppo “piccoli”