Algoritmi paralleli e distribuiti

Indice

1. Introduzione

1.1. Definizione

Un **algoritmo** è una sequenza finita di istruzioni che non sono ambigue e che terminano, ovvero restituiscono un risultato

Gi **algoritmi sequenziali** avevano un solo esecutore, mentre gli algoritmi di questo corso utilizzano un **pool di esecutori**

Le problematiche da risolvere negli algoritmi sequenziali si ripropongono anche qua, ovvero:
- **progettazione**: utilizzo di tecniche per la risoluzione, come *Divide et Impera*, *programmazione dinamica* o *greedy*
- **valutazione delle prestazioni**: complessità spaziale e temporale
- **codifica**: implementare con opportuni linguaggi di programmazione i vari algoritmi presentati

I programmi diventano quindi una *sequenza di righe*, ognuna delle quali contiene *una o più* istruzioni

1.2. Algoritmi paralleli

Un **algoritmo parallelo** è un algoritmo **sincrono** che risponde al motto *"una squadra in cui batte un solo cuore"*, ovvero si hanno più entità che obbediscono ad un clock centrale, che va a coordinare tutto il sistema

Abbiamo la possibilità di condividere le risorse in due modi:
- memoria, formando le architetture
  - **a memoria condivisa**, ovvero celle di memoria fisicamente condivisa
  - **a memoria distribuita**, ovvero ogni entità salva parte dei risultati parziali sul proprio nodo
- uso di opportuni collegamenti

Qualche esempio di architettura parallela:
- **supercomputer**: cluster di processori con altissime prestazioni
- **GPU**: usate in ambienti grafici, molto utili anche in ambito vettoriale
- **processori multicore**
- **circuiti integrati**: insieme di gate opportunamente connessi

1.3. Algoritmi distribuiti

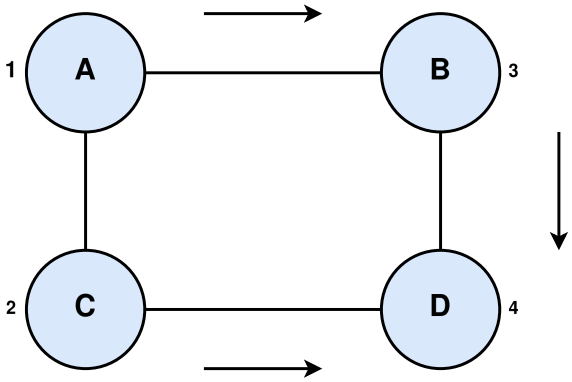
Un **algoritmo distribuito** è un algoritmo **asincrono** che risponde al motto *"ogni membro del pool è un mondo a parte"*, ovvero si hanno più entità che obbediscono al proprio clock personale

Abbiamo anche in questo caso dei collegamenti ma non dobbiamo supporre una memoria condivisa o qualche tipo di sincronizzazione, quindi dobbiamo utilizzare lo **scambio di messaggi**

Qualche esempio di architettura distribuita:
- **reti di calcolatori**: internet
- **reti mobili**: uso di diverse tipologie di connessione
- **reti di sensori**: sistemi con limitate capacità computazionali che rispondono a messaggi *ack*, *recover*, *wake up*, eccetera

1.4. Differenze

Vediamo un problema semplicissimo: *sommare quattro numeri A,B,C,D*



Usiamo la primitiva `send(sorgente,destinazione)` per l'invio di messaggi

Un approccio parallelo a questo problema è il seguente

SOMMA DI QUATTRO NUMERI(A, B, C, D):
1 send(1,2), send(3,4)
2 A+B, C+D
3 send(2,4)
4 A+B+C+D

Un approccio distribuito invece non può seguire questo pseudocodice, perché le due send iniziali potrebbero avvenire in tempi diversi

Notiamo come negli algoritmi paralleli ciò che conta è il **tempo**, mentre negli algoritmi distribuiti ciò che conta è il **coordinamento**

1.5. Definizione di tempo

Il **tempo** è una variabile fondamentale nell'analisi degli algoritmi: lo definiamo come la funzione $t(n)$ tale per cui

$$T(x) = \text{numero di operazioni elementari sull'istanza } x$$
$$t(n) = \max\{T(x) \mid x \in \Sigma^n\},$$

dove n è la grandezza dell'input

Spesso saremo interessati al *tasso di crescita* di $t(n)$, definito tramite funzioni asintotiche, e non ad una sua valutazione precisa

Date $f, g : \mathbb{N} \longrightarrow \mathbb{N}$, le principali funzioni asintotiche sono
- $f(n) = O(g(n)) \iff f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$
- $f(n) = \Omega(g(n)) \iff f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$
- $f(n) = \Theta(g(n)) \iff c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$

Il tempo $t(n)$ dipende da due fattori molto importanti: il **modello di calcolo** e il **criterio di costo**

1.5.1. Modello di calcolo

Un modello di calcolo mette a disposizione le **operazioni elementari** che usiamo per formulare i nostri algoritmi

Ad esempio, una funzione *palindroma* in una architettura con memoria ad accesso casuale impiega $O(n)$ accessi, mentre una DTM impiega $\Theta(n^2)$ accessi

1.5.2. Criterio di costo

Le dimensioni dei dati in gioco contano: il **criterio di costo uniforme** afferma che le operazioni elementari richiedono una unità di tempo, mentre il **criterio di costo logaritmico** afferma che le operazioni elementari richiedono un costo che dipende dal numero di bit degli operandi, ovvero dalla sua dimensione

1.6. Classi di complessità

Un problema è **risolto efficientemente** in tempo se e solo se è risolto da una DTM in tempo polinomiale

Abbiamo tre principali classi di equivalenza per gli algoritmi sequenziali:

- *P*, ovvero la classe dei problemi di decisione risolti efficientemente in tempo, o risolti in tempo polinomiale
- *FP*, ovvero la classe dei problemi generali risolti efficientemente in tempo, o risolti in tempo polinomiale
- *NP*, ovvero la classe dei problemi di decisione risolti in tempo polinomiale su una NDTM

Il famosissimo problema *P = NP* rimane ancora oggi aperto

2. Algoritmi paralleli

2.1. Sintesi

Il problema della **sintesi** si interroga su come costruire gli algoritmi paralleli, chiedendosi se sia possibile ispirarsi ad alcuni algoritmi sequenziali

2.2. Valutazione

Il problema della **valutazione** si interroga su come misurare il tempo e lo spazio, unendo questi due in un un parametro di efficienza E

Spesso lo spazio conta il **numero di processori/entità** disponibili

2.3. Universalità

Il problema dell'Universalità cerca di descrivere la classe dei problemi che ammettono problemi paralleli efficienti

Definiamo infatti una nuova classe di complessità, ovvero la classe *NC*, che descrive la classe dei problemi generali che ammettono problemi paralleli efficienti

Un problema appartiene alla classe *NC* se viene risolto in tempo *polilogaritmico* e in spazio polinomiale

Teorema 2.3.1 $NC \subseteq FP$

Dimostrazione Per ottenere un algoritmo sequenziale da uno parallelo faccio eseguire in sequenza ad una sola identità il lavoro delle entità che prima lavoravano in parallelo
Visto che lo spazio di un problema *NC* è polinomiale, posso andare a "comprimere" un numero polinomiale di operazioni in una sola entità
Infine, visto che il tempo di un problema *NC* è polilogaritmico, il tempo totale è un tempo polinomiale □

Come per $P = NP$, qui il dilemma aperto è se vale $NC = FP$, ovvero se posso parallelizzare ogni algoritmo sequenziale efficiente

Per ora sappiamo che $NC \subseteq FP$, e che i problemi che appartengono a *FP* ma non a *NC* sono detti problemi *P*-completi