

Algoritmi paralleli e distribuiti

Indice

1. Introduzione	2
1.1. Definizione	2
1.2. Algoritmi paralleli	2
1.3. Algoritmi distribuiti	2
1.4. Differenze	2
1.5. Definizione di tempo	3
1.5.1. Modello di calcolo	3
1.5.2. Criterio di costo	4
1.6. Classi di complessità	4
2. Algoritmi paralleli	5
2.1. Sintesi	5
2.2. Valutazione	5
2.3. Universalità	5
2.4. Architetture	5
2.4.1. Memoria condivisa	5
2.4.2. Memoria distribuita	6
2.4.3. Modello PRAM	6
2.4.3.1. Definizione	6
2.4.3.2. Modelli per le istruzioni	7
2.4.3.3. Modelli per l'accesso alla memoria	7
2.4.3.4. Risorse di calcolo	7

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 processori 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 processore salva parte dei risultati parziali nei propri registri
- 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 processori che obbediscono al proprio clock personale

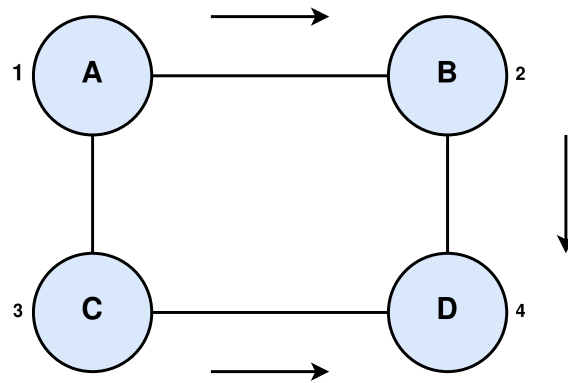
Abbiamo anche in questo caso dei collegamenti ma non dobbiamo supporre una memoria condivisa o qualche tipo di sincronizzazione, quindi dobbiamo utilizzare lo **scambio di messaggi**

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 $f, g : \mathbb{N} \rightarrow \mathbb{N}$, 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 parametro di efficienza E

Spesso lo spazio conta il **numero di processori** 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 un solo processore il lavoro dei processori 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 un solo processore
Infine, visto che il tempo di un problema NC è polilogaritmico, il tempo totale è un tempo polinomiale □

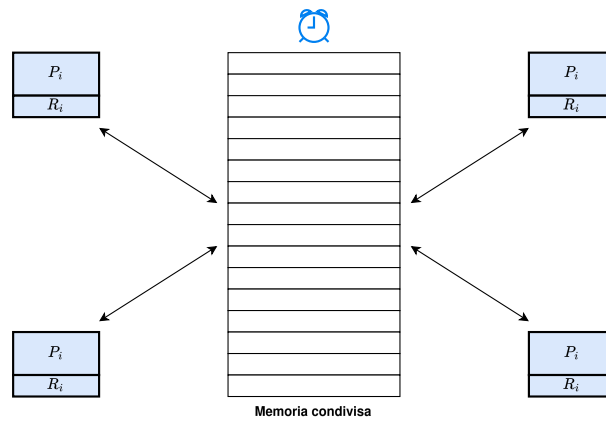
Come per $P = NP$, qui il dilemma aperto è se vale $NC = FP$, ovvero se posso parallelizzare ogni algoritmo sequenziale efficiente

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

2.4. Architetture

2.4.1. Memoria condivisa

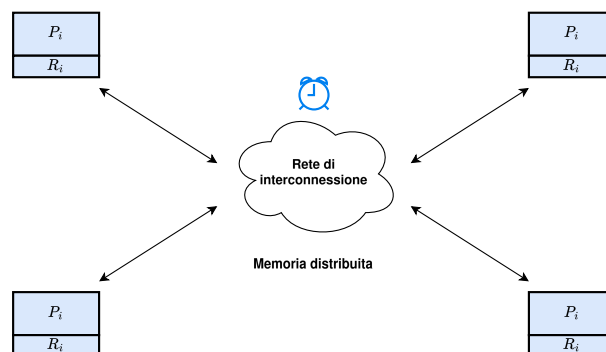
L'architettura a **memoria condivisa** utilizza una memoria centrale che permette lo scambio di informazioni tra un numero n di processori P_i , ognuno dei quali possiede anche una “memoria personale”, formata dai registri



Un **clock** centrale e comune coordina tutti i processori, che comunicano attraverso la memoria centrale in tempo costante $O(1)$, permettendo quindi una forte parallelizzazione

2.4.2. Memoria distribuita

L'architettura a **memoria distribuita** utilizza una rete di interconnessione centrale che permette lo scambio di informazioni tra un numero n di processori P_i , ognuno dei quali possiede anche una "memoria personale", formata dai registri



Un **clock** centrale e comune coordina tutti i processori, che comunicano attraverso la rete di interconnessione in un tempo che dipende dalla distanza tra i processori

2.4.3. Modello PRAM

2.4.3.1. Definizione

Il **modello PRAM** (Parallel RAM) utilizza una memoria M formata da registri $M[i]$ e una serie di processori P_i che si interfacciano con essa

Ogni processore P_i è una **RAM sequenziale**, ovvero contiene una unità di calcolo e una serie di registri $R[i]$

La comunicazione avviene con la memoria centrale tramite due primitive che lavorano in tempo costante $O(1)$:

- **LOAD** $R[dst]$ $M[src]$ per copiare nel registro dst il valore contenuto in memoria nella cella src
- **STORE** $R[src]$ $M[dst]$ per copiare in memoria nella cella dst il valore contenuto nel registro src

Le operazioni di ogni processore avvengono invece in locale, cioè con i dati della propria memoria privata

Il tempo di ogni processore P_i è scandito da un clock centrale, che fa eseguire ad ogni processore la “stessa istruzione” istruzione _{i}

Infatti, andiamo a definire il **passo parallelo** nel seguente modo

```
1  for all  $i \in \mathbb{I}$  par do:
2      istruzione $i$ 
```

In poche parole, tutti i processori con indice in \mathbb{I} eseguono l' i -esima istruzione, altrimenti eseguono una nop

2.4.3.2. Modelli per le istruzioni

L'istruzione eseguita dipende dal tipo di architettura:

- **SIMD** (Single Instruction Multiple Data) indica l'esecuzione della stessa istruzione ma su dati diversi
- **MIMD** (Multiple Instruction Multiple Data) indica l'esecuzione di istruzioni diverse sempre su dati diversi

2.4.3.3. Modelli per l'accesso alla memoria

Abbiamo diverse architetture anche per quanto riguarda l'accesso alla memoria:

- **EREW** (Exclusive Read Exclusive Write) indica una memoria con lettura e scrittura esclusive
- **CREW** (Concurrent Read Exclusive Write) indica una memoria con lettura simultanea e scrittura esclusiva
- **CRCW** (Concurrent Read Concurrent Write) indica una memoria con lettura e scrittura simultanee

Per quanto riguarda la scrittura simultanea abbiamo diverse modalità:

- **common**: i processori possono scrivere solo se scrivono lo stesso dato
- **random**: si sceglie un processore Π a caso
- **max/min**: si sceglie il processore Π con il dato massimo/minimo
- **priority**: si sceglie il processore P_i con priorità maggiore

La politica EREW è la più semplice, ma si può dimostrare che

$$\text{Algo(EREW)} \iff \text{Algo(CREW)} \iff \text{Algo(CRCW)}$$

Le implicazioni da sinistra verso destra sono “immediate”, mentre le implicazioni opposte necessitano di alcune trasformazioni










2.4.3.4. Risorse di calcolo

Essendo i singoli processori delle RAM, abbiamo ancora le risorse di tempo $t(n)$ e spazio $s(n)$, ma dobbiamo aggiungere

- $p(n)$ numero di processori richiesti su input di lunghezza n nel caso peggiore
- $T(n, p(n))$ tempo richiesto su input di lunghezza n e $p(n)$ processori nel caso peggiore

Notiamo come $T(n, 1)$ rappresenta il tempo sequenziale $t(n)$

Vediamo la struttura di un programma in PRAM

	p_1	p_2	...	$p(n)$
Passo 1	 $t_1^{(1)}(n)$	 $t_1^{(2)}(n)$...	 $t_1^{(p(n))}(n)$
Passo 2	 $t_2^{(1)}(n)$	 $t_2^{(2)}(n)$...	 $t_2^{(p(n))}(n)$
⋮	⋮	⋮	⋮	⋮
Passo $k(n)$	 $t_{k(n)}^{(1)}(n)$	 $t_{k(n)}^{(2)}(n)$...	 $t_{k(n)}^{(p(n))}(n)$

Ogni processore p_i esegue una serie di istruzioni nel passo parallelo, che possono essere più o meno in base al processore e al numero $p(n)$ di processori

Indichiamo con $t_i^{(j)}(n)$ il tempo che impiega il processore j -esimo per eseguire l' i -esimo passo parallelo su un input lungo n

Quello che vogliamo ricavare è il tempo complessivo del passo parallelo: visto che dobbiamo aspettare che ogni processore finisca il proprio passo, calcoliamo il tempo di esecuzione $t_i(n)$ dell' i -esimo passo parallelo come

$$t_i(n) = \max \left\{ t_i^{(j)}(n) \mid 1 \leq j \leq p(n) \right\}$$

Banalmente, il tempo complessivo di esecuzione del programma è la somma di tutti i tempi dei passi paralleli, quindi

$$T(n, p(n)) = \sum_{i=1}^{k(n)} t_i(n)$$

Notiamo subito come:

- T dipende da $k(n)$, ovvero dal numero di passi
- T dipende dalla dimensione dell'input
- T dipende da $p(n)$ perché diminuire/aumentare i processori causa un aumento/diminuzione dei tempi dei passi paralleli