

Algoritmi paralleli e distribuiti

Indice

1. Lezione 01	4
1.1. Introduzione	4
1.1.1. Definizione	4
1.1.2. Algoritmi paralleli	4
1.1.3. Algoritmi distribuiti	4
1.1.4. Differenze	4
2. Lezione 02	6
2.1. Definizione di tempo	6
2.1.1. Modello di calcolo	6
2.1.2. Criterio di costo	6
2.2. Classi di complessità	6
2.3. Algoritmi paralleli	6
2.3.1. Sintesi	6
2.3.2. Valutazione	7
2.3.3. Universalità	7
3. Lezione 03	8
3.1. Architetture	8
3.1.1. Memoria condivisa	8
3.1.2. Memoria distribuita	8
3.1.3. Modello PRAM	8
3.1.3.1. Definizione	8
3.1.3.2. Modelli per le istruzioni	9
3.1.3.3. Modelli per l'accesso alla memoria	9
3.1.3.4. Risorse di calcolo	9
4. Lezione 04	11
4.1. Parametri in gioco	11
5. Lezione 05	13
5.1. Sommatoria	13
5.1.1. EREW	15
5.1.2. Correttezza	15
6. Lezione 06	16
6.1. Ancora sommatoria	16
6.1.1. Dimostrazione	16
6.1.2. Valutazione	16
6.2. Sommatoria ottimizzata	17
6.2.1. Valutazione	17
7. Lezione 07	18
7.1. Ancora sommatoria	18
7.2. AND iterato	18
7.3. Prodotto interno di vettori	19
7.4. Prodotto matrice vettore	19
7.5. Prodotto matrice matrice	19
7.6. Potenza di matrice	20
7.7. Somme prefisse	20

8. Lezione 08	22
8.1. Ancora somme prefisse	22
9. Lezione 09	24
9.1. Ancora pointer doubling	24
9.2. Valutazione di polinomi	24
10. Lezione 10	26
10.1. Ancora valutazione di polinomi	26
10.2. Ricerca di un elemento	26
11. Lezione 11	28
11.1. Ordinamento	28
11.2. Primo approccio parallelo [counting sort]	28
11.3. Secondo approccio parallelo [bitonic sort]	29
12. Lezione 12	31
13. Lezione 13	34
13.1. BitSort	34
13.2. Osservazioni	35
13.3. Tecnica del ciclo euleriano	35

1. Lezione 01

1.1. Introduzione

1.1.1. Definizione

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

1.1.2. 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.

1.1.3. Algoritmi distribuiti

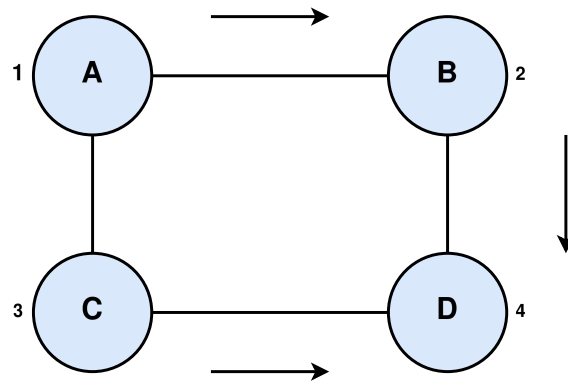
Un **algoritmo distribuito** è un algoritmo **asincrono** che risponde al motto «*ogni membro del pool è un mondo a parte*», ovvero si hanno più entità che obbediscono al proprio clock personale. Abbiamo anche in questo caso dei collegamenti ma non dobbiamo supporre una memoria condivisa o qualche tipo 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.

1.1.4. 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

input:

└ quattro numeri A, B, C, D

1: `send(1, 2), send(3, 4)`

2: calcola $A + B$ e $C + D$

3: `send(2, 4)`

4: calcola $(A + B) + (C + D)$

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**.

2. Lezione 02

2.1. 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**.

2.1.1. 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.

2.1.2. 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.

2.2. 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.

2.3. Algoritmi paralleli

2.3.1. Sintesi

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

2.3.2. 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/entità** disponibili.

2.3.3. 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 2.3.3.1:

$$NC \subseteq FP .$$

Dimostrazione 2.3.3.1: 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. ■

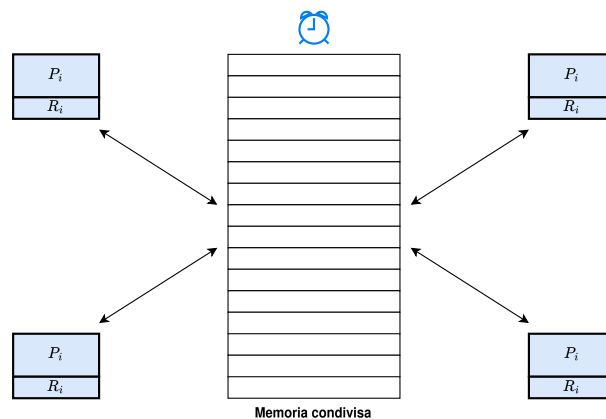
Come per $P = NP$, qui il dilemma aperto è se vale $NC = FP$, ovvero se posso parallelizzare ogni algoritmo sequenziale efficiente. Per ora sappiamo che $NC \subseteq FP$, e che i problemi che appartengono a FP ma non a NC sono detti problemi P -completi.

3. Lezione 03

3.1. Architetture

3.1.1. Memoria condivisa

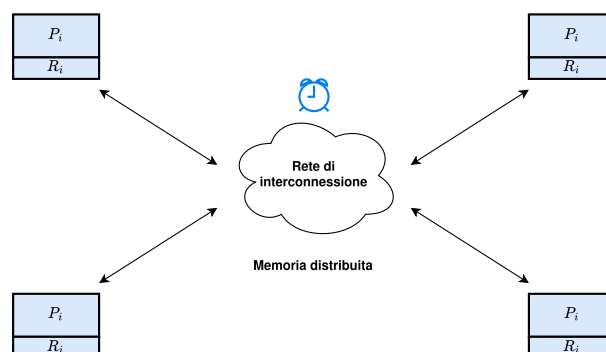
L'architettura a **memoria condivisa** utilizza una memoria centrale che permette lo scambio di informazioni tra un numero n di processori P_i , ognuno dei quali possiede anche una «memoria personale», formata dai registri.



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.

3.1.2. 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.

3.1.3. Modello PRAM

3.1.3.1. 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  $i \in \mathbb{I}$  par do:
2:    $\parallel$   $istruzione_i$ 

```

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

3.1.3.2. 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.

3.1.3.3. 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

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

Le implicazioni da sinistra verso destra sono «immediate», mentre le implicazioni opposte necessitano di alcune trasformazioni.










3.1.3.4. 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.

4. Lezione 04

4.1. Parametri in gioco

Confrontando $T(n, p(n))$ con $T(n, 1)$ abbiamo due casi:

- $T(n, p(n)) = \Theta(T(n, 1))$, caso che vogliamo evitare;
- $T(n, p(n)) = o(T(n, 1))$, caso che vogliamo trovare.

Introduciamo lo **speed-up**, il primo parametro utilizzato per l'analisi di un algoritmo parallelo: viene definito come

$$S(n, p(n)) = \frac{T(n, 1)}{T(n, p(n))}.$$

Se ad esempio $S = 4$ vuol dire che l'algoritmo parallelo è 4 volte più veloce dell'algoritmo sequenziale, ma questo vuol dire che sono nel caso di $T(n, p(n)) = \Theta(T(n, 1))$, poiché il fattore che definisce la complessità si semplifica.

Vogliamo quindi avere $S \rightarrow \infty$, poiché è la situazione di o piccolo che tanto desideriamo. Questo primo parametro è ottimo ma non basta: stiamo considerando il numero di processori? **NO**, questo perché $p(n)$ non compare da nessuna parte, e quindi noi potremmo avere $S \rightarrow \infty$ perché stiamo utilizzando un numero spropositato di processori.

Ad esempio, nel problema di soddisfacibilità SODD potremmo utilizzare 2^n processori, ognuno dei quali risolve un assegnamento, poi con vari passi paralleli andiamo ad eseguire degli *OR* per vedere se siamo riusciti ad ottenere un assegnamento valido di variabili, tutto questo in tempo $\log_2 2^n = n$. Questo ci manda lo speed-up ad un valore che a noi piace, ma abbiamo utilizzato troppi processori.

Introduciamo quindi la variabile di **efficienza**, definita come

$$E(n, p(n)) = \frac{S(n, p(n))}{p(n)} = \frac{T(n, 1)^*}{T(n, p(n)) \cdot p(n)},$$

dove $T(n, 1)^*$ indica il miglior tempo sequenziale ottenibile.

Teorema 4.1.1:

$$0 \leq E \leq 1.$$

Dimostrazione 4.1.1: La dimostrazione di $E \geq 0$ risulta banale visto che si ottiene come rapporto di tutte quantità positive o nulle.

La dimostrazione di $E \leq 1$ richiede di sequenzializzare un algoritmo parallelo, ottenendo un tempo $\tilde{T}(n, 1)$ che però «fa peggio» del miglior algoritmo sequenziale $T(n, 1)$, quindi

$$T(n, 1) \leq \tilde{T}(n, 1) \leq p(n) \cdot t_1(n) + \dots + p(n)t_{k(n)}(n).$$

La somma di destra rappresenta la sequenzializzazione dell'algoritmo parallelo, che richiede quindi un tempo uguale $p(n)$ volte il tempo che prima veniva eseguito al massimo in un passo parallelo.

Risolvendo il membro di destra otteniamo

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

Se andiamo a dividere tutto per il membro di destra otteniamo quello che vogliamo dimostrare, ovvero

$$T(n, 1) \leq p(n) \cdot T(n, p(n)) \Rightarrow \frac{T(n, 1)}{p(n) \cdot T(n, p(n))} \leq 1 \Rightarrow E \leq 1.$$

■

Se $E \rightarrow 0$ abbiamo dei problemi, perché nonostante un ottimo speed-up stiamo tendendo a 0, ovvero il numero di processori è eccessivo. Devo quindi ridurre il numero di processori $p(n)$ senza degradare il tempo, passando da p a $\frac{p}{k}$.

L'algoritmo parallelo ora non ha più p processori, ma avendone di meno per garantire l'esecuzione di tutte le istruzioni vado a raggruppare in gruppi di k le istruzioni sulla stessa riga, così che ogni processore dei $\frac{p}{k}$ a disposizione esegua k istruzioni.

Il tempo per eseguire un blocco di k istruzioni ora diventa $k \cdot t_i(n)$ nel caso peggiore, mentre il tempo totale diventa

$$T\left(n, \frac{p}{k}\right) \leq \sum_{i=1}^{k(n)} k \cdot t_i(n) = k \sum_{i=1}^{k(n)} t_i(n) = k \cdot T(n, p(n)).$$

5. Lezione 05

Iniziamo ad avere dei problemi quando $E \rightarrow 0$: infatti, secondo il **principio di Wyllie**, se $E \rightarrow 0$ quando $T(n, p(n)) = o(T(n, 1))$ allora è $p(n)$ che sta crescendo troppo. In poche parole, abbiamo uno speed-up ottimo ma abbiamo un'efficienza che va a zero per via del numero di processori.

Riprendiamo dalla scorsa lezione.

Calcoliamo l'efficienza con questo nuovo numero di processori, per vedere se è migliorata:

$$E\left(n, \frac{p}{k}\right) = \frac{T(n, 1)}{\frac{p}{k} \cdot T(n, \frac{p}{k})} \geq \frac{T(n, 1)}{\frac{p}{k} \cdot k \cdot T(n, p(n))} = \frac{T(n, 1)}{p(n) \cdot T(n, p(n))} = E(n, p(n)).$$

Notiamo quindi che diminuendo il numero di processori l'efficienza aumenta.

Possiamo dimostrare infine che la nuova efficienza è comunque limitata superiormente da 1

$$E(n, p(n)) \leq E\left(n, \frac{p}{k}\right) \leq E\left(n, \frac{p}{p}\right) = E(n, 1) = 1.$$

Dobbiamo comunque garantire la condizione di un buon speed-up, quindi $T(n, \frac{p}{k}) = o(T(n, 1))$

5.1. Sommatoria

Cerchiamo un algoritmo parallelo per il calcolo di una **sommatoria**.

Il programma prende in input una serie di numeri $M[1], \dots, M[n]$ inseriti nella memoria della PRAM e fornisce l'output in $M[n]$. In poche parole, a fine programma si avrà

$$M[n] = \sum_{i=1}^n M[i].$$

Un buon algoritmo sequenziale è quello che utilizza $M[n]$ come accumulatore, lavorando in tempo $T(n, 1) = n - 1$ senza usare memoria aggiuntiva.

Sommatoria sequenziale

input:

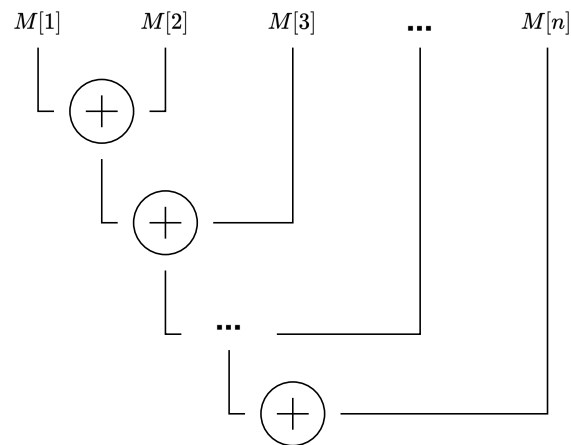
 ↳ vettore $M[]$ di grandezza n

1: for $i = 1$ to n do:

 2: ↳ $M[n] = M[n] + M[i]$

3: return $M[n]$

Un primo approccio parallelo potrebbe essere quello di far eseguire ad ogni processore una somma.



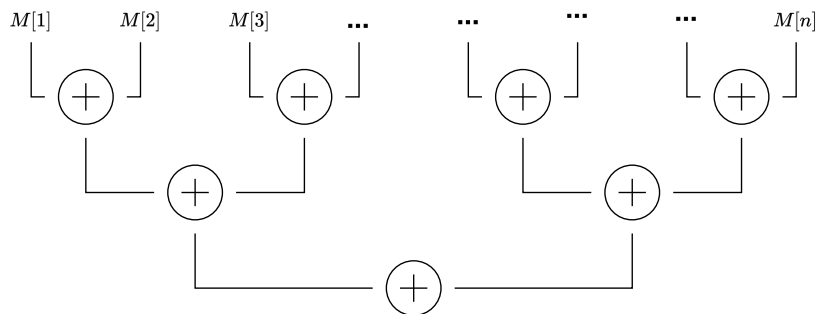
Usiamo $n - 1$ processori, ma abbiamo dei problemi:

- l'albero che otteniamo ha altezza $n - 1$;
- ogni processore deve aspettare la somma del processore precedente, quindi $T(n, n - 1) = n - 1$.

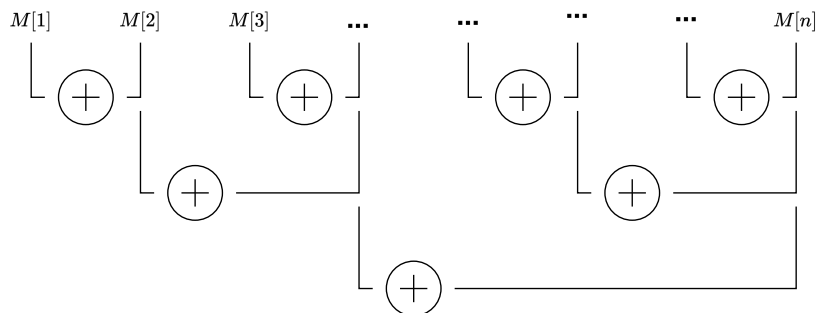
L'efficienza che otteniamo è

$$E(n, n - 1) = \frac{n - 1}{(n - 1) \cdot (n - 1)} \rightarrow 0.$$

Una soluzione migliore considera la *proprietà associativa* della somma per effettuare delle somme 2 a 2.



Quello che otteniamo è un albero binario, sempre con $n - 1$ processori ma l'altezza dell'albero logaritmica in n . Il risultato di ogni somma viene scritto nella cella di indice maggiore, quindi vediamo la rappresentazione corretta.



Quello che possiamo fare è sommare, ad ogni passo i , gli elementi che sono a distanza i : partiamo sommando elementi adiacenti a distanza 1, poi 2, fino a sommare al passo $\log(n)$ gli ultimi due elementi a distanza $\frac{n}{2}$.

Sommatoria parallela

```

1: for  $i = 1$  to  $\log(n)$  do:
2:   for  $k = 1$  to  $\frac{n}{2^i}$  par do:
3:      $M[2^i k] = M[2^i k] + M[2^i k - 2^{i-1}]$ 
4: return  $M[n]$ 

```

Nell'algoritmo k indica il numero di processori attivi nel passo parallelo.

5.1.1. EREW

Teorema 5.1.1.1: L'algoritmo di sommatoria parallela è EREW.

Dimostrazione 5.1.1.1: Dobbiamo mostrare che al passo parallelo i il processore a , che utilizza $2^i a$ e $2^i a - 2^{i-1}$, legge e scrive celle di memoria diverse rispetto a quelle usate dal processore b , che utilizza $2^i b$ e $2^i b - 2^{i-1}$.

Mostriamo che $2^i a \neq 2^i b$: questo è banale se $a \neq b$.

Mostriamo infine che $2^i a \neq 2^i b - 2^{i-1}$: supponiamo per assurdo che siano uguali, allora $2 \cdot \frac{2^i a}{2^i} = 2 \cdot \frac{2^i b - 2^{i-1}}{2^i} \implies 2a = 2b - 1 \implies a = \frac{2b-1}{2}$ ma questo è assurdo perché $a \in \mathbb{N}$. ■

5.1.2. Correttezza

Teorema 5.1.2.1: L'algoritmo di sommatoria parallela è corretto.

Dimostrazione 5.1.2.1: Per dimostrare che è corretto mostriamo che al passo parallelo i nella cella $2^i k$ ho i $2^i - 1$ valori precedenti, sommati a $M[2^i k]$, ovvero che $M[2^i k] = M[2^i k] + \dots + M[2^i(k-1) + 1]$.

Notiamo che se $i = \log(n)$ allora ho un solo processore $k = 1$ e ottengo la definizione di sommatoria, ovvero $M[n] = M[n] + \dots + M[1]$.

Dimostriamo per induzione.

Passo base: se $i = 1$ allora $M[2k] = M[2k] + M[2k - 1]$. ■

Continuiamo la prossima volta.

6. Lezione 06

6.1. Ancora sommatoria

6.1.1. Dimostrazione

Finiamo la dimostrazione della scorsa lezione.

Dimostrazione 6.1.1.1: Per dimostrare che è corretto mostriamo che al passo parallelo i nella cella $2^i k$ ho i $2^i - 1$ valori precedenti, sommati a $M[2^i k]$, ovvero che $M[2^i k] = M[2^i k] + \dots + M[2^i(k-1) + 1]$.

Notiamo che se $i = \log(n)$ allora ho un solo processore $k = 1$ e ottengo la definizione di sommatoria, ovvero $M[n] = M[n] + \dots + M[1]$.

Dimostriamo per induzione.

Passo base: se $i = 1$ allora $M[2k] = M[2k] + M[2k - 1]$.

Passo induttivo: supponiamo sia vero per $i - 1$, dimostriamo che vale per i . Sappiamo che al generico passo k eseguiamo l'operazione $M[2^i k] = M[2^i k] + M[2^i k - 2^{i-1}]$.

Andiamo a riscrivere i due fattori della somma in un modo a noi più comodo:

- $M[2^i k] = M[2^{i-1} \cdot 2k] = M[2^{i-1} \cdot 2k] + \dots + M[2^{i-1} \cdot (2k - 1) + 1]$ perché vale l'ipotesi del passo induttivo;
- $M[2^i k - 2^{i-1}] = M[2^{i-1} \cdot (2k - 1)] = M[2^{i-1} \cdot (2k - 1)] + \dots + M[2^{i-1} \cdot (2k - 2) + 1]$ sempre per l'ipotesi del passo induttivo.

Notiamo ora che il primo e il secondo fattore sono contigui: infatti, l'ultima cella del primo fattore è un indice superiore rispetto alla prima della del secondo fattore. Inoltre, l'ultima cella del secondo fattore $M[2^{i-1} \cdot (2k - 2) + 1]$ può essere riscritta come $M[2^i(k-1) + 1]$, quindi abbiamo ottenuto esattamente quello che volevamo dimostrare. ■

6.1.2. Valutazione

Se n è potenza di 2 usiamo un numero massimo di processori uguale a $\frac{n}{2}$ e un tempo $T(n, \frac{n}{2}) = 4 \log(n)$, dovuto alle microistruzioni che vengono fatte in ogni passo parallelo.

Se n non è potenza di 2 dobbiamo «allungare» l'input fino a raggiungere una dimensione uguale alla potenza di 2 più vicina, aggiungendo degli zeri in coda, ma questo non va ad intaccare le prestazioni perché la nuova dimensione è limitata da $2n$.

Infatti, con lunghezza $2n$ abbiamo un numero di processori uguale a n e un tempo $T(n, n) = 4 \log(2n) \leq 5 \log(n)$. In poche parole:

- $p(n) = O(n)$;
- $T(n, p(n)) = O(\log(n))$.

Se però calcoliamo l'efficienza otteniamo

$$E(n, n) = \frac{n - 1}{n \cdot 5 \log(n)} \rightarrow 0,$$

quindi dobbiamo trovare una soluzione migliore, anche se E tende a 0 lentamente.

6.2. Sommatoria ottimizzata

Il problema principale di questo approccio è che i processori sono un po' sprecati: prima vengono utilizzati tutti, poi ne vengono usati sempre di meno. Usiamo l'approccio di Wyllie: vogliamo arrivare ad avere $E \rightarrow k \neq 0$ diminuendo il numero di processori utilizzati.

Andiamo quindi ad utilizzare p processori, con $p < n$, raggruppando i numeri presenti in M in gruppi grandi $\Delta = \frac{n}{p}$, ognuno associato ad un processore.

Come prima, andiamo a mettere la somma di un gruppo Δ_i nella cella di indice maggiore. Al primo passo parallelo ogni processore esegue la somma sequenziale dei Δ valori contenuti nel proprio gruppo, ovvero $M[k\Delta] = M[k\Delta] + \dots + M[(k-1)\Delta + 1]$. I successivi passi paralleli eseguono l'algoritmo sommatoria proposto prima sulle celle di memoria $M[\Delta], M[2\Delta], \dots, M[p\Delta]$, e in quest'ultima viene inserito il risultato finale.

6.2.1. Valutazione

In questa versione ottimizzata usiamo $p(n) = p$ processori e abbiamo un tempo $T(n, p)$ formato dal primo passo parallelo «di ottimizzazione» sommato al tempo dei passi successivi, quindi $T(n, p) = \frac{n}{p} + 5 \log(p)$.

Andiamo a calcolare l'efficienza $E(n, p) = \frac{n-1}{p \cdot (\frac{n}{p} + 5 \log(p))} = \frac{n-1}{\underbrace{n + 5p \log(p)}_n} \approx \frac{n}{2n} = \frac{1}{2}$, che è il valore diverso da 0 che volevamo.

Per fare questo dobbiamo imporre $5p \log(p) = n$, quindi $p = \frac{n}{5 \log(n)}$ (anche se non ho ben capito questo cambio di variabile, ma va bene lo stesso).

Con questa assunzione riusciamo ad ottenere un tempo $T(n, p(n)) = 5 \log(n) + \dots + 5 \log(n) \leq 10 \log(n)$.

7. Lezione 07

7.1. Ancora sommatoria

Diamo un **lower bound**: per sommatoria possiamo visualizzare usando un albero binario, con le foglie dati di input e i livelli sono i passi paralleli. Il livello con più nodi dà il numero di processori e l'altezza dell'albero il tempo dell'algoritmo.

Se abbiamo altezza h , abbiamo massimo 2^h foglie, quindi

$$\text{foglie} = n \leq 2^h \implies h \geq \log(n)$$

quindi ho sempre tempo logaritmico.

La sommatoria può essere uno schema per altri problemi.

Operazione iterata: abbiamo op che è associativa, abbiamo:

- **input**: $M[1], \dots, M[n]$ valori
- **output**: calcolare $\text{op}_i M[i] \rightarrow M[n]$ ovvero calcolare op su una serie di valori e mettere nella cella finale.

Abbiamo soluzioni efficienti per questo:

- $p = O\left(\frac{n}{\log(n)}\right)$;
- $T = O(\log(n))$.

Con modelli PRAM più potenti (non EREW) possiamo ottenere un tempo costante (per AND e OR).

7.2. AND iterato

Supponiamo una CRCW-PRAM, vediamo il problema **and iterato**, ovvero $M[n] = \bigwedge_i M[i]$.

Qui abbiamo tempo costante perché la PRAM è più potente.

L'algoritmo è il seguente.

\bigwedge iterato

```
1: for  $1 \leq k \leq n$  par do
2:   if  $M[k] = 0$  then
3:      $M[n] = 0$ 
```

Serve CW con politica common, quindi scrivono i processori se il dato da scrivere è uguale per tutti, ma anche le altre vanno bene (random o priority).

Abbiamo:

- $p(n) = n$;
- $T(n, n) = 3$;
- $E(n, n) = \frac{n-1}{3n} \rightarrow \frac{1}{3}$.

Per \bigvee iterato stessa cosa, basta che almeno uno sia 1.

La sommatoria può essere usata anche come sotto-problema di altri, ad esempio:

- prodotto interno di vettori;
- prodotto matrice-vettore;
- prodotto matrice-matrice;
- potenza di una matrice.

7.3. Prodotto interno di vettori

- input $x, y \in \mathbb{N}^n$
- output $\langle x, y \rangle = \sum_{i=1}^n x_i \cdot y_i$

Il tempo sequenziale è $2n - 1$, n per le somme e $n - 1$ somme finali.

Sommatoria viene usata qua come modulo:

- prima fase: eseguo $\Delta = \log(n)$ prodotti in sequenza delle componenti e la somma dei valori del blocco in sequenza;
- seconda fase: somma di $p = \frac{n}{\log(n)}$ prodotti.

Per sommatoria ho:

- $p = c_1 \frac{n}{\log(n)}$;
- $t = c_2 \log(n)$.

Per la prima fase:

- $p = \frac{n}{\log(n)}$ quindi $\delta = \frac{n}{p} = \log(n)$;
- $t = c_3 \log(n)$.

Ma allora ho $p = \frac{n}{\log(n)}$ e $t = \log(n)$.

L'efficienza è

$$E = \frac{2n - 1}{\frac{n}{\log(n)} \cdot \log(n)} \rightarrow C \neq 0.$$

7.4. Prodotto matrice vettore

Roba di prima è modulo per questo.

- input: $A \in \mathbb{N}^{n \times n}$ e $x \in \mathbb{N}^n$
- output: $A \cdot x$

Il tempo sequenziale è $n(2n - 1) = 2n^2 - n$.

Idea: uso il modulo $\langle \dots, \dots \rangle$ in parallelo n volte. Il vettore se è acceduto simultaneamente dai moduli $\langle \rangle$ ci obbliga ad avere CREW.

Che prestazioni abbiamo? Abbiamo:

- $p(n) = n \frac{n}{\log(n)}$;
- $T(n, p(n)) = \log(n)$.

L'efficienza vale

$$E(n, T(n, p(n))) = \frac{n^2}{\frac{n^2}{\log(n)} \log(n)} \rightarrow C \neq 0.$$

7.5. Prodotto matrice matrice

Modulo uso sempre prodotto interno.

- input: $A, B \in \mathbb{N}^{n \times n}$;
- output: $A \cdot B$.

Il tempo sequenziale è $n^{2.8}$ per Strassen.

Uso n^2 prodotti interni in parallelo, anche qui CREW perché ogni riga di A e ogni colonna di B viene acceduta simultaneamente.

Prestazioni:

- $p(n) = n^2 \frac{n}{\log(n)}$;
- $T(n, p(n)) = \log(n)$.

L'efficienza vale

$$E(n, T(n, p(n))) = \frac{n^{2.80}}{\frac{n^3}{\log(n)} \log(n)} \rightarrow 0.$$

Tende a 0 ma lentamente.

7.6. Potenza di matrice

- input: $A \in \mathbb{N}^{n \times n}$;
- output: A^t con $t = 2k$.

Prodotto iterato della stessa matrice, sequenziale è:

Potenza di matrice sequenziale

1: for $i = 1$ to $\log(n)$ do
2: $A = A \cdot A$

Saltiamo i calcoli intermedi, facciamo $A \rightarrow A^2 \rightarrow A^4 \rightarrow A^8 \rightarrow \dots$

Il tempo è quindi $n^{2.8} \log(n)$.

L'approccio parallelo per $\log(n)$ volte esegue il prodotto $A \cdot A$, anche questo CREW.

Abbiamo:

- $p(n) = \frac{n^3}{\log(n)}$;
- $T(n, p(n)) = \log(n) \cdot \log(n) = \log^2(n)$.

L'efficienza è

$$E = \frac{n^{2.8} \log(n)}{\frac{n^3}{\log(n)} \cdot \log^2(n)} = \frac{n^{2.8}}{n^3} \rightarrow 0.$$

Sempre lentamente.

7.7. Somme prefisse

Contiene anche lui il problema della sommatoria.

- input: $M[1], \dots, M[n]$;
- output: $\sum_{i=1}^k M[i] \rightarrow k \quad 1 \leq k \leq n$.

Assumiamo n potenza di 2 per semplicità.

L'algoritmo sequenziale somma nella cella i quello che c'è nella cella $i - 1$.

Algoritmo sequenziale furbo

1: for $k = 2$ to n do
2: $M[k] = M[k] + M[k - 1]$

Il tempo di questo algoritmo è $n - 1$.

Vediamo una proposta parallela. Al modulo sommatoria passo tutti i possibili prefissi: un modulo somma i primi due, un modulo i primi tre, eccetera.

Problemi:

- non è EREW ma questo chill;
- ho un CREW su PRAM con $p(n) \leq (n-1) \frac{n}{\log(n)} = \frac{n^2}{\log(n)} = \sum_{i=2}^n \frac{i}{\log(i)} \geq \frac{1}{\log(n)} \sum_{i=2}^n i \approx \frac{n^2}{\log(n)}$ e $T(n, p(n)) = \log(n)$.

Ma allora

$$E = \frac{n-1}{\frac{n^2}{\log(n)} \log(n)} \rightarrow 0$$

buuuuu poco efficiente.

8. Lezione 08

8.1. Ancora somme prefisse

Usiamo il **pointer doubling**, di Kogge-Stone del 1973.

Idea: sti stabiliscono dei legami tra i numeri, ogni processore si occupa di un legame e ne fa la somma: il processore i fa la somma tra m e k e lo mette nella cella di indice maggiore (quella di k).

All'inizio ho link tra la cella e la successiva.

Alla prima iterazione ho il primo, poi primo secondo, poi secondo terzo, eccetera. Poi aggiorno i link: lego una cella non con quella che avevamo prima ma con quella a distanza doppia. Prima 1, poi 2, poi 4, eccetera. Ovviamente alcuni processori non hanno dei successori.

Mi fermo quando non riesco a mettere archi, quindi alla fine non ho nessun successore.

Rispondiamo ad alcune domande:

- al passo j quanti elementi senza successori ho? 2^j ;
- quanti passi dura l'algoritmo? se $2^j = n$ allora $j = \log(n)$, ovvero termino quando ho esattamente n elementi senza successori;
- quanti processori attivo al passo j ? Sempre almeno uno, ma faccio sempre $1 \leq k \leq n - 2^{j-1}$;
- sia $S[k]$ il successivo di $M[k]$, come inizializzo S ? Faccio $S[k] = k + 1$ e $S[n] = 0$, perché all'inizio ho tutti i successori e poi l'ultimo che non ce l'ha;
- dato il processore p_k quale istruzione su M deve eseguire? $M[k] + M[S[k]] \rightarrow M[S[k]]$;
- aggiornamento? $S[k] == 0 ? 0 : S[S[k]]$ faccio successore del successore.

```
1: for  $j = 1$  to  $\log(n)$  do
2:   for  $1 \leq k \leq n - 2^{j-1}$  par do
3:      $M[S[k]] = M[k] + M[S[k]]$ 
4:      $S[k] = (S[k] == 0 ? 0 : S[S[k]])$ 
```

Competiamo per la stessa cella? No, siamo in EREW, accediamo alle stesse celle ma in momenti diversi.

Correttezza:

- è una EREW-PRAM perché p_k lavora su $M[k]$ e $M[S[k]]$ e se $i =, \neg j$ allora $S[i] \neq S[j]$ quindi hanno successori diversi (solo se $S[i] = S[j] = 0$);
- dimostro che $M[k] = \sum_{i=1}^k M[i]$. Dimostro che al j esimo passo vale

$$M[t] = \begin{cases} M[t] + \dots + M[1] & \text{se } t \leq 2^j \\ M[t] + \dots + M[t - 2^j + 1] & \text{se } t > 2^j \end{cases}$$

Se questa è vera allora per $j = \log(n)$ allora vale.

Per induzione su j :

- caso base $j = 1$:
 - se $t \leq 2$ vedi $t = 1, 2$;
 - se $t > 2$ ho la seconda proprietà.
- vero $j - 1$ dimostro per j . Al passo j quanto vale S :

$$S[k] = \begin{cases} k + 2^{j-1} & \text{se } k \leq n - 2^{j-1} \\ 0 & \text{maggiore} \end{cases}.$$

Le celle con indice $\leq 2^{j-1}$ proprietà vera per ipotesi. Le celle con indice:

- $2^{j-1} \leq t \leq 2^j$ allora $t = 2^{j-1} + a$ e quindi

$$M[a + 2^{j-1}] = \dots (\text{non ho voglia})$$

• $t > 2^j$ ho $t = a + 2^j$, bla bla bla.

9. Lezione 09

9.1. Ancora pointer doubling

Valutazione:

- $p(n) = n - 1$;
- il passo di aggiornamento di M vale 5 mentre il passo di aggiornamento di S vale 4, quindi $T(n, n - 1) \approx 9 \log(n)$ (il log viene dal passo parallelo).

L'efficienza è quindi

$$E(n, p(n)) = \frac{n - 1}{(n - 1)9 \log(n)} = \frac{1}{9 \log(n)} \rightarrow 0$$

ma lentamente, non va bene.

Sfruttiamo Willye per far sparire $\log(n)$ da sotto.

Mettiamo $p(n) = O\left(\frac{n}{\log(n)}\right)$ quindi a gruppi di $\log(n)$, avremo sempre tempo logaritmico ma andremo ad avere efficienza diversa da 0.

Questo può essere usato come modulo per OP-prefissa, dove in output ho

$$M[k] = \text{op}_{i=1}^k M[i] \quad 1 \leq k \leq n$$

operazione associativa come prima.

9.2. Valutazione di polinomi

- **Input:** $p(x) = a_0 + a_1x + \dots + a_nx^n$ e α ;
- **Output:** $p(\alpha)$.

In memoria ho α e $A[0], \dots, A[n]$ che tiene i coefficienti.

L'algoritmo sequenziale fa $\sum_{i=0}^n i = n^2(\text{prodotti}) + n(\text{somme}) \approx n^2$ operazioni nel metodo tradizionale.

Con Ruffini-Horner possiamo renderlo migliore, ovvero una raccolta di x iterativa, ottenendo

$$p(x) = a_0 + x(a_1 + \dots(a_{n-2} + x(a_{n-1} + a_nx))\dots).$$

Chiamo $p = a^n$, calcolo $a_{n-1} + a_n\alpha$ e questo lo chiamo p di nuovo e ricomincio. Vale

$$p = a_j + p\alpha.$$

```
1:  $p = a_n$ 
2: for  $i = 1$  to  $n$  do
3:    $p = a_{n-i} + p\alpha$ 
4: output  $p$ 
```

Le operazioni sono 2 per n volte quindi $T(n, 1) = 2n$ ed è sequenziale.

Che idea abbiamo per quello parallelo:

- costruisco il vettore delle potenze di α e lo chiamo Q ovvero $Q[k] = \alpha^k \quad 0 \leq k \leq n$;
- eseguo il prodotto interno tra A e Q , ovvero $\sum_{k=0}^n A[k]Q[k]$;
- ritorno il valore appena calcolato.

Come lo calcolo il vettore delle potenze? Metto α in tutti gli elementi di Q da 1 a n , applico il prodotto prefisso per il problema REPLICA. Come risolvo replica in parallelo?

```
1: for  $k = 1$  to  $n$  par do
2:    $Q[k] = \alpha$ 
```

Da finire bene.

10. Lezione 10

10.1. Ancora valutazione di polinomi

```
1: for  $k = 1$  to  $n$  par do:
2:    $Q[k] = \alpha$ 
```

L'algoritmo scritto è CREW (perché α è in memoria quindi ho accesso simultaneo), ha processori $p = n$, tempo $t = 2$ e quindi efficienza $E \rightarrow \frac{1}{2} \neq 0$. Se REPLICA è un modulo da usare forse posso fare meglio perché al passo dopo (con prodotto prefisso) ne uso di meno.

Abbassiamo il numero di processori con Willye, raggruppiamo in $\log(n)$ elementi. Il k -esimo processore carica α nelle celle di pozione $(k-1)\log(n) + 1, \dots, k\log(n)$.

Il secondo metodo è il seguente.

```
1: for  $k = 1$  to  $\frac{n}{\log(n)}$  par do
2:   for  $i = 1$  to  $\log(n)$  do
      $Q[(k-1)\log(n) + i] = \alpha$ 
```

Ha processori $p = \frac{n}{\log(n)}$, tempo $t = c \log(n)$ e efficienza $E = \frac{1}{c} \neq 0$. Rimane sempre CREW per l'accesso ad α simultaneo.

Vorremmo un EREW, quindi:

- costruiamo il vettore $\alpha, 0, \dots, 0$;
- eseguiamo somme prefisse

```
1:  $Q[1] = \alpha$ 
2: for  $k = 2$  to  $n$  par do
3:    $Q[k] = 0$ 
```

Posso anche ridurre i processori con Willye, avendo $p = \frac{n}{\log(n)}$ e tempo $t = \log(n)$ per costruire il vettore e poi usare le somme prefisse con $p = \frac{n}{\log(n)}$ e tempo $t = \log(n)$. Ora abbiamo un EREW.

Cosa abbiamo fatto quindi:

- A ce l'abbiamo in memoria;
- REPLICA di α ;
- prodotto prefisso;
- prodotto interno.

I processori sono $\frac{n}{\log(n)}$ e il tempo $\log(n)$, quindi l'efficienza è $C \neq 0$.

10.2. Ricerca di un elemento

- **Input:** $M[1], \dots, M[n]$ e α ;
- **Output:** $M[n] = 1$ se $\alpha \in M$, altrimenti 0.

Il sequenziale classico ha $t(n) = n$ (se ordinato è logaritmico).

Un algoritmo quantistico su non ordinato è $t = \sqrt{n}$ (usa interferenza quantistica).

Vediamo un CRCW parallelo con una flag F .

```

1:  $F = 0$ 
2: for  $k = 1$  to  $n$  par do
3:   if  $M[k] == \alpha$ 
4:      $F = 1$ 
5:  $M[n] = F$ 

```

Perché usiamo F ? Perché non posso sapere se poi $M[n] == 1$ è perché è il suo valore o perché l'ho trovato.

Ho la CR in α e la CW in F . I processori sono n e il tempo è costante, quindi

$$E =$$

Vediamo un CREW ora, quindi senza flag.

```

1: for  $k = 1$  to  $n$  par do
2:    $M[k] = (M[k] == \alpha ? 1 : 0)$ 
3: MAX-iterato

```

Trasformiamo in un vettore booleano e poi vediamo il massimo. Abbiamo n processori e tempo costante, ma con Willye andiamo a $p = \frac{n}{\log(n)}$ e tempo $\log(n)$, che sono uguali a quelli del max iterato. L'efficienza è quindi $E \approx C \neq 0$. Ho la CR per l'accesso ad alpha.

Vediamo infine un EREW.

```

1: REPLICA  $\alpha$  in  $A[1], \dots, A[n]$ 
2: for  $k = 1$  to  $n$  par do
3:    $M[k] = (M[k] == A[k] ? 1 : 0)$ 
4: MAX-iterato

```

Le prestazioni di tutti hanno processori $p = \frac{n}{\log(n)}$ e tempo $\log(n)$, quindi l'efficienza vale $E = C \neq 0$.

Varianti:

- conteggio di α dentro M , usando sommatoria al posto di MAX;
- posizione massima di α in M , assegnando $M[k] = k$ se c'è alpha nel vettore;
- posizione minima di α in M , usando una OP iterata tale che

$$OP(x, y) = \begin{cases} \min(x, y) & \text{se } x \\ y \neq 0 & \\ \max(x, y) & \text{altrimenti} \end{cases}.$$

11. Lezione 11

11.1. Ordinamento

Detto problema del **ranking**, abbiamo in input $M[1], \dots, M[n]$ e vogliamo in output una permutazione $p : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ tale che $M[p(1)] \leq \dots \leq M[p(n)]$ con $p(i)$ indice dell'elemento del vettore M che va in posizione i . Potremmo anche ordinare direttamente in M , ma noi usiamo gli indici. Quindi la funzione, dato un indice, mi dice che elemento va in quell'indice.

In genere, gli algoritmi di ordinamento sono basati sui confronti, ovvero $M[i] \leq M[j]$? SI : NO. Questi algoritmi hanno tempo $t = \Theta(n \log(n))$:

- upper bound: esistono algoritmi che impiegano al massimo quello, tipo merge sort
- lower bound: gli algoritmi di ordinamento creano degli alberi di decisione, ogni nodo è un confronto, SX positiva DX negativa. Ottengo un albero binario di decisione. Le foglie sono le permutazioni dell'input: ogni foglia individua un cammino a partire dalla radice e quindi i confronti che mi permettono di ordinare l'input. L'altezza dell'albero è il numero di confronti effettuati nel caso peggiore, ma questo è anche il tempo dell'algoritmo di ordinamento. Le possibili permutazioni dell'input sono $n!$, quindi foglie $\geq n!$. Se t è l'altezza, allora il massimo numero di foglie è 2^t , quindi

$$2^t \geq \text{foglie} \geq n! \implies t \geq \log_2(n!) \\ \log_2(n!) \geq \log_2 \left(\prod_{i=\frac{n}{2}+1}^n i \right) \geq \log_2 \left(\left(\frac{n}{2} \right)^{\frac{n}{2}} \right) = \frac{n}{2} \log_2 \left(\frac{n}{2} \right) \sim n \log_2(n).$$

Con la formula di Stirling è sicuramente più bella la dimostrazione. Quindi l'altezza è almeno $n \log(n)$ ma questo era il tempo quindi il tempo è così.

11.2. Primo approccio parallelo [counting sort]

Algoritmo basato sul conteggio, ovvero conta i confronti, sequenzialmente ha $t = \Theta(n^2)$ perché deve confrontare tutte le coppie. Assumiamo che n sia potenza di 2 e che gli elementi siano diversi tra loro.

Prendiamo d'esempio il counting sort sequenziale, ovvero $M[i]$ va in posizione k se e solo se k elementi sono $\leq M[i]$ in M .

Usiamo il vettore $V[1], \dots, V[n]$ con $V[i]$ che contiene k . In poche parole, è la permutazione inversa di p , perché sto dicendo che l'elemento i va in posizione k . Dell'elemento so la sua posizione finale. Infatti:

- permutazione normale: ti do la posizione, mi dici che elemento ci va
- permutazione inversa: ti do l'elemento, mi dici in che posizione va

Counting Sort sequenziale

```
1: for  $i = 1$  to  $n$ 
2:    $V[i] = 0$ 
3: for  $i = 1$  to  $n$ 
4:   for  $j = 1$  to  $n$ 
5:     if  $M[j] \leq M[i]$ 
6:        $V[i] = V[i] + 1$ 
7: for  $i = 1$  to  $n$ 
8:    $F[v[i]] = M[i]$ 
```

Counting Sort sequenziale

```
9: for  $i = 1$  to  $n$ 
10:    $M[i] = F[i]$ 
```

Le prime due fasi vanno già bene, le ultime due servono solo per ordinare effettivamente il vettore. Il numero di confronti è n^2 , visto il doppio for della fase 2. Fase più pesante è questa, e il tempo è $t = n^2$.

La versione parallela ha $\forall i, j$ un processore p_{ij} che calcola $M[j] \leq M[i]$ e aggiorna una matrice booleana $V[i, j]$ con il risultato del confronto. L' i -esima riga individua gli elementi di M che sono $\leq M[i]$. Poi, $\forall i$ effettuo una sommatoria parallela dell' i -esima riga. Ottengo un vettore colonna $V[1, n], \dots, V[n, n]$ che coincide con $V[1], \dots, V[n]$.

Counting Sort parallelo

```
1: for  $i \leq n \wedge j \leq n$  par do
2:    $V[i, j] = (M[j] \leq M[i] ? 1 : 0)$ 
3: for  $i = 1$  to  $n$  par do
4:    $SOMMATORIA(V[i, 1], \dots, V[i, n])$ 
5: for  $i = 1$  to  $n$  par do
6:    $M[V[i]] = M[i]$ 
```

Non è mai nella vita EREW, faccio lettura concorrente, ma la scrittura non è concorrente visto che scrivo in ogni cella diversa, quindi CREW

Vediamo le prestazioni:

- fase 1 ho $p = n^2$ e $T(n, n^2) = 4$ per LD LD JE ST
- fase 1 con Wyllie ho $p = \frac{n^2}{\log(n)}$ e $T = \log(n)$
- fase 2 è sommatoria quindi $p = \frac{n^2}{\log(n)}$ perché n moduli sommatoria e $t = \log(n)$
- fase 3 ho $p = n$ e $T = 3$ LD LD ST

Totale quindi è $p = \frac{n^2}{\log(n)}$ e $T = \log(n)$, allora

$$E = \frac{n \log(n)}{\frac{n^2}{\log(n)} \log(n)} = \frac{\log(n)}{n} \rightarrow 0$$

e nemmeno lentamente.

Un algoritmo migliore è bit sort (bitonic sort), che ha efficienza $E = \frac{1}{\log(n)} \rightarrow 0$ ma lentamente. Un altro ottimo è quello di Cole del 1988 ma non lo vedremo, nonostante abbia $E = C \neq 0$.

11.3. Secondo approccio parallelo [bitonic sort]

Prendiamo spunto dal merge sort, che usa la tecnica divide et impera.

MergeSort

```
input  $A[1], \dots, A[n]$ 
1: if  $|A| > 1$ 
2:    $A_s = \text{MergeSort}(A[1], \dots, A[\frac{n}{2}])$ 
3:    $A_d = \text{MergeSort}(A[\frac{n}{2} + 1], \dots, A[n])$ 
```

MergeSort

```
4:  l  $A = \text{Merge}(A_s, A_d)$ 
5:  return  $A$ 
```

La routine merge scorre in sequenza i due array ordinati, confronta i valori correnti e li mette in A . Il tempo peggiore è n .

Il tempo di MergeSort è

$$t(n) = \begin{cases} 0 & \text{se } n = 1 \\ 2t(\frac{n}{2}) + n & \text{altrimenti} \end{cases}$$

Se svogliamo otteniamo

$$t(n) = 2t\left(\frac{n}{2}\right) + n \sim 2\left(2t\left(\frac{n}{4}\right) + n\right) + n \sim \dots \sim 2^k t\left(\frac{n}{2^k}\right) + kn \underset{k=\log(n)}{=} n \cdot 0 + n \log(n) = n \log(n).$$

Sfruttiamo questa cosa: divido continuamente il vettore fino ad arrivare ad un elemento. Qua devo fare il merge, se usassi il parallelo avrei $\log(n) - 1$ passi paralleli essendo un albero. Ma merge non è parallelizzabile e quindi avrei sempre $T \sim n \log(n)$.

Ci chiediamo: quando merge è facile? Supponiamo A_s e A_d ordinati ma gli elementi di A_s tutti minori di A_d : basta concatenarli e basta. Useremo sequenze di numeri particolari, dette **bitoniche**.

Dobbiamo trasformare l'input in quella forma per rendere la vita facile al merge.

Abbiamo bisogno di qualche routine:

- $\text{rev}(A[1], \dots, A[n]) : A[1] = A[n], \dots, A[n] = A[1]$;
- $\text{minmax}(A[1], \dots, A[n])$ che non ho capito.

12. Lezione 12

Due operazioni fondamentali:

- REV per fare il reverse **in parallelo**;
- MINMAX che costruisce gli array A_{\min} e A_{\max} ; divide a metà e prende i valori a distanza $\frac{n}{2}$ e mette il minimo in quello a sx e il massimo a dx; nella prima metà avrò i minimi e nella seconda metà avrò i massimi; ritorniamo poi $A_{\min} A_{\max}$.

Vediamo le procedure.

Reverse

```
1: for  $1 \leq k \leq \frac{n}{2}$  par do
2:   L SWAP( $A[k]$ ,  $A[n - k + 1]$ )
```

Ho $p = \frac{n}{2}$ processori e $t = 4$ per LD LD ST ST.

MinMax

```
1: for  $1 \leq k \leq \frac{n}{2}$  par do
2:   if  $A[k] > A[k + \frac{n}{2}]$ 
3:     L SWAP( $A[k]$ ,  $A[k + \frac{n}{2}]$ )
```

Ho $p = \frac{n}{2}$ processori e $t = 5$ per LD LD ST ST e confronto.

Diamo alcune definizioni di particolari sequenze numeriche:

- unimodale: A è unimodale se e solo se

$$\exists k \mid A[1] > A[2] > \dots > A[k] < A[k+1] < \dots < A[n]$$

oppure

$$A[1] < A[2] < \dots < A[k] > A[k+1] > \dots > A[n]$$

ovvero esiste un valore che mi fa da minimo/massimo e la sequenza è decrescente/crescente poi crescente/decescente. Non è perfettamente ordinato;

- bitonica: A è bitonica se e solo se esiste una permutazione ciclica di A che mi dà una sequenza unimodale, ovvero se

$$\exists j \mid A[j], \dots, A[n], A[1], \dots, A[j-1]$$

è unimodale. In poche parole, scelgo un elemento che va in testa e da lì, ciclicamente, prendo tutto il resto del vettore; una volta fatto ciò, ho una roba unimodale.

Graficamente, una sequenza unimodale ha un picco (massimo o minimo), mentre una sequenza bitonica ha due picchi, un minimo+massimo con i valori della coda-min più piccoli dei valori della coda-max (coda-max > coda-min) oppure un massimo+minimo con coda-max valori più grandi dei valori di coda-min.

Vediamo l'algoritmo per ordinare sequenze bitoniche di Batcher del 1968.

Osserviamo che:

- una sequenza unimodale è anche bitonica, con la permutazione identità;
- i valori di fine vettore devono essere maggiori di inizio vettore (minmax) oppure devono essere minori di inizio vettore (maxmin);

- siano A, B due sequenze ordinate, allora $A \cdot \text{REV}(B)$ è unimodale.

Vediamo delle proprietà ora.

Lemma 12.1: Sia A bitonica, se eseguo minmax su A ottengo:

- A_{\min} e A_{\max} bitonica;
- ogni elemento di A_{\min} è minore di ogni elemento di A_{\max}

Dimostrazione 12.1: Dimostrazione grafica. ■

Ci suggeriscono un DEI:

- minmax suddivide il problema in n elementi su istanze più piccole grazie alla prima parte;
- ordinando A_{\min} e A_{\max} la fusione di due sequenze ordinate avviene per concatenazione grazie alla seconda parte.

BitMerge sequenziale

```

1: input  $A[1], \dots, A[n]$  bitonico
2: minmax su  $A$ 
3: if  $|A| > 2$ 
4:   BitonicSort su  $A_{\min}$ 
5:   BitonicSort su  $A_{\max}$ 
6: return  $A$ 

```

Teorema 12.1: Corretto.

Dimostrazione 12.2: Per induzione su n .

Se $n = 2$ con minmax scambio se disordinati, poi ritorno A , quindi ok, banalmente ordinata da minmax.

Sia $n = 2^k$ corretto, mostriamo per $n = 2^{k+1}$:

- viene calcolato minmax su lunghezza 2^{k+1} che ritorna A_{\min} e A_{\max} di lunghezza $\frac{2^{k+1}}{2} = 2^k$;
- ma su lunghezza 2^k BitMerge ordina perfettamente A_{\min} e A_{\max} per HP;
- quindi A viene ritornato ordinato.

■

Vediamo l'implementazione parallela.

Applichiamo MM con $\frac{n}{2^0}$, poi ..., infine $\frac{n}{2^{i-1}}$. In questo caso avviene una normalissima concatenazione, visto che MM lavora su due elementi.

Algoritmo EREW perché lavoro ogni volta su elementi diversi, no letture concorrenti.

Mi fermo quando $\frac{n}{2^{i-1}} = 2$ quindi $i = \log(n)$. MM costa 5 quindi $T(n) = 5 \log(n)$. Il primo passo richiede $\frac{n}{2}$, il secondo $\frac{n}{4} \cdot 2$, poi ..., quindi sempre $\frac{n}{2}$ processori.

L'equazione di ricorrenza è

$$T(n) = \begin{cases} 5 & \text{se } n = 2 \\ T(\frac{n}{2}) + 5 & \text{altrimenti} \end{cases}$$

non metto costanti alla T perché sono in parallelo. Ottengo lo stesso $T(n) = 5 \log(n)$.

L'efficienza è

$$E = \frac{n \log(n)}{\frac{n}{2} 5 \log(n)} \rightarrow C \neq 0.$$

13. Lezione 13

13.1. BitSort

Vediamo ora BitSort di Batcher del 1968, algoritmo per una qualunque sequenza.

BitSort sequenziale

```
input  $A[1], \dots, A[n]$  generico
1: minmax su  $A$ 
2: if  $|A| > 2$ 
3:   BitSort su  $A_{\min}$ 
4:   BitSort su  $A_{\max}$ 
5:   BitMerge su  $A_{\min} \cdot \text{REV}(A_{\max})$  che è unimodale e quindi bitonica
6: return  $A$ 
```

Teorema 13.1.1: Correttezza.

Dimostrazione 13.1.1: Per induzione su n .

Caso base $n = 2$ facciamo minmax così ho minimo+massimo, viene ritornato il vettore che è ordinato quindi SIUM.

Suppongo vero per $n = 2^k$ e dimostro per $n = 2^{k+1}$.

Sia $|A| = 2^{k+1}$, ma:

- minmax divide A in A_{\min} e A_{\max} di lunghezza 2^k entrambi
- la chiamata ricorsiva (doppia) a BitSort prende 2^k ma per HP induttiva essi sono ordinati
- la chiamata poi a bitmerge avviene con parametri C+D oppure D+C, ma BitMerge è corretto, quindi ordinato, poi viene ritornato.

Vamos tutto corretto. ■

Vediamo l'implementazione parallela (serve immagine).

È un algoritmo PRAM-EREW perché dividiamo sempre l'input ma lavoriamo sempre su dati senza intersezione, accesso e scrittura esclusivi.

Tempo:

- prima fase è come bitmerge, quindi all'ultimo passo ho eseguito $i = \log(n)$;
- seconda passo all'ultimo passo ho $i = \log(n) - 1$ da moltiplicare per il costo di bitmerge che è logaritmico, quindi seconda fase è $T(n) = \log^2(n)$.

Il costo totale è quindi quello della seconda fase, per i processori ho sempre $\frac{n}{2}$ in tutte le fasi.

Possiamo usare anche l'equazione di ricorrenza

$$T(n) = \begin{cases} 5 & \text{se } n = 2 \\ T(\frac{n}{2}) + 5 + 4 + 5 \log(n) & \text{altrimenti} \end{cases}$$

senza costante sul $T(\frac{n}{2})$ perché sono in parallelo. Facendo i conti si ottiene

$$T(n) = \frac{5 \log^2(n) + 23 \log(n) - 18}{2}.$$

L'efficienza è

$$E = \frac{n \log(n)}{\frac{n}{2} 5 \log^2(n)} = \frac{\alpha}{\log(n)} \rightarrow 0$$

molto lentamente. Ci va così piano che si preferisce su istanze molto piccole.

13.2. Osservazioni

Buon algoritmo sequenziale non implica buon algoritmo parallelo: esempio è il MergeSort.

Ma anche buon algoritmo parallelo non implica buon algoritmo sequenziale: esempio è il BitSort.

Infatti, vediamo il tempo sequenziale di BitSort:

- prima BitMerge che vale

$$T_m(n) = \begin{cases} O(1) & \text{se } n = 2 \\ 2T_m(\frac{n}{2}) + O(n) & \text{se } n > 2 \end{cases}$$

quindi ci esce $T_b = O(n \log(n))$;

- vediamo BitSort come

$$T_s(n) = \begin{cases} O(1) & \text{se } n = 2 \\ 2T_s(\frac{n}{2}) + O(n \log(n)) & \text{se } n > 2 \end{cases}$$

che ci dà $T_s = O(n \log^2(n))$

Vediamo come parallelo è buono perché è $O(\log^2(n))$ mentre qua è $O(n \log^2(n))$ che è peggio di MergeSort.

13.3. Tecnica del ciclo euleriano

Vediamo un po' di basi di teoria dei grafi.

Un grafo diretto D è una coppia V, E dove $E \subseteq V^2$. Indichiamo un arco con $(v, e) \in E$. Un cammino è una sequenza di archi e_1, \dots, e_k tale che per ogni coppia di lati consecutivi il nodo pozzo del primo coincide con il nodo sorgente del secondo. Un ciclo è un cammino tale che il nodo pozzo di e_k è il nodo sorgente di e_1 .

Un ciclo è euleriano quando ogni arco in E compare una e una sola volta. Un cammino euleriano è la stessa cosa. Un grafo è euleriano se contiene un ciclo euleriano.

Il problema base è, dato un grafo D , è euleriano?

Diamo notazioni:

- $\forall v \in V$ definiamo $\rho^-(v) = |\{(w, v) \in E\}|$ numero di archi entranti in v ed è detto grado di entrata di v ;
- $\forall v \in V$ definiamo $\rho^+(v) = |\{(v, w) \in E\}|$ numero di archi uscenti da v ed è detto grado di uscita di v

Teorema 13.3.1 (di Eulero (1736)): Un grado D è euleriano se e solo se

$$\forall v \in V \quad \rho^-(v) = \rho^+(v).$$

Vediamo un problema simile: molto simile al ciclo hamiltoniano, ovvero un ciclo è hamiltoniano se e solo se è un ciclo dove ogni vertice in V compare una e una sola volta. D è hamiltoniano se e solo se contiene un ciclo hamiltoniano.

Per euleriano ho un algoritmo efficiente in $O(n^3)$ con $n = |V|$ mentre per hamiltoniano ho un problema NP completo.

Abbiamo una tecnica del ciclo euleriano: viene usata per costruire algoritmi paralleli efficienti che gestiscono strutture dinamiche come alberi binari.

Per trasformare un albero in una tabella con righe nodi e colonne figlio sx dx e il padre etichetto i nodi e poi popolo.

Molti problemi ben noti usano alberi binari:

- ricerca;
- costruzione di dizionari;
- query.

Fondamentale in questi problemi è la navigazione dell'albero (ricerca, manutenzione, modifica, cancellazione, inserimento).

Possiamo operare su queste strutture in parallelo con algoritmi efficienti.

Idea: usiamo delle liste che contengono dei puntatori ai nodi dell'albero, e le possiamo usare bene in parallelo (ad esempio Kogge-Stone per le somme prefisse).

Usiamo un vettore S dei successori dell'albero, gli elementi sono i nodi dell'albero.