Parallel sorting

Algoritmi Avanzati

SAPIENZA
UNIVERSITÀ DI ROMA

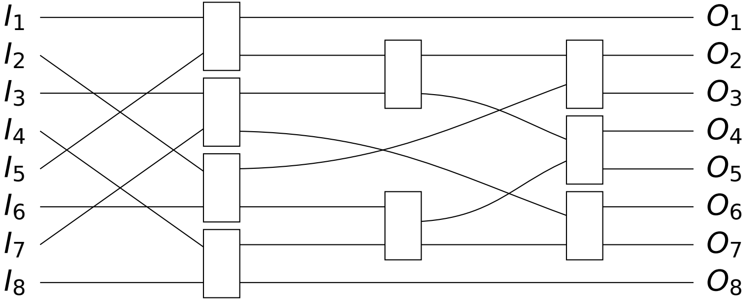
Davide Lo Re

Indice

Se si vuole ottimizzare il mergesort, ci si deve concentrare sul passo di merge

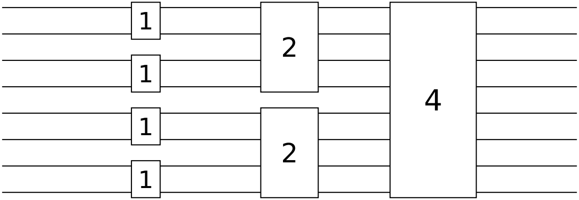
Circuito

Un comparatore è un oggetto che riceve in input due valori, e li ridà ordinati in output



Con un comparatore è abbastanza semplice costruire un circuito in grado di fare il merge di due sequenze ordinate lunghe k

Analisi circuito



Un circuito di merging implementa il mergesort in $\log k$ fasi di merging. La fase i-esima è costituita da un circuito di profondità $\log k - i$, per un totale di $\log^2 k$.
Il costo è quindi $O(n \log^2 n)$, che è subottimale

k-cover

Dati due array A, B sia f una funzione che associa un elemento di B ad un elemento di A
Diciamo che A è un k-cover di B se dati due elementi consecutivi $x, y \in A$,

$$index_B(f(x)) - index_B(f(y)) \leq k$$

Idea chiave

Fondiamo due array in questo modo:

Definizione

Sia o_L, o_R gli array contenenti i soli elementi in posizione dispari di L e di R
Analogamente per e_L, e_R.

Prima fondiamo o_L, o_R in OM e e_L, e_R in EM
Poi fondiamo OM ed EM per ottenere M

Osservazione

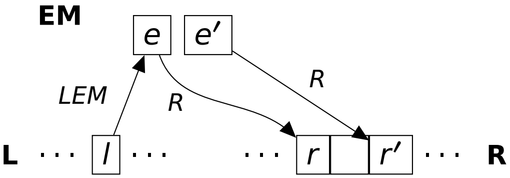
Dato EM, è facile costruire OM

Idea chiave

Supponiamo di aver già ordinato EM e di dover ordinare OM; di conoscere il predecessore in EM di ogni elemento in o_L, o_R (sia LEM, REM) e il predecessore in L, R di ogni elemento in EM (sia L_{ptr}, R_{ptr})

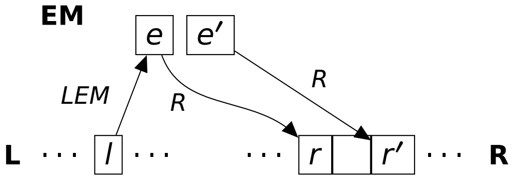
EM è un 2-cover di L, R rispetto alla funzione predecessore

2-cover



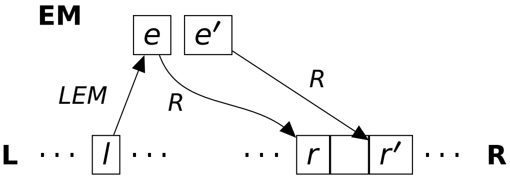
1. $e < l$
2. $e' \geq l$ (altrimenti sarebbe il predecessore)
3. $r < e < l$
4. $r' < e'$
5. $r'' \geq e' \geq l$ (altrimenti sarebbe il predecessore)

2-cover



1. $e < l$
2. $e' \geq l$ (altrimenti sarebbe il predecessore)
3. $r < e < l$
4. $r' < e'$
5. $r'' \geq e' \geq l$ (altrimenti sarebbe il predecessore)
6. Ogni elemento dispari in R è il predecessore di un elemento in EM

2-cover



1. $e < l$
2. $e' \geq l$ (altrimenti sarebbe il predecessore)
3. $r < e < l$
4. $r' < e'$
5. $r'' \geq e' \geq l$ (altrimenti sarebbe il predecessore)
6. Ogni elemento dispari in R è il predecessore di un elemento in EM

$$|(r, r']| \leq 2$$

Questo vuol dire che possiamo fare un solo confronto per stabilire la posizione di e in OM.

Bisogna però costruire queste strutture dati in modo efficiente

Fase 1.1: Predecessori da *EM* in *L*, *R*

R	2	4	6	8	13	15	16	20
EM	3	4	8	10	14	$\boxed{15}$	19	20
L	1	3	9	10	12	14	18	19

Sia $e \in EM, R$. Poiché e viene da R, conosciamo il suo predecessore in R. (viceversa per L)

Fase 1.1: Predecessori da *EM* in *L*, *R*

R	2	4	6	8	13	15	16	20
EM	3	4	8	10	14	$\boxed{15}$	19	20
L	1	3	9	10	12	14	18	19

Sia $e \in EM, R$. Poiché e viene da R, conosciamo il suo predecessore in R.
Posizione: $2 \cdot ie - R_{Ptr}(e) - 1$

(“*numero di elementi da cui è stato superato*”)

Fase 1.1: Predecessori da *EM* in *L*, *R*

R	2	4	6	8	13	15	16	20
EM	3	4	8	10	14	$\boxed{15}$	19	20
L	1	3	9	10	12	14	18	19

Sia $e \in EM, R$. Poiché e viene da R, conosciamo il suo predecessore in R.
Posizione: $2 \cdot ie - R_{Ptr}(e) - 1$
Confrontando con r_2 stabiliamo il predecessore

Fase 1.1: Predecessori da *EM* in *L*, *R*

R	2	4	6	8	13	15	16	20
EM	3	4	8	10	14	$\boxed{15}$	19	20
L	1	3	9	10	12	14	18	19

Sia $e \in EM, R$. Poiché e viene da R, conosciamo il suo predecessore in R.
Posizione: $2 \cdot ie - R_{Ptr}(e) - 1$
Confrontando con r_2 stabiliamo il predecessore
$\#\text{proc} = |EM| \Rightarrow$ tempo parallelo costante

Fase 1.2: Predecessori da L, R in EM

EM	$\cdots$	e	e'	e''	$\cdots$ $\cdots$	e	e'	e''		$\cdots$
		$\downarrow$	$\downarrow$			$\downarrow$	$\searrow$			
L	$\cdots$	$L(e)$	$L(e')$	$L(e'')$	$\cdots$ $\cdots$	$L(e)$	m	$L(e')$	$L(e'')$	$\cdots$

Siano e, e' consecutivi in EM; allora $(L_{Ptr}(e), L_{Ptr}(e')]$ comprende al massimo 2 elementi.

Perché ogni elemento dispari in L è il predecessore di un elemento in EM

Fase 1.2: Predecessori da L, R in EM

EM $\cdots \quad e \quad e' \quad e'' \quad \cdots \quad \cdots \quad e \quad e' \quad e'' \quad \cdots$

L $\cdots \quad L(e) \quad L(e') \quad L(e'') \quad \cdots \quad \cdots \quad L(e) \quad m \quad L(e') \quad L(e'') \quad \cdots$

Siano e, e' consecutivi in EM; allora $(L_{Ptr}(e), L_{Ptr}(e')]$ comprende al massimo 2 elementi.

In particolare, al più uno di indice dispari. Sia l.
Impostiamo $LEM_{Ptr}(l) = e$.
In questo modo *ogni* elemento dispari in L viene impostato

Fase 1.2: Predecessori da L, R in EM

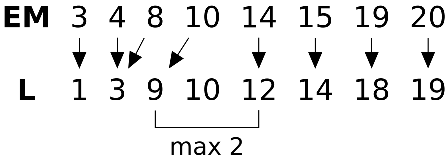
$$\begin{array}{lccccccccccc} \mathbf{EM} & \cdots & e & e' & e'' & \cdots & \cdots & e & & e' & e'' & \cdots \\ & & \downarrow & \downarrow & & & & \downarrow & & \searrow & & \\ \mathbf{L} & \cdots & L(e) & L(e') & L(e'') & \cdots & \cdots & L(e) & m & L(e') & L(e'') & \cdots \end{array}$$

Siano e, e' consecutivi in EM; allora $(L_{Ptr}(e), L_{Ptr}(e')]$ comprende al massimo 2 elementi.

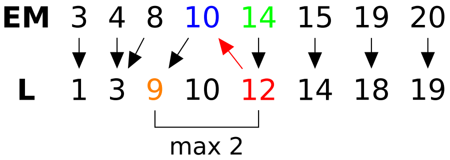
In particolare, al più uno di indice dispari. Sia l.
Impostiamo $LEM_{Ptr}(l) = e$.
In questo modo *ogni* elemento dispari in L viene impostato

Come prima, se abbiamo un processore per ogni elemento in EM possiamo eseguire in tempo parallelo costante

Fase 1.2: Esempio



Fase 1.2: Esempio



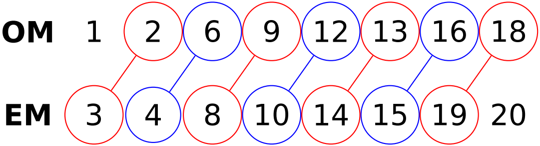
Fase 1.2: Esempio

EM	3	4	8	10	14	15	19	20
L	1	3	9	10	12	14	18	19

Merge

Sfruttando l'idea vista all'inizio possiamo conoscere la posizione di ogni elemento in *OM* in tempo parallelo costante

Se abbiamo *OM* ed *EM* possiamo fonderli in modo efficiente



In "orizzontale" e "verticale" le minorazioni sono ovvie; l'unica indecisione è in "obliquo".

In tempo parallelo costante possiamo costruire *M*

Analisi dei tempi di esecuzione

- Fase 1
  - Fase 1.1: L_{Ptr}, R_{Ptr}
  - Fase 1.2: LEM_{Ptr}, REM_{Ptr}
- Fase 2

Analisi dei tempi di esecuzione

- Fase 1
  - Fase 1.1: L_{Ptr}, R_{Ptr}
  - Fase 1.2: LEM_{Ptr}, REM_{Ptr}
- Fase 2

-
  - $O(1)$
  -
-

Analisi dei tempi di esecuzione

- Fase 1
  - Fase 1.1: L_{Ptr}, R_{Ptr}
  - Fase 1.2: LEM_{Ptr}, REM_{Ptr}
- Fase 2

- $O(1)$
  - $O(1)$
  - $O(1)$
-

Analisi dei tempi di esecuzione

- Fase 1
  - Fase 1.1: L_{Ptr}, R_{Ptr}
  - Fase 1.2: LEM_{Ptr}, REM_{Ptr}
- Fase 2

- $O(1)$
  - $O(1)$
  - $O(1)$
- $O(1)$

Analisi del merge

Dato *EM*, il merging impiega tempo parallelo costante; *EM* va però ottenuto ricorsivamente.

$$T(n) = O(1) + T(n/2)$$
$$T(2) = 1$$

$$T(n) = \log n$$

Algoritmo di ordinamento

Si procede come nel mergesort, usando come *merger* l'algoritmo appena descritto.
Abbiamo quindi $\Theta(\log n)$ fasi, tali che la fase i impiega tempo $\log n - i$

$$O(\log^2 n)$$

Costo

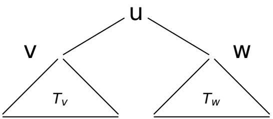
Per il teorema di Brent, possiamo eseguire l'algoritmo su $\frac{n}{\log n}$ processori

Costo: $\log^2 n \cdot \frac{n}{\log n} = n \log n$, che è l'ottimo

Algoritmo di Cole

- Mantiene esplicitamente l'albero della ricorsione
- Lavora in parallelo su *più* livelli, utilizzando dati parziali
- Fa uso di una proprietà 3-cover

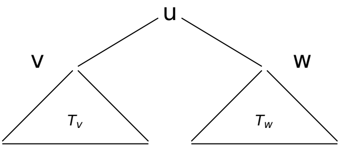
Strutture dati



- Vettore O (Origin), dove $O_t(u)[e] = L$ se e in $M_t(u)$ viene dal sottoalbero T_v
- Vettore E (Extract), $E_{t+1}(u)$ contiene gli elementi di $M_t(u)$ che andranno in $M_{t+1}(padre(u))$
- Vettore B_t (Before), per ogni elemento $e \in M_t(u)$ proveniente da T_v , punta al predecessore in $M_t(u)$ di e proveniente dal sottoalbero T_w
- Vettore A_t (After), per ogni elemento $e \in M_t(u)$ proveniente da T_v, punta al successore in $M_t(u)$ di e proveniente dal sottoalbero T_w

NOTA: Le stesse definizioni di A_t e B_t, simmetricamente, si applicano se il nodo e proviene da T_w.

Altre strutture



- Vettore *R_Ptr*, per ogni elemento $e \in M_t(u)$, se e proviene dal sottoalbero T_w, punta al predecessore di e in $E_{t+1}(v)$
- Vettore *L_Ptr*, per ogni elemento $e \in M_t(u)$, se e proviene dal sottoalbero T_v, punta al predecessore di e in $E_{t+1}(w)$
- Vettore *P_Ptr*, per ogni elemento $ev \in E_{t+1}(v)$ punta al predecessore di ev in $M_t(u)$