1 Algoritmi paralleli a memoria condivisa

1. Sommatoria:
   (a) Definizione del problema sommatoria

> **Solution:** Dato in input un vettore A, vorremmo avere in output
>
> $$A[\,n\,] = \sum_{i=1}^{n} A[\,i\,]$$

   (b) Dare una soluzione sequenziale indicando la complessità in tempo

> **Solution:** Una soluzione sequenziale consiste banalmente in un singolo ciclo che somma i valori in una variabile accumulatore
>
> $acc \leftarrow 0$
> **for** $i \leftarrow 1$ **to** n **do**
> $\quad acc \leftarrow acc + A[\,i\,]$
> **end for**
> $A[n] \leftarrow acc$
>
> questo algoritmo impiega tempo lineare $(n-1)$.

   (c) Dare una soluzione EREW su PRAM con relativo codice

> **Solution:** si sommano i valori a distanza 2^{i-1}, per i da 1 a $\log n$. quindi
>
> **for** $i \leftarrow 1$ **to** $\log n$ **do**
> $\quad$ **for** $k \leftarrow 1$ **to** $\frac{n}{2^i}$ **par do**
> $\quad\quad A[\,2^i k\,] \leftarrow A[\,2^i k\,] + A[\,2^i k - 2^{i-1}\,]$
> $\quad$ **end par do**
> **end for**
>
> L'algoritmo è chiaramente EREW:
>
> - le celle scritte ad ogni iterazioni sono banalmente diverse;
> - le celle lette allo stesso passo sono diverse, supponiamo infatti che non lo siano, ci sono 3 casi:
>   - per due k e k' vale che $2^i k$ sia uguale a $2^i k'$, che è banalmente falso;
>   - per due k e k' vale che $2^i k - 2^{i-1}$ sia uguale a $2^i k' - 2^{i-1}$, che è banalmente falso;
>   - per due k e k' vale che $2^i k$ sia uguale a $2^i k' - 2^{i-1}$, ma, raggruppando per 2^{i-1} otteniamo
>
> $$2k = 2k' - 1$$

che è falso visto che un numero è necessariamente pari e l'altro dispari.

(d) Valutare la complessità dell'algoritmo proposto e l'efficienza

Solution: L'algoritmo proposto utilizza

$$p(n) = \frac{n}{2}$$
$$T(n, p(n)) = \log n$$

ottenendo come efficienza

$$E(n, p(n)) = \frac{n}{n \log n} \rightsquigarrow 0$$

Proviamo quindi ad utilizzare Wyllie per ridurre il numero di processori. Scegliamo un numero di processori $p(n) = p$: ogni processore svolge $\frac{n}{p}$ somme sequenziali e poi si applica l'algoritmo di sopra, in modo da ottenere

$$T(n, p) = \frac{n}{p} + \log p$$

e

$$E(n, p) = \frac{n}{p\left(\frac{n}{p} + \log p\right)} = \frac{n}{n + p \log p}$$

scegliamo ora $p = \frac{n}{\log n}$, in modo da ottenere

$$E\left(n, \frac{n}{\log n}\right) = \frac{n}{n + n} \rightsquigarrow c \neq 0$$

2. Tecnica del cammino euleriano:

(a) Come si espandono i vertici

Solution: Ogni vertice viene espanso in 3: sinistra, centro, destra

$$v \in V \rightarrow (v, s), (v, c), (v, d)$$

(b) Costruzione lista successore

Solution: Per ogni vertice definiamo il suo successore:

- se v è una foglia, allora $S[\, v_s \,] = v_c$;

- se v è una foglia, allora $S[\,v_c\,] = v_d$;
- se v non è una foglia, allora $S[\,v_s\,] = \text{sin}(v)_s$;
- se v non è una foglia, allora $S[\,v_c\,] = \text{des}(v)_s$;
- se v è il figlio di sinistra del padre, allora $S[\,v_d\,] = \text{pad}(v)_c$;
- se v è il figlio di destra del padre, allora $S[\,v_d\,] = \text{pad}(v)_d$.

3. Sequenze unimodali, bitoniche e sorting

   (a) Definizione di sequenze unimodali e bitoniche

Solution: Una sequenza unimodale è una sequenza che contiene un "picco", cioè una sequenza di numeri per cui esiste un indice k tale che

$$A[\,1\,] \leq \ldots \leq A[\,k\,] \geq \cdots \geq A[\,n\,]$$

oppure

$$A[\,1\,] \geq \ldots \geq A[\,k\,] \leq \cdots \leq A[\,n\,]$$

Una sequenza bitonica è una sequenza dal quale – attraverso una permutazione ciclica – si può ottenere una sequenza unimodale. Cioè per cui esiste un i per cui

$$A[\,i\,], A[\,i+1\,], \ldots, A[\,n\,], \ldots, A[\,1\,], \ldots, A[\,i-1\,]$$

è unimodale.

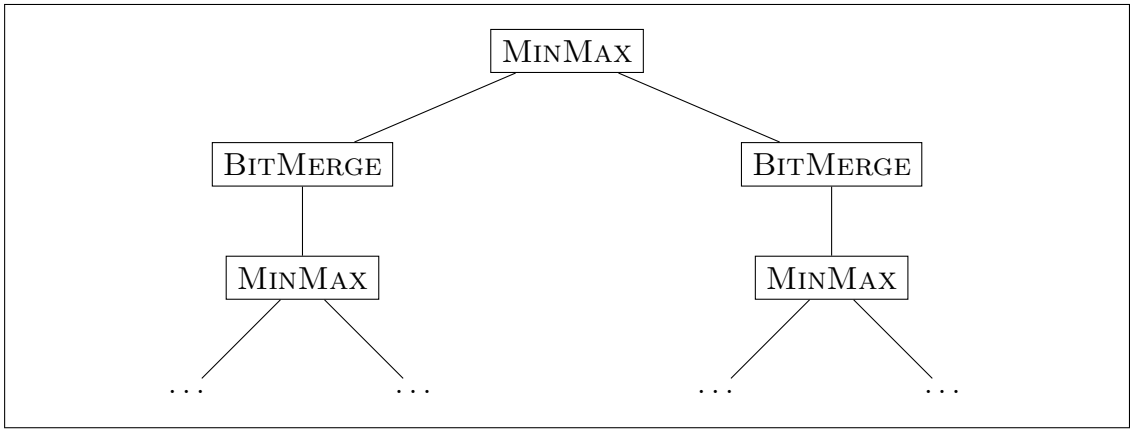
   (b) Implementazione della routine BitMerge di ordinamento di sequenze bitoniche

Solution:

$A_{\min}, A_{\max} \leftarrow \textsc{MinMax}(A)$
if $|A| > 2$ **then**
    $\textsc{BitMerge}(A_{\min})$
    $\textsc{BitMerge}(A_{\max})$
end if
return A

   (c) Esempio di dinamica ricorsiva di BitMerge, su una piccola istanza di input, usando un grafico

Solution:



(d) Valutazione dell'algoritmo, tempo e processori

Solution: Associamo ad ogni chiamata ricorsiva un processore, per cui, al massimo, serviranno $\frac{n}{2}$ processori. L'algoritmo genera la seguente equazione di ricorrenza

$$T(n) = \begin{cases} 1 & \text{se } n = 2 \\ T\left(\frac{n}{2}\right) + O(1) & \text{altrimenti} \end{cases}$$

da cui otteniamo un tempo di

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

da qui otteniamo come efficienza

$$E(n, p(n)) = \frac{n \log n}{n \log n} \rightsquigarrow c \neq 0$$

(e) Correttezza di BitMerge

Solution:
Procediamo per induzione sulla dimensione dell'array:

- Base: per $n = 2$, l'array è banalmente ordinato dalla chiamata a MinMax;
- Passo: supponiamo che l'algoritmo ordini correttamente sequenze bitoniche di dimensione 2^k, mostriamo che ordina correttamente anche sequenze bitoniche di 2^{k+1}. Vale che chiamare MinMax ritorna due sequenze bitoniche di dimensione 2^k, e che tutti gli elementi della sequenza bitonica $A_{\min}$ sono minori di tutti gli elementi della sequenza bitonica $A_{\max}$. Infine, chiamare BitMerge su $A_{\min}$ e $A_{\max}$ le ordina correttamente per ipotesi induttiva, concludendo la dimostrazione.

4. Rispondi alle seguenti domande

   (a) Definizione di "Speed-Up" ed "Efficienza"

Solution: Lo Speed-Up è il rapporto tra il tempo dell'algoritmo sequenziale ottimo e il tempo di un algoritmo parallelo. Se questo tende a 0 vuol dire che l'algoritmo parallelo è troppo inefficiente.

Il problema principale dello Speed-Up è che questo non tiene conto del numero di processori utilizzati. Per questo esiste l'efficienza, che corrisponde a il rapporto tra lo Speed-Up e il numero di processori, o

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

dove $T^*(n, 1)$ indica il tempo migliore sequenziale. L'efficienza è un valore compreso tra 0 e 1.

Mostriamo che l'efficienza è al massimo 1:

$$T(n, 1) \geq \hat{T}(n, 1) = \sum_{i=0}^{k(n)} p(n)t_i(n) = p(n) \sum_{i=0}^{k(n)} t_i(n) = p(n)T(n, p(n))$$

quindi $T(n, 1) \geq p(n)T(n, p(n))$, perciò il loro rapporto è per forza ≤ 1.

   (b) Dare uno schema di un algoritmo parallelo A che usa p processori. Modificare A per $\frac{p}{k}$ processori e indicare la relazione tra i tempi paralleli dell'algoritmo appena modificato e l'algoritmo A.

Solution: Un algoritmo parallelo è una matrice di istruzioni

	P_1	$\dots$	P_n
1:	i_{11}	$\dots$	i_{1n}
$\vdots$	$\vdots$	$\ddots$	$\dots$
m:	i_{m1}	$\dots$	i_{mn}

Applicare Wyllie significa far svolgere a $\frac{p}{k}$ processori k operazioni sequenziali ognuno. In questo modo il tempo si modifica nel seguente modo

$$T\left(n, \frac{p}{k}\right) \leq \sum_{i=0}^{k(n)} kt_i(n) = k \sum_{i=0}^{k(n)} t_i(n) = kT(n, p)$$

   (c) Mostra cosa succede al parametro efficienza se si diminuiscono i processori da p a p/k

Solution: Utilizzando la disequazione di sopra otteniamo che

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

quindi si può provare a utilizzare Wyllie per aumentare l'efficienza quando questa tende a 0.

5. Somme prefisse

   (a) Esempio spiegazione e algoritmo di Kogge-stone

Solution: Il problema somme prefisse prende in input un vettore A di dimensione n e ritorna in output il vettore tale che

$$\forall k \in n, A[\,k\,] = \sum_{i=1}^{k} A[\,k\,]$$

Per questo problema esiste una soluzione efficiente chiamata metodo di Kogge-Stone, basato sul pointer doubling. Le celle del vettore sono legate a distanza 2^i, e ogni processore si occupa di fare la somma di uno di questi legami. L'algoritmo codifica i legami attraverso un vettore S, che – per ogni cella – contiene il suo successore o 0 se questo è fuori range.

Inizialmente il vettore S è inizializzato nel seguente modo

$$S[\,k\,] = \begin{cases} 0 & \text{se k = n} \\ k+1 & \text{altrimenti} \end{cases}$$

for $i \leftarrow 1$ **to** $\log n$ **do**
    for $k \leftarrow 1$ **to** $n - 2^{i-1}$ **par do**
        $A[\,S[\,k\,]\,] \leftarrow A[\,k\,] + A[\,S[\,k\,]\,]$
        if $S[\,k\,] \neq 0$ **then**
            $S[\,k\,] \leftarrow S[\,S[\,k\,]\,]$
        end if
    end par do
end for

   (b) Prestazioni

Solution: L'algoritmo proposto utilizza

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

ottenendo come efficienza

$$E(n, p(n)) = \frac{n}{n \log n} \rightsquigarrow 0(\text{lentamente})$$

Possiamo quindi usare Wyllie:

$$p(n) = O\left(\frac{n}{\log n}\right)$$
$$T(n, p(n)) = O(\log n)$$

E di conseguenza:

$$E(n, p(n)) = \frac{n}{\frac{n}{\log n} \cdot \log n} \rightsquigarrow c \neq 0$$

6. Ricerca di un elemento
   (a) Definizione del problema

Solution: Dato un valore α e un vettore M di dimensione n in input, in $M[\,n\,]$ ci dovrà essere 1 se α è presente in M altrimenti -1.

   (b) Pseudocodice CRCW, CREW, EREW con le prestazioni in tempo e processori per ciascuno.

Solution: Possiamo dare una prima versione CRCW

```
F ← −1
for i ← 1 to n par do
    if M[i] = α then
        F ← 1
    end if
end par do
M[n] ← F
```

Dove il flag F serve per non sovrascrivere $M[\,n\,]$.

Mostriamo ora un algoritmo CREW che fa utilizzo di Max (definito come Sommatoria)

```
for i ← 1 to n par do
    if M[i] = α then
        M[i] ← 1
    else
        M[i] ← 0
    end if
```

end par do
MAX(M)

Qui possiamo usare direttamente Wyllie per ridurre il numero di processori a

$$\begin{aligned} p(n) &= \frac{n}{\log n} \\ T(n, p(n)) &= \log n \\ E(n, p(n)) &= \frac{n}{p(n)T(n, p(n))} \rightsquigarrow c \neq 0 \end{aligned}$$

Infine possiamo utilizzare BROADCAST, così definito

$A[\,1\,] \leftarrow \alpha$
for $i \leftarrow 1$ **to** $\log n$ **do**
  for $k \leftarrow 1$ **to** $2^i - 1$ **par do**
    $A[\,k + 2^{i-1}\,] \leftarrow A[\,k\,]$
  end par do
end for

per diffondere il dato α tra tutti i processori ed eliminare le letture concorrenti.
In questo modo otteniamo il seguente algoritmo

BROADCAST(A, α)
for $i \leftarrow 1$ **to** n **par do**
  if $M[\,i\,] = A[\,i\,]$ **then**
    $M[\,i\,] \leftarrow 1$
  else
    $M[\,i\,] \leftarrow 0$
  end if
end par do
MAX(M)

che è efficiente ed EREW.

2 Algoritmi paralleli a memoria distribuita

1. Nell'ambito degli array lineari presentare quanto segue:
   (a) Definizione del problema MAX

Solution: Dati dei valori tali che

$$\text{cont}(P_1) = a_1, \ldots, \text{cont}(P_n) = a_n$$

vogliamo che

$$\text{cont}(P_n) = \max\{\text{cont}(P_k) \mid k \in 1, \ldots, n\}$$

su array lineari sappiamo che il diametro è $n-1$ e sappiamo che la complessità del MAX deve essere per forza almeno d.

(b) Dare uno schema di un algoritmo parallelo per MAX

Solution: Similmente al problema sommatoria per i processori a memoria condivisa, facciamo il massimo di elementi a distanza 2^{i-1} per i da 1 a $\log n$, e poniamo il risultato nella cella di indice maggiore. A differenza del problema sommatoria per i processori a memoria condivisa, nella memoria distribuita è necessario anche gestire l'invio dei dati, che non è più necessariamente costante.

for $i \leftarrow 1$ **to** $\log n$ **do**
  for $k \leftarrow 1$ **to** $\frac{n}{2^i}$ **par do**
    SEND$(A[\,2^i k - 2^{i-1}\,], A[\,2^i k\,])$
  end par do
  for $k \leftarrow 1$ **to** $\frac{n}{2^i}$ **par do**
    $A[\,2^i k\,] \leftarrow$ MAX$(A[\,2^i k\,], A[\,2^i k - 2^{i-1}\,])$
  end par do
end for

(c) Dare il numero di processori e tempo parallelo richiesto dall'algoritmo presentato

Solution: Il numero di processori è $p(n) = \frac{n}{2}$. Il tempo richiesto dall'algoritmo è dato dalla somma dei tempi delle due fasi ripetute $\log n$ volte

$$\begin{aligned} T(n, p(n)) &= \sum_{i=1}^{\log n} 2 \cdot 2^{i-1} + \sum_{i=1}^{\log n} 2 \\ &= 2\frac{1 - 2^{\log n}}{1 - 2} + 2\log n \\ &= 2n - 2 + 2\log n \\ &= O(n) \end{aligned}$$

per cui otteniamo

$$E(n, p(n)) = \frac{n}{n^2} \rightsquigarrow 0$$

(d) Riduzione dei processori per migliorare il parametro efficienza

Solution: Possiamo provare quindi a ridurre il numero di processori a p, svol-

gendo alcuni dei MAX sequenzialmente:

$$p(n) = p$$
$$T(n, p(n)) = \frac{n}{p} + p$$

da cui si ottiene come efficienza

$$E(n, p(n)) = \frac{n}{p\left(\frac{n}{p} + p\right)} = \frac{n}{n + p^2} \xrightarrow{p=\sqrt{n}} \frac{n}{n+n} \rightsquigarrow c \neq 0$$

2. Nell'ambito delle architetture parallele a memoria distribuita:
   (a) Descrivere l'architettura mesh e fornire parametri di rete

Solution: L'architettura mesh è formata da una matrice di n processori, quindi di dimensione $\sqrt{n} \times \sqrt{n}$, collegati nelle direzioni cardinali. Quindi i parametri sono

$$\gamma = 4$$
$$\delta = 2\sqrt{n}$$
$$\beta = \sqrt{n}$$

Questa architettura è quella più comune per i supercomputer.

   (b) Definire il problema Max e sua soluzione su Mesh, processori e tempo

Solution: Dati dei valori caricati sulla mesh a serpentina (da sinistra a destra sulle righe dispari e da destra a sinistra sulle righe pari), vogliamo che

$$\text{cont}(P_n) = \max\{\text{cont}(P_k) \mid k \in 1, \ldots, n\}$$

Possiamo fornire un algoritmo che esegue il massimo su ogni riga considerandole come array lineari, e che infine svolge il massimo sulla riga finale. Questo algoritmo necessita

$$p(n) = n$$
$$T(n, p(n)) = \sqrt{n}$$

quindi non è efficiente.

   (c) Riduzione dei processori per una soluzione ottimale di Max, processori e tempo

Solution: Proviamo ancora a ridurre i processori a un certo valore p. Ogni processore svolgerà $\frac{n}{p}$ passi di MAX sequenziale e poi svolgerà l'algoritmo di sopra. In questo modo otteniamo che

$$p(n) = p$$
$$T(n, p(n)) = \frac{n}{p} + \sqrt{p}$$

e come efficienza

$$E(n, p(n)) = \frac{n}{p\left(\frac{n}{p} + \sqrt{p}\right)} = \frac{n}{n + p\sqrt{p}} \xrightarrow{p=n^{\frac{2}{3}}} \frac{n}{n+n} \rightsquigarrow c \neq 0$$

3. Definire il modello delle architetture parallele a memoria distribuita: mesh

   (a) Definire il problema Ordinamento su mesh

   Solution: Dati i valori caricati sulla mesh a serpentina, vogliamo che la mesh contenga una permutazione ordinata degli stessi valori sempre letta a serpentina.

   (b) Descrivere l'algoritmo LS3SORT

   Solution: L'algoritmo è un algoritmo di tipo divide-et-impera

```
function LS3Sort(M)
    if |M| = 1 then
        return M
    else
        for k ∈ {1, 2, 3, 4} par do
            LS3Sort(M_k)
        end par do
        LS3Merge(M_1, M_2, M_3, M_4)
        return M
    end if
end function
```

   dove ogni chiamata ricorsiva opera sui 4 quadranti risultato della divisone della matrice in componenti di dimensione $\frac{\sqrt{n}}{2} \times \frac{\sqrt{n}}{2}$.

   (c) Descrivere la procedura LS3MERGE e la sua valutazione in tempo

   Solution: La routine LS3MERGE si occupa di ricomporre la matrice a partire dalle 4 sottomatrici ordinate. Fa questo attraverso 3 passi:

- per ogni riga della matrice esegue SHUFFLE come se fosse un array lineare, quindi $\sqrt{n}$ passi;
- per ogni coppia di colonne esegue ODDEVEN leggendole a serpentina, in tempo $2\sqrt{n}$ passi;
- infine esegue i primi $2\sqrt{n}$ passi di ODDEVEN sull'intera matrice.

si può vedere facilmente che LS3MERGE impiega $O(\sqrt{n})$ passi. Ora possiamo scrivere l'equazione di ricorrenza dell'algoritmo

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T\left(\frac{n}{4}\right) + O(\sqrt{n}) & \text{altrimenti} \end{cases}$$

Quindi si ottiene

$$T(n, p(n)) = O(\sqrt{n})$$

che rimane non efficiente.

4. Nell'ambito delle architetture parallele a memoria distribuita:
   (a) Definire il problema ordinamento e la sorting network ODDEVEN,

Solution: Dati n valori caricati sui processori, si vuole che alla fine il contenuto dei processori sia una permutazione ordinata dei valori originali.

Per risolvere il problema dell'ordinamento su array lineari si utilizza una sorting network chiamata ODDEVEN. Questa per n volte confronta – e nel caso scambia – coppie di valori, alternando confronti di coppie pari e dispari.

   (b) Descrivere l'implementazione di ODDEVEN su array lineari,

Solution:

for $i \leftarrow 1$ **to** n **do**
  for $k \leftarrow 0$ **to** $\frac{n}{2} - 1$ **par do**
    MINMAX$(A[\,2k + \langle i \rangle_2\,], A[\,2k + \langle i \rangle_2 + 1\,])$
  end par do
end for

quindi sono necessari

$$p(n) = \frac{n}{2}$$
$$T(n, p(n)) = n$$

per cui l'algoritmo non è efficiente.

(c) Descrivere l'algoritmo MergeSplit su array lineari

Solution: Una versione alternativa di OddEven è l'algoritmo MergeSplit. In questo ci sono p processori, contenenti $\frac{n}{p}$ elementi ciascuno, i quali vengono inizialmente ordinati sequenzialmente in $\frac{n}{p}\log\frac{n}{p}$ passi. Successivamente, per p volte, si alternano due fasi:

- i processori pari inviano i loro $\frac{n}{p}$ dati al processore successivo, questo fa il merge con i propri dati in tempo $2\frac{n}{p}$ e ritorna i $\frac{n}{p}$ dati minori al mittente.
- lo stesso avviene coi processori dispari, a fasi alterne.

Per questo nuovo algoritmo

$$p(n) = p$$
$$T(n, p(n)) = \frac{n}{p}\log\frac{n}{p} + p\left(2\frac{n}{p} + 2\frac{n}{p}\right) = \frac{n}{p}\log\frac{n}{p} + n$$

da cui otteniamo una efficienza di

$$E(n, p(n)) = \frac{n\log n}{p\left(\frac{n}{p}\log\frac{n}{p} + n\right)} = \frac{n\log n}{n\log n - n\log p + pn}$$

scegliendo $p = \log n$ otteniamo infine

$$E(n, \log n) = \frac{n\log n}{n\log n - n\log^2 n + n\log n} \rightsquigarrow c \neq 0$$

Da notare come il tempo sia rimasto $O(n)$ perché non abbiamo modificato l'ampiezza di bisezione ma solo il diametro.

5. Reti di confrontatori:

(a) Cos'è una rete di confrontatori

Solution: Un confrontatore è un componente che prende due input ed ha due output: confronta i due valori in input e manda il massimo in un output e il minimo nell'altro. Di solito i confrontatori sono composti in reti, concatenando gli output di un confrontatore nell'input di un altro.
Una rete di confrontatori può essere vista anche come una funzione

$$R(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$$

dove $(y_1, \ldots, y_n)$ è una permutazione di $(x_1, \ldots, x_n)$.

(b) Cosa è sorting network e principio 0-1

Solution: Nello specifico, se $(y_1, \ldots, y_n)$ è una permutazione ordinata di $(x_1, \ldots, x_n)$, allora R è detta sorting network.
Per dimostrare che una certa sorting network sia corretta si può utilizzare il principio 0-1: una sorting network è corretta se ordina correttamente ogni $(x_1, \ldots, x_n) \in \{0,1\}^n$. Cioè per dimostrare la correttezza di una sorting network la si può valutare solo su input binari.

3 Algoritmi distribuiti

1. Shortest path:
   (a) Introdurre il problema ShortestPath

   Solution: Il problema consiste nel, data una rete con restrizioni IR, fare in modo che ogni entità sia a conoscenza del cammino di costo minimo verso ogni altra entità. Ci sono diverse versioni in base al numero di messaggi che si è disposti a inviare e alla quantità di memoria che si è disposti a usare per ogni processore.

   (b) Cos'è la Full Routing Table

   Solution: La Full Routing Table (FRT) è una tabella presente in ogni entità che, per ogni altra entità, contiene il costo e il cammino minimo. Per tutti gli algoritmi di ShortestPath lo stato contenuto all'interno dei ogni entità dovrà convergere verso una FRT.

   (c) Il protocollo Iterated-Construction

   Solution: Ogni entità x mantiene un vettore V_x, chiamato distance vector, che inizialmente contiene solo le distanze dei suoi vicini o infinito per le entità non direttamente collegate. La particolarità del distance vector è che non contiene il percorso completo alla destinazione, ma piuttosto contiene solo il prossimo nodo.
   Ogni entità manda il proprio distance vector ai vicini, e, alla ricezione, ogni entità aggiorna il proprio distance vector

   $$V_x[\,z\,] = \min_{y \in N(x)} \{\theta(x,y) + V_y[\,z\,]\}$$

   dove $\theta(x,y)$ è il costo del link tra x e y. Se c'è un percorso migliore di quelli già conosciuti aggiorna la propria FRT.

Questo protocollo ha il vantaggio che ogni processore deve mantenere solo $O(n)$ informazioni, al contrario delle $O(n^2)$ del protocollo Gossiping. Inoltre il numero di messaggi è

$$M[\text{IteratedConstruction}] = 2m \cdot n \cdot (n-1)$$

dove

- $2m$ è il numero di distance vector inviati;
- n è il tempo per inviare un distance vector;
- $(n-1)$ è il numero di iterazioni necessarie affinché ogni distance vector arrivi a tutti (alla j-esima iterazione ho informazioni sui nodi lontani j e la massima distanza è $n-1$).

mentre il tempo è

$$T[\text{IteratedConstruction}] = (n-1) \cdot \tau(n)$$

dove

- $(n-1)$ è ancora il numero di iterazioni necessarie affinché il distance vector di tutti arrivi a tutti;
- $\tau(n)$ è il tempo necessario ad inviare il distance vector, $O(1)$ se il sistema permette messaggi lunghi, $O(n)$ altrimenti.

(d) Protocollo Gossiping

Solution: Ogni nodo si costruisce l'intera matrice di adiacenza $MAP(G)$ al suo interno, e poi utilizza questa per calcolarsi la Full Routing Table. La problematica principale di questo algoritmo è che ogni nodo deve contenere l'intero grafo, che occupa $O(n^2)$.
In sostanza l'algoritmo si articola nel seguente modo:

1. si costruisce lo spanning tree su G;
2. ogni entità fa il broadcast delle sue informazioni ai vicini sullo spanning tree;
3. ogni entità che riceve il messaggio di un'altra aggiorna la propria matrice di adiacenza $MAP(G)$.

Il numero di messaggi corrisponde all'incirca a n broadcast, quindi

$$M[\text{Gossipping}] = 2mn \approx O(n^2)$$

se il grafo è sparso.
Il tempo invece è difficile da calcolare, ma dipende dal diametro.

2. Broadcast:
   (a) Presentazione BROADCAST

Solution: Il BROADCAST è il problema di diffondere un'informazione sull'intera rete. Formalmente questo è definito nel seguente modo:

$$\begin{aligned} P_{init} &: \forall x \in \mathcal{E}, (\text{stato}(x) = iniziatore \wedge \text{valore}(x) = I) \vee (\text{stato}(x) = inattivo) \\ P_{final} &: \forall x \in \mathcal{E}, \text{valore}(x) = I \\ R &: RI \end{aligned}$$

   (b) Descrizione protocollo

Solution: Come protocollo abbiamo visto FLOODING. Questo ha tre stati:

$$\begin{aligned} S_{start} &= \{iniziatore\} \\ S_{init} &= \{iniziatore, inattivo\} \\ S_{final} = S_{term} &= \{finito\} \end{aligned}$$

Quando un nodo nello stato *iniziatore* riceve l'impulso iniziale, manda a tutti i suoi vicini il dato I e passa allo $finito$. Quando invece un nodo nello stato *inattivo* riceve il valore I, lo inoltra a tutti i vicini tranne il mittente, e passa allo stato $finito$.

   (c) Prestazioni e lower bound

Solution: In questo modo l'algoritmo ha le seguenti caratteristiche

$$\begin{aligned} M[\text{FLOODING}] &= \sum_{x \in \mathcal{E}} (N(x) - 1) + 1 = 2m - n + 1 = O(m) \\ T[\text{FLOODING}] &\leq d \end{aligned}$$

Per un lower bound temporale, è chiaro che nel caso peggiore ci si metta

$$T[\text{BROADCAST}] \geq d$$

che corrisponde al caso in cui l'iniziatore sia un estremo del diametro.
Mentre si può dimostrare che

$$M[\text{BROADCAST}] \geq m$$

infatti supponendo che sia minore, in un grafo G ci deve essere almeno un arco su cui non passano messaggi. Sia questo arco l'arco le due entità x e y. Supponiamo di estendere G con un entità z e di collegare x a y attraverso z, associando l'etichetta che prima era quella dell'arco (x, y) ai nuovi archi (x, z) e (z, y).

Ora visto che nell'algoritmo precedente non viaggiavano nodi sull'arco etichettato $\lambda_x(x, y)$, e $\lambda_x(x, y) = \lambda_x(x, z)$, allora nel nuovo grafo non verrà esplorato z, che quindi non potrà ricevere il dato di cui dovevamo fare il BROADCAST.

3. Traversal:

   (a) Definire il problema TRAVERSAL

Solution: Il problema del TRAVERSAL consiste nell'esplorare il grafo in modo sequenziale. Questo problema ha lower bound per i messaggi

$$M[\text{TRAVERSAL}] \geq m$$

per le stesse ragioni del BROADCAST.

Lower bound per il tempo

$$T[\text{TRAVERSAL}] \geq n - 1$$

visto che ogni nodo deve essere visitato in sequenza.

   (b) Funzionamento di DFTRAVERSAL e sua complessità

Solution: Implementeremo il traversal su una rete RI come una visita in profondità. Utilizzeremo:

- quattro stati

$$\begin{aligned} S &= \{iniziatore, inattivo, attivo, finito\} \\ S_{start} &= \{iniziatore\} \\ S_{init} &= \{iniziatore, inattivo\} \\ S_{term} = S_{final} &= \{finito\} \end{aligned}$$

- tre tipi di messaggi

T token
R return
B back-edge

caratteristica di questo algoritmo è la presenza in ogni momento di un singolo messaggio T in circolo nella rete.

- nodi contenenti
  - un booleano che indica se sono o no l'iniziatore, e nel caso non lo siano il padre
  - una lista di nodi non visitati

L'algoritmo segue lo schema:

1. se sono l'iniziatore, alla ricezione dell'impulso iniziale imposto come non visitati tutti i miei vicini, ed inizio la procedura di visita;
2. se sono idle e ricevo il token T imposto come non visitati tutti i vicini meno il sender e come padre il sender, ed inizio la procedura di visita;
3. se sono attivo e ricevo o B o R, chiamo la procedura di visita;
4. se sono attivo e ricevo il token T rispondo con un messaggio B;

dove la procedura di visita è definita nel seguente modo:

1. divento attivo;
2. se la lista dei non visitati è finita allora rispondo R al padre se ne ho uno, e passo allo stato $finito$;
3. altrimenti estraggo un vicino dalla lista dei non visitati e gli invio il token T e aspetto la risposta.

Nell'algoritmo così definito su ogni arco passano 2 messaggi (token e risposta due volte), per cui

$$M[\text{DFTraversal}] = 2m$$

e visto che l'algoritmo è sequenziale, otteniamo che il tempo non può che essere dello stesso ordine di grandezza del numero di messaggi

$$T[\text{DFTraversal}] = 2m$$

Si può definire una versione migliore di DFTraversal, chiamata DFTraversal*, che opera senza i messaggi utilizzando invece un messaggio *visited* che viene inviato in maniera asincrona rispetto al token T nel momento in cui un'entità lo riceve per la prima volta. L'algoritmo così definito ha le seguenti caratteristiche

$$M[\text{DFTraversal}^*] = O(m)$$
$$T[\text{DFTraversal}^*] = O(n)$$

4. Nell'ambito dei sistemi distribuiti spiegare:
   (a) Cos'è un protocollo

> **Solution:** Un protocollo è l'unione di tutti i comportamenti per ogni entità, cioè
> $$B(\mathcal{E}) = \bigcup_{x \in \mathcal{E}} B(x)$$
> un protocollo deve essere omogeneo, cioè due entità nella stessa rete non devono avere comportamenti diversi in risposta allo stesso evento e stato.

   (b) La configurazione di un sistema al tempo t

> **Solution:** La configurazione di un sistema al tempo t, $C(t)$, è composta da due elementi:
>
> - $\Sigma(t)$, l'insieme di tutti gli stati di tutte le entità;
> - Futuro(t), l'insieme di tutti gli eventi non ancora processati.

   (c) Come viene definito un problema P

> **Solution:** Un problema è definito come una tripla
> $$P = \langle P_{init}, P_{final}, R \rangle$$
> dove
>
> - P_{init} è un predicato che una configurazione iniziale deve rispettare;
> - P_{final} è un predicato che una configurazione finale deve rispettare;
> - R è l'insieme delle restrizioni di rete che si assumono.
>
> Le restrizioni di rete sono delle particolari caratteristiche della rete che vengono assunte, nel nostro caso ne utilizzeremo 5:
>
> - link bidirezionali;
> - connettività totale (della rete);
> - affidabilità totale, cioè mancanza di errori in passato e in futuro;
> - iniziatore unico;
> - initial distinct values, cioè ogni entità è identificabile da un valore.

(d) Come viene formalizzata la soluzione di P mediante un protocollo che termina

Solution: Dato un protocollo, questo trasforma una configurazione $C(t)$ in una $C(t+1)$, dove lo stato $\Sigma(t+1)$ e gli eventi da processare Futuro$(t+1)$ sono ottenuti processando gli eventi in Futuro(t) nello stato $\Sigma(t)$ con il protocollo. Si dice che un protocollo termina se questo porta eventualmente ad una configurazione finale, cioè una configurazione in cui Futuro$(t') = \varnothing$.

Un protocollo è corretto se partendo da una configurazione che rispetta il predicato iniziale del problema, questo termina e la configurazione finale rispetta il predicato finale.

5. Problema dello SpanningTree

(a) Definizione

Solution: Il problema dello SpanningTree è definito su reti RI, e consiste nel trovare un sottografo del grafo G privo di cicli.

Alcuni problemi hanno complessità che dipende dal numero di archi, di conseguenza ridurre questo valore permette di ridurre la complessità del problema.

(b) Protocollo Shout+

Solution: Nel protocollo Shout+ ci sono

- quattro stati

$$\begin{aligned} S &= \{iniziatore, inattivo, attivo, finito\} \\ S_{start} &= \{iniziatore\} \\ S_{init} &= \{iniziatore, inattivo\} \\ S_{term} = S_{final} &= \{finito\} \end{aligned}$$

- due tipi di messaggi

Q question
Y yes

- nodi contententi
  - un booleano che indica se l'entità è o no la radice, e nel caso non lo siano il padre;
  - un insieme, inizialmente vuoto, dei vicini membri dell'albero;
  - un contatore, inizializzato al numero di vicini.

Il protocollo procede nel seguente modo:

- se sono l'iniziatore e ricevo l'impulso iniziale, mando a tutti i vicini il messaggio Q e passo nello stato attivo;
- se sono inattivo e ricevo il messaggio Q, imposto come padre il sender, mando a tutti i vicini (tranne il sender) il messaggio Q e passo nello stato attivo;
- se sono attivo e ricevo un messaggio Y, aggiungo il sender all'insieme dei vicini che fanno parte del mio albero e decremento il contatore;
- se sono attivo e ricevo un messaggio Q, decremento il contatore;
- se sono attivo e il contatore è a 0, divento $finito$.

Questo protocollo impiega

$$M[\textsc{Shout+}] = 2m$$
$$T[\textsc{Shout+}] = d + 1$$