Algoritmi paralleli e distribuiti

Indice

Introduzione

Algoritmi 101

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.

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.

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**: rete internet;
- **reti di sensori**: sistemi con limitate capacità computazionali che rispondono a messaggi *ack*, *recover*, *wake up*, eccetera.

La risorsa tempo

Il **tempo** è una variabile fondamentale nell'analisi degli algoritmi. Definiamo prima la funzione

$$T(x) = \#\text{operazioni elementari sull'istanza } x.$$

Questo valore dipende fortemente dall'istanza x. Per risolvere questo problema, visto che noi vogliamo ragionare nel caso **worst case**, ovvero il caso che considera la situazione peggiore possibile, così da avere dei bound ragionevoli, definiamo ora il tempo come la funzione

$$t(n) = \max(\{T(x) \mid x \in \Sigma^n\}),$$

dove n è la grandezza dell'input. Abbiamo quindi raggruppato le istanze di grandezza n e abbiamo preso tra queste il tempo massimo. Spesso saremo interessati al *tasso di crescita* di $t(n)$, definito tramite **funzioni asintotiche**, e non ad una sua valutazione precisa.

Date due funzioni $f, g : \mathbb{N} \longrightarrow \mathbb{N}$, le principali funzioni asintotiche sono:

- **O grande**: $f(n) = O(g(n)) \iff f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$;
- **omega grande**: $f(n) = \Omega(g(n)) \iff f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$;

- **theta grande**: $f(n) = \Theta(g(n)) \iff c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$.

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

Un **modello di calcolo** mette a disposizione le **operazioni elementari** che usiamo per formulare i nostri algoritmi. Modelli di calcolo sono le macchine RAM, le MdT, le macchine WHILE, eccetera.

Esempio 0.1 (*Funzione palindroma*): La funzione *palindroma*, che dice se una stringa x di lunghezza n data in input è palindroma, ha tempo:
- $O(n^2)$ in una MdT;
- $O(n)$ in una architettura con memoria ad accesso casuale.

Anche 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.

Con tutte queste nozioni possiamo creare una serie di **classi di complessità**. Un problema è **risolto efficientemente** in tempo se e solo se è risolto da una MdT in tempo polinomiale. Abbiamo tre principali classi di equivalenza per gli algoritmi sequenziali:
- P, classe dei problemi di decisione risolti in tempo polinomiale su una MdT;
- FP, classe delle funzioni risolte in tempo polinomiale su una MdT;
- NP, classe dei problemi di decisione risolti in tempo polinomiale su una MdTnd.

Per quanto riguarda gli algoritmi paralleli, definiamo una nuova classe di complessità, ovvero NC, classe delle funzioni che ammettono algoritmi paralleli efficienti. Un problema appartiene alla classe NC se viene risolto in tempo *polilogaritmico* e in spazio *polinomiale*.

Teorema 0.1: Vale

$$NC \subseteq FP\,.$$

Dimostrazione 0.1: 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, ovvero abbiamo un numero polinomiale di processori, abbiamo un *«programma unico»* con una grandezza polinomiale.

Ogni parte di questo *«super programma»* ha tempo polinomiale, il tempo totale è un polinomio per un polilogaritmo, quindi il tempo è polinomiale. ■

Come per il famosissimo problema $P = NP$, 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 P-completi.

Parte I — Algoritmi paralleli

1. Architetture parallele

Prima di vedere l'architettura parallela che ci accompagnerà durante tutta questa parte del corso, diamo una brevissima introduzione delle due architetture principali che possiamo utilizzare quando ci troviamo in un ambiente di calcolo parallelo.

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, ognuno dei quali possiede anche una *«memoria personale»*, formata da 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.

1.2. Memoria distribuita

L'architettura a **memoria distribuita** utilizza una rete di interconnesione centrale che permette lo scambio di informazioni tra un numero n di processori, ognuno dei quali possiede anche una *«memoria personale»*, formata da registri. Un **clock** centrale e comune coordina tutti i processori, che comunicano attraverso la rete di interconnesione in un tempo che dipende dalla distanza tra i processori.

1.3. Modello PRAM

Vediamo finalmente l'architettura che ci accompagnerà durante tutte questa parte del corso.

1.3.1. Struttura

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 M tramite due primitive che lavorano in tempo costante. Esse sono **LOAD** e **STORE**, usate rispettivamente per caricare un dato dalla memoria e salvare un dato nella memoria.

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

Passo parallelo

```
1: for all i ∈ 𝕀 par do
2:     istruzione_k
```

Il passo parallelo fa eseguire a tutti i processori con indice in $\mathbb{I}$ la k-esima istruzione, altrimenti fa eseguire una **nop**, l'operazione nulla.

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.

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 a caso;
- **max/min**: si sceglie il processore con il dato massimo/minimo;
- **priority**: si sceglie il processore con priorità maggiore.

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

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

1.3.2. 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** $t(n)$.

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 di lunghezza 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

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

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

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. Esso viene definito come la quantità

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

Ad esempio, $S = 4$ indica che l'algoritmo parallelo è 4 volte più veloce dell'algoritmo sequenziale, ma questo ricade nel caso $T(n, p(n)) = \Theta(T(n,1))$, poiché il fattore che definisce la complessità si semplifica. Vogliamo avere $S \longrightarrow \infty$, 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 $S \longrightarrow \infty$ perché stiamo utilizzando un numero spropositato di processori.

Esempio 1.3.2.1: Ad esempio, nel **problema di soddisfacibilità** (*SODD*) potremmo utilizzare 2^n processori, ognuno dei quali risolve un assegnamento. Con una serie di *OR* paralleli andiamo a vedere se siamo riusciti ad ottenere un assegnamento valido di variabili, tutto questo in tempo

$$T(n, p(n)) = \log_2(2^n) = n.$$

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.

Per comodità, da qui in avanti chiameremo $E(n, p(n))$ semplicemente con E.

Teorema 1.3.2.1: Vale

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

Dimostrazione 1.3.2.1: 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 $\hat{T}(n,1)$ che però *«fa peggio»* del miglior algoritmo sequenziale $T(n,1)^*$, quindi

$$T(n,1)^* \leq \hat{T}(n,1) \leq p(n) \cdot t_1(n) + ... + p(n) \cdot t_{k(n)}(n).$$

La somma di destra rappresenta la sequenzializzazione dell'algoritmo parallelo, che richiede quindi un tempo uguale a $p(n)$ 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

$$T(n,1)^* \leq p(n) \cdot T(n,p(n)) \Longrightarrow \frac{T(n,1)^*}{p(n) \cdot T(n,p(n))} \leq 1 \Longrightarrow E \leq 1. \qquad \blacksquare$$

Se $E \longrightarrow 0$ abbiamo dei problemi, perché nonostante un ottimo speed-up stiamo tendendo a 0, ovvero il numero di processori utilizzati è eccessivo. Devo quindi ridurre il numero di processori p 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 $\frac{p}{k}$ totali 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

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

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

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

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

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

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

Questa idea di ridurre il numero di processori è quella che viene definita come **principio di Wyllie**: se $E \longrightarrow 0$ quando $T(n,p(n)) = o(T(n,1))$ allora è $p(n)$ che sta crescendo troppo, e dobbiamo quindi ridurre il numero di processori che utilizziamo.

2. Sommatoria

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

Il problema:

- prende in **input** una serie di numeri $M[1], ..., M[n]$;
- restituisce in **output**, dentro il registro $M[n]$, la somma dei valori $M[1], ..., M[n]$.

Un buon algoritmo sequenziale è quello che utilizza $M[n]$ come **accumulatore**, lavorando in tempo

$$T(n, 1) = n - 1$$

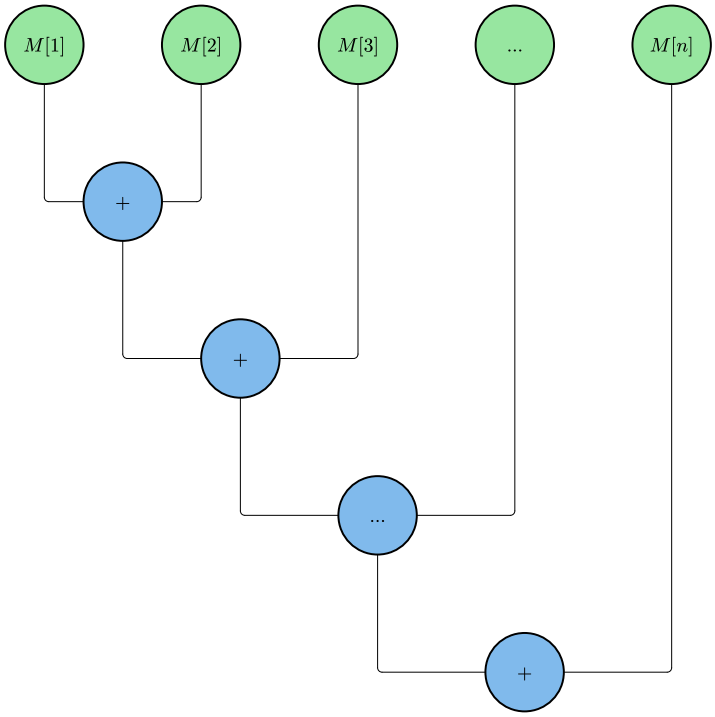
senza usare della memoria aggiuntiva.

Sommatoria sequenziale

```
1: for i = 1 to n do
2:     M[n] = M[n] + M[i]
3: output M[n]
```

2.1. Prima versione

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



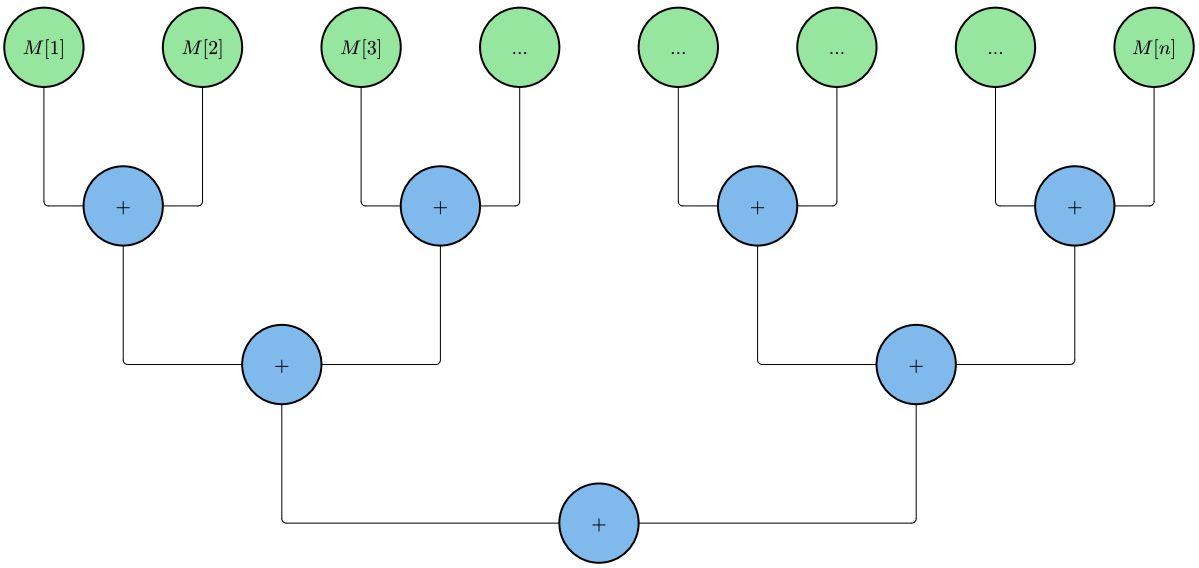
Usiamo $n - 1$ processori, ma abbiamo dei problemi:

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

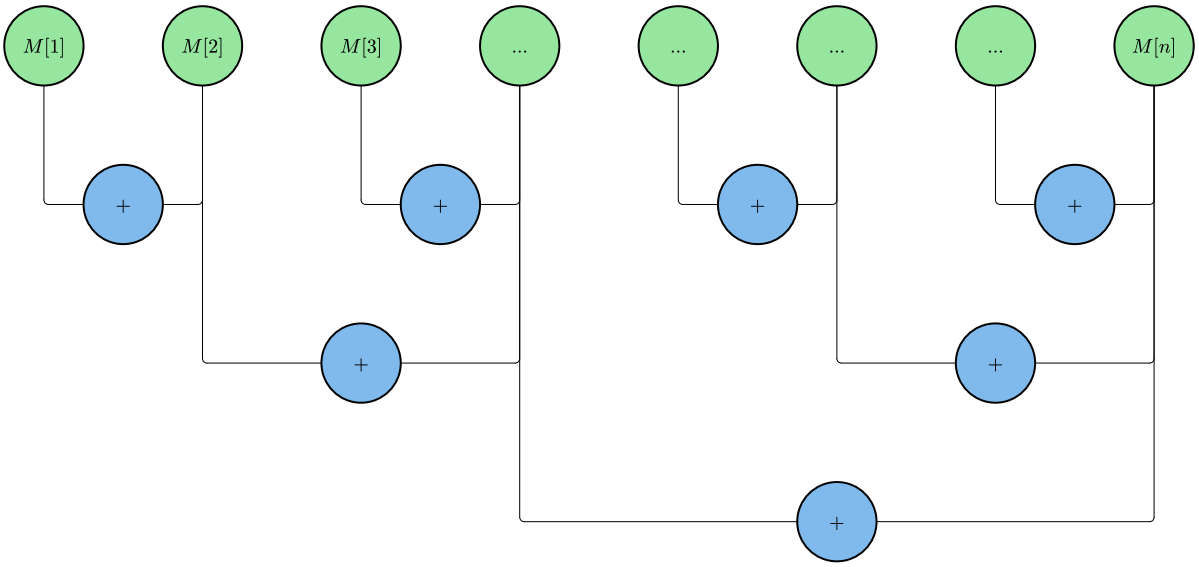
L'efficienza che otteniamo è

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

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



Quello che otteniamo è un albero binario, sempre con $n-1$ processori ma 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

Sommatoria parallela

3: ⌊ ⌊ $M[2^i k] = M[2^i k] + M[2^i k - 2^{i-1}]$

4: return $M[n]$

Teorema 2.1.1: L'algoritmo di sommatoria parallela è EREW.

Dimostrazione 2.1.1: 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 prima che $2^i a \neq 2^i b$, ma 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 $a \in \mathbb{N}$. ■

Teorema 2.1.2: L'algoritmo di sommatoria parallela è corretto.

Dimostrazione 2.1.2: Per dimostrare che l'algoritmo 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

$$M[2^i k] = M[2^i k] + ... + M[2^i(k-1) + 1].$$

Notiamo che se $i = \log_2(n)$ allora ho un solo processore attivo, quindi abbiamo $k = 1$ e otteniamo la definizione di sommatoria, ovvero

$$M[n] = M[n] + ... + M[1].$$

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 i 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.

Il primo fattore è

$$M[2^i k] = M[2^{i-1} \cdot (2k)] \underset{\text{HP-I}}{=} M[2^{i-1} \cdot (2k)] + ... + M[2^{i-1} \cdot (2k-1) + 1].$$

Il secondo fattore è

$$M[2^i k - 2^{i-1}] = M[2^{i-1}(2k-1)] \underset{\text{HP-I}}{=} M[2^{i-1}(2k-1)] + \ldots + M[2^{i-1}(2k-2)+1].$$

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 cella del secondo fattore. Inoltre, l'ultima cella del secondo fattore può essere riscritta come

$$M[2^i(k-1)+1],$$

quindi abbiamo ottenuto esattamente quello che volevamo dimostrare. ■

Se n è potenza di 2 usiamo $p(n) = \frac{n}{2}$ processori con 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»* il nostro 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 $2n$ usiamo $p(n) = \frac{2n}{2} = n$ processori e abbiamo tempo $T(n, n) = 4\log(2n) \leq 5\log(n)$.

In poche parole, in entrambi i casi abbiamo $p(n) = O(n)$ e $T(n, p(n)) = O(\log(n))$.

Se però calcoliamo l'efficienza otteniamo

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

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

2.2. Seconda versione [ottimizzata]

Usiamo il **principio di Wyllie**: vogliamo arrivare ad avere $E \longrightarrow C \neq 0$, diminuendo il numero di processori utilizzati. Andiamo quindi ad utilizzare p processori raggruppando i numeri presenti in M in gruppi grandi $\Delta = \frac{n}{p}$, ognuno associato ad un processore.

Al primo passo dell'algoritmo ogni processore esegue la somma sequenziale dei Δ valori contenuti nel proprio gruppo, ovvero $M[k\Delta] = M[k\Delta] + \ldots + M[(k-1)\Delta + 1]$. Successivamente, si esegue l'algoritmo sommatoria proposto prima sulle celle di memoria $M[\Delta], M[2\Delta], \ldots, M[p\Delta]$, e in quest'ultima viene inserito il risultato finale.

In questa versione ottimizzata usiamo $p(n) = p$ processori con un tempo $T(n, p)$ formato dal primo passo parallelo *«di ottimizzazione»* sommato al tempo di sommatoria, quindi

$$T(n, p) = \frac{n}{p} + 5\log(p).$$

Andiamo a calcolare l'efficienza

$$E = \frac{n-1}{p \cdot \left(\frac{n}{p} + 5\log(p)\right)} = \frac{n-1}{n + \underbrace{5p\log(p)}_{n}} \approx \frac{n}{2n} = \frac{1}{2} \neq 0.$$

Per arrivare ad avere questa efficienza che tanto volevamo, dobbiamo imporre $5p\log(p) = n$, quindi

$$p = \frac{n}{5\log(n)}.$$

Con questa assunzione riusciamo ad ottenere un tempo

$$T(n, p(n)) \leq 10 \log(n).$$

Diamo un **lower bound**: sommatoria possiamo visualizzarlo usando un albero binario, dove le foglie sono i 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 dà 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.

2.3. Operazione iterata

La sommatoria può essere uno schema per altri problemi.

L'**operazione iterata** è una operazione associativa **OP** sulla quale definiamo un problema che:

- prende in **input** una serie di valori $M[1], \ldots, M[n]$;
- restituisce in **output**, dentro il registro $M[n]$, l'operazione **OP** calcolata su tutti gli $M[i]$.

Abbiamo visto che sommatoria ammette soluzioni efficienti con $p(n) = O\left(\frac{n}{\log(n)}\right)$ processori utilizzati e tempo $T(n, p(n)) = O(\log(n))$.

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

Supponiamo una CRCW-PRAM e vediamo il problema **AND-iterato**, ovvero

$$M[n] = \bigwedge_i M[i].$$

Come detto poco fa, il tempo è costante perché la PRAM è più potente.

$\bigwedge$-iterato

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

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

In questo specifico caso abbiamo $p(n) = n$ processori e tempo $T(n, n) = 3$. Calcoliamo l'efficienza di questo problema come

$$E = \frac{n-1}{3n} \longrightarrow \frac{1}{3}.$$

Per il problema **OR-iterato** vale la stessa cosa, basta modificare leggermente il codice proposto.

Indice

3. Applicazioni di sommatoria

Il problema sommatoria lo possiamo utilizzare come modulo per risolvere dei problemi più complessi. In questo capitolo vediamo quattro problemi molto semplici.

3.1. Prodotto interno di vettori

Questo problema:

- prende in **input** due vettori $x, y \in \mathbb{N}^n$;
- restituisce in **output** il **prodotto scalare**.

Il prodotto scalare è un numero reale definito dalla formula

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i.$$

Il miglior tempo sequenziale è $T(n, 1) = 2n - 1$, formato da n prodotti e $n - 1$ somme finali.

Il modulo sommatoria viene usato per eseguire le somme finali in parallelo. L'algoritmo che vedremo si articola in due fasi:

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

Per sommatoria ci serviranno $p = c_1 \frac{n}{\log(n)}$ processori, con tempo $t = c_2 \log(n)$. Per la prima fase ci serviranno invece $p = \frac{n}{\log(n)}$ processori, con tempo $t = c_3 \log(n)$.

Ma allora utilizziamo in totale $p = \frac{n}{\log(n)}$ processori con tempo $t = \log(n)$. L'efficienza è

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

3.2. Prodotto matrice vettore

Questo problema:

- prende in **input** una matrice $A \in \mathbb{N}^{n \times n}$ e un vettore $x \in \mathbb{N}^n$;
- restituisce in **output** il prodotto Ax.

Per questo problema usiamo il prodotto interno di vettori come modulo.

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

Per l'approccio parallelo, l'idea è usare il modulo del prodotto interno in parallelo per n volte. Il vettore se è acceduto simultaneamente dai moduli precedenti ci obbliga ad avere una politica CREW.

Questa idea utilizza $p(n) = \frac{n^2}{\log(n)}$ processori con tempo $T(n, p(n)) = \log(n)$. "efficienza vale

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

3.3. Prodotto matrice matrice

Questo problema:

- prende in **input** due matrici $A, B \in \mathbb{N}^{n \times n}$;
- restituisce in **output** il prodotto matriciale AB.

Usiamo ancora il prodotto interno come modulo per risolvere questo problema.

Il miglior tempo sequenziale è $T(n,1) = n^{2.8}$, ottenuto con **l'algoritmo di Strassen**.

Come prima, l'idea è fare dei prodotti interni paralleli, solo che in questo caso sono n^2. Ci servirà ancora la politica CREW, per via dell'accesso simultaneo alle righe di A e alle colonne di B.

Questo algoritmo usa $p(n) = \frac{n^3}{\log(n)}$ processori con tempo $T(n, p(n)) = \log(n)$. L'efficienza vale

$$E = \frac{n^{2.80}}{\frac{n^3}{\log(n)}\log(n)} \longrightarrow 0.$$

L'efficienza tende a 0 ma lentamente, quindi lo accettiamo come risultato.

3.4. Potenza di matrice

Questo ultimo problema:

- prende in **input** una matrice $A \in \mathbb{N}^{n \times n}$;
- restituisce in **output** la potenza A^t, con $t = 2^k$.

Questo problema si risolve come prodotto iterato con tempo $T(n,1) = n^{2.80}\log(n)$.

Potenza di matrice sequenziale

```
1: for i = 1 to log(n) do
2:     A = A · A
```

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

Per questo problema utilizziamo $p(n) = \frac{n^3}{\log(n)}$ processori con tempo $T(n, p(n)) = \log^2(n)$.

Purtroppo, l'efficienza che otteniamo è

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

Come prima però, accettiamo l'efficienza visto che tende a 0 lentamente.

4. Somme prefisse

Anche il problema delle **somme prefisse** userà il modulo della sommatoria per essere risolto.

Questo problema

- prende in **input** una serie di numeri $M[1], ..., M[n]$;
- restituisce in **output**, nella cella k-esima, la somma di tutti gli elementi precedenti più se stesso.

In poche parole, nella cella k-esima abbiamo la quantità

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

Assumiamo, per semplicità, che n sia una potenza di 2.

Il migliore 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 $T(n, 1) = n - 1$.

4.1. Prima versione

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

Con questo approccio abbiamo un paio di problemi:

- l'algoritmo non è EREW, ma questo è facilmente risolvibile;
- l'algoritmo usa $p(n) = \frac{n^2}{\log(n)}$ processori con tempo $T(n, p(n)) = \log(n)$.

Con questo dispendio di tempo e spazio, l'efficienza vale

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

4.2. Seconda versione [pointer doubling]

Usiamo il **pointer doubling**, un algoritmo ideato da **Kogge-Stone** nel 1973.

Dobbiamo stabilire dei legami tra i numeri, e ognuno di questi legami viene preso in carico da un processore, che ne fa la somma. Quest'ultima viene poi inserita nella cella di indice maggiore.

Alla prima iterazione ho dei link tra una cella e la successiva. Alla seconda iterazione ho dei link tra una cella e quella due posizioni dopo. Alla terza iterazione ho dei link tra una cella e quella quattro posizioni dopo. Alla quarta iterazione eccetera.

Notiamo due cose:

- la distanza dei link raddoppia ad ogni iterazione;
- alcuni processori non hanno dei successori, ovvero qualche processore non riesce a linkarsi con nessun altro elemento del vettore.

L'algoritmo termina quando non riesco più a mettere dei link tra celle di memoria.

Vediamo un po' di fatti che riguardano questo algoritmo:

- ogni volta raddoppiamo la distanza dei link, quindi l'algoritmo lavora in **tempo logaritmico**.

- prima di eseguire una somma e un aggiornamento di link al passo i, in quel momento erano attivi $n - 2^{i-1}$ processori.
- alla fine del passo i, il numero di processori senza successore è 2^i.

Ci serve sicuramente un vettore di successori, che usiamo per ricavare le celle da sommare tra loro.

Sia S tale vettore. Sia $S[k]$ il successore di $M[k]$. Come inizializzo S?

Prima della prima iterazione assegno
- $S[k] = k + 1$;
- $S[n] = 0$.

Algoritmo di Kogge-Stone

```
1: for i = 1 to log(n) do
2:     for 1 ≤ k ≤ n − 2^{i−1} par do
3:         M[S[k]] = M[k] + M[S[k]]
4:         S[k] = S[S[k]]
```

L'algoritmo incredibilmente è EREW perché accediamo sì alle stesse celle, ma in momenti diversi.

Teorema 4.2.1: L'algoritmo di Kogge-Stone è corretto.

Dimostrazione 4.2.1: Siamo in una EREW-PRAM, quindi il processore P_i lavora su $M[i]$ e $M[S[i]]$, e se considero $i \neq j$ allora $S[i] \neq S[j]$ e quindi abbiamo successori diversi (*accettiamo il caso in cui entrambi i successori sono nulli*).

Devo dimostrare che nella cella k-esima cella ho la somma degli elementi precedenti più l'elemento stesso. Vale la proprietà che dice che all'i esimo passo vale

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

Se $i = \log(n)$, ovvero sono all'ultima iterazione, siamo nel primo caso della funzione definita a tratti e quindi vale

$$M[t] = M[t] + ... + M[1].$$

■

Mostriamo che vale la proprietà descritta nella dimostrazione precedente.

Dimostrazione 4.2.2: Dimostriamo questa proprietà per induzione su i

Caso base: se $i = 1$ allora:
- se $t \leq 2$ allora $M[1] = M[1]$ e $M[2] = M[2] + M[1]$;
- se $t > 2$ allora $M[t] = M[t] + M[t - 1]$.

Passo induttivo: assumo vero per $i - 1$ e dimostro vero per i.

Prima di iniziare il passo i-esimo il vettore S contiene

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

Le celle con indice $\leq 2^{i-1}$ hanno la proprietà vera per ipotesi induttiva.

Concentriamoci sugli indici che avanzano da analizzare.

Se $2^{i-1} < t \leq 2^i$ allora possiamo scrivere t come $t = 2^{i-1} + a$ e quindi

$$\begin{aligned} M[a + 2^{i-1}] &= \underbrace{M[a]}_{a \leq 2^{i-1}} + \underbrace{M[a + 2^{i-1}]}_{a + 2^{i-1} > 2^{i-1}} = \\ &= M[1] + \ldots + M[a] + | + M[a+1] + \ldots + A[a + 2^{i-1}]. \end{aligned}$$

Se invece $t > 2^i$ allora possiamo scrivere t come $t = a + 2^i$ e quindi

$$\begin{aligned} M[a + 2^i] = M[(a + 2^{i-1}) + 2^{i-1}] &= \underbrace{M[a + 2^{i-1}]}_{a + 2^{i-1} > 2^{i-1}} + \underbrace{[a + 2^i]}_{a + 2^i > 2^{i-1}} = \\ &= M[a+1] + \ldots + M[a + 2^{i-1}] + | + M[a + 2^{i-1} + 1] + \ldots + M[a + 2^i]. \end{aligned}$$

In entrambi gli indici mancanti la proprietà risulta vera. ■

Valutiamo infine questo algoritmo: esso utilizza $p(n) = n - 1$ processori con un utilizzo di tempo uguale a $T(n, p(n)) = 8\log(n)$. Il fattore logaritmico viene dal passo parallelo. L'efficienza è quindi

$$E = \frac{n-1}{(n-1)8\log(n)} = \frac{1}{8\log(n)} \longrightarrow 0.$$

Anche se l'efficienza tende a 0 lentamente, non siamo soddisfatti di questa soluzione.

Sfruttiamo il **principio di Wyllie** per far sparire il fattore $\log(n)$ dal denominatore. Usiamo quindi $p(n) = O\left(\frac{n}{\log(n)}\right)$ processori, ottenendo sempre un tempo logaritmico ma andremo ad avere

$$E = \frac{n-1}{c_1 \frac{n}{\log(n)} c_2 \log(n)} = \frac{n-1}{c_1 c_2 n} \longrightarrow C \neq 0.$$

Questo problema può essere usato come modulo per risolvere il problema **OP-prefissa**, dove non devo più utilizzare la somma come operazione associativa ma devo usare una operazione **OP** generica.

5. Valutazione di polinomi

Questo problema

- prende in **input** un polinomio $p(x) = a_0 + a_1 x + ... + a_n x^n$ e un valore $\alpha \in \mathbb{R}$;
- restituisce in **output** la valutazione di p nel valore α, ovvero $p(\alpha)$.

In memoria ho il valore α e i coefficienti del polinomio $A[0], ..., A[n]$.

L'algoritmo sequenziale esegue circa n^2 operazioni, tra somme e prodotti.

Con il metodo di **Ruffini-Horner** possiamo abbassare il numero di operazioni. L'idea che hanno avuto questi pazzi è quella di eseguire un raccoglimento di x iterativo, ottenendo

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

Partiamo con $p = a^n$. Ad ogni passo dell'algoritmo calcoliamo somma+prodotto di una parentesi e assegniamo questo valore di nuovo a p. Vale quindi che

$$p = a_i + p\alpha.$$

Algoritmo di Ruffini-Horner

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

Contando le 2 operazioni elementari, questo tempo sequenziale vale $T(n, 1) = 2n$.

Per l'algoritmo parallelo dobbiamo:

- costruire il vettore delle potenze di α, chiamato Q, e tale che $Q[k] = \alpha^k \mid 0 \leq k \leq n$;
- eseguire il prodotto interno tra A e Q.

Il vettore Q possiamo calcolarlo grazie al problema **replica**: dobbiamo mettere α in tutti gli elementi di Q da 1 a n, e poi applicare il prodotto prefisso.

Replica

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

L'algoritmo scritto è CREW, perché α è in memoria e quindi ho accesso simultaneo al dato, usa $p(n) = n$ processori con tempo $T(n, p(n)) = 2$. L'efficienza vale

$$E = \frac{n}{2n} = \frac{1}{2} \neq 0.$$

Possiamo modificare leggermente il modulo replica, perché se poi lo dobbiamo utilizzare prima del prodotto prefisso molti processori vengono inutilizzati. Abbassiamo quindi il numero di processori con il **principio di Wyllie**, raggruppando in gruppi di $\log(n)$ elementi l'input. Il k-esimo processore carica α nelle celle di indice

$$(k-1)\log(n) + 1 \quad | \quad ... \quad | \quad k\log(n).$$

Replica migliorato

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

Questa nuova versione usa $p(n) = \frac{n}{\log(n)}$ processori con tempo $t = c\log(n)$. L'efficienza vale

$$E = \frac{n}{c\frac{n}{\log(n)}\log(n)} = \frac{1}{c} \neq 0.$$

Anche questa versione però rimane CREW per l'accesso ad α. Noi però vorremmo un EREW, quindi:

- costruiamo il vettore $[\alpha, 0, \ldots, 0]$;
- eseguiamo somme prefisse

Replica ancora migliore

1: $Q[1] = \alpha$
2: for $k = 2$ to n par do
3: ⌊ $Q[k] = 0$
4: SommePrefisse(Q)

Posso anche ridurre i processori con il **principio di Wyllie**, usando

- $p(n) = \frac{n}{\log(n)}$ processori e tempo $T(n, p(n)) = \log(n)$ per costruire Q;
- $p(n) = \frac{n}{\log(n)}$ processori e tempo $T(n, p(n)) = \log(n)$ per le somme prefisse.

Finalmente abbiamo un algoritmo EREW.

Valutazioni di polinomi parallela

1: Replica(Q, α)
2: ProdottoPrefisso(Q)
3: ProdottoInterno(A, Q)

Tutti i moduli utilizzati in questa ultima versione dell'algoritmo usano $p(n) = \frac{n}{\log(n)}$ processori con tempo $T(n, p(n)) = \log(n)$. L'efficienza, anche in questo caso, tende a $C \neq 0$.

6. Ricerca

Questo problema:

- prende in **input** una serie di valori $M[1], ..., M[n]$ e un valore α;
- restituisce in **output** 1 se $\alpha \in M$, 0 altrimenti, mettendo il risultato della valutazione in $M[n]$.

Il migliore algoritmo sequenziale ha tempo $T(n, 1) = n$ se consideriamo un array generico. Nel caso di array ordinato, tramite **ricerca dicotomica**, abbassiamo il tempo ad un fattore logaritmico. Considerando invece un algoritmo quantistico su un array non ordinato, il tempo vale $T(n, 1) = \sqrt{n}$, ma questo è dato dall'uso dell'**interferenza quantistica**.

Vediamo un prima versione CRCW parallela che utilizza una flag F.

CRCW con flag

```
1: F = 0
2: for k = 1 to n par do
3:     if M[k] == α
4:         F = 1
5: M[n] = F
```

L'uso della flag è necessario: senza flag, se a fine algoritmo vale $M[n] == 1$ avrei due casi:

- ho avuto esito positivo della ricerca;
- il valore 1 era il valore originale che avevo in memoria.

Con questa versione ho quindi CR in α e CW in F. Vengono utilizzati $p(n) = n$ processori con tempo costante. L'efficienza vale

$$E = \frac{n}{nC} = \frac{1}{C} \neq 0.$$

Vediamo una seconda versione del problema, utilizzando un CREW (*quindi senza l'uso di flag*).

CREW con MAX-iterato

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

Trasformiamo M in un vettore booleano e poi vediamo il massimo valore presente in esso. Come prima, abbiamo la CR per l'accesso al valore α.

Stiamo utilizzando $p(n) = n$ processori, sempre con tempo costante. Con il **principio di Wyllie** andiamo a ridurre i processori a $p(n) = \frac{n}{\log(n)}$ e il tempo a $T(n, p(n)) = \log(n)$, che sono uguali a quelli del MAX-iterato. L'efficienza è quindi

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

La terza versione del problema usa invece una politica EREW, che a noi piace molto.

EREW con replica e MAX-iterato

1: Replica(A, α)
2: for $k = 1$ to n par do
3: $\quad M[k] = (M[k] == A[k] \ ? \ 1 : 0)$
4: MAX-iterato(M)

Usando sempre il **principio di Wyllie**, ogni modulo di questo algoritmo usa $p(n) = \frac{n}{\log(n)}$ processori con tempo $T(n, p(n)) = \log(n)$. L'efficienza vale quindi

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

Il problema di ricerca ha alcune **varianti**:

- numero di volte in cui α compare dentro M, risolto usando il modulo sommatoria al posto del modulo MAX-iterato, così da contare effettivamente quante volte α è presente;
- indice massimo di α dentro M, risolto assegnando $M[k] = k$ quando cerchiamo α dentro M, mantenendo poi il MAX-iterato alla fine;
- posizione minima di α dentro M, risolto usando una **OP iterata** tale che

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

7. Ordinamento

Detto anche problema del **ranking**, questo problema:

- prende in **input** una serie di valori $M[1], ..., M[n]$;
- restituisce in **output** una **permutazione** $p : \{1, ..., n\} \longrightarrow \{1, ..., n\}$ tale che

$$M[p(1)] \leq ... \leq M[p(n)],$$

  con $p(i)$ indice dell'elemento del vettore M che va in posizione i.

Potremmo anche ordinare direttamente in M, ma noi decidiamo di usare gli indici.

In genere, gli algoritmi di ordinamento sono basati sui **confronti**, ovvero dei check nella forma

$$M[i] \leq M[j] \ ? \quad \text{SI} : \text{NO} \ .$$

Teorema 7.1: Gli algoritmi di ordinamento basati sui confronti hanno tempo

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

Dimostrazione 7.1: Dimostriamo i due bound di questi algoritmi:

- upper bound: esistono algoritmi, tipo il merge sort, che impiegano al massimo $n \log(n)$ passi per essere eseguiti;
- lower bound: costruiamo un albero binario di decisione, dove ogni nodo è un possibile confronto. Il numero di foglie che otteniamo è il numero di permutazioni dell'input, che sono $n!$. L'altezza dell'albero è il numero di confronti che devo fare, perché un cammino dalla radice ad una foglia percorre tutti i confronti che devono essere fatti per ordinare l'input in quel preciso modo. Sappiamo che il numero di foglie di un albero binario è al massimo 2^h, con h altezza dell'albero, quindi

$$2^h \geq \#\text{foglie} = n! \underset{\text{STIRLING}}{\Longrightarrow} h \geq \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) \Longrightarrow h \geq n \log(n).$$

Quindi vale $\Theta(n \log(n))$. ■

7.1. Counting sort

Vediamo un primo algoritmo di ordinamento, basato sul **conteggio**. Il **counting sort** conta i confronti. Sequenzialmente, il tempo è $T(n, 1) = \Theta(n^2)$ perché deve confrontare tutte le coppie. Assumiamo, per semplicità, che n sia potenza di 2 e che gli elementi siano tutti diversi tra loro.

Nel counting sort, $M[i]$ va in posizione k se e solo se ci sono k elementi $\leq M[i]$ in M.

Per effettuare questo conteggio, usiamo un vettore V che contiene i vari valori di k per ogni valore in M. Questo vettore è la **permutazione inversa** di p:

- prima, data una posizione, mi veniva detto che indice doveva finire in quel posto;
- ora, dato un indice, mi viene detto in che posizione devo finire.

Counting Sort sequenziale

1: for $i = 1$ to n

Counting Sort sequenziale

```
2:  ⌊ V[i] = 0
3: for i = 1 to n
4:  |  for j = 1 to n
5:  |  |  if M[j] ≤ M[i] then
6:  ⌊  ⌊  ⌊ V[i] = V[i] + 1
7: for i = 1 to n
8:  ⌊ F[V[i]] = M[i]
9: for i = 1 to n
10: ⌊ M[i] = F[i]
```

I primi due cicli for andrebbero già bene, ma abbiamo aggiunto gli ultimi due for per ordinare effettivamente il vettore. Il numero di confronti è $O(n^2)$, visto il doppio ciclo for, quindi anche il tempo $T(n, p(n))$ assume quel valore.

La versione parallela utilizza un processore per ogni coppia di indici (i, j) che calcola $M[j] \leq M[i]$ e aggiorna una matrice booleana $V[i, j]$ con il risultato del confronto. Notiamo come l'i-esima riga individua gli elementi di M che sono $\leq M[i]$ quando la cella contiene il valore 1.

Per ottenere il numero di elementi complessivo, che effettivamente ci interessa, eseguo la sommatoria parallela sulle righe, ottenendo un vettore colonna $V[i, n]$ che coincide con il vettore V della versione sequenziale precedente.

Counting Sort parallelo

```
1: for i ≤ n ∧ j ≤ n par do
2:  ⌊ V[i, j] = (M[j] ≤ M[i] ? 1 : 0)
3: for i = 1 to n par do
4:  ⌊ Sommatoria(V[i, 1], ..., V[i, n])
5: for i = 1 to n par do
6:  ⌊ M[V[i]] = M[i]
```

Questo algoritmo non assolutamente EREW, vista la lettura concorrente, ma la scrittura non lo è invece, visto che scriviamo in celle ogni volta diverse. L'algoritmo è quindi CREW.

Per il primo ciclo for usiamo $p(n) = n^2$ processori con tempo $T(n, p(n)) = 4$, che però con il **principio di Wyllie** trasformiamo in $p(n) = \frac{n^2}{\log(n)}$ processori con tempo $T(n, p(n)) = \log(n)$.

Per il secondo ciclo for usiamo $p(n) = \frac{n^2}{\log(n)}$ processori, ovvero n modulo sommatoria da $\frac{n}{\log(n)}$ processori ciascuno, con tempo $T(n, p(n)) = \log(n)$.

Per il terzo e ultimo ciclo for usiamo $p(n) = n$ processori con tempo $T(n, p(n)) = 3$.

I processori totali sono quindi $p(n) = \frac{n^2}{\log(n)}$ con tempo $T(n, p(n)) = \log(n)$. L'efficienza vale

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

e nemmeno lentamente, quindi dobbiamo trovare una soluzione alternativa.

7.2. Bitonic sort

Un algoritmo parallelo migliore è il **bitonic sort**, che avrà comunque efficienza $E \longrightarrow 0$ ma molto lentamente. Un altro algoritmo parallelo ottimo è quello di **Cole**, del 1988, ma non lo vedremo, nonostante abbia $E \longrightarrow C \neq 0$.

Per il bitonic sort prendiamo spunto dal merge sort, che usa la tecnica Divide et Impera.

Merge sort

1: if $|A| > 1$
2: $\quad A_s = \text{MergeSort}(A[1], ..., A[\frac{n}{2}])$
3: $\quad A_d = \text{MergeSort}(A[\frac{n}{2} + 1], ..., A[n])$
4: $\quad A = \text{Merge}(A_s, A_d)$
5: return A

La routine di merge effettua l'unione dei due array ordinati A_s e A_d scorrendoli in sequenza e mettendo i valori ordinati in A. Nel caso peggiore, questa routine impiega tempo $T_m(n) = n$. Il tempo di complessivo di merge sort è invece $T(n) = n \log(n)$.

Possiamo sfruttare l'idea che ci dà il merge sort? Dividiamo continuamente il vettore fino ad arrivare ad un elemento. Qua devo fare il merge, se usassi un approccio parallelo avrei $\log(n) - 1$ passi paralleli essendo un albero. Purtroppo, il merge non è parallelizzabile, e quindi avrei sempre e comunque un tempo $T(n, p(n) = n \log(n)$.

Quando l'operazione di merge è facile? Supponiamo A_s e A_d ordinate con gli elementi di A_s tutti minori di A_d. La routine di merge è facilissima: basta concatenare le due sequenze.

Cercheremo di ottenere questa situazione usando delle sequenze di numeri particolari, che sono dette **sequenze bitoniche**.

Abbiamo due operazioni fondamentali:

- **reverse**, per fare il reverse di una sequenza;
- **minmax**, che costruisce i vettori $A_{\min}$ e $A_{\max}$ effettuando i seguenti passi:
  - divide a metà il vettore A nei vettori $A_{\min}$ (*sx*) e $A_{\max}$ (*dx*);
  - prese le posizioni a distanza $\frac{n}{2}$ in A, nel vettore $A_{\min}$ viene messa la componente più piccola delle due e nel vettore $A_{\max}$ viene messa la componente più grande delle due.

Reverse

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

Vengono utilizzati $p(n) = \frac{n}{2}$ processori con tempo $T(n, p(n)) = 4$.

MinMax

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

Vengono utilizzati $p(n) = \frac{n}{2}$ processori con tempo $T(n, p(n)) = 5$.

Vediamo ora le due sequenze numeriche che useremo per risolvere questo problema.

Definizione 7.2.1 (*Sequenza unimodale*): Una sequenza A è **unimodale** se e solo se

$$\exists k \in \{1, ..., n\} \mid A[1] > A[2] > ... > A[k] < A[k+1] < ... < A[n]$$

oppure

$$\exists k \in \{1, ..., n\} \mid A[1] < A[2] < ... < A[k] > A[k+1] > ... > A[n].$$

In poche parole, esiste un indice che mi individua un valore che mi fa da minimo/massimo e la sequenza è decrescente/crescente poi crescente/decrescente. In soldoni, il vettore **non è perfettamente ordinato**.

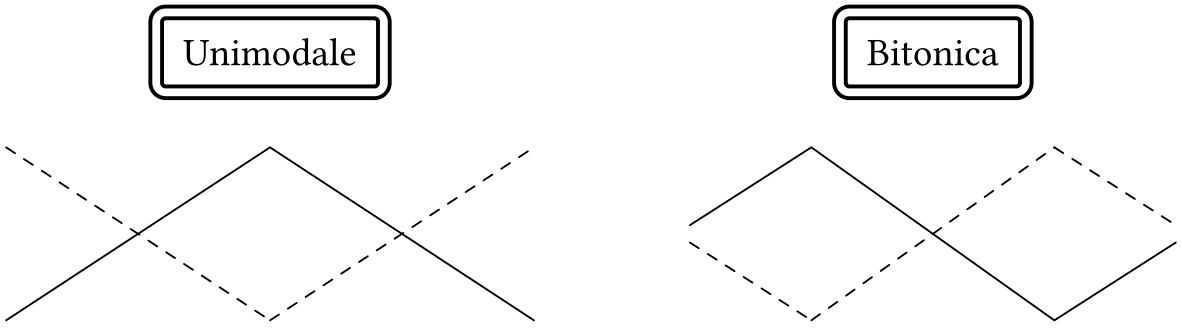
Definizione 7.2.2 (*Sequenza bitonica*): Una sequenza A è **bitonica** se e solo se esiste una permutazione ciclica di A che mi dà una sequenza unimodale, ovvero se

$$\exists k \in \{1, ..., n\} \mid A[k], ..., A[n], A[1], ..., A[k-1]$$

è una sequenza unimodale.

Graficamente, una sequenza unimodale ha un picco *massimo/minimo*, mentre una sequenza bitonica ha due picchi:

- un *minimo+massimo* con i valori della coda iniziale più piccoli dei valori della coda finale;
- un *massimo+minimo* con i valori della coda iniziale più grandi dei valori della coda finale.



Vediamo finalmente l'algoritmo per ordinare sequenze bitoniche, ideato da **Batcher** nel 1968.

Osserviamo che:

- una sequenza unimodale è anche bitonica, grazie alla permutazione identità;
- siano A e B due sequenze ordinate, allora $A \cdot \text{Reverse}(B)$ è unimodale.

Vediamo delle proprietà ora.

Lemma 7.2.1: Sia A una sequenza bitonica. Se eseguo MinMax su A ottengo:

- $A_{\min}$ e $A_{\max}$ bitoniche;

- ogni elemento di $A_{\min}$ minore di ogni elemento di $A_{\max}$.

Le osservazioni e le proprietà ci suggeriscono un approccio Divide et Impera:

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

Bitonic merge sequenziale

```
1: A_min | A_max = MinMax(A)
2: if |A| > 2
3:     BitonicMerge(A_min)
4:     BitonicMerge(A_max)
5: return A
```

Dobbiamo stare attenti a considerare **solo sequenze bitoniche**.

Teorema 7.2.1: L'algoritmo di bitonic merge è corretto.

Dimostrazione 7.2.1: Dimostriamolo per induzione su n.

Caso base: se $n = 2$ con MinMax scambio i due elementi se sono disordinati, poi ritorno il vettore A ordinato senza fare altro.

Passo induttivo: supponiamo vero per $n = 2^k$, mostriamo vero per $n = 2^{k+1}$

Calcolando MinMax su input di lunghezza 2^{k+1} otteniamo due sequenze di lunghezza 2^k, ma le sequenze di lunghezza 2^k bitonic merge le riesce a ordinare perfettamente per ipotesi induttiva. Grazie al lemma precedente, ogni elemento di $A_{\min}$ è minore di ogni elemento di $A_{\max}$, quindi il vettore A è ordinato. ■

Vediamo l'implementazione parallela di bitonic merge:

- eseguiamo MinMax su input di lunghezza

$$\frac{n}{2^{i-1}},$$

  partendo con $i = 1$, ottenendo di volta in volta due sequenze bitoniche $A_{\min}$ e $A_{\max}$ con i valori più piccoli nella prima e i valori più grandi nella seconda;
- al passo finale l'input è di lunghezza 2, quindi fermo le esecuzioni di MinMax;
- le sequenze di 2 elementi sono ordinate, e ora avviene una normalissima concatenazione di tutte le sequenze ottenute nei vari passi.

L'algoritmo è EREW perché lavora ogni volta su elementi diversi delle sequenze.

L'algoritmo termina i passi paralleli quando

$$\frac{n}{2^{i-1}} = 2 \Longrightarrow i = \log(n).$$

Il MinMax ha costo $T(n, p(n)) = 5$ quindi $T(n, p(n)) = 5\log(n)$. Il primo passo richiede $p(n) = \frac{n}{2}$ processori, il secondo $p(n) = \frac{n}{4} + \frac{n}{4} = \frac{n}{2}$ processori, eccetera. Il numero di processori che andiamo ad utilizzare è quindi $p(n) = \frac{n}{2}$.

Se vogliamo calcolare il tempo tramite l'equazione di ricorrenza, essa è

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

Non mettiamo costanti davanti $T(n)$ perché sto lavorando in parallelo. Il tempo che otteniamo, dopo aver risolto questa equazione di ricorrenza, è ancora $T(n, p(n)) = 5\log(n)$. L'efficienza vale

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

7.3. Bitonic sort generico

Quello che abbiamo fatto per ora è ordinare sequenze bitoniche in modo efficiente. Vediamo, per completezza, il bitonic sort di Batcher applicato a qualunque sequenza.

Bitonic sort generico sequenziale

```
1: A_min | A_max = MinMax(A)
2: if |A| > 2
3:     BitSort(A_min)
4:     BitSort(A_max)
5:     BitMerge(A_min · Reverse(A_max))
6: return A
```

Teorema 7.3.1: L'algoritmo di bitonic sort generico sequenziale è corretto.

Dimostrazione 7.3.1: Dimostriamolo per induzione su n.

Caso base: se $n = 2$ con MinMax scambio i due elementi se sono disordinati, poi ritorno il vettore A ordinato senza fare altro.

Passo induttivo: supponiamo vero per $n = 2^k$, mostriamo vero per $n = 2^{k+1}$

Calcolando MinMax su input di lunghezza 2^{k+1} otteniamo due sequenze di lunghezza 2^k, ma le sequenze di lunghezza 2^k bitonic sort le riesce a ordinare perfettamente per ipotesi induttiva. La chiamata finale a BitMerge avviene con una sequenza unimodale, che è al tempo stesso bitonica, ma sappiamo che BitMerge è corretto.

Quindi il vettore A è ordinato. ■

Vediamo l'implementazione parallela del bitonic sort:

- come prima, eseguiamo MinMax su input di lunghezza

$$\frac{n}{2^{i-1}},$$

  partendo con $i = 1$, ottenendo di volta in volta due sequenze $A_{\min}$ e $A_{\max}$;
- al passo finale l'input è di lunghezza 2, quindi fermo le esecuzioni di MinMax, visto che le sequenze di 2 elementi sono ordinate;
- presa una coppia di sequenze $A_{\min}$ e $A_{\max}$, applichiamo Reverse ad $A_{\max}$ e passiamo le due sequenze a BitMerge.

L'algoritmo è EREW perché lavoriamo sempre su dati diversi, usando letture e scritture esclusive.

Il tempo per la prima fase, ovvero la fase di applicazione di MinMax, è come quello del bitonic merge precedente, quindi mi fermo al passo $i = \log(n)$. Nella seconda fase ho sempre $\log(n)$ passi da moltiplicare per il costo del bitonic merge, che è $\log(n)$, quindi il costo è $\log^2(n)$.

Il tempo totale è quindi $T(n, p(n)) = \log^2(n)$. Per quanto riguarda i processori, ne abbiamo sempre $p(n) = \frac{n}{2}$, utilizzati per intero ad ogni passo dell'algoritmo.

Per il tempo possiamo usare anche l'equazione di ricorrenza, che è

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

ancora senza la costante sul $T(\frac{n}{2})$ perché sono in un ambiente parallelo.

L'efficienza per questo algoritmo vale

$$E = \frac{n\log(n)}{\frac{n}{2}5\log^2(n)} = \frac{C}{\log(n)} \longrightarrow 0$$

molto lentamente. Per questa sua proprietà, si preferisce usarlo su istanze molto piccole.

7.4. Osservazioni

Un buon algoritmo sequenziale non implica un buon algoritmo parallelo: un esempio è il merge sort, come abbiamo visto durante il tentativo di parallelizzazione. Ma anche un buon algoritmo parallelo non implica un buon algoritmo sequenziale: un esempio è il bitonic sort. Vediamo perché.

Per la routine di bitonic merge, il tempo è definito dall'equazione di ricorrenza

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

che, una volta risolta, ci dà tempo $T_b(n) = O(n\log(n))$.

Per la routine di bitonic sort, il tempo è definito dall'equazione di ricorrenza

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

che, una volta risolta, ci dà tempo $T_s(n) = O(n\log^2(n))$.

Vediamo come l'algoritmo parallelo sia molto buono perché ha tempo $T(n, p(n)) = O(\log^2(n))$ mentre l'algoritmo sequenziale ha tempo $T(n) = O(n\log^2(n))$, che è peggio del merge sort.

8. Navigazione di strutture connesse

8.1. Teoria dei grafi

Un **grafo diretto** D è una coppia (V, E) con:

- V insieme di **vertici**;
- $E \subseteq V^2$ insieme di **archi**, indicati con la notazione (s, d) oppure con un nome.

Un **cammino** è una sequenza di archi $[e_1, ..., e_k]$ tale che, per ogni coppia di lati consecutivi, il nodo pozzo (*destinazione*) del primo coincide con il nodo sorgente del secondo. Un **ciclo** è un cammino $[e_1, ..., e_k]$ nel quale il nodo pozzo di e_k è il nodo sorgente di e_1.

Definizione 8.1.1 (*Ciclo euleriano*): Un ciclo è **euleriano** quando ogni arco di E compare una e una sola volta nel ciclo.

La definizione è praticamente identica se parliamo di cammino euleriano. Un **grafo euleriano** è un grafo che contiene un ciclo euleriano. Dato un grafo D, ci possiamo chiedere se esso sia euleriano.

Diamo ancora qualche notazione:

- **grado di entrata**: dato $v \in V$ definiamo $\rho^-(v) = |\{(w, v) \in E\}|$ numero di archi entranti in v;
- **grado di uscita**: dato $v \in V$ definiamo $\rho^+(v) = |\{(v, w) \in E\}|$ numero di archi uscenti da v.

Teorema 8.1.1 (*di Eulero* (1736)): Un grafo D è euleriano se e solo se

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

Vediamo un problema simile a quello che abbiamo appena definito.

Definizione 8.1.2 (*Ciclo hamiltoniano*): Un ciclo è **hamiltoniano** se e solo se ogni vertice di V compare nel ciclo una e una sola volta.

Similmente a prima, D è un **grafo hamiltoniano** se e solo se contiene un ciclo hamiltoniano.

La richiesta di controllo dell'esistenza di un ciclo euleriano ammette un algoritmo efficiente con tempo $T(n) = O(n^3)$, con $n = |V|$, mentre il controllo dell'esistenza di un ciclo hamiltoniano è, purtroppo per noi, un problema NP-completo.

8.2. Alberi binari

Utilizzeremo i cicli euleriano per costruire algoritmi paralleli efficienti per **alberi binari**.

L'operazione fondamentale che useremo nei problemi è la **navigazione** dell'albero. Come possiamo fare una navigazione parallela efficiente?

Gli alberi spesso sono rappresentati come liste di puntatori, ma noi queste le sappiamo manipolare molto bene. Cercheremo quindi di trasformare queste strutture ad albero in liste a noi comode, così da comporre algoritmi paralleli efficienti che abbiamo già visto per risolvere i nostri nuovi problemi.

Il primo passo che facciamo è associare ad un albero binario un **ciclo euleriano**: sostituisco ogni arco dell'albero con un doppio arco orientato, così che possa effettivamente trovare un ciclo.

Ora trasformiamo questo ciclo in un cammino, espandendo ogni vertice v in una terna

$$(v, s) \mid (v, c) \mid (v, d).$$

Con questi nuovi vertici posso costruire un **cammino euleriano**:

- quando devo scendere di un livello collego il nodo corrente v al nodo figlio f nella sua componente sinistra (f, s);
- quando devo salire di un livello collego il nodo corrente v al nodo padre p nella sua componente centrale o destra, in base alla posizione del nodo v rispetto a p:
  - se v è il figlio sinistro di p mi collego al nodo (p, c);
  - se v è il figlio destro di p mi collego al nodo (p, d);
- quando non posso scendere di un livello scorro tutte le componenti del nodo corrente v.

Infine, devo costruire la lista

$$S[(v, x)] \quad | \quad v \in V \land x \in \{s, c, d\}.$$

Per costruire questa lista utilizziamo la **rappresentazione tabellare** dell'albero, ovvero una tabella che indica, per ogni vertice, chi sono il figlio sinistro, il figlio destro e il padre.

Se sono in un **nodo foglia** v allora

$$\begin{aligned} S[(v, s)] &= (v, c) \\ S[(v, c)] &= (v, d) \\ S[(v, d)] &= \begin{cases} (\text{pad}(v), c) & \text{se } v = \text{sin}(\text{pad}(v)) \\ (\text{pad}(v), d) & \text{se } v = \text{des}(\text{pad}(v)) \end{cases}. \end{aligned}$$

Se sono in un **nodo interno** v allora

$$\begin{aligned} S[(v, s)] &= (\text{sin}(v), s) \\ S[(v, c)] &= (\text{des}(v), s) \\ S[(v, d)] &= \begin{cases} (\text{pad}(v), c) & \text{se } v = \text{sin}(\text{pad}(v)) \\ (\text{pad}(v), d) & \text{se } v = \text{des}(\text{pad}(v)) \end{cases}. \end{aligned}$$

Diamo un algoritmo parallelo molto semplice per costruire questo vettore S:

- usiamo un processore per ogni vertice, ovvero per ogni riga della tabella;
- ogni processore per v deve costruire le celle $S[v, \{s, c, d\}]$.

L'algoritmo non è EREW, perché facciamo letture concorrenti quando leggiamo la nostra riga e la riga dei padri. Con un piccolo accorgimento, questa concorrenza può essere eliminata. Non vedremo come, possiamo solo trustare il processo (**JOEL EMBIID**).

Algoritmo diventa EREW, usando $p(n) = n$ processori con tempo $T(n, p(n)) = O(1)$. Con il **principio di Wyllie** otteniamo $p(n) = \frac{n}{\log(n)}$ processori e tempo $T = \log(n)$.

8.3. Attraversamento in pre-ordine

Definiamo, per ogni vertice $v \in V$, la quantità $N(v)$, che indica l'**ordine di attraversamento** di v durante una visita in **pre-ordine** dell'albero.

Definiamo un array A tale che

$$A[(v, x)] = \begin{cases} 1 & \text{se } x = s \\ 0 & \text{altrimenti} \end{cases}.$$

Sulla coppia (A, S) andiamo ad applicare l'algoritmo di somme prefisse. Dentro la cella $A[(v, s)]$ avremo $N(v)$: questo è vero perché quando facciamo il cammino e visitiamo un nuovo nodo andiamo sempre nel suo nodo sinistro.

L'algoritmo è EREW con $p(n) = \frac{n}{\log(n)}$ processori e tempo $T(n, p(n)) = \log(n)$. L'efficienza vale

$$E = \frac{n}{\frac{n}{\log(n)} \log(n)} \longrightarrow C \neq 0.$$

8.4. Profondità di un albero

Definiamo, per ogni vertice $v \in V$, la quantità $P(v)$, che indica la **profondità** di v nell'albero.

Definiamo un array A tale che

$$A[(v, x)] = \begin{cases} 1 & \text{se } x = s \\ 0 & \text{se } x = c. \\ -1 & \text{se } x = d \end{cases}$$

Anche sulla coppia (A, S) applichiamo l'algoritmo di somme prefisse. Otteniamo il valore $P(v)$:

- dentro la cella $A[(v, s)]$ se partiamo da 1 a contare le altezze;
- dentro la cella $A[(v, d)]$ se partiamo da 0 a contare le altezze.

L'efficienza vale

$$E = \frac{n}{\frac{n}{\log(n)} \log(n)} \longrightarrow C \neq 0.$$

Parte II — Algoritmi paralleli a memoria distribuita

1. Introduzione

Le **architetture parallele a memoria distribuita** era il paradigma utilizzato prima del multicore (*PRAM*), usato dai supercomputer più famosi come Cray, Intel Paragon, Blue Gene, Red Storm, Earth Simulator o Tianhe-2.

Queste architetture sono dei **grafi**, dove:

- i **nodi** sono dei processori, ovvero delle **RAM sequenziali** che hanno istruzioni per il calcolo e una memoria privata per effettuare calcoli; vedremo che questi nodi sono anche dei **router**;
- gli **archi** sono **reti di connessioni**.

La comunicazione avviene con le primitive **SEND** e **RECEIVE** in parallelo. **ATTENZIONE**: la comunicazione è in parallelo, ovvero le send e le receive avvengono in parallelo, ma un singolo processore lavora sequenzialmente quindi se arrivano k messaggi al processore P_i saranno necessarie k receive.

I collegamenti sono di tipo **full-duplex**, ovvero il grafo che stiamo considerando è non orientato.

Come nella PRAM, abbiamo un **clock** centrale che scandisce il tempo per tutti i processori.

Il programma, come nelle PRAM, utilizza il **passo parallelo** con architettura SIMD.

Passo parallelo

```
1: for i ∈ 𝕀 par do
2:     istruzione_k
```

Due sono i fattori che sono profondamente modificati:

- l'**input** non lo leggiamo più dalla memoria condivisa, come nella PRAM, ma lo dobbiamo distribuire tra i vari processori;
- l'**output** viene messo in un processore dedicato o si legge in un certo ordine tra i processori.

Le **risorse di calcolo** sono:

- **numero di processori**;
- **tempo di calcolo** e **tempo di comunicazione**, legato alla rete di connessioni.

Data l'architettura $G = (V, E)$, definiamo i seguenti **parametri di rete**:

- **grado** di G: definiamo

  $$\gamma = \max\{\rho(v) \mid v \in V\},$$

  dove $\rho(v)$ è il numero di archi incidenti su v. Un valore alto permette buone comunicazioni, ma rende più difficile la realizzazione fisica;
- **diametro** di G: definiamo

  $$\delta = \max\{d(v, w) \mid v, w \in V \land v \neq w\},$$

  dove $d(v, w)$ è la distanza minima tra i due vertici. Un valore basso è da preferire, ma aumenta il valore del parametro γ;
- **ampiezza di bisezione** di G: definiamo β come il minimo numero di archi in G che tolti da quest'ultimo mi dividono i nodi in circa due metà. Questa quantità rappresenta la capacità di trasferire le informazioni in G. Un valore alto di β alto è da preferire, ma aumenta ancora il valore del parametro γ.

2. Max e ordinamento

Due problemi che vedremo nelle prossime architetture sono **max** e **ordinamento**.

Il problema **Max** si risolve facendo comunicare ogni coppia di processori, così che ogni processore possa calcolare nella sua memoria il valore massimo della sequenza.

Lemma 2.1: Il tempo richiesto per Max in G è almeno δ.

Dimostrazione 2.1: Ogni coppia di processori deve comunicare, quindi anche i due processori a distanza massima, ma la distanza massima è il diametro δ. ■

Il problema **Ordinamento** si risolve trasferendo valori tra i processori per avere un ordinamento crescente. Vediamo un bound anche per questo problema.

Lemma 2.2: Il tempo richiesto per Ordinamento in G è almeno

$$\frac{n}{2\beta}.$$

Dimostrazione 2.2: Dividiamo il grafo in due metà:

- in $\frac{n}{2}$ nodi ho i numeri più alti;
- in $\frac{n}{2}$ nodi ho i numeri più bassi.

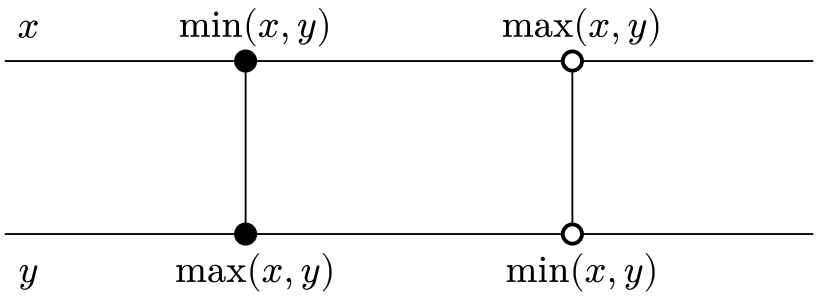
Il caso peggiore che possiamo avere è una sequenza ordinata in modo decrescente. Devo quindi scambiare tutte le posizioni delle due zone.

Posso trasferire da una zona all'altra usando i β ponti che definiscono la ampiezza di bisezione. Dovendo trasferire $\frac{n}{2}$ valori usando β ponti, il numero di iterazioni è

$$\frac{n}{2\beta}.$$ ■

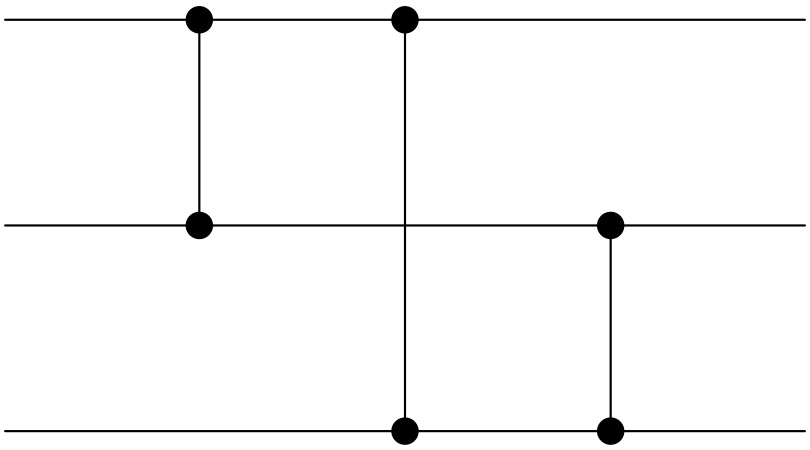
Per analizzare questi problemi abbiamo bisogno di una serie di oggetti molto carini, i **confrontatori** (*comparatori*), e anche delle loro primitive. Questi oggetti sono dei **ponti** che collegano due fili; una volta che il confrontatore prende in input i valori dei due fili, in quello sopra mette il **valore minimo** e in quello sotto mette il **valore massimo**.

Ci sono alcuni confrontatori che invertono l'ordine dei due fili. Come possiamo distinguere le due tipologie? Se il confrontare lavora come piace a noi allora il cerchio utilizzato è pieno, altrimenti il cerchio utilizzato è vuoto.

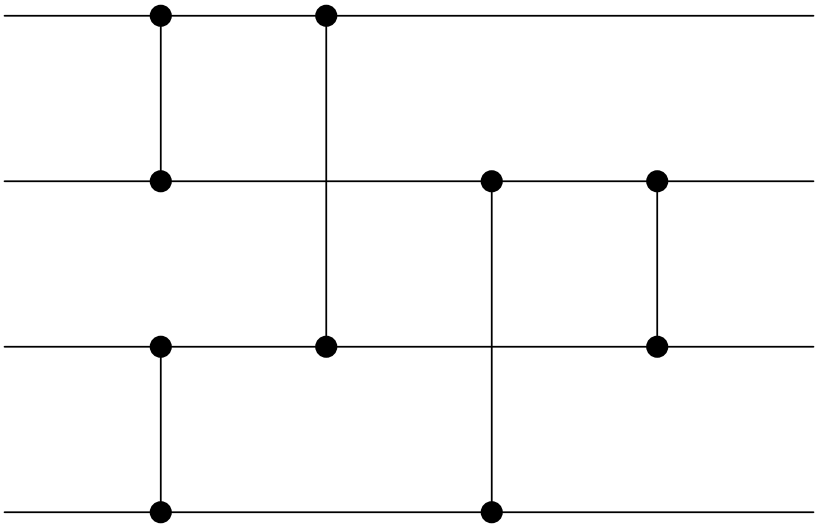


Con i confrontatori possiamo create delle **reti di confrontatori**, ovvero delle reti che spostano sopra e sotto i valori inseriti nei fili. Una **sorting network** è una rete di confrontatori capace di ordinare una sequenza di valori contenuta nei fili.

Qua sotto vediamo un esempio di sorting network per input di grandezza $n = 3$.



Mentre qua sotto vediamo un esempio di sorting network per input di grandezza $n = 4$.



L'idea per un algoritmo parallelo è quella di raggruppare, in un passo, i confrontatori che agiscono su fili diversi, così da evitare accessi concorrenti allo stesso dato e controlli complicati.

Una **rete di confrontatori** la indichiamo con

$$R(x_1, ..., x_n) = (y_1, ..., y_n).$$

La rete R è una **sorting network** se e solo se

$$\forall (x_1, ..., x_n) \in \mathbb{N}^n \quad R(x_1, ..., x_n) = (y_1, ..., y_n)$$

con

$$y_1 < ... < y_n.$$

Queste reti sono anche dette **reti di ordinamento test/swap oblivious**. Quest'ultimo aggettivo deriva dal fatto che i confronti non dipendono dall'input dato, ma sono fissati a priori.

Per sapere se una rete di confrontatori R è una sorting network possiamo utilizzare il **principio 01**, ideato da **Donald Knuth** (*mio fratello*) nel 1972.

Teorema 2.1 (*Principio 01*): Vale

$$\forall x \in \{0,1\}^n \ R \text{ ordina } x \longrightarrow \forall y \in \mathbb{N}^n \ R \text{ ordina } y.$$

In poche parole, se riesco ad ordinare ogni possibile vettore booleano allora riesco ad ordinare ogni possibile vettore intero. Questo è comodo perché i vettori booleani sono molto meno di quelli interi.

Molto comoda anche l'implicazione inversa, ovvero

$$\exists y \in \mathbb{N}^n \mid R \text{ non ordina } y \Longrightarrow \exists x \in \{0,1\}^n \mid R \text{ non ordina } x.$$

Una cosa molto comoda dei confrontatori è inoltre la **linearità** rispetto ad una funzione f. Spieghiamo meglio: sia f una **funzione monotona crescente**. Allora

$$R(f(x_1), ..., f(x_n)) = f(R(x_1, ..., x_n)) = (f(y_1), ..., f(y_n)).$$

Questo strumento è detto **f-shift** su R. Cosa abbiamo mostrato? Abbiamo fatto vedere che:

- applicare R ad x e poi applicare f OPPURE
- applicare f ad x e poi applicare R

mi genera lo stesso risultato.

Teorema 2.2: Se R è una rete non corretta allora

$$\exists x \in \mathbb{N}^n \mid R \text{ non ordina } x.$$

In poche parole, esistono due indici t, s tali che

$$y_t > y_s \wedge t < s.$$

Dimostrazione 2.3: Definiamo la funzione $g : \mathbb{N} \longrightarrow \{0,1\}$ tale che

$$g(x) = \begin{cases} 1 & \text{se } x \geq y_t \\ 0 & \text{altrimenti} \end{cases}.$$

Questa funzione è monotona crescente: vale 0 fino a y_t (*escluso*) poi vale 1 dai valori successivi.

Vado ad applicare g alla rete R ottenendo

$$R(g(x_1), ..., g(x_n)).$$

Per la regola dello shift questa quantità è

$$(g(y_1), ..., g(y_n)).$$

Questo vettore binario non è ordinato perché $g(y_t) = 1$ mentre $g(y_s) = 0$. Ma allora la rete R non ha ordinato la nostra sequenza. ■

3. Array lineari

La prima architettura parallela a memoria distribuita che vediamo sono gli **array lineari**. Questa architettura è formata da n processori P_i collegati in sequenza. I parametri di questa rete sono:

- **grado** $\gamma = 2$, ottimo per la realizzazione;
- **diametro** $\delta = n - 1$, lower bound per Max quindi non sono soddisfatto;
- **ampiezza di bisezione** $\beta = 1$.

Ricordiamoci che sulla PRAM avevamo:

- Max risolto con $p(n) = \frac{n}{\log(n)}$ processori in tempo $T = \log(n)$;
- Ordinamento risolto con $p(n) = n$ processori in tempo $T = \log(n)$.

3.1. Shuffle

La prima primitiva che vogliamo trovare è **shuffle**.

Per ora noi sappiamo fare molto bene lo **swap contiguo**, ovvero il processore P_k deve avere il dato $A[k+1]$ mentre il processore P_{k+1} deve avere il dato $A[k]$.

Avrò bisogno di 3 passi paralleli:

- doppia send per mandare:
  - $A[k]$ a P_{k+1};
  - $A[k+1]$ a P_k;
- doppia receive per ricevere il dato;
- assegnare i dati appena ricevuti.

Il problema shuffle:

- prende in **input** un numero pari di valori $A[1], ..., A[s], A[s+1], ..., A[2s]$;
- restituisce in **output** la sequenza $A[1], A[s+1], A[2], A[s+2], ..., A[s], A[2s]$.

Ci servirà l'operazione di swap che abbiamo appena visto: l'idea per l'algoritmo parallelo scambia i due elementi centrali, poi i due elementi appena a sinistra/destra, poi eccetera.

Questo albero di swap contigui ci richiedono $p(s) = 2(s-1)$ processori con un utilizzo di tempo $T(s, p(s)) = 3(s-1)$. Il migliore algoritmo sequenziale richiede tempo $\Theta(s^2)$, quindi l'efficienza vale

$$E = \frac{s^2}{2(s-1) \cdot 3(s-1)} = C \neq 0.$$

3.2. Max

Per la primitiva **max** dobbiamo mandare il dato di un certo processore a tutti gli altri. La primitiva **SEND** esegue un numero di passi uguale alla distanza $d(i, j)$ tra i due nodi, e questa distanza è proprio un costo del problema.

Un processore P_i, per mandare un dato al processore P_j, effettua una SEND, poi avvengono una serie di RECEIVE-SEND, e infine P_j effettua una RECEIVE. Il numero totale di operazioni è

$$2d(i, j) = 2|i - j|.$$

La prima cosa che notiamo è che la trasmissione di un dato non è più costante, come nelle PRAM.

Il problema Max:

- prende in **input** una serie di valori $A[1], ..., A[n]$;
- restituisce in **output**, nel processore P_n, il valore $\max\{A[i] \mid 1 \leq i \leq n\}$.

Il tempo per Max su array lineari è limitato inferiormente da $\delta = n - 1$, che è ben peggiore del tempo $\log(n)$ che avevamo prima nelle PRAM.

L'idea per un algoritmo parallelo per Max considera l'algoritmo sommatoria delle PRAM e cerca di abbassare il numero di processori per averne $\log(n)$. Come facciamo?

Al j-esimo passo confrontiamo i numeri a distanza 2^{j-1}, selezioniamo il massimo e lo memorizziamo nel processore di indice massimo. Il numero di passi che eseguiamo è $\log(n)$.

Max parallelo

1: for $j = 1$ to $\log(n)$
2: | for $k \in \left\{2^j t - 2^{j-1} \mid 1 \leq t \leq \frac{n}{2^j}\right\}$ par do
3: | | Send$(k, k + 2^{j-1})$
4: | for $k \in \left\{2^j t \mid 1 \leq t \leq \frac{n}{2^j}\right\}$ par do
5: | | if $(A[k] < A[k - 2^{j-1}])$ then
6: | | | $A[k] = A[k - 2^{j-1}]$

Vediamo il tempo impiegato dalle due fasi:

- la fase di **send** è 2 volte la distanza tra i processori, quindi $2 \cdot 2^{j-1} = 2^j$;
- la fase di **compare** vale 2, perché faccio solo un confronto e un assegnamento.

Il tempo che abbiamo appena visto deve essere eseguito per ogni passo j, quindi

$$\sum_{j=1}^{\log(n)} 2^j + 2 = \sum_{j=1}^{\log(n)} 2^j + 2\log(n) = \frac{2^{\log(n)+1} - 1}{2 - 1} \underbrace{-1}_{\text{non parto da 0}} + 2\log(n) =$$
$$= 2n - 2 + 2\log(n) = O(n).$$

Utilizzando $p(n) = n$ processori, l'efficienza vale

$$E = \frac{n}{n \cdot n} \longrightarrow 0.$$

Questo non ci piace: riduciamo il numero di processori da n a p. Con questo accorgimento operiamo sul parametro δ della nostra rete, visto che non abbiamo più n processori in fila ma p. Ogni processore ora prende $\frac{n}{p}$ elementi, sui quali viene calcolato il Max sequenziale in tempo $\frac{n}{p}$. Su questi massimi poi viene eseguito Max parallelo.

I processori ora sono $p(n) = p$, utilizzati in tempo $T(n, p(n)) = O\left(\frac{n}{p}\right) + O(p)$. L'efficienza vale

$$E = \frac{n}{p\left(\frac{n}{p} + p\right)} = \frac{n}{n + \underbrace{p^2}_{n}} = \frac{1}{2} \neq 0.$$

Per avere questa efficienza dobbiamo imporre $n = p^2$, ovvero $p = \sqrt{n}$.

Con questo valore di p, i processori sono $p(n) = \sqrt{n}$ e il tempo è $T(n, p(n)) = O(\sqrt{n})$.

3.3. Ordinamento

Per risolvere **ordinamento** ci serve una primitiva di swap che lavora tra due processori generici, e non contigui come nella shuffle. Qua abbiamo diverse opzioni per questa primitiva.

La prima soluzione (*peggiore*) esegue due send in sequenza, una (i, j) e una (j, i), e poi un assegnamento parallelo. Questo viene eseguito in tempo $T(n, p(n)) = 4d(i, j) + 1$, non va bene.

La seconda soluzione (*mid*) esegue le send in simultaneo, ma qui abbiamo due casi:

- se la distanza tra i processori è **dispari**, ovvero è $2k+1$, i processori impiegati sono $2k+2$, divisi in due metà da $k+1$ processori. I processori centrali hanno indici $k+1$ e $k+2$, che ricevono in simultanea i dati dai due bordi. La send tra questi due processori avviene in simultaneo, tanto abbiamo delle connessioni full duplex, e poi mandiamo i dati ricevuti ai bordi.
- se la distanza tra i processori è **pari**, ovvero è $2k$, i processori impiegati sono $2k+1$, divisi in due metà da k processori con un processore di indice $k+1$ centrale. La send dai bordi fino al processore $k+1$ arrivano in contemporanea, ma la receive è sequenziale quindi impiega tempo 2 prima di poter mandare i dati ricevuti ai bordi.

Il tempo nel primo caso è

$$T(n, p(n)) = 2k + 2 + 2k + 1 = 4k + 2 + 1 = 2(2k + 1) + 1 = 2d(i, j) + 1$$

mentre nel secondo caso il tempo è

$$T(n, p(n)) = 2k + 1 + 2k + 1 + 1 = 4k + 3 = 2(2k) + 3 = 2d(i, j) + 3.$$

L'altra primitiva per l'ordinamento che ci serve è il **minmax** tra celle contigue, che assegna al processore di indice minimo il valore minimo e al processore di indice massimo il valore massimo. Questa primitiva in poche parole implementa il comportamento di un **confrontatore**, ovvero

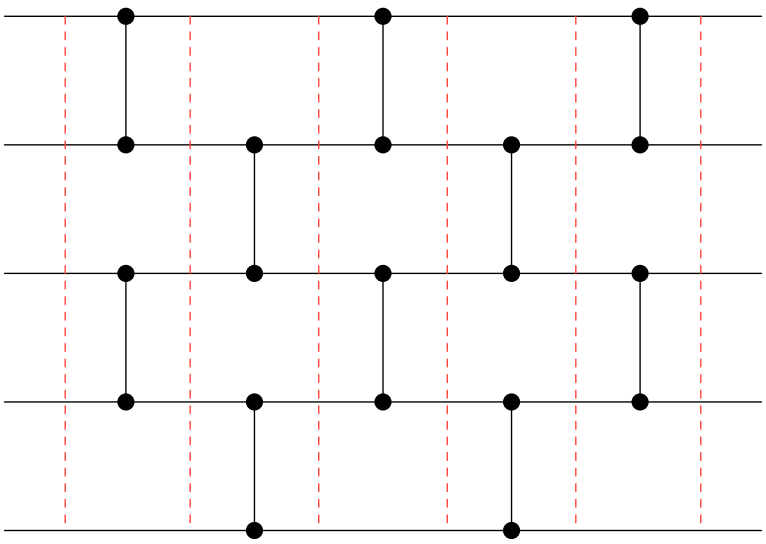
$$P_k = \min\{A[k], A[k+1]\} \wedge P_{k+1} = \max(A[k], A[k+1]).$$

Ricordiamoci che la MinMax ha 5 come costo del passo parallelo.

Il problema ordinamento:
- prende in **input** una serie di valori $A[1], ..., A[n]$ assegnati ai processori $P[1], ..., P[n]$;
- restituisce in **output** i valori dei processori ordinati in senso crescente.

Usiamo una sorting network chiamata **ODD/EVEN**. Numeriamo le righe dei dati dall'alto a partire da 1. La SN ha esattamente n step di confrontatori, formati da un'alternanza di colonne di confrontatori dispari a colonne di confrontatori pari. Un **confrontatore dispari** è un confrontatore che inizia su una riga dispari e finisce sulla riga pari sottostante. Un **confrontatore pari** ha la definizione analoga ma adattata.

Qua sotto vediamo un esempio di ODD/EVEN per una rete di grandezza $n = 5$.



Teorema 3.3.1: ODD/EVEN è corretto.

Dimostrazione 3.3.1: La dimostrazione viene fatta con il **principio 01**.

Diamo $\{0,1\}^n$ in pasto ad ODD/EVEN, ottenendo 0^j1^e tale che $j+e=n$, facendo esattamente n round di confrontatori.

Nel caso peggiore, avendo tutti gli 1 ad inizio sequenza, ogni 1 deve scendere di $n-e=j$ posizioni. Inoltre, prima di effettuare la discesa, ogni 1 fa anche un «*ritardo*» (***momento Trenord***) in base alla sua posizione i nella sequenza.

Il numero di passi impiegato è al massimo $n-e+i$, ma visto che $i \leq e$ otteniamo al più

$$n-e+e=n$$

passi. Ma allora n passi sono necessari e sufficienti. ■

Il numero di passi è necessario (*visto nell'esempio che non ho messo*) e sufficiente (*dimostrazione*).

Il tempo per implementare questo algoritmo sequenzialmente è $T(n)=n\cdot\frac{n}{2}\approx n^2$.

Per l'implementazione parallela vediamo l'idea di **Haberman** del 1972.

Ordinamento parallelo

```
1: for i = 1 to n
2:     for k ∈ {2t − (i % 2) | 1 ≤ t ≤ n/2} par do
3:         MinMax(k, k + 1)
```

Stiamo utilizzando $p(n)=n$ processori in tempo $T(n,p(n))=4n$. L'efficienza vale

$$E=\frac{n\log(n)}{n\cdot 4n}\longrightarrow 0.$$

Ricordiamoci che con $\beta=1$ il minimo tempo per l'ordinamento è $\frac{n}{2}$, e questo non va bene perché noi stiamo eseguendo con tempo n. Riduciamo i processori a p con il **principio di Wyllie**: ogni processore ora prende $\frac{n}{p}$ dati e li ordina sequenzialmente in un tempo $O\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)$.

Questa versione dell'algoritmo è una versione che non usa minmax ma **merge-split**. Questa operazione avviene tra due processori contigui e agisce come segue:

- entrambi i processori ricevono $\frac{n}{p}$ dati e li ordinano in tempo $O\left(\frac{n}{p}\log\left(\frac{n}{p}\right)\right)$:
- il processore di SX spedisce $\frac{n}{p}$ dati ordinati al processore di DX in un tempo $O\left(\frac{n}{p}\right)$;
- il processore di DX riceve e fonde (*merge*) i nuovi dati con i suoi $\frac{n}{p}$ ordinati in tempo $O\left(\frac{n}{p}\right)$;
- il processore di DX invia (*split*) gli $\frac{n}{p}$ dati più piccoli al processore di SX in un tempo $O\left(\frac{n}{p}\right)$.

Ordinamento parallelo con merge-split

```
1: for i = 1 to p
2:     for k ∈ {2t − (i % 2) | 1 ≤ t ≤ p/2} par do
3:         MergeSplit(k, k + 1)
```

Ordinamento parallelo con merge-split

Usando $p(n) = p$ processori con tempo $T(n, p(n)) = \frac{n}{p}\log\left(\frac{n}{p}\right) + p \cdot \frac{n}{p} = \frac{n}{p}\log\left(\frac{n}{p}\right) + n$, l'efficienza che otteniamo vale

$$E = \frac{n\log(n)}{p \cdot \left(\frac{n}{p}\log\left(\frac{n}{p}\right) + n\right)} = \frac{n\log(n)}{n\log\left(\frac{n}{p}\right) + \underbrace{np}_{n\log(n)}} = C \neq 0.$$

Per avere questo dobbiamo imporre $np = n\log(n)$ e quindi $p = \log(n)$.

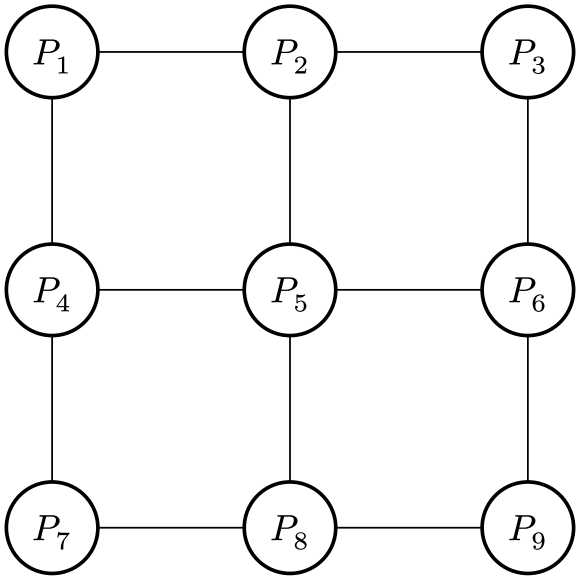
Il tempo rimane comunque un $O(n)$ perché la riduzione dei processori abbassa il diametro del grafo ma non modifica l'ampiezza di bisezione, che rimane sempre $\beta = 1$.

4. Architettura MESH

Le **MESH** sono un'architettura parallela a memoria distribuita che rappresenta i processori come un array bidimensionale a griglia. Avendo a disposizione n processori, la MESH è un quadrato $m \times m$ con $m = \sqrt{n}$. Se n non è un quadrato perfetto allora prendiamo $m = \left(\lfloor \sqrt{n} \rfloor + 1\right)^2$, che per $n \geq 6$ ci dà un valore $\leq 2n$, che ci va molto bene.

I **parametri** di rete di questa architettura sono:

- **grado** $\gamma = 4$;
- **diametro** $\delta = 2\sqrt{n}$
- **ampiezza di bisezione** $\beta \sim \sqrt{n}$ in base alla parità di m.



I nostri lower bound diventano:

- $\Omega(\sqrt{n})$ per **Max**;
- $\Omega(\sqrt{n})$ per **Ordinamento**.

4.1. Max

L'idea per un algoritmo parallelo è quella di usare la MESH come se fosse un array lineare, andando *«a serpentello»* nelle varie celle, ma un array lineare ha come tempo minimo n e noi invece abbiamo $\sqrt{n}$, quindi questa idea è cestinata immediatamente.

Usiamo un algoritmo **righe-colonna**: abbiamo a disposizione tante connessioni, perché non usarle?

Consideriamo ogni riga della MESH come se fosse un array lineare di $\sqrt{n}$ processori. Il massimo di ogni riga verrà messo nell'ultimo processore, e sulla colonna formata dai *«processori massimi»* eseguiamo ancora un'esecuzione di Max su array lineari.

Max righe-colonna

1: for $i = 1$ to m par do
2: ⌊ $\text{Max}(P_{i1}, ..., P_{im})$
3: $\text{Max}(P_{1m}, ..., P_{mm})$

Il tempo, per entrambe le parti, è $T(n, p(n)) = O(\sqrt{n})$. L'efficienza purtroppo è

$$E = \frac{n}{n\sqrt{n}} \longrightarrow 0.$$

Riduciamo il numero di processori con il **principio di Wyllie** da n a p, dove ognuno di questi calcola il massimo di $\frac{n}{p}$ dati in tempo $O\left(\frac{n}{p}\right)$. Poi, si attiva l'algoritmo di prima sulla MESH $\sqrt{p} \times \sqrt{p}$, che viene eseguito però in un tempo minore, ovvero $T(n, p(n)) = O(\sqrt{p})$.

Il tempo totale di questo nuovo algoritmo è è $T(n, p(n)) = \frac{n}{p} + \sqrt{p}$. L'efficienza vale

$$E = \frac{n}{p\left(\frac{n}{p} + \sqrt{p}\right)} = \frac{n}{n + \underbrace{p\sqrt{p}}_{n}} = C \neq 0.$$

Per avere questa efficienza scelgo $p\sqrt{p} = n$, ovvero $p = n^{\frac{2}{3}}$.

Con questa scelta di p il tempo totale diventa $T(n, p(n)) = O(\sqrt[3]{x})$. E questo valore è ottimo: usando una MESH di dimensione $\sqrt{p}$ il limite teorico per Max è $\sqrt{p} = \sqrt{n^{\frac{2}{3}}} = \sqrt[3]{x}$, che è esattamente il valore che abbiamo ottenuto noi, quindi siamo soddisfatti.

4.2. Ordinamento

L'algoritmo di **ordinamento LS3** deve il suo nome a due ricercatori degli anni '80.

L'algoritmo che utilizziamo è ricorsivo con **Divide et Impera**

Sia M il quadrato dei processori. L'algoritmo procede come segue:

- **dividi**: M viene diviso in 4 quadrati $M_1 \mid M_2 \mid M_3 \mid M_4$ di dimensione $\frac{m}{2}$ con $m = \sqrt{n}$. Questi sono ottenuti da M dividendolo come se fossero i quadranti di un piano cartesiano, ovvero

$$\begin{pmatrix} M_1 & M_2 \\ M_3 & M_4 \end{pmatrix};$$

- **ordina**: i quadrati M_i vengono ordinati *«a serpentello»* in parallelo;
- **fondi**: i quadrati M_i vengono rimessi nelle loro posizioni originali.

LS3sort

```
1: if |M| == 1
2:     return M
3: else
4:     LS3sort(M_1)
5:     LS3sort(M_2)
6:     LS3sort(M_3)
7:     LS3sort(M_4)
8:     LS3merge(M_1, M_2, M_3, M_4)
```

Per implementare questo algoritmo ci servono shuffle e ODD/EVEN.

Data una riga i della MESH, essa è un **array lineare** di processori. Su questa riga eseguiamo lo **shuffle**. Questa operazione alterna elementi della matrice M_1 con elementi della matrice M_2 (*sopra*) ed elementi della matrice M_3 con elementi della matrice M_4 (*sotto*).

Dopo lo shuffle devo eseguire **ODD/EVEN** tra due colonne adiacenti i e $i+1$. Prendiamo le due colonne assieme e creiamo un **array lineare** andando *«a serpentello»*. Il numero di round di questo algoritmo è uguale alla grandezza di questo array, che è $2\sqrt{n}$.

LS3merge

1: for $i = 1$ to $\sqrt{n}$ par do
2: ⌊ SHUFFLE(i)
3: for $i = 1$ to $\frac{\sqrt{n}}{2}$ par do
4: ⌊ OddEven$(2i-1, 2i)$
5: Esegui i primi $2\sqrt{n}$ passi di ODD/EVEN sull'intera mesh a serpente

Il tempo per questo algoritmo è:
- $O(\sqrt{n})$ per lo shuffle;
- $O(\sqrt{n})$ per ODD-EVEN;
- $O(\sqrt{n})$ per l'ultima esecuzione.

Ma allora tempo totale della merge è $T_m(n) = h\sqrt{n}$

Risolviamo l'equazione di ricorrenza per sto schifo:

$$T(n) = \begin{cases} 1 & \text{se } n = 1 \\ T(\frac{n}{4}) + h\sqrt{n} & 6 \text{ altrimenti} \end{cases}$$

Ma allora

$$\begin{aligned} T(n) &= T\left(\frac{n}{4}\right) + h\sqrt{n} = T\left(\frac{n}{4^2}\right) + h\sqrt{\frac{n}{4}} + h\sqrt{n} = T\left(\frac{n}{4^3}\right) + h\sqrt{\frac{n}{4^2}} + h\sqrt{\frac{n}{4}} + h\sqrt{n} = \\ &= \text{mi fermo quando } M_i \text{ è grande } 4 = \\ &= \sum_{i=0}^{\log_4(n)-1} h\sqrt{\frac{n}{4^i}} + 1 = \\ &= h\sqrt{n} \sum_{i=0}^{\log_4(n)-1} \sqrt{\frac{1}{4^i}} + 1 = \\ &= h\sqrt{n} \sum_{i=0}^{\frac{\log_2(n)}{\log_2(4)}-1} \frac{1}{2^i} + 1 = \\ &= h\sqrt{n} \sum_{i=0}^{\frac{\log(n)}{2}-1} \left(\frac{1}{2}\right)^i + 1 = \\ &= h\sqrt{n}\left(\frac{1-\left(\frac{1}{2}\right)^{\frac{\log(n)}{2}-1+1}}{\frac{1}{2}}\right) + 1 = 2h\sqrt{n}\left(1 - \frac{1}{\sqrt{n}}\right) + 1 = O(\sqrt{n}). \end{aligned}$$

Nel caso sequenziale, aggiungendo un 4 all'equazione di ricorrenza, otteniamo tempo $T(n) = n\sqrt{n}$, un tempo peggiore del merge sort.

Per il caso parallelo, abbiamo $p(n) = n$ processori con tempo $T(n, p(n)) = O(\sqrt{n})$. L'efficienza è

$$E = \frac{n \log(n)}{n\sqrt{n}} \longrightarrow 0.$$

Non ci piace molto come valore, possiamo migliorare l'efficienza riducendo i processori, ma non lo vedremo. Non vedremo nemmeno una versione del bitonic sort (*bentornato tra noi*) su MESH che usa $p(n) = O(\log^2(n))$ processori in tempo $T(n, p(n)) = O\left(\frac{n}{\log(n)}\right)$ con una buonissima efficienza, ma vediamo che come tempo è peggiore di LS3.

Parte III — Algoritmi distribuiti

1. Introduzione

Le **architetture distribuite** sono rappresentate da **grafi orientati** dove:

- i **nodi** sono le entità del sistema, dotate di memoria locale, capacità di calcolo, capacità di comunicazione e un clock locale proprio;
- gli **archi** sono link/connessioni, non per forza full-duplex.

Vediamo come non abbiamo più un **clock globale**, ma ogni nodo pensa a se stesso.

Nella **memoria locale** di ogni processore abbiamo il **registro di input**, identificato da valore(x), e il **registro di stato**, identificato da stato(x). Il primo registro identifica il valore di input dell'entità x, mentre il secondo registra il *valore attuale* dell'entità x, e questo valore è cambiato localmente dalla stessa x.

Le entità hanno una serie di **proprietà**:

- **sono reattive**: all'accadere di un **evento** compiono una **azione**. Definiamo questi due concetti:
  - **eventi**: ciò che succede alle entità, essi possono essere:
    - **interni al sistema**: ricezione di messaggi;
    - **esterni al sistema**: impulso spontaneo (*START*);
  - **azioni**: sequenza finita di operazioni indivisibili che sono eseguite in risposta ad un evento. Con *«indivisibili»* si intende un'azione che inizia e viene portata a termine. Un'azione particolare è **nil**, che indica l'azione vuota.
- **seguono delle regole**: una **regola** è un oggetto della forma

$$\text{STATO} \times \text{EVENTO} \longrightarrow \text{AZIONE} .$$

  Sia x una entità, definiamo con $B(x)$ l'insieme delle regole a cui è soggetta x. Questo insieme deve essere **completo** e **non ambiguo**. In poche parole, $B(x)$ rappresenta il codice di x.

Sia E l'insieme delle entità che cooperano tra loro. Allora

$$B(E) = \bigcup_{x \in E} B(x)$$

rappresenta il **comportamento del sistema**, ed è importante che sia **omogeneo**, ovvero

$$\forall x, y \in E \quad B(x) = B(y).$$

In poche parole, il codice/protocollo/algoritmo distribuito deve essere uguale per tutte le entità.

Lemma 1.1: È sempre possibile ottenere un insieme $B(E)$ omogeneo.

Dimostrazione 1.1: L'idea della dimostrazione è utilizzare un registro locale aggiuntivo che differenzia quelle entità che alla stessa coppia (STATO $\times$ EVENTO) hanno AZIONE diversa. Questo è il registro ruolo(x), che rappresenta appunto il ruolo di x. La regola viene modificata in

$$\text{STATO} \times \text{EVENTO} \longrightarrow \{\text{if ruolo}(x) = a \text{ then AZIONE}_a \text{ else AZIONE}_b\}. \qquad \blacksquare$$

La rete ha una serie di **parametri**: essi sono il **numero di entità** n, il **numero di link** m e il **diametro** della rete d, uguale a quello che avevamo nelle architetture precedenti.

La comunicazione avviene usando una **etichettatura** sui link. Per l'entità x, l'etichettatura è denotata con λ_x e poi l'arco che abbiamo davanti. Indichiamo inoltre con:

- $N_{\text{in}}(x)$ insieme dei vicini di ingresso ad x;
- $N_{\text{out}}(x)$ insieme dei vicini di uscita di x;

La rete ha una serie di **assiomi**:

- **ritardo finito di comunicazione**: in assenza di errori, un messaggio spedito prima o poi arriverà, non sappiamo quando ma arriverà;
- **orientamento locale**, ogni entità riesce a distinguere tra i suoi vicini gli insiemi $N_{\text{in}}(x)$ e $N_{\text{out}}(x)$ grazie alla conoscenza di λ_x;

La rete, inoltre, può essere utilizzata con delle **restrizioni**: esse sono dichiarate al momento della scrittura del codice e sono delle **proprietà positive** della rete su cui facciamo affidamento. Vediamone alcune di quelle più usate:

- **restrizioni sulla comunicazione**:
  - **link bidirezionali**: le connessioni diventano full-duplex, ovvero $N_{\text{in}}(x) = N_{\text{out}}(x)$ e anche $\lambda_x(x, y) = \lambda_x(y, x)$. In questo caso indichiamo i vicini di x semplicemente con $N(x)$;
  - **ordinamento dei messaggi**: i messaggi su un link vengono prelevati con la politica **FIFO**;
- **restrizioni sull'affidabilità**:
  - **rilevazione di errori**: entità e link possono rilevare gli errori;
  - **affidabilità parziale**: non ci saranno errori in futuro;
  - **affidabilità totale**: non ci sono stati errori prima e non ce ne saranno in futuro.
- **restrizioni sulla topologia di rete**:
  - **connettività del grafo**: imponiamo un grafo fortemente connesso (*se il grafo è orientato*) oppure un grafo connesso (*se il grafo è non orientato*);
- **restrizioni sul tempo**:
  - **tempi di comunicazione unitari**: il nome parla da sé;
  - **clock sincronizzati**: i clock delle entità scattano tutti allo stesso tempo.

Tali restrizioni a volte vengono considerate per il calcolo delle **prestazioni ideali** del codice distribuito. Le misure che possiamo fare prendono in considerazione:

- **tempo**: intervallo tra la prima entità che si attiva e l'ultima che termina;
- **quantità di comunicazione**: numero di messaggi spediti e/o numero di bit spediti.

Il tempo, così come lo usiamo di solito, non va più bene: esecuzioni diverse dello stesso codice distribuito possono portare a tempi diversi. Risolviamo questo problema con il **tempo ideale**: esso è il tempo misurato considerando comunicazioni unitarie e clock sincronizzati.

Il **tempo causale** (*tempo del caso peggiore*) è invece il tempo misurato considerando la catena più lunga di comunicazione richiesta dal codice. A differenza del tempo ideale, questa quantità è molto difficile da calcolare.

Definiamo quello che è un problema. Un **problema** è una tripla

$$\langle P_{\text{init}}, P_{\text{final}}, R \rangle$$

dove:

- P_{init} rappresenta i predicati che descrivono il sistema all'avvio;
- P_{final} rappresenta i predicati che descrivono il sistema alla sua terminazione;
- R rappresenta le restrizioni del sistema.

L'insieme P_{init} contiene tutti i valori iniziali di stato che possiamo trovare nel sistema al suo avvio, ovvero abbiamo che

$$\forall x \in E \quad \text{valore}(x) \in P_{\text{init}}.$$

Una definizione simile si può dare anche per P_{final}.

L'esecuzione di un protocollo genera una **sequenza di configurazioni** successive del sistema.

Sia $\Sigma(t)$ il contenuto dei registri delle entità al tempo t, e sia $\text{futuro}(t)$ l'insieme degli eventi già generati al tempo t ma che non sono ancora stati processati.

Indichiamo con $C(t)$ la **configurazione del sistema** del tempo t, definita dalla coppia

$$(\Sigma(t), \text{futuro}(t)).$$

Definiamo

$$C(0) = (\Sigma(0), \text{futuro}(0))$$

la **configurazione iniziale** che contiene i registri inizializzati e l'impulso spontaneo. Parallelamente, si può definire $C(f)$ come la **configurazione finale**.

Dentro $C(0)$ troveremo tutti gli stati presenti in S_{init}, insieme degli stati che troviamo all'avvio, mentre dentro $C(f)$ troveremo tutti gli stati presenti in S_{term}, insieme degli stati che troviamo alla fine.

Per finire, definiamo quali saranno le restrizioni che useremo praticamente sempre:

- link bidirezionali **BL**;
- affidabilità totale **TR** (*total reliability*);
- connettività **CN**;
- unico iniziatore **UI**.

Le prime tre restrizioni vengono indicate con **R**, l'ultima con **I**. La loro unione dà la restrizione **RI**.

2. Broadcasting

Per il problema **broadcasting** vogliamo spargere l'informazione presente in una entità in tutte le altre presenti nella rete.

Definiamo P_{init} la proprietà che indica che una sola entità contiene l'informazione I:

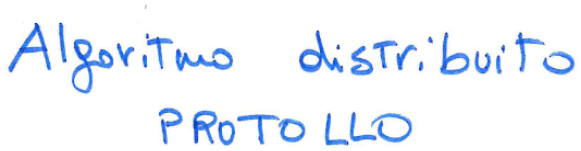
$$\exists x \in E \mid \text{valore}(x) = I$$
$$\wedge$$
$$\forall y \neq x \in E \quad \text{valore}(y) = \oslash.$$

Definiamo anche P_{final} la proprietà che indica che tutte le entità contengono l'informazione I:

$$\forall x \in E \quad \text{valore}(x) = I.$$

2.1. Prima versione

Vediamo una prima versione del **PROTOLLO** per broadcast.



Per questo protollo andiamo ad usare:

- **stati** $S = \{\text{iniziatore}, \text{inattivo}\}$;
- **stati iniziali** $S_{\text{init}} = \{\text{iniziatore}, \text{inattivo}\}$;
- **stati terminali** $S_{\text{term}} = \{\text{inattivo}\}$.

Definiamo anche le **regole** che devono seguire le nostre entità.

Iniziatore

1: Se riceve un impulso spontaneo
   - send(M) to $N(x)$
   - become inattivo

Inattivo

1: Se riceve M
   - processa M (*preleva le informazioni*)
   - send(M) to $N(x)$

Il messaggio è nella forma

$$M = (t, o, d, I),$$

formato dai campi:

- t **tipologia** del messaggio;
- o e d entità **origine** e entità **destinatario**;
- I **informazione**.

Questa prima versione ha un problema: **non termina mai**. Infatti, ogni stato diventa inattivo dopo l'impulso iniziale e questi, ogni volta che ricevono qualcosa, anche se già ce l'hanno, la mandano ai loro vicini. Vediamo alcune modifiche che possiamo fare.

Imporre di non mandare il messaggio a chi ce l'ha mandato non modifica il comportamento.

Per rendere futuro(t) vuoto ad un certo punto, modifichiamo gli stati:
- definiamo $S_{\text{start}} \subseteq S_{\text{init}}$ insieme degli stati che fanno iniziare il protollo;
- definiamo $S_{\text{final}} \subseteq S_{\text{term}}$ insiemi degli stati che eseguono solo l'azione nulla.

I nostri **stati** ora diventano:
- $S_{\text{init}} = \{\text{iniziatore}, \text{inattivo}\}$;
- $S_{\text{start}} = \{\text{iniziatore}\}$;
- $S_{\text{term}} = S_{\text{final}} = \{\text{finito}\}$.

Con questo nuovo stato terminale riusciamo a far terminare la computazione.

La soluzione che abbiamo costruito è **corretta** e **termina**. Cosa vogliono dire questi due termini?

Un problema è **corretto** se

$$\forall C(0) \in P_{\text{init}} \quad \exists t' \mid \forall t > t' \quad C(t) \in P_{\text{final}}$$

ovvero ogni configurazione iniziale che rispetta i propri predicati, da un certo t in poi, finisce in una configurazione finale che rispetta i propri predicati.

Un problema **termina** se

$$\exists t \mid \forall x \in E \quad \text{stato}_t(x) \in S_{\text{final}}$$

ovvero esiste un istante di tempo nel quale tutte le entità sono in uno stato finale.

2.2. Seconda versione [flooding]

Una versione migliore sfrutta la tecnica del **flooding**.

Abbiamo a disposizione i seguenti **stati**:
- $S = \{\text{iniziatore}, \text{inattivo}, \text{finito}\}$;
- $S_{\text{start}} = \{\text{iniziatore}\}$;
- $S_{\text{init}} = \{\text{iniziatore}, \text{inattivo}\}$;
- $S_{\text{final}} = S_{\text{term}} = \{\text{finito}\}$.

Iniziatore

1: Se riceve impulso spontaneo
   - send(M) to $N(x)$
   - become finito

Inattivo

1: Se riceve M
   - processa M (*preleva le informazioni*)
   - send(M) to $N(x) - \{\text{sender}\}$
   - become finito

Non l'abbiamo detto, ma le regole sono delle **funzioni totali**: infatti, se non definiamo un'azione per una coppia stato+evento allora la funzione di default esegue **nil**.

Il **numero di messaggi** è

$$M[\text{flooding}] = \sum_{x \in E} (N(x) - 1) + \underbrace{1}_{\text{iniziatore}} = 2m - n + 1.$$

Il **tempo** impiegato è

$$T[\text{flooding}] \leq d$$

perché nel caso peggiore l'iniziatore è nel nodo che definisce il diametro della rete.

Abbiamo anche dei **lower bound** per questo problema.

Teorema 2.2.1: Vale

$$M[\text{broadcast / RI}] \geq m.$$

Dimostrazione 2.2.1: Supponiamo per assurdo di poter risolvere broadcast con meno di m messaggi totali. Sia A questo protollo. Supponiamo che A non mandi mai messaggi sull'arco (x, y) del grafo G.

Il protollo A è corretto, quindi lavora bene su ogni grafo. Creiamo G' a partire da G:

- aggiungendo un nodo z non iniziatore;
- togliendo l'arco (x, y);
- aggiungendo gli archi (x, z) e (z, y) con etichette $\lambda_x(x, z) = \lambda_x(x, y)$ e $\lambda_y(y, z) = \lambda_y(y, x)$.

Se eseguo A su G' allora z non riceve mai il messaggio I, quindi A non è corretto. ■

Il tempo invece ha il seguente lower bound:

$$T[\text{broadcast / RI}] \geq d.$$

Questo è il **tempo causale**, ovvero il tempo nel caso peggiore.

Visti questi risultati, il protollo che abbiamo costruito è ottimale.

2.3. Problema wake-up

Il problema del **wake-up** è una versione generale del broadcast: in quest'ultimo partiamo con l'informazione in una sola entità, nel wake-up rilassiamo il vincolo di avere un unico iniziatore.

Useremo il protocollo **w-flood**, che utilizza i seguenti **stati**:

- $S = \{\text{dormiente}, \text{attivo}\}$;
- $S_{\text{init}} = S_{\text{start}}\{\text{dormiente}\}$;
- $S_{\text{final}} = S_{\text{term}} = \{\text{attivo}\}$.

Vediamo le **regole** per questa versione rilassata di broadcast.

Dormiente

1: Se riceve impulso spontaneo
   - send(W) to $N(x)$
   - become attivo
2: Se riceve W
   - send(W) to $N(x) -$ sender
   - become attivo

Come prima, il **tempo** impiegato è

$$T[\text{w-flooding}] \leq d,$$

mentre il **numero di messaggi** spediti è

$$\underbrace{2m - n + 1}_{\text{un iniziatore}} \leq M[\text{w-flooding}] \leq \underbrace{2m}_{n \text{ iniziatori}}.$$

3. Traversal

Il problema **traversal** richiede di visitare **SEQUENZIALMENTE** ogni entità del nostro sistema. In poche parole, ad ogni unità di tempo devo visitare al massimo una entità nuova.

3.1. Prima versione

Per risolvere questo problema useremo una **visita in profondità** con il protocollo **depth-first traversal**. Inoltre, useremo un messaggio particolare, un **token** T, che può viaggiare nelle rete al massimo una volta in ogni istante di tempo.

Vediamo i passi che segue questo protocollo:

- un nodo che riceve T per la prima volta ricorda il sender e invia il token ad uno dei suoi vicini, aspettando un messaggio di **return/back-edge**. Quando riceve questo messaggio, effettua queste operazioni per ogni entità vicina. Quando la lista finisce, invia un return al sender;
- un nodo che ha già ricevuto T spedisce un back-edge al sender.

Notiamo subito che abbiamo tre tipi di messaggi: il **token** T, il **return** (*ho finito la visita dei vicini*) e il back-edge (*ho già ricevuto il token, quindi sono già stato visitato*).

Utilizziamo per il DF-traversal i seguenti **stati**:

- $S = \{\text{initiator}, \text{idle}, \text{visited}, \text{done}\}$;
- $S_{\text{init}} = \{\text{initiator}, \text{idle}\}$;
- $S_{\text{term}} = \{\text{done}\}$.

Come restrizioni usiamo ancora **RI**.

Initiator

1: Se riceve un impulso spontaneo
    initiator = true
    unvisited = $N(x)$
    visit()

Idle

1: Se riceve T
    entry = sender
    unvisited = $N(x)$ — sender
    initiator = false
    visit()

Visited

1: Se riceve R
    visit()
2: Se riceve B
    visit()
3: Se riceve T
    unvisited = unvisited - sender
    send B to sender

Visited

Procedura visit

```
1: if unvisited ≠ ⦰
2:     next = unvisited
3:     send(T) to next
4:     become visited
5: else
6:     if initiator == false
7:         send(R) to entry
8:     become done
```

Per tutto il resto c'è **mastercard**.

Per calcolare la **complessità**, notiamo che se x e y sono due entità, sul loro canale passa sempre il token T e il return R o il back-edge B. Il traversal è **sequenziale**, quindi passo per tutte le entità una per volta con il token, ma allora **tempo** e **numero di messaggi** sono $2m$, perché mando due messaggi per ogni arco. Vediamo i **lower bound** di questo problema.

Il numero di messaggi è $M[\text{traversal}] \geq m$ per il teorema che abbiamo visto nel problema broadcast. Il tempo invece è $T[\text{traversal}] \geq n-1$ perché ogni nodo viene visitato in sequenza. Notiamo che in un grafo, il numero di archi è tale che

$$n-1 \leq m \leq \frac{n(n-1)}{2},$$

quindi il tempo che abbiamo con questo algoritmo è $O(n^2)$.

Con questi bound, il nostro algoritmo è ottimale per il numero di messaggi, ma non il tempo.

3.2. Seconda versione

Cerchiamo di migliorare il nostro protocollo. Osserviamo che ad ogni istante di tempo viaggia un solo messaggio: cerchiamo di aggiungere **concorrenza** con una quantità di messaggi dell'ordine di $O(m)$ per rendere più veloce il protocollo.

Notiamo che un nodo non visitato che riceve il token T potrebbe dire ai suoi vicini che lo ha ricevuto e che non dovrebbero poi mandarglielo in futuro. Questa idea cerca di evitare l'invio di un token T su un link che sarebbe back-edge. Questa situazione effettivamente non si presenta, perché il tutto dipende dai clock delle singole entità, ma migliora considerevolmente le prestazioni.

Il nuovo numero di messaggi è $2n-2$ per i token T e le return R, sommati a $2m-(n-1)$ per i messaggi ai visited, sommati a $2(m-(n-1))$ per gli errori di invio de token sui back-edge. In totale abbiamo un numero di messaggi che è $O(m)$.

Calcolando il tempo ideale di questa soluzione, ovvero il tempo che non contiene ritardi, errori, e fa viaggiare token T e avvisi di visited assieme, esso diventa $T(n) = n-1$ che è il nostro lower bound.

Questo nuovo protocollo, che chiamiamo **DF*-traversal**, è ottimo per messaggi e tempo.

Vediamo gli **stati** che utilizza:

- $S = \{\text{initiator}, \text{idle}, \text{available}, \text{visited}, \text{done}\}$;

- $S_{\text{init}} = \{\text{initiator}, \text{idle}\}$;
- $S_{\text{term}} = \{\text{done}\}$.

Vediamo infine come funziona effettivamente il protocollo.

Initiator

1: Se riceve impulso spontaneo
   - initiator = true
   - unvisited = $N(x)$
   - next = unvisited
   - send(T) to next
   - send(V) to $N(x)$ − next
   - become visited

Idle

1: Se riceve T
   - unvisited = $N(x)$
   - first-visit()

2: Se riceve V
   - unvisited = $N(x)$ − sender
   - become available

Available

1: Se riceve T
   - first-visit()

2: Se riceve V
   - unvisited = unvisited - sender

Visited

1: Se riceve V
   - unvisited = unvisited - sender
   - if (next == sender) then visit()

2: Se riceve T
   - unvisited = unvisited - sender
   - if (next == sender) then visit()

3: Se riceve R
   - visit()

Procedura first-visit

la usiamo la prima volta che si riceve il token T

Procedura first-visit

```
1:  initiator = false
2:  entry = sender
3:  unvisited = unvisited - sender
4:  if unvisited ≠ ⦰
5:      next = unvisited
6:      send(T) to next
7:      send(V) to N(x) − {sender, next}
8:      become visited
9:  else
10:     send(R) to sender
11:     send(V) to N(x) − sender
12:     become done
```

Procedura visit

la usiamo quando abbiamo già mandato il visited V

```
1:  if unvisited ≠ ⦰
2:      next = unvisited
3:      send(T) to next
4:  else
5:      if initiator == false
6:          send(R) to entry
7:      become done
```

4. Spanning tree

Il problema dello **spanning tree** è molto comodo perché permette di ridurre un grafo completo ad uno molto più leggero che semplifica la complessità di comunicazione. Dobbiamo stare comunque attenti ai costi, soprattutto a quelli di costruzione dell'albero e a quelli a cui andremo incontro con questa nuova rappresentazione.

Il problema dello spanning tree ci richiede di costruire una **sotto-rete** tale che:

- ogni entità è presente;
- ogni entità è connessa;
- è priva di cicli.

Per risolvere questo problema dobbiamo dare un po' di conoscenza dell'albero alle entità.

Definiamo tree-$N(x) \subseteq N(x)$ il sottoinsieme dei vicini di x che partecipano all'albero e che sono collegati direttamente a x. Diciamo che un arco (x, y) sta nell'insieme $\text{link}(\text{tree-}N(x))$ se e solo se $y \in \text{tree-}N(x)$. Questo insieme è l'insieme degli archi uscenti da x diretti nei nodi di tree-$N(x)$.

Infine, definiamo Tree come

$$\bigcup_{x \in E} \text{link}(\text{tree-}N(x)).$$

4.1. Prima versione

Usando sempre le restrizioni RI, definiamo il protocollo **shout**.

La **radice** dell'albero è l'entità che inizierà il protocollo. Il protocollo fa quello che dice il suo nome: ogni entità chiederà ai suoi vicini se vogliono partecipare, con il loro arco, all'albero.

Vediamo i passi che segue questo protocollo:

- la radice s spedisce Q ai suoi vicini e attende le risposte;
- ogni entità x diversa da s che riceve Q:
  - per la prima volta risponde **SI** e invia Q ai suoi vicini, mettendosi in attesa come ha fatto s;
  - per l'n-esima volta ($n \geq 2$) risponde **NO**;
- ogni entità memorizza il padre dal quale ha ricevuto Q e tutti i figli che hanno risposto **SI**;
- una entità termina quando riceve tutte le risposte.

Abbiamo a disposizione i seguenti **stati**:

- $S = \{\text{iniziatore}, \text{inattivo}, \text{attivo}, \text{finito}\}$;
- $S_{\text{init}} = \{\text{iniziatore}, \text{inattivo}\}$;
- $S_{\text{term}} = \{\text{finito}\}$.

Iniziatore

1: Se riceve impulso spontaneo
    root = true
    counter = 0
    tree-$N(x) = \varnothing$
    send(Q) to $N(x)$
    become attivo

Inattivo

1: Se riceve Q
- root = false
- parent = sender
- counter = 1
- tree-$N(x)$ = sender
- send(SI) to sender
- if counter == $|N(x)|$
  - become finito
- else
  - send(Q) to $N(x) -$ sender
  - become attivo

Attivo

1: Se riceve Q
- send(NO) to sender

2: Se riceve SI
- tree-$N(x)$ = tree-$N(x) \cup \{\text{sender}\}$
- counter = counter + 1
- if counter == $|N(x)|$
  - become finito

3: Se riceve NO
- counter = counter + 1
- if counter == $|N(x)|$
  - become finito

Questo protocollo è corretto:

- **terminazione**: ogni entità entra nello stato terminale quando ha ricevuto tutte le risposte;
- **albero**: tutte le entità sono presenti e connesse per flooding + SI al primo Q che ricevono, e non ho dei cicli perché rispondo SI una e una sola volta (la radice unica che dice sempre NO).

Il **numero di messaggio** inviati è

$$M[\text{shout}] = \underbrace{2M[\text{flooding}]}_{Q+\text{ risposta}} = 2[2m - (n-1)] \approx 4m$$

mentre il **tempo** è

$$T[\text{shout}] = T[\text{flooding}] + \underbrace{1}_{\text{ultimo } Q} \leq d + 1.$$

4.2. Seconda versione

Una seconda versione di shout, che chiameremo **shout++**,cerca di eliminare qualche messaggio, perché come tempo ci siamo con il lower bound.

Questa versione cancella i NO. La decisione dietro a questa pazza idea è perché i NO vengono inviati al sender di una Q quando il nodo ha già ricevuto un Q. Teniamo quindi tutti i SI e interpretiamo i Q doppioni come se fossero dei NO.

Il nuovo **numero di messaggi** è $2m$, perché mandiamo su uno stesso link:

- due Q (*se già ricevuto*);
- un Q e un SI (*se non ricevuto*).

Una soluzione alternativa usa il protocollo traversal, però costruendo l'albero in sequenza andiamo un po' controllo l'approccio parallelo che ci piace. In ogni caso, per traversal i link sui quali viaggiano le return sono i link dentro Tree.

5. Election

Il problema **election** vuole rompere la simmetria: dobbiamo individuare una entità tra tanta autonome e omogenee che diventi **leader**, rendendo le altre **follower**.

Lemma 5.1 (*Risultato di impossibilità*): È impossibile individuare deterministicamente un leader sotto le restrizioni R.

Dimostrazione 5.1: Idea della dimostrazione: siano $x, y \in E$ due entità omogenee inizializzate nello stesso modo e nello stesso stato. Essendo identiche, eseguono anche lo stesso algoritmo, trovandosi poi in uno stato finale uguale. Ma allora non ho trovato un leader. ■

Lemma 5.2 (*Risultato di possibilità*): Sotto le restrizioni RI l'entità di partenza diventa subito leader, ma il problema è risolto dall'esterno e non dal sistema.

Aggiungiamo una nuova restrizione, la **initial distinct values**, denotata con **ID**. Se aggiunta alle restrizione R otteniamo le restrizioni **IR**. Questa restrizione aggiunge un campo $\mathrm{id}(x)$ ad ogni entità.

Abbiamo possibili **strategie** di soluzione:

- **elect minimum**: trova l'entità con $\mathrm{id}(x)$ minimo la rende leader;
- **elect minimum initiator**: trova l'entità initiator con $\mathrm{id}(x)$ minimo e la rende leader.

Risolveremo questo problema in una **topologia ring**, ovvero ad **anello**. In questa topologia le entità sono disposte ad anello, ovvero abbiamo $A = (x_0, ..., x_{n-1})$ con una connessione tra x_{n-1} e x_0. Il numero di archi e il numero di entità sono uguali in questo caso.

Aggiungiamo un'ulteriore restrizione, ovvero che ogni entità sa di essere in un ring.

Infine, da ora chiameremo OTHER la quantità $N(x) -$ sender.

5.1. Prima versione

Una prima versione di protocollo per questo problema (*elect minimum*) è il protocollo **all the way**. I messaggi viaggiano intorno all'anello, inoltrati dalle varie entità in una direzione prestabilita. I messaggi mandati, per ora, sono nella forma (select, $\mathrm{id}(x)$).

Quando una entità x riceve un messaggio E dall'entità y inoltra E all'entità successiva, assieme ad un messaggio E' con $\mathrm{id}(x)$ al posto di $\mathrm{id}(y)$.

Con questo continuo inoltro di messaggi, ogni entità x vede il valore $\mathrm{id}(y)$ di ogni entità y e può così calcolarne il minimo.

Quando facciamo terminare ogni entità? Una prima idea è fermare x quando si riceve un messaggio E con il proprio $\mathrm{id}(x)$. Siamo sicuri di aver finito? Rispondiamo:

- **SI**: se supponiamo la restrizione **message ordering** (*prelevo FIFO*), ma noi non ce l'abbiamo;
- **solo se ne ha visti n diversi**: se supponiamo che le entità siano a conoscenza della dimensione dell'anello, ma noi non ce l'abbiamo;
- **NO**: giusto, dobbiamo riempire in maniera opportuna i messaggi per far terminare correttamente le altre entità.

Quello che aggiungiamo al messaggio è un **contatore** che, partendo da 1, viene continuamente incrementato ogni volta che una entità inoltra il messaggio. Quando il messaggio ritorna all'entità che ha generato il messaggio, essa saprà esattamente la dimensione della rete poiché contenuta dentro il contatore. Grazie a questa informazione, ora l'entità sa se può fermarsi e calcolare il minimo oppure aspettare ancora qualche messaggio mancante.

Vediamo gli **stati** utilizzati da questo protocollo:

- $S = \{\text{asleep}, \text{awake}, \text{leader}, \text{follower}\}$;
- $S_{\text{init}} = \{\text{asleep}\}$;
- $S_{\text{term}} = \{\text{leader}, \text{follower}\}$.

Asleep

```
1: Se riceve impulso spontaneo
       initialize()
       become awake
2: Se riceve (elect, value, counter)
       initialize()
       send((elect, value, counter + 1)) to OTHER
       min = min(min, value)
       count = count + 1
       become awake
```

Awake

```
1: Se riceve (elect, value, counter)
       if value ≠ id(x)
           send((elect, value, counter + 1)) to OTHER
           min = min(min, value)
           count = count + 1
           if know == true
               check()
       else
           size = counter
           know = true
           check()
```

Procedura initialize

```
1: count = 0
2: size = 1
3: know = false
4: send((elect, id(x), 1)) to RIGHT
5: min = id(x)
```

Procedura check

```
1: if count == size
2:     if min == id(x)
3:         become leader
4:     else
5:         become follower
```

Il **numero di messaggi** che vengono inviati con questo protocollo è

$$M[\text{all-the-way} \ / \ \text{IR} \cup \text{RING}] = n^2.$$

5.2. Seconda versione

Questa prima versione è troppo costosa, quindi passiamo al piano due: scegliamo elect minimum initiator come politica di ricerca. In questo caso, solo gli iniziatori spediscono un proprio messaggio E, tutte le altre entità inoltrano e basta. Quando gli initiator hanno finito il calcolo del leader, mandano un messaggio di fine a tutti gli altri. aggiungendo quindi n messaggi finali.

Il **numero di messaggi** diventa ora $nk + n$, con k numero di initiator, mentre il **tempo** è $\leq 3n - 1$, che si raggiunge quando 2 initiator si attivano in momenti diversi.

6. Routing

Vogliamo mandare un messaggio da x a y utilizzando il cammino minimo tra queste due entità. Il problema è che l'entità x non conosce questo cammino minimo. Il problema del **routing** vuole risolvere le paturnie dell'entità x così che possa fare sogni tranquilli.

Una soluzione è fare broadcast da x del messaggio E, ma questo è totalmente inefficiente. Decidiamo quindi di scegliere uno tra tutti i possibili cammini tra x e y e, se questo è il migliore, siamo anche più contenti.

Visto che poi dobbiamo saperci spostare nella rete, dobbiamo salvare le informazioni sui costi della rete per ogni entità, così che possiamo calcolare i cammini minimi verso ogni entità.

Le informazioni le salviamo nella **full routing table**, una tabella con le varie destinazioni sulle righe e path minimo + costo sulle colonne (*grazie **Rossi** che me le hai insegnate queste cose*).

6.1. Prima versione

Sotto restrizioni IR, vediamo il protocollo **gossiping**. L'idea è che ogni entità si costruisce una mappa del grafo G, una matrice che è praticamente la matrice di adiacenza, e all'occorrenza si calcola le righe della full routing table.

Per costruire questa tabella, detta $\mathrm{MAP}(G)$, che contiene i costi dei vari archi, ogni entità diffonde le proprie informazioni sui vicini ad ogni altra entità vicina.

Vediamo cosa fa questo protocollo:

- costruisce l'albero T per il grafo G;
- ogni entità diffonde le proprie informazioni a tutte le altre usando i link di T;
- ogni entità acquisisce dai vicini $\mathrm{id}(x)$ e costi dei vari link.

Il numero di messaggi è:

- n^2 per la comunicazione;
- $m + n\log(n)$ per la creazione dello spanning tree;
- $2m$ per acquisire informazioni dai vicini;
- $2m(n-1)$ per il broadcast delle informazioni su T.

In totale, il **numero di messaggi** è $\approx 2mn$, che vale n^2 quando G è sparso. Il **tempo**, invece, è molto difficile da calcolare.

Vediamo come questo protocollo richieda tanta memoria per poter essere eseguito.

6.2. Seconda versione

Il protocollo **iterated-construction** costruisce la FRT di ogni entità a più riprese senza usare $\mathrm{MAP}(G)$. Infatti, all'inizio ogni FRT contiene solo le informazioni dei vicini. Inoltre, nella FRT possiamo evitare di tenere tutto il cammino minimo, ma possiamo limitarci a salvare quale nodo è il prossimo nel cammino. Definiamo inoltre il **distance vector** V come la FRT ristretta alle colonne con solo destinazione e costo, senza path.

Cosa fa questo protocollo:

- ogni entità diffonde la propria V ai suoi vicini;
- sulla base delle informazioni ricevute, ogni entità stabilisce se sono stati trovati cammini minimi migliori di quelli della propria FRT e in tal caso la aggiorna.

Il **numero di iterazioni** di questo protocollo è $n-1$, e si dimostra per induzione.

Vediamo come fa x ad individuare, ad ogni iterazione, il cammino minimo per un nodo z.

Sia $V_y^i[z]$ il costo del cammino da y a z alla i-esima iterazione. Alla $(i+1)$-esima iterazione questo costo arriva ai vicini di y, e sia x uno di questi. Esso calcola il valore

$$w[z] = \min_{y \in N(x)} \{\vartheta(x,y) + V_y^i[z]\},$$

dove $\vartheta(x,y)$ rappresenta il costo del link (x,y). Se $w[z] < V_x^i[z]$ allora x sceglie $w[z]$ come costo per il path per z, aggiornando la FRT e memorizzando anche il vicino y che ci ha dato l'informazione.

Vediamo come utilizziamo molta meno memoria della $\mathrm{MAP}(G)$, perché i DV sono lineari.

Il **numero di messaggi** è $2mn(n-1)$, ovvero eseguo $n-1$ iterazioni dove mando n volte il distance vector su $2m$ link del grafo. Il **tempo ideale** lo indichiamo con $\tau(n)$, ed è tale che

$$\tau(n) = \begin{cases} O(1) & \text{se G consente messaggi lunghi} \\ O(n) & \text{altrimenti} \end{cases}.$$

Se $\tau(n) = O(1)$ il tempo diventa lineare in n e il numero di messaggi è $O(mn)$. Se invece $\tau(n) = O(n)$ il tempo diventa quadratico in n e il numero di messaggi è $O(mn^2)$.