

Algoritmi Paralleli e Distribuiti

Massimo Perego

Contents

Introduzione	3
Primo caso: CPU Sincrone	4
Secondo caso: CPU Asincrone	6
Teoria e realtà: motivazioni	7
Problematiche parallele e distribuite	8
Il Tempo	10
Teoria della Complessità	13
Efficienza nel caso sequenziale	13
1 Algoritmi Paralleli	15
1.1 Tipi di architetture parallele	18
1.1.1 PRAM	19
1.2 Broadcast P-RAM	24
1.3 Efficienza	26
1.3.1 Speed-up	27
1.3.2 Definizione di Efficienza	28
1.4 Problema Sommatoria	32
1.4.1 Valutazione dell'algoritmo	36
1.4.2 Sommatoria come schema per altri problemi	39
1.5 Problema delle somme prefisse	45
1.5.1 Pointer Doubling	46
1.6 Valutazione di polinomi	52
1.7 Ricerca di un elemento	56
1.8 Problema dell'ordinamento	59
1.8.1 CountingSort	60
1.8.2 BitSort	63
1.9 Tecnica dei Cicli Euleriani	71
1.9.1 Attraversamento in preordine	74

1.9.2	Calcolo della profondità dei nodi	76
	Osservazioni Finali su PRAM	77
2	Architetture parallele a Memoria Distribuita	78
2.1	Parametri di Rete	79
2.2	Tipici problemi	80
2.3	Confrontatori	81
2.3.1	Valutare una rete	83
2.4	Array Lineari	85
2.4.1	Primitiva Swap contiguo	86
2.5	Problema Shuffle	87
2.6	Trasmettere Dati	88
2.7	Problema Max	89
2.8	Swap Non Contiguo	92
2.9	Primitiva MinMax	93
2.10	Problema Ordinamento	94
2.10.1	Merge-Split	96
2.11	Architettura Mesh	97
2.11.1	Max	98
2.11.2	Ordinamento	100
3	Algoritmi Distribuiti	103
3.1	Misure di Complessità	107
3.2	Definizione di un Problema	108
3.3	Protocollo	109
3.3.1	Broadcast	110
3.4	Problema Wake-Up	115
3.4.1	Protocollo WFlood	115
3.5	Problema Traversal	117
3.5.1	Depth-First Traversal	117
3.6	Problema Spanning Tree	122
3.6.1	Protocollo Shout	125
3.6.2	Shout migliorato: Shout+	128

Introduzione

All'interno del corso sarà presente solo la progettazione di algoritmi, non si parla di architetture.

Contesto: Generalmente parlando di algoritmi si considera un singolo esecutore, mentre con “paralleli e distribuiti” si aggiunge l’idea di avere una **pool di esecutori**, velocizzando possibilmente la risoluzione del problema, ma introducendo nuove problematiche.

Per un algoritmo non parallelo, quindi sequenziale, le **fasi** sono:

- **Progettazione:** scrittura dell’algoritmo, secondo diverse tecniche come *divide et impera* (dividere il problema in sottoproblemi e combinare le sotto-soluzioni per la soluzione globale, es mergesort), *programmazione dinamica* (utilizzare la memoria per memorizzare soluzioni parziali che saranno utilizzate per istanze più grandi, es calcolo Fibonacci), *greedy* (per problemi di ottimizzazione, c’è una funzione da massimizzare/minimizzare, il problema generale viene risolto scegliendo a ogni passo la soluzione localmente ottima, migliore per quel singolo step, es algoritmo di Kruskal), ...
- **Valutazione delle prestazioni:** in termini di tempo, spazio di memoria e in generale valutare l’algoritmo nella sua complessità
- **Codificazione:** fase di implementazione con un opportuno linguaggio di programmazione

Per gli algoritmi paralleli le problematiche sono simili, ma l’unica tecnica che può essere riciclata è quella di divide et impera.

Per gli algoritmi paralleli questi vengono valutati sempre in termini di tempo ma, al posto dello spazio, si conta il numero di processori.

Ma la pool di esecutori può avere diverse caratteristiche, ci sono due casi:

- **primo caso:** “una squadra in cui batte un solo cuore”, più processori sincroni
- **secondo caso:** “ogni membro un mondo a parte”, computer interfacciati ma ognuno ha il suo clock ed il suo tempo per risolvere le operazioni (es. diversi pc su internet), membri del pool lavorano in modo asincrono

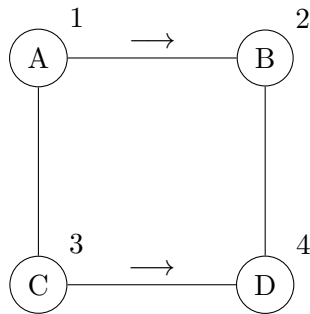
Primo caso: CPU Sincrone

I processori sono sincroni, quindi vanno all'unisono, si ha un **clock centrale** che scandisce il tempo per l'intero insieme di processori quindi le **istruzioni vengono eseguite tutte assieme** in un tempo determinato per tutti.

Può essere che condividano risorse, come ad esempio una memoria comune, oppure i processori possono essere solo collegati tra loro .

Con la **memoria condivisa** il sistema si chiama **PRAM**, l'altro si chiama **sistema parallelo a memoria distribuita** (manca quella condivisa).

Esempio:



Memoria distribuita, ogni dato è in un processore diverso, i dati sono distribuiti. Generalmente i collegamenti sono full duplex (avanti e indietro, ambo le direzioni).

Operazioni:

send(1,2)	send(3,4)
A+B	C+D
send(2,4)	
A+B+C+D	

Riga per riga:

- Il processore 1 invia A al 2, il 3 invia C al 4
- Nel processore 2 si fa la somma $A+B$, nel 4 si fa $C+D$
- Il processore 2 invia la somma al 4
- Il 4 farà il totale.

Ogni riga permette un'operazione, si tratta di un ciclo di clock, quindi per fare questa somma in totale sono 4 cicli di clock.

In genere un solo processore contiene la risposta, solitamente quello con l'indice maggiore (ultimo, nel caso dei processori 1, 2, 3 e 4 il 4 contiene la risposta).

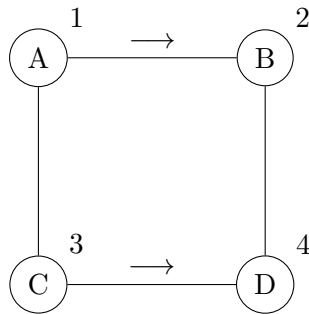
Con questo esempio non si capisce bene il guadagno dato dal parallelismo rispetto all'algoritmo sequenziale, ma si può generalizzare l'idea su un'istanza di lunghezza n , con un numero di processori adeguato (ma anche in caso di un numero di processori ridotti l'algoritmo si può adeguare alla situazione). In questo modo si può notare che il numero di passi scala in modo logaritmico con la dimensione n (prima si sommano 2 a 2, poi 4 a 4, poi 8 a 8, ...). Si passa da n passaggi con il sequenziale a $\log n$ con il parallelo.

Secondo caso: CPU Asincrone

In questa casistica **ogni processore ha i propri tempi** nonostante siano **collegati tra loro**, e ovviamente non si può assumere memoria condivisa. Non avere un clock centrale porta ad avere algoritmi distribuiti e NON paralleli come nel caso precedente.

Non si può assumere **nessun tipo di sincronizzazione gratuita**, devono **scambiarsi dei messaggi** (es: attraverso internet, se non mi viene detto nulla non so niente degli altri processori).

Esempio:



In questo caso potrebbero esserci **tempi** più lunghi e che **variano da collegamento a collegamento**, anche tra scambi diversi sullo stesso collegamento (come può accadere su internet).

Avendo velocità diverse, **un processore potrebbe dover aspettare** che gli altri terminino di elaborare/scambiare i messaggi. Serve scambiare messaggi per il coordinamento.

Teoria e realtà: motivazioni

Si studiano gli algoritmi a livello teorico, basandosi sulla realtà, esistono architetture distribuite per l'implementazione di tali algoritmi. La nascita di architetture parallele e distribuite comincia dalla seconda metà degli anni '60, di conseguenza si è sviluppata la teoria.

Esempi di architetture parallele:

1. **Supercomputer:** cluster di processori con incredibili prestazioni di calcolo (CDC, CRAY), solitamente a scopo di simulare sistemi complessi (fisici, militari, ...; tutto ciò che richiede un grande numero di operazioni aritmetiche). Di solito sono a memoria distribuita in quanto sarebbe difficile realizzarli a memoria condivisa dato che tutti i processori dovrebbero essere in grado di accedervi
2. **GPU:** più processori collegati tra loro, in genere dedicate a problemi di grafica ma permettono di risolvere bene problemi di algebra lineare quindi viene usata in senso più generale
3. **Multicore processor:** più unità di calcolo all'interno dello stesso processore, per migliorare le prestazioni ottimizzando l'assorbimento di energia (non bisogna aumentare il clock che porterebbe a un aumento dell'assorbimento) e relativi problemi di raffreddamento (meno energia meno calore); sistemi PRAM
4. **Circuiti integrati:** sistemi di calcolo formati da gate opportunamente connessi; sostanzialmente una struttura a livelli che lavorano in parallelo prendendo input dal livello precedente

Esempi di architetture distribuite:

1. **Reti di calcolatori:** Internet (dagli anni '60), connettono dispositivi in molti diversi, quindi servono protocolli di comunicazione come TCP/IP
2. **Reti mobili:** la topologia di connessione dei dispositivi cambia continuamente (gli smartphone si muovono)
3. **Reti di sensori:** dispositivi con capacità limitate, per la maggior parte del tempo in "stand by", eseguono solo quello per cui sono costruite; solitamente hanno scopo di monitoraggio degli ambienti; necessitano di meccanismi di "wake up" (a un certo punto devo svegliarli), "acknowledge" (devono dire quando finiscono) e "recovery" (memorizzare le proprie informazioni/quelle ricevute)

Negli **algoritmi paralleli** il fattore rilevante è il **tempo**. Nel sommare 4 numeri sono 4 passi, per sommarne 1000 ne servono 10. Per vedere se un algoritmo parallelo è efficiente si valuta il tempo.

Per gli **algoritmi distribuiti** quello che conta è il **coordinamento**, ovvero il **numero di messaggi da scambiare** (meno messaggi = più veloce). Il numero di messaggi definisce una sorta di tempo e permette anche di avere un'idea della congestione della rete.

Problematiche parallele e distribuite

La **progettazione** richiede **nuove idee**, mentre la il processo di **valutazione** delle prestazioni richiede **nuove misure**.

Perché nuove idee? Un algoritmo sequenziale efficiente non necessariamente porta a un algoritmo parallelo e/o distribuito efficiente, di conseguenza servono tecniche ad hoc per entrambi le tipologie.

Inoltre buoni algoritmi paralleli non sempre portano a buoni algoritmi distribuiti e viceversa. Ad esempio, nel **distribuito serve gestire problemi di ritardo e comunicazione**, che nel parallelo non si presentano.

Le misure che verranno definite non faranno riferimento a una architettura in particolare ma saranno definite solo in maniera teorica.

Per ogni paradigma si dovrà definire:

1. **Modello teorico**
2. Come **valutare le performance** degli algoritmi
3. **Semplici problemi** per apprendere le tecniche

Nel caso di **architetture parallele**:

A. Primo caso:

1. **PRAM** (comunicazione immediata): memoria condivisa
2. Risorse di calcolo: tempo, hardware (inteso come numero di processori)
3. Problemi: sommatoria, somme prefisse, ordinamento

B. Secondo caso

1. Modello a **memoria distribuita**:

- array lineari: n processori collegati all'interno di un array lineare, ovvero P_i collegato a P_{i+1} , il quale è collegato a P_{i+2} e così via
 - mesh: versione bidimensionale dell'array lineare
 - albero: albero binario completo di n foglie
 - ipercubo: per $n = 2^d$, un ipercubo (o d -cubo) è un grafo i cui vertici sono elementi $c_1, \dots, c_d \in \{0, 1\}^d$ e due vertici x e y sono estremi di un lato se le parole x e y differiscono in una sola posizione, quindi P_x e P_y sono collegati se x e y differiscono di una sola posizione. Sostanzialmente, un cubo ha 3 dimensioni ed ogni vertice è collegato ad altri 3, il d -cubo è la stessa cosa generalizzata in d dimensioni, con un processore su ogni vertice. Inoltre è una struttura ricorsiva, un $d + 1$ -cubo è formato da 2 d -cubi collegati tra loro, dove uno rappresenta la parte del $d + 1$ -cubo con bit più significativo a 0, mentre l'altro la parte con bit più significativo ad 1
2. Risorse di calcolo: tempo, hardware (processori non connessi direttamente comunicano più lentamente)
 3. Problemi: shuffle, max, ordinamento

Nel caso di **architetture parallele**:

1. Definizione del modello astratto
2. Risorse di calcolo: tempo, numero di messaggi (troppi = congestione)
3. Problemi: broadcast, wake up, traversal, spanning tree, election, routing

Il Tempo

Gli algoritmi vanno **valutati** (anche) **sul tempo**, in quanto sia nel caso di algoritmi paralleli che di algoritmi distribuiti, la risorsa tempo è cruciale.

Come nel caso sequenziale, la **definizione formale** è:

- $T(x)$ = numero di operazioni elementari su input x (istanza).
- $t(n) = \max\{T(x) | x \in \Sigma^n\}$, il massimo valore di T tra tutti gli input di dimensione n , ovvero il caso peggiore tra tutte le istanze di dimensione n . Si tratta quindi di una funzione in n , dove n è la dimensione dell'input.

Spesso non saremo interessati a una valutazione precisa ma a un **tasso di crescita della funzione**: quindi si usano le funzioni asintotiche O, Ω, Θ .

Definizioni (non formali, già viste troppe volte): siano $f, g : \mathbb{N} \rightarrow \mathbb{N}$ due funzioni definite sui numeri naturali

- $f(n) = O(g(n))$, la funzione è limitata superiormente (da un certo punto in poi, non per forza da subito) dalla funzione $c \cdot g(n)$, per un valore di c
- $f(n) = \Omega(n)$, la funzione è limitata inferiormente (da un certo punto in poi) da $c \cdot g(n)$, per un valore di c
- $f(n) = \Theta(n)$, unione delle due sopra, limite stretto, ovvero la funzione è compresa tra $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Tutte valgono asintoticamente, quindi da un certo punto in poi.

Da qui in poi sarà dato per scontato il concetto di MdT (chiamata Deterministic Turing Machine DTM), non lo riporterò, è già su almeno altri 2 blocchi di appunti e mi sono rotto il cazzo di scriverlo, comunque testina, quintupla, bla bla bla.

Valutazione di $t(n)$:

1. Viene **valutata su** un particolare **modello di calcolo**, quando calcolo la funzione vado a contare il numero di operazioni
2. Va scelto il **criterio di costo: uniforme o logaritmico**

Le **operazioni da contare** sono quelle **primitive messe a disposizione dal modello di calcolo**.

Esempio: palindromi

- Input: $x \in \{0, 1\}^*$
- Output: x è palindroma

Soluzione:

```
for (i=1, j=|x|; i<j && x_i = x_j; i++, j--);  
return i >= j;
```

Two pointers, controlla se ogni lettera è uguale a quella “dall’altro lato”. Se si blocca (quindi non è palindroma) gli indici rimarranno tali che $i < j$ (e di conseguenza restituirà falso alla fine), altrimenti i supererà j , dato che se arrivano a metà il controllo sarà valido fino alla fine portando gli indici a essere invertiti rispetto alla partenza.

Tale programma:

- su RAM (memoria ad accesso casuale) ha

$$t(n) = O(n)$$

(basta leggere una volta ogni lettera, posso accedere dove voglio nella memoria)

- Su DTM (una sola testina di lettura) ha

$$t(n) = O(n^2)$$

(devo fare avanti e indietro a ogni lettera che controllo)

Inoltre bisogna **fare attenzione alla dimensione dei dati** in gioco. Esempio: Fattoriale

- Input: $n \in \mathbb{N}$
- Output $n!$

Soluzione:

```
k = 1;  
for(i = 1; i <= x; k = k*i, i++);  
return k;
```

Effettua semplicemente il calcolo del fattoriale.

Tale programma:

- Contando la moltiplicazione come operazione elementare di una RAM

$$t(\log n) = n$$

($\log n$ dato che n è un numero naturale che si scrive in $\log n$ bit)

- su DTM, dovendo scrivere il binario il risultato $n! \simeq n^n$ abbiamo

$$t(\log n) \geq \log n^n = n \log n$$

Criteri di costo:

- **Uniforme:** le operazioni elementari richiedono una unità di tempo
- **Logaritmico:** ogni operazione elementare ha un costo che dipende dal numero di bit degli operandi

Teoria della Complessità

Possibili tassi di crescita di $t(n)$. Dobbiamo **stabilire quando qualcosa è efficiente per algoritmi paralleli e quando per i sequenziali**.

La funzione $t(n)$ si dirà:

- logaritmica quando è $O(\log n)$
- polilogaritmica quando è $O(\log^k n)$ per una costante $k > 0$
- lineare quando è $O(n)$
- polinomiale quando è $O(n^k)$ per una costante $k > 0$
- esponenziale quando NON è $O(n^k)$ per ogni costante $k > 0$

Efficienza nel caso sequenziale

Definizione formale: un problema è efficiente se ammette un **algoritmo efficiente**, ovvero viene **risolto** da una DTM in **tempo polinomiale**.

Si dicono:

- \mathcal{P} = classe dei problemi di decisione risolti in efficiente tempo
- \mathcal{FP} = classe dei problemi generali risolti in efficiente tempo

Oppure

- \mathcal{P} = classe dei problemi di decisione risolti in tempo polinomiale
- \mathcal{FP} = classe dei problemi generali risolti in tempo polinomiale

La differenza tra \mathcal{P} e \mathcal{FP} è che i problemi di decisione richiedono solo *s* o *no*, che sostanzialmente è il calcolo di un solo bit, mentre gli altri richiedono di calcolare più bit.

Perché consideriamo il tempo polinomiale come efficiente? Esempio: considerando un processore da 4 GHz (\neq da una DTM) su un'istanza di 4000 caratteri (mezza pagina circa)

complessità $t(n)$	tempo di attesa
n	$1\mu s$
n^2	$4ms$
2^n	> 1 secolo
3^n	~ 2 secoli e mezzo

Come si può vedere i tempi esponenziali esplodono anche su istanze piccole.

Quindi: \mathcal{P} e \mathcal{FP} hanno una definizione robusta che non cambia con il modello di calcolo (non importa se sia su RAM o DTM).

Inoltre \mathcal{P} e \mathcal{FP} contengono molti problemi fondamentali, come ordinamento, prodotto di matrici, alcuni problemi di ottimizzazione, ...

Purtroppo alcuni problemi interessanti non hanno ancora soluzioni efficienti e stanno in

- \mathcal{NP} = problemi di decisione risolti in tempi polinomiale su una DTM non deterministica

Si congettura $\mathcal{P} \neq \mathcal{NP}$, ma non è ancora noto (non siamo certi che i problemi attualmente non risolvibili in tempo polinomiale non abbiano una soluzione efficiente).

I problemi che possono essere parallelizzati non possono appartenere ad \mathcal{NP} , altrimenti sarebbe come provare che $\mathcal{P} = \mathcal{NP}$.

All'atto pratico siamo interessati a una **sottoclasse di \mathcal{P}**

1. La **valutazione asintotica** dei tempi può **nascondere costanti o termini** che all'atto pratico fanno la differenza. Esempio: a volte si preferisce il QuickSort ($O(n^2)$) rispetto al MergeSort ($O(n \log n)$).
2. Da un punto di vista pratico il **grado del polinomio deve essere basso**. Es: n^{1000} anche su piccole istanze risulta peggiore di un esponenziale. Esempio pratico: in $O(n^6)$ è il tempo per un test di primalità, ma il grado è troppo alto, quindi si preferisce un test probabilistico con possibilità di errore.

1 Algoritmi Paralleli

Possono essere affrontate 3 diverse problematiche:

- **Sintesi:** progettazione degli algoritmi
- **Valutazione:** delle prestazioni degli algoritmi
- **Universalità:** quali problemi sono efficientemente risolubili e quali no

Sintesi: Come costruire algoritmi paralleli? Ci si può ispirare ad algoritmi sequenziali, ma quasi mai si può riciclare l'intero algoritmo.

Esempio: sommare due numeri binari. L'algoritmo sequenziale ottimo somma i bit per colonna.

Porta a un buon algoritmo parallelo? Ogni processore si può occupare di una colonna.

Sorge un problema con il riporto, il quale sostanzialmente fa tornare l'algoritmo a uno sequenziale, con in più uno spreco ingiustificato di hardware (ogni processore deve aspettare quello prima...).

Ma esistono algoritmi paralleli efficienti per sommare numeri binari.

Esempio 2: ordinamento. L'algoritmo sequenziale ottimo è il MergeSort.

Ma questo non porta a un buon algoritmo parallelo, mentre esistono algoritmi paralleli efficienti per l'ordinamento.

Servono nuove tecniche e punti di vista totalmente diversi da quelli usati per gli algoritmi sequenziali.

Valutazione delle prestazioni: Come si misurano le prestazioni degli algoritmi paralleli? Vanno definiti:

- **Tempo impiegato**
- **Hardware occupato** = numero di processori

Cos'è efficiente nel caso di **algoritmi paralleli**?

Si ha un **parametro** $E = \text{efficienza}$. Tiene conto sia del tempo, sia del numero di processori, permettendoci di dire se l'uso aggiuntivo di hardware è giustificato o meno.

Universalità: Si tratta di una problematica teorica, riesco a caratterizzare la classe dei problemi che ammettono algoritmi paralleli efficienti?

Nel caso sequenziale la classe di problemi si chiama \mathcal{FP} . Nel caso parallelo si **definisce la classe** \mathcal{NC} .

\mathcal{NC} = coincide con problemi risolti da algoritmi paralleli efficienti, ovvero con:

- **Tempo polilogaritmico** (quindi meglio che polinomiale, altrimenti userei un algoritmo sequenziale efficiente, dato anch'esso sarebbe polinomiale).
- **Hardware polinomiale** (richiede un numero di processori che scala polinomialmente con la lunghezza dell'input).

Esempi di problemi in \mathcal{NC} :

- somma di due numeri
- ordinamento

In realtà è ragionevole una sottoclasse di \mathcal{NC} , come per \mathcal{P} , se il grado del polinomio o le costanti nascoste sono alte, il tempo è sì polinomiale ma rimane poco pratico.

Si può **dimostrare che** $\mathcal{NC} \subseteq \mathcal{FP}$.

I problemi che possiamo parallelizzare efficientemente rientrano nella classe di problemi risolvibili con algoritmi di calcolo sequenziali efficienti.

Per dimostrarlo: per ottenere un algoritmo sequenziale a partire da uno parallelo basta eseguire le istruzioni, che nel parallelo sono divise su vari processori, in fila su un processore singolo, mantenendo il tempo polinomiale (tempo polilogaritmico moltiplicato per il numero polinomiale dei processori, quindi dominato ancora da un polinomiale) e ottenendo un algoritmo sequenziale.

Ci si può chiedere se $\mathcal{NC} = \mathcal{FP}$, ma è un problema aperto.

Si pensa di no, la questione riprende il problema “ogni algoritmo sequenziale efficiente è parallelizzabile?” Esiste un metodo per trasformare ogni algoritmo sequenziale in un algoritmo parallelo?

Se $\mathcal{NC} \neq \mathcal{FP} \implies \mathcal{FP} \setminus \mathcal{NC} =$ problemi \mathcal{P} –completi; esistono problemi in \mathcal{P} ma non in \mathcal{NC} , chiamati problemi \mathcal{P} –completi, problemi tali che la loro appartenenza a \mathcal{NC} farebbe collassare \mathcal{NC} e \mathcal{FP} (ovvero $\mathcal{NC} = \mathcal{FP}$; un problema \mathcal{P} –completo permetterebbe di risolvere tutti gli altri problemi di \mathcal{FP} , facendo coincidere le due classi).

Pertanto non sarà possibile parallelizzare i problemi \mathcal{P} –completi (duh).

L’universalità non sarà tratta ulteriormente in questo corso.

Nel mondo parallelo bisogna:

- fissare un modello di calcolo teorico da mappare su architetture reali
- fissare le risorse computazionali, ovvero tempo e numero di processori
- Fissare il parametro efficienza, per dire se una soluzione parallela è preferibile ad una efficiente soluzione sequenziale

1.1 Tipi di architetture parallele

Ci sono due tipologie:

- a **memoria condivisa**
- a **memoria distribuita**

Memoria condivisa: Ci sono dei processori che si affacciano su una **memoria centrale condivisa tra tutti**. Si ha un **clock centrale**.

La **comunicazione avviene in tempo costante**: avendo due processori P_i e P_j , P_j vuole comunicare un dato x a P_i

- P_j scrive x in memoria centrale
- P_i legge x dalla memoria centrale

In questo modo in 2 cicli di clock avviene la comunicazione. Si dice “comunicazione costante” perché non importa la dimensione del dato da trasferire. Questa è l’attuale architettura di CPU multicore.

Questa architettura permette una **forte parallelizzazione**, dato che i tempi di comunicazione sono trascurabili, rimuovendo quindi il costo dovuto alla comunicazione (o comunque se considerato, è minimo).

Memoria distribuita: Da non confondere con “architetture distribuite”, questa è un’architettura parallela a memoria distribuita.

Si ha sempre un unico **clock centrale**, processori sincroni, ma **ognuno di questi ha una memoria privata** e sono **collegati tra loro** tramite fili (con qualche topologia, solitamente non fully connected), si ha una **rete di interconnessione tra i processori**.

La comunicazione dipende dalla “distanza” tra i processori: se P_j vuole comunicare x a P_i dobbiamo chiederci quanti processori collegano P_j a P_i . Se sono collegati direttamente allora il tempo è costante.

La rete di interconnessione è fissa, può essere:

- **array lineare:** processori collegati su una linea
- **mesh:** collegati su una matrice, sia per riga che per colonna
- **ipercubo:** come prima, ma aumentando le dimensioni

1.1.1 PRAM

Parallel RAM, sta ad indicare che si hanno **RAM (Random Access Machine) che lavorano in parallelo**, si tratta di un modello teorico ma in pratica lo si ritrova nelle CPU multicore.

Si ha un **determinato numero di processori** ed una **memoria centrale** M , la quale i – lesima cella viene indicata con $M[i]$. All’inizio contiene l’input, alla fine può contenere l’output (che può anche essere in uno dei processori).

Ogni processore è una RAM (= Random Access Machine) sequenziale e mettendole assieme queste possono lavorare in parallelo.

Ogni processore è costituito da una unità di calcolo e dei registri, i quali costituiscono la memoria privata del processore, chiamati $R[0], R[1], \dots$

Tipi di istruzioni dei processori P_i :

- Operazioni aritmetico/logiche.
- Istruzioni da/per la memoria centrale
 - **store** $R[k] \ M[h]$: mette il dato nella memoria centrale
 - **load** $R[k'] \ M[h']$: carico nel registro dati dalla memoria centrale alla memoria privata

Un processore non può usare i dati x e y se questi sono in memoria centrale, vanno prima caricati nella memoria privata (la ALU è collegata solo ai registri). **Ogni processore può effettuare operazioni solo sulla propria memoria privata.**

Esempio P_j vuole comunicare a P_i il contenuto di $R[t]$. Istruzioni:

$$\begin{array}{lll} P_j : & \text{store} & R[t] \ M[x] \\ P_i : & \text{load} & R[s] \ M[x] \end{array}$$

Ovviamente sia **store** che **load** devono lavorare sulla stessa cella $M[x]$ della memoria centrale.

La comunicazione avviene in tempo $O(1)$.

Come si programma con una PRAM? Si ha un **unico programma** per tutti i processori ed una **memoria condivisa**:

1. Il tempo per ogni P_i è scandito dal clock centrale
2. Ogni P_i esegue la “stessa istruzione”

Cosa si intende con “la stessa istruzione”? Forma dell'istruzione:

```
for all  $i \in I$  par do  
    istruzione $_i$ 
```

Con I definito come l'insieme di indici dei processori che devono eseguire quella istruzione. Si ha un **for** parallelo, quindi tempo costante e non n passi diversi, ogni istruzione è indicizzata con l'indice del processore.

I processori con indice in I eseguono **istruzione $_i$** . I processori con indice NON in I eseguono l'istruzione nulla.

Cosa si intende con “istruzione $_i$ ”? Dipende dal tipo di architettura:

- **SIMD**: Single Instruction Multiple Data
- **MIMD**: Multiple Instruction Multiple Data

SIMD: L'istruzione eseguita è la **stessa per ogni processore** $\in I$, ma ognuno di questi la effettua **su dati diversi**.

In funzione della **capacità di accedere alla memoria** M si possono avere più architetture:

1. **EREW**: Exclusive Read Exclusive Write, scrittura e lettura sono esclusivi, quindi una stessa cella di memoria può essere letta o scritta da un solo processore per volta
2. **CREW**: Concurrent Read Exclusive Write, la lettura può essere simultanea, la scrittura è esclusiva
3. **CRCW**: Concurrent Read Concurrent Write, lettura e scrittura possono essere simultanee

Per la **scrittura simultanea servono politiche di accesso** (se più processori vogliono scrivere sulla stessa cella, chi vince?):

- **common:** i processori possono scrivere solo lo stesso dato, pena l'arresto del sistema
- **random:** si sceglie un P_i a caso
- **max/min:** vince il P_i con il dato max/min
- **priority:** vince il P_i con priorità massima (più alta, va definita una priorità)

Indicando con i numeri 1,2,3 le architetture come sopra

$$Alg(1) \implies Alg(2) \implies Alg(3)$$

Ovvero un algoritmo EREW funziona su architetture CREW, ed un CREW funziona su architetture di tipo CRCW. Questo è abbastanza ovvio, ma vale il contrario? Di base no, ma si possono trasformare CRCW in CREW ed eventualmente anche EREW.

L'architettura più ragionevole e più semplice è EREW.

Esempio di algoritmo su PRAM di tipo CREW:

- numero di processori = n
- dimensione input = n (lunghezza dell'input pari al numero di processori)
- assumiamo input array A con valori distinti

Volendo cercare in A un valore particolare:

```
Cerca (A, n, x) {  
    indice=-1  
    for i=0 to n-1 pardo  
         $P_i$ :  
            if A[i]=x then  
                indice=i  
    return indice  
}
```

Ogni processore di carica nella memoria locale il un dato di A e lo confronta con il valore x cercato (il processore 1 controlla la cella 1, il 2 la cella 2, ...). La variabile **indice** parte $=-1$, ma viene settato quando trovato il valore corretto, restituito alla fine.

Si ha tempo parallelo e costante. Quanti clock sono? Vanno effettuati: `load x`, `load A[i]`, il confronto ed eventualmente la scrittura della variabile `indice`, quindi 4 cicli di clock.

Questo algoritmo è concurrent read perché la x viene letta assieme da tutti, mentre su A vengono lette sempre celle diverse. Exclusive write perché la scrittura invece viene effettuata da un solo processore, solo quando viene scritto l'indice, con il presupposto che A contenga solo valori distinti.

Se A può contenere elementi ripetuti allora questo algoritmo diventa CRCW, dato che posso trovare più valori pari a x tutti nello stesso momento.

Risorse di calcolo:

- nel **sequenziale**: $t(n)$ (tempo), $s(n)$ (spazio)
- nel **parallelo**: $p(n)$ (numero di processori), $T(n, p(n))$ (tempo, in funzione di input e numero di processori)

Definizioni informali:

- $p(n)$: numero di processori richiesti su input di lunghezza n (nel caso peggiore)
- $T(n, p(n))$: tempo richiesto da un input di lunghezza n su $p(n)$ processori. Di conseguenza $T(n, 1)$ è il tempo sequenziale

Valutazione precisa del tempo $T(n, p(n))$: Possiamo pensare all'esecuzione del programma con una sorta di matrice con:

- colonne che rappresentano ognuna uno dei $p(n)$ processori
- righe che rappresentano ognuna un passo parallelo

Quindi nella prima riga ho il primo passo parallelo e sotto la prima colonna avrò l'istruzione eseguita dal primo processore.

Il numero dei passi generalmente dipende da n , per cui diventa una funzione $k(n)$.

Il costo, in termini di tempo, per il primo processore al primo step del programma viene indicato con $t_1^{(1)}(n)$, dove la t è piccola perché è un tempo sequenziale, all'apice si trova il numero del processore e al pedice il numero del passaggio.

Vogliamo **definire $T(n, p(n))$ in funzione** di questi **tempi richiesti dalle singole RAM**, ovvero la **somma dei tempi richiesti da ognuno dei $k(n)$ passi**. Supponendo un'architettura MIMD, istruzioni diverse potrebbero richiedere tempo diverso, quindi a **ogni passo dobbiamo considerare il tempo massimo** impiegato da un singolo processore; il massimo per ogni riga, per ogni t_1 .

Indichiamo il tempo impiegato al passo i -esimo con

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

Di conseguenza:

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

Il **tempo parallelo** è la somma del tempo a ogni passaggio:

- **T dipende da $k(n)$** (devo sommare $k(n)$ passi)
- **T dipende anche dalla dimensione dell'input** (costo logaritmico/uniforme, il tempo sequenziale a ogni passo dipende dalla dimensione dell'input in quanto elaborato da macchine sequenziali)
- **T dipende da $p(n)$** (le operazioni da eseguire sono sempre quelle, se i processori sono troppo pochi devo assegnare le operazioni rimanenti in coda ad altre su altri processori; eventualmente alcuni processori in un solo step dovranno svolgere più operazioni)

1.2 Broadcast P-RAM

Al posto di processori consideriamo entità, che collaborano tra loro ed una informazione i , detenuta da una sola entità, deve essere conosciuta da tutte le altre entità che compongono il sistema. Broadcast o replica, si ha un array in memoria centrale/condivisa in cui **una cella contiene il valore x** e vogliamo che **nelle celle seguenti compaia il valore x per un certo numero di elementi n** , in modo che gli altri processori possano conoscere l'informazione nella loro cella associata. Le celle della memoria condivisa vengono indicate come A .

```

Broadcast ( $x$ )
{
 $P_0$  :       $A[0] = x$ 
          for  $i = 0$  to  $\log n - 1$  do
                for  $j = 2^i$  to  $2^{i+1} - 1$  par do
 $P_j$  :       $A[j] = A[j - 2^i]$ 
          }
```

Per la condivisione ci sono $\log n$ passi paralleli, un certo numero di processori prendono il valore x e lo copiano in una delle celle successive (ogni processore lo passa ad un'altra cella, raddoppiando il numero di celle contenenti x ogni volta e ripetendo), rendendo il numero di passaggi richiesti logaritmico rispetto al numero n di copie dei dati da avere.

Questo processo potrebbe essere eseguito perché in un programma EREW ogni processore deve accedere ad una sua copia del dato non avendo letture concorrenti.

Il broadcast può essere usato per risolvere **la ricerca**:

```

Cerca (a, n, x)
{
    Broadcast(x)
    Indice=-1
    for i = 0 to n-1 par do
Pi :         if A[i]=x[i] then indice=i
    return indice
}
```

Faccio il broadcast del dato x e lo faccio cercare da tutti i processori, restituendo l'indice i del dato, se trovato. Di conseguenza impiega tempo $\log n$ (solamente 1 passo per la ricerca, $\log n$ per il broadcast).

Se tutti gli indici sono distinti l'algoritmo è EREW, altrimenti ERCW (ma solitamente si usa solo CRCW, non ha senso avere scritture concorrenti e letture esclusive). ERCW può essere trasformato in EREW (con un aumento della funzione tempo).

1.3 Efficienza

Per progettare algoritmi paralleli è necessario stabilire cosa significhi in questo contesto “buone prestazioni”, in modo da essere sicuri che questi migliorino effettivamente gli algoritmi sequenziali.

Bisogna valutare la funzione tempo dell'algoritmo parallelo $T(n, p(n))$ in confronto al tempo sequenziale $T(n, 1)$: se il tempo parallelo ha un tasso di crescita migliore di quello sequenziale allora l'algoritmo parallelo ha migliorato la soluzione.

Per il confronto tra i due tempo ci sono due possibilità:

- Il tempo **parallelo** è **circa uguale** al tempo **sequenziale**

$$T(n, p(n)) = \Theta(T(n, 1))$$

Il tempo parallelo è limitato sia superiormente che inferiormente dal tempo sequenziale, **stesso tasso di crescita**. Questo è il caso **da evitare**. Il confronto dei tempi tramite parametro efficienza esclude la possibilità di ricadere in questo caso.

- Il tempo **parallelo** **cresce più lentamente** del **tempo sequenziale**

$$T(n, p(n)) = o(T(n, 1))$$

Viene **limitato superiormente** (in qualche modo) dal **tempo sequenziale**, la funzione del tempo parallelo è sempre più piccola di quello sequenziale. Questo è il caso **da preferire**, il parametro efficienza vuole questa situazione.

1.3.1 Speed-up

Parametro che serve a definire l'efficienza, si tratta del **rapporto tra il tempo sequenziale per risolvere il problema e il tempo richiesto dall'algoritmo parallelo**:

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

Esempio: se $S = 4$ l'algoritmo parallelo è 4 volte più veloce del sequenziale.

Ma in realtà se desse 4 sarebbe un caso in cui il parallelo è un Θ del sequenziale. Ponendoci nel **caso in cui** $T(n, p(n)) = o(T(n, 1))$ lo speed-up **tende a infinito**

$$S(n, p(n)) \rightarrow +\infty$$

Con un rapporto dei tempi (questo parametro), viene considerato il numero di processori? In realtà no, **non considera il numero di processori**, potremmo averne utilizzata una quantità troppo elevata/non vantaggiosa, ma questo parametro non fornisce modo di saperlo, sappiamo il rapporto tra i tempi ma non riusciamo a considerare se il numero di processori è adeguato.

Per questa mancanza, il parametro di speed-up viene poi **usato solamente per definire il parametro efficienza**.

Esempio di un problema: Soddisfacimento di formule (dove la lunghezza della formula è legata linearmente al numero di variabili coinvolte); si tratta di un problema $\in \mathcal{NP}$:

$$T(n, 1) = 2^n$$

Utilizzando un processore per ogni possibile assegnamento (sostanzialmente il funzionamento di una macchina non deterministica):

$$T(n, p(n)) = O(n)$$

$$S(n, p(n)) = \frac{2^n}{n} \rightarrow +\infty \text{ ma } p(n) = 2^n$$

Tecnicamente lo speed-up tende a infinito, ma il numero di processori è decisamente non polinomiale, costo hardware esponenziale.

1.3.2 Definizione di Efficienza

Il parametro efficienza è definito come il **rapporto tra lo speed-up e il numero di processori** $p(n)$

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

Viene usato $T(n, 1)^*$, ovvero il miglior tempo sequenziale conosciuto per risolvere il problema o un lower bound di quest'ultimo, giusto per essere più cattivi con il nostro algoritmo parallelo.

Il parametro efficienza **può assumere i valori**

$$0 \leq E(n, p(n)) \leq 1$$

Che sia sempre ≥ 0 lo si vede dalla formula, essendo un rapporto tra valori sempre positivi.

Esempio per il problema di soddisfacimento di formule:

$$E(n, p(n)) = \frac{2^n}{2^n \cdot n} = \frac{1}{n} \rightarrow 0$$

Principio di Wyllie: Quando l'efficienza tende a 0 $E \rightarrow 0$, l'**utilizzo di processori è troppo elevato**, che magari rimarranno utilizzati la maggior parte del tempo.

Che l'efficienza sia sempre ≤ 1 invece lo si può intuire dal fatto che è un rapporto tra tempo sequenziale e tempo parallelo moltiplicato per numero di processori. Portato all'estremo, ipotizzando di poter usare n processori per risolvere in 1 passo un problema con input n il tempo sequenziale dello stesso algoritmo diventerebbe n (mettere tutti i passi uno dopo l'altro), il numero di processori è n mentre il tempo parallelo sarà 1, quindi diventa n/n (inventato da me, sotto spiegazione un po' più formale).

Per dimostrarlo, chiamo $\tilde{T}(n, 1)$ la versione sequenziale dell'algoritmo parallelo (tutti i passaggi svolti in fila), di conseguenza

$$T(n, 1) \leq \tilde{T}(n, 1) \leq p(n)t_1(n) + p(n)t_2(n) + \dots + p(n)t_{k(n)}(n)$$

l'algoritmo sequenziale “normale” è sicuramente \leq del nostro \tilde{T} , che impiega al massimo la somma del tempo di ogni passaggio moltiplicato per il numero di processori, ma questa è la definizione del tempo parallelo.

$$p(n)t_1(n) + p(n)t_2(n) + \dots + p(n)t_{k(n)}(n) = p(n) \sum_{i=1}^{k(n)} t_i(n) = p(n)T(n, p(n))$$

Il miglior tempo sequenziale è limitato superiormente dal miglior tempo parallelo moltiplicato per il numero di processori

$$T(n, 1) \leq p(n) \cdot T(n, p(n))$$

E da qua possiamo ricavare

$$\frac{T(n, 1)}{p(n)} \leq T(n, p(n))$$

Il tempo parallelo $T(n, p(n))$ ha un lower bound, ovvero il rapporto tra il miglior tempo sequenziale e il numero di processori. Il meglio che posso fare con un algoritmo parallelo è distribuire equamente il lavoro tra tutti i processori.

Spostando anche il tempo parallelo nella disequazione si ottiene

$$\frac{T(n, 1)}{p(n)T(n, p(n))} \leq 1 \implies E(n, p(n)) \leq 1$$

l'efficienza è ≤ 1 .

Di conseguenza **l'efficienza è un parametro che varia tra 0 e 1:**

- Se $E \rightarrow 0$ non va bene: implica che $p(n)$ cresce troppo velocemente, dato che $T(n, p(n)) = o(T(n, 1))$
- Il meglio da avere è $E \rightarrow k \leq 1$ dove k è una costante

Altro modo per dimostrare che $E \leq 1$, basandosi sull'idea che se $E \rightarrow 0$ allora per migliorare l'algoritmo provo a ridurre $p(n)$ senza degradare il tempo. Cambio il numero di processori da p a p/k .

Ricetta presa dalla PhD thesis di J. Wyllie.

Su un algoritmo generico, ridurre i processori a p/k significa che a ogni passo ogni processore deve svolgere in sequenza le operazioni di k processori, sostanzialmente vengono raggruppate le operazioni k alla volta. Il tempo di ogni passo sarà limitato superiormente da $k \cdot t_i(n)$ (k il tempo di ogni passo, sicuramente sarà maggiore in quanto il tempo di ogni passo è definito come il max di tutti).

Il tempo parallelo richiesto dall'algoritmo con p/k processori è limitato superiormente dalla somma dei tempi all' i -esimo passo:

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

Quindi si ha che

$$T(n, p/k) \leq k \cdot T(n, p)$$

Di conseguenza il tempo con p/k processori è limitato superiormente dal tempo che usa tutti i p processori, moltiplicato per k .

Partendo da questa disuguaglianza si può far vedere che **l'efficienza E cresce al diminuire dei processori.**

Quanto vale $E(n, p/k)$?

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

Sostituisco $T(n, p/k)$ con $k \cdot T(n, p)$ (ricordando che $T(n, p/k) \leq k \cdot T(n, p)$). Abbiamo sostituito un valore più grande, dividendo per un valore più grande ottengo un valore più piccolo, quindi la disuguaglianza diventa \geq .

$$E(N, p/k) \geq E(n, p)$$

La formula mostra che **diminuendo i processori migliora il parametro efficienza** (aumenta, rendendo vera la ricetta di Wyllie). Se il miglioramento è significativo è un altro discorso.

Considerando $k \rightarrow p$

$$1 = E(n, 1) = E(n, p/p) \geq E(n, p/k) \geq E(n, p)$$

Quindi si ottiene l'efficienza di un algoritmo sequenziale. Attenzione a mantenere $T(n, p/k) = o(T(n, 1))$ (perché $E(n, 1) = 1$, ma $T(n, p = 1) = T(n, 1)$ cioè sequenziale).

1.4 Problema Sommatoria

Perché prendiamo esempi di problemi? Motivazioni:

- **Tecnica:** scomposizione del problema in sottoproblemi e fusione dei risultati
- **Schema:** “imparando” la sommatoria posso adattare la soluzione per altre operazioni associative
- **Modulo:** sotto-problema di altri problemi più grandi

Problema della sommatoria:

- **Input:** $M[1], M[2], \dots, M[n]$, n elementi in n celle della memoria
- **Output:** $M[n] = \sum_{i=1}^n M[i]$, nell’ultima cella verrà scritta la somma di tutti i valori (presupponendo non sia importante mantenere le informazioni preesistenti nelle celle)

Algoritmo sequenziale:

```
for i = 1 to n - 1 do:  
    M[n] = M[n] + M[i]
```

Accumulo tutto in $M[n]$ sequenzialmente. Il tempo diventa $T(n, 1) = n - 1$ passaggi.

L’idea per parallelizzare è “*una somma a processore*”: somme a due a due, ogni processore somma una coppia di valori e ripeto il processo fino ad ottenere la somma finale. Servono quindi $\log n$ passi per sommare n elementi.

Questo **funziona** solo **perché** il $+$ è **associativo**:

$$((a + b) + c) + d = (a + b) + (c + d)$$

Quindi questo **schema può essere applicato ad ogni altra operazione associativa**.

Al primo passo sommo elementi a distanza 1, al secondo passo distanza 2, al terzo passo distanza 4, ..., all'ultimo passo rimarranno 2 valori da sommare a distanza $n/2$; la distanza raddoppia ad ogni passo. Ogni risultato parziale viene memorizzato nella cella di indice più alto.

Istruzioni (con $k =$ numero del processore):

$$\begin{aligned} 1^\circ \text{ passo: } & M[2k] = M[2k] + M[2k-1] & 1 \leq k \leq \frac{n}{2} \\ 2^\circ \text{ passo: } & M[4k] = M[4k] + M[4k-2] & 1 \leq k \leq \frac{n}{4} \\ & \dots \\ \log n^\circ \text{ passo: } & M[n] = M[n] + M[n/2] & 1 \end{aligned}$$

Pseudo-codice:

```

for j = 1 to log n
    for k = 1 to n/2j par do
        M[2jk] = M[2jk] + M[2jk - 2j-1]
return M[n]

```

Questo algoritmo è EREW? Dobbiamo valutare se l'uso delle celle è **esclusivo**, ovvero se ci sono letture/scritture simultanee.

Considerando due processori a, b con $a \neq b$, a questo scopo dobbiamo verificare che $2^j a$, $2^j a - 2^{j-1}$, $2^j b$ e $2^j b - 2^{j-1}$ siano tutti diversi tra loro.

I confronti possibili quindi sono:

- $2^j a \neq 2^j b$, che vale per ogni $a \neq b$
- $2^j a \neq 2^j b - 2^{j-1}$ ovvero $2a = 2b - 1$ e non esistono valori $\in \mathbb{N}$ che soddisfano l'equazione

Quindi abbiamo dimostrato che è **un algoritmo EREW**.

Dimostrare la correttezza: Per la correttezza dobbiamo **dimostrare la proprietà**:

$$M[2^j k] = M[2^j k] + \dots + M[2^j(k-1) + 2] + M[2^j(k-1) + 1]$$

Nelle celle multiple di 2^j (con $1 \leq j \leq \log n$) ci sono 2^j valori sommati, ovvero quelli appartenenti a tutte le 2^j celle precedenti.

Per $j = \log n$ e di conseguenza $k = 1$ la proprietà sopra risulta:

$$M[n] = M[n] + \dots + M[1]$$

In $M[n]$ ci va la somma degli n precedenti valori.

Si può dimostrare **per induzione**:

- **Caso base:** $j = 1$ e $1 \leq k \leq \frac{n}{2}$. L'istruzione dell'algoritmo è

$$M[2k] = M[2k] + M[2k - 1]$$

Quindi per $j = 1$ è facile verificare che la proprietà è vera.

- **Induzione:** si presuppone che la proprietà sia vera per $j - 1$ e si dimostra che vale per j . Consideriamo l'istruzione del programma al passo j

$$M[2^j k] = M[2^j k] + M[2^j k - 2^{j-1}]$$

e dobbiamo dimostrare che questa è in realtà un'istruzione che mette nella cella di memoria $2^j k$ la somma dei suoi precedenti 2^j valori.

Raccolgo 2^{j-1} ottenendo $M[2^{j-1} 2k]$, quindi per la proprietà e per ipotesi induttiva posso dire che:

$$M[2^{j-1} 2k] = M[2^{j-1} 2k] + \dots + M[2^{j-1}(2k-1) + 1]$$

Quindi dentro $M[2^{j-1} 2k]$ sono presenti tutti i valori precedenti con un fattore moltiplicativo $(2k-1)$; è uguale alla proprietà iniziale, cambia solo che al posto di k è presente $2k$. Nella cella di indice $2k$ abbiamo la somma di tutti i precedenti 2^{j-1} valori.

Raccogliendo 2^{j-1} anche da $M[2^j k - 2^{j-1}]$ risulta $M[2^{j-1}(2k - 1)]$, che come prima può essere visto come un multiplo di 2^{j-1} , quindi applichiamo ancora la proprietà per ipotesi di induzione:

$$M[2^{j-1}(2k - 1)] = M[2^{j-1}(2k - 1)] + \dots + M[2^{j-1}(2k - 2) + 1]$$

Allora abbiamo che la cella $M[2^{j-1}(2k - 1)]$ contiene tutte le precedenti 2^{j-1} celle.

Quindi abbiamo che in $M[2^j k]$ sono contenuti 2^{j-1} valori, così come in $M[2^j k - 2^{j-1}]$, ovvero le due celle considerate all'inizio. Il totale degli elementi sommati quindi è $2^{j-1} + 2^{j-1} = 2^j$. Di conseguenza in $M[2^j k]$ vengono sommati 2^j valori.

Possiamo verificare che siano tutti valori differenti notando che sono tutti valori in sequenze decrescenti, ma l'ultimo della prima uguaglianza e il primo della seconda sono rispettivamente $M[2^{j-1}(2k - 1) + 1]$ e $M[2^{j-1}(2k - 1)]$ ovvero il secondo valore è il successivo del primo.

$$M[2^{j-1}(2k - 1) + 1] \xleftrightarrow[\text{precedente}]{\text{successivo}} M[2^{j-1}(2k - 1)]$$

Essendo sequenze decrescenti abbiamo verificato che siano tutti i valori diversi, quindi la somma delle due celle racchiude tutti i 2^j valori.

Di conseguenza

$$M[2^j k] = M[2^j k] + \dots + M[2^j(k - 1) + 1]$$

è vera.

1.4.1 Valutazione dell'algoritmo

Numero di processori: Il numero di processori impiegati è dato dal livello più “costoso”, ovvero il primo, nel quale ne vengono usati $n/2$ (anche se a ogni passo ne verranno usati la metà di quello precedente)

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

Tempo: Il tempo impiegato deve considerare, oltre al numero di passaggi, le microistruzioni necessarie:

- LD per caricare il primo numero
- LD per caricare il secondo numero
- ADD per sommare i numeri
- ST per rimettere il numero in memoria centrale

ovvero 4 operazioni per $\log n$ passaggi, quindi

$$T(n, n/2) = 4 \log n$$

Se n non è potenza di 2 bisogna allungare l'input con degli 0 fino ad arrivare al multiplo di 2 successivo ad n . Non può peggiorare drasticamente le prestazioni in quanto in binario, chiamando t il numero di bit necessari per rappresentare n , la potenza di 2 più vicina ad n diventa t zeri con in testa un 1 e quindi sarà sempre compresa tra n e $2n$. Nel peggiore dei casi l'input si raddoppia.

Quindi le valutazioni viste per n potenza di 2 valgono anche nelle altre casistiche, però:

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

$$T(n, n) = 4 \log 2n \leq 5 \log n$$

$$\implies p(n) = O(n) \quad \text{e} \quad T(n, n) = O(\log n)$$

Efficienza: Possiamo calcolare l'efficienza

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

I processori sono un po' “sprecati”, in quanto vengono utilizzati tutti solamente al primo passo.

Proviamo a vedere se, applicando Wyllie, si può **diminuire il numero di processori** fino a sub-lineare, quindi con $p(n) = o(n)$, **mantenendo un tempo logaritmico**, anche con diverso coefficiente. Questo per avere una efficienza che tende ad una costante diversa da zero, $E \rightarrow c \neq 0$.

Quindi partendo da n numeri, al posto di avere $n/2$ processori ne abbiamo p (per ora indefinito). Ognuno dei p processori dovrà **farsi carico di** una certa quantità di **numeri da sommare** $\Delta = n/p$. Ogni processore effettuerà la somma di Δ elementi.

I **risultati** verranno messi nella **cella coinvolta dalla somma** con **indice più alto**, quindi $M[\Delta], M[2\Delta], \dots, M[p\Delta]$.

Con k indice del processore, 1° passo parallelo:

$$M[k\Delta] = M[k\Delta] + \dots + M[(k-1)\Delta + 1] \quad 1 \leq k \leq p$$

Quindi nella cella $M[k\Delta]$ verrà messa la somma dei Δ numeri affidati al processore k .

Passi **paralleli successivi**: i risultati parziali sono nelle celle

$$M[\Delta], M[2\Delta], \dots, M[p\Delta]$$

A queste celle applichiamo l'algoritmo sommatoria visto prima, in modo tale che $M[p\Delta] = M[n] = \sum_i M[i]$, ovvero le somme di tutti i gruppetti parziali. Questo **dimostra** la **correttezza** dell'algoritmo.

Di conseguenza applico l'algoritmo precedente per la sommatoria alle p somme parziali ottenute. Tempo n/p per il primo passo e $5 \log p$ per i restanti.

Valutazione: Per il nuovo algoritmo:

- I **processori** sono

$$p(n) = p$$

- Il **tempo** diventa

$$T(n, p) = T(1^\circ \text{ passo}) + T(\text{passi succ.}) = \frac{n}{p} + 5 \log p$$

Di conseguenza l'**efficienza**

$$E(n, p) = \frac{n-1}{p \left(\frac{n}{p} + 5 \log p \right)} = \frac{n-1}{n + 5p \log p} \rightarrow c \neq 0$$

Non abbiamo dato un valore a p , ma sarebbe carino se $5p \log p = n$ in modo tale che $n + n$ al denominatore abbatta il tempo sequenziale sopra, **tendendo a una costante** $c \neq 0$. Presupponendo $5p \log p = n$:

$$E(n, p) \sim \frac{n}{2n} \rightarrow \frac{1}{2}$$

Per fissare p :

$$p \log p = \frac{n}{5} \implies p = \frac{n}{5 \log n}$$

e si può facilmente verificare che il valore è corretto.

Ricapitolando: i valori diventano:

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

$$T(n, p(n)) = 5 \log n + 5 \log n - \dots \leq 10 \log n$$

Quindi il numero di processori è sub-lineare, mantenendo il tempo logaritmico, seppur con una diversa costante.

Si può fare di meglio? Si può migliorare l'algoritmo per la sommatoria? Si può dimostrare un lower bound sfruttando il fatto che un algoritmo per la sommatoria non è altro che un albero binario e può essere sempre rappresentato come tale. Il livello con più nodi determina il numero di processori, mentre l'altezza dell'albero determina il tempo.

Di conseguenza **meglio di $\log n$ non si può fare.**

1.4.2 Sommatoria come schema per altri problemi

Una operazione iterata, con la proprietà di essere associativa:

- Input $M[1], \dots, M[n]$
- Output $Op_i M[i] \rightarrow M[n]$, ovvero effettuo l'operazione su tutti gli elementi e metto il risultato in $M[n]$

Esempi di operazioni associative: $+, *, \wedge, \vee, \oplus$.

Questo permette di avere una soluzione efficiente parallela:

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

Con modelli di PRAM più potenti si possono ottenere tempi migliori, fino a costante.

Problema dell'AND-iterato:

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

Programma

```
for  $i \leq k \leq n$  par do
  if  $M[k] = 0$  then
     $M[n] = 0$ 
```

Sostanzialmente, mi basta uno 0 ed il risultato diventa 0.

Per questo programma però serve un'architettura **CRCW** (lettura esclusiva in realtà), in quanto potenzialmente **più processori** dovranno **accedere** alla **cella** $M[n]$. Si può usare la politica common.

Le **prestazioni** che ne risultano:

$$p(n) = n$$

$$T(n, n) = 3$$

$$E(n, n) = \frac{n-1}{n \cdot 3} \rightarrow \frac{1}{3}$$

Si può usare lo stesso programma per l'OR, basta cambiare 0 con 1 e diventa “basta un 1 per far diventare 1 il risultato”.

La sommatoria può essere un sottoproblema di altri:

- prodotto interno di vettori: $\langle \dots, \dots \rangle$
- prodotto matrice-vettore
- prodotto matrice-matrice
- potenza di una matrice

Prodotto interno di vettori:

- Input: $x, y \in \mathbb{N}^n$, due vettori di n elementi
- Output: il prodotto interno dei due vettori, $\langle x, y \rangle = \sum_{i=1}^n x_i * y_i$

Sono n prodotti e sommarne i risultati, quindi $n - 1$ somme, il tempo sequenziale risulta $2n - 1$.

La soluzione EREW è applicare il **modulo sommatoria** con in **input** ad **ogni processore** un **gruppo di** $\Delta = n/p$ coppie di $x_i * y_i$ (sommatoria parallela con la riduzione dei processori descritta prima).

- Fase I: Δ prodotti in sequenza per processore e somma sequenziale
- Fase II: somma di p numeri in parallelo

Per la sommatoria (Fase II) serve un numero di processori:

$$p = c_1 \left(\frac{n}{\log n} \right) \quad t_{II} = c_2 \log n$$

Mentre per la Fase I:

$$p \sim \frac{n}{\log n} \implies \Delta = \frac{n}{p} = \log n \implies t_I = c_3 \log n$$

Riassumendo: $\langle x, y \rangle$ **costa**: $p(n) \sim \frac{n}{\log n}$, $t = t_I + t_{II} \sim \log n$, quindi:

$$E \sim \frac{2n - 1}{\frac{n}{\log n} \cdot \log n} \rightarrow c \neq 0$$

Ci sono costanti non riportate, viene un numero minore di 1 ovviamente.

Prodotto matrice vettore:

- Input: una matrice $A \in \mathbb{N}^{n \times n}$ ed un vettore $x \in \mathbb{N}^n$
- Output: il prodotto tra i due Ax

Ognuna delle n **righe** della matrice A viene **moltiplicata per il vettore**, quindi sono necessarie $n(2n - 1) = 2n^2 - n$ operazioni (ognuna delle n righe della matrice è un vettore moltiplicato per un altro, quindi n volte il prodotto vettoriale).

Sapendo già come fare il prodotto interno, si può usare il modulo di quest'ultimo in parallelo n -volte per risolvere il prodotto matrice-vettore.

Il vettore x è acceduto simultaneamente dai moduli per il prodotto interno \implies algoritmo CREW.

Le **prestazioni** che ne risultano:

$$p(n) = n \cdot \frac{n}{\log n} \quad T(n, p(n)) \sim \log n$$

$$E(n, p(n)) \sim \frac{n^2}{\frac{n^2}{\log n} \cdot \log n} \rightarrow c \neq 0$$

Prodotto matrice matrice:

- Input: due matrici $A, B \in \mathbb{N}^{n \times n}$
- Output: il prodotto $A \times B$

Si può risolvere usando n^2 **prodotti interni in parallelo**.

Tempo sequenziale: $n^{2.8}$ (Strassen), oppure $n^{2.37}$ (Le Gall 2014)

Ogni riga di A e colonna di B vengono accedute simultaneamente \implies algoritmo CREW. La lettura concorrente si potrebbe eliminare usando un broadcast dei dati.

Le **prestazioni** che ne risultano:

$$p(n) \sim n^2 \cdot \frac{n}{\log n} \quad T(n, p(n)) \sim \log n$$

$$E(n, p(n)) \sim \frac{n^{2.8}}{\frac{n^3}{\log n}} = \frac{n^{2.8}}{n^3} \rightarrow 0 \quad (\text{lentamente})$$

Anche se lentamente, tende a zero, quindi in questo modo non è considerato efficiente.

Potenza di matrice:

- Input: una matrice $A \in \mathbb{N}^{n \times n}$
- Output: A^n con $n = 2^k$

Si tratta di un **prodotto iterato** della **stessa matrice**, quindi posso moltiplicare $A \times A$, poi $A^2 \times A^2$, ecc.; così richiedendo un numero logaritmico di passaggi.

Sequenzialmente:

```
for i = 1 to log n do  
    A = A × A
```

Tempo sequenziale: $n^{2.8} \cdot \log n$.

Per il parallelo: sfrutta la stessa idea del sequenziale, quindi bisogna effettuare $\log n$ volte il prodotto $A \times A$, sostituendo il prodotto sequenziale con quello parallelo. Ovviamente algoritmo CREW.

Le **prestazioni** che ne risultano:

$$p(n) = \frac{n^3}{\log n} \quad T(n, p(n)) = \log n \log n = \log^2 n$$

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

Tende a zero, anche se lentamente, quindi non è efficiente, secondo questo parametro.

1.5 Problema delle somme prefisse

Definizione del problema:

- **Input:** n celle $M[1], \dots, M[n]$
- **Output:** $\sum_{i=1}^k M[i] \rightarrow M[k]$, per ogni $1 \leq k \leq n$

Ogni cella deve contenere i valori di tutte le celle precedenti

Per semplicità verrà assunto n potenza di 2.

Algoritmo sequenziale:

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

Quindi tempo $n - 1$.

Prima proposta parallela: risolvo con un **modulo sommatoria** tutti i **possibili prefissi**.

Problemi:

1. **Non EREW** in quanto ogni cella è acceduta da tutti i moduli sommatoria, ma risolvibile.
2. Un algoritmo CREW su PRAM con:

$$p(n) \leq (n - 1) \cdot \frac{n}{\log n} \sim \frac{n^2}{\log n}$$

$n - 1$ moduli sommatoria, ognuno composto di $n / \log n$ processori Di conseguenza

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

$$E \sim \frac{n - 1}{\frac{n^2}{\log n} \cdot \log n} \rightarrow 0$$

L'efficienza tende a 0, quindi non eccezionale.

1.5.1 Pointer Doubling

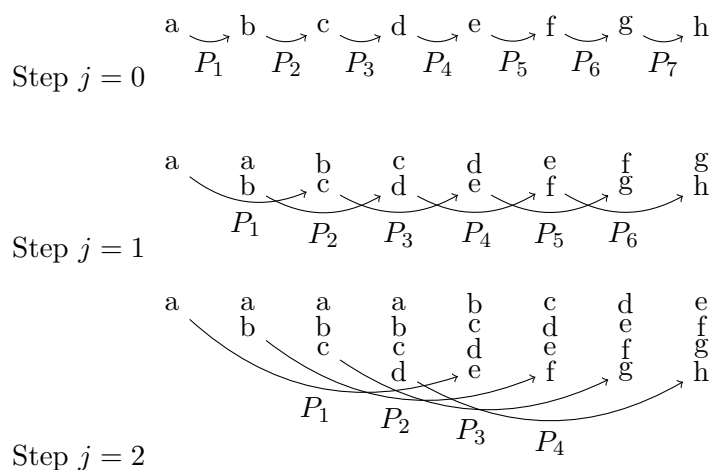
Nel '73 Kogge-Stone ha fornito un algoritmo chiamato pointer doubling.

L'idea: stabilire dei **legami** tra i numeri ed **ogni processore** si occupa di **un legame** e ne fa la somma.

Se i valori m e k sono nelle celle i e j e queste hanno un legame allora il processore i farà la **somma** $m+k$ e la **metterà in posizione** j (la più alta).

Come si continua? Prima si “**legano**” i numeri a distanza 1, poi a distanza 2, poi 4, a **potenze di 2**, fino al termine. Al passo j si legano i valori a distanza 2^j .

Esempio di passaggi:



Facendo tutte le somme in quest'ordine, ogni cella **conterrà la somma delle precedenti**, l'algoritmo **termina** una volta arrivati al punto in cui **nessun elemento ha un successore** in range, ovvero il “salto” di celle porta tutti gli elementi fuori range.

Quanti passi sono necessari? Si completa quando nessun elemento ha un successore, considerando che al j -esimo passo sono presenti 2^j elementi senza successori, l'ultimo passo dell'algoritmo è quando $2^j = n \implies j = \log n$, **servono** $\log n$ **passi**.

Ad ogni passaggio vengono **usati** $n - 2^j$ **processori**, quindi vengono usati tutti i processori con indice $1 \leq k \leq n - 2^{j-1}$.

Poniamo nel vettore $S[k]$ la posizione del **successore** di $M[k]$ (la cella legata), come viene **inizializzato** prima dell'esecuzione dell'algoritmo vero e proprio?

$$S[k] = k + 1 \text{ e } S[n] = 0$$

Con n indice dell'ultima cella (dimensione del problema).

Dato il processore P_k , quale istruzione su M deve eseguire?

$$M[S[k]] \leftarrow M[k] + M[S[k]]$$

Poi bisogna aggiornare S :

$$S[k] \leftarrow (S[k] == 0 ? 0 : S[S[k]])$$

Se $S[k] = 0$ rimane 0, altrimenti diventa $S[S[k]]$.

Codice dell'algoritmo parallelo (con M ed S già inizializzati):

```

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

```

All'inizio l'unico elemento che non ha successore, ovvero con $S[k] = 0$ è l'ultimo, ad ogni passo parallelo se ne aggiungono 2^{j-1} (si arriva a 2^j).

Questo algoritmo è EREW? La scrittura è esclusiva ma ogni processore deve **leggere** anche da una **cella che verrà usata anche dai processori adiacenti**, però **ognuno** di questi lo farà in **tempi diversi**: k accederà prima alla cella $M[k]$ poi a $M[S[k]]$, mentre $k + 1$ accederà ad $M[S[k]]$ e poi ad $M[S[S[k]]]$. Questi sono passi paralleli quindi **la stessa cella non verrà usata nello stesso momento**.
L'algoritmo è **EREW**.

Correttezza:

1. è **EREW PRAM**: P_k lavora su $M[k]$ e $M[S[k]]$: se $i \neq j \implies S[i] \neq S[j]$, quindi hanno successori diversi (se non sono 0)
2. **dimostro** $M[k] = \sum_{i=1}^k M[i]$, $1 \leq k \leq n$. Si lavora sulla proprietà del j -esimo passo:

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

C'è sempre la somma dei 2^j elementi precedenti, sia che si arrivi a 1 che a $t - 2^j + 1$.

Se questa proprietà è vera si ha per $j = \log n$ (ultimo passo):

$$M[k] = \begin{cases} M[k] + \dots + M[1] & k \leq 2^j = 2^{\log n} = n \\ \dots & k > 2^j = n \end{cases}$$

Si dimostra per induzione su j :

- **Base:** $j = 1$, per $k \leq 2$

$$\begin{array}{ll} k = 1 & M[1] \leftarrow M[1] \\ k = 2 & M[2] \leftarrow M[1] + M[2] \end{array}$$

per $k > 2$:

$$M[t + 1] \leftarrow M[t] + M[t + 1] = M[k - 1] + M[k] \rightarrow M[k]$$

Quindi il caso base è vero.

- **Passo induttivo:** si presuppone vera la proprietà per il passo $j - 1$ e bisogna dimostrare che vale per j .

Prima di iniziare il j -esimo passo quanto vale S ?

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

I legami al j -esimo passo legano numeri a distanza 2^{j-1} .

Le celle con indice $\leq 2^{j-1}$ sono già a posto, in quanto la proprietà è vera per $j - 1$.

Le celle con indice compreso tra 2^{j-1} e 2^j , hanno indice t indicato come:

$$2^{j-1} \leq t \leq 2^j \implies t = 2^{j-1} + a$$

Quindi sappiamo che

$$M[a + 2^{j-1}] \leftarrow M[a] + M[a + 2^{j-1}]$$

Ma a è per forza $\leq 2^{j-1}$ e t invece è $t > 2^{j-1}$, quindi, per ipotesi induttiva, i valori di $M[a]$ e $M[a + 2^{j-1}]$ corrispondono a:

$$M[1] + \dots + M[a] + M[a + 1] + \dots + M[a + 2^{j-1}]$$

Invece, per le celle con indice $t > 2^j$, quindi $t = 2^j + a$. Gli elementi sommati in $M[a + 2^j]$ sono:

$$M[a + 2^j] \leftarrow M[a + 2^{j-1}] + M[a + 2^j]$$

Ma, considerando che l'indice è $> 2^{j-1}$, per ipotesi induttiva, corrispondono a:

$$M[a + 1] + \dots + M[a + 2^{j-1}] + M[a + 2^{j-1} + 1] + \dots + M[a + 2^j]$$

E di conseguenza la proprietà continua a essere vera.

Valutazione dell'algoritmo: Il numero di processori massimo è 1 in meno rispetto alla dimensione dell'input, usati nel primo passo:

$$p(n) = n - 1$$

Per quanto riguarda il **tempo**, si ha un ciclo **for** con $\log n$ passi, con all'interno un **par do** che fa eseguire in parallelo ai processori 2 istruzioni, somma e aggiornamento del vettore S .

La prima istruzione

$$M[S[k]] = M[k] + M[S[k]]$$

è composta da 5 microistruzioni

```

LOAD    M[k]
LOAD    S[k]
LOAD    M[S[k]]
ADD
STORE   M[S[k]]

```

Mentre l'istruzione di aggiornamento

$$S[k] = (S[k] == 0 ? 0 : S[S[k]])$$

è composta da 4 microistruzioni

```

LOAD    S[k]
JZERO
LOAD    S[S[k]]
STORE   S[k]

```

Quindi in totale sono 9 operazioni svolte $\log n$ volte, ottenendo un **tempo**:

$$T(n, n - 1) \sim 9 \log n$$

Di conseguenza l'**efficienza** è:

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

Sfruttando **Wyllie**, come per la sommatoria, in modo da far **tendere l'efficienza** a **una costante** (far sparire la funzione $\log n$ da E).

Vogliamo migliorare l'uso dei processori, raggruppiamo di $\log n$ in $\log n$ i valori da sommare, con un processore che esegue somme in sequenza per ognuno. Di conseguenza:

$$p(n) = o\left(\frac{n}{\log n}\right), \quad T(n, p(n)) = O(\log n) \quad E \rightarrow C \neq 0$$

Il primo passo è sequenziale su $\log n$ numeri, ogni processore effettua la somma sequenziale di $\log n$ numeri, di conseguenza il numero di processori passa da lineare a $n/\log n$. Il tempo rimane logaritmico per la fase parallela, ma si aggiunge tempo logaritmico per la fase sequenziale $\sim O(\log n)$. Nel calcolo dell'efficienza $\log n$ adesso si semplifica e rimane che tende a una costante $\neq 0$.

Come per la sommatoria, anche l'**algoritmo** per le somme prefisse può essere **usato per un problema generale** “op-prefissa”, con operazione associativa:

- Input: $M[1], \dots, M[n]$, n celle
- Output: $M[k] = OP_{i=1}^k M[i]$, $1 \leq k \leq n$, in ogni cella ci sia il risultato dell'operazione sulle k celle precedenti

L'operazione deve essere associativa come ad esempio: $+$, $*$, \wedge , \vee , \min , \max ,

1.6 Valutazione di polinomi

Si tratta di un problema che prende in input un polinomio $p(x)$ di grado n e un valore α , restituendo il polinomio valutato sul valore α , quindi $p(\alpha)$.

Definizione:

- **Input:** $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$, α
- **Output:** $p(\alpha)$

Dati in memoria:

- il **valore** α
- $a_0, a_1, \dots, a_n \rightarrow A[0], A[1], \dots, A[n]$, n celle nella memoria condivisa, da $A[0]$ ad $A[n]$, per **contenere i coefficienti**

Algoritmo sequenziale tradizionale:

- prodotti: $\sum_{i=0}^n i \sim n^2$
- somme: n

Di conseguenza, in totale $\sim n^2$.

Ma si può fare di meglio, **miglioramento di Ruffini-Horner**: l'idea è raccogliere x in maniera iterata:

$$\begin{aligned} p(n) &= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 \\ &= a_0 + x(a_1 + a_2x + a_3x^2 + a_4x^3) \\ &= a_0 + x(a_1 + x(a_2 + a_3x + a_4x^2)) \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + a_4x))) \end{aligned}$$

Generalizzando:

$$p(x) = a_0x(a_1 + \dots a_{n-2} + x(a_{n-1} + a_nx) \dots)$$

Questa forma del polinomio suggerisce un algoritmo, **sostituendo** α a **partire** dall'**ultimo coefficiente** del polinomio, partendo dalla parentesi più interna (e chiamo questo valore ottenuto p). Quindi, sostituisco α , sommo un coefficiente e ripeto fino ad arrivare alla fine

$$p = a_n \cdot \alpha + a_{n-1} \rightarrow p = a_{n-2} + p \cdot \alpha \rightarrow \dots$$

Quindi:

$$p = a_j + p \cdot \alpha$$

Codice per l'algoritmo sequenziale Ruffini-Horner:

```
Input ( $\alpha$ )  
p =  $a_n$   
for  $i = 1$  to  $n$   
    p =  $a_{n-i} + p \cdot \alpha$   
Output (p)
```

Prestazioni sequenziali: 2 operazioni per il numero di iterazioni

$$T(n, 1) = 2n$$

Lineare, meglio che n^2 di prima.

Possibile **algoritmo parallelo**:

- **Costruisco** il **vettore delle potenze di α** : Q

$$Q[k] = \alpha^k \quad 0 \leq k \leq n$$

Memorizzo le potenze di α nel vettore Q .

- Eseguo il **prodotto interno** $\langle A, Q \rangle$

$$\langle A, Q \rangle = \sum_{k=0}^n A[k] \cdot Q[k]$$

dove A è il vettore dei coefficienti. Questo effettivamente è valutare il polinomio su α .

- **Restituisco** $\langle A, Q \rangle$.

Il prodotto interno parallelo lo abbiamo già visto ed è efficiente, rimane da capire se è parallelizzabile efficientemente la creazione del vettore Q .

Per creare Q :

- Pongo α in tutti gli elementi di Q da 1 a n

$$Q[1] = \alpha, Q[2] = \alpha, \dots, Q[n] = \alpha$$

Non si considera la cella $Q[0]$, che deve contenere 1. Questo è un problema di replica.

- Applico il **prodotto-prefisso** su Q :

$$Q[1] = \alpha, Q[2] = \alpha^2, \dots, Q[n] = \alpha^n$$

Come risolvere replica in parallelo: prima idea, algoritmo CREW, n processori copiano nelle n celle il valore α

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

Prestazioni:

$$p = n, \quad t = 2, \quad E \sim \frac{n}{n \cdot 2} \rightarrow c \neq 0$$

Seconda idea, per **abbassare il numero di processori** usati dalla replica: **Wyllie**, raggruppo gli n processori in $\log n$ elementi, quindi il processore k si occuperà di caricare α nelle celle di posizione tra $(k-1) \log n + 1$ e $k \log n$.

Codice:

```
for k = 1 to  $n/\log n$  par do
  for i = 1 to  $\log n$  do
    Q[(k-1)log n + i] =  $\alpha$ 
```

Il numero di processori è di $n/\log n$, di conseguenza le **prestazioni**:

$$p = \frac{n}{\log n}, \quad t = c \log n, \quad E = \frac{n}{\frac{n}{\log n} \cdot c \log n} = \frac{1}{c} \neq 0$$

Comunque si tratta di un algoritmo CREW.

Problema: tutti leggono α contemporaneamente, rendendo l'algoritmo CREW, quindi, terza idea, **per renderlo EREW-PRAM**:

- **Costruisco il vettore** $\alpha, 0, 0, \dots, 0$
- Eseguo **somme-prefisse**

Codice per **ottenere il vettore** $\alpha, 0, 0, \dots, 0$

```

Input  ( $\alpha$ )
Q[1]  =  $\alpha$ 
for  $k = 2$  to  $n$  par do
    Q[k] = 0

```

Lo 0 è una costante e non deve essere letta.

Riduzione dei processori: richiederà numero di processori $p = n/\log n$ e $t = \log n$ ad entrambi gli step, quindi in totale

$$p = \frac{n}{\log n}, \quad t = \log n, \quad E = c \neq 0$$

Riassunto: valutazione polinomio con EREW-PRAM:

- Costruisco $Q[k] = \alpha$ con replica

$$p = \frac{n}{\log n}, \quad t = \log n$$

- Costruisco $Q[k] = \alpha^k$ con prodotto prefisso

$$p = \frac{n}{\log n}, \quad t = \log n$$

- Calcolo il prodotto interno $\langle A, Q \rangle$

$$p = \frac{n}{\log n}, \quad t = \log n$$

In totale:

$$p = \frac{n}{\log n}, \quad t = \log n, \quad E = \frac{T(n, 1)}{p(n)T(n, p(n))} \sim \frac{2n}{\frac{n}{\log n} \log n} \rightarrow c \neq 0$$

1.7 Ricerca di un elemento

Trovare se il valore α è presente tra le n celle, l'ultima cella assumerà valore 1 se esiste il valore cercato all'interno della memoria considerata.

Definizione:

- **Input:** $M[1], M[2], \dots, M[n], \alpha$
- **Output:** $M[n] = 1$ se $\exists k$ t.c. $M[k] = \alpha$, altrimenti 0

Algoritmo sequenziale classico: $t = n$ controllo tutte le celle; se il vettore è ordinato (ordinamento costo $O(n \log n)$) posso fare ricerca dicotomica ($t = \log n$).

Algoritmo quantistico (recente, del 1996) su input non ordinato (interferenza quantistica): $t = \sqrt{n}$.

Algoritmi **paralleli per ricerca** di α : prima idea, CRCW, si usa un flag F

```
F = 0
for k = 1 to n par do
    if (M[k] == α)
        F = 1
M[n] = F
```

Se uno degli n processori trova nella sua cella dedicata il valore corretto il flag viene messo ad uno, il valore del flag viene messo in $M[n]$ alla fine. Abbiamo una lettura concorrente di α ed una scrittura concorrente di F per tutti i processori che trovano α nello stesso momento.

Prestazioni:

$$p(n) = n \quad t = c$$

Perché usiamo F ? Per non perdere il valore di $M[n]$?

Seconda idea: algoritmo CREW

```
for  $k = 1$  to  $n$  par do  
     $M[k] = (M[k] == \alpha ? 1 : 0)$   
  
Max-Iterato( $M[1], \dots, M[n]$ )
```

Il risultato viene scritto in ogni cella, e poi bisogna mettere 1 in $M[n]$ se c'è almeno un 1 all'interno delle celle. Per l'ultimo passaggio si può fare un Max-Iterato su tutte le celle (stesso modulo della somma-iterata).

Prestazioni:

- Per la prima parte: $p(n) = n$, $t = c$, ma applicando Wyllie:

$$\implies p(n) = \frac{n}{\log n} \quad t = \log n$$

- Per il Max-Iterato

$$p(n) = \frac{n}{\log n} \quad t = \log n$$

Risultato:

$$p(n) = O(n/\log n) \quad t = O(\log n)$$

L'efficienza:

$$E \rightarrow c \neq 0$$

Terza idea: algoritmo EREW

1. Usa modulo replica per α

$$\alpha \rightarrow A[1], \dots, A[n]$$

2. Confronto con tutte le celle:

```
for k = 1 to n par do
    M[k] = (M[k] == A[k] ? 1 : 0)
```

3. Max-iterato($M[1], \dots, M[n]$) per spostare il valore 1 nell'ultima cella

Prestazioni: Step 2 e 3 vengono risolti (come nel precedente) con $p(n) = n/\log n$ e $t = \log n$. Mentre il passo 1 $p(n) = n/\log n$ e $t = \log n$.

Totale:

$$p = \frac{n}{\log n}, \quad t = \log n, \quad E = c \neq 0$$

Quindi abbiamo un algoritmo parallelo EREW efficiente.

Varianti di questo codice per problemi legati:

- **conteggio di α in M :** il Max-Iterato diventa una sommatoria, conta quante celle hanno trovato il valore
- **posizione massima di α in M :** al posto di scrivere 1 se trovo il valore scrivo k , dove questo è l'indice della cella
- **posizione minima di α in M :** modifico come sopra scrivendo k al posto che 1 ma l'ultimo passaggio diventa una OP-Iterata, dove

$$OP(x, y) = \begin{cases} \min(x, y) & \text{if } x \neq 0 \text{ and } y \neq 0 \\ \max(x, y) & \text{otherwise} \end{cases}$$

1.8 Problema dell'ordinamento

Formalmente chiamato **ranking**. Definizione:

- **Input:** $M[1], \dots, M[n]$
- **Output:** permutazione $p : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ tale che

$$M[p(1)] \leq M[p(2)] \leq \dots \leq M[p(n)]$$

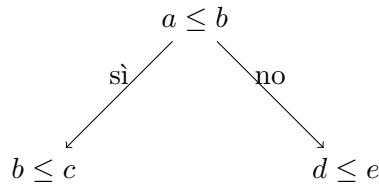
dove $p(i)$ indica l'indice dell'elemento del vettore M in che va in posizione i

In genere gli algoritmi di ordinamento sono basati sui confronti, guardano chi è il più piccolo.

Gli algoritmi di ordinamento basati sul confronto hanno

$$t = \Theta(n \log n)$$

Dimostrazione (circa) del **lower bound**: albero di decisione = algoritmo di ordinamento. Costruisco un albero in cui ad ogni nodo avviene un confronto. Esempio di nodo:



Le foglie sono una permutazione che permette di ordinare l'input. L'altezza dell'albero rappresenta il numero di confronti effettuati nel caso peggiore, di conseguenza il tempo dell'algoritmo di ordinamento.

Osservazione: il numero di foglie deve essere $\geq n!$ in quanto questi sono i possibili ordinamenti di n elementi.

Per un albero di altezza t il numero massimo di foglie è 2^t

$$2^t \geq \text{foglie} \geq n! \implies t \geq \log n!$$

$$\log n! \geq \log \prod_{i=n/2+1}^n i \geq \log \left(\frac{n}{2}\right)^{n/2} = \frac{n}{2} \log \frac{n}{2} \sim n \log n$$

1.8.1 CountingSort

Primo approccio parallelo (basato su algoritmo di conteggio $t = \Theta(n^2)$).

Assunzione: n è una potenza di 2 e gli elementi sono \neq tra loro.

Sequenziale CountingSort: $M[i]$ va in posizione $k \Leftrightarrow k$ elementi sono $\leq M[i]$ in M .

Usiamo il vettore

$$V[1], V[2], \dots, V[n] \text{ dove } V[i] = k$$

Algoritmo sequenziale counting:

```
for i = 1 to n
    V[i] = 0

for i = 1 to n
    for j = 1 to n
        if (M[j] < M[i])
            V[i]++

for i = 1 to n
    F[V[i]] = M[i]
for i = 1 to n
    M[i] = F[i]
```

Conto quante celle hanno valore minore di $M[i]$ ed ottengo la posizione del valore nel vettore finale.

Gli ultimi due **for** servono a riscrivere il vettore in modo ordinato.

Prestazioni: la fase pesante è quella dei 2 **for** innestati, effettua n^2 confronti, quindi $T(n, 1) = n^2$.

Versione parallela: passaggi

1. $\forall j, i$ ho un **processore** $P_{i,j}$ che effettua

$$M[j] \leq M[i]?$$

Poniamo il **risultato** del confronto in una **matrice booleana**

$$V[i, j] = (M[j] \leq M[i] ? 1 : 0)$$

Quindi la i -esima riga individua gli elementi di M che sono $\leq M[i]$.

2. $\forall i$ effettuo la **sommatoria parallela** della i -esima riga (sommo i valori interni a tutta la riga, ogni riga), ottenendo

$$\begin{array}{ccc} & V[1, n] & V[1] \\ & V[2, n] & V[2] \\ \Rightarrow & \vdots & \vdots \\ & V[n, n] & V[n] \end{array} \xrightarrow[\text{(di prima)}]{\text{coincide con}}$$

In questo modo, alla fine della somma, trovo quanti valori sono inferiori di i .

```
for 1 ≤ i, j ≤ n par do
    V[i, j] = (M[j] ≤ M[i] ? 1 : 0)
for i = 1 to n par do
    Sommatoria(V[i, 1], V[i, 2], ..., V[i, n])
for i = 1 to n pardo
    M[V[i, n]] = M[i]
```

Per **ogni coppia** i, j c'è un **processore** che lavora in parallelo per eseguire il confronto, il **risultato** viene **memorizzato** nella cella $V[i, j]$ di una **matrice booleana**. Vuol dire che ci sono n^2 **processori**.

Poi viene effettuata la **somma parallela** dei valori presenti in **ogni riga**. Infine si va a **scrivere** nel vettore M l'**array ordinato**. Ogni valore $V[i, n]$ va ad indicare quanti elementi sono minori di i .

Prestazioni: si tratta di un algoritmo CREW (accesso concorrente nella prima fase)

- Prima fase:

$$p(n) = n^2, \quad T(n, n^2) = 4$$

Per il tempo le fasi sono LD, LD, JZ, ST. Usando Wyllie si può ottenere

$$p(n) = n^2 / \log n, \quad t \sim \log n$$

- Seconda fase: n moduli sommatoria, quindi

$$p(n) = n^2 / \log n, \quad t \sim \log n$$

- Terza fase

$$p(n) = n \quad t = 3$$

In totale

$$p \sim \frac{n^2}{\log n}, \quad t \sim \log n$$

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

Quindi non è efficiente.

Algoritmi di ordinamento paralleli:

- **CountingSort:**

$$E = \frac{\log n}{n} \rightarrow 0$$

- **BitSort:**

$$E = \frac{1}{\log n} \rightarrow 0$$

ma ci tende lentamente

- **Cole (1988)**

$$E \rightarrow c \neq 0$$

ma è complicato

1.8.2 BitSort

Algoritmo sequenziale MergeSort: dai che sai come funziona, non devo scrivertelo. Tempo: $T(n, 1) = n \log n$.

Prendendo ispirazione dal MergeSort, effettuarlo in parallelo vorrebbe dire effettuare $\log n$ passi paralleli.

Ma purtroppo **NON è parallelizzabile** ed ottengo ancora $t \sim n \log n$.

Quando il passaggio di merge dei valori diventa facile? Quando i **due vettori** sono **ordinati** ed i **valori del primo** sono **tutti minori** dei **valori del secondo**, in questo caso per effettuare il merge è sufficiente concatenare i due vettori.

Con l'uso di sequenze di numeri bitoniche possiamo garantire che questa proprietà sia rispettata.

Operazioni elementari su sequenze:

- **Reverse:** inverte il vettore

$$REV(A[1], A[2], \dots, A[n])$$

$$A[1] \leftarrow A[n], \quad A[2] \leftarrow A[n-1], \quad \dots \quad A[n] \leftarrow A[1]$$

- **MinMax:** divide il vettore in due, prende l'elemento k e l'elemento $k + n/2$, rispettivamente per la prima e seconda metà. Nel primo dei due scrive il minimo dei due, nel secondo scrive il massimo. Ripeto l'operazione per tutti gli elementi delle metà. In questo modo nella prima metà avrò tutti i minimi, nella seconda tutti i massimi.

$$A[1] \dots A[k], \dots A[n/2] \dots A[k + n/2] \dots A[n]$$

$$A[k] \leftarrow \min\{A[k], A[k + n/2]\}, \quad A[k + n/2] \leftarrow \max\{A[k], A[k + n/2]\}, \quad \dots$$

Algoritmi paralleli per queste operazioni:

- **Procedura** $Rev(A)$: da 1 a $n/2$, scambio i valori di ogni cella con la sua simmetrica

```
for  $1 \leq k \leq n/2$  par do  
    Swap ( $A[k]$ ,  $A[n - k + 1]$ )
```

Prestazioni: lavora su metà dei valori e deve fare LD, LD, ST, ST

$$p(n) = \frac{n}{2}, \quad t = 4$$

- **Procedura** $MinMax(A)$:

```
for  $1 \leq k \leq n/2$  par do  
    if ( $A[k] > A[k + n/2]$ )  
        Swap ( $A[k]$ ,  $A[k + n/2]$ )
```

Prestazioni: richiede solamente un'operazione in più di prima (il confronto)

$$p(n) = \frac{n}{2}, \quad t = 5$$

Particolari sequenze numeriche: Definizioni formali:

- **Unimodale**: A è unimodale iff $\exists k$ tale che

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

oppure

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

- **Bitonica**: A è bitonica iff \exists una permutazione ciclica di A tale che dia una sequenza unimodale: $\exists j$ tale che

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

è unimodale

Sostanzialmente, una sequenza unimodale ha un picco (massimo o minimo), mentre una bitonica ha due picchi, un minimo ed un massimo (scende, sale e poi scende di nuovo o viceversa) e se la “giro” può diventare un picco solo.

Esempi:

2 4 7 9 5 3

è una serie unimodale, con 9 come picco.

7 9 5 3 2 4

è una serie bitonica, con 9 e 2 come picchi (sale, poi scende, poi risale) e posso “gitarla” fino a farla diventare quella di prima.

Osservazioni:

- **Unimodale** \implies **bitonica**, grazie alla permutazione identità.
- In una serie **bitonica**, gli **elementi di fine array** devono essere **maggiore** di quelli di **inizio array** (o **viceversa**, dipende dal caso).
- Siano A, B due sequenze ordinate crescenti (decrescenti), la **sequenza** $A \cdot REV(B)$ è **unimodale**.

Proposizione su sequenze bitoniche: Sia A bitonica, eseguo $MinMax(A)$, ottengo:

- Due sequenze A_{min} e A_{max} , le quali sono **bitoniche**.
- Ogni elemento di A_{min} è **minore** di ogni elemento di A_{max} .

Quindi si può pensare ad un approccio *divide et impera* per le **serie bitoniche**:

- $MinMax$ **suddivide il problema** di dimensione n in istanze più piccole (che rimarranno bitoniche)
- Il **merge** di A_{max} e A_{min} avviene **per concatenazione**

Quindi si minmaxa ricorsivamente fino ad arrivare a coppie di elementi, che $MinMax$ è in grado di ordinare, e la fusione avviene per concatenazione.

BitMerge sequenziale:

```
MinMax(A)
  if (|A| > 2)
    BitMerge(Amin)
    BitMerge(Amax)
  return (A)
```

Funziona **solo** con sequenze **bitoniche**.

Correttezza di BitMerge: si procede per **induzione**:

- **Base:** $n = 2$, una sequenza di lunghezza 2 è banalmente ordinata da *MinMax*.
- **Passo induttivo:** supponiamo corretto per $n = 2^k$, dimostriamo che è valido per $2^{k+1} = |A|$
 - *MinMax* restituisce A_{min} e A_{max} di lunghezza 2^k
 - *BitMerge*(A_{min}) e *BitMerge*(A_{max}) ordinano A_{min} e A_{max} per ipotesi induttiva
 - Alla fine torna A ordinato

Implementazione parallela di BitMerge: Ad ogni divisione viene chiamato *MinMax*, fino ad arrivare a coppie, le quali saranno ordinate, alla fine basta concatenare tutte le coppie.

Tutti i **moduli MinMax** sullo **stesso livello** vengono eseguiti **in parallelo**.

Valutazione: del BitMerge parallelo

- Si tratta di un codice **EREW-PRAM**, tutte le chiamate di *MinMax* sullo stesso livello vengono effettuate su elementi differenti di *A*
- **Tempo:** dividendo sempre a metà il vettore, per arrivare a coppie di elementi devo dividere $\log n$ volte, e servono 5 operazioni per il *MinMax*:

$$t = 5 \log n$$

- **Processori:** Lavorando sempre su tutti i valori ad ogni iterazione, ed ogni processore confronta 2 valori, servono processori pari alla metà dei valori in input:

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

Sono i processori richiesti da *MinMax*, su lunghezza n ne chiede $n/2$, 2 chiamate su $n/2$ ne richiedono $2 \cdot n/4$, quindi sono sempre $n/2$. Ad ogni passo è sempre $n/2$

- **Tempo** visto con un **equazione di ricorrenza:**

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

$$T(n) = 5 \log n$$

- **Efficienza:**

$$E = \frac{n \cdot \log n}{\frac{n}{2} \cdot 5 \log n} \rightarrow c \neq 0$$

Da BitMerge a BitSort: Si può sfruttare BitMerge come modulo per ordinare sequenze generiche, con l'algoritmo BitSort.

BitSort sequenziale:

```

MinMax(A)
if (|A| > 2)
    BitSort(Amin)
    BitSort(Amax)
    BitMerge(Amin · REV(Amax))
return (A)

```

Aggiunge un passaggio, BitMerge su A_{min} concatenato a $REV(A_{max})$, ovvero due sequenze unimodali che vengono concatenate in una bitonica. Sostanzialmente serve ad unire in modo corretto le due stringhe, in quanto non ho più la garanzia che tutti gli elementi della prima parte siano minori di tutti gli elementi della seconda.

BitMerge lavora solo su sequenze bitoniche, quindi le chiamate ricorsive procedono fino a lunghezza 2 che vengono ordinate da *MinMax*, il BitMerge ordina le coppie alla fine della ricorsione, creando dal basso delle sequenze bitoniche.

Correttezza di BitSort:

- **Base:** $n = 2$, una sequenza di lunghezza 2 è banalmente ordinata da *MinMax*.
- **Passo induttivo:** supponiamo corretto per $n = 2^k$, dimostriamo che è valido per $2^{k+1} = |A|$
 - *MinMax* restituisce A_{min} e A_{max} di lunghezza 2^k
 - *BitSort*(A_{min}) e *BitSort*(A_{max}) ordinano i rispettivi vettori per ipotesi induttiva
 - *BitMerge*($A_{min} \cdot REV(A_{max})$) ordina il vettore in quanto sequenza bitonica

Implementazione parallela di BitSort: Si procede fino a coppie ordinate, ma rispetto a prima si ha una seconda fase in più nella quale si effettuano tutti i BitMerge (ed i relativi *REV*).

Valutazione:

- Si tratta, come prima di un algoritmo **EREW-PRAM**.
- **Tempo:** per la prima fase, come il BitMerge, $t = O(\log n)$. Per la seconda fase si esegue *REV* (t costante) e BitMerge, che richiede $O(\log n)$, quindi in totale

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

- **Processori:** per tutte le fasi ne servono al massimo $n/2$, quindi:

$$p(n) = n/2$$

- **Tempo** visto con un **equazione di ricorrenza:**

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

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

- **Efficienza:**

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

Quindi tende a zero, ma **lentamente**.

Osservazioni:

- Un buon algoritmo sequenziale, non necessariamente può essere trasformato in un buon algoritmo parallelo (es. MergeSort).
- Ma vale anche il contrario, da un buon algoritmo parallelo non sempre si può passare ad un buon algoritmo sequenziale (es. BitSort).

Valutazione sequenziale di BitSort: dobbiamo considerare le chiamate ricorsive per risolvere sequenzialmente il BitSort

- BitMerge:

$$t_{bm}(n) = \begin{cases} O(1) & n = 2 \\ 2t_{bm}\left(\frac{n}{2}\right) + O(n) & n > 2 \end{cases}$$

$O(1)$ se la lunghezza è 2, altrimenti chiamata ricorsiva sulle due metà più il *MinMax*. In totale:

$$t_{bm}(n) = O(n \log n)$$

- BitSort:

$$t_{bs}(n) = \begin{cases} O(1) & n = 2 \\ 2t_{bs}\left(\frac{n}{2}\right) + O(n \log n) & n > 2 \end{cases}$$

Come sopra, ma diventa $O(n \log n)$ per il BitMerge interno. In totale:

$$t_{bs}(n) = O(n \log^2 n)$$

Ovvero peggio del MergeSort sequenziale.

1.9 Tecnica dei Cicli Euleriani

Definizioni base di teoria dei Grafi:

- **Grafo diretto** D : è una coppia (V, E) dove $E \subseteq V^2$, $(v, w) \in E$ indica la presenza di un arco che va da v a w .
- **Cammino**: una sequenza di archi tale che il nodo finale di ogni arco coincida con il nodo iniziale dell'arco successivo.
- **Ciclo**: un cammino tale che il nodo finale di e_k coincide con il nodo iniziale di e_1 .
- **Ciclo Euleriano**: ciclo in cui ogni arco in E compare una e una sola volta.
- **Cammino Euleriano**: un cammino in cui ogni arco viene attraversato una e una sola volta.
- **Grafo Euleriano**: un grafo si definisce tale se contiene un ciclo Euleriano

Notazione: $\forall v \in V$ definiamo

$$\rho^-(v) = |\{(w, v) \in E\}|$$

come il **grado di entrata** di v (numero di archi entranti) e

$$\rho^+(v) = |\{(v, w) \in E\}|$$

come il **grado di uscita** di v (numero di archi uscenti).

Teorema: D è Euleriano se e solo se $\forall v \in V : \rho^-(v) = \rho^+(v)$, tutti i nodi devono avere lo stesso numero di archi entranti ed uscenti.

Giusto per non fare confusione: un ciclo è Hamiltoniano se e solo se è un ciclo nel grafo dove ogni vertice compare una e una sola volta (vertici, non archi).

Calcolare se D è Euleriano è un problema efficiente ($O(n^2)$, con $n = |V|$), mentre capire se D è Hamiltoniano si tratta di un problema \mathcal{NP} -completo.

La **tecnica del Ciclo Euleriano**: viene usata per costruire algoritmi paralleli efficienti che gestiscono strutture dinamiche, come alberi binari.

Considerando un albero binario, $\forall v \in V$ **chiamiamo** $\text{sin}(v)$ il figlio sinistro di v , $\text{des}(v)$ il figlio destro di v e $\text{pad}(v)$ il padre di v .

Molti problemi ben noti usano alberi, es. ricerca, dizionari, query, etc.

Fondamentale in tali problemi la **navigazione dell'albero** e per farlo con algoritmi paralleli efficienti **considereremo delle liste**.

Per definire una lista:

- **Primo passo:** associo all'albero binario un ciclo Euleriano, ogni collegamento diventano due archi, uno entrante ed uno uscente. Seguendo il ciclo navigo l'albero.
- **Secondo passo:** da ciclo a cammino Euleriano: ogni nodo v viene espanso in sinistra, centro e destra, (v, s) , (v, c) , (v, d) . Parto dal nodo $(1, s)$ e sostanzialmente una DFS per visitare tutto l'albero. Il nodo centrale permette di passare da figlio sinistro a destro.

- **Terzo passo:** dal cammino Euleriano costruisco la lista:

$$S((v, x)) \text{ dove } 1 \leq v \leq n, x \in \{s, c, d\}$$

Le regole cambiano nel caso in cui v sia:

- Foglia:

$$S[(v, s)] = (v, c)$$

$$S[(v, c)] = (v, d)$$

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

Dalla destra mi sposto al (padre, centro) se sono il figlio di sinistra, altrimenti mi sposto a (padre, destra).

- Nodo interno:

$$S[(v, s)] = (\text{sin}(v), s)$$

$$S[(v, c)] = (\text{des}(v), s)$$

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

Algoritmo parallelo per costruire S :

- Un processore per ogni $v \in V$, quindi per ogni riga della tabella.
- Se un nodo ha due figli, verrà letta la riga del padre contemporaneamente da entrambi i figli, due nodi accedono alla stessa riga. Si possono evitare letture concorrenti per avere un algoritmo EREW, usando

$$p(n) = n \quad T(n, p(n)) = O(1)$$

Applicando Wyllie

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

N.B: Manca l'esempio, non penso sia particolarmente complicata la cosa, ma nel caso è sulle slide (piuttosto che rifare 14 alberi e frecce su tikz metto parallelamente i testicoli in un tritacarne).

Possiamo usare l'array S per **risolvere problemi**, come:

1. **Attraversamento dell'albero in preordine** (ordine: radice, figlio sinistra, figlio destra), calcolare l'ordine di attraversamento
2. **Calcolare la profondità** dei nodi, per ogni nodo dell'albero

Servono due **definizioni**:

- $\forall v \in V : N(v) = \text{ordine di attraversamento di } v \text{ in preordine}$; es: $N(\text{radice}) = 1$ in quanto primo nodo visitato, mentre per la foglia più a destra possibile il valore sarà n , in quanto ultima visitata.
- $\forall v \in V : P(v) = \text{profondità di } v \text{ nell'albero}$.

1.9.1 Attraversamento in preordine

Definiamo un array A :

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

indicizzato dai nodi (v, x) , nel quale **se** x è l'etichetta di **sinistra** allora il valore sarà 1, 0 altrimenti.

Quindi avrò un valore di 1 solo per tutti i nodi di sinistra. **Seguendo il cammino Euleriano** presente nel vettore S ed effettuando le **somme prefisse** in A secondo l'ordine indicato da S **ottengo l'ordine di attraversamento**, per ogni nodo il risultato verrà posto nella sua componente con etichetta sinistra

$$A[(v, s)] = N(v)$$

Seguo l'ordine di S all'interno di A e faccio somme prefisse, tenendo il conto riesco a tracciare l'ordine dei nodi.

Algoritmo parallelo per l'ordine di $N(v)$: Richiede di fare

1. Calcolo di A e S (successore)
2. Calcolo di Somme-Prefisse su A e S

L'output risulta, $\forall v \in V$, all'interno di $A[(v, s)]$.

Prestazioni:

- è **EREW**.

- **Numero di processori:**

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

- **Tempo:**

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

Per entrambe le fasi.

- **Efficienza:**

$$E = \frac{n}{\frac{n}{\log n} \cdot \log n} \rightarrow c \neq 0$$

Quindi è un **buon algoritmo** parallelo.

1.9.2 Calcolo della profondità dei nodi

Definiamo un array A :

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

Se x è l'etichetta di sinistra il valore è 1, 0 se è centro, -1 se è destra.

Come prima, applichiamo **Somme-Prefisse sull'ordine dato dal ciclo Euleriano**, quindi seguendo l'ordine di S .

Sostanzialmente, quando scendo a sinistra aggiunge 1, quando vado al centro rimane uguale, toglie 1 quando scendo a destra.

Algoritmo parallelo per la profondità $P(v)$: Richiede di fare

1. Calcolo di A e S (successore)
2. Calcolo di Somme-Prefisse su A e S

L'output risulta, $\forall v \in V$, all'interno di $A[(v, s)]$ (se si comincia a contare da 1) oppure $A[(v, d)]$ (se si comincia a contare da 0).

Prestazioni: Come prima

$$E = \frac{n}{\frac{n}{\log n} \cdot \log n} \rightarrow c \neq 0$$

Osservazioni Finali su PRAM

Interesse **Teorico**:

- I processori sono uguali e alla pari.
- Il tempo è strettamente legato alla computazione (comunicazione costante).

Interesse **pratico**:

- Realizzazione fisica dei multicore.

La realizzazione dei multicore ha portato l'interesse del calcolo parallelo da ambiti scientifici precisi a un ambiente più ampio (consumatore generico).

Prima del 2000 non c'era multicore, si puntava ad aumentare il clock, con i relativi problemi riguardanti assorbimento di energia e raffreddamento.

Dopo il 2000 si è aumentato il grado di parallelismo con il multicore, i quali permettono di avere clock minori e di conseguenza migliorare consumi e raffreddamento.

Ciò ha portato a nuovi sviluppi teorici in ambito di algoritmi paralleli, per portare alla creazione di software per i multicore.

2 Architetture parallele a Memoria Distribuita

Possiamo rappresentare l'architettura come un **grafo** G , con un **nodo per ogni processore** e gli **archi** rappresentano la **rete di connessione**.

Non si ha più una **memoria condivisa** e non è nemmeno detto che il processore a cui devo inviare i dati sia collegato al processore che li possiede, l'**informazione** potrebbe dover **passare da più “processori router”** prima di arrivare a destinazione.

Caratteristiche:

- **Processori:** RAM sequenziali, con istruzioni per il calcolo e memoria privata, ma devono anche spedire informazioni (fare da “router”), quindi servono **istruzioni per la comunicazione**, ovvero **send** e **receive**.
Da notare che anche la **comunicazione avviene in parallelo**, quindi se n processori provano a spedire dati a un solo processore, questo dovrà impiegare $n + 1$ passi per ricevere tutto.
- **Collegamenti:** di tipo Full-duplex, gli archi del grafo sono non diretti, non avrebbe senso avere canali di comunicazione a un solo senso. Due processori collegati, per comunicare tra di loro impiegano 2 passi (tempo costante), cosa non vera per 2 processori più lontani.
- **Clock centrale:** scandisce il tempo per tutti i processori.
- **Programma:** come nelle PRAM, ci sono i **par do** e tutti i processori eseguono la stessa istruzione (SIMD, vedi PRAM). Come già detto, si aggiungono le istruzioni di **send** e **receive**.
- **Input/Output:** non abbiamo una memoria condivisa, quindi:
 - Input: distribuito tra i processori
 - Output: o su un processore dedicato, o si legge in un certo ordine tra i vari processori
- **Risorse di calcolo**
 - Numero di processori
 - Tempo, dato da tempo di calcolo + tempo di comunicazione

2.1 Parametri di Rete

Data l'architettura $G = (V, E)$, **definiamo**:

- **Grado** di G :

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

dove $\rho(v)$ è il numero di archi incidenti su v (grado). Quindi è il **grado massimo del grafo**. Un γ alto permette buone comunicazioni ma rende più difficile la realizzazione fisica.

- **Diametro** di G :

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

dove $d(v, w)$ rappresenta la distanza (numero di nodi da attraversare) minima per andare da v a w . Il diametro è la **massima distanza tra 2 nodi**. Valori bassi di δ sono preferibili, ma aumentano il parametro γ .

- **Ampiezza di bisezione** di G : si indica con β e indica il **minimo numero di archi** in G che **tolti dividono i nodi in circa due metà**. Rappresenta la capacità di trasferire le informazioni in G , un valore di β alto è preferibile ma incrementa γ .

2.2 Tipici problemi

Tipici problemi che mettono in **risalto pregi e difetti** di queste architetture:

- **Max:** massimo, richiede una comunicazione a coppie di processori, quindi è determinato da δ .
- **Ordinamento:** si richiede lo spostamento di parti dell'input, quindi è determinato da β .

Fact I: Il tempo richiesto per risolvere **Max** in G è almeno δ .

Dimostrazione: Ogni coppia di processori deve comunicare.

Fact II: il tempo richiesto per risolvere l'ordinamento in G è almeno $n/2 \cdot 1/\beta$.

Dimostrazione: come minimo devo controllare ed eventualmente scambiare ogni coppia di dati, quindi $n/2$, e posso controllarne parallelamente β per volta.

Quindi la **topologia di rete impatta il tempo**

$$T_{Max} = \Omega(\delta)$$

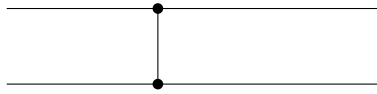
$$T_{Ord} = \Omega\left(\frac{n}{2\beta}\right)$$

2.3 Confrontatori

Per affrontare i problemi **Max** e **Ordinamento** introduciamo i confrontatori e primitive.

Definizione: Istruzioni di confronto per i confrontatori (o comparatori), possono essere:

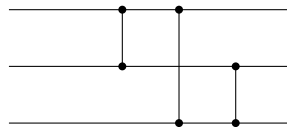
- if ($A[i] > A[j]$) then Swap($A[i]$, $A[j]$)
- if ($A[i] < A[j]$) then Swap($A[i]$, $A[j]$)



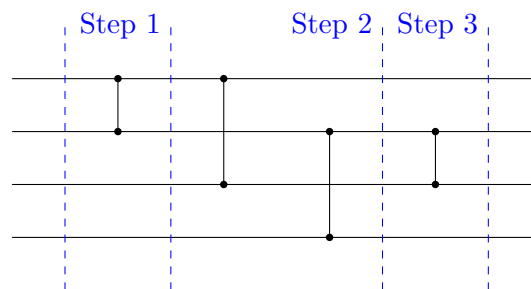
Sostanzialmente, due fili collegati da un confrontatore il quale mette sopra il minimo e sotto il massimo (o viceversa, in base alla forma).

Sono stati definiti per le **sorting network**, reti di confrontatori che permettono di ordinare n elementi.

Esempio a 3 elementi:



Le sorting network danno ispirazione ad **algoritmi di ordinamento**:



Si può assegnare ad ogni linea un processore ed uno step è il massimo insieme contiguo di confrontatori che occupano ogni confrontatore al più una volta sola, senza rompere la sequenza di esecuzione (da formalizzare un po').

Quindi per l'esempio sopra:

$$T(n) = \# \text{ Step} = 3, \quad p(n) = \# \text{ Fili} = 4$$

Formalizziamo una **rete di confrontatori** come:

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

dove (x_1, \dots, x_n) rappresenta gli n input e (y_1, \dots, y_n) gli n output.

Si dice che R è una **sorting network** iff $\forall (x_1, \dots, x_n) \in \mathbb{N}$ vale

$$R(x_1, \dots, x_n) = (y_1, \dots, y_n) \text{ con } y_1 < \dots < y_n$$

ovviamente l'output deve essere una permutazione dell'input.

Dette anche reti di ordinamento "*oblivious*", non dipende dall'input, la rete è ignara di ciò che ci passa sopra.

2.3.1 Valutare una rete

Per capire se **una rete è una sorting network** si può usare il principio 0-1.

Formalmente, è una sorting network se:

- $\forall x \in \{0, 1\}^n$, $R(x)$ è ordinato
 $\implies \forall x \in \mathbb{N}^n$ si ha $R(x)$ è ordinato.

Il che è equivalente a dire che:

- $\exists x \in \mathbb{N}^n$ tale che $R(x)$ non è ordinato
 $\implies \exists x \in \{0, 1\}^n$ tale che $R(x)$ non è ordinato.

Se ordina bene vettori booleani allora ordina bene anche vettori di naturali, quindi posso valutare la rete solo su vettori booleani.

f -shift: Prima di applicare la rete R possiamo applicare una funzione f monotona crescente ed è equivalente ad applicare f dopo R

$$R(f(x_1), \dots, f(x_n)) = \vec{f}(R(x_1, \dots, x_n)) = (f(y_1), \dots, f(y_n))$$

Insomma, una funzione monotona crescente posso applicarla sia prima che dopo.

Ipotizzando di avere una sorting network R **non corretta**:

$$\exists x \in \mathbb{N}^n \text{ t.c. } R(x) = (y_1, \dots, y_k, \dots, y_s, \dots, y_n)$$

con $y_k > y_s$ e $k < s$ (il vettore non è ordinato per un qualche input x).

Definiamo una **funzione** $g : \mathbb{N} \rightarrow \{0, 1\}$:

$$g(x) \begin{cases} 1 & x \geq y_k \\ 0 & \text{altrimenti} \end{cases}$$

g è monotona crescente. y_k sarà mappato in 1 mentre y_s in 0.

Se applico g prima di R ottengo un **vettore binario** a cui **applicare la rete**

$$R(g(x_1), \dots, g(x_n))$$

ma per la regola dello shift diventa uguale a

$$= (g(y_1), \dots, g(y_k), \dots, g(y_s), \dots, g(y_n))$$

Ma abbiamo $g(y_k) = 1$ prima di $g(y_s) = 0$, di conseguenza **non ha ordinato il vettore binario**. Ogni vettore naturale può essere trasformato in questo modo.

Osservazione: per capire se R è una sorting network valuto R solo su input binari.

2.4 Array Lineari

Gli array lineari sono un'architettura **parallela** a **memoria distribuita**, tutti i processori da P_1 a P_n sono collegati “in linea”, **ogni processore è collegato al successivo e al precedente**.

Parametri di rete:

- **Grado** $\gamma = 2$, ottimo per la realizzazione, ogni processore è collegato ad altri 2.
- **Diametro** $\delta = n - 1$, lower bound per **Max** $\delta \sim n$ (non eccezionale, in quanto pari al sequenziale, andrà abbassato), i processori più distanti in assoluto sono P_1 e P_n e serve passare $n - 1$ collegamenti.
- **Ampiezza di bisezione** $\beta = 1$, lower bound per Ordinamento $n/2$, rappresenta il bottleneck per quanto riguarda l'afflusso dei dati da una metà all'altra dell'architettura.

Ricordiamo che su PRAM il massimo e ordinamento hanno tempo logaritmico e processori $n/\log n$ e n rispettivamente.

Negli array lineari è **necessaria comunicazione**, bisogna quindi aggiungere, come minimo, il tempo necessario per quest'ultima.

2.4.1 Primitiva Swap contiguo

Scambiare i dati in k e $k + 1$. Abbiamo due **processori consecutivi**, quindi collegati direttamente, P_k contiene il dato $A[k]$, p_{k+1} contiene il dato $A[k + 1]$ e bisogna scambiarli.

Per scambiare i dati:

$$\begin{array}{ccc} P_k & & P_{k+1} \\ \mathbf{S} \rightarrow & & \leftarrow \mathbf{S} \\ \mathbf{R} \leftarrow & & \rightarrow \mathbf{R} \\ A[k] = A[k + 1] & A[k + 1] = A[k] \end{array}$$

Dove **S** e **R** indicano i comandi **send** e **receive**.

Bisogna fare i due **send**, al passo dopo i due **receive** e infine lo swap effettivo dei dati. Servono quindi 3 passi paralleli.

Quindi il tempo è costante, dato che i processori sono collegati tra loro, altrimenti bisognerebbe considerare la distanza tra i processori.

2.5 Problema Shuffle

In **input** un vettore di lunghezza pari, in **output** la prima metà dell'input innestata dalla seconda metà.

Idea:

Input:	A[1]	A[2]	...	A[S]	A[S+1]	...	A[2S]
Output:	A[1]	A[S+1]	A[2]	A[S+2]	...	A[S]	A[2S]

Questo problema si può risolvere tramite **swap contigui** tra processori, **partendo dal centro**, creando una sorta di albero di swap contigui, scambiando elementi fino a portarli nella posizione corretta, con numero di scambi crescente fino a raggiungere le estremità dell'array.

Prestazioni:

- **Processori:** non bisogna arrivare all'estremità quindi ne basta uno in meno, ma comunque lineare

$$p = 2(S - 1)$$

- **Tempo:** il numero di passi è $S - 1$, più il costo dello swap, comunque lineare

$$t = 3(S - 1)$$

- **Efficienza:** considerando che lo shuffle sequenziale (senza memoria aggiuntiva) richiede $\Theta(S^2)$, abbiamo

$$E \sim \frac{S^2}{S \cdot S} = c \neq 0$$

quindi è un buon algoritmo.

2.6 Trasmettere Dati

Visto che i **dati** sono **distribuiti** tra le memorie private dei processori, servirà un'istruzione per spedirli. Ci sarà una **richiesta di spedizione** del dato $A[i]$ contenuto in P_i verso il processore P_j .

Definiamo la **primitiva** $\text{SEND}(i, j)$, dove i è l'indice del processore che spedisce e j del recipiente. Entra in gioco la distanza tra i due processori, dovendo seguire un cammino da P_i a P_j all'interno dell'architettura.

Per **inviare i dati**:

- P_i fa una **Send** verso P_{i+1}
- P_{i+1} fa una **Receive** ed una **Send** verso il processore successivo
- Si **ripete** fino ad arrivare a P_j che farà solo una **Receive**

In **totale** si hanno $2d(i, j)$ **passi paralleli**, dove $d(i, j)$ è la distanza tra P_i e P_j . Il **costo di trasmissione** per un dato in un array lineare è sempre 2 volte la distanza tra i processori.

La trasmissione **non è costante** come nella PRAM.

2.7 Problema Max

Vogliamo calcolare il **massimo** tra n elementi in input.

Definizione:

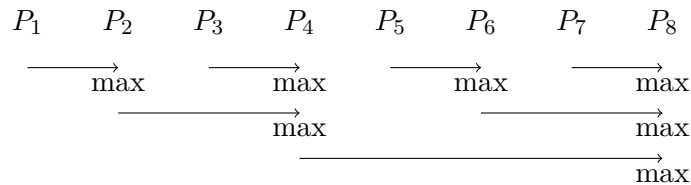
- **Input:** $A[1] \ A[2] \ \dots \ A[n]$, n valori
- **Output:** $\text{cont}(P_n) = \max\{A[i] | 1 \leq i \leq n\}$, ovvero il contenuto del processore con indice più alto deve essere il massimo tra tutti i valori

Il **tempo** per **Max** su array lineari è **limitato inferiormente da n** (su PRAM è $\log n$, su sequenziale è n).

Idea dell'algoritmo:

- Considerando l'algoritmo per sommatoria su PRAM (dato che vale per ogni operazione associativa)
- Riduzione dei passi per abbassate $\Omega(n)$ su array ad n processori

Esempio: P_1 invia a P_2 il valore e quest'ultimo calcola il massimo, stessa cosa con P_3 e P_4 e così via; al passaggio dopo P_2 invia a P_4 , P_6 a P_8 e si continuo fino al massimo globale.



Ogni riga un passaggio.

Procedimento: al j -esimo passo ogni processore si confronta con i valori contenuti a distanza 2^{j-1} e il risultato viene memorizzato nel processore di indice $2^j \cdot t$. Per il passaggio j -esimo si ha il confronto tra i processori $2^j t - 2^{j-1}$ e $2^j t$, memorizzando il risultato del calcolo in quest'ultimo, per ogni $t \in [1, n/2^j]$.

Il **numero di passi** è $\log n$, in quanto confronto a distanza doppia a ogni passo.

Codice:

```

for  $j = 1$  to  $\log n$ 
  for  $k \in \{2^j t - 2^{j-1} | 1 \leq t \leq n/2^j\}$  par do
    Send( $k$ ,  $k + 2^{j-1}$ )
  for  $k \in \{2^j t | 1 \leq t \leq n/2^j\}$  par do
    if ( $A[k] < A[k - 2^{j-1}]$ ) then
       $A[k] = A[k - 2^{j-1}]$ 

```

Si possono vedere **due fasi**, una in cui vengono inviati i dati (con il **Send**) e una seconda in cui viene effettuato il **Compare**.

Prestazioni:

- **Tempo:** dobbiamo considerare le due fasi
 - per la fase di **Send** il costo è $2 \cdot 2^{j-1}$, per ogni $j = 1, \dots, \log n$
 - per la fase di **Compare** il costo è 2, per ogni $j = 1, \dots, \log n$

In totale

$$t = \sum_{j=1}^{\log n} 2 \cdot 2^{j-1} + \sum_{j=1}^{\log n} 2 = 2n - 2 + 2 \log n = O(n)$$

- **Processori:** ne usa n , ma non va troppo bene, $E \rightarrow 0$.

Riduciamo i processori da n a p , **riducendo la distanza** δ e facendo in modo che ogni processore avrà n/p elementi al suo interno.

Quindi **ogni processore** farà il **Max** sequenziale tra i suoi n/p numeri e poi si esegue il **Max parallelo** su p **processori**.

Nuove **prestazioni**:

- **Processori**:

$$n = p$$

- **Tempo**:

$$t = O\left(\frac{n}{p}\right) + O(p)$$

- **Efficienza**:

$$E = \frac{n}{p \cdot \left(O\left(\frac{n}{p}\right) + O(p)\right)} = \frac{n}{O(n) + O(p^2)}$$

Volendo $O(n)$ al denominatore, per avere $E \rightarrow c \neq 0$ scelgo

$$p^2 = n \implies p = \sqrt{n}$$

Di conseguenza: **Max** è risolto in modo **efficiente** su **array lineari** con

$$p = \sqrt{n} \text{ e } T = O\left(\frac{n}{\sqrt{n}}\right) + O(\sqrt{n}) = O(\sqrt{n})$$

2.8 Swap Non Contiguo

Generalizziamo l'operazione dello **Swap** contiguo a processori non vicini.

Le connessioni sono full duplex, quindi le **Send** possono essere effettuate simultaneamente. Bisogna distinguere tra due casi, in quanto la distanza tra i processori può essere pari o dispari:

- **Primo caso:** distanza **dispari**

$$d(i, j) = 2k + 1$$

Quindi i processori coinvolti sono pari, sono $2k + 2$. Al centro abbiamo i processori P_{k+i} e P_{k+i+1} i quali saranno in tempo per fare due **Send** simultanee e due **Receive** simultanee al passaggio successivo. Si “incrociano” i dati e si continua.

Tempo:

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

I $2k$ sono dovuti alle **Send**, il 2 e l'1 sono dovuti, rispettivamente, allo scambio centrale e all'assegnamento finale. In totale diventa il costo delle **Send** più 1 per l'assegnamento

- **Secondo caso:** distanza **pari**

$$d(i, j) = 2k$$

Il numero di processori è dispari e ci sarà un processore centrale che deve gestire le due **Receive** contemporaneamente. Queste richieste andranno gestite sequenzialmente, allungando di 2 operazioni il tempo visto nel caso precedente.

Tempo

$$T = 2d(i, j) + 3$$

2.9 Primitiva MinMax

Vogliamo avere sul processore di indice inferiore il numero più basso.

Tra processori contigui basta fare:

P_k	P_{k+1}
Send	Send
Receive	Receive
if (A[k]>A[k+1])	if (A[k]<A[k+1])
A[k] = A[k+1]	A[k+1] = A[k]

Quindi impiega **tempo costante** $T = 4$.

Si può **generalizzare** a $\text{MinMax}(i, j)$, sfruttando $\text{Swap}(i, j)$ per poi fare il confronto.

Tempo per la versione generica:

- Distanza **pari**:

$$T = 2d(i, j) + 2$$

- Distanza **dispari**:

$$T = 2d(i, j) + 4$$

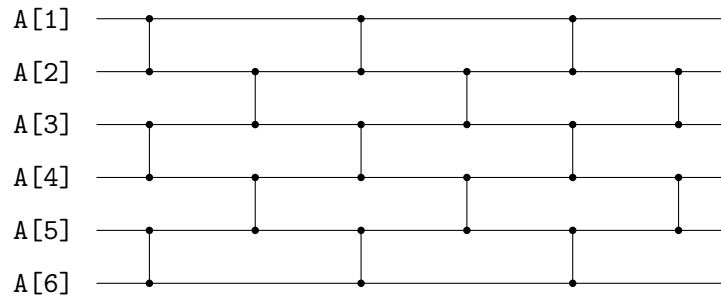
2.10 Problema Ordinamento

Definizione:

- **Input:** $A[1] \ A[2] \ \dots \ A[n]$
- **Output:** $cont(P_1) < cont(P_2) < \dots < cont(P_n)$, permutazione ordinata

Algoritmo **test/swap oblivious** descritto da una sorting network, chiamata **Odd-Even sorting network**.

Esempio con 6 elementi:



Si alternano **confrontatori** “*dispari*” con confrontatori “*pari*”, per n passaggi.

Correttezza: possiamo usare il principio 0/1, mostrando che l'algoritmo è in grado di **ordinare qualsiasi vettore binario**:

$$\{0, 1\}^n \rightarrow \boxed{\text{Odd-Even}} \rightarrow o^j 1^e$$

Partendo dal caso peggiore 111100, $e = 4$, $j = 2$. Ogni 1 nell'input deve **scendere** di un **numero di posizioni** $n - e = j$ (numero di zeri).

Regola generale: l' i -esimo 1 dal basso, impiega (al più) $n - e + i$ **passi** per posizionarsi correttamente.

Dato che $i \leq e$ si ottiene: $n - e + e = n$, al più n passi.

Per quest'algoritmo n **passi** sono **necessari e sufficienti**.

Implementazione **sequenziale**: $T(n, 1) = O(n^2)$ (sostanzialmente un BubbleSort).

Implementazione parallela: richiede n round di comparatori implementati con $\text{MinMax}(k, k+1)$.

Codice:

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

La condizione del **for** fa attivare alternativamente processori pari e dispari.

Prestazioni:

- **Tempo**:

$$t = n \cdot 4 = O(n)$$

- Numero di **processori**:

$$p = n$$

- **Efficienza**:

$$E = \frac{n \log n}{n \cdot n} \rightarrow 0$$

Osservazione: Per n processori in un array lineare l'ordinamento richiede, come **lower bound**, tempo $\Omega\left(\frac{n}{2^\beta}\right)$, dove $\beta = 1$.

2.10.1 Merge-Split

Riduzione dei processori: Al posto di usarne n ne **usiamo** p .

Ogni processore conterrà n/p dati e li **ordina sequenzialmente** in un **tempo**

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

La **primitiva MergeSplit** **sostituisce** il MinMax, avviene tra due **processori contigui**:

- Il processore di sinistra **spedisce** n/p dati ordinati al processore di destra. Tempo $O(n/p)$.
- Il processore di destra **riceve e fonde** (Merge) i nuovi n/p dati con i suoi dati (anch'essi ordinati). Tempo $O(n/p)$
- Il processore di destra **invia** gli n/p dati più piccoli al processore di sinistra (Split). Tempo $O(n/p)$

Ogni primitiva **MergeSplit** viene **ripetuta** per p round. Questo è l'algoritmo Odd-Even con MergeSplit al posto di MinMax.

Prestazioni:

- **Tempo**

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

- Numero di **processori**:

$$p(n) = p$$

- **Efficienza:**

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

Scegliendo $p = \log n$

$$E = \frac{n \log n}{n \log n} \rightarrow c \neq 0$$

Osservazione: Il **tempo** è **rimasto** $O(n)$. La riduzione dei processori agisce sul diametro e non sull'**ampiezza di bisezione** $\beta = 1$.

2.11 Architettura Mesh

Per migliorare i valori ottenuti sull'array lineare si può pensare di modificare l'architettura, formando un array bidimensionale, ovvero una griglia di processori (vengono utilizzate per i supercomputer), chiamata **mesh**.

Per n processori abbiamo una **griglia** $m \times m$, dove $m = \sqrt{n}$. Ogni processore è collegato a i processori adiacenti, ma anche a quelli sopra e sotto (sai come funziona una griglia dai, ogni nodo è un processore).

Se n non è un quadrato perfetto:

$$(\lfloor \sqrt{n} \rfloor + 1)^2 \leq 2n \text{ per } n \geq 6$$

Tutte le considerazioni per valutare il numero di processori rimangono uguali anche se n non è un quadrato perfetto. Da n processori, per ottenere una mesh che lavora su un quadrato perfetto mi servono al massimo $2n$ processori, quindi sempre **lineare in** n .

Parametri di rete:

- **Grado** $\gamma = 4$, un qualunque processore nella griglia può essere legato a destra, sinistra, sopra e sotto.
- **Diametro** $\delta \sim 2\sqrt{n}$, i processori più distanti sono agli estremi in diagonale, dobbiamo fare una “scala” da uno all'altro.
- **Ampiezza di bisezione** $\beta \sim \sqrt{n}$

I parametri sono diversi, ma sostanzialmente è la stessa cosa ma su un quadrato al posto che una linea.

Lower bound per:

- **Max:** $\Omega(\sqrt{n})$, guardando δ
- **Ordinamento:** $\Omega(\sqrt{n})$, guardando β

2.11.1 Max

Per trovare il massimo potremmo **usare la mesh come fosse un array lineare**, andando da sinistra verso destra, scendere per poi andare da destra verso sinistra e ripetere, usando lo stesso algoritmo usato per l'array lineare (la mesh diventa un array “piegato”, come una serpentina). Ma questo richiede $\Omega(n)$ **passi**, lontano dal lower bound di $\Omega(\sqrt{n})$.

Algoritmo righe-colonna: Ogni **riga** è un **array** lineare di \sqrt{n} **processori** e l'**ultima colonna** è anch'essa un array di \sqrt{n} processori. Quindi calcolo il massimo su ogni riga, spostando il valore nell'ultima cella, e poi calcolo il massimo sulla colonna finale.

Codice:

```
for i = 1 to  $\sqrt{n}$  par do
    Max( $p_{i1}, \dots, p_{im}$ )
Max( $p_{1m}, \dots, p_{mm}$ )
```

Il **for** richiede \sqrt{n} **passi**, l'ultimo max ne richiede altrettanti, quindi in **totale** si ottiene che $= O(\sqrt{n})$.

Ma l'efficienza:

$$E = \frac{n}{n \cdot \sqrt{n}} \rightarrow 0$$

Uso n processori per \sqrt{n} passi, al posto che solo n passi sequenziali.

Riduzione dei processori: Da n a p .

- Ogni processore calcola il **Max** tra n/p dati (sequenzialmente), tempo $O(n/p)$.
- Si usa l'algoritmo di prima sulla griglia di $\sqrt{p} \times \sqrt{p}$ processori, tempo $O(\sqrt{p})$.

Quindi in **totale** il tempo:

$$t = \frac{n}{p} + \sqrt{p}$$

Quindi l'**efficienza**:

$$E = \frac{n}{p \left(\frac{n}{p} + \sqrt{p} \right)} = \frac{n}{n + p^{\frac{3}{2}}}$$

Per farla tendere a un valore $\neq 0$ vorremmo qualcosa della forma n/n , quindi **scelgo**:

$$p^{\frac{3}{2}} = n \implies p = n^{\frac{2}{3}}$$

Il **tempo totale**:

$$t = \frac{n}{p} + \sqrt{p} = \sqrt[3]{n} + \sqrt[3]{n} = O(\sqrt[3]{n})$$

Così l'**efficienza tende a una costante**.

Il **lower bound** va **ricalcolato**, in quanto è cambiata la dimensione della mesh: $\sqrt{p} \times \sqrt{p}$ ha come lower bound $\Omega(\sqrt{p})$ per il **Max**, ma con $p = n^{\frac{2}{3}}$:

$$\Omega(\sqrt{p}) \implies \Omega(\sqrt[3]{n})$$

Quindi questo algoritmo risulta ottimale, vicino al limite inferiore.

2.11.2 Ordinamento

Su n processori il lato della mesh è \sqrt{n} . Una matrice M di lato m .

L'**input** è distribuito in una serpentina, da sinistra a destra nelle righe dispari, da destra a sinistra per le righe pari, un dato per processore.

L'**output** sono gli n valori ordinati in ordine crescente quando letto nella stessa maniera dell'input.

Ordinamento LS3: Si tratta di un algoritmo ricorsivo *divide et impera*. Prende come input il “quadrato” di dati.

Fasi:

- **Dividi:** Si divide il quadrato M di dimensione $\sqrt{n} \times \sqrt{n}$ in 4 parti M_1, M_2, M_3, M_4 , ognuna di dimensione $\sqrt{n}/2 \times \sqrt{n}/2$.
- **Ordina:** Con le chiamate ricorsive viene ordinata ognuna delle parti.
- **Fondi:** Vengono riunite le sotto-matrici

Codice:

```
if (|M| == 1)
    return M
else
    LS3Sort(M1) //In parallelo
    LS3Sort(M2)
    LS3Sort(M3)
    LS3Sort(M4)

    LS3Merge(M1, M2, M3, M4)
```

Quindi **LS3Merge** si occupa di in una sola le 4 matrici ordinate. Come viene costruita? Riunisce la matrice “in place” ed effettua due routine, **Shuffle** (per la descrizione vedi il capitolo relativo) e **Odd-Even** (vedi capitolo relativo).

In **LS3Merge** dopo che vengono riunite le matrici, lo **Shuffle** viene effettuato su **ogni riga della matrice**, come se fosse un array lineare.

In seguito effettua l'ordinamento **Odd-Even** su due **colonne adiacenti**, viste come fossero un array lineare lungo $2\sqrt{n}$, considerandole come connesse a serpentina.

Codice per LS3Merge:

```
for i = 1 to  $\sqrt{n}$  par do
    Shuffle(i)
for i = 1 to  $\sqrt{n}/2$  par do
    Odd-Even( $2i - 1$ ,  $2i$ )
Esegui i primi  $2\sqrt{n}$  passi di Odd-Even
sull'intera mesh (a serpente)
```

Quindi fa lo **Shuffle** di tutte le colonne (in parallelo), poi fa **Odd-Even** di tutte le colonne adiacenti. L'ultimo passo è eseguire **Odd-Even** per $2\sqrt{n}$ passi sull'intera mesh (a serpente).

Sostanzialmente, abbiamo le 4 sotto-matrici ordinate, **Shuffle** e **Odd-Even** “spingono” tutti i valori più grandi verso il basso, la parte finale di **Odd-Even** corregge i possibili sbagli, che per forza di cose devono trovarsi tra i primi $2\sqrt{n}$ elementi.

Tempo per LS3Merge: sia **Shuffle** che **Odd-Even** hanno **tempo lineare**, quindi $O(\sqrt{n})$, in totale il merge richiede un certo numero di volte \sqrt{n} , che chiameremo $h\sqrt{n}$.

Tempo totale per LS3: Richiede di dividere l'input in 4, poi tempo di LS3Merge. Risolviamo:

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

$$T(n) = T\left(\frac{n}{4}\right) + h\sqrt{n} = \sum_{i=0}^{\log_4(n-1)} h\sqrt{\frac{n}{4^i}} + 1 = 2h\sqrt{n} \left(1 - \frac{1}{\sqrt{n}}\right) + 1 = O(\sqrt{n})$$

Possiamo vedere che la ricorsione si ferma a $n = 4$ e che la sommatoria corrisponde alla serie geometrica.

Eseguendo l'algoritmo sequenzialmente, quindi inserendo un 4 prima di $T(n/4)$, dall'equazione di ricorrenza risulterà $O(n\sqrt{n})$, quindi un tempo non ottimale per il sequenziale.

Prestazioni:

- **Tempo:**

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

(minimo teorico)

- **Processori:**

$$p(n) = n$$

- **Efficienza:**

$$E = \frac{n \log n}{n\sqrt{n}} \rightarrow 0$$

Ma si può migliorare riducendo i processori tramite una versione del **BitonicSort** su mesh che ha processori $p(n) = O(\log^2 n)$ e tempo $T(n) = O(n/\log n)$, quindi efficiente, anche se peggiore in termini di tempo.

3 Algoritmi Distribuiti

In un ambiente di calcolo distribuito, possiamo pensare alle entità di calcolo come nodi all'interno di un **grafo direzionato** \vec{G} , dove ogni arco indica un collegamento.

Ogni **entità possiede**:

- Memoria locale
- Capacità di calcolo
- Capacità di comunicazione
- **Clock locale**

La differenza dal capitolo precedente è che i **processori sono asincroni**.

Nella memoria locale abbiamo:

- Registro di **input**: $\text{valore}(x)$, input dell'entità x
- Registro di **stato** = $\text{stato}(x)$, stato dell'entità x ; questo valore può essere cambiato localmente dalla stessa x

Per il **clock locale** è possibile settare o resettare una **sveglia**.

Proprietà delle entità:

1. **Reattive:** all'accadere di un "evento" compiono un'azione. Gli eventi possono essere:

- **Interni** al sistema: ricezione di messaggi, sveglia
- **Esterni** al sistema: impulso spontaneo (impulso di start)

L'entità sollecitata dall'evento risponde con un'azione: una **sequenza finita di operazioni indivisibili** (deve arrivare al termine).

2. **Le entità seguono delle regole:** una **regola** è un oggetto della forma $\text{stato} \times \text{evento} \rightarrow \text{azione}$.

Sia x un'entità, allora $B(x)$ è l'**insieme delle regole a cui è soggetta** x . $B(x)$ deve essere **completo** (ogni coppia stato-evento deve avere un'azione specificata) e **non ambiguo** (con una singola interpretazione possibile).

Se E è un'**insieme di entità** che collaborano tra loro allora $B(E)$ è il **comportamento del sistema**

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

è importante che E sia **omogeneo**:

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

Quando $B(E)$ è omogeneo viene chiamato **protocollo** per E o **algoritmo distribuito** per E .

Fact: Sempre possibile ottenere $B(E)$ omogeneo.

Proof. L'idea è di utilizzare un registro locale aggiuntivo che **differenzia** quelle **entità** che alla **stessa coppia stato-evento hanno azioni diverse**. Si usa

- **ruolo(x)**: registro locale di x che contiene il ruolo per x

La regola diventa:

$\text{stato} \times \text{evento} \rightarrow \text{If ruolo}(x) \quad \text{then } A_a$
 $\quad \quad \quad \text{else } A_b$

□

Proprietà della rete:

1. La **comunicazione** tra entità **avviene usando link**, ed è importante che ogni entità conosca i propri collegamenti. Per ogni entità x , si ha un **etichettatura** denotata con λ_x e dato che si trova in \vec{G} , si indicano con:

- $N_{in}(x)$ = vicini in **ingresso** ad x
- $N_{out}(x)$ = vicini in **uscita** da x



Per x la funzione sarà $\lambda_x(x, y)$, mentre per y l'etichettatura sarà $\lambda_y(x, y)$.

2. **Assiomi di rete:**

- **Ritardo finito di comunicazione:** in assenza di errori, un messaggio spedito prima o poi arriverà
- **Orientamento locale:** ogni entità riesce a distinguere tra i suoi vicini $N_{in}(x)$ e $N_{out}(x)$ grazie alla conoscenza della funzione λ_x

Parametri di rete:

- **Numero di entità** n (nodi).
- **Numero di link** m (archi).
- **Diametro** della rete d (massima distanza tra due nodi).

Oltre agli assiomi possiamo avere delle **restrizioni sulla rete**, da dichiarare al momento della scrittura del codice. Generalmente sono proprietà positive della rete, su cui fare affidamento.

Restrizioni sulla comunicazione:

- **Link bidirezionali:** le connessioni tra le entità sono full duplex, si passa da un grafo diretto ad uno non diretto..
 $\forall x, N_{in}(x) = N_{out}(x) = N(x)$, quindi anche $\lambda_x(x, y) = \lambda_x(y, x)$.
- **Ordinamento dei messaggi:** i messaggi sullo stesso link vengono prelevati con politica FIFO (il primo inviato è il primo ad arrivare)

Restrizioni sull'affidabilità:

- **Rilevazione di errori:** ad esempio, quando cade un'entità o si guasta un canale di comunicazione.
- **Affidabilità parziale:** non ci saranno errori in futuro.
- **Affidabilità totale:** non ci sono stati errori e non ce ne saranno in futuro.

Restrizioni sulla topologia di rete:

- **Connettività del grafo:** \vec{G} è fortemente connesso, G è connesso.

Restrizioni sul tempo:

- **Tempi di comunicazione unitari:** la comunicazione impiega sempre un singolo ciclo di clock.
- **Clock sincronizzati:** come fosse un'architettura parallela a memoria distribuita.

Nota: tali restrizioni a volte vengono considerate per il calcolo delle prestazioni ideali del codice distribuito.

3.1 Misure di Complessità

Tempo: Viene considerato l'**intervallo** tra la **prima entità che si attiva** e l'**ultima che termina**. Esecuzioni diverse dello stesso codice può portare a tempi diversi in base alla congestione della rete.

Questo problema si risolve considerando il **tempo ideale**, si misura il tempo supponendo comunicazioni unitarie e clock sincroni.

Il tempo ideale si contrappone al **tempo causale** (caso peggiore), ovvero il tempo misurato considerando la catena più lunga di comunicazione richiesta dal codice; il peggiore di tutte le situazione che possono accadere.

Quantità di Comunicazione: Si misura in termini di **numero di messaggi spediti**, se i messaggi sono omogenei/della stessa dimensione, altrimenti si valuta il **numero di bit spediti**.

3.2 Definizione di un Problema

In genere un **problema** viene definito da una **trippla**:

$$P = \langle P_{init}, P_{final}, R \rangle$$

Dove:

- P_{init} e P_{final} sono dei **predicati logici** che descrivono le **configurazioni del sistema all'inizio** (come sono inizializzati i registri all'interno delle entità all'inizio del protocollo) **e alla fine** (come devono essere i registri al termine).
- R rappresenta le restrizioni del sistema, ad esempio full-duplex, no-errori, ...

Esempio di definizione P per Broadcast:

- P_{init} : una sola entità detiene l'informazione I e le altre identità non la hanno, i.e.,

$$\exists x \in E \text{ t.c. } valore(x) = I \wedge \forall x \neq y \text{ } valore(y) = \emptyset$$

- P_{final} : tutte le entità posseggono I

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

- R : link bidirezionali BL, totale affidabilità TR, connettività CN; queste 3 restrizioni si indicano con la sigla R; l'ultima restrizione è unico iniziatore UI; queste 4 restrizioni assieme si indicano con RI

3.3 Protocollo

Un algoritmo distribuito, o **protocollo**, è un **insieme di regole della forma**:

$$\text{stato} \times \text{evento} \rightarrow \text{azione}$$

Lo stato_t(*x*) è lo stato dell'entità *x* al tempo *t*.

L'evento può essere l'impulso spontaneo, la sveglia o la ricezione di un messaggio.

L'azione è un mini programma indivisibile che l'entità deve eseguire alla ricezione di un evento.

L'**esecuzione di un protocollo** è una **sequenza di configurazioni successive del sistema** (l'esecuzione porta a cambiamenti).

Definizione di configurazione: Gli elementi che definiscono una configurazione sono:

- $\Sigma(t)$: il **contenuto dei registri** delle entità al tempo *t*
- *Futuro*(*t*): **eventi** già **generati** al tempo *t* ma **non ancora processati** (messaggi inviati non arrivati, sveglie non ancora suonate, ...)

Indichiamo la **configurazione del sistema** al tempo *t* con *C*(*t*)

$$C(t) = (\Sigma(t), Futuro(t))$$

Per esempio, la configurazione iniziale *C*(0) è data dai registri inizializzati ($\Sigma(0)$) e dall'impulso spontaneo, presente ma non ancora processato (*Futuro*(0)).

L'**esecuzione del protocollo** distribuito viene descritta da una **sequenza di configurazioni successive**, tale che

$$C(0) \xrightarrow{\text{protocollo}} C(f)$$

Va dallo stato iniziale *C*(0) allo stato finale *C*(*f*).

Notazione: Quando una **configurazione** C soddisfa un **predicato** P scriveremo $C \in P$.

Bisogna **definire** come un protocollo distribuito **risolve un problema**.

3.3.1 Broadcast

Prima versione: Riprendendo l'**esempio** del Broadcast:

- P_{init} : $\exists x \in E$ t.c. $valore(x) = I \wedge \forall y \neq x$ $valore(y) = \emptyset$
- P_{final} : $\forall x \in E$ $valore(x) = I$
- $R = RI = (BL, TR, CN, UI)$

Definendo due stati:

$$S = \{iniziatore, inattivo\}$$

E indichiamo con:

- S_{init} gli stati delle entità nella configurazione iniziale $C(0)$
- S_{term} gli stati delle entità nella configurazione finale $C(f)$

Quindi, al termine vogliamo che tutte le entità siano sullo stato *inattivo*.

Le regole vengono specificate con stato, evento e azione, in quest'ordine, quindi:

Iniziatore:

```

    impulso spontaneo
    {
        send(M) to N(x)
        become inattivo
    }

```

Inattivo:

```

    ricezione(M)
    {
        processa(M)
        send(M) to N(x)
    }

```

L'iniziatore manda a tutti i suoi vicini il dato, per poi diventare inattivo, mentre tutte le altre entità, alla ricezione del dato, lo inoltrano ai vicini.

Quindi possiamo ipotizzare che il messaggio M sia una quadrupla:

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

Con:

- t : tipologia del messaggio
- o e d : origine e destinazione
- I : informazione

Ma c'è un problema: il protocollo è **corretto** (diffonde l'informazione tra tutte le entità) ma **non termina**.

Bisogna evitare di inviare il messaggio anche all'entità dalla quale è stato ricevuto. Inoltre, un'entità viene attivata anche se è già stata attivata in precedenza, formando cicli.

$Futuro(t) \neq \emptyset$, non c'è un momento senza messaggi in transito, quindi la comunicazione non termina.

La soluzione al problema è stabilire più stati, **raffinando gli stati**:

- $S_{start} \subseteq S_{init}$: stati che fanno iniziare il protocollo
- $S_{final} \subseteq S_{term}$: stati per cui la **sola azione è quella nulla**

Per **Broadcast**, $S_{start} = \{iniziatore\}$ mentre $S_{init} = \{iniziatore, inattivo\}$, per gli stati finali ne introduciamo uno nuovo: $S_{term} = S_{final} = \{finito\}$. 3 stati totali.

Grazie all'uso dello stato "*finito*" il protocollo riesce a terminare. Dopo essere *inattivo* un'entità entra nello stato *finito*, non rimane più nello stesso stato.

Formalmente, possiamo dire che la soluzione per P :

- è **corretta** se

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

per ogni condizione iniziale, a un certo punto si arriva a una delle configurazioni finali.

- **Termina** se

$$\forall x \in E \quad stato_t(x) \in S_{final}$$

ogni entità presente è in uno stato finale.

Devono valere entrambe.

Seconda versione (Flooding): Gli stati sono

$$S = \{iniziatore, inattivo, finito\}$$

$$S_{start} = \{iniziatore\}$$

$$S_{final} = \{finito\}$$

L'iniziatore fa partire il protocollo, le entità nello stato finito non eseguono più operazioni.

Codice:

```
Iniziatore:
impulso spontaneo
{
    send(M) to N(x)
    become finito
}

Inattivo:
ricezione(M)
{
    processa(M)
    send(M) to N(x) - sender
    become finito
}
```

Le coppie stato-evento non indicate eseguono azione nulla.

Complessità:

- **Numero di messaggi M :**

$$M[\textit{Flooding}] = \sum_{x \in E} (N(x) - 1) + 1 = 2m - n + 1$$

Dove m è il numero di archi e n il numero di nodi. Fa mandare il messaggio a tutti i vicini, eccetto il sender.

- **Tempo T :**

$$T[\textit{Flooding}] \leq d$$

Dove d è il diametro della rete. Per questo calcolo si suppone clock sincroni e ritardo unitario per la consegna dei messaggi. Il tempo è pari al diametro nel caso peggiore, può essere meno.

Lower bound del problema:

- **Tempo causale** (caso peggiore) per il problema

$$T[\textit{BCast/RI}] \geq d$$

Non si possono impiegare meno di d passi.

- **Numero di messaggi**, c'è un teorema che dice

$$M[\textit{BCast/RI}] \geq m$$

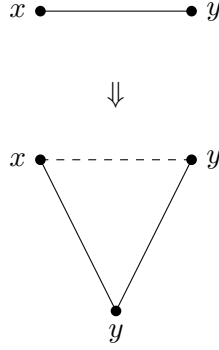
Non si può scendere sotto m .

Flooding è quindi ottimale ($2m$ messaggi, lineare) per il problema di **Broadcast**.

Teorema: $M[BCast/RI] \geq m$.

Proof. Per assurdo, risolvo il problema con meno di m messaggi. Vuol dire che esiste un arco su cui non viaggiano messaggi. Sia A il protocollo che non manda il messaggio sull'arco (x, y) . Se A è corretto deve lavorare bene su ogni G .

Creo il grafo G' aggiungendo un nodo z tra x e y , rimuovendo l'arco (x, y) e aggiungendo (x, z) e (z, y) . Cambio le etichettature $\lambda_x(x, z) = \lambda_x(x, y)$ e $\lambda_y(y, z) = \lambda_y(y, x)$.



Se eseguo il protocollo A su G' il nodo z non riceve I , dato che sull'arco (x, y) non viaggiava nessun messaggio questo accade anche su (x, z) , quindi A non è corretto.

□

3.4 Problema Wake-Up

Nel problema **Broadcast**, c'era una singola entità che doveva raggiungere tutte le altre, mentre per **Wake-up** possono esserci più entità che iniziano il protocollo. Si rilassa il vincolo di unico iniziatore UI. Ci sono entità “dormienti” e altre che sono “svegliate”, bisogna svegliarle tutte.

3.4.1 Protocollo WFlood

Stati

$$S = \{dormiente, attivo\}$$

$$S_{init} = \{dormiente\} = S_{start}$$

$$S_{term} = \{attivo\} = S_{final}$$

Ora però l'impulso spontaneo può arrivare a un certo numero di entità.

Dormiente:

```
    impulso spontaneo
    {
        send(W) to N(x)
        become attivo
    }
    ricezione(W)
    {
        send(w) to N(x) - sender
        become attivo
    }
```

Ci può essere l'impulso spontaneo oppure la ricezione di un messaggio per passare allo stato attivo.

Costo:

- **Tempo:**

$$T[W\textit{Flood}] \leq d$$

Dove d è il diametro.

- **Numero di messaggi:**

$$2m - n + 1 \leq M[W\textit{Flood}] \leq 2m$$

Il bound inferiore è se una sola entità si attiva, mentre il $2m$ è il caso in cui si attivano tutte.

3.5 Problema Traversal

Ogni entità della rete deve essere visitata, ma **sequenzialmente**, ovvero una dopo l'altra.

Risulta utile in ambito di gestione condivisa delle risorse.

La configurazione iniziale è che tutti i nodi sono *unvisited*, tranne uno.

3.5.1 Depth-First Traversal

Il protocollo da definire sarà **Depth-First Traversal**, equivalente alla visita in profondità di un grafo; si scende sempre verso il vicino non ancora visitato.

Idea: viene usato un messaggio particolare **Token** T . Quando un nodo riceve il token T diventa *visited*, ma ad ogni istante di tempo viaggia **al più un solo** T .

Passaggi principali dell'algoritmo:

1. Un nodo che **riceve** T per la **prima volta** deve:
 - ricordarsi il sender (non dovrà rinviargli il token)
 - fare una **lista dei vicini non visitati** (ma non lo sa, per ora tutti eccetto il sender)
 - invia T ad uno di essi
 - aspetta un messaggio da quest'entità, che può essere **return** o **back-edge**
2. Il **vicino che riceve** T :
 - se è il **primo** T che riceve, va al **passaggio 1**
 - altrimenti, se è **già stato visitato**, spedisce un **back-edge**
3. Solo dopo aver **finito la lista dei vicini non visitati** deve inviare un **return** al sender.

Quindi sono presenti **3 tipi di messaggi**:

- Token T
- Back-edge B
- Return R

Protocollo: Stati:

$$S = \{initiator, idle, visited, done\}$$

$$S_{init} = \{initiator, idle\}$$

$$S_{term} = \{done\}$$

Codice:

Initiator

```
Spontaneously
{
    initiator = true
    unvisited = N(x)
    visit
}
```

Idle

```
Receive (T)
{
    entry = sender
    unvisited = N(x) - sender
    initiator = false
    visit
}
```

```

Procedure visit
{
    if unvisited  $\neq \emptyset$  then
        next  $\leftarrow$  unvisited
        send(T) to next
        become visited
    else
        if initiator = false then
            send(return) to entry
            become done
}

```

```

Visited
Receiving(return)
{
    visit
}
Receiving(back-edge)
{
    visit
}
Receiving(T)
{
    unvisited = unvisited - sender
    send(back-edge) to sender
}

```

Complessità: Osservazione: su ogni link deve viaggiare il **token** ed in risposta ci sarà un **return** o **back-edge**. Il tutto avviene praticamente in maniera **sequenziale**.

Quindi il **tempo coincide con i messaggi**

$$T[DF - Traversal] = M[DF - Traversal] = 2m$$

Lower bound al problema Traversal:

- $M[Traversal] \geq m$, vale lo stesso teorema visto per il Broadcast
- $T[Traversal] \geq n - 1$, dato che ogni nodo deve essere visitato in sequenza

Nel caso di un grafo connesso, il **numero di archi** m è

$$n - 1 \leq m \leq \frac{n(n - 1)}{2} = O(n^2)$$

Di conseguenza DF-Traversal:

- è **ottimo** per il **numero di messaggi**
- ma nel caso peggiore **NON** per il **tempo** (n)

Osservazione: Il problema per il costo del tempo è che ad ogni istante di tempo **viaggia un messaggio T solo**.

La soluzione è introdurre concorrenza, aggiungendo messaggi in quantità opportuna ($O(m)$).

Possiamo evitare di inviare T su un link **back-edge**?

Idea: un nodo **non visitato** che **riceve T comunica** l'evento ai **vicini** mandando un **messaggio visited** (in contemporanea con T), i vicini che ricevono **visited** aggiornano la propria lista unvisited.

Ma questo risolve il problema di inviare T su **back-edge**? Non completamente, si può vedere in casi di ritardo della comunicazione, in cui il **visited** impiega tempo ad arrivare.

Ma quando un'entità “capisce l'errore” e riceve un **visited**, spedisce T ad un'altra entità, oppure invia un **return** e sarà l'entità che lo riceve a spedire un altro T .

Inoltre, se un nodo riceve T : lo interpreta come fosse un **back-edge** ed elimina il nodo dalla lista di unvisited, in quanto sicuramente è già stato visitato.

Nuova complessità:

- **Messaggi:**

$$\begin{array}{ll}
 2n - 2 & \leftarrow T + R \\
 2m - (n - 1) & \leftarrow \textit{visited} \\
 2(m - (n - 1)) & \leftarrow \text{errori di invio di } T \\
 \Rightarrow O(m) & \text{totale}
 \end{array}$$

- **Tempo:** considerando tempo ideale senza ritardi nè errori. Inoltre i **visited** son in sovrapposizione con T . Totale:

$$2(n - 1) = O(n)$$

Chiamando il protocollo modificato DF^* , questo è **ottimo** per

- Quantità di **messaggi** $M[DF^*] = O(n)$
- **Tempo** ideale $T[DF^*] = O(n)$

In quanto rispetta i lower bound.

Codice non richiesto all'esame, non lo riporterò, definisce il protocollo DF^* .

3.6 Problema Spanning Tree

I problemi Broadcast, Wake up e Traversal hanno una **complessità di messaggi** che è $\Theta(m)$, dove m è il numero di link e n il numero di entità.

Quindi la **complessità varia** in base alla **topologia del grafo**

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

$n - 1$ è il minimo numero di link per collegare n entità, come in un albero, mentre l'estremo opposto è un grafo completamente connesso.

Idea: Quindi per **minimizzare** la **complessità di comunicazione** si potrebbe **utilizzare una sottorete** al posto dell'intero grafo G , e la sottorete minima è quella generata da uno **spanning tree**. Poi si possono risolvere gli altri problemi sullo spanning tree.

Il **costo** per risolvere i problemi diventa somma di:

- **costo di costruzione dell'albero**
- **costo originale** del problema eseguito sull'albero

Ad esempio, il costo di **Broadcast** con algoritmo Flooding su uno spanning tree di n entità è esattamente $n - 1$ messaggi.

Problema dello **Spanning Tree**: vogliamo **costruire una sottorete** tale che:

- coinvolga **tutte le entità**
- le entità siano **connesse**
- **priva di cicli**

Risolvere un algoritmo distribuito sulla sottorete richiede di tenere conto del fatto che l'algoritmo deve essere a **conoscenza dell'albero su cui viene eseguito**. **Ogni entità** vedrà (tiene traccia di) una **piccola parte dell'albero**.

Notazione:

- $\forall x \in E$ definiamo

$$Tree_N(x) \subseteq N(x)$$

sottoinsieme che contiene i **vicini** di x **all'interno dell'albero**.

- L'insieme dei **lati** per l'albero:

$$(x, y) \in link(Tree_N(x)) \Leftrightarrow y \in Tree_N(x)$$

quindi un lato (x, y) è all'interno di $link(Tree_N(x))$ solo se $y \in Tree_N(x)$.

- L'intero albero è

$$Tree = \bigcup_{x \in E} link(Tree_N(x))$$

ovvero, l'unione di tutti i collegamenti per ogni nodo.

Risolviamo il problema **Spanning Tree** con le **restrizioni RI** (Bidirectional Link, Total Reliability, Connectivity, Unique Initiator).

Il protocollo che risolve il problema dello **Spanning Tree** si chiama **Shout**:

- ogni **entità vede** solo i suoi $Tree_N(x)$
- e tiene traccia del **padre**

La **radice** dell'albero è data dall'**entità che inizia** il protocollo.

La strategia di **Shout** è chiedere ai vicini se questi vogliono diventare vicini anche nell'albero.

Schema:

- L'iniziatore s (root) **spedisce** la **domanda** Q ai suoi **vicini** e attende le risposte (chiede se vogliono diventare figli).
- Ogni entità $x \neq s$ che **riceve** Q :
 - la **prima volta** risponde “*yes*” (viene connessa all'albero) e **invia** Q ai suoi vicini e si mette in attesa
 - le **volte successive** risponde “*no*” (già connessa)
- Inoltre serve **memorizzare l'entità padre** e le **entità che rispondono** “*yes*”.
- L'entità **termina quando riceve tutte le risposte**.

Shout è sostanzialmente un Flooding + Reply. Si diffonde un messaggio Q a cui bisogna rispondere “*yes*” o “*no*”.

Linee guida per definire il codice:

- Tre **tipi di messaggi**: Q , *yes*, *no*.
- Ogni entità ha la **visibilità locale del proprio** albero, quindi bisogna **aggiornare le variabili locali**: *root* (bool per indicare se è la radice), *parent* (tiene traccia del padre, uno dei nodi all'interno della lista), *Tree_N(x)* (lista di vicini di albero), *counter* (conta il numero di risposte ricevute a Q , permette di terminare correttamente).
- **Aggiornare lo stato**, vanno utilizzati in maniera corretta per raggiungere la terminazione.

3.6.1 Protocollo Shout

Gli **stati** sono:

$$S = \{iniziatore, inattivo, attivo, finito\}$$

$$S_{init} = \{iniziatore, inattivo\}$$

$$S_{final} = \{finito\}$$

Nella configurazione iniziale ci sarà un solo *iniziatore* e tutte le altre entità *inattive*, che diventeranno *attive* quando ricevono il primo Q , e ci rimarranno fino al momento in cui a loro volta non riceveranno tutte le risposte, diventando *finite*.

Bisogna definire le azioni per *iniziatore*, *inattivo*, *finito*.

Per quanto riguarda l'iniziatore:

Iniziatore

```
Impulso spontaneo
{
    root = true
    counter = 0
    Tree_N( $x$ ) =  $\emptyset$ 
    send( $Q$ ) to  $N(x)$ 
    become attiv
}
```

Ogni inattivo:

Inattivo

```
Ricezione(Q)
{
    root = false
    parent = sender
    counter = 1
    Tree_N(x) = {sender}
    send("yes") to sender
    if counter = |N(x)| then
        become finito
    else
    {
        send(Q) to N(x)\sender
        become attivo
    }
}
```

Ogni attivo:

Attivo

```
Ricezione(Q)
{ send("no") to sender }
Ricezione("yes")
{
    Tree_N(x) = Tree_N(x)  $\cup$  {sender}
    counter = counter + 1
    if counter = |N(x)| then
        become finito
}
Ricezione("no")
{
    counter = counter + 1
    if counter = |N(x)| then
        become finito
}
```

Correttezza di Shout:

- **Terminazione:** in assenza di errori, viene sempre ricevuto un numero di risposte pari ai Q inviati, diventando finito.
- **Tutte le entità presenti:** grazie al Flooding di Q .
- **Le entità sono connesse:** grazie al fatto che al primo Q rispondo “yes”.
- **Privo di cicli:** ogni entità risponde solo una volta con “yes” (tranne la radice, che risponde sempre “no”)

Costo:

- **Numero di messaggi:**

$$M[Shout] = 2M[Flooding] = 2[2m - (n - 1)] \sim 4m$$

- **Tempo:**

$$T[Shout] = T[Flooding] + 1 \leq d + 1$$

I **lower bound** per questo problema SPT sono:

$$M[SPT/RI] \geq m$$

$$T[SPT/RI] \geq d$$

Quindi abbiamo un protocollo **ottimale** $O(m)$ per i messaggi e $d + 1$ per il tempo.

3.6.2 Shout migliorato: Shout+

Per quanto riguarda il tempo non si può migliorare, ma posso **ridurre il numero di messaggi** (abbiamo un 4 come coefficiente)? I messaggi

- di “*yes*” sono necessari
- di “*no*” possono essere eliminati

Se la risposta è un “*no*” è perché l’entità ha già ricevuto in precedenza un Q a cui ha risposto “*yes*”, inviando contestualmente altri Q .

Quindi una entità attiva che **riceve un** messaggio Q può **interpretare** quest’ultimo **come un** “*no*”.

Questo rimuove la necessità di rispondere “*no*”, su ogni arco viaggerà una richiesta Q e una risposta, la quale può essere “*yes*” oppure un altro Q .

Nuovo costo di comunicazione:

$$M[Shout+] = 2m$$

Su **ogni link** viaggiano:

- o un Q seguito da *yes*
- o un Q seguito da un Q

Altra soluzione per lo spanning tree: Utilizzare il protocollo Traversal. Un arco fa parte dell’albero se su quel link passa un **return**, si costruisce l’albero in maniera sequenziale (tendenzialmente peggio di Shout che lo fa in parallelo).