# Appunti Di Calcolo Parallelo

June 2, 2018

# Contents

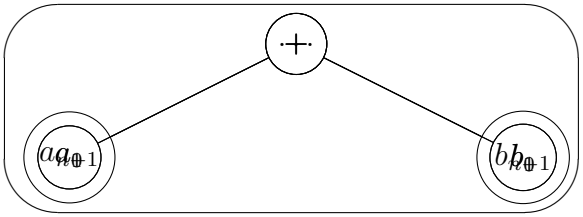
# 1 Grado di parallelismo

Il grado di parallelismo è il parallelismo potenziale, cioè un valore tale per cui se riscrivo l'algoritmo in modo opportuno posso esplicitare tale livello di parallelismo. Bisogna capire dunque quali operazioni possono essere svolte concorrentemente.

## 1.1 Esempi di algoritmi paralleli

**Somma di due vettori:**

$$c = a + b \tag{1}$$
$$a, b, c \in \mathbb{R}^n \tag{2}$$




Tale algoritmo è parallelizzabile se considero $a, b$ vettori con $n$ elementi, allora con $n$ addizionatori posso sommare gli $n$ elementi ottenendo il vettore $c$ con una sola operazione per ogni addizionatore.
$P_{MAX} = N$ (il livello di parallelismo è $N$)

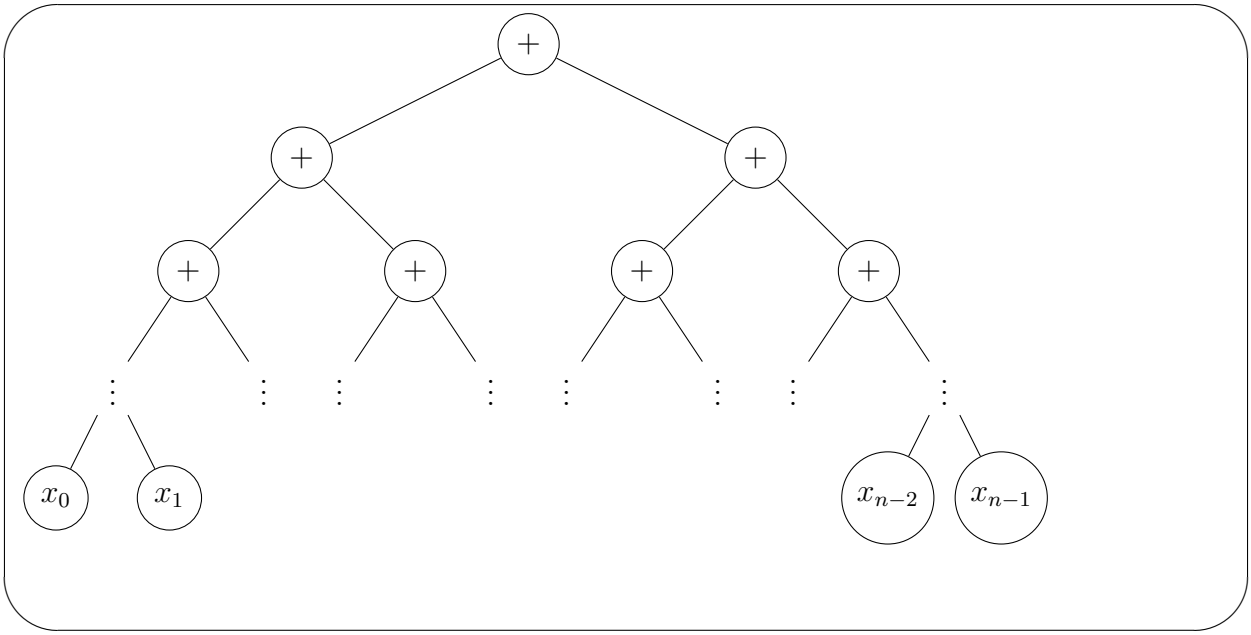
Cammino più lungo per arrivare da un input ad un output: si osserva il grafo che rappresenta le operazioni concorrenti per arrivare all'output e si identificano i cammini critici (nel caso precedente tutti i cammini hanno lunghezza $L = 1$).

**Somma componenti di un vettore:**

$$x = (x_0, x_1, ..., x_{n-1})$$
$$S = \sum_{j=0}^{N-1} x_j$$

$S \leftarrow x_0$
for $j \leftarrow 1$ to $n - 1$ do
    $S \leftarrow S + x_j$
end

Secondo questo schema l'algoritmo permette l'utilizzo concorrente di un numero di processori pari a $P_{MAX} = 1$, con lunghezza del cammino critico $L = N - 1$.
Tuttavia il problema può essere affrontato in modi diversi, ad esempio la seguente soluzione permette di sfruttare $P_{MAX} = N/2$ processori, con cammini critici di lunghezza $L = \log(N)$.




Il fatto che i due grafi precedenti relativi al problema della somma delle componenti del vettore producano lo stesso risultato è dovuto alla proprietà associativa dell'addizione. Tale proprietà vale per i numeri reali ma si deve considerare non applicabile a numeri floating point.

Ci possiamo fare un'altra domanda: se io utilizzo $P_{MAX}$ processori riesco ad utilizzarli in modo efficiente? Nel caso del primo problema la risposta è sì. Supponendo che l'utilizzo dei processori abbia un costo per ogni operazione (si pagano anche i cicli inutilizzati se ho deciso di utilizzare tali processori), nel primo caso ogni processore svolge una operazione e il programma termina e dunque non ho uno spreco di risorse. Nel secondo caso invece, solamente al primo stadio sono richiesti tutti gli $N/2$ processori, tuttavia ad ogni passo successivo la metà di questi rimangono inutilizzati.
Nel secondo caso dunque ho effettivamente accelerato il calcolo ma ho speso un costo eccessivo in quanto i processori inutilizzati potrebbero essere stati sfruttati altrove. Dunque se usiamo $N/2$ processori otteniamo un tempo migliore ma spendiamo troppo. Se utilizziamo un processore non abbiamo sprechi ma i tempi si allungano. E' interessante allora cercare di capire come variano sprechi e tempi al variare del numero di processori.

Per $1 \leq P \leq N/2$

Divido il vettore in blocchi di $N/P$ sottovettori. Faccio poi sommare i blocchi di $N/P$ ad un singolo processore seguendo lo schema sequenziale (idea di un tipo, poi il prof la deve impestare un po' perchè questa idea potrebbe non essere generalizzabile). Prendo le operazioni del primo stadio e le divido in gruppi da $P$, ottenendo il segunte schema:

Tale operazione è un insieme di $n^2$ di prodotti interni, già questo evidenzia un grado $N^2$ di parallelismo. A sua volta un prodotto interno consiste di $N$ moltiplicazioni.
Senza entrare troppo nei dettagli, posso ottenere $L = 1 + \log_2(N)$. Ogni prodotto interno ha infatti lunghezza del cammino $log_2(N)$ e gli $N^2$ distinti prodotti interni sono indipendenti l'uno dall'altro e sono eseguibili concorrentemente contribuendo per un fat-

tore 1 sulla lunghezza del cammino per generare ogni output. Tale valore di $L$ è vero per $P_{MAX} = N^3$.

Si può fare a questo punto la stessa analisi di prima:

$$P_{MAX} \cdot T(P_{MAX}) \approx N^3 \cdot \log_2(N)$$

Per ridurre lo spreco si può utilizzare un numero pari a $P = N^3/\log_2(N)$ di processori, analiticamente si può suddividere la matrice in blocchi di dimensione $P$, ripetendo l'analisi svolta per l'esempio precedente. Questo algoritmo esibisce dunque un altissimo grado di parallelismo.

## 1.2 Computation DAG

Vogliamo ora generalizzare i concetti visti precedentemente.

**Definizione 1.1.** Computation DAG (CDAG)

$$C = (I, V, 0, E)$$

$I$: insieme dei nodi di ingresso
$V$: insieme dei nodi di operazione
$O$: insieme dei nodi di uscita
$O \subseteq I + V$
$I \cap V = \emptyset$
$E \subseteq (I + V) \times V$ : insieme delle dipendenze funzionali
$P_1$: ogni vertice in $I + V$ è su almeno un cammino $a \longrightarrow b$ con $a \in I$ e $b \in O$

Spesso si aggiungono delle proprietà che non appartengono strettamente alla definizione ma sono ragionevolmente utili.

Una volta definito questo grafo, bisogna determinare quanto veloce esso può essere eseguito con il numero di risorse fornite. Ci si chiede inoltre quale sia il minimo numero di processori per poter risolvere il grafo in un determinato numero di passi, potenzialmente il numero minimo. E‘ importante inoltre considerare quale sia un numero ragionevole di processori $P^\star$ per avere un valore di spreco entro il coeffic1ente 2 ed avvicinarsi il più possibile al numero minimo di passi di risoluzione.

Si inizia cercando di svolgere il calcolo più velocemente possibile: si considera il caso in cui tutti gli input siano disponibili, ci si pone allora il problema di quali operazioni sia possibile svolgere su di essi. Tali operazioni sono rappresentate da $V_1$. $V_2$ è invece l'insieme delle operazioni che possono essere eseguite se gli operandi sono in $V_0$ e in $V_1$ ma almeno un operando dev'essere in $V_1$, altrimenti $V_2$ sarebbe stata eseguita prima di $V_1$. Questa proprietà vale per ogni $V_i$ e $V_{i+1}$. Arriverà ad un certo punto a $V_L$ dove il processo termina in quanto non vi sono più operazioni i cui operandi sono in stadi già sistemati.

$$P_1 \to \sum_{j=0}^{L} V_j = I + V$$

Siccome il grafo non ha cicli, i cammini sono tutti di lunghezza finita e dunque esiste una lunghezza massima denotata con $L$. Oggi si è definita una procedura per scomporre il grafo in pezzi, senza considerare gli input tali pezzi sono $L$.

**Proposizione 1.** Se $(V_0, V_1, ..., V_L)$ è la *Greedy Schedule* di $C$ allora $L$ è la massima lunghezza di un cammino di $C$.

**Proof:** Sia una quantità $L$ definita attraverso la Greedy Schedule e si definisce un'altra quantità $M$ come la lunghezza massima di un cammino di $C$, si verifica che tali grandezze coincidono: $*M \geq L$. Se io costruisco un cammino di lunghezza $L$, dato che $M$ è il cammino di lunghezza massima allora questa relazione è necessariamente vera. Dalla Greedy Schedule si ricava che se un nodo appartiene allo stadio $i$, esso ha almeno un predecessore nello stadio $i-1$. Di conseguenza esiste dallo stadio $L$ un cammino di lunghezza $L$ e termina allo stadio $V_0$ (per costruzione).

Per dimostrare formalmente la relazione: si costruisce un cammino $\gamma$ partendo da un vertice $v_L \in L$ e scegliendo (per $i = L-1, ..., 0$), $v_1 \in V_i$ tale che $(v_1, v_{i+1}) \in E$. Ovviamente per la Greedy Schedule tale cammino esiste, e $M \geq \|\gamma\| = L$
$*M \leq L$
Osservazione: se $[(a,b) \in E, a \in V_i, b \in V_j]$ allora $j > i$, cioè due nodi diversi dello stesso cammino devono essere collocati su stadi successivi della Greedy Schedule. Dunque se ho un cammino di lunghezza $M$ allora devono esistere almeno $M$ stadi differenti successivi, dunque $L = M$. Formalmente se $\gamma = (u_0, u_1, ..., u_k)$ è un cammino di $C$, allora $L \geq K$, scegliendo $k = M$ ottengo la relazione cercata.

Dunque fra tutti i cammini che arrivano in $V_i$, il più lungo ha lunghezza $i$. Si sta comunque assumendo l'idealità della macchina, cioè una macchina tale per cui nel momento in cui i valori sono pronti per l'esecuzione dell'operazione successiva, essa viene immediatamente eseguita. Per ogni stadio sono necessari un numero di processori pari alla cardinalità dello stadio stesso, secondo la Greedy Schedule è dunque necessario avere un numero di processori $P_{MAX} = \max V_j$, *per* $j \leftarrow 1$ *a* $L$. Potrei tuttavia riuscire a percorrere il cammino in tempo $L$ con un numero di processori $P < P_{MAX}$. Sicuramente non è possibile migliorare il tempo $L$ essendo esso cammino critico per $P_{MAX}$. Il problema di determinare il numero minimo di processori per svolgere il cammino critico in tempo $L$ è NP-Hard. Ci sono problemi dove trovare una soluzione quando esiste un solo tipo di operazione è un problema polinomiale, mentre se vi sono più operazioni tale problema è NP-Hard. A noi interessa calcolare queste quantità $L$ o $P_{MAX}$ o ci interessa solo una definizione? Dipende dal contesto, in generale la lunghezza del cammino critico è funzione dell'input e dunque l'analisi dev'essere effettuata in maniera matematica piuttosto che algoritmica. Così come è interessante analizzare il tempo e lo spazio di esecuzione di un algoritmo sequenziale, ci occupiamo di analizzare algoritmi paralleli. Tuttavia questo problema dev'essere trattato anche da algoritmi di compilazione poiché essi devono esporre un certo livello di parallelismo interno al codice per renderlo più efficiente. Il passo successivo che si vuole fare è: quanto efficientemente andremmo ad usare $P_{MAX}$ processori? Analizzando i Vi della Greedy Schedule si vede che se ad ogni stadio è presente

un numero differente di elementi, allora vi è sempre uno spreco. Come precedentemente affermato, una buona misura del grado di parallelismo si ottiene con:

$$P^\star = \frac{|V|}{L}$$

Si dimostrerà che tale numero $P^\star$ rallenterà la Greedy Schedule di un fattore al più 2, ottenendo dunque uno spreco al più pari a 2.

Condizione necessaria e sufficiente affinchè un grafo $C$ sia una schedule per un solo processore è che devono essere rispettate le dipendenze topologiche del grafo, ossia semplicemente se un nodo $b$ ha un arco entrante da $a$, allora l'attività $b$ dev'essere eseguita dopo $a$. La Greedy Schedule non individua un unico ordinamento topologico, però indica le dipendenze tra i nodi. Il numero dei possibili ordinamenti topologici che si possono ricavare cresce esponenzialmente con il numero di nodi. Per esempio nel caso di due catene indipendenti gli ordinamenti possibili per generare un'unica catena ammissibile che rispetta gli ordinamenti è $\binom{2n}{n}$.

Questo valore si ricava dal fatto che per identificare la catena finale è sufficiente indicare quali posizioni sono occupate da nodi $a$ e quali da nodi $b$, poichè l'ordinamento tra i nodi $a$ e $b$ è univoco.

A causa del numero esponenziale di differenti ordinamenti che si possono generare il problema di trovare la schedule ottima è computazionalmente difficile. L'ordinamento topologico in sè può essere ricavato in tempo lineare.

**Definizione 1.2.** $T(P)$ è il tempo minimo per eseguire il cdag $C$ con $P$ processori.

Nei casi $P = 1$ e $P = P_{MAX}$ è facile capire quanto vale mentre per gli altri valori si possono ottenre delle stime superiori o inferiori.

- $T(P) \geq L$
- $T(P) \geq \frac{|V|}{P}$
- $T(P) \geq \max L, \frac{|V|}{P} \geq \frac{1}{2}(\frac{|V|}{P} + L)$
- $T(P) \leq T_{GREEDY|P} = \sum_{j=1}^{L} \lceil \frac{|V_j|}{P} \rceil \leq \sum_{j=1}^{L} (\frac{|V_j|}{P} + 1) = \frac{|V|}{P} + L$

Non si può essere certi dell'ottimalità della Greedy Schedule, tuttavia essa costituisce un upper bound essendo una schedule valida. Unendo i lower e gli upper bound precedentemente ricavati si ottiene la formula:

$$\frac{1}{2}(\frac{|V|}{P} + L) \leq T(P) < \frac{|V|}{P} + L$$

E' conveniente definire $P^\star = \frac{|V|}{L}$ (in generale minore di $P_{MAX}$ anche se ci sono casi in cui $P^\star = P_{MAX}$).

- $T_{GREEDY|P}(P^\star) < \frac{|V|}{P^\star} + L = 2L$

- $T_{GREEDY|P}(P^\star)P^\star < 2L\frac{|V|}{P^\star} = 2|V|$ (numero massimo di operazioni farre con $P^\star$ processori in un tempo $T_{GREEDY|P}(P^\star)$)

Quest'analisi permette di ridurre gli sprechi massimizzando il guadagno con un numero ragionevole di processori sebbene non permetta di capire in tempo reale (mentre i processori selezionati lavorano) come ridurre gli sprechi.

# 2 Algoritmi paralleli

Nelle ultime lezioni abbiamo definito il modello di calcolo *computation dag* definendo il parallelismo che è possibile individuare in un calcolo. Dobbiamo ora definire la relazione tra un algoritmo e il suo cdag. Un algoritmo è un qualcosa che si può vedere come una scatola nera che in funzione di input restituisce un output. Bisogna tenere presente che un cdag associato ad un algoritmo è dipendente dell'input. Questa non è un'eccezione nell'analisi degli algoritmi poichè anche la più tipica analisi worst-case è basata sull'input.

Gli algoritmi presentati in precedenza hanno una particolarità, una volta fissata la taglia essi generano lo stesso grafo indipendentemente dell'input. Ci sono dunque dei casi significativi di problemi interessanti che hanno questa proprietà, cioè che il grafo di calcolo dipende solo della taglia $n$ ma non dell'input specifico. Per questi problemi il cdag è particolarmente utile come strumento di analisi. Se ho un algoritmo scritto in maniera sequenziale, l'applicazione del modello del cdag su tale algoritmo mi permette di capirne il livello di parallelismo. Ad oggi il campo degli algoritmi paralleli e molto vasto e tutti i problemi studiati sequenzialmente sono stati ristudiati in modo parallelo.

## 2.1 Problema dell'ordinamento

Andiamo a vedere alcuni algoritmi che conosciamo per analizzarli dal punto di vista del parallelismo. Sotto opportune ipotesi l'ordinamento richiede $n\log(n)$ confronti e ci sono algoritmi come MERGE-SORT che si avvicinano molto a tale lower bound.

Dal punto di vista sequenziale, il merge si può eseguire in tempo lineare e da luogo a $n\log(n)$ confronti, ma poichè le due chiamate di SORT lavorano a dati completamente scorrelati e indipendenti, se avessi due processori potrei eseguirle contemporeaneamente. Inoltre sviluppando le chiamate ricorsive il parallelismo aumenta ulteriorimente. Prima di definire il parallelismo complessivo bisogna osservare il comportamento di merge che comporta problemi in quanto non è molto parallelo. Il MERGE-SORT classico sequenziale ha le seguenti proprietà:

- **Analisi temporale:** $T(N) = 2T(\frac{N}{2}) + T_M$.
- **Numero di operazioni:** $V_S(N) \leq 2V_S\frac{N}{2}) + V_M(N)$. Rimane da analizzare il merge e sostituire le funzioni corrette per esso, inoltre si utilizza il segno minore o uguale poichè si prende in considerazione il caso peggiore.
- **Cammino critico:** $L_S(N) \leq L_S(\frac{N}{2}) + L_M$. Tale ricorrenza corrisponde al percorrimento di uno dei rami dell'albero delle ricorsioni per determinarne la lunghezza. Non posso utilzzare l'uguaglianza ma sono costretto usare una maggiorazione per tenere in considerazione tutti i casi in cui il worst-case non avvenga in tutti i blocchi.

*Osservazione:* Supponiamo che un grafo sia suddiviso in più parti, come nel caso di MERGE-SORT dove possiamo distinguere due diverse funzioni, SORT e MERGE, è possibile che i cammini più lunghi della prima metà del grafo si raccordino con cammini più corti della seconda metà del grafo? Quindi nessuno dei cammini è uguale come lunghezza alla somma dei cammini critici. Devo dunque cercare un cammino che sia il più lungo possibile in tutti i blocchi ma è possibile che tale cammino non si realizzi mai nell'algoritmo.

*Esempio:* Quando tutti i cammini indipendenternente dagli input hanno la stessa lunghezza allora posso ottenere un risultato senza ricorrere alla maggiorazione.

Tornando al MERGE-SORT, se usiamo il MERGE "tradizionale", $V_M(N) \leq N$, $V_S(1) = 0 \Rightarrow V(N) \leq N \log_2(N)$. Cosa si può dire invece dell'$L_N$? Ogni operazione dipende da un'operazione precedente e dunque per nel cdag dovremmo aggiungere dipendenze aggiuntive rispetto a quelle funzionali, dipendenze di controllo e di indirizzo. Allora $L_M(N) \leq N, L_S(1) = 0 \Rightarrow L_S(N) = \Theta(N)$. Dunque il MERGE-SORT con MERGE classico ha un parallelismo:

$$P_S^\star = \frac{V_S(N)}{L_S(N)} \leq \frac{O(N \log(N))}{\Omega(N)} \leq O(\log(N))$$

## 2.2 QUICK-SORT

C'è una differenza sostanziale tra i due schemi MERGE-SORT e QUICK-SORT, le taglie di $x_0$ e $x_1$ dopo la fase di pivoting sono molto probabilmente differenti, al punto che nel caso peggiore riesco a ridurre la taglia del problema ad ogni iterazione di 1, portando il tempo nel worst-case a $O(N^2)$, con profondità dell'albero della ricorsione pari a $N$. Il QUICK-SORT è uno dei casi in cui l'analisi al caso peggiore non è molto affidabile, poichè il worst-case è molto peggiore del caso medio ma è estremamente raro. Tale algoritmo infatti risulta tipicamente più veloce del MERGE-SORT nel caso medio, anche se analiticamente questa differenza non si vede. Con una probabilità $\geq 50\%$ in due passi ho almeno dimezzato la taglia del problema, e dunque anche se la profondità non è costante, la profondità massima sarà $k \log(N)$ per una qualche costante $k$. Per aumentare la probabilità di bilanciamento della suddivisione si possono prendere 3 elementi e scegliere quello intermedio come pivot, ottenendo così una riduzione della profondità dell'albero facendo qualche confronto in più.

$$\tilde{V}_Q(N) = \Theta(N \log_2(N))$$
$$L(N) \leq O(\log(N)) + \alpha L(\beta N) = O(\log^2(N)), \quad \beta > \frac{1}{2}$$
$$\tilde{P}_S^\star = \Theta(\frac{N}{\log_2(N)})$$

Quegli elementi che rendevano il QUICK-SORT sequenziale più veloce del MERCE-SORT tendono ad attenuarsi nel calcolo parallelo. Tale algoritmo è comunque spesso utilizzato in modo parallelo ma esistono altri algoritmi di sorting che sfruttano altre proprietà per ottenere un parallelismo migliore.

## 2.3 BITONIC SORTING

Questo algoritmo è una forma di MERGE-SORT proposto da Kenneth E. Batcher nel 1964 e pubblicato nel 1968. In quegli anni si progettavano algoritmi booleani per realizzare algoritmi cablati e si stimava che circa un quarto del tempo macchina veniva utilizzato per operazioni di ordinamento. Si cercava dunque di realizzare un circuito cablato per eseguire ordinamenti con l'idea che prima o poi tale circuito sarebbe diventato funzionale sui calcolatori. In realtà questa idea non ha mai visto la luce, i processori odierni non possiedono unità funzionali per svolgere l'ordinamento perché probabilmente non ha la stessa importanza che aveva ai tempi e inoltre perché il circuito ha una taglia fissa e ridotta. Non è facile, tenendo presente questo, realizzare algoritmi veloci che utilizzano come operazione primitiva il confronto tra più elementi (ipotizzando la presenza di un circuito in grado di fare questo) invece che il singolo confronto.

*Osservazione:* I processori odierni:
$\tau = 10\ ps = 10^{-11}\ s$ (tempo di commutazione di un transistor odierno)
$c = 3 \cdot 10^8\ ms^{-1}$
$c\tau = 3 \cdot 10^8 \cdot 10^{-11} = 3\ mm$
Più o meno un chip ha un lato di 3 $cm$, dunque i transistor commutano fino a 10 porte logiche nel tempo che il segnale va da una parte all'altra del chip. Perché non si fanno i processori più grandi? Per una questione di difetti, essi infatti contengono con probabilità significativa dei difetti, probabilità dovuta alle tecniche di produzione distribuita su ogni unità di area. Dunque più grande è il chip, più è probabile che sia difettoso, e la probabilità che non vi siano difetti decresce esponenzialmente in funzione dell'area. Oggi come oggi, quando viene progettato e realizzato un nuovo chip, solamente un 5% riesce senza difetti e non viene buttato. A partire da questo risultato i processi produttivi vengono raffinati e ottimizzati per tale specifico chip per ottenere una percentuale di yield migliore. Dove manca il parallelismo di MERGE-SORT è nell'operazione di MERGE, bisogna dunque trovare un modo più efficiente di svolgere tale opreazrone. Supponiamo di avere due sequenze $A$ e $B$, non necessariamente della stessa lunghezza, e di voler fare il MERGE. Prendo $B$, la rovescio e la concateno con $A$, in questo modo utilizzo due processori, uno che parte dai massimi (più o meno dal centro della stringa), e uno dai minimi e li ordina.

Questa idea può essere riproposta per un numero di processori $> 2$? Intuitivarnente posso suddividere ogni stringa a metà e su ognuna di questa metà applico due processori come visto precedentesmente ma questa operazione non è cenetta. Devo trovare una suddivisione delle due sequenze in problemi tali per cui valgono le proprietà:

- $\max(A_1, B_1) < \min(A_2, B_2)$
- $|A_1| + |B_1| \cong |A_2| + |B_2|$

IDEA: Si cerca l'elemento mediano della sequenza $A$, tale operazione si può svolgere in tempo costante e si cerca tale elemento, o un elemento con valore immediatamente successivo o precedente nella sequenza $B$. Con questo metodo la stringa $B$ è molto sbilanciata e l'obbiettivo dell'equipartizione non è risolto. Si cerca dunque di migliorare il bilanciamento delle sequenze rendendo più efficiente questa idea invece di riformulare un'altra idea da capo.

Prendiamo allora sia la mediana di $A$ che la mediana di $B$. Naturalmente bisogna osservare che la prima proprietà non è soddisfatta. Se $m_A > m_B$ lo split dev'essere modificato spostando l'indice in $A$ verso sinistra e l'indice in $B$ verso destra, se $m_A < m_B$ l'operazione è analoga ma contraria. Di quanto vanno spostati questi indici? Sfrutto la ricerca binaria per trovare gli indici adatti. Se $A$ e $B$ hanno la stessa lunghezza il risultato sarà bilanciato. Tuttavia non essendo $A$ e $B$ della stessa lunghezza, la ricerca binaria procede dividendo a metà solamente la stringa più corta e spostando l'indice sulla più lunga della stessa quantità.

Tuttavia questa ricerca binaria ha un problema, è un algoritmo sequenziale e inoltre il risultato de confronto ci dice dove andare a prendere i dati, considerazione dipendente dell'input. Questo causa problemi visto che Batcher aveva in mente di realizzare un circuito. Infatti gli algoritmi che più si apprestano ad essere realizzati con circuiti cablati sono quelli che presentano lo stesso cdag per ogni input. L'idea di Batcher per eliminare quest'incertezza sull'esito dei confronti e la seguente: invece di svolgere una ricerca binaria intelligente si svolgono una serie di confronti tutti contemporaneamente. Nel momento in cui il valore trovato in $A$ è maggiore dell'elemento in $B$ ho individuato, effettuo lo scambio tra $A$ e $B$ di tutti quegli elementi di $A$ che sono maggiori degli elementi di $B$.

In questo modo ottengo automaticamente, senza ulteriore elaborazione, due sequenze $A$ e $B$ in cui $A_i > B_j$ per ogni $i, j$. Per effettuare questi confronti posso sfruttare l'idea iniziale di Batcher, se io concateno $A$ con $B$ rovesciato ottengo una sequenza molto semplice e di confronti come illustrato in figura.

A questo punto posso risolvere i sottoproblemi ricorsivamente.

Se si va a provare si nota un problema da superare. Chiamiamo *saliscendi* una sequenza che comincia con una porzione crescente e diventa poi decrescente. Quando si procede ricorsivamente su una sequenza di questo tipo non ho la garanzia che i due sottoproblemi che ricavo siano anch'esse sequenze saliscendi. Ma il buon Batcher scoprì che quando l'input è un saliscendi, la sequenza viene spezzata in una rotazione di saliscendi.

**Definizione 2.1.** L'operazione *Left Cyclìc Shift* si definisce come:

$$SHIFT_j(x_0, x_1, ..., x_{N-1}) \hat{=} (x_j, ..., x_{N-1}, x_0, ..., x_{j-1})$$

**Definizione 2.2.** Una sequenza $(x_0, ..., x_{N-1}$ si dice *bitonica* se esite una sua rotazione che è in forma saliscendi, $j \in \{0, ..., N-1\}$ tale che ponendo $(y_0, y_1, ..., y_{N-1}) = SHIFT_j(x_0, ..., x_{N-1})$ si ha che esiste un $i \in \{0, ...N-1\}$ tale che $y_0 \leq y_1 \leq ... \leq y_i \geq y_{i+1} \geq y_{i+2} \geq y_{N-1}$.

**Definizione 2.3.** Sia $N$ pari, sia $\underline{x} = (x_0, ..., x_{N-1})$, COMPARISON-EXCHANGE è l'operazione che confronta e scambia direttamente due elementi della sequenza. Si tratta di un'operazione primitiva in molti algoritimi di sorting.

Si nota che tutti gli elementi di $L_{\underline{X}} \leq U_{\underline{X}}$. Se effettuo lo shift su una sequenza bitonica allora gli elementi che saranno confrontati dal comparatore saranno gli stessi, sebbene in posizione diverse. Cioè elementi che nella figura precedente erano confrontati dal comparatore saranno ancora accoppiati se la sequenza viene shiftata. L'idea è dunque di

applicare ricorsivamente le comparazioni sulle due nuove sequenze $L_{\underline{X}}$ e $U_{\underline{X}}$ ottenute dal precedente comparatore. Tale rete è il BITONIC-MERGE con input di 8 elementi.

**Proposizione 2.** Se $\underline{x}$ è bitonica, allora:

- $\max L_{\underline{X}} \leq \min U_{\underline{X}}$
- $\max L_{\underline{X}}$ e $\min U_{\underline{X}}$ sono sequenze bitoniche

*Osservazione.* Il principio di induzione matematica dice che se $P(O)$ vale e inoltre $\forall n, P(n) \Rightarrow P(n+1)$ allora $\forall n, P(n)$ vale. Tale principio è un assioma. Se $Q(n) \Rightarrow P(n)$, ovvero $Q(n)$ è affermazione più generale di $P(n)$, allora dimostrare che $Q$ è vera per ogni $n$ mi dimostra che, transitivamente, anche $P$ è vera per ogni $n$ (non vale il viceversa). In certi casi può valere la pena puntare a dimostrare l'ipotesi $Q(n)$ piuttosto che $P(n)$, rafforzando così l'ipotesi induttiva. Nel caso in esame:

- $P(n)$ = "tutte le sequenze saliscendi sono ordinabili";
- $Q(n)$ = "tutte le sequenze bitoniche sono ordinabili".

Numero di confronti che l'algoritmo svolge (caso $N$ potenza di 2) è: $C_{BM}(N) = \frac{N}{2} \log_2 N$. Il cammino critico dell'algoritmo è: $L_{BM}(N) = \log_2(N)$. Il parallelismo è dunque $P^\star = \frac{N}{2}$, un valore ottimale. Tuttavia il problema è che $C_{BM}$ è elevato per essere solamente relativo alla componente MERGE dell'algoritmo, c'è dunque un costo da pagare per ottenere questo livello di parallelismo.

**Proposizione 3** (Floyd)**.** In un cdag per il merging il numero di confronti è almeno $\frac{1}{4} N \log_2 N$.

D'algoritmo ha una struttura statica, questo significa che il grafo che rappresenta i confronti (attraverso il quale si può realizzare l'algoritmo cablato) non dipende dall'input. Se tale grafo invece varia in funzione dell'input allora si dice che l'algoritmo ha una struttura dinamica, come ad esempio per l'algoritmo di ricerca binaria, dove primo confronto è indipendente dall'input, tuttavia il secondo e i successivi dipendono dal valore dell'input.

**Proposizione 4** (Bilardi-Nicolau '86)**.** ADAPTIVE-BITONIC-MERGE ha un numero di confronti $C_{ABM} = O(N)$ ed un cammino critico $L_{ABM} = O(\log_2 N)$.

Per quanto riguarda il SORT, invece, se vado a contare il numero dei confronti:

$$C_{BS}(N) = \frac{N}{2} \frac{\log_2 N(\log_2 N + 1)}{2}$$
$$L_{BS}(N) = L_{BM}(2) + ... + L_{BM}(N) = \frac{\log_2 N(1 + \log_2 N)}{2}$$

**Proposizione 5** (AKS '83)**.** Si è trovata una rete AKS che è in grado di svolgere il SORTING con:

$$C_{AKS}(N) = O(N \log_2 N)$$
$$L_{AKS}(N) = O(\log_2 N)$$

Tuttavia la notazione $O$ grande nasconde una costante di fronte a $N \log N$ che è stato stimato essere oltre il migliaio, dunque per tutti gli scopi pratici il BITONIC-SORTING è migliore.

Mettendo assieme le due componenti dell'algoritmo che abbiamo visto possiamo osservare com'è strutturata la rete per fare il BITONIC—SORT:

**Definizione 2.4.** Sia $B(N)$ il numero di permutazioni bitoniche di $\{0, 1, ..., N-1\}$.

$$B(1) = 1$$
$$B(2) = 2 = 2!$$
$$B(3) = 6 = 3!$$
$$B(4) \leq 4!$$

Vediamo allora se riusciamo a catalogare tutti i saliscendi e poi vediamo le loro rotazioni. Quest'idea è buona ma bisogna considerare che alcuni elementi vengono conteggiati più volte. Si cerca allora di bloccare il maggior numero di elementi possibili: se la sequenza è $\{0, ..., N-1\}$, un suo saliscendi comincerà con l'elemento 0 e la fase di salita terminerà sul numero $N-1$.

Se io stabilisco quali elementi sono presenti nella porzione di sequenza tra l'elemento 0 e l'elemento $N-1$ (nell'insieme $S$) allora ho identificato univocamente tale stringa, poichè so che tali elementi si configureranno necessariamente in ordine crescente e i rimanenti elementi si ordineranno in maniera decrescente successivamente all'elemento $N-1$.
Allora:

$$B(N) = N \cdot 2^{N-2}$$

Potrei avere che due rotazioni in $S_1$ e in $S_2$ mi danno la stessa sequenza, dunque considerando che io posso sommare le cardinalità di due insiemi solo se la loro intersezione è vuota, allora devo dimostrare questo fatto. L'unico modo di ottenere due rotazioni uguali è di ruotare le sequenze lo stesso numero di volte, altrimenti il risultato avrebbe l'elemento 0 in posizioni diverse. Tuttavia se le due sequenze sono state ruotate lo stesso numero di volte, allora è banale che le due sequenze devono essere identiche per poter avere la rotazione di indice $i$ identica. E' dunque allora lecito sommare le cardinalità.

*Osservazione.* Le sequenze bitoniche sono in numero inferiore alle permutazioni totali, che sono invece $N!$. La differenza tra la rappresentazione binaria di queste due grandezze è di questo tipo:

$$\log_2 N! = N \log_2 N - 0(N)$$
$$\log_2 B(N) = \log_2 N + (N-2)$$

In realtà il BITONIC-MERGE è in grado di ordinare molte più sequenze di quelle bitoniche.

**Proposizione 6.** Il BITONIC-MERGE($N$) ordina $2^{C_{BM}(N)}$ permutazioni distinte, dove $C_{BM}(N) = \frac{1}{2} N \log_2 N$.

**Proof:** La dimostrazione viene lasciata per esercizio al lettore.

Supponiamo di prendere la rete del BITONIC-MERGE, ci si chiede: tale rete ordina o no tutte le permutazioni possibili?

**Proposizione 7** (0-1 Principle)**.** Se una rete di comparator-exchanger ordina tutte le sequenze in $\{0,1\}^N$, allora ordina tutte le sequenze.

## 2.4 Problema del routing

Il *problema dell'instradamento* o *routing* è un problema che prima ancora di presentarsi nel calcolo si è presentato nell'ambito della telefonia. Esso può essere suddiviso in vari sottoproblemi, tra cui una una distinzione molto nota è:

- **routing off-line**, il sistema che deve instradare i pacchetti conosce in anticipo sorgenti e destinazioni dei pacchetti e può dunque svolgere un lavoro di pianificazione;
- **routing on-line**, tutto è determinato in tempo reale, destinazioni, sorgenti e contenuto dei messaggi.

L'algoritmo che studieremo principalmente è nato nel campo della telefonia dalle ricerche condotte negli anni 60 da Benes, che lavorava nei Bell Labs. Supponiamo di avere una situazione in cui ho $N$ terminali di ingresso e di uscita. Il problema può essere così schematizzato:

Supponiamo il problema con le seguenti restrizioni (permutation routing):

- Term($i$) ha un pacchetto per Term($\pi(i)$);
- $\pi$ è una permutazione.

Ogni terminale in ingresso ha uno e un solo pacchetto da inviare ad un terminale di uscita (questo è un caso specifico del problema generale studiato da Benes). Se la centralina me la devo costruire io, come la costruisco? Benes prese come mattone elementare per costruire questa rete di routing l'interruttore o switch binario, un dispositivo a due ingressi, due uscite e uno stato binario $S$:

- Se $S = 0$ allora l'input in $a$ esce in $c$ e l'input in $b$ esce in $d$.
- Se $S = 1$ allora l'input in $a$ esce in $d$ e l'input in $b$ esce in $c$.

Gli output sono dunque rappresentati dalle equazioni:

$$\begin{cases} c = Sb + (1-S)a \\ d = Sa + (1-S)b \end{cases}$$

L'idea di Benes è dunque di costruire la rete di permutazione con tali switch.

*Osservazione.* Si noti la somiglianza tra i comparator-exchanger (vedi capitolo bitonic sort) e gli switch. I primi possono essere visti come degli switch in grado di autoconfigurarsi dinamicamente in base all'input. Un modo per fare una rete di permutazioni è quello di mettere gli interruttori disposti secondo una rete che svolge l'ordinamento.